Timely Fine-grained Interference-sensitive Run-time Adaptation of Time-triggered Schedules

Stefanos Skalistis, Angeliki Kritikakou

Timely Fine-grained Interference-sensitive Run-time Adaptation of Time-triggered Schedules

Stefanos Skalistis
Univ Rennes, Inria, CNRS, IRISA
F-35708 Rennes, France
stefanos.skalistis@inria.fr

Angeliki Kritikakou
Univ Rennes, Inria, CNRS, IRISA
F-35708 Rennes, France
angeliki.kritikakou@irisa.fr

***Abstract*—In time-critical systems, run-time adaptation is required to improve the performance of time-triggered execution, derived based on Worst-Case Execution Time (WCET) of tasks. By improving performance, the systems can provide higher Quality-of-Service, in safety-critical systems, or execute other best-effort applications, in mixed-critical systems. To achieve this goal, we propose a parallel interference-sensitive run-time adaptation mechanism that enables a fine-grained synchronisation among cores. Since the run-time adaptation of offline solutions can potentially violate the timing guarantees, we present the Response-Time Analysis (RTA) of the proposed mechanism showing that the system execution is free of timing-anomalies. The RTA takes into account the timing behavior of the proposed mechanism and its associated WCET. To support our contribution, we evaluate the behavior and the scalability of the proposed approach for different application types and execution configurations on the 8-core Texas Instruments TMS320C6678 platform. The obtained results show significant performance improvement compared to state-of-the-art centralized approaches.**

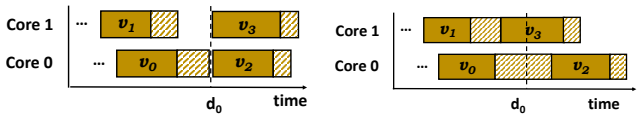
***Index Terms*—Worst-Case Execution Time, Interference-sensitive, Run-time Adaptation, Time-triggered, Response Time Analysis, Multi-cores**

I. Introduction

A. Context

The constant growth of processing demand, combined with the heat dissipation issues of single-core platforms, has led to platforms consisting of several cores that concurrently execute a high number of applications. However, the use of such multi-/many-core platforms in time-critical systems poses several challenges, such as timing interference, timing composability and the "one-out-of-m-cores" problem [1], [2].

Time-critical systems, such as safety-critical and mixed-criticality systems, have hard real-time applications for which timing guarantees must be provided, i.e. their worst-case response time must be less than their respective deadlines. In multi-core platforms, several resources are shared among the cores. The concurrent accesses to these resources must be arbitrated, introducing non-deterministic variations to the access times. This behavior is called *timing interference*. The amount of interferences a task may suffer during execution depends on which tasks are running in parallel and their number of accesses to shared resources. The tasks running in parallel are defined by the task scheduling and allocation. To claim timing guarantees, the Worst-Case Execution Time (WCET) of tasks must be used during system design. To estimate the WCET of a task either: i) the worst-case interference is used, i.e. all accesses of a task to a shared resource are assumed to conflict with all other cores, and it is valid for any task scheduling and allocation, or ii) the task scheduling and allocation is known providing essential information that allows more accurate estimation of interferences, but this WCET estimation is valid only for the given task scheduling and allocation solution. In the first case, the pessimism introduced in the WCET estimation (by being unaware of the task scheduling and allocation solution) can potentially negate the performance benefit coming from the parallel execution of the tasks on multi-cores [1] or even make the problem infeasible, if the system becomes unschedulable. In the second case, the known task scheduling and allocation solution is used to compute *interference-sensitive* WCET (*is*WCET), which are lower than the pessimistic WCET of the first case [3], [4], [5].



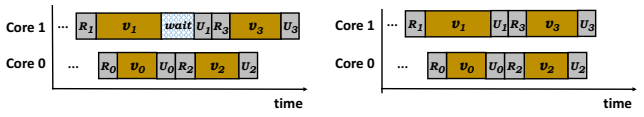
(a) Time-triggered execution (b) Unsafe re-scheduling

Fig. 1: Task execution considering *is*WCET

The majority of existing works on task scheduling and allocation using *is*WCET (*interference-sensitive* task scheduling and allocation), e.g. [6], [7], uses time-triggered execution so as to guarantee that the task scheduling and allocation solution remains unchanged during execution. In time-triggered executions, the tasks are executed exactly at their start times provided by a solution given by an offline scheduling and allocation algorithm. For instance, Figure 1a schematically illustrates the time-triggered execution of the task scheduling and allocation solution for four tasks v_0, v_1, v_2, v_3 that access a shared resource. The delays, that each task suffers due to the interference caused by the task running in parallel, are denoted with the light stripped box.

On top of timing guarantees, the actual run-time performance for the majority of time-critical systems is also of high importance. A performance improvement creates slack that allows to increase the Quality-of-Service in safety-critical

This work was partially funded by European Union's Horizon 2020 research and innovation program under grant agreement No 688131, project Argo.



(a) *is*RA-GLO (b) *is*RA-FG

Fig. 2: Run-time adaptation through: a) global synchronisation (*is*RA-GLO) and b) fine-grained synchronisation (*is*RA-FG)

systems or to execute other best-effort applications in mixed-critical systems (at specific time instances or at the end of execution). For example, in cruise control systems the created slack can be used to further improve quality of the result produced by the control law, whereas in satellite systems less essential functions, such as scientific instrument data collection can be activated [8]. Therefore, the only means to improve the system performance, while preserving the timing guarantees, is through run-time adaptation.

B. Motivation

Run-time adaptation uses the *actual execution time* (AET) of tasks to potentially allow earlier start times, thus improving performance. When run-time adaptation is applied on task scheduling and allocation solutions derived using the pessimistic WCET, no timing anomalies can occur since the pessimistic WCET remains valid under any adaptation. However, this is not the case when run-time adaptation is applied on *interference-sensitive* task scheduling and allocation solutions. If a task is re-scheduled at run-time to an earlier start time, this action changes the offline task scheduling and allocation solution used to derive the *is*WCET. As a consequence, the interference occurring to the tasks running in parallel may increase, and thus violate the timing guarantees. For instance, in Figure 1b, task v_1 finishes earlier. Then, a run-time re-scheduling of v_3 can cause additional interferences to v_0, increasing the *is*WCET of v_0, and, thus, violating its deadline.

The work in [9] allows run-time adaptation of interference-sensitive time-triggered schedules that preserves the timing guarantees. It inserts extra scheduling dependencies to the task scheduling and allocation solution to prohibit any additional task overlapping in case of run-time re-scheduling. In this way, no increase in the interferences occurs at run-time, and the *is*WCET are preserved. The run-time adaptation is achieved through controllers that monitor the tasks on each core. Before executing a task v_i, the controller checks if the scheduling dependencies are met, and thus the task is ready to be executed. In Figure 2a, this corresponds to the *ready* phase $\mathcal{R}_i$ of the controller of task v_i. When the task finishes execution, the controller updates a *status* array and notifies the rest of the controllers that the task has finished its execution. In Figure 2a, this corresponds to the *update* phase $\mathcal{U}_i$ of the controller of task v_i.

Nevertheless, two assumptions of this technique highly undermine performance, a limitation that we address with the proposed approach. Firstly, no protection mechanism is presented for the status array that is shared among the cores. The parallel execution of updates and checks of the monitors of different cores can lead to concurrency issues, i.e. race conditions creating data inconsistencies. The only mean for the technique to be safe is to assume a global centralized synchronisation mechanism, which will be referred as *is*RA-GLO henceforth. However, such centralized mechanism executes the updates and the checks of the monitors of different cores sequentially having a negative impact on performance. Figure 2a illustrates this negative impact for v_0 and v_2, since the monitor of Core 1 has to wait (rectangle *wait*) for the monitor phases of Core 0 to finish. Secondly, in order to bound the number occurring updates and checks of the monitors, a non-polling notification mechanism – such as interrupts – is assumed, which, however, degrades performance [10]. The result of these two assumptions is that the Response-Time Analysis (RTA) of the technique is significantly simplified, as the number of monitor invocations is upper-bounded and execution of the monitors is serialized due to the global centralised synchronisation.

C. Contributions

In this work we propose an interference-sensitive adaptation approach capable of fine-grained synchronisation (*is*RA-FG), that alleviates the above limitations. In particular, this work extends the state-of-the-art as follows:

- Parallel execution of the control phases on each core, whenever possible, via fine-grained protection of the shared variables. The impact is illustrated in Figure 2b, where the unnecessary blocking (rectangle *wait*) is eliminated and the control phases are overlapping. The proposed *is*RA-FG improves over *is*RA-GLO [9] both in terms of i) the WCET of the controller and ii) run-time performance.
- RTA-based formal proof that execution of time-triggered schedules with *is*RA-FG does not introduce additional interference and is free of timing anomalies. The RTA of the *is*RA-FG approach, is far more challenging than the one required for *is*RA-GLO, since the number of the invocations of control phases cannot be trivially bounded. With the proposed RTA, we prove that the execution of both interference-sensitive and contention-free time-triggered schedules preserves offline timing guarantees, under any AET.
- Extensive evaluation of the proposed *is*RA-FG approach on a real platform (8-core Texas Instruments TMS320C6678) with respect to scalability and application parallelism considering three application types and six application execution configurations.

From the obtained results, we observed a significant performance gain compared with the global synchronisation mechanism [9] as the number of cores increases. When a single application is parallelized and executed on several cores, the observed gain is on average 1.64%, using only two cores, up to 37.32%, when eight cores are used. When several application instances are executed on several cores, the observed gain is on average 1.81%, using only two cores up to 30.46%, when eight cores are used.

TABLE I: Notation Summary

Tasks & Graph	
V, v	Task v belonging to task-set V
$\mathcal{K}$, k	Core k from set of cores $\mathcal{K}$
E_{isRA}	Scheduling dependencies
$deg^-(v)$, $deg^+(v)$	The *indegree* and *outdegree* of task v
$pred(v)$, $succ(v)$	Predecessors/succesors of task v
Time-triggered solution	
$\mu(v)$	Core allocation $\mu(v)$ of task v
$\beta(v)$, $\epsilon(v)$	Start time $\beta(v)$ and end time $\epsilon(v)$ of task v
$prev(v)$	Set of tasks notified by the predecessor of v
Response Time & WCET	
$\mathcal{R}_v$, $\mathcal{X}_v$, $\mathcal{U}_v$	The ready $\mathcal{R}_v$, execute $\mathcal{X}_v$, update $\mathcal{U}_v$ phases
$R(v)$, $R(\mathcal{R}_v)$, $R(\mathcal{X}_v)$, $R(\mathcal{U}_v)$	Absolute response time of task v and its phases
$C_{[S]}$	The WCET of code snippet S
$C_{\mathcal{R}_v}$, $C_{\mathcal{U}_v}$	Controller WCET for the corresponding phase
$C_{TA_{\mathcal{R}}}$, $C_{TA_{\mathcal{U}}}$	Timing alignment constants
$t^v_{\mathcal{R}}$, $t^v_{\mathcal{U}}$,	Time instance when the corresponding phase can execute successfully (all branches are not taken)

The rest of the paper is organized as follows: Section II presents the system model and necessary notation; Section III presents the proposed fine-grained run-time adaptation mechanism and Section IV provides its response timing analysis. Section V presents the evaluation of the proposed approach, whereas Section VI discusses the related work and Section VII concludes this study.

II. System model

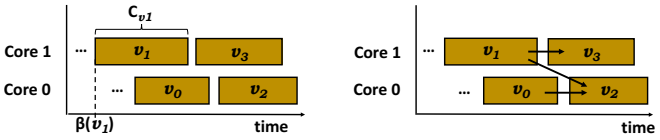
Let V denote the set of tasks of an application to be executed on the set of cores $\mathcal{K}$ of the target platform. The input to the proposed mechanism is a *time-triggered schedule* that provides the start/end times of the tasks and their allocation to cores. Formally, we model a *time-triggered schedule* for the task-set V as the tuple (μ, β, ϵ), where $\mu(v)$ denotes the core allocation, i.e. the core k at which task v is executed, and $\beta(v)$, $\epsilon(v)$ are the start and end times of the task, respectively. Such time-triggered schedule can be constructed by a scheduling algorithm that provides timing guarantees, applied offline using the *is*WCET of the task-set V. For reasons of clarity, we will assume that tasks *is*WCET does not consider restrictions on the length of task overlapping or timing of the interference (see Section VI for further discussion).

The tasks of V can be dependent, or independent, and are periodically executed in a non-preemptive manner. Since the proposed approach acts upon the time-triggered schedule, any limitation stems from the task model and scheduling algorithm used offline to derive the time-triggered schedule. A time-triggered schedule is considered *safe*, iff it satisfies the system-defined timing constraints, i.e. each task deadline and/or a global deadline must be met.

Given a safe time-triggered schedule, a set of scheduling dependencies E_{isRA} exists, s.t. a task v depends on the tasks $\{v'\}$ that finished immediately before it on all cores $\mathcal{K}$, according to $\beta(v)$, i.e.:

$$(v, v') \in E_{isRA} \iff \nexists\, v'' \text{ s.t. } \mu(v) = \mu(v'') \wedge \beta(v) + C_v < \beta(v'') + C_{v''} \leq \beta(v') \quad (1)$$

where C_v is the *is*WCET of task v.



(a) Time-triggered schedule (b) Scheduling dependencies E_{isRA}

Fig. 3: Construction of E_{isRA} scheduling dependencies, according to a given time-triggered schedule

Figure 3 illustrates the construction of E_{isRA} given a time-triggered schedule. Notice that, for any task v in the dependency relation E_{isRA}, the number of incoming edges (denoted as $deg^-(v)$) and the number of outgoing edges (denoted as $deg^+(v)$) is upper bounded by the number of cores $|\mathcal{K}|$, i.e. $deg^-(v) \leq |\mathcal{K}|$ and $deg^+(v) \leq |\mathcal{K}|$.

III. Fine-grained *is*RA (*is*RA-FG)

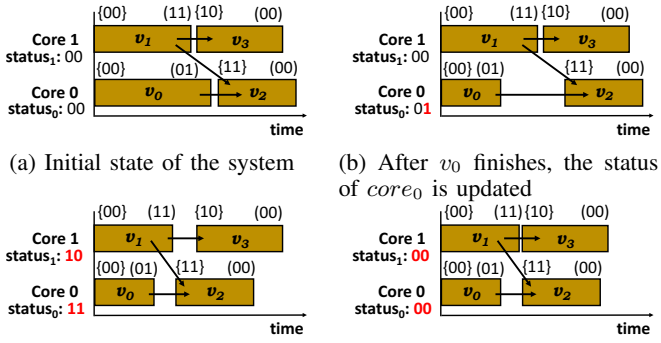
To obtain a fine-grained synchronization approach that removes the performance degradation of global synchronisation approaches, the parallel execution of control phases of different cores must be allowed, whenever it is possible, without creating any concurrency issues. To achieve that, we propose a control mechanism (Algorithm 1) that is executed independently on each core and for each task. The task *execution* is extended with two control phases, namely *ready* and *update*. During the *ready* phase, the controller waits for the task to become ready, i.e. all previous tasks have finished and, thus, its dependencies are met. Then, the task is executed. When the task finishes, the controller enters in the *update* phase, where it notifies all relevant cores that the task has finished execution. A core k' is called relevant for any task v executed on core k, when there exists an outgoing edge from task v towards a task v' on core k'. To achieve fine-grained synchronisation, each core must have its own *status* vector (of size $|\mathcal{K}|$). Each bit of the *status* vector corresponds to a core. The *status* vector of each core represents the notifications received from other cores at any point in time and it must be updated during execution by all cores.

A. Ready phase

To implement the *ready* phase, a *ready* vector (of size $|\mathcal{K}|$) is required for each task v. Each bit in the *ready* vector represents the core k on which the incoming edge of the scheduling dependencies originates from, i.e.:

$$readyVector_v[k] = 1 \quad \Leftrightarrow \quad (v', v) \in E_{isRA} \wedge \mu(v') = k \quad (2)$$

where $readyVector_v$ is the ready vector of task v. The *ready* vectors are created offline for each task v, based on the dependency relation E_{isRA}, and they are not modified during execution. For instance, in Figure 4a, the *ready* vector of task



(a) Initial state of the system

(b) After v_0 finishes, the status of $core_0$ is updated

(c) After v_1 finished, $core_1$ notifies $core_0$ which re-schedules v_2

(d) $core_1$ re-schedules v_3

Fig. 4: Example of control phases for four tasks on two cores. For each task the *ready* vector is in curly brackets. The *notification* vector is in parentheses and is also illustrated with arrows.

v_2 is $\{11\}$, since it is has to wait for i) task v_1 running on core 1 and ii) task v_0 running on core 0, to finish before being executed. These dependencies ensure that the number of interferences will not increase due to an earlier execution of v_2. On the other hand, the *ready* vector of task v_1 is $\{00\}$, since no dependency exists from another task.

The functionality of the *ready* phase of the controller on core k is described by the lines 2-9 of Algorithm 1. The controller reads the *ready* vector of the task v to be executed next (L. 2), which depicts the dependencies that have to be met before the execution of the task v. Then, it has to gain access to the critical section of the *status* vector through the protection mechanism related to the core k (L. 4). Once it has been granted access to its status vector, it checks if all task dependencies are already met (L. 5). If this is true, the task v can be executed. For instance, tasks v_2 and v_3 in Figure 4c are considered ready, since the corresponding bits of the status vectors of Core 0 and Core 1 are set and the status vectors are equal with the ready vectors. Before advancing to the execution phase, the controller has to reset the bits indicated by the *ready vector* of task v in its *status* (L. 6). This is illustrated in Figure 4d, where the corresponding bits in the status vectors are reset and tasks v_2 and v_3 are executed. In this way, any already-met dependencies from other cores to subsequent tasks on core k are preserved. Then, the protection mechanism is released and the phase finishes. Otherwise, the while loop is re-executed.

B. Update phase

To implement the *update* phase, a *notification* vector (of size $|\mathcal{K}|$) is required for each task v that describes which cores have to be informed that the task has finished execution. Each bit in the *notification* vector represents the core k to which the outgoing edge of scheduling dependencies ends, i.e.:

$$notifyVector_v[k] = 1 \;\Leftrightarrow\; (v, v') \in E_{isRA} \wedge \mu(v') = k \quad (3)$$

Algorithm 1: Fine-grained *is*RA-FG mechanism on core k.

Input: Task v, *status* array of all cores; ($status[i][j]$: the j-th *status* bit of the i-th core)

```
1  Function isRA-FG (v, status[ ][ ]):
                                   /* Ready Phase */
2      readyVector ← readyVector_v
3      while true do
4          enterSection(k)
5          if (status[k] & readyVector) = readyVector
            then
6              status[k] ← status[k] ⊕ readyVector
7              exitSection(k)
8              break
9          exitSection(k)
                                /* Execution Phase */
10     v.execute()
                              /* Notification Phase */
11     notifyVector ← notifyVector_v
12     while notifyVector ≠ 0 do
13         for i ← 1 to |K| do
14             if notifyVector[i] = 1 then
15                 enterSection(i)
16                 if status[i][k] = 0 then
17                     status[i][k] ← 1
18                     notifyVector[i] ← 0
19                 exitSection(i)
```

where $notifyVector_v$ is the notify vector of task v. The *notification* vectors are created offline for each task v, based on the dependency relation E_{isRA}, and they are not modified during execution. For instance, in Figure 4a, the notification vector of task v_1 is (11); when it finishes execution, it has to notify task v_2 running on core 0 (bit 0) and task v_3 running on core 1 (bit 1). Through the notification, the k-th bit in the *status* vector of core i is set by core k, when the finished task of core k has an outgoing edge to a task on core i. For example, in Figure 4b, the *status* vector of core 0 is 01, since task v_0 finished execution and notified only core 0.

The lines 11-19 of Algorithm 1 describe the functionality of the *update* phase of the controller on core k. After the task v on core k completes its execution, the controller has to update the *status* of all the relevant cores. To do so, it initially reads the *notification vector* of v (L. 11). For each core i, it checks if that core should be notified (by updating its status) (L. 14). For each such core, the controller tries to gain access to the critical section through the core's protection mechanism (L. 15). If access is granted, the controller verifies if the previously occurred update of the core k has been already consumed (L. 16) by core i. If this is true, the k-th bit in the status of core i is set, indicating that the dependency from core k has been met. The corresponding bit in the notification vector

is cleared to show that the controller has already updated the core. For instance, Figure 4b and Figure 4c illustrate this case. In Figure 4b the controller of Core 0 updates its own bit after task v_0 finishes. In Figure 4c, the controller of Core 1 updates the corresponding bit in the status of Core 0 after task v_1 finishes. The while loop is executed until all cores have been notified.

C. Concurrency

The proposed *is*RA-FG allows parallel execution of the control phases and avoids any potential concurrency issues that may lead to race conditions and inconsistent data during execution. Two concurrency issues could potentially exist during the run-time adaptation:

- During the ready phase the controller of core k tests if the dependencies of the task v have been met and clears the corresponding status bits, before starting execution; if another core updates the status of core k, while core k read its status, clears the corresponding status bits and writes back its status, then the write back would have invalid value, thus resulting in a data inconsistency in the status vector.
- During the execution of some task on core k, if k has already received a status update from core k', and core k' performs another status update to core k; this can end up in losing that status update and, eventually, resulting into a deadlock.

Through the proposed fine-grained protection of the shared variables, the aforementioned concurrency issues cannot occur.

IV. Response time analysis

The introduction of the controllers into time critical systems alters the timing behavior, and thus can potentially violate the timing guarantees, if the controller cost is not properly accounted for. To overcome this issue, either additional tasks are incorporated into the model used to derive the safe time-triggered schedule, or the WCET of the controller is incorporated into the WCET of each task (modulo some timing alignment). In addition to the WCET of the controller, attention must be paid to the controller's accesses to shared variables among cores (*status* vectors). If these variables are placed alongside with the data of the tasks, additional interferences must be accounted because of the parallel execution of the controller. To reduce this additional negative effect, the *status* vectors should be placed in: i) a separate memory accessed by a separate bus, when the platform provides such a feature, or ii) the local memory of each core and accessed through remote writes [11].

In Algorithm 1, we have divided the controller into three consecutive phases, namely *ready*, *execute* and *update*, which are denoted with $\mathcal{R}_v$, $\mathcal{X}_v$ and $\mathcal{U}_v$ for any task v. Notice that, while the execution phase $\mathcal{X}_v$ has a fixed *is*WCET (according to the task being executed), the *ready* and *update* phases have varying *is*WCET depending on several factors, as shown in the next paragraphs.

Let, $R(\mathcal{R}_v)$, $R(\mathcal{U}_v)$ be the *absolute*[1] response time of the ready/update phases and C_v be the *is*WCET of task v. Since the phases are executed sequentially, we consider the absolute response time of a task v to be when it finishes execution and the control phase has performed all the status updates, i.e. the absolute response time of a task v is the same as response time of its update phase:

$$R(v) = R(\mathcal{U}_v) \tag{4}$$

For any task v, the absolute response time of the ready phase depends on: (i) when it will gain access to its critical section, and (ii) when the task is going to be ready, i.e. all previous tasks have finished and all updates have been performed. The WCET of the update phase depends on: (i) when it will gain access to its critical section, (ii) the number of cores to notify, and (iii) when previous tasks, which depended on this core, finish their ready phase (s.t. $status[i][k] = 0$).

In order to make our response time analysis accurate, we derive parametric response times R based on the number of outgoing edges of a task v, according to the scheduling dependencies E_{isRA}. We denote with $C_{[S]}$ the WCET of the part of the controller that corresponds to the snippet S, i.e. the sequence of lines S of Algorithm 1. For example $C_{[12-14,13]}$, denotes the WCET of executing the iteration of the update phase, when the branch (in line 14) is not taken. Such WCETs can be acquired by static analysis or measurements.

A. Ready Phase

Let $t^v_{\mathcal{R}}$ be the time instance that task v running on core k becomes ready, i.e. all of its predecessors and their corresponding update phases have finished execution:

$$t^v_{\mathcal{R}} = \max_{v' \in pred(v)} R(v') \tag{5}$$

If the controller is invoked precisely at time $t^v_{\mathcal{R}}$, then the response time of the controller for the ready phase is the cost of executing one iteration of the control-loop plus the waiting time to gain access to the critical section for core k. The worst-case cost of executing one iteration is constant and bounded by $C_{[2-8]}$, whereas the waiting time for gaining access to its critical section depends on several factors; other cores could be in the update phase of their tasks, upon which task v did not depend. For example, consider the $deg^-(v)$ predecessors of task v and let all other $|\mathcal{K}| - deg^-(v)$ currently running tasks on other cores to have a dependency to the successor of task v. It is possible that the $|\mathcal{K}| - deg^-(v)$ tasks reach the update phase right at the start of the ready phase of core k. Hence, the waiting time to gain access to its critical section is $(|\mathcal{K}| - deg^-(v)) * C_{[15-19]}$. In addition, when the subsequent tasks (on the same core) of the $deg^-(v)$ tasks are significantly smaller than the response time of the ready phase $R(\mathcal{R}_v)$, it is possible that they reach the update phase and gain access to the critical section k, resulting to an additional cost of $deg^-(v) *$

[1]Since time-triggered schedules refer to absolute time, we use the term *absolute* response time to refer to the time elapsed from the start of the execution

$C_{[15-19]}$. Combining these waiting times, the response time of $R(\mathcal{R}_v)$ is:

$$R(\mathcal{R}_v) \leq t^v_{\mathcal{R}} + C_{\mathcal{R}_v} + C_{TA_{\mathcal{R}}} \tag{6}$$

$$\text{with } C_{\mathcal{R}_v} = C_{[2-8]} + |\mathcal{K}| * C_{[15-19]} \tag{7}$$

where $C_{TA_{\mathcal{R}}}$ is a timing alignment constant analysed in Section IV-D. Notice that the response time $R(\mathcal{R}_v)$ is defined recursively, as it depends on the maximum response time of preceding tasks ($t^v_{\mathcal{R}}$). This will assist us in proving that under any AET, the execution will respect the timing guarantees.

B. Execute Phase

Since there are no preemptions during the execution of a task, the response time for the execution phase of the controller is given by:

$$R(\mathcal{X}_v) \leq R(\mathcal{R}_v) + C_v \tag{8}$$

C. Update Phase

Let $t^v_{\mathcal{U}}$ be the time instance that task v, executed on core k, has already finished its execution and it can perform the status updates, i.e. $status[k'][k] = 0$ for all cores k' that v has an outgoing dependency to:

$$t^v_{\mathcal{U}} = \max\left(\max_{v' \in prev(v)}(R(\mathcal{R}_{v'})),\ R(\mathcal{X}_v)\right) \tag{9}$$

where $prev(v)$ is the set of tasks that previously received a status update from core k. More formally these are:

$$prev(v) = succ(v') \ \text{ s.t. } \ v' \in pred(v) \wedge \mu(v') = \mu(v) \tag{10}$$

Following similar reasoning to the ready phase, the response time of the controller for the update phase is the cost of executing one iteration of the control-loop plus the waiting times to gain access to the required critical sections. The cost of one loop-iteration is $C_{[11-12]} + |\mathcal{K}| * C_{[13-15,19]} + deg^+(v) * C_{[16-18]}$. The worst-case waiting time to gain access to the required critical sections is when all cores execute a ready phase or update phase (whichever is greater) that tries to gain access to the same critical sections. The number of critical sections, for task v, is equal to its outdegree $deg^+(v)$, whereas the worst-case waiting time to acquire each critical section is $(|\mathcal{K}| - 1) * \max\left(C_{[4-7]}, C_{[15-19]}\right)$.

Thus, the response time of the update phase is:

$$R(\mathcal{U}_v) \leq t^v_{\mathcal{U}} + C_{\mathcal{U}_v} + C_{TA_{\mathcal{U}}} \ \text{ with} \tag{11}$$

$$\begin{aligned} C_{\mathcal{U}_v} = {} & C_{[11-12]} \\ & + \ |\mathcal{K}| * C_{[13-15,19]} \ + \ deg^+(v) * C_{[16-18]} \\ & + \ (|\mathcal{K}| - 1) * deg^+(v) * \max\left(C_{[4-7]}, C_{[15-19]}\right) \end{aligned} \tag{12}$$

where $C_{TA_{\mathcal{U}}}$ is a timing alignment constant analysed in Section IV-D.

D. Timing alignment

In the analysis of the update/ready phases, we assumed that the controller starts precisely at the time when: i) it can perform its status updates (for the update phase), and ii) the required status updates have been performed (for the ready phase). Nevertheless, since the tasks can execute in less time than their WCET, there is a possibility that the controller is already inside the control-loop, when some task finishes execution. Therefore, the timing alignment constants $C_{TA_{\mathcal{R}}}$ and $C_{TA_{\mathcal{U}}}$ account for timing delays due to this potential misalignment for the ready and the update phase, respectively.

For the ready phase, the worst case for $C_{TA_{\mathcal{R}}}$ is when the controller has just released the critical section. This is because the controller cannot receive a new status update, while it is already inside the critical section. Therefore, the cost is $C_{[4]}$. Recall that the waiting time to access the critical section has been already accounted for in the response time of the ready phase.

For the update phase, the worst case for $C_{TA_{\mathcal{U}}}$ is equal to the execution of one iteration of the for-loop without taking the branch at line 16, i.e. $|\mathcal{K}| * C_{[13-16,19]}$. This time delay is sufficient because: i) the waiting times to gain access to the required critical sections have already been accounted for in the response time of the update phase, and ii) the branch is considered to be taken once, which has also been accounted for in the response time of the update phase. Thus, the timing alignment constants are:

$$C_{TA_{\mathcal{R}}} = C_{[4]} \tag{13}$$

$$C_{TA_{\mathcal{U}}} = |\mathcal{K}| * C_{[13-16,19]} \tag{14}$$

E. Safety

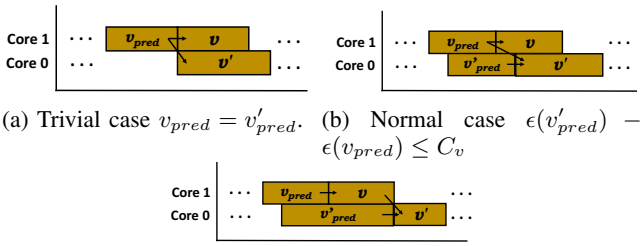
Having established the WCET and the response time of the phases of the controller, we have to prove that, if these costs are added upfront to the *is*WCET of tasks, the timing guarantees of any time-triggered schedule are not violated under any reduction of execution times, i.e. $R(v) \leq \epsilon(v)$ for all tasks v.

Let $C_{\mathcal{R}_v}, C_{\mathcal{U}_v}$ denote the WCET of the ready/update phases and $C_{TA_{\mathcal{R}}}, C_{TA_{\mathcal{U}}}$ the timing alignment constants, as they were analysed in the previous sections. Assume a safe solution (μ, β, ϵ), derived by a safe scheduling algorithm, such that it includes the controller WCETs in the *is*WCET of each task:

$$\epsilon(v) - \beta(v) \geq C_{\mathcal{R}_v} + C_{TA_{\mathcal{R}}} + C_v + C_{\mathcal{U}_v} + C_{TA_{\mathcal{U}}} \tag{15}$$

Before we prove the safety of the approach, we first need to establish a property regarding the tasks that previously received a status update from core k and the start time of the current task of core k.

Property 1. For any given task v running on core k, the tasks $\{v'\}$ that received a status update from core k (i.e. $v' \in prev(v)$) have finished their ready phase before the worst-case start time of the update phase of task v, if all preceding



(a) Trivial case $v_{pred} = v'_{pred}$. (b) Normal case $\epsilon(v'_{pred}) - \epsilon(v_{pred}) \leq C_v$

(c) Contradicting case $\epsilon(v'_{pred}) - \epsilon(v_{pred}) > C_v$

Fig. 5: Possible arrangements of task v and $v' \in prev(v)$.

tasks $v''pred(v')$ have finished at most at their worst-case end time $\epsilon(v$", i.e. $R(v'') \leq \epsilon(v'')$. That is:

$$R(\mathcal{R}_{v'}) \leq \epsilon(v) - (C_{\mathcal{U}_v} + C_{TA_{\mathcal{U}}}) \qquad \forall v' \in prev(v) \tag{16}$$

Proof. Since the WCET cost of the ready phase is constant, it is sufficient to show that this holds for task v_p with the latest start time, i.e. $v_p = \arg \max_{v' \in prev(v)} \beta(v')$

$$R(\mathcal{R}_{v_p}) \leq \epsilon(v) - (C_{\mathcal{U}_v} + C_{TA_{\mathcal{U}}}) \overset{(15)}{\Longrightarrow} \tag{17}$$

$$R(\mathcal{R}_{v_p}) \leq \beta(v) + C_{\mathcal{R}_v} + C_{TA_{\mathcal{R}}} + C_v \overset{(6),(5)}{\Longrightarrow} \tag{18}$$

$$\max_{v' \in pred(v_p)} R(v') + C_{\mathcal{R}_{v_p}} + C_{TA_{\mathcal{R}}} \leq \beta(v) + C_{\mathcal{R}_v} + C_{TA_{\mathcal{R}}} + C_v \tag{19}$$

$$\overset{C_{\mathcal{R}_{v_p}} = C_{\mathcal{R}_v}}{\Longrightarrow} \max_{v' \in pred(v_p)} R(v') \leq \beta(v) + C_v \tag{20}$$

Let v_{pred} be the predecessor of v that finishes the latest among its predecessors, thus $\beta(v) = \epsilon(v_{pred})$. Also, let v'_{pred} be the predecessor of v_p that finishes the latest among its predecessor, i.e. $\max_{v' \in pred(v_p)} R(v') \leq \epsilon(v'_{pred})$. Therefore is suffices to show:

$$\epsilon(v'_{pred}) \leq \epsilon(v_{pred}) + C_v \tag{21}$$

This trivially holds, when $v_{pred} = v'_{pred}$ (Figure 5a). Assume that it does not hold for $v_{pred} \neq v'_{pred}$, i.e. $\epsilon(v'_{pred}) - \epsilon(v_{pred}) > C_v$. In that case, according to the definition of E_{isRA} (Equation 1) there would exist a dependency between v and v' (Figure 5c), which is a contradiction, since $v' \in prev(v)$. □

Theorem IV.1. *For any safe scheduling solution, derived by adding the controller costs ($C_{\mathcal{R}_v}$, $C_{\mathcal{U}_v}$, $C_{TA_{\mathcal{R}}}$, $C_{TA_{\mathcal{U}}}$) to the is*WCET *(C_v) of the tasks V, the* is*RA-FG execution is safe under any AET.*

Proof. Since the *is*WCET is not increased due to enforcement of the E_{isRA}, it is sufficient to show that the response time of all the tasks is not greater than their end time:

$$R(v) \leq \epsilon(v) \tag{22}$$

We prove it by induction on the dependency relation E_{isRA}.
Base Case: For tasks v with no predecessors ($pred(v) = prev(v) = \emptyset$, $\beta(v) = 0$) from Eq. 4 and 11 we establish:

$$R(v) \leq t^v_{\mathcal{U}} + C_{\mathcal{U}_v} + C_{TA_{\mathcal{U}}} \overset{(9)}{\Longrightarrow} \tag{23}$$

$$R(v) \leq \max\left(\max_{v' \in prev(v)} R(\mathcal{R}_{v'}), R(\mathcal{X}_v)\right) + C_{\mathcal{U}_v} + C_{TA_{\mathcal{U}}} \tag{24}$$

Since $prev(v) = \emptyset$, the first term of the max predicate is zero; according to Eq. 8 we have:

$$R(v) \leq \max(0,\ R(\mathcal{R}_v) + C_v) + C_{\mathcal{U}_v} + C_{TA_{\mathcal{U}}} \Rightarrow \tag{25}$$

$$R(v) \leq R(\mathcal{R}_v) + C_v + C_{\mathcal{U}_v} + C_{TA_{\mathcal{U}}} \overset{(5),(6)}{\Longrightarrow} \tag{26}$$

$$R(v) \leq \max_{v' \in pred(v)} R(v') + C_{\mathcal{R}_v} + C_{TA_{\mathcal{R}}} + C_v + C_{\mathcal{U}_v} + C_{TA_{\mathcal{U}}} \tag{27}$$

With $pred(v) = \emptyset$ we conclude that:

$$R(v) \leq 0 + C_{\mathcal{R}_v} + C_{TA_{\mathcal{R}}} + C_v + C_{\mathcal{U}_v} + C_{TA_{\mathcal{U}}} \overset{(15)}{\Longrightarrow} \tag{28}$$

$$R(v) \leq \epsilon(v) - \beta(v) \overset{\beta(v)=0}{\Longrightarrow} R(v) \leq \epsilon(v) \tag{29}$$

Induction step: Suppose that (22) holds for all predecessors of task v. Starting from equation (24), it is sufficient to show that for both terms of the max predicate the inequality holds. For the second term, the proof follows the same steps as in the base case, reaching the induction hypothesis. Thus, for the first term:

$$R(v) \leq \max_{v' \in prev(v)} (R(\mathcal{R}_{v'})) + C_{\mathcal{U}_v} + C_{TA_{\mathcal{U}}} \leq \epsilon(v) \tag{30}$$

which holds due to Property 1, therefore, concluding the proof. □

Lemma IV.2. Execution of time-triggered schedules with *is*RA-FG is free from timing-anomalies and it does not increase task interference.

Proof. Trivially from Theorem IV.1. □

We have therefore established that *is*RA-FG is timely safe, and safe-guards against additional task interference. In addition to these properties, the execution is work-conserving, w.r.t. E_{isRA}, which improves run-time performance, as shown in section V.

V. Experimental Evaluation

A. Experimental Setup

Platform: We implemented the proposed approach and performed evaluation experiments on a real multi-core COTS platform, i.e. the TMS320C6678 chip (TMS in short) of Texas Instrument [12]. The characteristics of the platform are depicted in Table II. In our experimental framework, the proposed *is*RA-FG mechanism is implemented as a bare-metal library with the low-level functions for the various controller phases using the hardware semaphores of the TMS platform. The *is*RA-FG implemented through semaphores is applicable to any platform, since semaphores are a fundamental building block in systems, while in the rare case there is no hardware support, they can be implemented in software. However, the

TABLE II: TMS platform characteristics.

DSP char/stics	Instr/cycle	Freq.	L1P	L1D	L2
	8	1 GHz	32 KB	32 KB	512 KB
No. DSPs	8	**NoC**	TeraNet (Delay: 11 cycles)		
Shared L3	4 MB SRAM	**DDR3**	512 MB	**Sem.**	32 cycles

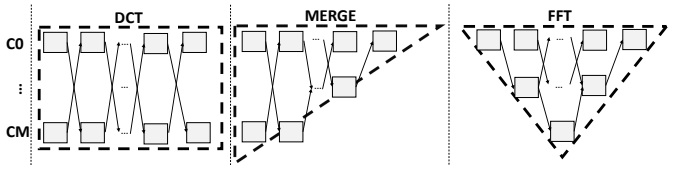


Fig. 6: Shape of parallelism of each benchmark (cores C0 through CM)

proposed *is*RA-FG approach can be implemented by other protection mechanisms.

Benchmarks and execution configurations: To experimentally evaluate our *is*RA-FG approach, we have conducted experiments using three different types of applications with respect to the number of tasks and shape of parallelism and which are taken from the StreamIT benchmarks [13]: i) Discrete Cosine Transformation (DCT) with few data dependencies allowing high parallelism, ii) Mergesort (MERGE) with a larger number of data dependencies reducing the available parallelism, and iii) Fast Fourier Transformation (FFT) with a significant number of data dependencies further reducing the amount of exploitable parallelism. To enable the parallelisation of the benchmarks, each benchmark has been divided into a number of tasks with their corresponding data dependencies (Table III). Figure 6 schematically depicts the shape of parallelism, where each box corresponds to a task of the benchmark and the arrows correspond to the dependencies E_{isRA} among cores.

To explore in details the behavior of the proposed approach, we have generated a number different execution configurations by varying the number of cores used and the number of parallel instances of each application:

- 1x2, 1x4, 1x8 configurations: we explore the parallel execution of a single application instance using 2, 4 and 8 cores, respectively.
- 2x2, 4x4, 8x8 configurations: we explore the execution of two application instances executed on two cores, four application instances executed on four cores and eight application instances executed on eight cores.

In addition, we explore whether the parallel execution of one application instance several times or the execution of several application instances on different cores provides better performance.

WCET acquisition: Since no existing static WCET analysis tool supports our platform, we applied a measurement-based approach to acquire the WCET of each task of the applications in isolation, i.e. when the task is executed alone on one core of the TMS platform. To perform our measurements on the real platform we compiled each application with -O0, i.e. no optimizations, disabled the data-caches and used the local timer of the core that executes the task, as per standard practice. To increase the reliability of the measurements, we have repeated our experiments more than 50 times and maintained the maximum observed value for the WCET. The Worst-Case Resource Accesses (WCRA) of each task was manually extracted from the produced binary of each application. These values were used to produce the near-optimal solutions offline. A summary of these values can be found in Table III. In addition, Table IV depicts the corresponding WCET values for the controller.

Compilation and data-placement: During execution, the compiler optimization flag is also set to -O0 and data-caches are disabled, in order to avoid claiming their performance benefits as benefits of *is*RA-FG. The controller and timing data are placed on the on-chip Multicore Shared SRAM Memory (MSM), while application data are placed in the off-chip main memory (DDR3), thus ensuring that the *is*RA-FG does not interfere with the task's execution.

Evaluation criteria: The evaluation goals in this study are multiple: (i) scalability of the method, as the number of cores increases, (ii) performance impact with respect to the shape of parallelism of the application, and (iii) performance impact of sequential and parallel execution of the same application. For all execution configurations the method for *is*WCET scheduling [14] was used to generate the offline solution. This offline *is*WCET solution is used as an input to our evaluation. The makespan of the *is*RA-FG is compared with the makespan given by (i) the *global* synchronisation approach (isRA-GLO), (ii) the *time-triggered* execution (isTT), where the *is*WCET computed offline is extended with the cost of the time-triggered mechanism for each task. The time-triggered mechanism cost is computed 270 cycles, since 70 cycles is required to write a control register and at least 200 cycles for serving an interrupt handling routine [15]. Each experiment has been executed ten consecutive iterations. The depicted makespan of a core is the average makespan observed for that core during these ten iterations. During the execution of all experiments, we observed no timing violations using the *is*RA-FG according to the offline solution.

B. Parallel execution of single application

Figures 7, 8 and 9 compare the makespan of each core in millions of cycles given by the proposed approach *is*RA-FG, the global run-time approach *is*RA-GLO and the time-triggered approach *is*-TT for the three applications, when a single instance is executed in parallel on a varying number of cores (configurations 1x2, 1x4 and 1x8). The first and foremost observation from the experiments is that the *is*RA-FG brings significant performance gains compared to the global run-time approach.

For the fully parallelizable DCT application, the makespan given by *is*RA-FG is similar for all cores for the same configuration. Since DCT is fully parallelisable and the tasks are symmetrical (in code and WCET), the makespan proportionally decreases with the number of cores used. When four

TABLE III: Benchmarks characteristics

Exec.	Number tasks													Sequential WCET	Total WCRA
	2x2, 4x4, 8x8	1x2		1x4			1x8								
Core	C0-7	C0	C1	C0	C1	C2-3	C0	C1	C2	C3	C4-5	C6-7		(cycles)	
DCT	32	16	16	8	8	8	4	4	4	4	4	4		981,120	69,808
MERGE	47	23	24	13	12	11	8	7	6	6	5	5		669,026	55,415
FFT	47	26	21	15	12	10	11	9	7	6	4	3		275,891	41,981

TABLE IV: Control phases WCET

	*is*RA-FG WCET		*is*RA-GLO WCET
	Fixed Cost	Cost/Edge	Fixed Cost
$C_{\mathcal{R}_v}$	355 + 251*$\|\mathcal{K}\|$	-	251*$\|\mathcal{K}\|^2$
$C_{\mathcal{U}_v}$	169 +131*$\|\mathcal{K}\|$	89+213*($\|\mathcal{K}\|$-1)	344*$\|\mathcal{K}\|^2$
$C_{TA_{\mathcal{R}}}$	81	-	-
$C_{TA_{\mathcal{U}}}$	178*$\|\mathcal{K}\|$	-	-

cores are used instead of two, the application makespan (which is given by the makespan of Core 0 –denoted C0– since the last task of the application is executed on that core) decreases by 23.18%. When eight cores are used, the application makespan decreases by 69.04% (compared to 2 cores) and by 59.70% (compared to 4 cores). The application makespan of *is*RA-GLO is higher and the makespan reduction, as the number of cores increases, is less, i.e. 13.98% when 4 cores are used instead of 2, 50.17% when 8 cores are used instead of 2 cores and 42.07% when 8 cores are used instead of 4 cores. Similar is the behavior of the DCT application for the rest of the cores since the tasks of the benchmark are symmetrical.

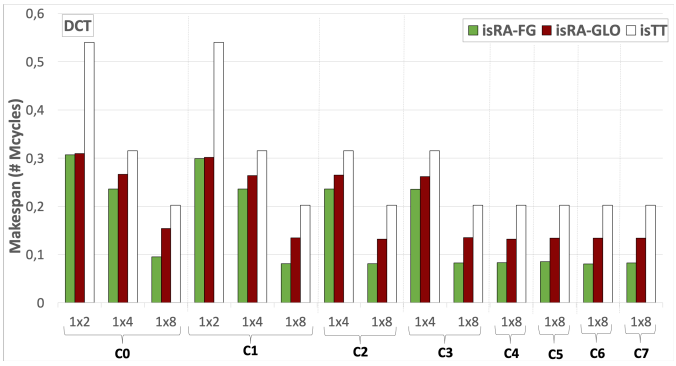


Fig. 7: Per core makespan for DCT: 1x2, 1x4 and 1x8 configurations.

For the partially parallelisable MERGE application, when four cores are used instead of two, the application makespan given by *is*RA-FG decreases by 30.22%. When eight cores are used, the application makespan decreases by 40.71% (compared to 2 cores) and by 20.76% (compared to 4 cores). Similar as before, the makespan reduction of *is*RA-GLO as the number of cores increases is less, i.e. 25.63% when 4 cores are used instead of 2, 20.62% when 8 cores are used instead of 2 cores and 4.02% when 8 cores are used instead of 4 cores. Due to the triangular shape of the application's parallelism (Figure 6) the reduction on the remaining cores is higher. For core 1, we observe 42.04% (from 2-to-4 cores), 61.22% (from 2-to-8 cores) and 33.09% (from 4-to-8 cores) for the *is*RA-FG approach. These values are compared to 35.23% (from 2-to-4 cores), 41.69% (from 2-to-8 cores) and 9.97% (from 4-to-8 cores) achieved by *is*RA-GLO approach. For core 2 and 3, we observe 48.89% (from 4-to-8 cores) for the *is*RA-FG approach compared to 13.92% achieved by *is*RA-GLO approach.

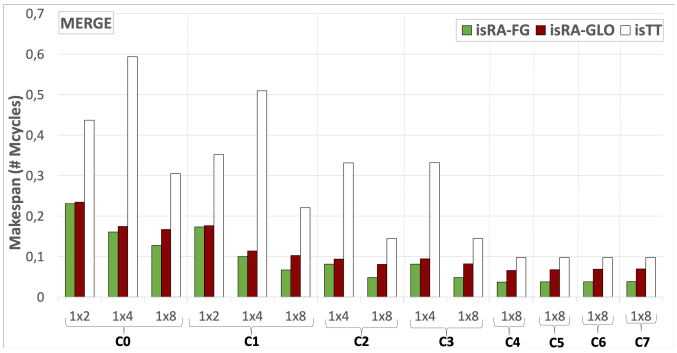


Fig. 8: Per core makespan for MERGE: 1x2, 1x4 and 1x8 configurations.

For the low parallelisable FFT application, when four cores are used the makespan of the application achieved by the *is*RA-FG approach is increased by 6.32% comparing to the 1x2 configuration. When eight cores are used, the makespan is decreased by 11.21% comparing to the 1x2 configuration and by 16.49% comparing to the 1x4 configuration. The *is*RA-GLO approach increases always the makespan by: 13.00% (from 2-to-4 cores), 13.54% (from 2-to-8 cores) and 0.49% (from 4-to-8 cores). Due to the triangular symmetric shape of the application's parallelism (Figure 6) the impact of the *is*RA-FG approach on the makespan of the remaining cores is higher. For core 1, we observe 13.66% increase (from 2-to-4 cores), 3.66% decrease (from 2-to-8 cores) and 15.24% decrease (from 4-to-8 cores) for the *is*RA-FG. The *is*RA-GLO approach increases by 30.04% (from 2-to-4 cores), whereas there is almost no affect when we transition from 2-to-8 cores and from 4-to-8 cores. For core 2 and 3, we observe 22.15% decrease in the makespan when we transition from 4 cores to 8 cores for the *is*RA-FG approach compared to 1.48% makespan reduction achieved by *is*RA-GLO approach.

To compare the gains of the proposed *is*RA-FG approach with respect to the *is*RA-GLO approach, i.e. $\frac{(isRA-GLO)-(isRA-FG)}{(isRA-GLO)}$, we depict in Figure 10 the minimum, average and maximum gains observed, for all the cores of each configuration. From the obtained results, we observe that for all the single-instance parallel configurations of the benchmarks, the *is*RA-FG approach achieves performance

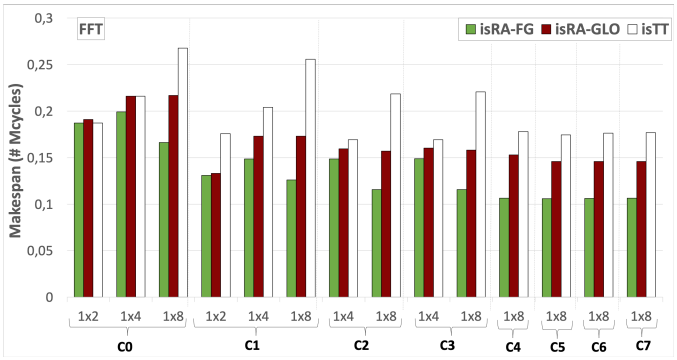


Fig. 9: Per core makespan for FFT: 1x2, 1x4 and 1x8 configurations.

improvements compared to the *is*RA-GLO approach. The gains of the *is*RA-FG approach are increased with increasing number of cores, as it is observed for all the applications. The minimum gain is observed when the application is parallelized using only 2 cores. This is expected as the negative impact, due to the sequential execution of the control phases in *is*RA-GLO, is small, since only two cores are involved. We also observe that as the number of cores increases, the range of the minimum gain and the maximum gain is also increased. This occurs due to the lower number of interferences accounted during the offline task scheduling for the 1x2 configuration than for the 1x8 configuration. In addition, the highest range of the makespan gain is observed for the MERGE application due to the triangular shape of the application parallelism.

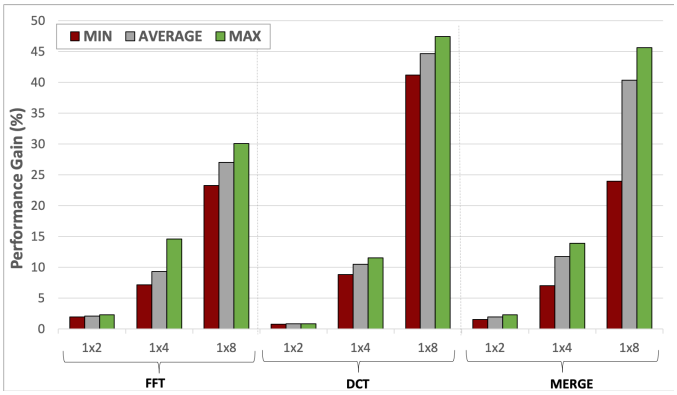


Fig. 10: Performance gain of 1x2, 1x4 and 1x8 configurations compared with the *is*RA-GLO approach

To summarize, significant performance gains are observed compared to the global run-time approach even for asymmetric and low parallelizable applications.

C. Sequential execution of applications

In these experiments, we execute several instances of the same application on different cores. Each core runs sequentially one instance of the application. Since the makespan of the cores is similar, Figure 11 depicts the average makespan among all iterations and all cores of the TMS for each configuration. We observe that for all benchmarks, as the number of application instances increases, the makespan of the application is also increased. This occurs due to the increasing number of interferences occurring during the parallel execution. For the time-triggered execution and the global run-time approach the increase in the makespan is significant compared to the proposed *is*RA-FG. For the DCT application, the increase in the makespan of *is*RA-FG from 2x2 configuration to 8x8 configuration is very low (7.34%) compared to the global approach (38.53%) leading to 80.94% less makespan increase. For the MERGE application, the increase in the makespan of *is*RA-FG from 2x2 configuration to 8x8 configuration is 69.20% compared to the 137.79% of the global approach leading to 49.78% less makespan increase. For the FFT application, the increase in the makespan of *is*RA-FG from 2x2 configuration to 8x8 configuration is 84.30% compared to the 187.63% of the global approach leading to 55.07% less makespan increase.

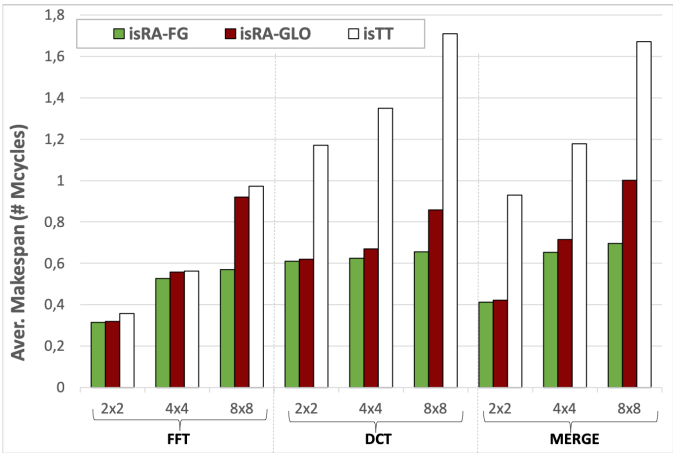


Fig. 11: Average makespan: 2x2, 4x4 and 8x8 configurations.

Similar to the previous section, Figure 12 compares the gains of the proposed *is*RA-FG approach with respect to the *is*RA-GLO approach. For all the configurations the *is*RA-FG approach achieves performance improvements compared to the *is*RA-GLO approach. The gains of the *is*RA-FG approach are increased with increasing the number of cores for all the applications. The minimum gain is observed when only 2 cores are used. However, compared to the parallel execution of a single application instance, the range of the minimum gain and the maximum gain is smaller and less affected by the number of used cores. This behavior is explained by the high probability of running the same tasks in parallel due to the parallel execution of several sequential instances of the same application.

D. Parallel Vs. Sequential execution

Figure 13 compares the makespan of core 0 (C0) of the proposed *is*RA-FG approach when i) we execute eight iterations of the parallelized application on 8 cores (8*(1x8)

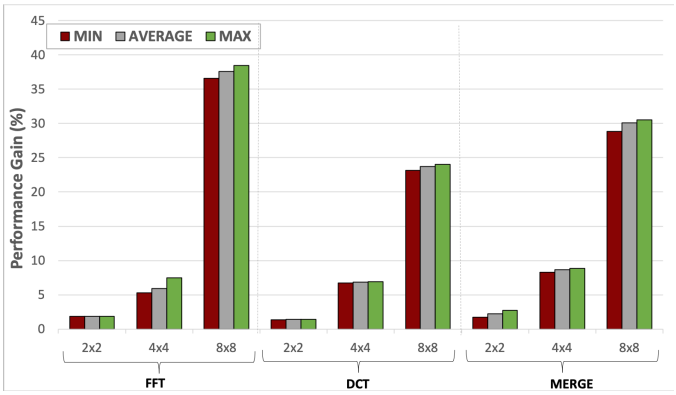


Fig. 12: Performance gain of 2x2, 4x4 and 8x8 configurations compared with the *is*RA-GLO approach

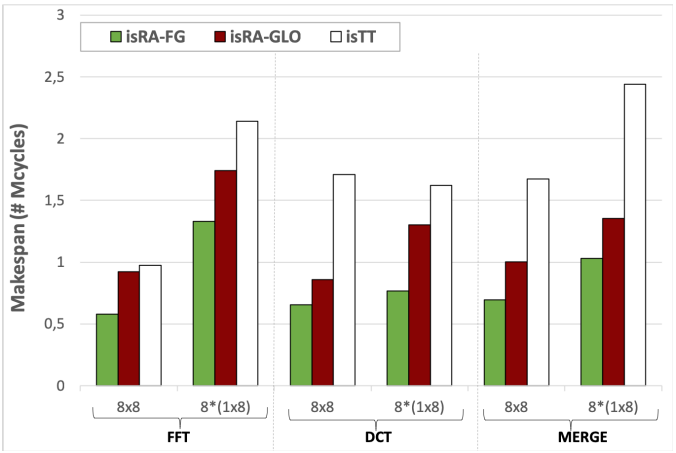


Fig. 13: Makespan comparison of sequential execution (8x8) and eight executions of the parallel execution (8*(1x8))

configuration) and 2) we execute eight sequential instances of the same application on the cores (8x8 configuration). We generally observe that executing sequentially multiple instances of the same applications *is*RA-FG has lower makespan compared to their equivalent parallelised version. For the FFT and MERGE applications, we observe that the offline time-triggered makespan is different in the parallel execution (8*(1x8) configuration) and in the sequential execution (8x8 configuration). For the FFT application, the time-triggered sequential execution is smaller than the parallel execution by 54.54%. For the MERGE application, the time-triggered makespan of the sequential execution is smaller than the parallel one by 31.46%. This is due to the fact that in the parallel execution the tasks concurrently executed on the cores are of different type and have different number of interferences leading to an increased *is*WCET. On the other hand, in the sequential execution, tasks of the same type are running on the cores, and thus the interferences used for the computation of the *is*WCET are only the ones defined by the running tasks. However, this is not the case for the DCT application since the tasks are symmetrical and have similar number of interferences. For a similar reason, but now due to the number of interferences occurring during execution, we observe that the *is*RA-FG applied on the sequential execution achieves lower makespan than the makespan of the *is*RA-FG applied on the parallel execution. More precisely, for the FFT we observe 56.42%, for MERGE we observe 32.27% and for DCT we observe 14.41% lower makespan for the sequential execution compared to the parallel execution.

VI. Related Work

The run-time mechanisms used for time critical systems on multi-cores can be categorized based on whether: i) only time-critical tasks are considered or best-effort tasks are also considered , and ii) the WCET used is pessimistic or interference-sensitive. A detailed survey on real-time systems is available in [16].

The run-time mechanisms considering only time-critical tasks must guarantee the timely execution of the complete task set. The mechanisms that consider the pessimistic WCET can start the execution of a new task as soon as a task finishes earlier than its pessimistic WCET. Typical examples of such approaches come from scheduling theory, e.g. [17], [18]. However, the use of pessimistic WCET over-approximates the interferences having a negative impact in performance and in schedulability. To tackle with over-approximated WCETs due to interferences, several approaches incorporate interference analysis and provide *interference-sensitive* WCETs, e.g. [19], [20], [4], [5], [6]. In general, these approaches result in improved timing guarantees, compared to over-approximated WCETs, as they provide a context-dependent upper-bound of the interferences for a particular schedule. To improve the provided upper-bounds, some approaches take into account the length of task overlapping, e.g. [5], or the precise timing of the requests, e.g. [21], or even provide contention-free schedules, e.g. [7]; a detailed survey of such approaches can be found in [22]. Regarding approaches that consider the length of task overlapping, *is*RA-FG can be directly applied by inserting checkpoints where the overlap occurs. Hence, each task is split into sub-tasks, where the *is*RA-FG is applied. However, the additional overhead compared to the added benefit has to be further explored. For approaches that provide *is*WCETs based on the precise timing of the requests, *is*RA-FG could be applied if the induced delays by the underlying hardware are no larger that the time offset of each request. Contention-free schedules, which do rely on precise timing of the requests, can be safely executed with *is*RA-FG. To further reduce the impact of the inherent pessimism in any kind of WCET estimations, several run-time mechanisms have been proposed, e.g. [15], [9]. In [9], the authors provide a run-time approach suitable for interference-sensitive WCET estimation. However, a single global synchronization mechanism is considered that serializes the run-time mechanism having a negative impact in performance. In contrast, the proposed *is*RA-FG approach provides a fine-grained synchronisation mechanism allowing the parallel execution of the core control phases.

The run-time mechanisms considering both time-critical and best-effort tasks assume that time-critical tasks can be timely executed, when they run alone (in isolation). Then, they take run-time decisions for the execution of the best-effort tasks, so as to still guarantee the timely execution of the time-critical tasks. Several of those approaches allow the concurrent execution of both time-critical tasks and best-effort tasks. The majority usually uses different confidence levels in the computation of the WCET, called criticality levels. The higher the criticality level, the larger and safer the WCETs [23]. At run-time, it is observed 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, the system switches to the higher criticality mode. Other approaches compute at run-time the remaining WCET, when the time-critical tasks run in isolation [24], [25]. The concurrent execution of time-critical and best-effort tasks is allowed, as long as the time-critical tasks can still finish their execution, if the best-effort tasks are paused. Other mechanisms allow to run the best-effort tasks only after the termination of the time-critical tasks. For instance, in [26] time critical and best effort tasks are scheduled. When a core finishes its execution before the estimated WCET of a time critical tasks, this slack is reallocated to a best effort task. Other approaches compute *interference-sensitive* WCETs 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 [27]. A similar work is presented in [28] where dynamic changes in the resource partitioning are allowed, when resources are underutilized. In [29], memory accesses are prioritized for time-critical tasks and a controller regulates the accesses to the shared memory whenever possible for the best-effort tasks. The proposed *is*RA-FG approach is orthogonal to the aforementioned works, since it focuses on providing timing guarantees for the time-critical tasks.

Another category of approaches that are related to our work are *non-blocking* synchronisation protocols, which provide synchronisation among tasks without the need of protection mechanism; we review them in terms of their impact in the WCET of the controller. In general, these are subdivided into *obstruction-free* [30], *lock-free* and *wait-free* [31]. Obstruction-free and lock-free approaches, do not impose any fairness, thus may suffer from starvation, resulting in unbounded WCET of the controller. Although wait-free algorithms must finish within finite number of steps, the upper-bound (if exists) is over-approximated, thus inflating the controller's WCET. However, incorporating a controller requires that its WCET is known and as tight as possible. Such tight bounds are achieved in protection mechanisms through ordering, e.g. FIFO semaphores, which isRA-FG uses.

VII. Conclusion

In this work, we propose the fine-grained interference-sensitive run-time adaptation technique *is*RA-FG that alleviates the limitations of the existing *is*RA, since it allows the parallel execution of the core control phases, whenever this is possible. We have presented the corresponding Response Time Analysis for our technique and have formally argued regarding its safety, under any possible execution. Our experimental evaluation indicates that the *is*RA-FG is scalable with the number of cores and outperforms the global run-time approach. For the parallel execution configurations, it provides an average performance gain of 1.64% for two cores, 10.54% for four cores and 37.32% for eight cores. For the sequential execution configurations, it provides an average performance gain of 1.81% when two cores are used, 7.15% when four cores are used and 30.46% when eight cores are used. Finally, we have shown that executing sequentially multiple instances of a benchmark yields better makespan compared to their equivalent parallelised version.

As future work, we will enhance the proposed fine-grained mechanism with dynamic capabilities during execution so as to further improve the performance. In addition, we will extend and evaluate our approach by exploiting the placement of the data in different levels of the memory hierarchy.

References

[1] N. Kim, B. C. Ward, M. Chisholm, C.-Y. Fu, J. H. Anderson, and F. D. Smith, "Attacking the one-out-of-m multicore problem by combining hardware management with mixed-criticality provisioning," in *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2016, pp. 1–12.

[2] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra *et al.*, "The worst-case execution-time problem—overview of methods and survey of tools," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 3, p. 36, 2008.

[3] J. Nowotsch, "Interference-sensitive worst-case execution time analysis for multi-core processors," Ph.D. dissertation, 2014.

[4] S. Skalistis and A. Simalatsar, "Worst-case execution time analysis for many-core architectures with NoC," in *International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS)*. Springer, 2016, pp. 211–227.

[5] H. Rihani, M. Moy, C. Maiza, R. I. Davis, and S. Altmeyer, "Response time analysis of synchronous data flow programs on a many-core processor," in *Proceedings of the 24th International Conference on Real-Time Networks and Systems*, ser. RTNS '16. New York, NY, USA: ACM, 2016, pp. 67–76. [Online]. Available: http://doi.acm.org/10.1145/2997465.2997472

[6] B. Rouxel, S. Derrien, and I. Puaut, "Tightening contention delays while scheduling parallel applications on multi-core architectures," *ACM Trans. Embed. Comput. Syst. (TECS)*, vol. 16, no. 5s, pp. 164:1–164:20, Sep. 2017.

[7] B. Rouxel, S. Skalistis, S. Derrien, and I. Puaut, "Hiding communication delays in contention-free execution for spm-based multi-core architectures," in *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[8] A. Esper, G. Nelissen, V. Nélis, and E. Tovar, "An industrial view on the common academic understanding of mixed-criticality systems," *Real-Time Systems*, vol. 54, no. 3, pp. 745–795, 2018.

[9] S. Skalistis, F. Angiolini, A. Simalatsar, and G. De Micheli, "Safe and efficient deployment of data-parallelizable applications on many-core platforms: Theory and practice," *IEEE Design & Test*, vol. 35, no. 4, pp. 7–15, 2018.

[10] T. David, R. Guerraoui, and V. Trigonakis, "Everything you always wanted to know about synchronization but were afraid to ask," in *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, 2013, pp. 33–48.

[11] A. Biondi and M. Di Natale, "Achieving predictable multicore execution of automotive applications using the let paradigm," in *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, April 2018, pp. 240–250.

[12] Texas Instruments, "TMS320C6678 Multicore fixed and floating-point digital signal processor," TI, Tech. Rep. SPRS691D, 2013.

[13] W. Thies and S. Amarasinghe, "An empirical characterization of stream programs and its implications for language and compiler design," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. ACM, 2010, pp. 365–376.

[14] S. Skalistis and A. Simalatsar, "Near-optimal deployment of dataflow applications on many-core platforms with real-time guarantees," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2017, pp. 752–757.

[15] A. Kritikakou, C. Rochange, M. Faugère, C. Pagetti, M. Roy, S. Girbal, and D. G. Pérez, "Distributed run-time wcet controller for concurrent critical tasks in mixed-critical systems," in *International Conference on Real-Time Networks and Systems (RTNS)*. ACM, 2014, pp. 139:139–139:148.

[16] A. Burns and R. I. Davis, "A survey of research into mixed criticality systems," *ACM Comput. Surv. (CSUR)*, vol. 50, no. 6, pp. 82:1–82:37, 2018.

[17] M. Bertogna, "Real-time scheduling analysis for multiprocessor platforms," Ph.D. dissertation, 2008.

[18] R. I. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM Comput. Surv. (CSUR)*, vol. 43, no. 4, p. 35, 2011.

[19] S. Altmeyer, R. I. Davis, L. Indrusiak, C. Maiza, V. Nelis, and J. Reineke, "A generic and compositional framework for multicore response time analysis," in *Proceedings of the 23rd International Conference on Real Time and Networks Systems*. ACM, 2015, pp. 129–138.

[20] M. Jacobs, S. Hahn, and S. Hack, "Wcet analysis for multi-core processors with shared buses and event-driven bus arbitration," in *Proceedings of the 23rd International Conference on Real Time and Networks Systems*. ACM, 2015, pp. 193–202.

[21] A. Schranzhofer, J.-J. Chen, and L. Thiele, "Timing analysis for TDMA arbitration in resource sharing systems," in *2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE, 2010, pp. 215–224.

[22] C. Maiza, H. Rihani, J. M. Rivas, J. Goossens, S. Altmeyer, and R. I. Davis, "A survey of timing verification techniques for multi-core real-time systems," *ACM Computing Surveys (CSUR)*, vol. 52, no. 3, p. 56, 2019.

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

[24] A. Kritikakou, C. Rochange, M. Faugère, C. Pagetti, M. Roy, S. Girbal, and D. G. Pérez, "Distributed run-time WCET controller for concurrent critical tasks in mixed-critical systems," in *International Conference on Real-Time Networks and Systems (RTNS)*. ACM, 2014, p. 139.

[25] A. Kritikakou, T. Marty, and M. Roy, "DYNASCORE: dynamic software controller to increase resource utilization in mixed-critical systems," *ACM Trans. Design Autom. Electr. Syst. (TODAES)*, vol. 23, no. 2, pp. 13:1–13:26, 2018.

[26] B. B. Brandenburg and J. H. Anderson, "Integrating hard/soft real-time tasks and best-effort jobs on multiprocessors," in *Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE, 2007, pp. 61–70.

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

[28] J. Nowotsch and M. Paulitsch, "Quality of service capabilities for hard real-time applications on multi-core processors," in *International Conference on Real-Time Networks and Systems (RTNS)*. ACM, 2013, pp. 151–160.

[29] 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 *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2013, pp. 55–64.

[30] M. Herlihy, V. Luchangco, and M. Moir, "Obstruction-free synchronization: Double-ended queues as an example," in *23rd International Conference on Distributed Computing Systems, 2003. Proceedings.* IEEE, 2003, pp. 522–529.

[31] M. Herlihy, "Wait-free synchronization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 13, no. 1, pp. 124–149, 1991.