Dynamic Interference-Sensitive Run-time Adaptation of Time-Triggered Schedules

Stefanos Skalistis, Angeliki Kritikakou

▶ **To cite this version:**

Stefanos Skalistis, Angeliki Kritikakou. Dynamic Interference-Sensitive Run-time Adaptation of Time-Triggered Schedules. ECRTS 2020 - 32nd Euromicro Conference on Real-Time Systems, Jul 2020, Virtual, France. pp.1-22, 10.4230/LIPIcs.ECRTS.2020.4 . hal-02927451

HAL Id: hal-02927451
https://hal.science/hal-02927451

Submitted on 1 Sep 2020

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

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

Dynamic Interference-Sensitive Run-time Adaptation of Time-Triggered Schedules

Stefanos Skalistis
Raytheon Technologies, Cork, Ireland
SkalistSt@rtx.com

Angeliki Kritikakou
University of Rennes, Inria, IRISA, France
angeliki.kritikakou@inria.fr

Abstract

Over-approximated Worst-Case Execution Time (WCET) estimations for multi-cores lead to safe, but over-provisioned, systems and underutilized cores. To reduce WCET pessimism, interference-sensitive WCET (*is*WCET) estimations are used. Although they provide tighter WCET bounds, they are valid only for a specific schedule solution. Existing approaches have to maintain this *is*WCET schedule solution at run-time, via time-triggered execution, in order to be safe. Hence, any earlier execution of tasks, enabled by adapting the *is*WCET schedule solution, is not possible. In this paper, we present a dynamic approach that safely adapts *is*WCET schedules during execution, by relaxing or completely removing *is*WCET schedule dependencies, depending on the progress of each core. In this way, an earlier task execution is enabled, creating time slack that can be used by safety-critical and mixed-criticality systems to provide higher Quality-of-Services or execute other best-effort applications. The Response-Time Analysis (RTA) of the proposed approach is presented, showing that although the approach is dynamic, it is fully predictable with bounded 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, obtaining significant performance improvements compared to static approaches.

2012 ACM Subject Classification Computer systems organization → Embedded software; Computer systems organization → Multicore architectures; Computer systems organization → Real-time systems

Keywords and phrases Worst-Case Execution Time, Interference-sensitive, Run-time Adaptation, Time-Triggered, Response Time Analysis, Multi-cores

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2020.4

Funding Partially supported by ARGO (`http://www.argo-project.eu/`), funded by the European Commission under Horizon 2020 Research and Innovation Action, Grant Agreement Number 688131.

1 Introduction

The constantly growing processing demand of applications has led the processor manufacturing industry towards multi-/many-core architectures. These architectures have multiple processing elements, called *cores*, providing massive computing capabilities, by being able to concurrently execute a high volume of tasks. Hard real-time systems have to provide *timing guarantees*, i.e., guarantee that tasks are completed before their respective latency requirements (deadlines). To rigorously provide such guarantees, deployment approaches schedule tasks on cores considering the Worst-Case Execution Time (WCET) of tasks.

However, in multi-core architectures, several resources are shared among the cores (such as memories and interconnects) introducing timing delays (interferences), changing the timing behavior of tasks and varying their WCET. Indeed, tasks WCETs, which include interferences, can be 7.5 times larger than the corresponding estimations without interferences,

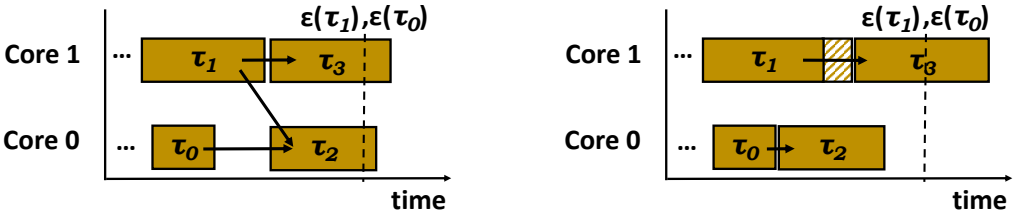
© Stefanos Skalistis and Angeliki Kritikakou;
licensed under Creative Commons License CC-BY

both experimentally measured [11, 13] and analytically computed [23, 24]. To account for all possible interferences, the WCET has to be over-approximated. This over-approximation practice has led to the "one-out-of-m processors" problem [8], where the additional processing capacity is negated by the pessimism of the WCET. As a result, the sequential execution (on a single core) may provide better timing guarantees than any parallel execution, which seriously undermines the advantages of utilizing multi-cores. To reduce the WCET pessimism, recent state-of-the-art research [15, 16, 19, 24] has proposed tighter WCET, called *interference-sensitive* WCET (*is*WCET). *is*WCET are computed by accounting for the interference that can occur only by the parallel-scheduled tasks. Hence, *is*WCETs are schedule-dependent, and they are valid only for the schedule solution they have been estimated for.

In order to guarantee a time-safe execution, this *is*WCET schedule solution has to be maintained during execution. Otherwise, additional interferences may occur, which have not been accounted for. To achieve that, time-triggered execution is usually applied, where the tasks are executed exactly at their start time assigned in *is*WCET schedule [18, 19]. Although time-triggered execution is time-safe, it prohibits any improvement on performance. Performance improvement can create slack, that can be used to increase the Quality-of-Service in safety-critical systems or execute other best-effort applications in mixed-critical systems. 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 [7]. The means to obtain any performance improvement is through run-time adaptation, using information of the task *actual execution time* (AET), that becomes available as the execution progresses. However, any adaptation occurring at run-time must be safe.

Existing *is*WCET run-time adaptation approaches [21, 22] allow tasks to be executed earlier-than-originally scheduled. Despite the potential earlier task execution, these approaches enforce the partial order of all tasks, provided by the *is*WCET schedule. In this way, additional interference due to earlier task execution cannot be introduced, maintaining the *is*WCET estimations valid. However, this enforced partial order of tasks limits the performance improvements that can be achieved through run-time adaptation. This limitation is illustrated in Figure 1, where arrows denote the partial order of tasks. As depicted in Figure 1a, a static run-time mechanism with enforced partial order can allow an earlier execution of successor tasks (τ_2 and τ_3), only when all their predecessor tasks have finished (τ_0 and τ_1). However, in permissive cases, τ_2 could be executed even earlier, since τ_0 has already finished execution before τ_1. Static mechanisms, that enforce the partial order, do not permit this earlier execution of τ_2, as τ_2 will insert interference to τ_1, which has not been computed during the creation of the *is*WCET schedule. Therefore, existing static mechanisms cannot exploit such opportunities created by the varying actual execution time of tasks across cores. However, assuming that τ_1 started earlier-than-originally scheduled, some time slack has been created at run-time, which can be exploited to further improve performance. As depicted in Figure 1b, if the additional *is*WCET, due to the interferences inserted by a new task running in parallel (τ_2 in this example, with interference illustrated in a stripped pattern), is less than the time slack, then the partial order of tasks can be safely relaxed, and τ_2 can be executed in parallel with τ_1.

In this work, we propose such a dynamic interference-sensitive run-time adaptation approach (*is*RA-DYN) that safely relaxes the partial order of tasks. The proposed approach exploits the actual execution time of tasks across cores to allow concurrent tasks to sustain more interference, than the one computed during the *is*WCET schedule, as long as the timing guarantees are preserved. Compared to existing approaches, the proposed approach



(a) Enforced partial order. **(b)** Relaxed partial order.

Figure 1 Motivational example: four tasks running on two cores and their *is*WCET dependencies.

is capable of exploiting the run-time variability due to a shorter task execution compared to the *is*WCET schedule computed offline. This run-time variability is created due to i) fewer interferences occurred during execution than the maximum possible interferences, used to offline compute the *is*WCET schedule, and ii) the executed path is different than the worst-case path of the task, used to compute the *is*WCET schedule. We provide the response timing analysis of the proposed approach, showing that timing guarantees are satisfied and any run-time adaptation does not alter the timing behavior of the system. To support our contributions, we perform extensive evaluation of the proposed approach on a real platform (8-core Texas Instruments TMS320C6678) for three applications and several execution configurations. The obtained results show that the proposed dynamic approach is able to provide performance improvements compared to the existing static approaches.

2 System model

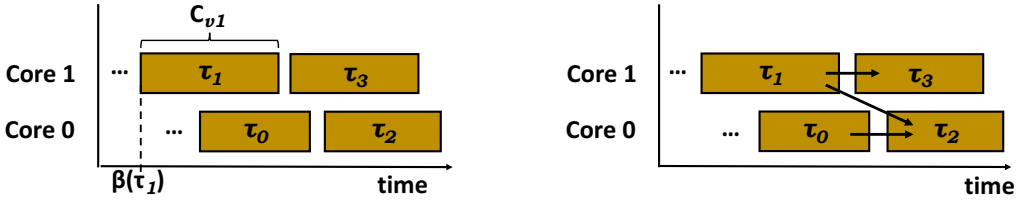
Let T denote the set of tasks of an application to be executed on the set of cores $\mathcal{K}$ of the target platform. The tasks of T can be either dependent, or independent, and are periodically executed in a non-preemptive manner. The proposed dynamic adaptation uses as input a *time-triggered schedule* that provides the start/end times of the tasks and their allocation to cores. Formally, we model such a *time-triggered schedule* S for the task-set T with the tuple (μ, β, ϵ), where $\mu(\tau)$ denotes the core allocation, i.e., the core k at which task τ is executed, and $\beta(\tau)$, $\epsilon(\tau)$ are the start and end times of the task, respectively. These times refer to the absolute time elapsed from the start of the period, and shall not be confused with the typical notion of *release time*. Such time-triggered schedule can be constructed by a scheduling algorithm providing timing guarantees, applied offline. Since the approach operates upon the input time-triggered schedule, any limitation will stem from the task model and scheduling algorithm, used offline to construct the time-triggered schedule. For clarity reasons, we will assume that tasks are released at the start of the period and their *is*WCET does not consider restrictions on the length of task overlapping or timing of the interference (see Section 6).

A time-triggered schedule S defines the partial ordering $\prec_S$ of the tasks, i.e., $\tau \prec_S \tau'$, iff task τ finishes its execution before τ' starts. Additionally, a schedule S 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 S, let E_{isRA} be the transitive reduction of the tasks partial order, i.e. $(E_{isRA})^* \equiv \prec_S$. Essentially, E_{isRA} is a set of scheduling dependencies E_{isRA}, such that a task τ depends only on the tasks $\{\tau'\}$ that finished immediately before it, on all cores $\mathcal{K}$, according to $\beta(\tau)$, i.e.:

$$(\tau, \tau') \in E_{isRA} \iff \nexists\, \tau'' \text{ s.t. } \mu(\tau) = \mu(\tau'') \wedge \epsilon(\tau) < \epsilon(\tau'') \leq \beta(\tau')$$

Table 1 Notation Summary.

Tasks & Graph	
T, τ	Task τ belonging to task-set T
$\mathcal{K}$, k	Core k from set of cores $\mathcal{K}$
E_{isRA}, E_{dyn}	Offline and run-time scheduling dependencies
$deg^-(\tau)$, $deg^+(\tau)$	The *indegree* and *outdegree* of task τ
$pred(\tau)$, $succ(\tau)$	Predecessors/successors of task τ
$\iota_E(\tau)$, $\iota_{max}(\tau)$	Context-dependent and upper bound of interference of task τ
Time-triggered solution	
$\mu(\tau)$	Core allocation $\mu(\tau)$ of task τ
$\beta(\tau)$, $\epsilon(\tau)$	Start time $\beta(\tau)$ and end time $\epsilon(\tau)$ of task τ
Response Time & WCET	
$\mathcal{R}_\tau$, $\mathcal{S}_\tau$, $\mathcal{X}_\tau$, $\mathcal{N}_\tau$	Ready $\mathcal{R}_\tau$, relax $\mathcal{S}_\tau$, execute $\mathcal{X}_\tau$, notify $\mathcal{N}_\tau$ phase
$R(\tau)$, $R(\mathcal{R}_\tau)$, $R(\mathcal{X}_\tau)$, $R(\mathcal{N}_\tau)$	Absolute response time of task τ and its phases
σ_τ	Time-slack of task τ
$C^N_{[L]}$	The WCET of code snippet L of Algorithm N
$C_{\mathcal{R}_\tau}$, $C_{\mathcal{S}_\tau}$ $C_{\mathcal{N}_\tau}$	Controller WCET for the corresponding phase
$t^\tau_{\mathcal{R}}$, $t^\tau_{\mathcal{S}}$, $t^\tau_{\mathcal{N}}$,	Time instance when the corresponding phase can execute successfully (all branches are not taken)



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

Figure 2 Construction of E_{isRA} scheduling dependencies, based on a given time-triggered schedule.

Figure 2 illustrates the construction of E_{isRA} given a time-triggered schedule. Notice that, for any task τ in the dependency relation E_{isRA}, the number of incoming edges (denoted as $deg^-(\tau)$) and the number of outgoing edges (denoted as $deg^+(\tau)$) is upper bounded by the number of cores $|\mathcal{K}|$, i.e. $deg^-(\tau) \leq |\mathcal{K}|$ and $deg^+(\tau) \leq |\mathcal{K}|$. The proposed approach relaxes, whenever possible, the dependency relation E_{isRA}, which we shall denote as $E_{dyn} \subseteq E_{isRA}$.

The proposed dynamic adaptation mechanism is divided into four phases, namely *ready*, *relax*, *execute* and *notify*, which are respectively denoted with $\mathcal{R}_\tau$, $\mathcal{S}_\tau$, $\mathcal{X}_\tau$ and $\mathcal{N}_\tau$ for any task τ. Since time-triggered schedules refer to absolute time, we shall denote the *absolute* response time of the control phases with $R(\mathcal{R}_\tau)$, $R(\mathcal{S}_\tau)$, $R(\mathcal{X}_\tau)$, $R(\mathcal{N}_\tau)$.

Finally, given a set of potentially parallel tasks T^E_τ to task τ, given a dependency relation $E \subseteq E_{isRA}$, we assume that the interference $\iota_E(\tau)$ that τ can cause to, and sustain from, T^E_τ is computable and upper-bounded by $\iota_{max}(\tau)$. This is a realistic assumption, e.g., a task τ with Worst Case Resource Accesses, $WCRA(\tau)$, to an arbitrated resource considering a fair Round-Robin arbiter with arbitration delay of D_{RR} will cause/sustain [18]:

$$\iota_E(\tau) = D_{RR} * \sum_{k \in \mathcal{K} \setminus \{\mu(k)\}} \min(WCRA(\tau), \sum_{\tau' \in T^E_\tau, \mu(\tau')=k} WCRA(\tau')) \quad (1)$$

$$\iota_{max}(\tau) = D_{RR} * (|\mathcal{K}| - 1) * WCRA(\tau) \quad (2)$$

3 Dynamic *is*RA (*is*RA-DYN)

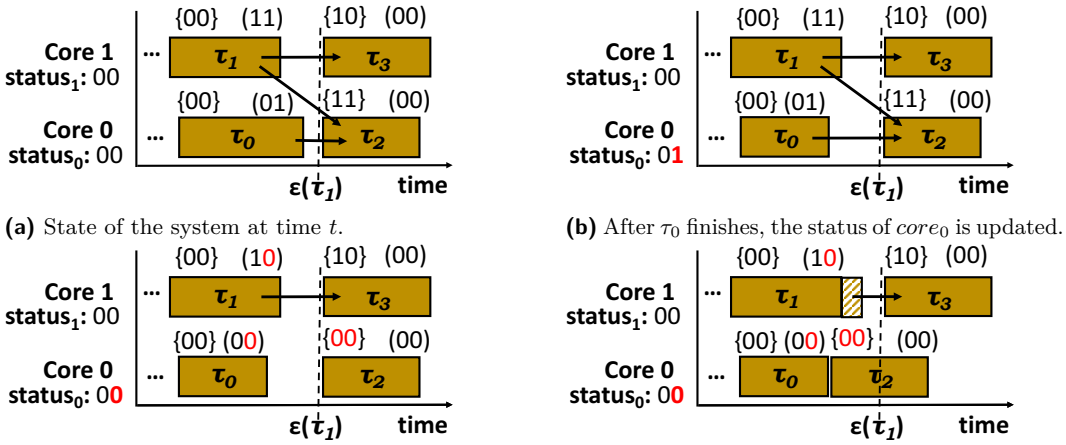
Run-time adaptation mechanisms for hard real-time systems, must guarantee that any adaptation decision does not violate the real-time constraints and the resulting execution is correct, i.e., no concurrency issue can occur. Furthermore, compared to traditional WCET schedules based on pessimistic WCET estimations, adapting interference-sensitive schedules poses an extra challenge: re-scheduling a task can increase the interference that another task sustains, as shown in Figure 1, potentially violating the timing guarantees. *Static* run-time adaptation [22] can safely adapt interference-sensitive time-triggered schedules by keeping fixed the partial order of task execution, preventing additional tasks overlaps, thus unaccounted interference to occur. Yet, static approaches miss adaptation opportunities due to this fixed partial order, being unable to provide further performance improvements.

Algorithm 1 *is*RA-DYN mechanism on core k.

```
   Input: Task τ, Array of all status vectors. (status_i[j]: the j-th status bit of the
          i-th core)
1  Function isRA-DYN (τ, status[ ]):
2      updateMinStart(k, β(τ))
3      while ¬ isReady(τ, status_k) do
4          if relax(τ, status_k) then
5              break;
6      execute(τ)
7      updateStatus(τ, status)
```

To address such cases, we propose a *dynamic* run-time adaptation mechanism, outlined in Algorithm 1, that is executed independently on each core and for each task. Each task *execution* is extended with control phases, *ready*, *relax* and *notify*. During the *ready* phase (L. 3), the controller checks if the current *active* task is ready, i.e., all previous tasks have finished, and thus, its dependencies have been met. In case the task is not ready, it tries to relax the partial order (L. 4) and checks again if the task is ready. To achieve a behavior that allows the partial order to change only when it is safe, a *global slack* is computed during execution, which is the minimum time-slack among all cores. The time-slack of a core is given by the amount by which the execution of its tasks has been sped up. The partial order, according to E_{isRA}, is allowed to be modified, if the introduced interference by any new task running in parallel is less than this global slack. This process continues, alternating a *ready* phase and a *relax* phase, until the task becomes ready and it is executed (L. 6). When the task finishes, the controller performs the *notify* phase, where it notifies all relevant cores that the task has finished (L. 7) and updates the necessary information for slack calculation (L. 2). A core k' is called relevant for any task τ executed on core k, when there exists an outgoing edge from task τ towards a task τ' on core k'. In order to enforce a particular ordering of the tasks (either the original partial ordering of E_{isRA} or any relaxation of it E_{dyn}) each core holds 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.

The following sections explain the controller phases with respect to the dependencies where relaxation can occur, i.e., scheduling dependencies. In case of data-dependent tasks, the data-dependencies are never removed.



(a) State of the system at time t.

(b) After τ_0 finishes, the status of $core_0$ is updated.

(c) Vector and status updates after τ_3 is relaxed.

(d) $core_1$ re-schedules τ_3.

Figure 3 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 illustrated with arrows.

3.1 Ready phase

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

$$readyVector_\tau[k] = 1 \quad \Leftrightarrow \quad (\tau', \tau) \in E_{dyn} \wedge \mu(\tau') = k \tag{3}$$

where $readyVector_\tau$ is the ready vector of task τ. The *ready* vectors are created offline for each task τ, based on the dependency relation E_{isRA}, but may be modified during a relax phase of the same core, or by a notify phase of another core to reflect the dependency relation E_{dyn}. For instance, in Figure 3a, the *ready* vector of task τ_2 is $\{11\}$, since it is has to wait for i) task τ_1 running on core 1 and ii) task τ_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 τ_2. On the other hand, the *ready* vector of task τ_1 is $\{00\}$, as no dependency exists from another task.

The functionality of the *ready* phase of the controller is described in Algorithm 2. Initially, the controller tries to gain access to the critical section of the *status* vector through the protection mechanism related to core k (L. 2). Once it has been granted, it checks if all task

Algorithm 2 Ready phase of *is*RA-DYN mechanism on core k.

Input: Task τ, $status_k[\,]$ bit vector.
Output: **true** if all dependencies $readyVector_\tau$ have been met; otherwise **false**

```
1 Function isReady(τ, status_k[ ]):
2     enterSection(k)
3     if (status_k & readyVector_τ) = readyVector_τ then
4         status_k ← status_k ⊕ readyVector_τ
5         exitSection(k)
6         return true
7     exitSection(k)
8     return false
```

dependencies have been already met, encoded by the task's ready vector (L. 3). If this is true, the task τ can be executed. For instance, tasks τ_0 and τ_1 in Figure 3a are considered ready, since the corresponding bits of the status vectors of Core 0 and Core 1 are clear 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 τ in its *status* (L. 4). 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 task is executed.

3.2 Notify phase

To implement the *notify* phase, a *notification* vector (of size $|\mathcal{K}|$) is required for each task τ 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.:

$$notifyVector_{\tau}[k] = 1 \;\Leftrightarrow\; (\tau, \tau') \in E_{dyn} \wedge \mu(\tau') = k \tag{4}$$

where $notifyVector_{\tau}$ is the notify vector of task τ. The *notification* vectors are created offline for each task τ, based on the dependency relation E_{isRA}, but may be modified during a relax phase of another core to reflect the dependency relation E_{dyn}. For instance, in Figure 3a, the notification vector of task τ_1 is (11); when it finishes execution, it has to notify task τ_2 running on core 0 (bit 0) and task τ_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 3b, the *status* vector of core 0 is 01, since task τ_0 finished execution and notified only core 0.

Algorithm 3 describes the functionality of the *notify* phase of the controller on core k. After the task τ on core k completes its execution, the controller has to update the *status* of all the relevant cores. To do so, for each successor τ' of task τ, the controller tries to gain access to the critical section of the successor's core protection mechanism (L. 3). If access is granted, the controller verifies if the dependency still exists (L. 4). If it exists, the controller tests if the previously occurred update of the core k has been already consumed by the core $\mu(\tau')$, where τ' is mapped to (L. 5). If this is true, the k-th bit in the status of core $\mu(\tau')$ is set, otherwise it clears the k-th bit from the ready vector of task τ', indicating that the dependency from core k has been met. For instance, Figure 3b illustrates this case

Algorithm 3 Notify phase of *is*RA-DYN mechanism on core k.

Input: Task τ, Array of all *status* vectors. ($status_i[j]$: the j-th *status* bit of the i-th core)

```
1 Function updateStatus(τ, status[ ]):
2     for τ' ∈ succ(τ) do
3         enterSection(μ(τ'))
4         if notifyVector_τ[μ(τ')] = 1 then
5             if status_μ(τ')[k] = 0 then
6                 status_μ(τ')[k] ← 1
7             else
8                 readyVector_τ'[k] ← 0
9         exitSection(μ(τ'))
```

where Core 0 updates its own bit after task τ_0 finishes. After the controller has updated all relevant status, it updates the start time of its active task with the time of the next task (L. 2, Algorithm 1) and computes the minimum among the active tasks (see Section 3.4).

3.3 Relax phase

In case the task is not ready to be executed, *is*RA-DYN tries to relax the partial ordering of the tasks, iff the introduced interference is less than the global slack (the amount that the execution has already advanced). That is, task τ is allowed to overlap with the active tasks, iff all active tasks started at least n time units before their time-triggered start time β, and task τ would introduce interference less than n time units to each one of them. This is illustrated in Algorithm 4 (L. 2) where the global *slack* has to be greater than the interference that task τ will introduce, denoted as $\iota_{max}(\tau)$, in addition to the WCET required of executing the relaxation, denoted as $C_{\mathcal{S}}$.

The relaxation strategy that *is*RA-DYN follows is an "all-or-nothing" approach, in the sense that either all the incoming scheduling dependencies, but no data ones, E^-_τ of task τ will be removed or the relaxation is postponed for a later invocation. The reasoning behind such design choice is that it provides short alternation between ready and relax phases. This minimizes the worst-case response time from the time a task becomes ready to when the task is executed by the controller. More formally, the result of such relaxation is:

$$E'_{dyn} = E_{dyn} \setminus E^-_\tau \quad \text{where} \quad E^-_\tau \subseteq pred(\tau) \times \{\tau\} \subset E_{isRA} \tag{5}$$

To achieve such relaxation, the controller of core k tries to gain access to its critical section (L. 4) and clears the k-th bit of the notification vector for each predecessor task τ' (L. 5-8), indicating that the dependency has been removed, as illustrated in Figure 3c. In order to reflect these changes to its own status and ready vectors, it registers which dependencies have been removed in a local variable, i.e., $modMask$ (L. 8). By definition, a dependency from a predecessor task τ' on the same core k is met, i.e., the notification from

Algorithm 4 Relax phase of *is*RA-DYN mechanism on core k.

Input: Task τ, $status_k[\,]$ bit vector.
Output: **true** if task τ is ready after the relaxation; otherwise **false**

```
1  Function relax(τ, status_k[ ]): bool
2      if getSlack() ≥ ι_max(τ) + C_{S_τ} then
3          modMask ← ¬(1 ≪ k)
4          for τ' ∈ pred(τ) do
5              if μ(τ') == k then continue
6              if ¬isDataDependent(τ',τ) then
7                  notifyVector_{τ'}[k] ← 0
8                  modMask[μ(τ')] ← 0
9          enterSection(k)
10         status_k ← status_k & modMask
11         readyVector_τ ← readyVector_τ & modMask
12         exitSection(k)
13         return readyVector_τ == 0
14     return false
```

core k has already occurred. Hence, the local variable is initialized with all bits set, except the k-th bit (L. 3). For the same reason, the k-th bit of that task's τ' notification vector is not reset (L. 6). Finally, the controller resets all the bits of its status and ready vector that were modified by the relaxation process, according to the local variable (L. 10-11), and tests (L. 13) if the task is indeed ready (to avoid re-execution of the ready phase), as shown in Figure 3d. Notice that, in case some of the tasks are data-dependent, the dependency is preserved (L. 6), thus ensuring proper ordering of data-dependent tasks.

3.4 Global slack computation

In order to relax the partial order of tasks, it is essential to know at run-time the amount of *global slack*, i.e., the minimum current time-slack across all cores. The time-slack of a core expresses the amount of time by which the execution of its tasks has been sped up. Speed-up occurs when the actual execution of a task is shorter than its *is*WCET. Formally, we define time-slack as the difference between the actual response time $R(\tau)$ of a task τ and its end time $\epsilon(\tau)$, and shall not be confused with the typical term slack, meaning laxity. As the actual response time $R(\tau)$ is not known a-priori, we use a safe slack approximation σ_τ:

$$\sigma_\tau = \beta(\tau) - t \quad \text{with} \quad \max_{\tau' \in pred(\tau)} R(\tau') \leq t \leq \beta(\tau) \tag{6}$$

where t is any time instance between the time-triggered start time and when the task becomes ready, i.e., all its predecessors have finished. Notice that σ_τ is safe since, $\epsilon(\tau) - R(\tau) \geq \beta(\tau) - \max_{\tau' \in pred(\tau)} R(\tau')$, i.e., the difference in response time between two consecutive (in time) tasks cannot be greater than the *is*WCET of the latter task, $\epsilon(\tau)$ - $\beta(\tau)$.

This safe approximation enables an efficient computation of the *global slack*, i.e., the minimum slack of all cores, at any time instance t, in a distributed manner, without requiring any sort of synchronisation or explicit exchange of information among cores. This is achieved by subtracting the current time instance t from the minimum start time of all active tasks, as outlined in Algorithm 5 (L. 9). To avoid inter-core information exchange, a global array is used to store the start time of the active task on each core and a global variable obtains the minimum value of the array. The start time of an active task is updated every time a core has to execute a new task (L. 3 of Algorithm 1). As soon as a core k finishes the notify phase of a task, it proceeds to its next task τ. As a new task is now active, the controller of

Algorithm 5 Slack computation on core k.

Input: Start time $\beta(\tau)$ of active task τ, Global array $startTimes$ of active tasks τ and global variable $minTime$ with the minimum value of $startTimes$.

```
1 Function updateMinStart(β(τ)):
2     prevStart ← startTimes[k]
3     startTimes[k] ← β(τ)
4     if minTime = prevStart then
5         for i ← 0 to |K| do
6             if minTime > startTimes[i] then
7                 minTime ← startTimes[i]
8 Function getSlack():
9     return minTime − getCurrentTime()
```

core k stores the old start time in a local variable and updates the start time of its active task with the new one (L. 2-3). If its old start time is equal to the minimum value of the array, it means that this controller was the owner of the minimum value, and thus, it has to recalculate the new minimum of the global array (L. 4-7). Otherwise, it delegates this computation to the controller that is the owner of the minimum value of the array.

3.5 Deadlock freedom

Since *is*RA-DYN is a distributed approach, it is important to establish its correctness. Here we prove that *is*RA-DYN is free from deadlocks, while time-correctness is addressed in Section 4. As a stepping stone, we first prove its static behavior, i.e., no relaxation occurs.

▶ **Lemma 1.** *Assuming that each set/reset operation on the k-th bit of the bit-vectors (status, readyVector, notifyVector) is atomic, the static behavior of* is*RA-DYN is free from deadlocks.*

Proof. Consider two dependent tasks, $(\tau, \tau') \in E_{isRA}$; there are two distinct cases when task τ finishes its execution and notifies core $\mu(\tau')$:

1. the $\mu(\tau)$-bit of the status for core $\mu(\tau')$ is not set ($status_{\mu(\tau')}[\mu(\tau)] = 0$), which results in setting the bit after notification (L. 6, Algorithm 3).
2. the $\mu(\tau)$-bit is already set ($status_{\mu(\tau')}[\mu(\tau)] = 1$) by some other task, which results in resetting the $\mu(\tau)$-bit of $readyVector_{\tau'}$ of task τ'. (L. 8, Algorithm 3).

At the ready phase of task τ' either the $\mu(\tau)$-bit of the ready vector is zero, or both the $\mu(\tau)$-bits of the status and the ready vector are set; in both cases task τ' is considered ready w.r.t. its dependency with task τ. In the former case, the value of that status vector bit is preserved (via XOR with the zero of the ready vector), in order to be reset by the corresponding task, while in the latter case that bit is reset. Since the value of the $\mu(\tau)$-bit of the status vector is the same before the notify phase and after the ready phase, it is straight forward to show that *is*RA-DYN is deadlock-free, via induction on E_{isRA} on all $|\mathcal{K}|$ bits. ◀

▶ **Theorem 2.** *Assuming that each set/reset operation on the k-th bit of the bit-vectors (status, readyVector, notifyVector) is atomic,* is*RA-DYN is free from deadlocks.*

Proof. Consider two dependent tasks, $(\tau, \tau') \in E_{dyn}$ and τ' is about to be executed; there are two distinct cases for task τ', namely either it is ready or it could be relaxed. The former case corresponds to the static behavior, which we have established its correctness from Lemma 1. In the latter case, there are two options for the $\mu(\tau)$-bit of $status_{\mu(\tau')}$:

1. the $\mu(\tau)$-bit has been set by τ
2. the $\mu(\tau)$-bit is not set

In both cases the $\mu(\tau)$-bit of $status_{\mu(\tau')}$ is reset, resetting also the $\mu(\tau)$-bit of $readyVector_{\mu(\tau')}$ (L. 10-11, Algorithm 4). Since τ' is about to execute, if the $\mu(\tau)$-bit is set, it cannot have been set by any other task τ'' than τ, as it would have been already reset by the succesive task of τ'' on core $\mu(\tau')$. It is thus, straightforward to show that *is*RA-DYN is deadlock-free, via induction on E_{dyn} on all $|\mathcal{K}|$ bits. ◀

In the presented algorithms, the modification of the bit-vectors is protected, satisfying the assumption for atomic set/reset operation of bits. In particular, since a ready and a relax phase cannot be executed simultaneously on the same core, it suffices to use a single protection mechanism per core. Additionally, since data-dependencies are always preserved (L. 7 in Algorithm 4), the execution with *is*RA-DYN cannot introduce race conditions to the application itself. It should be stressed that the global variable, with the minimum start

time of all active tasks across cores, does not require protection in order to be safe. The minimum start time is genuinely increasing, as the execution progresses, and its used for the calculation of the global slack. Accessing the global variable without protection can result in missing write from another core. This means that the controller uses an older value, which is smaller then the new one. This only results in smaller global slack computation, and thus, only missed opportunities of relaxation. This is deliberately done so, in favor of run-time performance.

4 Response time analysis

Introducing run-time mechanisms into time-critical systems can improve system performance, by better utilisation of the system resources. Nevertheless, the controllers themselves require processing time, thus they can alter the timing behavior and potentially violate timing guarantees, if the controller WCET 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). Especially for interference-sensitive system, attention must be paid to potential interference created by the controller, i.e., accesses to shared variables among cores (*status*, *ready* and *notify* vectors). If these variables are placed alongside with the task data, additional interferences must be accounted, due to the parallel execution of a control phase and a task. Multiple approaches exist to mitigate this effect; the control data can be placed in a separate memory accessed by a separate bus, when the platform provides such a feature. Alternatively, shared control data can be placed in the local memories of each core and accessed through remote reads/writes [2]. If such solutions are not possible, the amount of induced interference can be controlled, either by using resource-partitioning techniques common in real-time systems or by bounding the number of invocations of the controller, e.g., using non-interrupting hardware events [21].

In Algorithm 1, the proposed control mechanism is divided into two alternating phases, namely *ready* and *relax*, followed by two consecutive phases, *execute* and *notify*. We consider the absolute response time of a task τ to be when it finishes execution and the notify phase has performed all the status updates, i.e., the absolute response time of a task τ is the same as response time of its update phase:

$$R(\tau) = R(\mathcal{N}_\tau) \tag{7}$$

Notice that, while the execution phase $\mathcal{X}_\tau$ has a fixed *is*WCET (without considering the additional interferences due to relaxation), the *ready* and *notify* phases have varying *is*WCET, which depends on a number of factors. For any task τ, the *is*WCET 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 τ, according to the scheduling dependencies E_{isRA}. We denote with $C^N_{[L]}$ the WCET of the controller part that corresponds

to the snippet L, i.e., the sequence of lines L of Algorithm N. We perform the RTA for the most restrictive case, i.e. for tasks with data-dependencies. An RTA for independent tasks, would at least provide the same response time bounds for the same task set, if not better.

Accessing a critical section. While one core can be only at one control phase at any time instance, different cores can be in different phases. Hence, for a core to enter any particular critical section, it may have to wait for all the other cores to finish their critical section. The WCET of the critical sections of the ready, notify and relax phase, are $C^2_{[3-6]}$, $C^3_{[3-9]}$ and $C^4_{[9-12]}$, respectively. Thus, the worst-case wait time for a core to access critical section i is:

$$C_i = (|\mathcal{K}| - 1) * \max\left(C^2_{[3-6]}, C^3_{[3-9]}, C^4_{[9-12]}\right) \tag{8}$$

Ready Phase. Let $t^\tau_{\mathcal{R}}$ be the time instance that task τ, running on core k, becomes ready, i.e., all predecessor tasks have finished their corresponding notify phases:

$$t^\tau_{\mathcal{R}} = \max_{\tau' \in pred(\tau)} R(\tau') \tag{9}$$

The response time of the ready phase, if the controller is invoked precisely at time $t^\tau_{\mathcal{R}}$, is the WCET of acquiring access to critical section k plus the WCET of executing that section:

$$R(\mathcal{R}_\tau) \leq t^\tau_{\mathcal{R}} + C_{\mathcal{R}_\tau} + C_{TA_{\mathcal{R}}} \quad \text{with} \quad C_{\mathcal{R}_\tau} = C^2_{[3-6]} + C_k \tag{10}$$

where $C_{TA_{\mathcal{R}}}$ is a timing alignment constant, analysed below. The response time $R(\mathcal{R}_\tau)$ is defined recursively, as it depends on the maximum response time of preceding tasks ($t^\tau_{\mathcal{R}}$). This will assist us in proving that under any AET, the execution respects the timing guarantees.

Execute Phase. As there are no preemptions during the execution of a task, if $\iota_{E_{dyn}}(\tau)$ is the interference that task sustains because of the relaxed dependency relation E_{dyn}, the response time for the execution phase of the controller is:

$$R(\mathcal{X}_\tau) \leq R(\mathcal{R}_\tau) + C_\tau + \iota_{E_{dyn}}(\tau) \tag{11}$$

Notify Phase. Following the task execution τ, on core k, the controller updates each core's status and the minimum start time of the active tasks. For each outgoing dependency, the core k has to gain access to a distinct critical section and perform a write to either the status or the ready vector:

$$R(\mathcal{N}_\tau) \leq R(\mathcal{X}_\tau) + C_{\mathcal{N}_\tau} \quad \text{with} \tag{12}$$

$$C_{\mathcal{N}_\tau} = C^5_{[1-7]} + \sum_{\tau' \in succ(\tau)} \left(C_{\mu(\tau')} + C^3_{[4-8]}\right) = C^5_{[1-7]} + deg^+(\tau) * \left(C_k + C^3_{[4-8]}\right) \tag{13}$$

Since the worst-case waiting time in Equation 8 is constant, we have replaced $C_{\mu(\tau')}$ with C_k to derive the final expression.

Relax Phase. Let time instance $t^\tau_{\mathcal{S}}$ be a time instance, where task τ running on core k is not ready yet, i.e., $t^\tau_{\mathcal{S}} < t^\tau_{\mathcal{R}}$, but the global slack is large enough to accommodate the additional interferences. If the relax phase is invoked at time $t^\tau_{\mathcal{S}}$, it has to remove all the

dependencies (excluding the data-dependencies) and acquire access to its critical section, in order to write the status vector. Hence, its response time is:

$$R(\mathcal{S}_\tau) \leq t^\tau_{\mathcal{S}} + C_{\mathcal{S}_\tau} \quad \text{with} \tag{14}$$

$$C_{\mathcal{S}_\tau} = C^4_{[2-4]} + \left(deg^+(\tau) - 1\right) * C^4_{[9-13]} + C^4_{[4-5]} + C_k + C^4_{[5-8]} \tag{15}$$

Notice that the loop body (L. 9-13) in Algorithm 4 is executed only $deg^+(\tau) - 1$ times, since the dependency from the core itself is by default met. This is reflected by the term $(deg^+(\tau) - 1) * C^4_{[9-13]} + C^4_{[4-5]}$ in the response time $R(\mathcal{S}_\tau)$.

Timing alignment. In the RTA of the ready phase, we assumed that the controller starts precisely at the time when all the required status updates have been performed (for the ready phase). Nevertheless, since the tasks can be executed in less time than their *is*WCET, there is a possibility that the controller is already inside a relax phase, when the last status update occurs. In the worst-case, there will be a single data-dependency that is not removed by the relax phase. Therefore, the timing alignment constant $C_{TA_\mathcal{R}}$ for the ready phase is:

$$C_{TA_\mathcal{R}} = C_\mathcal{S} - C^4_{[6-8]} + C^1_{[3]} \tag{16}$$

where $C_\mathcal{S}$ is the WCET of relax phase, with the maximum number of dependencies, i.e., $|\mathcal{K}|$.

4.1 Safety

Having the WCET and the response time of the controller phases, we need 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 run-time reduction of execution times, i.e., $R(\tau) \leq \epsilon(\tau)$ for all tasks τ.

Let $C_{\mathcal{R}_\tau}$, $C_{\mathcal{N}_\tau}$, $C_{\mathcal{S}_\tau}$ denote the WCET of the control phases, and $C_{TA_\mathcal{R}}$ the timing alignment constants, as 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:

$$\epsilon(\tau) - \beta(\tau) = C_{\mathcal{R}_\tau} + C_{TA_\mathcal{R}} + C_\tau + C_{\mathcal{N}_\tau} + C_{\mathcal{S}_\tau} \tag{17}$$

Before proving the safety of the approach, we establish some properties regarding the impact of relaxations to *is*WCET of the task and control phases.

▶ **Property 1.** *Relaxation does not increase the WCET of control phases ($C_{\mathcal{R}_\tau}$, $C_{\mathcal{N}_\tau}$, $C_{\mathcal{S}_\tau}$).*

Proof. The WCET of the control phases depends on the indegree and outdegree of each task τ according to the dependency relation E_{isRA} (Equations 10, 13, 15). The dependency relation E_{dyn} is a genuinely decreasing relation (Equation 5) starting from E_{isRA}, thus the WCET of the control phases decreases with each relaxation. ◀

▶ **Property 2.** *Given a relaxed dependency relation $E_{dyn} \subseteq E_{\text{is}RA}$, a task τ can suffer additional interference at most equal to its slack:*

$$\sigma_\tau \geq \iota_{E_{dyn}}(\tau) \tag{18}$$

Proof. In case task τ is the task with the minimum start time among the active tasks, then $\sigma_\tau \geq \iota_{max}(\tau)$ (L. 2 in Algorithm 4). Otherwise, let τ_{min} be the task with minimum start time, i.e., $\beta(\tau) \geq \beta(\tau_{min})$. By equation 6:

$$\sigma_\tau - t \geq \sigma_{\tau_{min}} - t \Rightarrow \sigma_\tau \geq \sigma_{\tau_{min}} \tag{19}$$

Therefore, $\sigma_\tau \geq \iota_{max}(\tau) \geq \iota_{E_{dyn}}(\tau)$. ◀

▶ **Theorem 3.** *For any safe scheduling solution, derived by adding the controller costs ($C_{\mathcal{R}_\tau}$, $C_{\mathcal{N}_\tau}$, $C_{\mathcal{S}_\tau}$, $C_{TA_\mathcal{R}}$) to the is*WCET *(C_τ) of the tasks T, the* is*RA-DYN execution is safe under any AET, i.e.:*

$$R(\tau) \leq \epsilon(\tau) \tag{20}$$

Proof. Proof by induction on the dependency relation E_{dyn}.
Base Case: For tasks τ with no predecessors ($pred(\tau) = \emptyset$, $\beta(\tau) = 0$) from Eq. 7 and Eq. 12 we establish:

$$R(\tau) \leq R(\mathcal{X}_\tau) + C_{\mathcal{N}_\tau} \overset{(11)}{\Longrightarrow} R(\tau) \leq R(\mathcal{R}_\tau) + C_\tau + C_{\mathcal{N}_\tau} + \iota_{E_{dyn}}(\tau) \overset{(10)}{\Longrightarrow} \tag{21}$$

$$R(\tau) \leq t^\tau_\mathcal{R} + C_{\mathcal{R}_\tau} + C_{TA_\mathcal{R}} + C_\tau + C_{\mathcal{N}_\tau} + \iota_{E_{dyn}}(\tau) \overset{(9)}{\Longrightarrow} \tag{22}$$

$$R(\tau) \leq \max_{\tau' \in pred(\tau)} R(\tau') + C_{\mathcal{R}_\tau} + C_{TA_\mathcal{R}} + C_\tau + C_{\mathcal{N}_\tau} + \iota_{E_{dyn}}(\tau) \tag{23}$$

Since $pred(\tau) = \emptyset$, the response time of the predecessors is zero. Also no adaptation can occur since $\beta(\tau) = 0$, thus no additional interference is introduced, i.e. $\iota_{E_{dyn}}(\tau) = 0$:

$$R(\tau) \leq 0 + C_{\mathcal{R}_\tau} + C_{TA_\mathcal{R}} + C_\tau + C_{\mathcal{N}_\tau} + 0 \overset{(17)}{\Longrightarrow} \tag{24}$$

$$R(\tau) \leq \epsilon(\tau) - \beta(\tau) \overset{\beta(\tau)=0}{\Longrightarrow} R(\tau) \leq \epsilon(\tau) \tag{25}$$

Induction step: Suppose that (20) holds for all predecessors of task τ. Starting from equation (23) and through equation (6) we establish:

$$R(\tau) \leq \beta(\tau) - \sigma_\tau + C_{\mathcal{R}_\tau} + C_{TA_\mathcal{R}} + C_\tau + C_{\mathcal{N}_\tau} + \iota_{E_{dyn}}(\tau) \overset{(17)}{\Longrightarrow} \tag{26}$$

$$R(\tau) \leq \epsilon(\tau) - \sigma_\tau + \iota_{E_{dyn}}(\tau) \tag{27}$$

By Property 2 we know that $\iota_{E_{dyn}}(\tau) - \sigma_\tau \leq 0$, therefore, concluding the proof. ◀

We have therefore established that *is*RA-DYN is timely safe, and relaxes the dependency relation when enough global slack exists to accommodate the additional interference. Furthermore, the execution is work-conserving, w.r.t. E_{dyn}, which improves run-time performance, as shown in Section 5.

5 Experimental Evaluation

5.1 Experimental Setup

Platform. A real multi-core COTS platform, i.e., the TMS320C6678 chip (TMS in short) of Texas Instrument [25] is used for the experiments. The platform characteristics are depicted in Table 2. The *is*RA-DYN mechanism is implemented as a bare-metal library, with low-level functions for the controller phases using TMS hardware semaphores. The *is*RA-DYN semaphore implementation is applicable to any platform, since semaphores are a fundamental building block. In the rare case that no such hardware support exists, a software implementation can be used. However, the *is*RA-DYN approach can be implemented by other protection mechanisms.

Benchmarks. To experimentally evaluate our *is*RA-DYN approach, we have conducted experiments using three different applications with respect to the number of tasks, WCET, and WCRA taken from the StreamIT benchmarks [26]: i) Discrete Cosine Transformation (DCT), ii) Mergesort (MERGE), and iii) Fast Fourier Transformation (FFT).

Table 2 Benchmark and platform characteristics.

(a) Benchmark.

	No. tasks	Seq. WCET (cycles)	No. WCRA
DCT	32	981,120	69,808
MERGE	47	669,026	55,415
FFT	47	275,891	41,981

(b) TMS platform.

DSP char.	Instr.	Freq.	L1P	L1D	L2
	8	1 GHz	32 KB	32 KB	512 KB
DSPs	8	**NoC**	TeraNet (Delay: 11 cycles)		
L3	4 MB	**DDR3**	512 MB	**Sem.**	32 cycles

WCET and WCRA acquisition. Since no existing static WCET analysis tool supports the TMS platform, a measurement-based approach has been used to acquire the WCET of each task. Obtaining safe and context-independent measurements requires to eliminate the sources of timing variability [6], by disabling data-caches, removing interferences (i.e., the task is executed alone on one core) and providing input data to enforce the worst-case path. To perform our measurements on the real platform, we used the local timer of the core. To increase the reliability of the measurements, we have followed the approach of multiple executions. Each task has been executed 50 times, which has been shown to provide a small standard deviation [14], and maintained the largest observed value. The application has been compiled with -O0, i.e., no optimizations, in order to obtain the WCRA of each task by the produced binary. Table 2 depicts the overall WCET, WCRA and number of tasks of each benchmark, used to obtain the offline near-optimal solutions.

Data-placement. The controller data are placed on the on-chip Multicore Shared SRAM Memory (MSM), while application data are placed in the off-chip main memory (DDR3), ensuring that the *is*RA-DYN does not interfere with the task's execution.

Comparison. The evaluation of the proposed approach can be achieved by comparing run-time execution time of the tasks allocated on each core (a.k.a. makespan) obtained by the proposed *dynamic* approach (*is*RA-DYN) and the *static* run-time approach (*is*RA-LOCK) that enforces the partial order of tasks [22]. The offline *is*WCET schedule has been generated by [24] and it is used as an input to both approaches. To attribute the observed gains to isRA-DYN, any system parameter, that may lead to timing variability at run-time, should be controlled and explored independently, whenever possible. These parameters are mainly the interferences, the different execution paths of the benchmarks and the impact of caches. Therefore, we initially explore the timing variability that each benchmark can have, when executed on TMS platform and a single system parameter is tuned each time. Then, we present the gains provided by *is*RA-DYN and *is*RA-LOCK by comparing the makespan under different variations at the timing variability of benchmarks. Last, but not least, we compare the overhead of *is*RA-DYN and *is*RA-LOCK approaches.

5.2 Evaluation results

Characterization of timing variability. The main system parameters that can alter the execution time are the occurring interferences, the diverse execution paths of the benchmarks and the caches. In this first experimental section, we tune each of these parameters independently in order to characterize its impact to the timing variability per benchmark. To compute the timing variability, the execution time of the best observed case and the worst observed case are compared. Table 3 shows the timing variability due to caches and

Table 3 Benchmark timing variability.

(a) Interference variability

Caches	Path	DCT	MERGE	FFT
Disabled	**Best-Path**	32.13%	46.67%	55.03%
	Worst-Path	44.80%	43.78%	52.70%
Enabled	**Best-Path**	0.43%	0.91%	3.58%
	Worst-Path	0.32%	0.91%	3.29%

(b) Cache variability (No interferences)

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

(c) Path variability (No interferences)

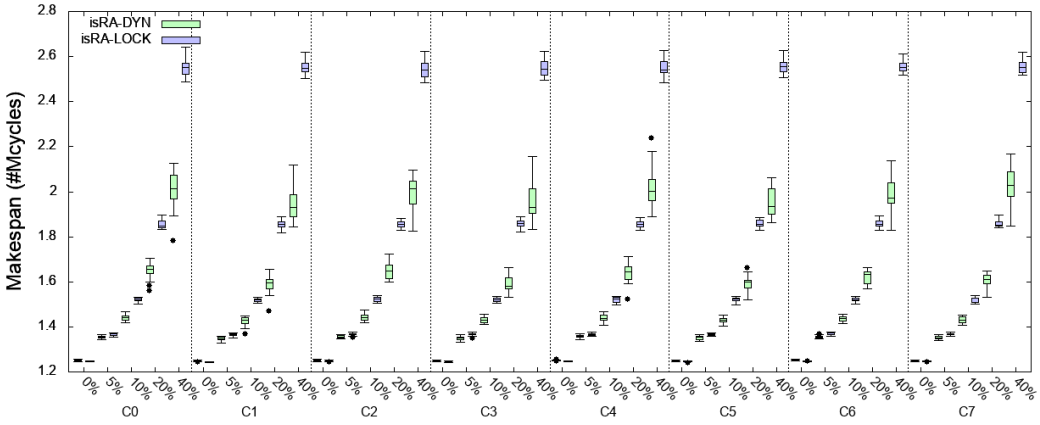
Caches	DCT	MERGE	FFT
Disabled	46.65%	12.84%	0.15%
Enabled	40.51%	14.69%	0.46%

diverse paths (computed without any interferences, i.e., running the benchmark alone on a core), and the timing variability due to interferences, when all cores are running the same benchmark. We observe that even when the application is executed in isolation, the impact of caches in execution time is quite high for all applications, with 71.14% on average. The impact of different execution paths depends on the application type, thus it is higher for the DCT, since it is an application that has several execution paths and, much smaller for FFT, which is a single-path application. Last but not least, we observe an important impact of the interferences, with 45.85% on average, with disabled caches. When caches are enabled the interference impact is reduced, since the cache sizes are large enough to keep the data locally.

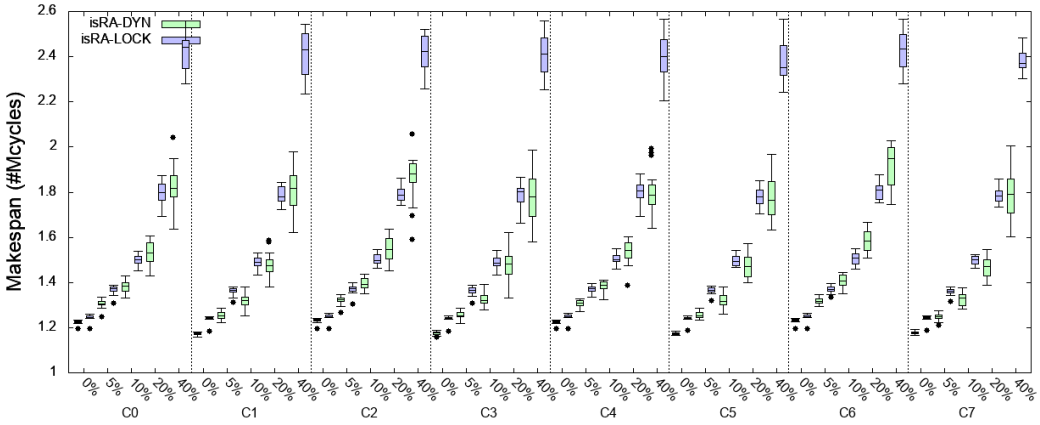
Makespan comparison. We perform an exhaustive set of experiments to explore and quantify the behavior of the proposed approach. We have used three configurations, i.e., two, four and eight application instances on two, four and eight cores, respectively. In addition, in order to explore the behavior of *is*RA-DYN with respect to the timing variability, due to interferences, caches, and multiple execution paths of the application, we have performed experiments, where we insert at each task a timing variability from 0% up to 40%, on average. Each experiment has been executed twenty consecutive iterations. During the execution, we observed no timing violations according to the offline solution. Due to page limitations, we only present the measured makespan of each core for the eight core configuration in Figure 4, in the form of box plot. However, we thoroughly analyze the behavior of our approach by providing the gains of the proposed *is*RA-DYN compared to *is*RA-LOCK for all experiments. The gain is given by computing the makespan gain, i.e., $\frac{(isRA-LOCK)-(isRA-DYN)}{(isRA-LOCK)}$, for each core. Tables 5, 4 and 6 depict the average makespan gain per core and the average makespan gain across all cores, for all configurations.

a) General observations: The first and important observation is that the behavior of *is*RA-LOCK is similar, in terms of minimum, maximum and average makespan, for all cores for all benchmarks, under any timing variability. This behavior of *is*RA-LOCK is due to the fixed partial task order. This behaviour motivates the use of a dynamic approach that can explore the variability occurring at run-time. Compared to the *is*RA-LOCK, the makespan distribution of the *is*RA-DYN among cores is varying. As *is*RA-DYN performs partial order relaxations, allows earlier task execution and additional interference to occur, which varies the core's makespan. When the variability is increased, this variation becomes more important.

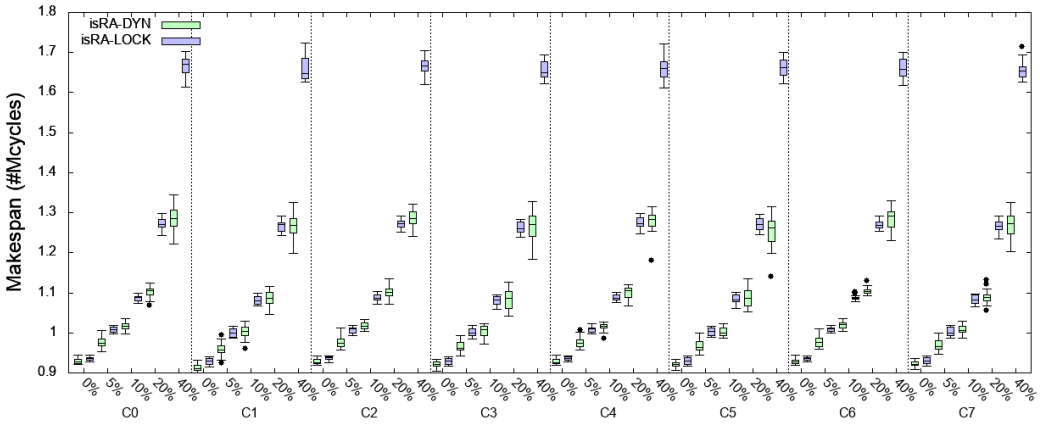
b) Minimum timing variability (0%): This experimental set-up is the worst for the proposed approach, since the timing variability of the benchmarks is eliminated as much as possible.



(a) DCT.



(b) MERGE.



(c) FFT.

Figure 4 Eight core configuration: *is*RA-DYN and *is*RA-LOCK makespan per core.

To achieve that, the same execution path is used among executions and caches are disabled. However, it is not possible to eliminate the interferences occurring from the parallel execution of tasks. For all the experiments, we observe that the behaviour of *is*RA-DYN improves over the behavior of *is*RA-LOCK, in all cores, as the number of cores increases. More precisely, for the configuration with two cores and 0% variability, *is*RA-DYN provides a small gain (from 0.08% for MERGE up to 0.22% for FFT, with an average of 0.145% among all applications). As the number of cores is increased, the gains are also increased, especially for MERGE. Compared to the two core configuration, the gain is increased on average by a factor of x3.11 for DCT, x36.63 for MERGE and x1.93 for FFT, when four cores are used, and by x2.95 for DCT, x44.64 for MERGE and x4.28 for FFT. The high gain factor of the MERGE benchmark is due to the low gain when only two cores are used. The lower gain, when only two cores are used, is attributed to the small number of interferences occurring during execution in combination with a bit higher run-time overhead, due to the relax phase, compared to *is*RA-LOCK. However, as the number of cores is increased, the number of occurring interference is increased, and thus, the gain is higher. As the only source of timing variability is the interferences in this experimental, the achieved gain of *is*RA-DYN verifies that the proposed approach is capable of exploring the occurring interferences during execution, compared to *is*RA-LOCK.

c) Tuning timing variability (from 5% to 40%): To quantitatively characterize the behavior of the proposed approach, when other sources of timing variability occur on top of the interferences, we insert an average variability of 5%, 10%, 20% and 40% in the WCET of the tasks (WCRA remains unchanged). For all the experiments, we observe that as task variability is inserted, *is*RA-LOCK fails to take advantage of it during execution, due to its fixed partial order policy. On the other hand, as the variability is increased, the proposed approach provides higher gains. More precisely, we observe that with the configuration with two cores (which is the configuration with the minimum possible interferences), the gains of *is*RA-DYN are significant compared to *is*RA-LOCK. In particular, with an increasing timing variability of 5%, the average gains are increased to 1.125% for DCT, 0.935% for MERGE, and 0.490% for FFT (with an average of 0.85% for all benchmarks). Tables 5, 4 and 6 show that with timing variability increasing, the gains are increased. Considering all

Table 4 MERGE: Average gains (%) per core (C) and among all cores (A).

MERGE									
Timing	**2 cores**			**4 cores**					
Var. (%)	**CO**	**C1**	**A**	**CO**	**C1**	**C2**	**C3**	**A**	
0	0.08	0.08	0.08	1.90	6.17	1.81	1.84	2.930	
5	1.03	0.84	0.935	2.10	7.40	2.46	2.56	3.630	
10	2.29	2.09	2.190	4.04	9.37	4.48	4.63	5.630	
20	4.84	4.76	4.800	9.59	13.06	9.59	10.09	10.583	
40	9.35	9.29	9.32	16.80	19.85	17.96	17.50	18.028	
Timing	**8 cores**								
Var. (%)	**CO**	**C1**	**C2**	**C3**	**C4**	**C5**	**C6**	**C7**	**A**
0	1.96	5.42	1.64	5.30	2.06	5.37	1.69	5.13	3.571
5	4.58	8.01	3.62	7.84	4.56	8.07	3.76	8.22	6.083
10	7.78	11.43	7.19	11.15	7.98	11.60	6.40	11.52	9.381
20	14.90	17.27	13.70	17.69	14.95	13.40	15.25	17.77	15.616
40	24.49	24.92	23.04	26.11	25.31	25.17	20.91	24.78	24.341

Table 5 DCT: Average gains (%) per core (C) and among all cores (A)

DCT									
Timing	**2 cores**			**4 cores**					
Var. (%)	**C0**	**C1**	**A**	**C0**	**C1**	**C2**	**C3**	**A**	
0	0.14	0.13	0.135	0.46	0.41	0.40	0.41	0.42	
5	1.10	1.15	1.125	1.51	1.69	1.56	1.66	1.605	
10	2.56	2.63	2.595	3.67	4.95	4.04	3.83	4.123	
20	5.94	5.58	5.760	6.28	10.77	7.95	7.61	8.153	
40	11.35	10.48	10.915	13.85	17.23	16.48	14.54	15.525	
Timing	**8 cores**								
Var. (%)	**C0**	**C1**	**C2**	**C3**	**C4**	**C5**	**C6**	**C7**	**A**
0	0.36	0.44	0.33	0.42	0.36	0.46	0.42	0.40	0.398
5	0.64	1.16	0.75	1.06	0.70	1.06	0.71	1.05	0.891
10	3.77	6.17	4.37	5.78	3.87	5.73	5.76	5.49	5.118
20	11.68	14.56	11.90	14.21	12.21	14.51	13.08	13.86	13.251
40	21.30	23.49	21.87	23.44	21.46	23.64	21.78	21.33	22.288

benchmarks, we observe an average gains of 2.09% (10%), 4.95% (20%), and 9.33% (40%). The maximum gain for 40% variability is 11.35% observed for C0 running DCT. As the number of cores is increased, the gains are also increased. This occurs due to the fact that the proposed approach is able to take advantage of both the inserted timing variability and the occurring interferences. When four cores are used, the average gain over all benchmarks is 1.89%, 3.82%, 8.46% and 15.86%, for 5%, 10%, 30% and 40% variability, respectively. The maximum gain for 40% variability is 19.85% observed for C1 running MERGE. When eight cores are used, the gains are even higher, i.e., with an average gain over all benchmarks equal to 3.45%, 7.11%, 14.22% and 23.26%, for 5%, 10%, 30% and 40% variability, respectively. The maximum gain for 40% variability is 25.31% observed for C4 running MERGE.

5.3 Controller cost

Table 7 depicts the corresponding WCET values for the *is*RA-DYN and *is*RA-LOCK approaches. Due to the additional relax phase, the overhead of the *is*RA-DYN controller is higher than *is*RA-LOCK controller. Despite the increased overhead, *is*RA-DYN can provide further performance improvements, as it has been shown in the previous paragraphs.

6 Related Work

The run-time mechanisms are categorized whether: i) the considered tasks are *only time-critical* or *also best-effort*, ii) the WCET is *pessimistic* or *interference-sensitive*, and iii) the adaptation is *static* or *dynamic*. A detailed survey of the state-of-the-art is available in [4].

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 are typically based on static decisions, i.e., the execution of a new task can start as soon as a task finishes earlier than its pessimistic WCET. Typical examples of such approaches come from scheduling theory, e.g. [1, 5]. 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

Table 6 FFT: Average gains (%) per core (C) and among all cores (A).

FFT									
Timing	**2 cores**			**4 cores**					
Var. (%)	**C0**	**C1**	**A**	**C0**	**C1**	**C2**	**C3**	**A**	
0	0.22	0.22	0.220	0.51	0.39	0.42	0.38	0.425	
5	0.50	0.48	0.490	0.40	0.40	0.44	0.36	0.400	
10	1.49	1.48	1.485	1.67	1.78	1.67	1.68	1.700	
20	3.75	4.80	4.275	6.70	7.24	7.10	6.75	6.948	
40	7.45	8.05	7.750	13.76	14.81	13.61	14.00	14.045	
Timing	**8 cores**								
Var. (%)	**C0**	**C1**	**C2**	**C3**	**C4**	**C5**	**C6**	**C7**	**A**
0	0.68	1.69	0.85	0.88	0.82	0.98	0.73	0.90	0.942
5	3.04	4.15	3.04	3.58	3.22	3.56	3.03	3.31	3.366
10	6.44	7.38	6.40	7.09	6.53	7.53	6.35	6.99	6.839
20	13.33	14.23	13.45	14.28	13.65	14.41	13.02	14.00	13.797
40	22.79	23.59	22.90	23.64	22.97	24.65	22.53	22.23	23.163

Table 7 Control phases WCET.

	***is*RA-LOCK WCET**		***is*RA-DYN WCET**	
	Fixed Cost	**Cost/Edge**	**Fixed Cost**	**Cost/Edge**
$C_{\mathcal{R}_\tau}$	355	251	355	251
$C_{\mathcal{N}_\tau}$	260	263	260	263
$C_{\mathcal{S}_\tau}$	-	-	183	201
$C_{TA_\mathcal{R}}$	45	-	45	-

analysis and provide *interference-sensitive* WCETs, such as [17, 19, 24]. In general, these approaches provide improved timing guarantees, since they compute 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. [17], or the precise timing of the requests, e.g. [20], or even provide contention-free schedules, e.g. [19]; a detailed survey of such approaches can be found in [12]. To further reduce the impact of the inherent pessimism in any kind of WCET estimations, several run-time mechanisms have been proposed. In [21, 22, 24], the authors provide a run-time approach suitable for interference-sensitive WCET. However, these approaches act upon static decisions, being unable to modify the partial order of tasks, created offline during the interference-sensitive scheduling. In contrast, the proposed *is*RA-DYN approach embraces dynamic decisions allowing safe modification of the *is*WCET scheduling, leading to performance improvements.

The run-time mechanisms for time-critical and best-effort tasks assume the timely execution of time-critical tasks, when they run in isolation. Based on this assumption, they decide the best-effort tasks execution, so as to still guarantee the timely execution of time-critical tasks. Such approaches use different confidence levels in the WCET estimation [3], compute the remaining WCET in isolation for the time-critical tasks [9–11], use resource usage capacities [15, 16] and partitioning of the memory accesses [27]. The *is*RA-DYN approach is orthogonal to these approaches, since it focuses on providing timing guarantees for the time-critical tasks.

7 Conclusion

In this work, we propose a dynamic interference-sensitive run-time adaptation technique *is*RA-DYN that alleviates the limitations of the existing *is*RA-LOCK, since it allows to safely relax the partial order of *is*WCET schedule solutions, whenever this is possible. We have presented the corresponding RTA for our technique and have formally argued regarding its safety, under any execution. The obtained results show that *is*RA-DYN outperforms *is*RA-LOCK as it can exploit variability in actual execution of tasks. When using two cores, the interferences are few and without any variability, the *is*RA-DYN provides small gains. However, with increasing variability, even with under few interferences, *is*RA-DYN provide significant gains. As the number of cores is increased, *is*RA-DYN provides better gains.

References

1 Marko Bertogna. *Real-time scheduling analysis for multiprocessor platforms.* PhD thesis, Scuola Superiore Sant'Anna, Pisa, 2008.

2 Alessandro Biondi and Marco Di Natale. Achieving predictable multicore execution of automotive applications using the let paradigm. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 240–250. IEEE, April 2018. doi:10.1109/RTAS.2018.00032.

3 Alan Burns and Sanjoy K. Baruah. Timing faults and mixed criticality systems. In CliffB. Jones and JohnL. Lloyd, editors, *Dependable and Historic Computing*, volume 6875 of *Lecture Notes in Computer Science*, pages 147–166. Springer Berlin Heidelberg, 2011. doi:10.1007/978-3-642-24541-1_12.

4 Alan Burns and Robert I. Davis. A survey of research into mixed criticality systems. *ACM Comput. Surv. (CSUR)*, 50(6):82:1–82:37, 2018. doi:10.1145/3131347.

5 Robert I. Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv. (CSUR)*, 43(4):35, 2011. doi:10.1145/1978802.1978814.

6 Jean-François Deverge and Isabelle Puaut. Safe measurement-based WCET estimation. In Reinhard Wilhelm, editor, *5th International Workshop on Worst-Case Execution Time Analysis (WCET'05)*, volume 1 of *OpenAccess Series in Informatics (OASIcs)*, Dagstuhl, Germany, 2007. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/OASIcs.WCET.2005.808.

7 Alexandre Esper, Geoffrey Nelissen, Vincent Nélis, and Eduardo Tovar. An industrial view on the common academic understanding of mixed-criticality systems. *Real-Time Systems*, 54(3):745–795, 2018.

8 Namhoon Kim, Bryan C. Ward, Micaiah Chisholm, Cheng-Yang Fu, James H. Anderson, and F. Donelson 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)*, pages 1–12. IEEE, 2016.

9 Angeliki Kritikakou, Olivier Baldellon, Claire Pagetti, Christine Rochange, and Matthieu Roy. Run-time control to increase task parallelism in mixed-critical systems. In *ECRTS*, 2014.

10 Angeliki Kritikakou, Thibaut Marty, and Matthieu Roy. DYNASCORE: dynamic software controller to increase resource utilization in mixed-critical systems. *ACM Trans. Design Autom. Electr. Syst. (TODAES)*, 23(2):13:1–13:26, 2018. doi:10.1145/3110222.

11 Angeliki Kritikakou, Christine Rochange, Madeleine Faugère, Claire Pagetti, Matthieu Roy, Sylvain Girbal, and Daniel Gracia Pérez. Distributed run-time WCET controller for concurrent critical tasks in mixed-critical systems. In *International Conference on Real-Time Networks and Systems (RTNS)*, pages 139–148. ACM, 2014. doi:10.1145/2659787.2659799.

12 Claire Maiza, Hamza Rihani, Juan M Rivas, Joël Goossens, Sebastian Altmeyer, and Robert I Davis. A survey of timing verification techniques for multi-core real-time systems. *ACM Computing Surveys (CSUR)*, 52(3):56, 2019.

13 Sébastien Martinez, Damien Hardy, and Isabelle Puaut. Quantifying wcet reduction of parallel applications by introducing slack time to limit resource contention. In *Proceedings of the 25th International Conference on Real-Time Networks and Systems*, pages 188–197. ACM, 2017.

14 C. Moreno and S. Fischmeister. Accurate measurement of small execution times—getting around measurement errors. *IEEE Embedded Systems Letters*, 9(1):17–20, March 2017.

15 Jan Nowotsch and Michael Paulitsch. Quality of service capabilities for hard real-time applications on multi-core processors. In *International Conference on Real-Time Networks and Systems (RTNS)*, pages 151–160. ACM, 2013. doi:10.1145/2516821.2516826.

16 Jan Nowotsch, Michael Paulitsch, Daniel Bühler, Henrik Theiling, Simon Wegener, and Michael Schmidt. Multi-core interference-sensitive wcet analysis leveraging runtime resource capacity enforcement. In *Euromicro Conference on Real-Time Systems (ECRTS),*, pages 109–118. IEEE, 2014.

17 Hamza Rihani, Matthieu Moy, Claire Maiza, Robert I. Davis, and Sebastian 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*, RTNS '16, pages 67–76, New York, NY, USA, 2016. ACM. doi:10.1145/2997465.2997472.

18 Benjamin Rouxel, Steven Derrien, and Isabelle Puaut. Tightening contention delays while scheduling parallel applications on multi-core architectures. *ACM Trans. Embed. Comput. Syst. (TECS)*, 16(5s):164:1–164:20, September 2017. doi:10.1145/3126496.

19 Benjamin Rouxel, Stefanos Skalistis, Steven Derrien, and Isabelle 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.

20 Andreas Schranzhofer, Jian-Jia Chen, and Lothar Thiele. Timing analysis for TDMA arbitration in resource sharing systems. In *2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 215–224. IEEE, 2010.

21 Stefanos Skalistis, Federico Angiolini, Alena Simalatsar, and Giovanni De Micheli. Safe and efficient deployment of data-parallelizable applications on many-core platforms: Theory and practice. *IEEE Design & Test*, 35(4):7–15, 2018.

22 Stefanos Skalistis and Angeliki Kritikakou. Timely fine-grained interference-sensitive run-time adaptation of time-triggered schedules. In *Real-Time Systems Symposium (RTSS)*. IEEE, 2019.

23 Stefanos Skalistis and Alena Simalatsar. Worst-case execution time analysis for many-core architectures with NoC. In *International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS)*, pages 211–227. Springer, 2016.

24 Stefanos Skalistis and Alena Simalatsar. Near-optimal deployment of dataflow applications on many-core platforms with real-time guarantees. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 752–757. IEEE, 2017.

25 Texas Instruments. TMS320C6678 Multicore fixed and floating-point digital signal processor. Technical Report SPRS691D, TI, 2013.

26 William Thies and Saman 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)*, pages 365–376. ACM, 2010.

27 Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 55–64, 2013.