Multi-core interference over-estimation reduction by static scheduling of multi-phase tasks

Rémi Meunier, Thomas Carle, Thierry Monteil

▶ **To cite this version:**

Rémi Meunier, Thomas Carle, Thierry Monteil. Multi-core interference over-estimation reduction by static scheduling of multi-phase tasks. Real-Time Systems, 2024, pp.1–39. 10.1007/s11241-024-09427-3 . hal-04689317

HAL Id: hal-04689317

https://hal.science/hal-04689317v1

Submitted on 5 Sep 2024

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

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

Distributed under a Creative Commons Attribution 4.0 International License

Real-Time Systems
https://doi.org/10.1007/s11241-024-09427-3

Multi-core interference over-estimation reduction by static scheduling of multi-phase tasks

Rémi Meunier[1] · **Thomas Carle**[2] · **Thierry Monteil**[3]

Accepted: 5 August 2024
© The Author(s) 2024

Abstract

Interference between tasks running on separate cores in multi-core processors is a major challenge to predictability for real-time systems, and a source of over-estimation of worst-case execution duration bounds. This paper investigates how the multi-phase task model can be used together with static scheduling algorithms to improve the precision of the interference analysis. The paper focuses on single-period task systems (or multi-periodic systems that can be expanded over an hyper-period). In particular, we propose an Integer Linear Programming (ILP) formulation of a generic scheduling problem as well as 3 heuristics that we evaluate on synthetic benchmarks and on 2 realistic applications. We observe that, compared to the classical 1-phase model, the multi-phase model allows to reduce the effect of interference on the worst-case makespan of the system by around 9% on average using the ILP on small systems, and up to 24% on our larger case studies. These results pave the way for future heuristics and for the adoption of the multi-phase model in multi-core context.

Keywords Multi-core · WCET · Interference · Static scheduling

✉ Thomas Carle
thomas.carle@irit.fr

Rémi Meunier
meunierremi85@gmail.com

Thierry Monteil
thierry.monteil@irit.fr

[1] Randstadt Digital, Toulouse, France

[2] Univ. Toulouse 3 - IRIT, Toulouse, France

[3] INSA - IRIT, Toulouse, France

Published online: 05 September 2024

1 Introduction

Multi-core processors raise several challenges for the design, implementation and validation of critical real-time systems. One serious issue regards the ability to derive safe yet precise Worst-Case Execution Time (WCET) bounds for each of the tasks composing a system. Indeed, traditional WCET analysis for single core processors (AbsInt 2022; Ballabriga et al. 2010) works separately on each task (i.e. in isolation) and then composes the bounds of all tasks using a Worst-Case Response Time analysis (WCRT) in which interference between tasks is taken into account. On single core targets, this interference is usually caused by task preemption, and is dealt with by computing a penalty for saving/restoring the values of the registers, a Cache-Related Preemption Delay (CRPD) and the WCET of the interrupt handling routine, that are added to the preempted task WCET bound. The relative simplicity of this approach makes it appealing, however additional obstacles must be overcome in order to extend it to multi-core processors. Indeed, multi-cores are capable of executing multiple threads in parallel but share some hardware components, in particular the memory that the threads use to communicate with one another. These shared components generate additional interference that is hard to characterize precisely at a high level of abstraction. It results that applying WCRT analysis for systems running on multi-cores with the traditional task abstraction (one task represented by its WCET, and additionally its worst-case number of memory accesses) leads to huge timing over-estimations.

In order to deal with these issues, finer-grained abstractions have been developed that model the tasks execution as a sequence of so-called phases. The first of these abstractions modelled tasks using 2 or 3 phases (Pellizzoni et al. 2011; Durrieu et al. 2014; Rouxel et al. 2019; Jati et al. 2021): some phases correspond to memory copies between the shared memory and a memory local to a core, and others to the purely local execution of the task. Using this abstraction it is possible to build interference-free schedules or to account more precisely for the interference in the system. However, this model strongly restricts the way the code must be written or compiled in order to fit the phases. More recently, in an attempt to generalize the model and to uplift these restrictions to manage legacy code, a multi-phase model was introduced (Carle and Cassé 2021; Degioanni and Puaut 2022). In this model, each task is represented by a sequence composed of an arbitrary number of phases that may or may not perform memory accesses. This model accounts for the time windows in which each memory access can occur, and thus can cause or be subject to interference with accesses from other tasks. The abstraction (i.e. the phases) is mapped to the actual code of the tasks using synchronizations, but does not require the task to be compiled in a particular fashion.

The objective of this paper is to leverage the multi-phase model using dedicated scheduling algorithms in order to characterize more precisely, and if possible reduce, the effects of interference in the shared memory. We focus on non-preemptive static scheduling algorithms for single-periodic task systems[1] and restrict our interference analysis to accesses to the shared memory through a first-come first-served (FCFS) sequential bus.

In this paper, we make the following contributions:

[1] Although these methods also apply to multi-periodic systems expanded over an hyper-period.

Table 1 List of abbreviations

WCET	Worst-case execution time
WCRT	Worst-case response time
CRPD	Cache-related Preeption delays
FCFS	First-come first-served
ILP	Integer linear programming
ASAP	As soon as possible
DAG	Directed acyclic graph
mksp	Makespan
SDE	Starting date enumeration
IPH	Iterative priority scheduling
LB	Lower bound
UB	Upper bound
N	Normal distribution
U	Uniform distribution
BN	Bi-Normal distribution
pos	positive
av	Average
std dev	Standard deviation
UAV	Unmanned aerial vehicle

- We provide an Integer Linear Programming (ILP) formulation of a generic multi-core scheduling problem, using the multi-phase model.
- We describe 2 greedy heuristics based on list scheduling: one using the As Soon As Possible (ASAP) policy, and one that more aggressively exploits the multi-phase model.
- We introduce a more complex heuristic that adapts the method introduced in Hanzálek and Šůcha (2017) to the multi-phase model.
- We conduct an evaluation campaign, both on synthetic benchmarks and on two case-study applications: Pagetti et al. (2014) and Nemer et al. (2006).

The paper is organized as follows: Sect. 2 presents the related work and Sect. 3 introduces the formal definitions. Then, Sect. 4 describes multi-core scheduling techniques adapted to the multi-phase model. Section 5 details an evaluation of our scheduling techniques, followed by a conclusion in Sect. 6. Table 1 summarizes the abbreviations that are used throughout the paper.

2 Related work

2.1 Multi-core interference and the multi-phase model

Interference on multi-cores has been a hot topic of research in the real-time community for years, and although multiple models and methods have been developed, none of them seems to be entirely satisfying (Maiza et al. 2019).

This paper focuses on the multi-phase model of tasks, which has been formalized in Rémi et al. (2022) and can be seen as a generalization of the PREM (Pellizzoni et al. 2011) or AER/REW models (Durrieu et al. 2014; Rouxel et al. 2019). In the PREM and AER models, tasks are represented as sequences of phases that alternate between phases that may access the memory, and phases that never access the memory. In order to obtain tasks that fit this representation, dedicated compilation techniques must be applied, which precludes the use of legacy code. Then the objective is to schedule the phases of the tasks so that no interference occurs in the system. On the contrary, the phase sequence in the multi-phase model follows the behavior of the code (whether it is legacy code, or code that has been specifically compiled like for PREM and AER). Our objective in this work is not to avoid interference altogether, but rather to tolerate a certain level of interference in the system, and to show that by leveraging the precision of the multi-phase model, it is possible to limit the amount of accounted interference in the analysis.

Two main approaches have been developed in order to obtain multi-phase representations of tasks. The Time Interest Points (TIPs) approach (Carle and Cassé 2021) has been developed to reconcile the multi-phased task model with legacy code. In this approach, a multi-phase representation of the tasks is obtained through static analysis of the binary code of the tasks. As a consequence, no restrictions are imposed on how the source code must be written or compiled. More recently, StAMP (Degioanni and Puaut 2022) introduced an approach based on single-entry single-exit (SESE) sections of code to determine the boundaries of phases at compile time. Then, static analysis is performed to obtain the worst-case duration and number of accesses of each phase.

Using the multi-phase model, an interference analysis can be performed as part of a WCRT analysis such as the one in Davis et al. (2018), or as part of a static scheduling/compiling approach (e.g. de Dinechin et al. 2020; Didier et al. 2019). Moreover, the analysis can be tuned in order to produce different representations of the same task (Carle and Cassé 2021), with e.g. a different number of phases.

2.2 Interference-aware scheduling on multi-core processors

Multiple methods construct contention-free schedules using ILP and a PREM or AER representation of tasks (Becker et al. 2016; Pagetti et al. 2018; Matějka et al. 2019; Senoussaoui et al. 2022). Iterative heuristics are also introduced in Senoussaoui et al. (2022) and Matějka et al. (2019). A heuristic for PREM tasks that are already partitioned to the cores is proposed in Senoussaoui et al. (2022). Matějka et al. (2019), the authors transform AER/REW tasks into take-give activities to apply a heuristic from Hanzálek and Šůcha (2017). This heuristic assigns priorities to the tasks of the system and builds a schedule accordingly. The heuristic then iterates, adapting some priorities at each iteration, until it converges towards the best solution. Schuh et al. (2020) compare different PREM and AER scheduling scenarios with or without interference and show that tolerating interference usually results in a lower WCRT/makespan than suppressing it altogether.

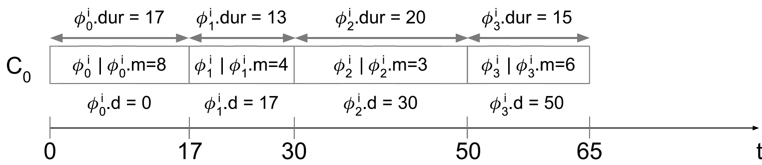


Fig. 1 Example of a task profile before interference analysis, scheduled on core C_0

3 Formal model

This section formally describes the multi-phase model following the definitions of Rémi et al. (2022).

3.1 The multi-phase task model

We model a system T of real-time tasks τ^i $(i \geq 0)$. Each task τ^i is represented by its *profile*, which is a sequence of phases denoted $\mathbb{P}^i = \{\phi^i_l | 0 \leq l < \Phi^i\}$ with Φ^i the number of phases. A phase ϕ^i_j is defined by:

- $\phi^i_l.d$: its start time.
- $\phi^i_l.dur$: its worst-case duration in isolation (i.e. without interference).
- $\phi^i_l.m$: the worst-case number of memory accesses that may be performed within $[\phi^i_l.d, \phi^i_l.d + \phi^i_l.dur[$.

The start time of the task is determined during the construction of the static schedule of the system and corresponds to the start time of ϕ^i_0. Then, for each ϕ^i_l $(l > 0)$ the start time is defined by:

$$\phi^i_l.d = \phi^i_0.d + \sum_{0 \leq q < l} \phi^i_q.dur = \phi^i_{l-1}.d + \phi^i_{l-1}.dur \tag{1}$$

Note that this model only reflects the execution behavior and memory access profile of the task code, not the classical real-time attributes of tasks (e.g. period, deadline). As such, the model applies naturally to systems in which all tasks have the same period: the start time of a task represents its start time relatively to the start of each activation of the system. For multi-periodic task systems, this model applies to task instances (a.k.a. jobs). In the remainder of the paper, the term *task* is used indiscriminately to describe either a task in single period systems or a job in multi-periodic systems (and in particular, two different jobs are modeled as two different tasks). In the multi-periodic case, we consider all task instances that are released over an hyperperiod as separate tasks. Task deadlines are not explicitly considered in the model: as stated in Sect. 4.1, the considered scheduling problem amounts to minimizing the makespan of the task system. Then, if this makespan is small enough to fit the task deadlines, the system is deemed schedulable. This is a classical problem for critical control systems specified e.g. with synchronous languages (Halbwachs

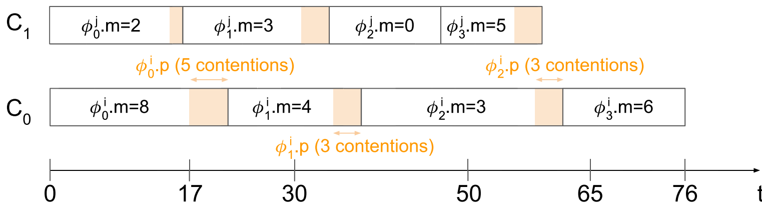


Fig. 2 Two tasks τ^i and τ^j scheduled in parallel on cores C_0 and C_1 and the penalties resulting from the interference analysis (τ^i is the same as in Fig. 1)

1992; Carle et al. 2015). For other multi-periodic systems, the proposed scheduling heuristics can be applied locally to reduce the makespan of groups of jobs, and adapted to use the deadlines to select the priority order in which tasks/jobs are selected to be scheduled. However, the main objective of the paper is not to present scheduling techniques that directly target the respect of task deadlines, but rather to demonstrate the capacity of the multi-phase model to mitigate the amount of interference accounted for during the system construction and timing validation.

Figure 1 shows an example of profile for a task τ^i with 4 phases.

3.2 Consequences of the interference analysis

Once a task system has been either partially or entirely statically scheduled by setting a $\phi^i_l.d$ for each phase of each scheduled task, an interference analysis such as the one described in de Dinechin et al. (2020) is applied to bound the effects of inter-core contentions in the system. This analysis upper bounds the worst-case effect of the architectural elements where interference may occur. In the scope of this paper only shared memory buses and/or simple memory controllers that serve requests in a FCFS/Round-Robin fashion are considered (in particular, cache coherence mechanisms and shared cache effects are not considered). This way, each memory access initiated by a core can be interfered by at most one request of each other core. As a result, each phase that may suffer from contentions is extended to take into account the potential additional execution delay and the next phases are postponed accordingly. In order to express these modifications, new attributes are introduced to the formal model of phases:

- $\phi^i_l.p \geq 0$ is the timing penalty added to ϕ^i_l to account for potential contentions.
- $\phi^i_l.d^\#$ is the *interference-aware* date of ϕ^i_l, i.e. its start date taking into account the potential contentions in the system.

Note that the computed interference delays for a given task τ^i may also postpone the starting date $\phi^j_0.d^\#$ of some other tasks, because τ^i occupies its core for a longer time, or because of a data dependency between τ^i and τ^j. The (post interference analysis) start dates of the other phases ϕ^i_l are given by:

Table 2 Summary of notations

Notation	Definition
G	DAG defining task dependencies
E	Set of edges (dependency relations) of G
$preds(\tau^i)$	Set of predecessors of τ^i
$succs(\tau^i)$	Set of successors of τ^i
$\mathbb{C}$	Set of cores composing the architecture
C_k	Core with index k
$\mathbb{S}$	Schedule
$\mathbb{S}(C_k)$	Schedule of core C_k
$\mathbb{S}(C_k).end$	End date of schedule of core C_k
ϕ^i_j	Phase j of task τ^i
$\phi^i_j.d$	Start date of phase j of task τ^i, before interference analysis
$\phi^i_j.dur$	Duration of phase j of task τ^i, before interference analysis
$\phi^i_j.d^\#$	Start date of phase j of task τ^i, after interference analysis
$\phi^i_j.p$	Timing penalty added to phase ϕ^i_j to bound the interference
$\phi^i_{\Phi^i}.d^\#$	End date of τ^i in the presence of interference
$\phi^i_j.\gamma$	Number of potential contentions suffered by ϕ^i_j
$\phi^i_j.\gamma_k$	Number of potential contentions suffered by ϕ^i_j from C_k
ω^i_k	True if τ^i is mapped to C_k
ρ^i_j	True if τ^i and τ^j are mapped to the same core
$\chi^{i,j}_{k,l}$	True if the intervals covered by ϕ^i_j and ϕ^k_l overlap
$\theta_{i,j_k,l}$	True if ϕ^i_j starts before the end of ϕ^k_l

$$\phi^i_l.d^\# = \phi^i_0.d^\# + \sum_{0\leq q<l}(\phi^i_q.dur + \phi^i_q.p) = \phi^i_{l-1}.d^\# + \phi^i_{l-1}.dur + \phi^i_{l-1}.p \tag{2}$$

An important point to note here is that the post interference analysis start times of the phases must be respected in the implementation, even when previous phases end earlier than their (extended) worst-case duration, otherwise the assumptions on which phases may run in parallel would not be respected, and the analysis may under-estimate the amount of interference in the system. This limits the implementation of the proposed methods to Time-Triggered solutions. In the same fashion, preemptions are not considered in the scope of this paper. Supporting them is possible, but requires the adaptation of Cache-Related Preemption Delays analyses to the multi-phase model, which has not been done yet.

Figure 2 represents two tasks scheduled on 2 different cores with the results of the interference analysis (task τ^i has the same profile as in Fig. 1). The orange rectangles represent the penalty $\phi^i_l.p$ that extends the worst-case duration of the phases. For example, ϕ^i_0 is scheduled in parallel with two phases ϕ^j_0 and ϕ^j_1 that perform respectively 2 and 3 accesses in the worst case. ϕ^i_0 performs 8 accesses in the worst case. As a result, the interference analysis considers that in the worst case, 5 accesses of

ϕ_0^i are interfered. In the same fashion, the 2 accesses of ϕ_0^j (resp. the 3 accesses of ϕ_1^j) are interfered in the worst case.

4 Multi-core scheduling

This section discusses several approaches to benefit from the multi-phase representation of tasks when scheduling tasks on multi-core platforms. Our main objective is to minimize the makespan of the task system in the presence of interference.

4.1 Problem definition

The following static scheduling problem is targeted: given a set of homogeneous cores[2] connected to a shared memory through a FCFS bus and a system composed of data-dependent tasks specified as a directed acyclic graph (DAG), schedule the tasks on the cores in order to minimize the interference-aware makespan of the system. This problem instance considers non-preemptive tasks only, and tasks are not partitioned to the cores prior to the scheduling phase . The notations that compose the model are summarized in Table 2.

Formally, let $\mathbb{C} = \{C_k | 0 \leq k < N_c\}$ be a multi-core architecture composed of N_c cores. Dependencies between the tasks of T are specified using a DAG $G = (T, E)$ in which vertices are the tasks of T and each edge $e_{i,j} \in E$ between τ^i and τ^j indicates that τ^i must be completed before τ^j can start. Moreover, $preds(\tau^i) = \{\tau^k | e_{k,i} \in E\}$ denotes the set of predecessors of τ^i and $succs(\tau^i) = \{\tau^k | e_{i,k} \in E\}$ the set of successors of τ^i.

Our objective is to build a schedule $\mathbb{S}$ of the tasks of T on the cores composing $\mathbb{C}$.

For each core C_k, the following attributes in $\mathbb{S}$ are defined:

- $\mathbb{S}(C_k)$: the schedule of C_k, which is a sequence of phases, ordered by their starting date.
- $\mathbb{S}(C_k).end$: the end date of the last phase scheduled on C_k.

The makespan of the task system in schedule $\mathbb{S}$ is $mksp(\mathbb{S}) = max_{C_k \in \mathbb{C}}(\mathbb{S}(C_k).end)$.

4.2 ILP formulation

We now provide an ILP formulation of the problem. In this formulation we use bold font to denote the variables of the ILP system, ILP.1 is the objective function and the other equations numbered ILP.X are the constraints.

We first introduce variable *mksp* denoting the makespan of the task system. It appears in the objective function that minimizes the makespan:

[2] The proposed heuristics also apply to heterogeneous cores but experiments have not been conducted in this setting.

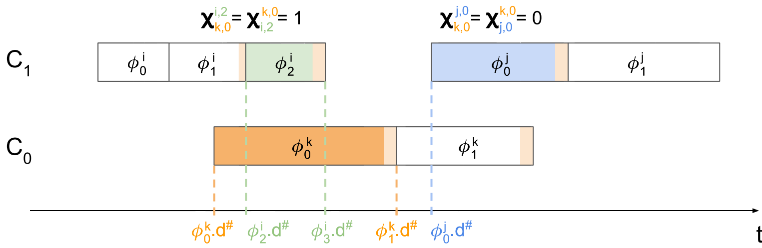


Fig. 3 Three tasks scheduled on 2 cores

$$\textit{minimize}\ \boldsymbol{mksp} \tag{ILP.1}$$

We use $\boldsymbol{\phi^i_{\Phi^i}.d^\#}$ to denote the end time of τ^i, which is the end date of its last phase:

$$\forall \tau^i, \boldsymbol{\phi^i_{\Phi^i}.d^\#} = \boldsymbol{\phi^i_{\Phi^i-1}.d^\#} + \phi^i_{\Phi^i-1}.dur + \boldsymbol{\phi^i_{\Phi^i-1}.p} \tag{ILP.2}$$

The makespan of the system is greater than the end time of all tasks:

$$\forall \tau^i, \boldsymbol{mksp} \geq \boldsymbol{\phi^i_{\Phi^i}.d^\#} \tag{ILP.3}$$

Moreover, each task τ^i starts after date 0 and after the end of all its predecessors:

$$\forall \tau^i, \boldsymbol{\phi^i_0.d^\#} \geq 0 \tag{ILP.4}$$

$$\forall \tau^k \in preds(\tau^i), \boldsymbol{\phi^i_0.d^\#} \geq \boldsymbol{\phi^k_{\Phi^k}.d^\#} \tag{ILP.5}$$

Following the definition of the start time of a phase in Eq. (2), we can express the date of each subsequent phase as:

$$\forall \tau^i, \forall 0 < j \leq \Phi^i, \boldsymbol{\phi^i_j.d^\#} = \boldsymbol{\phi^i_{j-1}.d^\#} + \phi^i_{j-1}.dur + \boldsymbol{\phi^i_{j-1}.p} \tag{ILP.6}$$

We use boolean variable $\boldsymbol{\omega^i_k}$ to express the mapping of task τ^i: $\boldsymbol{\omega^i_k} = 1$ if and only if τ^i is mapped on C_k. Each task is mapped to a unique core so we add the constraints:

$$\forall \tau^i : \sum_{0 \leq k < N_c} \boldsymbol{\omega^i_k} = 1 \tag{ILP.7}$$

We also introduce variable $\boldsymbol{\rho^i_j}$ that is equal to 1 if and only if τ^i and τ^j are mapped to the same core:

$$\forall \tau^i, \tau^j, \boldsymbol{\rho^i_j} = \sum_{0 \leq k < N_c} \boldsymbol{\omega^i_k} \wedge \boldsymbol{\omega^j_k}$$

Because of the conjunction $\wedge$, the above equation is not linear. Therefore, we have to use a new variable $\boldsymbol{\Omega^{i,j}_k} = \boldsymbol{\omega^i_k} \wedge \boldsymbol{\omega^j_k}$ and add the following equations:

$$\forall \tau^i, \tau^j, 0 \le k < N_c,$$

$$\boldsymbol{\Omega^{i,j}_k} \le \boldsymbol{\omega^i_k} \tag{ILP.8}$$

$$\boldsymbol{\Omega^{i,j}_k} \le \boldsymbol{\omega^j_k} \tag{ILP.9}$$

$$\boldsymbol{\Omega^{i,j}_k} + 1 \ge \boldsymbol{\omega^i_k} + \boldsymbol{\omega^j_k} \tag{ILP.10}$$

Therefore, the equation becomes:

$$\boldsymbol{\rho^i_j} = \sum_{0 \le k < N_c} \boldsymbol{\Omega^{i,j}_k} \tag{ILP.11}$$

In the following, any other conjunction will be converted to a linear form in the same way. For clarity reasons, we do not provide the details for the other linearizations of conjunctions.

Two phases may contend with each other if they are scheduled on different cores and their execution intervals overlap. We introduce the boolean variable $\boldsymbol{\chi^{i,j}_{k,l}}$ that is true if the intervals covered by ϕ^i_j and ϕ^k_l overlap:

$$\boldsymbol{\chi^{i,j}_{k,l}} \Leftrightarrow \neg((\boldsymbol{\phi^k_{l+1}.d^\#} \le \boldsymbol{\phi^i_j.d^\#}) \vee (\boldsymbol{\phi^i_{j+1}.d^\#} \le \boldsymbol{\phi^k_l.d^\#}))$$
$$\boldsymbol{\chi^{i,j}_{k,l}} \Leftrightarrow (\boldsymbol{\phi^i_j.d^\#} < \boldsymbol{\phi^k_{l+1}.d^\#}) \wedge (\boldsymbol{\phi^k_l.d^\#} < \boldsymbol{\phi^i_{j+1}.d^\#})$$

The overlapping is illustrated by Fig. 3. Phase ϕ^k_0 overlaps with ϕ^i_2 but not with ϕ^j_0 so $\chi^{k,0}_{i,2} = \chi^{i,2}_{k,0} = 1$ and $\chi^{k,0}_{j,0} = \chi^{j,0}_{k,0} = 0$.

We need to decompose the equivalence relation into several constraints in the ILP system. That is why we define $\boldsymbol{\theta_{i,j_k,l}}$ as:

$$\boldsymbol{\theta_{i,j_k,l}} \Leftrightarrow \boldsymbol{\phi^i_j.d^\#} < \boldsymbol{\phi^k_{l+1}.d^\#}$$

so that the equivalence becomes:

$$\boldsymbol{\chi^{i,j}_{k,l}} \Leftrightarrow \boldsymbol{\theta_{i,j_k,l}} \wedge \boldsymbol{\theta_{k,l_i,j}} \tag{ILP.12}$$

$\boldsymbol{\theta_{i,j_k,l}}$ is defined using the big-M notation and a cancellation variable $\boldsymbol{\beta_{i,j_k,l}}$:

$$\forall \tau^i, \tau^j, 0 \le j < \Phi^i, 0 \le i < \Phi^k,$$

$$1 + \boldsymbol{\phi^i_j.d^\#} \le \boldsymbol{\phi^k_{l+1}.d^\#} + M(1 - \boldsymbol{\theta_{i,j_k,l}}) \tag{ILP.13}$$

$$\boldsymbol{\phi^i_j.d^\#} \ge \boldsymbol{\phi^k_{l+1}.d^\#} - M(1 - \boldsymbol{\beta_{i,j_k,l}}) \tag{ILP.14}$$

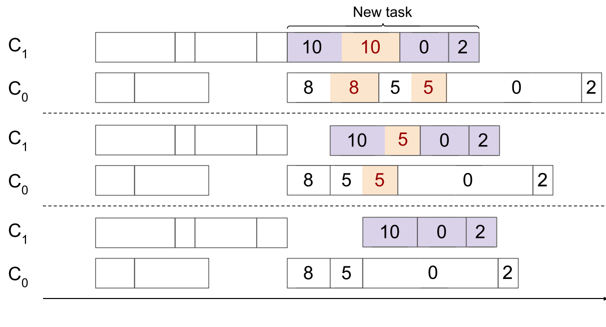


Fig. 4 3 different placements for a new task: the numbers within phases indicate their worst-case number of accesses and the orange rectangles are the additional penalty due to possible interference

$$\boldsymbol{\beta}_{i,j_k,l} + \boldsymbol{\theta}_{i,j_k,l} = 1 \tag{ILP.15}$$

The overlapping of 2 phases is forbidden if their tasks (resp. τ^i and τ^k) are scheduled on the same core ($\boldsymbol{\rho}^i_k = 1$). Therefore:

$$\boldsymbol{\chi}^{i,j}_{k,l} \leq 1 - \boldsymbol{\rho}^i_k \tag{ILP.16}$$

In order to compute the time penalty of ϕ^i_j, we multiply the number of contentions it may encounter ($\boldsymbol{\phi}^i_j.\boldsymbol{\gamma}$) by the cost of one penalty denoted *penalty_cost*, so we have:

$$\forall \tau^i, 0 \leq j < \Phi^i, \boldsymbol{\phi}^i_j.\boldsymbol{p} = \boldsymbol{\phi}^i_j.\boldsymbol{\gamma} \times penalty_cost \tag{ILP.17}$$

$\boldsymbol{\phi}^i_j.\boldsymbol{\gamma}$ is the sum of the contentions that may be caused by tasks on all the cores:

$$\forall \tau^i, 0 \leq j < \Phi^i, \boldsymbol{\phi}^i_j.\boldsymbol{\gamma} = \sum_{0 \leq k < N_c} \boldsymbol{\phi}^i_j.\boldsymbol{\gamma}_k \tag{ILP.18}$$

with $\boldsymbol{\phi}^i_j.\boldsymbol{\gamma}_k$ the number of contentions that ϕ^i_j may experience from tasks scheduled on core k. As we consider a shared memory bus following a FCFS policy, $\boldsymbol{\phi}^i_j.\boldsymbol{\gamma}_k$ is bounded by $\phi^i_j.m$:

$$\forall \tau^i, 0 \leq j < \Phi^i, 0 \leq k < N_c, \boldsymbol{\phi}^i_j.\boldsymbol{\gamma}_k = \min(\phi^i_j.m, \sum_{\tau^l \in T} \sum_{0 \leq q < \Phi^l} \phi^l_q.m \times (\boldsymbol{\chi}^{i,j}_{l,q} \wedge \boldsymbol{\omega}^l_k)) \tag{3}$$

The term $(\boldsymbol{\chi}^{i,j}_{l,q} \wedge \boldsymbol{\omega}^l_k)$ states that ϕ^i_j receives contentions from ϕ^l_q if and only if ϕ^l_q is mapped to core k and overlaps with ϕ^i_j.

Finally, to linearize the minimum operator, we use the following equations with $\boldsymbol{\alpha}^i_{j,k} \in \{0, 1\}$ guaranteeing that one of the proposed values is taken:

$$\forall \tau^i, \tau^j, 0 \leq j < \Phi^i, 0 \leq k < N_c,$$

$$\boldsymbol{\phi}^i_j.\boldsymbol{\gamma}_k \le \phi^i_j.m \tag{ILP.19}$$

$$\boldsymbol{\phi}^i_j.\boldsymbol{\gamma}_k \le \sum_{\tau^l \in T} \sum_{0 \le q < \Phi^l} \phi^i_j.m \times (\boldsymbol{\chi}^{i,j}_{l,q} \wedge \boldsymbol{\omega}^l_k) \tag{ILP.20}$$

$$\boldsymbol{\phi}^i_j.\boldsymbol{\gamma}_k \ge \left(\sum_{\tau^l \in T} \sum_{0 \le q < \Phi^l} \phi^i_j.m \times (\boldsymbol{\chi}^{i,j}_{l,q} \wedge \boldsymbol{\omega}^l_k) \right) - M \times \boldsymbol{\alpha}^i_{j,k} \tag{ILP.21}$$

$$\boldsymbol{\phi}^i_j.\boldsymbol{\gamma}_k \ge \phi^i_j.m - M(1 - \boldsymbol{\alpha}^i_{j,k}) \tag{ILP.22}$$

4.3 Heuristics

Computing the penalties directly in the optimization problem is inherently non-linear. As a consequence, we observe in practice that the ILP resolution time does not scale when the number of tasks or phases grows. In order to tackle large systems that cannot be handled by ILP, we designed heuristics. We present three of them in this section.

In the following algorithms, we use the *computeContentions*($\mathbb{S}$) function to compute the values of the $\phi^i_j.d^{\#}$ by applying the formulas from Eqs. (ILP.17) and (3) on each phase ϕ^i_j of $\mathbb{S}$.[3]

4.3.1 Greedy policies

The first scheduling policies that we present are two variants of list scheduling: the algorithm selects a task from a list of ready tasks, schedules it following the policy, updates the list of ready tasks, and iterates until all tasks have been scheduled.

As soon as possible scheduling (ASAP) The ASAP policy takes the current partial schedule (initially empty) and builds as many schedules as there are cores in $\mathbb{C}$ by selecting a task and scheduling it as soon as possible on each of the cores, without preemption and while respecting task dependencies. Once the ASAP date is determined for a core, all the phases of the task are scheduled according to Eq. (1). It then selects the partial schedule that has the lowest makespan and moves on to the next task. The interference analysis is performed only once all the tasks of the system have been scheduled. Consequently, this is the simplest and the fastest algorithm of all the presented heuristics.

Starting date enumeration (SDE) The ASAP strategy is not always the best choice to minimize the makespan in the presence of interference. For instance, Fig. 4 shows 3 different ways to schedule a new task (the purple one) on core C_1. At the top, when scheduling the task as soon as possible, the phase with 10 accesses overlaps with 2 other phases in parallel and creates in the worst case

[3] In our experiments, we use an efficient Python implementation of this function.

13 $(8+5)$ contentions on core 0 (depicted in orange). In the schedule below, we postponed the task start time to the end of the phase with 8 accesses so the 10-accesses phase may only create 5 contentions, and this choice yields a reduction of the makespan. In the last schedule at the bottom, the task is postponed even more, to the next starting date of a phase in parallel. This results in no interference at all, yielding the smallest makespan. Following that idea, we developed the *SDE* heuristic that attempts to schedule the current task at several dates on each of the cores and performs an interference analysis for each possibility before selecting the one that minimizes the makespan.

Algorithm 1 describes SDE. It takes as inputs the current task to schedule, τ^i, and the current partial schedule $\mathbb{S}$, on which an interference analysis has been performed. The enumeration of the possible start time for τ^i is limited to the interval $[\![minDate, maxDate]\!]$ in which minDate is the earliest possible start time of τ^i due to precedence constraints, and maxDate is the current makespan of the partial schedule. For each core C_k (Algorithm: 1-line: 4), function *parallelDates* extracts the start and end times of phases scheduled on the other cores that fall within the $[\![minDate, maxDate]\!]$ interval. Then, τ^i is iteratively scheduled at each of these dates on C_k in $\mathbb{S}$ (Algorithm: 1-line: 7), and only the result yielding the smallest makespan (after interference analysis) is kept (Algorithm: 1-line: 9). In the end, τ^i is scheduled on the core and at the date that yielded the best makespan.

SDE considers candidate dates only for the start of the first phase of the task. The method could be extended to consider these dates as possible start times for each of the phases, but the algorithmic complexity would increase accordingly, making the method impractical.

Algorithm 1 SDE

Require: τ^i ; $\mathbb{S}$
1: $minDate = max_{\tau^h \in preds(\tau^i)}(\phi^h_{\Phi^h}.d^{\#})$
2: $maxDate = mksp(\mathbb{S})$
3: $bestMakespan, bestSched = +\infty, \mathbb{S}$
4: **for** C_k **in** $\mathbb{C}$ **do**
5: $\quad dates = parallelDates(\mathbb{S}, C_k, minDate, maxDate, \tau^i)$
6: $\quad$ **for** d **in** $dates$ **do**
7: $\quad\quad \mathbb{S}' = scheduleTask(\mathbb{S}, C_k, \tau^i, d)$
8: $\quad\quad computeContentions(\mathbb{S}')$
9: $\quad\quad$ **if** $mksp(\mathbb{S}') < bestMakespan$ **then**
10: $\quad\quad\quad bestMakespan = mksp(\mathbb{S}')$
11: $\quad\quad\quad bestSched = \mathbb{S}'$
12: $\quad\quad$ **end if**
13: $\quad$ **end for**
14: **end for**
15: **return** bestSched

4.3.2 Iterative priority scheduling heuristic (IPH)

The IPH, detailed in Algorithm 2, is an adaptation of the main algorithm of Hanzálek and Šůcha (2017). The principle of the original algorithm is to iteratively test different combinations of task priorities, called *priority vectors*, while converging to the best makespan. The task priorities are to be understood as the order in which the tasks are selected to be statically scheduled. Basically, the algorithm maintains an objective value for the desired makespan, as well as an interval around this objective. At each iteration of the algorithm, the tasks are scheduled following the current priority vector. The objective and interval bounds are then updated according to the makespan of the produced schedule: when the makespan is lower than the objective, it becomes the new objective for the next iterations and the interval is reset around the new objective. Otherwise, the objective and the lower bound of the interval are increased. Before the next iteration, the priority vector is modified to try to prevent the tasks that interfere the most to be in conflict in the next schedule. The algorithm ends after multiple successive iterations fail to improve the makespan, and the lower bound of the interval meets the upper bound. At each iteration, when producing the schedule, the algorithm applies a few optimizations to either speed up the computations or try to improve the result:

Optimization 1 Using a symmetric instance of the scheduling problem because scheduling backwards may open scheduling options.

Optimization 2 Implementing the algorithm in parallel so that several priority vectors are tested at the same time by separate threads.

Optimization 3 Using a hash set to store the priority vectors that have already been tried to avoid repetition.

Optimization 4 Modifying the priority vector using information about conflicting tasks that prevent each other to be scheduled before the objective.

Optimization 5 De-scheduling and re-scheduling the tasks at the end of the schedule, when the objective is not met.

This algorithm has already been successfully adapted to the AER model in Matějka et al. (2019), but our task model is more generic and some assumptions made in Matějka et al. (2019) are not applicable here. As a consequence Algorithms 2 and 3 were adapted from Hanzálek and Šůcha (2017) to handle the multiphase model.

Algorithm 2 IPH

```
Require: G = (T, E), ℂ
 1: UB, LB, 𝕊^best = init(G, ℂ)
 2: Obj = (LB + UB)/2
 3: failCount = 0
 4: G^forward = G
 5: G^backward = reverse(G)
 6: prioHashSet = {}
 7: init_prio = [UB − φ_0^i.d^#]_{∀τ^i ∈ T}
 8: sQueue = [(G^forward, Obj, init_prio)]
 9: while (LB < UB) ∧ (sQueue ≠ []) do
10:     (G^c, Obj, prio) = sQueue.pop()
11:     hash = Hash(eq_class(prio, G^c))
12:     if hash ∈ prioHashSet then
13:         continue
14:     end if
15:     prioHashSet.add(hash)
16:     𝕊 = findSchedule(G^c, ℂ, Obj, prio)
17:     if mksp(𝕊) < mksp(𝕊^best) then
18:         𝕊^best = 𝕊
19:         if UB > mksp(𝕊) then
20:             UB, LB = update(UB, LB, 𝕊)
21:         end if
22:         Obj^new = UB − 100
23:         priority = [Obj − φ_0^i.d^#]_{∀τ^i ∈ T}
24:     else
25:         failCount + +
26:         if failCount ≥ log2(|T|) then
27:             LB = LB + (UB − LB)/4
28:             failCount = 0
29:         end if
30:         Obj^new = ⌈min(UB, 1.1 × Obj)⌉
31:     end if
32:     prio_1 = [Obj − prio[i]]_{∀τ^i ∈ T}
33:     prio_2 = modPrio(prio, 𝕊)
34:     G^c1, G^c2 = switchOrder(G^c, G^backward, G^forward)
35:     sQueue.push({G^c1, Obj^new, prio_1})
36:     sQueue.push({G^c2, Obj^new, prio_2})
37: end while
38: return 𝕊^best
```

In Algorithm 2, the objective variable is denoted by *Obj*. In the initialization, we build the initial best schedule $\mathbb{S}^{best}$ using our ASAP greedy heuristic. Then,

$\mathbb{S}^{best}$ is used to build the initial target interval for the makespan $[\![LB, UB]\!]$ (Algorithm: 2-line: 1, *LB*, *UB* standing for respectively Lower Bound and Upper Bound) and *Obj* is chosen as the median value of this target interval. The initial values of the bounds do not have a huge impact on the quality of the result because the interval is re-adjusted throughout the iterations, but setting them close to a viable objective can save a few initial iterations. In order to speed up the computations, we implemented and when necessary, adapted, the following optimizations that were present in the original algorithm of Hanzálek Šůcha 2017. Optimization 1 was directly implemented in our heuristic, which is why we distinguish the two graphs $G^{forward}$ and $G^{backwards}$ (Algorithm: 2-line 5) and the priority vectors on both directions (Algorithm: 2-line 34). Optimization 2 is also implemented directly. We adapted Optimization 3 (Algorithm: 2-line: 6) to exploit the fact that two different priority vectors may produce the same scheduling order because of tasks dependencies. For example, if we consider tasks A, B and C with B and C successors of A, then assigning priorities 3, 2, 1 to respectively A, B and C yields the same scheduling order (A then B then C) as when assigning priorities 2, 3, 1 because task A must be executed before B and C has a lower priority than B. Therefore, instead of saving the priority vectors in the hash set, our algorithm computes and saves an equivalence class of the priority vectors given the dependencies of the system (i.e. the scheduling order of the tasks) (Algorithm: 2-line: 11). We also adapted Optimization 4 (Algorithm: 2-line: 33) so that, when there are no conflicting tasks, the algorithm relies on the amount of contentions to modify the priority vector. However, relying on contentions in a more systematic way did not yield any improvement of the results. Finally, Optimization 5 is implemented as part of the *findSchedule* function that we describe thereafter.

At each iteration, the algorithm calls function *findSchedule* (described in Algorithm 3 that we detail later) to build a schedule $\mathbb{S}$ from scratch using a task system G^c, a vector *prio* that gives priorities to the tasks, and an objective *Obj* for the makespan of the schedule (Algorithm: 2-line: 16). Once $\mathbb{S}$ is built, the algorithm compares its makespan with the makespan of the best schedule found so far: $\mathbb{S}^{best}$. If it is inferior, schedule $\mathbb{S}$ is saved as the new $\mathbb{S}^{best}$, the *UB* and *LB* are updated (Algorithm: 2-line: 20) in order to lower the makespan objective in the next iteration, and changes are made to the task priorities to reflect the order of the starting dates of tasks in $\mathbb{S}$ (Algorithm: 2-lines: 19–23). If it is superior to *Obj* however, *Obj* is increased in order to give some more slack to the algorithm in the next iteration, and *LB* is increased as well if the algorithm has failed enough times (Algorithm: 2-lines: 25–30). The algorithm then iterates, until either *LB* reaches *UB* or it runs out of new priority vectors to test.

There are several constants impacting the computation cost of the algorithm that are defined in an empirical way:

- Algorithm: 2-Line 22: Obj^{new}, the next objective is set to $UB - 100$. The value must not be too ambitious to allow *findSchedule* to find suitable schedules and the convergence towards the best priority vectors. As the minimum contention duration that we applied in our tests is 50 cycles, the number of contentions to avoid in order to improve the makespan is reasonable and 100 is also an order of magnitude below the duration of the tasks we scheduled who had a WCET superior to 1000 cycles (and sometimes superior to 20,000 cycles).
- Algorithm: 2-Line 26: $log2(|T|)$ bounds the number of consecutive attempts of *findSchedule* without finding a better schedule than $\mathbb{S}^{best}$ before increasing LB. This bound must be high enough to let *findSchedule* reach Obj but is also responsible for stopping the search when it is not possible. The number of tasks in the system impacts the size of the solution space. Our experiments are composed of systems ranging from 4 to 329 tasks so the $log2$ allows enough attempts for small systems while limiting them for the largest systems.
- Algorithm: 2-Line 30: whenever a failure occurs, the objective is increased by at least 10% of its value (bounded by the current UB). This value has been kept from the original algorithm in Hanzálek and Šůcha (2017).

One important point here is that the heuristic does not test all possible combinations of task priorities: at each iteration the current priority vector is modified, and the resulting vector is used in the next iteration if it has not already been used in a prior iteration. The way the algorithm modifies the priority vector does not guarantee that all priority combinations will be explored. In fact the objective of the heuristic is precisely to converge to a solution without having to explore all the combinations.

Algorithm 3 *findSchedule*

```
Require: G^c, ℂ, Obj, prios
 1: readyTasks = initRT(G^c)
 2: budget = α × |T|                    ▷ α is tuned according to the size of the task system
 3: while (readyTasks ≠ ∅) ∧ (budget > 0) do
 4:     τ^i = getNext(readyTasks, prios)
 5:     d = max_{τ^h ∈ preds(τ^i)}(φ^h_{Φ^h}.d^#)
 6:     𝕊' = scheduleASAP(𝕊, ℂ, τ^i, d)
 7:     computeContentions(𝕊')
 8:     if mksp(𝕊') > Obj then
 9:         resched, desched, 𝕊_temp = unsched(𝕊, d, Obj, τ^i)
10:         readyTasks = readyTasks ∪ desched
11:         for τ^j in resched do
12:             𝕊_temp = scheduleASAP(𝕊_temp, ℂ, τ^j, d)
13:             budget = budget − 1                    ▷ τ^j is scheduled again
14:         end for
15:         𝕊' = scheduleASAP(𝕊_temp, ℂ, τ^i, d)
16:         computeContentions(𝕊')
17:     end if
18:     updateRT(readyTasks, τ^i)
19:     budget = budget − 1                            ▷ accounting for τ^i
20: end while
21: while readyTasks ≠ ∅ do
22:     τ^i = getNext(readyTasks, prios)
23:     d = max_{τ^h ∈ preds(τ^i)}(φ^h_{Φ^h}.d^#)
24:     𝕊' = scheduleASAP(𝕊', ℂ, τ^i, d)
25:     updateRT(readyTasks, τ^i)
26: end while
27: computeContentions(𝕊')
28: return 𝕊'
```

Algorithm 3 describes the *findSchedule* function. This function iteratively creates a schedule $\mathbb{S}$ of the tasks of G^c on $\mathbb{C}$, using tasks priorities *prios* and an objective value *Obj* for the makespan of $\mathbb{S}$. A set of tasks ready to be scheduled (i.e. whose predecessors have already been scheduled) is maintained, and at each iteration, the ready task with the highest priority is selected for scheduling (Algorithm: 3-line: 4). The selected task τ^i is scheduled following a given policy (in our experiments we used ASAP) and an interference analysis is performed on the resulting partial schedule $\mathbb{S}'$ (Algorithm: 3-line: 6). Note that the priority vector does not define the mapping of the tasks so the scheduling policy is responsible for choosing the cores where tasks are scheduled. If $mksp(\mathbb{S}')$ is smaller than objective *Obj*, the algorithm updates the set of ready tasks and iterates with the next ready task (Algorithm: 3-line: 18). If, however, the partial schedule spans more than *Obj* cycles, the algorithm is allowed to de-schedule some tasks that are put back in the set of ready tasks in order to make room for τ^i before *Obj* (line: 9). The de-scheduled tasks are the tasks that start after the end of the last predecessor of τ^i and before $Obj - WCET(\tau^i)$, as well as all their (already scheduled) successors. Tasks that start after this date and are not successors of de-scheduled tasks are not put back in the ready set, but are directly rescheduled following the ASAP policy, in respect of their potential dependencies, in

order to benefit from the free intervals in the schedule left empty by the de-scheduled tasks (Algorithm: 3-lines: 11–14). Task τ^i is then scheduled (again, using ASAP in our experiments) (Algorithm: 3-line: 15). Even if objective *Obj* is still unmet, the algorithm then goes on to the next task to schedule, hoping that further de-schedulings in the next iterations will allow to meet the objective. The de-scheduling of tasks significantly affects the execution time of the algorithm compared to a greedy solution, and can create an infinite loop under certain circumstances. In order to prevent it, an exploration budget (defined in Algorithm: 3-line: 2) guarantees that the main scheduling loop does not iterate more than a fixed number of times, even though some tasks remain to be scheduled. If the number of iterations reaches the budget, the algorithm exits the loop and falls back to a greedy strategy (Algorithm: 3-line: 21) for the tasks that remain to be scheduled. Tuning the budget value thus allows to trade execution time for precision.

We define a constant α (Algorithm: 3-line 2) that sets the number of rescheduling operations allowed to reach the objective. We set $\alpha = 3$ for tasks systems with less than 26 tasks so that up to 78 tasks can be rescheduled and $\alpha = 1.2$ for the others which allows 394 rescheduling operations for the largest task system.

4.3.3 Merging optimization

In certain situations, the multi-phase model may incur an overestimation of the number of contentions during the interference analysis. In the example depicted in Fig. 5 (left), the yellow phase may contend with the three phases in parallel. As a result, the interference analysis counts 3 contentions coming from the yellow phase for each of these phases, resulting in 9 contentions in total. In practice this is impossible, as the yellow phase only performs 3 accesses in total. In order to reduce this pessimism, we developed a phase merging algorithm that can be applied on a partial or complete schedule. This optimization detects local situations in which merging together multiple phases of a task can reduce the overestimation of the number of contentions during the interference analysis.

In practice, the optimization looks for phases ϕ^i_j (called saturated phases in the following) that create more than $(|\mathbb{C}| - 1) \times \phi^i_j.m$ contentions to phases in parallel during the interference analysis. This formula was chosen as another trade-off between speed and precision. Once a saturated phase is discovered, the algorithm looks for phases scheduled in parallel and assesses whether or not it would be beneficial to merge them together. Indeed, the local benefits of merging phases (w.r.t. a given saturated phase) can be outweighed by the effects of the merge on adjacent tasks. This can be illustrated using Fig. 5:

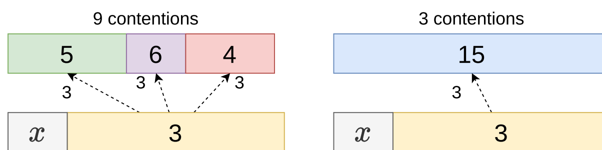


Fig. 5 An example of local overestimation of contentions (Color figure online)

- In the left part of the figure, the maximum number of contentions each phase may suffer is:
  - $min(5, x + 3)$ for the green phase.
  - $min(6, 3) = 3$ for the purple phase.
  - $min(4, 3) = 3$ for the red phase.
  - $min(x, 5)$ for the grey phase.
  - $min(3, (5 + 6 + 4)) = 3$ for the yellow phase.

  So if the phases are not merged, the interference analysis counts $9 + min(x, 5) + min(5, x + 3)$ contentions in total for the two cores.
- On the right part, when the phases are merged, this number is:
  - $min(15, x + 3)$ for the blue phase.
  - $min(x, 15)$ for the grey phase.
  - $min(3, 15) = 3$ for the yellow phase.

  So the interference analysis counts $min(15, x + 3) + min(x, 15) + 3$ contentions in total.

Therefore, if the value of x is strictly greater than 6, the merge is not globally beneficial.

Algorithm 4 *mergeOptimization*

Require: τ^i ; $\mathbb{S}$; start; end
1: $phases = getPhasesIn(\mathbb{S}, start, end)$
2: $idx = 0$
3: **while** $idx < size(phases)$ **do**
4: $\quad \phi^i_j = phases[idx]$
5: $\quad alreadyAttempted = []$
6: $\quad$ **while** $isSaturated(\mathbb{S}, \phi^i_j)$ **do**
7: $\quad\quad end = \phi^i_j.d^{\#} + \phi^i_j.dur + \phi^i_j.p$
8: $\quad\quad candidates = getPhasesWithin(\mathbb{S}, \phi^i_j.d^{\#}, end)$
9: $\quad\quad \phi^k_l, \phi^k_{l+1} = getMergeablePhases(candidates, alreadyAttempted)$
10: $\quad\quad$ **if** $\phi^k_l == null$ **then** $\quad$ ▷ no phases left that can be merged together in $candidates$
11: $\quad\quad\quad$ **break**
12: $\quad\quad$ **end if**
13: $\quad\quad alreadyAttempted.push((\phi^k_l, \phi^k_{l+1}))$
14: $\quad\quad \mathbb{S}' = mergePhases(\mathbb{S}, \phi^k_l, \phi^k_{l+1})$
15: $\quad\quad$ **if** $mksp(\mathbb{S}') < mksp(\mathbb{S})$ **then**
16: $\quad\quad\quad \mathbb{S} = \mathbb{S}'$
17: $\quad\quad$ **end if**
18: $\quad$ **end while**
19: $\quad idx = idx + 1$
20: **end while**

Algorithm 4 describes the merging optimization. As for the SDE algorithm, computing the contentions several times is necessary to identify the saturated phases and

to assess whether or not a merge is profitable. The algorithm retrieves the list of all scheduled phases and iterates over it until a saturated phase ϕ^i_j is found. When a phase is saturated, the algorithm enters the inner while loop (line 6) to try some merges. The merges are attempted using *candidates*, the list of phases in parallel of ϕ^i_j, that is retrieved by function *getPhasesWithin* (line 8). Then, function *getMergeablePhases* searches for two phases of *candidates* that are in the same task, consecutive and have not been studied before (otherwise they are present in *alreadyAttempted*). If no such phases have been found, the inner while is exited with a break (line 11). Otherwise, the phases are added to the *alreadyAttempted* list and a new schedule $\mathbb{S}'$ is created with the two phases ϕ^k_l and ϕ^k_{l+1} merged using function *mergePhases* that also recomputes the contentions. If the makespan of $\mathbb{S}'$ is better than $\mathbb{S}$ then the merge is confirmed at line 16.

The ASAP-based greedy heuristic described in Sect. 4.3.1 does not compute the contentions in the system before the schedule is produced. As a result, the scheduling decisions are not impacted by potential merges, so it is only useful to apply the merging optimization once the schedule has been entirely constructed. On the other hand, the SDE algorithm is interference-aware, so calling the merging optimization at each scheduling step can influence its decisions. In the remainder of the document, whenever the merging optimization is used, it is used after the schedule is produced with the ASAP policy, and during its construction with the SDE policy. We do not display the results of the optimization with IPH because it does not improve the trade-off between the computation speed and its efficiency to reduce the makespan of the schedule.

5 Experimental evaluation

In this section, we present a comparative study of the heuristics and an evaluation of the multi-phase model. Our evaluations use both synthetic tasks and task systems from real case studies.

5.1 Synthetic task sets generation

This section presents how the synthetic task sets used in the experiments are generated. In a nutshell, the task systems are single-periodic and the tasks are released synchronously. Precedence constraints between tasks are expressed by DAGs whose vertices represent the tasks of the system and whose edges represent the dependencies between tasks. For a given experiment, a multi-phase profile and an equivalent single-phase profile are generated for each task, and the task system with multi- and single-phase representations are scheduled independently. Then a separate interference analysis is performed on both resulting schedules. In the end the results obtained for the multi-phase and for the single-phase representations of the same task system are compared, for each synthetic task system.

Table 3 Description of the tests input parameters

Parameter	Description
Number of cores	Number of cores in the architecture
Access cost	Cost of a memory access in cycles
Penalty factor	Multiplier applied to the access cost to compute the interference penalty
Number of phases	Average number of phases per task
Temporal shape	Policy to generate the duration of the phases
Empty phases	Proportion of empty phases (i.e. phases ϕ_l^i s.t. $\phi_l^i.m = 0$) in each profile (%)
Access shape	Policy to generate the number of accesses into the phases
Access rate	Average number of accesses performed for 10,000 execution cycles
Over-approximation	Proportion of additional accesses in the multi-phase representation compared to the single-phase one in %
ov-app	Short for over-approximation
Number of tasks	Number of tasks to schedule

5.1.1 Generation of synthetic task systems

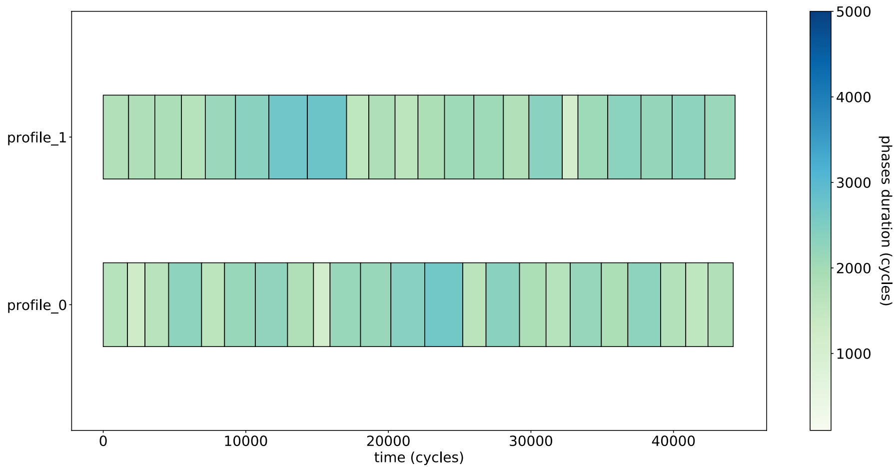
The first part of the experiments has been conducted on task systems composed of synthetic multi-phase profiles. Their generation is controlled by a set of input parameters presented in Table 3.

The first three parameters characterize the hardware architecture on which the tasks are scheduled. The penalty factor parameter tunes the cost of contentions as a multiple of the cost of an access in isolation. Indeed, the memory latency of an access in the presence of interference can be several times the cost of an access in isolation due to indirect effects, e.g. in the pipeline (Sebastian and Jan 2020; Gruin et al. 2023). Setting the penalty factor to 1 is therefore the most optimistic assumption for an architecture, which usually is an unfavorable assumption for our experiments. The experiments are conducted with a penalty factor of 1 and 3, which correspond respectively to an optimistic and a more realistic assumption.[4]

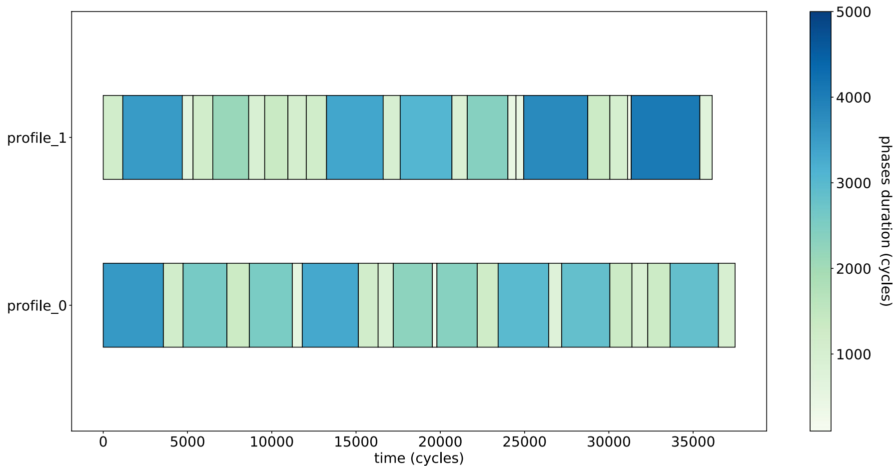
The number of phases for each task is drawn from a normal law centered around a target value specified by the “number of phases” parameter. As for the duration of the phases, a temporal shape parameter specifies the desired generation method:

1. Normal (N): each duration is drawn from a single normal law, with a unique average value and standard deviation for all the phases of a task. This method tends to generate little variation in the duration of the phases of a profile. Two examples of profiles generated with the Normal policy are given in Fig. 6a.
2. Bi-Normal (BN): the durations are drawn from two distinct normal laws that have a different average value such that there are short and long phases. In preliminary experiments, the duration ratio between short and long phases did

[4] 3 times the memory latency corresponds to the upper bound for the timing-compositional SIC core (Sebastian and Jan 2020)



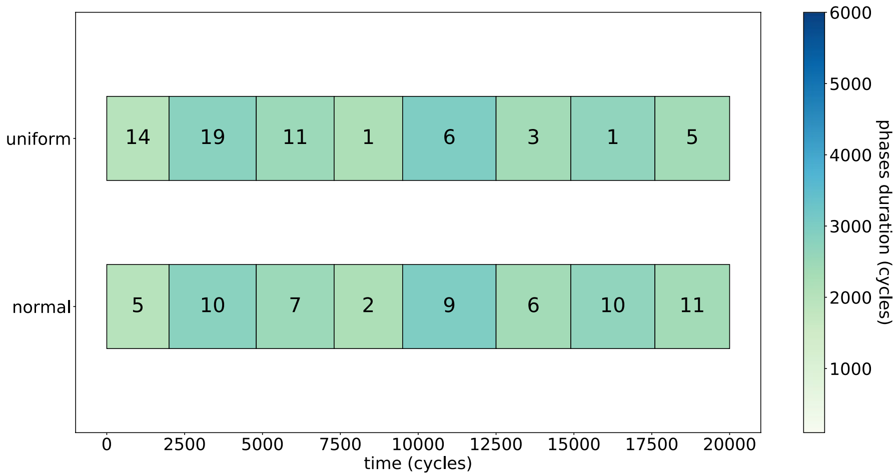
(a) Normal duration policy.



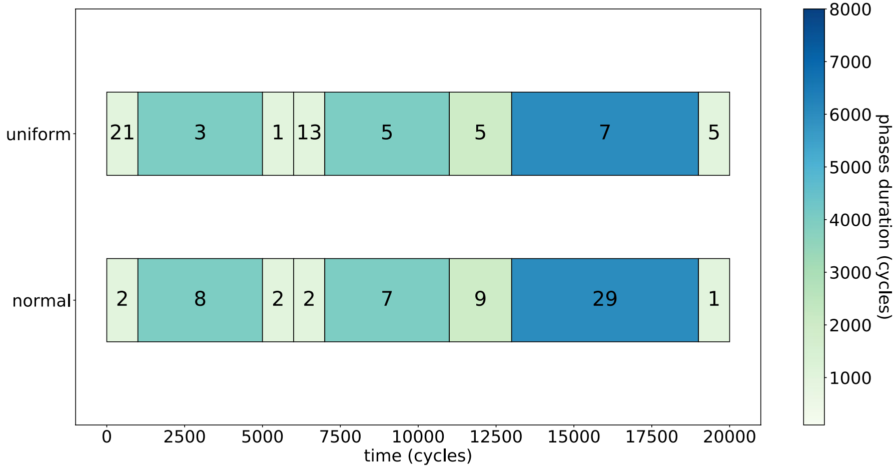
(b) Bi-Normal duration policy.

Fig. 6 Examples of profiles generated with the Normal and Bi-Normal duration policies

not seem to influence the results, so the ratio between the centers of both normal laws is arbitrarily set to 3 in our evaluations. Other preliminary experiments also suggested that profiles in which long phases and sequences of short phases alternate perform better than other profile shapes (Rémi et al. 2022). Thus, in BN generated profiles, a long phase is systematically followed by a short phase, while a short phase can be followed by a sequence of short phases. Figure 6b shows 2 profiles generated with this policy.



(a) Uniform and normal access shapes coupled with the normal temporal shape.



(b) Uniform and normal access shapes coupled with the bi-normal temporal shape.

Fig. 7 Examples of profiles generated with the Uniform (top) and Normal (bottom) access shapes

Moreover, when analyzing real code, phases in which there are no memory accesses (called empty phases in the remainder of the paper) are expected due to e.g. cache effects. Thus, the experiments include profiles without empty phases as well as profiles with around 20% of empty phases[5] to study their influence on the scheduling results.

The number of accesses in each phase can be generated with 2 different methods:

[5] For profiles in which 20% of the phases is not an integer number, we rounded up or down to the closest integer.

1. Normal (N): for each phase, its access rate is drawn from a unique normal law centered around the value specified by the access rate input parameter. Then, the number of accesses is computed by multiplying the drawn access rate for the phase by its duration. With this generation mode, on average, longer phases tend to perform more accesses than shorter ones.
2. Uniform (U): the task-wise number of accesses is computed using the access rate input parameter and the duration of the task, then the accesses are distributed between the phases using a uniform law, so that all phases have the same probability to get accesses. This method thus distributes the accesses independently from the duration of the phases, which leads to a greater variability in the generated distributions of accesses: some distributions are similar to the ones generated using the normal law, but compared to the normal law, there is a higher probability to generate distributions in which smaller phases have a high number of accesses and longer phases a small number of accesses.

Figure 7 displays examples of N and U distributions of memory accesses in a N temporal shape (Fig. 6a) and in a BN temporal shape (Fig. 6b).

The generation of a profile begins by the generation of the duration of the phases according to the temporal shape policy. Then, the number of accesses in the phases is chosen according to the access rate of the task, its WCET and the access shape policy. Once the duration and accesses of each phase have be chosen, a correction pass is performed to ensure that for any phase the sum of the duration of its accesses does not exceed its duration.

In order to assess the efficiency of the multi-phase model compared to single-phase, the generated multi-phase profiles are converted to their single-phase equivalent by summing the duration and the number of accesses of the phases.

In practice, depending on the analysis method and on the tuning of parameters, the multi-phase representation of a task may account for more accesses than its single-phase counterpart, because the phases are unaware of unfeasible paths in the code. To account for this possible over-estimation of accesses in our synthetic task profiles, the number of accesses in the single-phase model is reduced according to an access over-approximation input parameter. In the remainder of the paper, the terms access over-approximation or access over-estimation designate this overhead between the number of memory accesses accounted for in the multi-phase model of a task and in its single-phase counterpart. This way it is also possible to assess the impact of the access over-approximation level on the quality of the produced schedules.

5.1.2 Generation of tasks dependencies

Task dependencies are defined in a DAG. Such DAGs are generated as series–parallel graphs: for each vertex without a successor, we generate either a fork, resulting in a parallel sub-graph of successors (with a probability of 0.7) or a sequence of successors (with a probability of 0.3), until the number of generated vertices is equal to the desired number of tasks for the system. The first vertex is

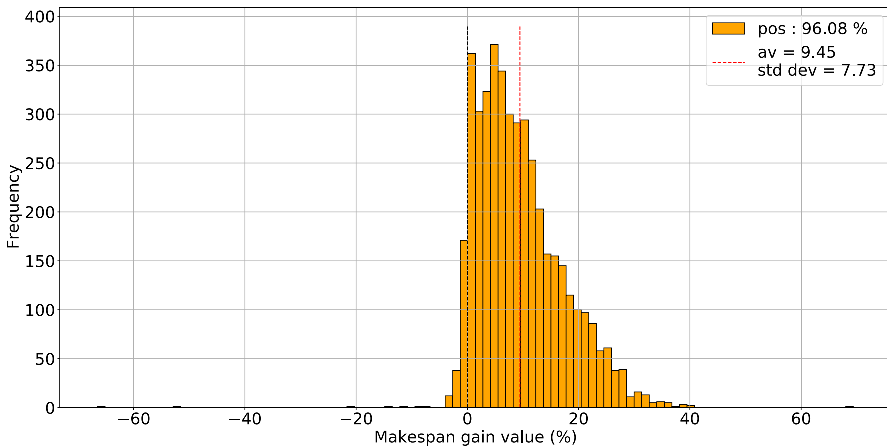


Fig. 8 Makespan gain of multi-phase ILP vs single-phase ILP in %

always followed by a parallel sub-graph (to avoid having only sequential tasks at the start of the schedules). When at least 2 forks have been generated (the second being a direct or indirect successor of the first), the algorithm can (with a probability of 0.2) generate a new vertex as a successor of all vertices that are currently without a successor, thus realizing a join in the graph. The probabilities (resp. 0.7, 0.3 and 0.2) were arbitrarily calibrated to obtain series–parallel graphs with enough parallelism to allow interference to occur in the system.

5.2 Tests metrics

The two metrics that we considered in the experiments are the *makespan* of the schedule in the presence of interference and the *total number of contentions* that may appear in the schedule according to the interference analysis.

For each metric m, the notion of *gain* is defined as the comparison of the value of m in a given schedule to the value of m in a baseline schedule (the single-phase variant of the schedule):

$$gain = (m_value_baseline - m_value_schedule)/m_value_baseline.$$

Moreover, a test is considered to be positive if $gain \geq 0$ for the corresponding task system, scheduling policy and metric m. In other terms, a positive test means that the multi-phase instance of the problem yields improved results compared to its single-phase counterpart.

5.3 Comparative study using the ILP formulation

The optimization problem is inherently nonlinear due to the interference computation. Consequently, the ILP solver (Gurobi 9.5.1 Gurobi 2022) encountered scalability issues. Thus, this section only presents the results for experiments

Table 4 Makespan gain results compared to ILP with multi-phase or single-phase

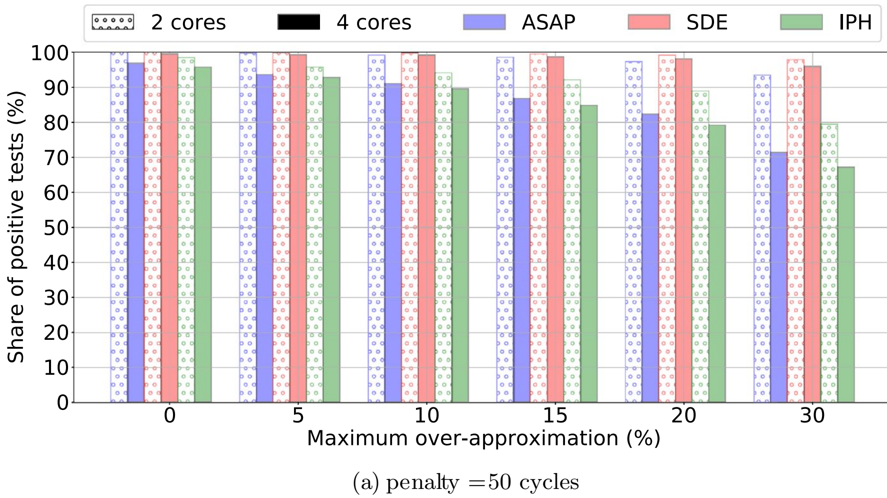
Cores	Heuristic	Gain vs ILP multi		Gain vs ILP single	
		Share of positive (%)	Average gain (%)	Share of positive (%)	Average gain (%)
2	IPH	7.31	− 3.71	90.51	5.17
	SDE	2.70	− 5.69	73.77	3.20
	+Merge	6.34	− 5.05	77.47	3.83
	ASAP	2.15	− 7.77	63.77	1.11
	+Merge	5.15	− 6.76	70.50	2.12
4	IPH	1.29	− 5.02	88.06	5.82
	SDE	1.37	− 5.77	81.29	5.07
	+Merge	3.06	− 5.43	83.87	5.42
	ASAP	0.56	− 9.35	64.84	1.50
	+Merge	1.69	− 8.42	70.40	2.42

with 4–6 tasks composed of 4–6 phases each, and whose solving time was inferior to 6 h (a timeout was set in the experiments). We set the cost of a memory access to 50 cycles in our experiments. The experiments include tasks with an average access rate of 25, 50 or 75 accesses per 10,000 cycles (corresponding to 12.5, 25 and 37.5% of the cycles spend in memory accesses), without access over-approximation (the number of accesses in the single phase variant of a task is equal to the sum of accesses of the phases in the multi-phase variant), and with either no empty phases, or with 20% of the phases empty.[6] When computing the time penalties due to contentions, the penalty duration is set to either 50 or 150 cycles, system-wide, corresponding to a penalty of 1 or 3 times the duration of a memory access of 50 cycles.

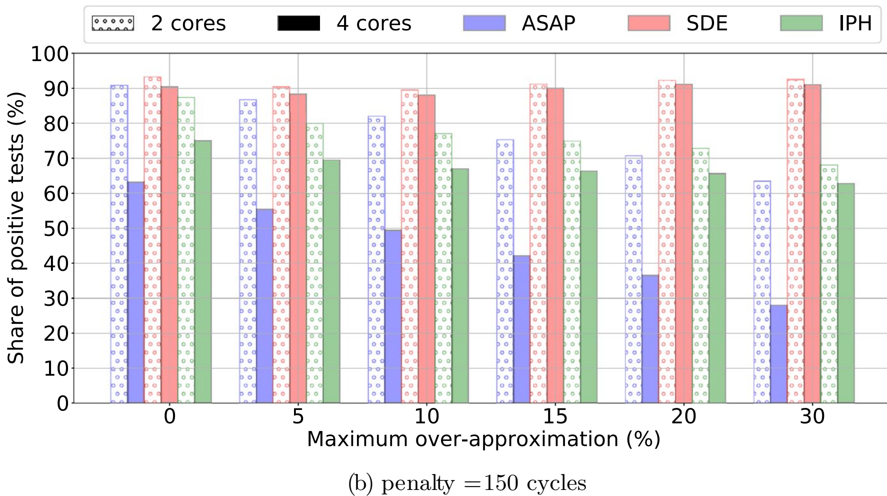
5.3.1 ILP results

The distribution of the gain values obtained by the multi-phase variants compared to their single-phase counterparts (on 4236 experiments), scheduled with ILP is represented in Fig. 8. The extreme values are not represented for readability reasons: the gain varies from − 66.49 to 69.38% and the average value is 9.45%. In other terms, we can expect to reduce the system worst-case makespan by around 9% on average by switching from single to multi-phase representation. Moreover, in more than 96% of the tests the results of the multi-phase ILP were positive, i.e. at least as good as the single-phase ILP. The rare cases where multi-phase ILP performs worse than single-phase ILP can be attributed to the problem described in Sect. 4.3.3. As these results have been obtained with only small instances of the problem, they do not allow to draw general conclusions. However, it is worth noting that the gap between

[6] Following preliminary analyses on the TACLe benchmark suite (Falk 2016), we estimate 20% to be a reasonable amount of empty phases.



(a) penalty =50 cycles



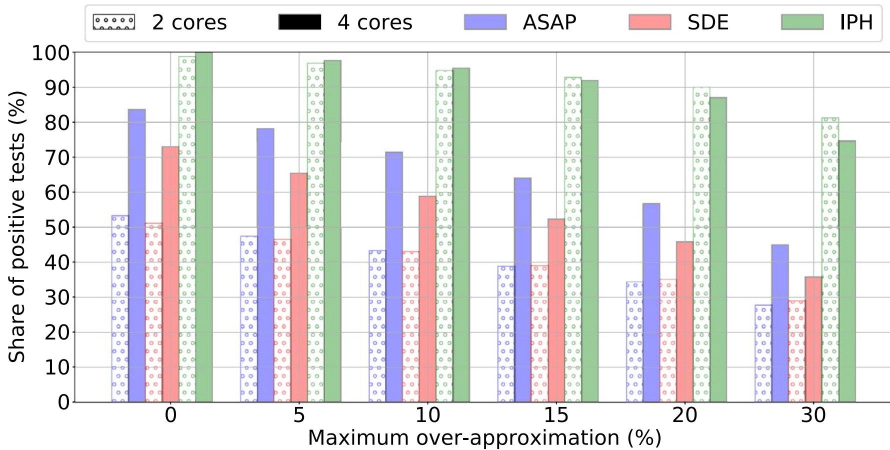
(b) penalty =150 cycles

Fig. 9 Share of positive results in terms of contentions according to the access over-approximation for 2 interference penalty values compared to single-phase IPH

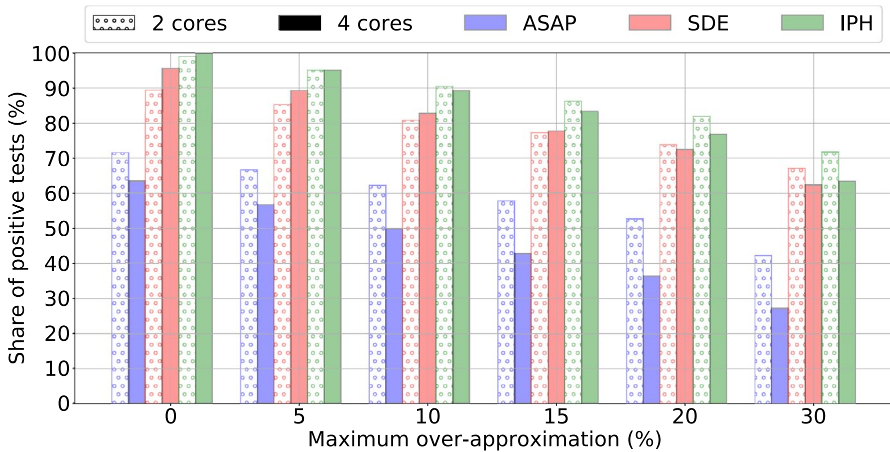
the two models increases when the number of cores goes from 2 to 4, since the experiments with 2 cores have an average gain of 8.88% while it is 10.85% for the tests with 4 cores. This is coherent with the fact that the effect of potential contentions increases with the number of cores.

5.3.2 Comparison of ILP and heuristics for multi-phase profiles

Table 4 shows the average gain and the proportion of results where each heuristic result was at least as good as the ILP solving the multi-phase or single-phase



(a) penalty =50 cycles



(b) penalty =150 cycles

Fig. 10 Share of positive results in terms of makespan according to the access over-approximation for 2 interference penalty values compared to single-phase IPH

problem. All the heuristics are on average less than 10% worse than the optimal multi-phase result and IPH is only at 3.71% on 2 cores and 5% on 4 cores. When the merging optimization was used, ASAP and SDE were even able to beat the multi-phase ILP (in respectively 2.5% and 3.3% of the tests) because of the new profiles generated by the optimization. A version of the ILP with possible merges would considerably increase its complexity and solving time so it is not proposed. Moreover, the heuristics applied on multi-phase profiles are at least as good as the optimal solution for the equivalent 1-phase profiles in at least 63% of the experiments (up to 90.51% for IPH with 2 cores). Finally, IPH finds the optimal

multi-phase schedule in 7.31 and 1.29% of these (simple) experiments respectively on 2 and 4 cores.

5.4 Comparative study on larger systems

In this section we study the influence of the parameters on the gain of the multi-phase model and compare the heuristics on larger task systems. They are composed of either 20 or 25 tasks,[7] with the number of phases drawn from a normal law centered around 15 or 20. Moreover, several over-approximation values from 0 to 30% of additional accesses are tested. The other parameters are the same as in the previous section.

5.4.1 Influence of the target architecture: interference penalty and number of cores

Figure 9 shows the share of experiments with a positive gain in terms of reduction of the worst-case number of contentions. It shows that SDE is the best heuristics to reduce the number of contentions, which is expected as it is the only one that makes decisions based on interference-aware (partial) schedules. In terms of makespan reduction, Fig. 10 shows that IPH dominates the other heuristics. This confirms the results of Schuh et al. (2020): tolerating a certain level of contentions in the system is more efficient, on average, than systematically postponing the start date of tasks to avoid interference, when it comes to reducing the makespan. However, when the penalty for a contention increases from 50 to 150 cycles, SDE improves, achieving results closer to IPH: as the penalty for each contention increases, postponing the execution of tasks to reduce the number of contentions becomes more profitable.

We started our experiments using ASAP as the baseline for single-phase as it was very fast. During the course of the experiments, we realized that IPH, although designed with multi-phase in mind, was also very efficient to schedule single-phase task systems, and outperformed ASAP in most cases. We thus decided to compare our multi-phase results with their single-phase counterparts scheduled with ASAP and IPH. The average makespan gain is represented by Figs. 11 and 12 respectively against single-phase ASAP and IPH. The same observations as with the share of positive results can be made: SDE is the least efficient heuristic when the penalty is 50 cycles but its gain improves as the penalty increases. With a 150 cycles penalty, the gain of the multi-phase model reaches 15.86% using IPH against single-phase ASAP, while the maximum is 7.47% against single-phase IPH.

In a nutshell, IPH is the most adapted to reduce the makespan of the task systems while SDE is the most efficient to reduce contentions. The reason for this is that SDE tends to take short-term decisions that mainly reduce the worst-case number of contentions. However, it is sometimes better to accept more contentions locally to reduce the makespan of the entire system. When the effects of contentions are more important (i.e. a higher number of cores or a greater interference penalty), avoiding contentions is more correlated to reducing the makespan of the schedule so SDE becomes more efficient to reduce the makespan.

[7] The reported results of this section combine the experiments with 20 and 25 tasks.

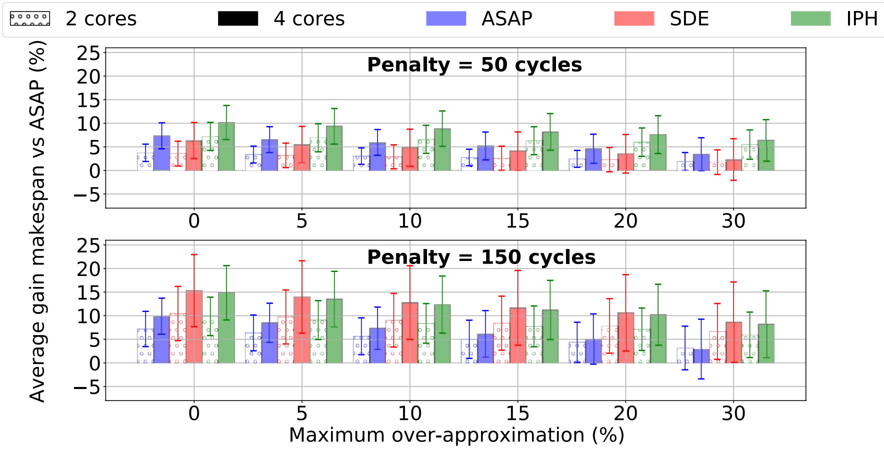


Fig. 11 Average makespan gain vs ASAP single-phase according to the access over-approximation

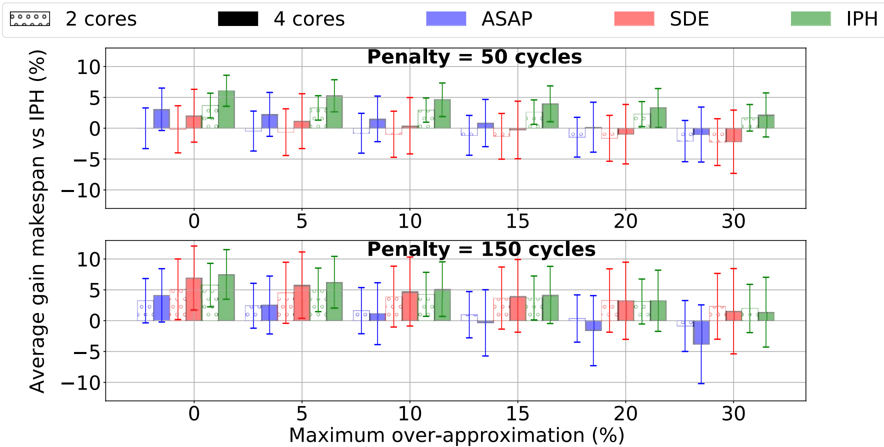


Fig. 12 Average makespan gain vs IPH single-phase according to the access over-approximation

5.4.2 Influence of the input parameters

Table 5 shows the share of positive tests for each pair of temporal/access shape. According to the results, regardless of the temporal shape, the uniform access shape performs better than the normal shape. In other words it seems that the variation in phase duration within a profile has a very limited impact on the reduction of the makespan, and that profiles in which the number of accesses is not correlated to the length of the phases perform significantly better than those where such a correlation exists. In practice, it means that the analysis methods that build the profiles should

Table 5 Makespan gain results compared to single-phase IPH for the different profile shapes

Cores (#)	Maximum over-approx. (%)	Share of positive tests (*gain* ≥ 0)			
		Bi-Normal		Normal	
		Normal	Uniform	Normal	Uniform
2	0	67.73	83.19	68.50	82.17
	5	58.14	81.32	61.58	79.83
	10	48.26	79.03	56.64	77.43
4	0	78.80	88.46	80.97	89.72
	5	64.35	87.53	71.04	86.18
	10	50.62	83.46	62.82	83.10

focus on optimizing the way the accesses are grouped rather than the duration of the phases.

Figure 13 shows the makespan gain of our three heuristics against IPH single-phase while varying the amount of empty phases (i.e. with no access). All heuristics perform significantly better when 20% of the tasks execution time is spent in empty phases, regardless of the value of the penalty, or of the level of over-approximation of accesses. When the penalty is set to 150 cycles, the difference in gain for SDE nearly doubles for 0% and 5% overestimation (and more than doubles for 15%). This demonstrates the crucial aspect of empty phases to improve the makespan of the scheduled systems.

Table 6 gives the average computation time of the heuristics according to the number of tasks in the system and the average number of phases. As expected, ASAP is the fastest of our 3 heuristics. Then SDE is faster than IPH for the systems composing our benchmark. However, when the workload increases the computation time increases comparatively more for SDE than for IPH. It is expected that SDE will be slower than IPH for very large systems.

5.5 Case Studies

In this section, we apply the heuristics on Pagetti et al. (2014), a multi-periodic flight controller, and Nemer et al. (2006) that is derived from an open-source UAV control application. The environmental simulation tasks of Rosace are not considered, as they are not embedded code and their WCET is several orders of magnitude larger than that of the other tasks.

The tasks have been compiled for ARM targets. We consider a multicore architecture in which each core has a private L1 LRU data cache and an instruction scratchpad. The memory latency was set to 50 cycles for non-cached accesses. The tasks were analyzed with OTAWA (Ballabriga et al. 2010) to extract their Control Flow Graph (CFG) and perform a cache analysis. Then, the multi-phase profiles of the tasks were computed using the Time Interest Points (TIPs) method from Carle and Cassé (2021). Based on the execution traces of a task, the TIPs method creates

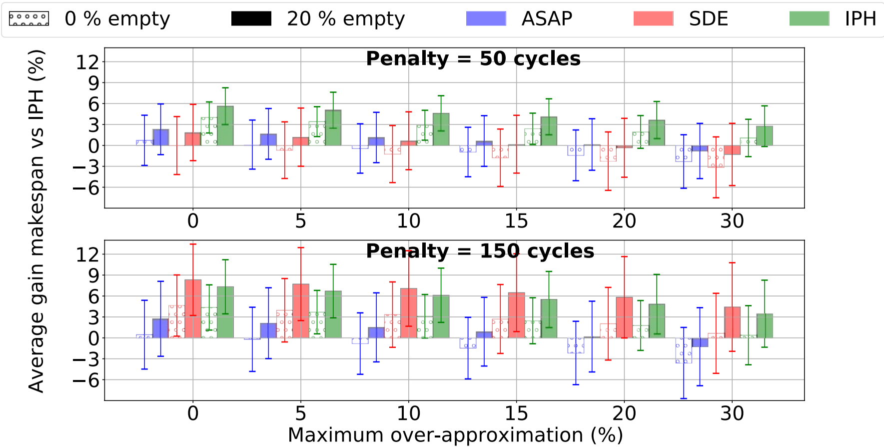


Fig. 13 Average makespan gain vs IPH single-phase according to the presence of empty phases

Table 6 Average computation time for the different heuristics

Tasks (#)	Phases per task (#)	Average computation time (s)		
		ASAP	SDE	IPH
20	15	< 1	55	334
	20	< 1	112	499
25	15	< 1	88	596
	20	< 1	172	911
All tests		< 1	105	574

short phases that correspond to the memory accesses, and then merges overlapping and adjacent phases. In order to tune the obtained profile, the user specifies a δ value that indicates the expected minimal duration of the phases in the profile.

Some of the original tasks of Papabench were split to reduce the complexity of the analysis. Then, as the systems are multi-periodic applications, the task system was converted into a DAG of single-period tasks over one hyperperiod following the methodology of Carle et al. (2015) but without using release dates for the jobs (i.e. with a synchronous release). The resulting DAG is composed of 77 tasks for Rosace and 329 tasks for Papabench. Statistics about the profiles are provided by Table 7 (*empty dur* is the proportion of execution time guaranteed without access). As expected, as δ diminishes, the number of phases increases, reaching up to 14.4 (resp. 27.38) phase per task on average for Rosace (resp. for Papabench). The number of synchronizations required to implement these profiles remains under 2 per phase on average for Rosace, and reaches 3 per phase in the worst case for Papabench, which remains acceptable. The over-approximation also increases but remains low (less than 2% for Rosace and 8% for Papabench), while the percentage of time spent in

Table 7 Statistics of Rosace and Papabench profiles

δ	Rosace				Papabench			
	Sync (#)	Phases (#)	ov-app (%)	Empty dur (%)	Sync (#)	Phases (#)	ov-app (%)	Empty dur (%)
1000	384	294	0.00	0.00	5622	2755	4.01	9.54
500	561	531	0.00	17.96	12,937	4788	4.73	22.87
200	1020	1110	1.22	28.55	24,891	9009	7.45	35.65

empty phases increases fast, and reaches up to 28% (resp. 35%) for Rosace (resp. Papabench).

Then IPH, SDE and ASAP were used to schedule the DAGs on 2, 3 or 4 cores and we applied interference analyses on the schedules with a penalty of 50 and 150 cycles.

The results presented in Tables 8 and 9 show that the gain tends to increase when the phases are smaller (i.e. when δ is lower). Following our previous observations, this can be the result of the increased proportion of time spent in empty phases, and of a better distribution of accesses among phases. The multi-phase model globally yields better results than the 1-phase model, with a makespan gain up to 16.31% for Papabench (IPH on 2 cores with $\delta = 500$ cycles and a penalty of 150 cycles) and 24.00% for Rosace (SDE on 4 cores with $\delta = 200$ cycles and a penalty of 150 cycles). For Papabench, IPH always performs the best improvements, ranging from 7 to 16% compared to the 1-phase ASAP and between 4 and 11% compared to the 1-phase IPH. When the penalty is 50 cycles, SDE is the worst heuristic and its makespan is often higher than if the tasks are represented with the single-phase model (i.e. $gain < 0$). However, with a 150 cycles penalty per contention, SDE is more efficient than ASAP with a gain ranging from nearly 7 to 13% for the makespan. For Rosace, SDE is more efficient, as the gain is always positive and often close to ASAP with 50 cycles of penalty, and it is even the best heuristic when the penalty is 150 cycles.

The two tables also display the gain in terms of contentions. For Papabench (resp. Rosace), this gain ranges from 6.92 to 64.36% (resp. 2.42–55.80%) compared to 1-phase ASAP scheduling. This means that on top of reducing the makespan of the computed schedules, our heuristics, coupled with the multi-phase model, are able to significantly improve the timing predictability of the scheduled applications because there is less variability in the number of contentions that may occur in the system (i.e. the maximum interference scenario is closer to the average case scenario). SDE is the best heuristic to reduce contentions, even when it obtains negative makespan gains, which is coherent with what we observed with the synthetic systems.

With $\delta = 1000$ on 2 cores, the time required to schedule Papabench (resp. Rosace) with ASAP was 1 min (resp. less than 1 s) while it took nearly 8 h when applying SDE (resp. 43 s) and 6 h (resp. 3 min) to run IPH with up to 31 threads (resp. 19) computing a schedule at a time. However, as IPH is an iterative heuristic, it is able to find the best result or at least a satisfying result within the early iterations. For Papabench the schedule was found in less than 3 h.

Table 8 Results of heuristics to schedule Papabench tasks with TIPs profile

	nb	δ	Penalty = 50 cycles			Penalty = 150 cycles		
	Cores		Gain makespan (%)		Gain contentions (%)	Gain makespan (%)		Gain cont. (%)
			vs ASAP	vs IPH	vs ASAP	vs ASAP	vs IPH	vs ASAP
ASAP	2	1000	7.17	*– 0.64*	8.32	9.15	3.37	17.58
		500	8.31	0.58	18.37	10.90	5.23	25.72
		200	**9.07**	1.41	24.71	**11.93**	**6.33**	28.96
	3	1000	4.81	1.18	14.18	6.13	1.60	7.32
		500	6.37	2.88	22.35	7.94	3.49	17.75
		200	6.96	3.41	23.68	9.00	4.61	13.18
	4	1000	***4.66***	1.25	19.85	*5.13*	*1.41*	13.96
		500	6.24	2.88	25.62	7.97	4.36	18.16
		200	6.84	**3.51**	27.43	9.01	5.44	19.61
SDE + merge	2	1000	– 2.36	*– 10.98*	47.13	9.94	4.21	57.06
		500	**0.96**	– 7.38	54.73	**13.61**	**8.11**	64.70
		200	0.75	– 7.61	40.59	12.85	7.31	62.83
	3	1000	*– 5.05*	– 9.06	30.50	7.44	*2.98*	59.58
		500	– 1.39	– 5.26	37.88	12.03	7.79	64.45
		200	– 1.02	– 4.86	38.46	9.88	5.53	68.35
	4	1000	– 4.72	– 8.48	36.38	*7.18*	3.53	65.11
		500	– 1.61	– 5.25	40.04	11.15	7.66	65.61
		200	– 0.57	– **4.17**	38.90	9.83	6.28	67.17
IPH	2	1000	12.86	5.52	14.25	13.02	7.49	25.28
		500	14.18	6.95	25.85	**16.31**	10.99	35.08
		200	**14.97**	**7.81**	31.29	16.12	10.79	38.56
	3	1000	8.88	5.56	30.97	*10.96*	*6.67*	38.59
		500	9.98	7.36	39.56	14.91	10.80	37.37
		200	9.59	6.18	35.28	14.51	10.38	49.37
	4	1000	*7.80*	*4.28*	42.53	11.62	8.15	42.51
		500	9.42	6.26	44.13	13.83	10.44	47.20
		200	9.63	6.65	45.61	14.52	**11.16**	47.74

6 Conclusion

We presented static scheduling methods to reduce the worst-case makespan of multi-core real-time applications, using the multi-phase model. We proposed an ILP formulation of the problem and then 3 heuristics to overcome the scalability issues of the ILP. The experimental section assesses the heuristics based on optimal solutions provided by the ILP, and by comparing them to the 1-phase model on larger synthetic task sets. Finally, the heuristics have been applied on 2 realistic applications. The results of the ILP for small systems show that one can expect a reduction of around 9% on average when switching from the single to the multi-phase model. On our case studies, our heuristics manage to improve the makespan by up to 16% for Papabench and 24% for Rosace, compared to their single-phase version scheduled

Table 9 Results of heuristics to schedule Rosace tasks with TIPs profile

	nb cores	δ	Penalty = 50 cycles			Penalty = 150 cycles		
			Gain makespan (%)		Gain contentions (%)	Gain makespan (%)		Gain cont. (%)
			vs ASAP	vs IPH	vs ASAP	vs ASAP	vs IPH	vs ASAP
ASAP	2	1000	*2.42*	0.76	9.03	*2.31*	1.28	3.80
		500	3.26	1.10	13.82	5.86	4.87	11.83
		200	4.90	2.77	22.35	9.15	**8.19**	18.93
	3	1000	4.71	*− 0.04*	2.42	5.01	2.83	3.98
		500	6.96	2.32	9.79	6.48	4.33	7.12
		200	8.71	4.16	14.28	8.65	6.56	9.88
	4	1000	11.18	3.17	5.94	13.23	*0.83*	9.70
		500	13.65	5.86	11.59	14.89	2.73	13.22
		200	**15.78**	**8.18**	17.15	**16.80**	4.92	16.76
SDE + merge	2	1000	*0.90*	− 1.32	15.82	10.75	9.81	39.66
		500	3.78	1.63	31.20	13.28	12.36	55.20
		200	2.85	0.67	37.11	17.04	16.17	50.69
	3	1000	1.11	*− 3.82*	10.79	*9.39*	*7.31*	51.99
		500	5.17	0.44	23.82	12.34	10.32	44.12
		200	7.64	3.03	26.23	17.55	**15.65**	44.12
	4	1000	7.37	− 0.99	14.58	20.36	8.98	55.80
		500	13.87	6.09	20.88	20.54	9.19	35.48
		200	**15.54**	**7.92**	26.31	**24.00**	13.14	40.11
IPH	2	1000	*4.26*	2.12	7.10	4.87	3.87	10.21
		500	5.70	3.59	14.32	9.20	8.24	21.86
		200	7.22	5.14	21.48	11.79	**10.86**	25.28
	3	1000	6.64	*1.98*	3.71	*5.19*	*3.01*	34.67
		500	9.05	4.52	10.54	8.95	6.86	24.95
		200	10.64	6.18	17.09	11.32	9.28	27.49
	4	1000	14.68	6.98	0.27	15.56	3.50	23.46
		500	17.35	9.90	7.27	17.82	6.08	45.18
		200	**19.04**	**11.73**	9.90	**19.69**	8.21	48.58

with ASAP (11 and 15% compared to single-phase IPH). These experiments show the potential that the multi-phase model yields when properly exploited by scheduling techniques. In future works, we are going to investigate to which extent these results can be generalized to on-line schedulers and preemptive systems.

Acknowledgements This work was supported by a grant overseen by the French National Research Agency (ANR) as part of the MeSCAliNe (ANR-21-CE25-0012) project. Experiments of Sect. 5 presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

Funding Open access funding provided by Université Toulouse III - Paul Sabatier.

Data Availability Not applicable.

Declarations

Conflict of interest The authors have not disclosed any competing interests.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

AbsInt (2022) Ait. https://www.absint.com/ait/index.htm

Ballabriga C, Cassé H, Rochange C, Sainrat P (2010) OTAWA: an open toolbox for adaptive WCET analysis. In: Min SL, Pettit R, Puschner P, Ungerer T (eds) 8th IFIP WG 10.2 International workshop on software technologies for embedded and ubiquitous systems (SEUS). Software technologies for embedded and ubiquitous systems, vol LNCS-6399, Waidhofen/Ybbs, Austria, Springer, pp 35–46

Becker M, Dasari D, Nikolic B, Akesson B, Nélis V, Nolte T (2016) Contention-free execution of automotive applications on a clustered many-core platform. In: 2016 28th Euromicro conference on real-time systems (ECRTS), pp 14–24

Carle T, Cassé H (2021) Static extraction of memory access profiles for multi-core interference analysis of real-time tasks. In: Hochberger C, Bauer L, Pionteck T (eds) Architecture of computing systems—34th international conference, ARCS 2021, virtual event, June 7–8, 2021, proceedings, vol 12800. Lecture Notes in Computer Science. Springer, Berlin, pp 19–34

Carle T, Dumitru P-B, Sorel Y, Lesens D (2015) From dataflow specification to multiprocessor partitioned time-triggered real-time implementation. Leibniz Trans Embed Syst 2(2):01:1-01:30

Davis RI, Altmeyer S, Leandro IS, Maiza C, Nélis V, Reineke J (2018) An extensible framework for multicore response time analysis. Real Time Syst 54(3):607–661

de Dinechin MD, Schuh M, Moy M, Maiza C (2020) Scaling up the memory interference analysis for hard real-time many-core systems. In: 2020 Design, automation & test in Europe conference & exhibition, DATE 2020, Grenoble, France, March 9–13, 2020. IEEE, pp 330–333

Degioanni T, Puaut I (2022) StAMP: Static analysis of memory access profiles for real-time tasks. In: Ballabriga C (ed) 20th International workshop on worst-case execution time analysis (WCET 2022). Open access series in informatics (OASIcs), vol. 103, Dagstuhl. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany, pp 1:1–1:13

Didier K, Potop-Butucaru D, Iooss G, Cohen A, Souyris J, Baufreton P, Graillat A (2019) Correct-by-construction parallelization of hard real-time avionics applications on off-the-shelf predictable hardware. ACM Trans Archit Code Optim 16(3):24:1-24:27

Durrieu G, Faugère M, Girbal S, Graciaèrez D, Pagetti C, Puffitsch W (2014) Predictable flight management system implementation on a multicore processor. In: ERTS'14

Falk H et al (2016) TACLeBench: A benchmark collection to support worst-case execution time research. In: Schoeberl M (ed) 16th International Workshop on worst-case execution time analysis (WCET 2016). OpenAccess Series in Informatics (OASIcs), Dagstuhl. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, Germany, pp 2:1–2:10

Gruin A, Carle T, Rochange C, Cassé H, Sainrat P (2023) Minotaur: A timing predictable RISC-V core featuring speculative execution. IEEE Trans Comput 72(1):183–195

Gurobi Optimization (2022) LLC. Gurobi optimizer reference manual. https://www.gurobi.com

Hahn S, Reineke J (2020) Design and analysis of SIC: a provably timing-predictable pipelined processor core. Real Time Syst 56(2):207–245

Halbwachs N (1992) Synchronous programming of reactive systems, vol 215. Springer, Berlin

Hanzálek Z, Š$\mathring{{\rm u}}$cha P (2017) Time symmetry of resource constrained project scheduling with general temporal constraints and take-give resources. Ann Oper Res 248(1–2):209–237

Jati A, Cláudi M, Aftab RS, Geoffrey N, Eduardo T (2021) Bus-contention aware schedulability analysis for the 3-phase task model with partitioned scheduling. In: Audrey Q, Iain B, Giuseppe L (eds) RTNS'2021: 29th International conference on real-time networks and systems, Nantes, France, April 7–9, 2021. ACM, pp 123–133

Maiza C, Rihani H, Juan MR, Goossens J, Altmeyer S, Davis RI (2019) A survey of timing verification techniques for multi-core real-time systems. ACM Comput Surv 52(3):56:1-56:38

Matějka J, Forsberg B, Sojka M, Š$\mathring{{\rm u}}$cha P, Benini L, Marongiu A, Hanzálek Z (2019) Combining PREM compilation and static scheduling for high-performance and predictable MPSoC execution. Parallel Comput 85:27–44

Meunier R, Carle T, Monteil T (2022) Correctness and efficiency criteria for the multi-phase task model. In: Maggio M (ed) 34th Euromicro conference on real-time systems (ECRTS 2022). Leibniz International Proceedings in Informatics (LIPIcs), Dagstuhl, vol 231 Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany, pp 9:1–9:21

Nemer F, Cassé H, Sainrat P, Bahsoun J-P, De Michiel M (2006) PapaBench: a free real-time benchmark. In: 6th International workshop on worst-case execution time analysis (WCET'06)

Pagetti C, Saussié D, Gratia R, Noulard E, Siron P (2014) The ROSACE case study: from simulink specification to multi/many-core execution. In: 20th IEEE real-time and embedded technology and applications symposium, RTAS 2014, Berlin, Germany, April 15–17, 2014. IEEE Computer Society, pp 309–318

Pagetti C, Forget J, Falk H, Oehlert D, Luppold A (2018) Automated generation of time-predictable executables on multicore. In: Proceedings of the 26th international conference on real-time networks and systems, RTNS'18, New York, NY, USA, Association for Computing Machinery, pp 104–113

Pellizzoni R, Betti E, Bak S, Yao G, Criswell J, Caccamo M, Kegley R (2011) A predictable execution model for cots-based embedded systems. In: 17th IEEE real-time and embedded technology and applications symposium, RTAS 2011, Chicago, Illinois, USA, 11–14 April 2011. IEEE Computer Society, pp 269–279

Rouxel B, Skalistis S, Derrien S, Puaut I (2019) Hiding communication delays in contention-free execution for spm-based multi-core architectures. In: Quinton S (ed) 31st Euromicro conference on real-time systems, ECRTS 2019, July 9–12, 2019, Stuttgart, Germany. LIPIcs, vol 133. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp 25:1–25:24

Schuh M, Maiza C, Goossens J, Raymond P, de Dinechin BD (2020) A study of predictable execution models implementation for industrial data-flow applications on a multi-core platform with shared banked memory. In: 41st IEEE real-time systems symposium, RTSS 2020, Houston, TX, USA, December 1–4, 2020. IEEE, pp 283–295

Senoussaoui I, Zahaf H-E, Lipari G, Benhaoua KM (2022) Contention-free scheduling of PREM tasks on partitioned multicore platforms. In: 2022 IEEE 27th international conference on emerging technologies and factory automation (ETFA), pp 1–8

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Rémi Meunier is a research engineer at Randstadt Digital. He obtained his PhD on the timing predictability of embedded real-time systems running on multi-core platforms in 2023.

Thomas Carle is a lecturer at IRIT-Université Toulouse 3. His research focus is on the timing predictability of embedded real-time systems, especially in parallel processors (multi/many-cores, GPUs), through the combination of hardware design, compilation, static analysis and static scheduling techniques.

Thierry Monteil is professor in computer science at INSA – University of Toulouse and researcher at IRIT. He has worked on parallel computing, scheduling, cloud resources management, autonomic middleware and Internet of Things architecture. He represents CNRS at OneM2M standardisation consortium and at Eclipse foundation. He has created the OM2M project and leads the OM2M eclipse open source project.