Memory-processor co-scheduling of AECR-DAG real-time tasks on partitioned multicore platforms with scratchpads

Ikram Senoussaoui, Giuseppe Lipari, Houssam-Eddine Zahaf, Mohammed Kamal Benhaoua

ScienceDirect
www.sciencedirect.com

Journal of Systems Architecture 00 (2024) 1–18

RIAI
www.elsevier.es/RIAI

Memory-processor co-scheduling of AECR-DAG real-time tasks on partitioned multicore platforms with scratchpads

Ikram Senoussaoui[a,b,*], Giuseppe Lipari[b], Houssam-Eddine Zahaf[c], Mohammed Kamal Benhaoua[a,d]

[a] *Oran1 University, LAPECI, Oran, Algeria*
[b] *Univ. Lille, CNRS, Inria, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France*
[c] *Nantes Université, École Centrale Nantes, IMT Atlantique, CNRS, INRIA, LS2N, UMR 6004, F-44000 Nantes, France*
[d] *University Of Mustapha Stambouli Mascara-Algeria*

Abstract

Multicore systems with core-level scratchpad memories offer appealing architectures for constructing efficient and predictable real-time systems. In this work, we aim to improve the usability of scratchpad memories and exploit their predictability to hide access latency to shared resources. We use a genetic algorithm to derive scheduling parameters for a set of directed acyclic task-graphs (DAGs). DAGs consist of dependent subtasks and their respective communications, following the Acquisition Execution Restitution (AER) model. Subtasks are partitioned onto the multicore platform while scheduling their memory requests and relative communications onto the shared buses, in order to prevent interference and ensure predictability. Specifically, all subtasks and communications are assigned appropriate intermediate offsets and deadlines to guarantee that they comply with the system's timing constraints. We conducted a large set of synthetic experiments to demonstrate the effectiveness of the proposed technique.

Keywords: Real-time systems, genetic-algorithm, scratchpad, multicore architecture, contention-free scheduling, DAG task

1. Introduction

Modern real-time applications, particularly those in the automotive domain, are becoming increasingly complex. They face the challenge of meeting strict real-time constraints while satisfying high performance requirements. Recent hardware platforms incorporate multiple cores, making them appealing candidates for meeting the requirements of these applications. As a result, modern real-time applications are typically structured as a collection of parallel and concurrent tasks, represented using the Directed Acyclic Graph (DAG) task model. This task model enables efficient utilization of the different cores and maximizes the platform utilization.

In a typical multicore platform, different cores share a complex memory hierarchy. Multiple tasks may have concurrent access to different memory subsystem components, leading to potential interference and contention. It is crucial to address the concurrent access to the memory to ensure the respect of timing constraints. Two approaches are explored in the literature on real-time systems to address these memory-related concerns. The first approach involves incorporating the potential delays caused by memory concurrency and contention into the worst-case execution time of tasks. This step is performed prior to the schedulability analysis, therefore, multiple combinations of execution patterns are considered, including those that might be impossible, increasing the pessimism. The second approach aims to avoid interference at the system design level by enforcing time isolation with time partitioning schemes like the Time-Division Multiple Access scheme (TDMA).

The in-between approach, which is the focus of this paper, aims to mitigate interference by regulating access to concurrent parts of the memory subsystem. Our approach seeks to avoid conflicts and contention by enabling memory-processor co-scheduling to meet the system's timing constraints using suitable application models such as the PRedictable Execution Model (PREM) Pellizzoni et al. (2011) or the Acquisition Execution Restitution Model (AER) Durrieu et al. (2014). In the latter, a task is modeled by three phases: acquisition (A), computation (E), and restitution (R). During the acquisition phase, task's data/instructions are loaded from the main memory into the core's local scratchpad memory. During the computation phase, the task achieves computations without any access to the main memory, therefore, it can be interleaved with the acquisition or restitution phases of the other tasks or with the computation phase of other tasks executing on different cores; in the restitution phase, the processed data are written back to the main memory. This execution model allows to improve the worst-case execution time estimates. This requires to use high-

*Corresponding Author

speed, explicitly managed memories like scratchpads rather than cache memories.

Scratchpad memories are becoming more and more popular in real-time embedded systems. A *scratchpad* is a small software-managed on-chip static RAM that has been widely accepted as an alternative to cache memory, as it offers better timing predictability compared to caches. The scratchpad memory is mapped into the address space of the processor, and is accessed whenever the address of a memory access falls within a predefined range. As mentioned, contrary to caches, the compiler and/or the programmer explicitly controls the allocation of instructions and data to the scratchpad memory. This operating principle makes the latency of each memory access, and thus program execution time, completely predictable. Significant effort has been invested in developing efficient static and dynamic allocation techniques for scratchpad memories Udayakumaran and Barua (2003); Suhendra et al. (2010); Yang et al. (2010); Kandemir et al. (2001). Many of the modern multicore platforms offer scratchpad memories, for example: NXP S32, Renesas R-car, STM Stellar, and Aurix Infineon platform series.

Although this execution model is increasingly popular, it is designed for non-dependent tasks, where each task is independent of the others. It can be complex to benefit from the parallel execution that multicore platforms offer at the task level because the granularity of these tasks can be excessively coarse. This coarse granularity can degrade system performance, especially for real-time tasks with high-performance requirements.

In this paper, we introduce a new task-graph model based on the AER model, which we call *Acquisition-Execution-Communication-Restitution-DAG (AECR-DAG)* model (see Figure 2). An AECR-DAG application is a Directed Acyclic Graph of communicating subtasks that must be executed on a multicore system within specified timing constraints (periods and deadlines). The subtasks represent computation phases (to be executed on cores) or memory phases (with local scratchpad memories or with the main shared memory). The graph edges represent precedence constraints between them. The AECR-DAG model is used along with scratchpad memories in order to eliminate contention on the bus.

The main goal of our work is to optimize the system's schedulability, and therefore we need to assign subtasks to cores, assign the scheduling parameters, and compute the worst-case latency of every task graph. This problem is a generalization of the task-to-core allocation problem, which itself is a variant of multiple knapsack Martello and Toth (1990); Chekuri and Khanna (2005), a well-known NP-complete problem.

In particular, the cost of communication between subtasks can increase the worst-case end-to-end latency of the DAGs, potentially jeopardizing schedulability. Therefore, it is crucial to minimize costly communications by assigning communicating subtasks to the same core. However, assessing schedulability for various allocations can be a highly time-consuming process, due to the exponential number of allocation possibilities. Therefore, the solution space for this problem is very large, comprising both the allocation search space and the scheduling search space.

In our approach, we decouple the problem of task-to-core allocation (inter-task communications) from the task schedulability. When we decouple these two problems, we observe that the design space for allocation is smaller than that for the intermediate deadline assignment problem and schedulability analysis.

In a first phase, we allocate sub-tasks to cores with the goal of minimizing the communication cost on the shared bus, but without checking the overall schedulability. We aim to identify a promising allocation candidate in an initial phase by the mean of Integer Linear Programming (ILP), which might likely allow to have schedulable system.

In a second phase, we tackle the problem of assessing the schedulability of the DAGs. For this, we need to assign intermediate deadlines and offsets to subtasks and conducting schedulability tests. Given that the second problem (intermediate deadline assignment) is more complex, we consider the use of a heuristic, specifically a *genetic algorithm*.

Contributions. We summarize our contributions as follow:

1. We extend the DAG task model to include main memory accesses and inter-core communications so to decouple the problem of scheduling on the buses from the core scheduling.
2. We partition subtasks on the different cores so to avoid the maximum of communication subtasks and to simplify the tasks. We propose an ILP formulation to solve the mapping problem.
3. We compute and assign intermediate offset and deadlines to subtasks of each DAG using a *genetic algorithm*.
4. We provide a large set of experimental evaluations showing the effectiveness of our algorithms.

Organization of the paper. The remainder of this paper is organized as follows. In the following section we review the state of the art. We present the hardware and task models in Section 3. Section 4 describes how computation subtasks are mapped to cores, the corresponding reduced DAG model and how our ILP model is built. In Section 5, we describe how the intermediate offsets and deadlines are assigned to subtasks. Results and simulations are described in Section 6. We draw the conclusions in Section 7.

2. Related work

Contention on memory resources was the subject of many research works Yao et al. (2012); Alhammad and Pellizzoni (2014); Maia et al. (2017); Rosen et al. (2007); Tabish et al. (2019). Some of them propose new execution models making use of pre-fetching techniques Rosen et al. (2007). It has been shown in Hoogeveen et al. (1996) that these techniques improve the cache/scratchpad locality and reduce in average worst-case execution times.

Another possibility to minimize contention in the shared memory resources is to decouple the memory requests from the actual application execution, so to guarantee that these requests are performed exclusively in isolation. This can easily

be ensured by using a Time Division Multiple Access (TDMA) based protocol to enforce timing isolation among all the tasks in the system. The AER model proposed in Durrieu et al. (2014) follows the same approach, with the aim of increasing the predictability of applications executing on COTS-based platforms.

The AER model is a generalization of the PREM model from Pellizzoni et al. (2011). It introduces two main advantages: (i) contention on memory resources is avoided by requiring memory phases to execute exclusively, by having only one memory phase (A or R) running at any time instance. (ii) the task's parallel execution in each core is allowed since the memory phases are decoupled from the computation phase.

The work we proposed in Senoussaoui et al. (2022) aimed to avoid contention for a set of tasks modeled using the PREM model. We proposed three novel approaches to avoid contention memory phases: (i) a task-level time-triggered approach, (ii) job-level time-triggered approach, and (iii) on-line scheduling approach. We compared the proposed approaches against the state of the art using a set of synthetic experiments in terms of schedulability and analysis time. Furthermore, we implemented the different approaches on an Infineon AURIX TC397 multicore microcontroller, and we validated the proposed approaches using sets of tasks extracted from benchmarks from the literature.

Several task models have been proposed to express data dependency within real-time task, most of them are based on DAGs, *e.g.*,Baruah et al. (2012); Melani et al. (2016); Zahaf et al. (2016, 2019). Nodes represent subtasks, and edges define communications and precedence constraints between nodes. In this manner, a subtask becomes ready for execution as soon as all its communications and precedence constraints are satisfied. In such model, memory transfers can take place on two different levels: (i) between cores and main memory and/or (ii) between cores. However, existing scheduling schemes for parallel real-time tasks assume data sharing or synchronization costs are included in the WCET of the subtasks. In our work, we propose a DAG task model that explicitly models all memory transfers in order to capture and manage contention at different levels.

Partitioned scheduling is generally more adapted to this type of application than global scheduling. In fact, in global scheduling tasks may migrate from one core to another at any time, and this implies large migration overheads, which are much greater in architectures with scratchpads since both needed data and code of the subtask have to migrate.

In partitioned scheduling, all subtasks of all DAGs are first allocated on cores, and then a separate scheduler is used for each core. Therefore, the problem is transformed into an allocation problem plus a number of separate and independent scheduling problems.

A number of partitioning algorithms that map dependent tasks to cores have been presented in the literature. Most of them are heuristics and operate on a single DAG task. They allocate each node of a DAG to a processor or a core. In Ben-Amor and Cucu-Grosjean (2022), a partitioning heuristic and graph reduction techniques are proposed for DAG tasks. They consider identical cores and fixed-priority preemptive schedulers. In this paper, we propose a ILP formulation to calculate an allocation that minimizes communication delay and allows us to eliminate costly communications.

The authors of Slim et al. (2020) have studied the probabilistic response time analysis of DAG tasks on multicore platforms using partitioned fixed-priority scheduling. They proposed a priority assignment algorithm at the subtask level to define the execution order between different nodes from the same graph in order to reduce the response time of the entire DAG task while considering communication times for the subtasks scheduled on different cores. Although this work tries to minimize the inter-subtask communications costs, they do not consider scratchpads nor their scheduling.

Another effective technique to schedule DAG tasks on multicore platforms is to assign intermediate deadlines and offsets to subtasks in order to enforce the precedence constraints. By completing the deadline and offset assignments, a DAG task is transformed into a set of independent subtasks. Two heuristic algorithms are popular: *Fair distribution* and *Proportional distribution*. These techniques were used in Zahaf et al. (2019); Houssam-Eddine et al. (2020); Zahaf et al. (2020); Qamhieh et al. (2013). Authors of Marinca et al. (2004) proposed *Fair Laxity Distribution* and *Unfair Laxity Distribution* heuristics for deadline assignment in distributed real-time systems and studied their impact on schedulability. An ILP formulation for the offsets and deadlines assignment problem is proposed in Wu et al. (2014).

Metaheuristic design space exploration techniques have gained popularity in the real-time systems scheduling. Authors in McLean et al. (2020) use a Simulated Annealing-based algorithm (SA) to generate static schedule tables by simulating Earliest Deadline First (EDF) scheduling parameterized by task offsets and local deadlines decided by SA. In Druetto et al. (2023), another algorithm based on Simulated Annealing was presented to partition tasks over a multicore architecture and assign them a priority value. Authors in Druetto et al. (2023) proposed a task and memory mapping approach based on Simulated Annealing of large size embedded applications over NUMA architecture. Mitra and Ramanathan (1993) proposed a genetic algorithm to non-preemptively schedule tasks with precedence constraints on multiprocessor platforms and compared the proposed algorithm with the Minimum-Laxity-First heuristic Ramamritham (1990). This approach considers the communication time between two tasks, but does not consider bus contention. Unlike Mitra and Ramanathan (1993), Monnier et al. (1998) presented a genetic algorithm implementation to solve a real-time non-preemptive scheduling problem for periodic tasks on a distributed hardware system and consider contentions for network access. The paper proposed by Oh and Wu (2004) presents and evaluates a new method for real-time task scheduling in multiprocessor systems. The authors aimed to minimize the number of processors required and the total tardiness of the tasks. The minimization is performed by a multi-objective genetic algorithm. Authors in Madureira et al. (2002) applied the genetic algorithm for assigning priorities and offsets to periodic tasks running on standard preemptive RTOS (Real-Time Operating System).

To the best of our knowledge, no work in the literature uses

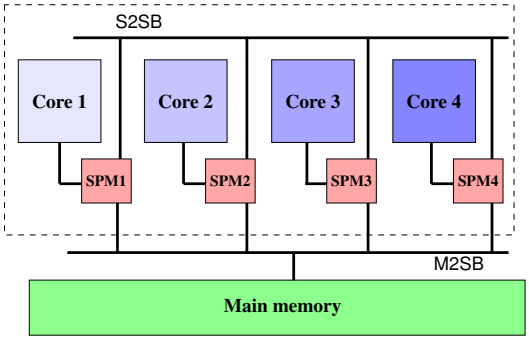


Figure 1: Multicore architecture featuring 4 cores, and its interconnection buses.

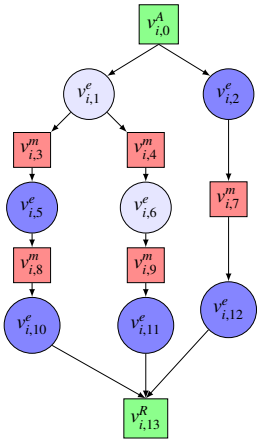


Figure 2: DAG task example, computation subtasks are mapped on a dual-core platform.

a genetic algorithm to compute DAG tasks scheduling parameters. The principle of this algorithm and how it works will be explained in Section 5.

3. System model

3.1. Architecture model

We consider a multicore platform $\mathcal{A}$ composed of m cores, i.e., $\mathcal{A} = \{\mathsf{p}_1, \cdots, \mathsf{p}_m\}$. Each core p_i has its own local scratchpad memory, onto which data and instructions are stored. All cores share a single main memory. Two types of memory copy operations are possible: (i) between scratchpad memories of different cores and (ii) between the main memory and a core's scratchpad memory. The two types of memory operations are performed by different buses: one bus, denoted as S2SB, interconnects all the scratchpad memories; a second separate bus, denoted as M2SB, interconnects the main memory and the scratchpads. The topology of our architectural model is illustrated in Figure 1. From a software point of view, we assume that all memories (main memory and local scratchpad memories) are directly accessible to all cores via different address spaces. The Infineon Aurix TC397 IT (2020) is an example of a real architecture that can be modeled as described above.

Memory copy operations are explicitly triggered by the software, therefore it is easier to schedule them. The proposed model allows us to reduce the complexity of the schedulability analysis by separating the memory operations between local and global memories.

3.2. Task model

We consider a task set $\mathcal{T} = \{\tau_1, \ldots, \tau_n\}$ consisting of n sporadic tasks. Each task $\tau_i \in \mathcal{T}$ is represented by a tuple $(\mathsf{G}_i, \mathsf{D}_i, \mathsf{T}_i)$, where G_i is a Directed Acyclic Graph (DAG) that describes the internal structure of τ_i, D_i is its end-to-end relative deadline, and T_i is its period. We consider a constrained deadline task set, that is $\mathsf{D}_i \leq \mathsf{T}_i$ for all tasks.

Each task-graph G_i is defined by $(\mathcal{V}_i, \xi_i)$, where $\mathcal{V}_i$ is a set of n_i *subtasks*, and ξ_i is the set of precedence constraints between them. A subtask $v_{i,j} \in \mathcal{V}_i$ can be one of four types (see Figure 2):

- an *acquisition subtask*, denoted as $v^A_{i,j}$;
- a *restitution subtask* denoted as $v^R_{i,j}$;
- a *communication subtask* denoted as $v^m_{i,j}$;
- a *computation subtask* denoted as $v^e_{i,j}$.

The upper index is omitted when referring to a subtask without any consideration of its type.

Edge $e(v_{i,j}, v_{i,k}) \in \xi_i$ represents the precedence constraint between subtasks $v_{i,j}$ and $v_{i,k}$, i.e., $v_{i,k}$ can not released before $v_{i,j}$ has completed its execution. Thus, $v_{i,k}$ is an *immediate successor* of $v_{i,j}$, and $v_{i,j}$ is an *immediate predecessor* of $v_{i,k}$.

We denote by $\mathsf{i\_succ}(v_{i,j})$ the set of all immediate successors of $v_{i,j}$ (i.e., $\mathsf{i\_succ}(v_{i,j}) = v_{i,k} \mid (v_{i,j}, v_{i,k}) \in \xi_i$). In the same way, we denote by $\mathsf{i\_pred}(v_{i,j})$ the set of all immediate predecessors of $v_{i,j}$ (i.e., $\mathsf{i\_pred}(v_{i,j}) = v_{i,k} \mid (v_{i,k}, v_{i,j}) \in \xi_i$).

A *communication subtask* is a memory copy operation that takes place between the scratchpad memories of the cores where its immediate predecessor and its immediate successor are allocated. Therefore, it has only one immediate successor and one immediate predecessor.

An *acquisition subtask* represents a memory copy operation from the main memory to one or more of the local scratchpads. Without loss of generality, in this paper we consider that a task-graph has only one acquisition subtask with no predecessors, and it is therefore the source node of the graph.

A *restitution subtask* represents a memory copy operation from one or more of the local scratchpads to the main memory. Without loss of generality, in this paper we consider that a task-graph has only one restitution subtask with no successors, and it is therefore the sink node of the graph.

A *computation subtask* represents code executing on one of the cores. While executing, the computation subtask can only access data and code on the corresponding local scratchpad memory. The immediate successors and immediate predecessors of a computation subtask are memory operations (either communication subtasks or acquisition or restitution subtasks). More generally, we consider that two subtasks of the same type cannot be immediate successors to each other.

Later on, after we allocate computation subtasks to cores, we will simplify the graph and allow two computation subtasks allocated on the same core to be directly linked by an edge without any communication needed between them.

A sequence of vertexes consisting of one computation subtask followed by a communication subtask followed by another computation subtask is called a *triplet*. We denote the triplet of subtasks $v_{i,j}, v_{i,k}, v_{i,h}$ as $\widehat{i : jkh}$.

Each subtask is characterized by its worst-case execution time, denoted by $\mathsf{C}_{v^e_{i,j}}$ for computation subtasks. When a subtask is an acquisition $\mathsf{C}_{v^A_i}$ (resp. restitution $\mathsf{C}_{v^R_i}$), its worst-case execution time represents an upper bound on the time required for the task to perform data transfers from (resp. to) the main memory and the scratchpad memories of the cores where its immediate successors (resp. predecessors) are allocated. If an acquisition (resp. restitution) has multiple successors (resp. predecessors), multiple memory copy operations may be performed by the subtask.

If a subtask is of the communication type, its worst-case execution time $\mathsf{C}_{v^m_{i,j}}$ represents an upper bound on the time required to perform data transfers between the scratchpad memories of the cores where its immediate predecessor and immediate successor are allocated.

Once a memory copy subtask starts its execution, it cannot be preempted. In contrast to memory copy subtasks, computation subtasks can be preempted, and they are partitioned among the platform cores.

We consider just one acquisition subtask (restitution subtask) in our model, in order to simplify the deadline assignment phase. If there are several data transfers for acquisition or for restitution, then we assume that these transfers are carried out sequentially (merged into one node A or R).

We define π^k_i as the k^{th} path of task τ_i. π^k_i is a sequence of vertexes, $\pi^k_i = \langle v_{i,j}, v_{i,j+1}, \cdots, v_{i,|\pi^k_i|}\rangle$ such that, $\forall j \in [0, |\pi^k_i|)$, $e(v_{i,j}, v_{i,j+1}) \in \xi_i$. The first subtask in a path represents the acquisition phase, while the last one represents the restitution phase. The set of all paths of task τ_i is denoted as Π_i. Figure 2 depicts a task-graph with three paths.

We define task utilization as the occupancy ratio of the task on all the shared resources (cores and communication buses). It is computed as follows:

$$\mathsf{u}(\tau_i) = \frac{1}{\mathsf{T_i}} \cdot (\mathsf{C}_{v^A_i} + \mathsf{C}_{v^R_i} + \sum_{v^m_{i,j} \in \mathcal{V}_i} \mathsf{C}_{v^m_{i,j}} + \sum_{v^e_{i,j} \in \mathcal{V}_i} \mathsf{C}_{v^e_{i,j}}) \quad (1)$$

The total $\mathsf{U}^{\mathsf{max}}$ utilization of the task set is computed as follows:

$$\mathsf{U}^{\mathsf{max}} = \sum_{\tau_i \in \mathcal{T}} \mathsf{u}(\tau_i) \quad (2)$$

We denote by $\mathcal{V}^M_i$ the set of all communication subtasks of task τ_i, and by $\mathcal{V}^C_i$ the set of all computation subtasks of task τ_i. When we describe a behavior that relates to a single task, its index might be removed to avoid overcharging the symbols.

4. DAG tasks allocation and transformation

In this section, we consider the problem of partitioning a set of Acquisition-Execution-Communication-Restitution-DAG (AECR-DAG) tasks on an identical core platform. Our approach consists of two distinct stages: the allocation stage and the schedulability analysis stage.

Computation subtasks are allocated to the different cores; inter-core communications are scheduled on the intercommunication bus; and acquisition and restitution phases are scheduled on the memory bus. The subtask-to-core allocation process has a significant impact on inter-core communications. Communicating subtasks that are allocated on the same core will not generate traffic on the shared inter-core communication bus, while those allocated onto different cores may generate traffic that may jeopardize the schedulability. Rather than using classical bin packing heuristics that optimize core utilization, such as Best-Fit (BF) that maximizes utilization per core or Worst-Fit (WF) that favors load balancing, we use Integer Linear Programming (ILP) for the subtask-to-core allocation. The goal of the ILP is to reduce the workload on the inter-core communication bus by favoring the allocation of communicating subtasks to the same core when possible or desirable.

Definition 1 (null communication)**.** *A communication subtask $v^m_{i,j}$ is called a* null-communication subtask *when its only immediate predecessor subtask and its only immediate successor subtask are allocated to the same core p.*

A null-communication subtask does not generate traffic on the shared inter-core bus, therefore, its worst-case execution time can be set to 0, and hence it can be eliminated from the graph.

Definition 2. *A reduced task, denoted as $\overline{\tau_i}$ of task τ_i is a DAG where all the null-communication subtasks are removed.*

Let $v^m_{i,k}$ be a null-communication subtask, and let $\widehat{i : jkh}$ be the triplet of subtasks consisting of the immediate predecessor of $v^m_{i,k}$ namely $v^m_{i,j}$ and its immediate successor, namely $v^m_{i,h}$. Then in the reduced task $\overline{\tau_i}$, $v^m_{i,k} \notin \overline{\mathcal{V}_i}$, edges $e(v_{i,j}, v_{i,k}) \notin \overline{\xi_i}$ and $e(v_{i,k}, v_{i,h}) \notin \overline{\xi_i}$, and there exist an edge $e(v_{i,j}, v_{i,h}) \in \overline{\xi_i}$, where $\overline{\mathcal{V}_i}$ and $\overline{\xi_i}$ are respectively the nodes set and edges set of $\overline{\tau_i}$.

Therefore, in the reduced task we allow a computational subtask to be an immediate successor of another.

To produce a reduced task, we need only to replace all triplets containing a null-communication subtask with a simple edge between the two computation subtasks of the triplet.

We denote by $\mathbf{cost}(\overline{\tau})$ the ratio of the communication workload that the reduced task $\overline{\tau}$ will require on the share inter-core bus. It can be computed as follows:

$$\mathsf{cost}(\overline{\tau}) = \sum_{v \in \mathcal{V}^M(\overline{\tau})} \frac{\mathsf{C}_v}{\mathsf{T}(\overline{\tau})} \quad (3)$$

Example 1. *In this example, subtasks are colored according to core where they are allocated (Figure 3). Therefore, we have four null-communication triplets: $\mathcal{V}^1_\tau(p_1) = \{v^e_5, v^e_{10}\}$, $\mathcal{V}^2_\tau(p_1) =$*

$\{v^e_{11}\}$, $\mathcal{V}^3_\tau(p_1) = \{v^e_2, v^e_{12}\}$ *and* $\mathcal{V}^4_\tau(p_2) = \{v^e_1, v^e_6\}$, *as indicated by the red boxes in the left-hand side graph.*
In the right-hand side graph, the memory subtasks of the different null-communication triplets have been dropped as their execution time is set to 0, resulting in a reduced task denoted as $\bar{\tau}$. *In this example, the execution time of* v^m_8 *is set to 0 for* $\mathcal{V}^1_\tau(p_1)$, *and the execution time of* v^m_7 *is set to 0 for* $\mathcal{V}^3_\tau(p_1)$ *and also the execution time of* v^m_4 *is set to 0 for* $\mathcal{V}^4_\tau(p_2)$.

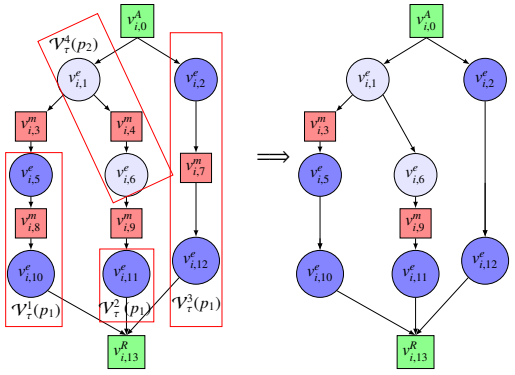


Figure 3: DAG task transformation.

Our ILP formulation has the objective to reduce the total communication cost over all tasks. In the remainder of this section, we detail the ILP formulation to optimally reduce the DAG tasks.

4.1. Decision variables and objective function

Let $\mathsf{a}^{\mathsf{p}}_j$ be a binary decision variable [1] to express that computation subtask $v_{i,j}$ is allocated to core p, *i.e*:

$$\mathsf{a}^{\mathsf{p}}_j = \begin{cases} 1, & \text{if } v_{i,j} \text{ is mapped to core } \mathsf{p} \\ 0, & \textit{otherwise} \end{cases} \tag{4}$$

Let us consider the triplet $\widehat{i : jkh}$. If v_{ij} and v_{ih} are allocated to the same core, the execution time of subtask v_{ik} is set to zero as it does not generate traffic on the communication bus. Therefore, for each triplet $\widehat{i : jkh}$, we define the decision variable $\mathsf{cost}(\widehat{i : jkh})$ as follows:

$$\mathsf{cost}(\widehat{i : jkh}) = \begin{cases} 0, & \text{if } \forall \mathsf{p},\ \mathsf{a}^{\mathsf{p}}_j = \mathsf{a}^{\mathsf{p}}_h \\ C(v_{ik}), & \text{otherwise} \end{cases} \tag{5}$$

The objective function consists in minimizing the total communication cost of all tasks by summing over all triplets $\widehat{i : jkh}$

$$\text{Minimize} \sum_{\tau_i \in \mathcal{T}} \sum_{\widehat{i:jkh} \in \Delta_i} \mathsf{cost}(\widehat{i : jkh}) \tag{6}$$

We will illustrate further in this section the different techniques to linearise these constraints.
Constraints.

[1] Please notice that this decision variable is generated only for the computation subtasks

1. Communication cost constraint. First, we describe the evaluation of the cost variables. The conditional construct and the different inequalities involved in computing the variable cost require linearisation.
For every triplet $\widehat{i : jkh}$ and for every core p, we introduce two artificial binary decision variables: $x(\widehat{i : jkh}, \mathsf{p})$ and $y(\widehat{i : jkh}, \mathsf{p})$. These variables indicate whether subtasks v_{ij} and v_{ih} are allocated to the same core, and are defined as follows:

$$x(\widehat{i : jkh}, \mathsf{p}) = \begin{cases} 1 & \text{if } a^{\mathsf{p}}_j > a^{\mathsf{p}}_h \\ 0, & \text{otherwise} \end{cases} \tag{7}$$

$$y(\widehat{i : jkh}, \mathsf{p}) = \begin{cases} 1 & \text{if } a^{\mathsf{p}}_j < a^{\mathsf{p}}_h \\ 0, & \text{otherwise} \end{cases} \tag{8}$$

If x or y are equal to 1, then the subtasks are not allocated to the same core. If both x and y are equal to 0, then the subtasks are allocated to the same core. x and y cannot both be equal to 1 at the same time, therefore we compute the cost as follows:

$$\mathsf{cost}(\widehat{i : jkh}) = C(v_k) \cdot \sum_{\mathsf{p} \in \mathcal{A}} (x(\widehat{i : jkh}, \mathsf{p}) + y(\widehat{i : jkh}, \mathsf{p})) \tag{9}$$

Once again, the conditional construct to compute $x(\widehat{i : jkh}, \mathsf{p})$ requires linearisation. First, we express the inequality in Equation (7) as follows:

$$a^{\mathsf{p}}_j - a^{\mathsf{p}}_h - 1 \geq 0$$

Let M be a very large constant. We linearise Equation (7) as follows:

$$a^{\mathsf{p}}_j - a^{\mathsf{p}}_h - 1 + M - M \cdot x(\widehat{i : jkh}, \mathsf{p}) \geq 0$$
$$a^{\mathsf{p}}_j - a^{\mathsf{p}}_h - 1 - M \cdot x(\widehat{i : jkh}, \mathsf{p})) < 0$$

Similarly, we linearise Equation (8) as follows :

$$a^{\mathsf{p}}_h - a^{\mathsf{p}}_j - 1 + M - M \cdot y(\widehat{i : jkh}, \mathsf{p}) \geq 0$$
$$a^{\mathsf{p}}_h - a^{\mathsf{p}}_j - 1 - M \cdot y(\widehat{i : jkh}, \mathsf{p})) < 0$$

2. Allocation constraint. In order to ensure that a subtask is allocated to one and only one core, we generate the following constraints:

$$\forall \tau \in \mathcal{T}, \forall v \in \mathcal{V}(\tau) : \sum_{\mathsf{p} \in \mathcal{A}} a^{\mathsf{p}}_v = 1 \tag{10}$$

3. Core workload constraint. We can enforce the utilization per core to not exceed a given bound $U^{\max}$ using the following constraints:

$$\forall \mathsf{p} \in \mathcal{A}, \sum_{\tau} \sum_{v \in \mathcal{V}^C(\tau)} a^{\mathsf{p}}_v \cdot \frac{C(v)}{\mathsf{T}_\tau} \leq U^{\max} \tag{11}$$

In the following listing, we report our complete ILP formulation.

$$\text{Minimize} \sum_{\tau_i \in \mathcal{T}} \sum_{\widehat{i:jkh} \in \Delta_i} \text{cost}(\widehat{i:jkh})$$

under the constraints:

$$\forall \tau \in \mathcal{T}, \forall \widehat{i:jkh} \in \Delta_i :$$

$$\text{cost}(\widehat{i:jkh}) = C(v_k) \cdot \sum_{\mathsf{p} \in \mathcal{A}} (x(\widehat{i:jkh}, \mathsf{p}) + y(\widehat{i:jkh}, \mathsf{p}))$$

$$\forall \tau \in \mathcal{T}, \forall \widehat{i:jkh} \in \Delta_i :$$

$$a_j^{\mathsf{p}} - a_h^{\mathsf{p}} - 1 + M - M \cdot x(\widehat{i:jkh}, \mathsf{p}) \geq 0$$

$$a_j^{\mathsf{p}} - a_h^{\mathsf{p}} - 1 - M \cdot x(\widehat{i:jkh}, \mathsf{p})) < 0$$

$$a_h^{\mathsf{p}} - a_j^{\mathsf{p}} - 1 + M - M \cdot y(\widehat{i:jkh}, \mathsf{p}) \geq 0$$

$$a_h^{\mathsf{p}} - a_j^{\mathsf{p}} - 1 - M \cdot y(\widehat{i:jkh}, \mathsf{p})) < 0$$

$$\forall \tau \in \mathcal{T}, \forall v \in \mathcal{V}(\tau) : \sum_{\mathsf{p} \in \mathcal{A}} a_i^{\mathsf{p}} = 1$$

$$\forall \mathsf{p} \in \mathcal{A}, \sum_{\tau} \sum_{v \in \mathcal{V}(\tau)} a_v^{\mathsf{p}} \cdot \frac{C(v)}{\mathsf{T}_\tau} \leq U^{\max}$$

Finally, the ILP is submitted to the CPLEX ILP-solver IBM (1987). When subtask-to-core mapping is computed, reduced tasks are derived by removing the null-communication subtasks.

5. Deadline based DAG memory-processor co-scheduling

The second stage of our approach is memory and processor co-scheduling. This work considers partitioned scheduling, where subtasks are assigned to different cores and scheduled using a fully preemptive EDF scheduler per core. Acquisition and restitution subtasks are scheduled non-preemptively on the memory-to-scratchpad bus (M2SB), while communication subtasks are scheduled on the inter-core bus (S2SB). Scheduling on the system's buses is achieved using a non-preemptive EDF scheduler. To simplify dealing with precedence constraints, we impose intermediate offsets and deadlines on each subtask. In this way, precedence constraints are automatically respected if each subtask is activated after its offset and completes no later than its assigned deadline.

Several techniques have been proposed in the real-time systems literature to assign intermediate deadlines, such as *fair* and *proportional* deadline assignments. However, these techniques have been designed for the scheduling of computation tasks only and are not tuned to handle delays that may occur due to contention on the shared communication buses. In this section, we present our approach to assign intermediate deadlines to both memory and computation subtasks, in order to respect the system's timing constraints and take into account the co-scheduling of both types of subtasks.

In the following, every subtask will be additionally characterized by its *intermediate deadline* d_v and *offset* ϕ_v. The offset

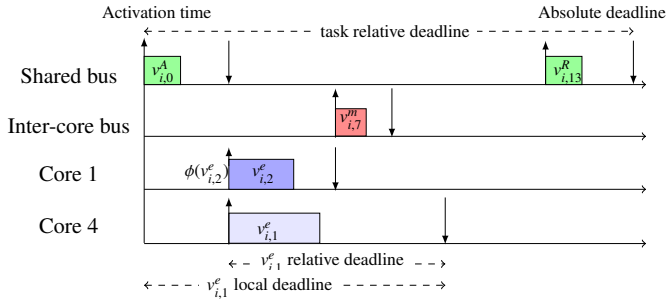


Figure 4: Example of offset and local deadline.

of a subtask is the distance between the activation of the task-graph and the activation of the subtask. The intermediate deadline of a subtask represents its relative deadline with respect to its offset. We define the subtask *local deadline* dl_v as the sum of its intermediate deadline and its offset. Figure 4 illustrates the relationship between the activation, end-to-end deadline, intermediate deadline, offset, and local deadline of a subtask.

In this context, the activation time of the acquisition subtask corresponds to the activation of the task itself. The local deadline of a subtask represents the interval between the task-graph activation and the subtask's absolute deadline.

One strategy to solve the problem of assigning intermediate deadlines is to exhaustively search among all possible intermediate deadline combinations. While this method provides an optimal solution, it suffers from a high computational complexity. In our previous work on the scheduling of PREM tasks Senoussaoui et al. (2022), we proposed a binary search-based method to compute and assign intermediate deadlines to memory phases, where each task consists only of a memory subtask and a computation subtask. However, it is not straightforward to extend this approach to DAG tasks, as the approach proposed in Senoussaoui et al. (2022) already has a high complexity for a simpler problem. Therefore, in this section, we propose to use a genetic algorithm (GA) to explore the assignments of intermediate deadlines to subtasks. We will describe our approach in the rest of this section.

5.1. Fair and proportional deadline assignment

First, we review the *fair* and *proportional* deadline assignment techniques. The idea is to divide the slack time along a path π_i^k in the graph between all the subtasks of the path. We consider paths in decreasing order of their cumulative execution time, therefore, we start with the critical, *i.e.* having the largest cumulative execution time.

We first define the slack function $\mathsf{Sl}(\pi_i^k, \mathsf{D_i})$ along path π_i^k of τ_i as:

$$\mathsf{Sl}(\pi_i^k, \mathsf{D_i}) = \mathsf{D_i} - \sum_{v \in \pi_i^k} \mathsf{C}_v \tag{12}$$

- **Fair distribution:** distribute slack as the ratio of the original slack by the number of subtasks along the path:

$$\text{calculate\_share}(v_{i,j}, \pi_i^k) = \frac{\mathsf{Sl}(\pi_i^k, \mathsf{D_i})}{|\pi_i^k|} \tag{13}$$

- **Proportional distribution:** distribute slack according to the contribution of the subtask execution time in the path:

$$\text{calcule\_share}(v_{i,j}, \pi_i^k) = \frac{\mathsf{C}_{v_{i,j}}}{\mathsf{C}(\pi_i^k)} \cdot \mathsf{Sl}(\pi_i^k, \mathsf{D_i}) \qquad (14)$$

where $\mathsf{C}(\pi_i^k)$ represents the total cumulative execution time of the subtasks in π_i^k.

Once the relative deadlines of the subtasks along the critical path have been assigned, we select the next path in order of decreasing cumulative execution time, and assign the deadlines to the remaining subtask by appropriately subtracting the already assigned deadlines. The complete procedure is not reported here and can be found in Wu et al. (2014).

5.2. GA-based intermediate deadline assignment

In this paper, we use a genetic algorithm to assign intermediate deadlines to memory and computation subtasks for a set of reduced tasks $\overline{\mathcal{T}}$. In a genetic algorithm, a population of candidate solutions (called individuals) is evolved towards better solutions. The goal is to move from unschedulable solutions to at least one schedulable solution.

Each candidate solution has a set of *chromosomes*, which in our case is the task set with intermediate deadlines assigned, that can be mutated and crossed over. The evolution process starts from multiple solutions, called the initial population, each having intermediate deadlines generated randomly.

The iterative process then evaluates the **fitness** function for every individual in the population. In each generation, a portion of the existing population is selected to reproduce and create a new generation through three operations: selection, crossover, and mutation. The new generation of candidate solutions is then used as input for the next iteration. The algorithm terminates when a maximum number of generations has been produced or a schedulable intermediate deadline assignment task set is found.

In the following, we describe the general structure of the genetic algorithm, the representation of a solution, the generation of the initial population, the fitness function, the selection, the crossover, and the mutation operations.

Algorithm 1 presents our approach for assigning intermediate deadlines. The algorithm starts by generating the initial population (Line 4). It goes through several iterations until a schedulable task set is found, namely: (i) population evaluation (Line 6), (ii) selection (Line 8), (iii) crossover (Line 9), and (iv) mutation (Line 10). If the algorithm finds a feasible schedule, it terminates with SUCCESS, otherwise, if a maximum number of iterations is reached, it aborts with FAIL.

5.2.1. Individual representation

Each individual in the genetic algorithm consists of a set of subtasks, each with an intermediate deadline assigned. In this work, we represent an individual as an ordered list of pairs, where each pair consists of a subtask and its corresponding local deadline. In our approach, we only consider subtasks that are part of reduced tasks while constructing the ordered list of subtask-deadline pairs for an individual.

Algorithm 1 GA intermediate deadlines assignment

```
1:  function DeadlinesGA(𝒯, p_size)        ▷ The DAG task set, population size
2:      found = false
3:      𝒫_l ← ∅                            ▷ The initial population
4:      ip_generation(𝒯, p_size)           ▷ The initial population generation
5:      while (not (found)) do
6:          found = ga_evaluation(𝒫_l, 𝒜)
7:          if (not found) then
8:              selection(𝒫_l)
9:              crossover(𝒫_l, η_cr)
10:             mutation(𝒫_l, η_mu)
11:         else
12:             return SUCCESS             ▷ If schedulable at all levels
13:         end if
14:     end while
15:     return FAIL
16: end function
```

Definition 3. *Individual An* individual *is denoted as*

$$\begin{aligned}\mathsf{ind} = (&\langle v_{1,1}, dl_{1,1}\rangle, \langle v_{1,2}, dl_{1,2}\rangle, \cdots,\\ &\langle v_{1,n_1}, dl_{1,n_1}\rangle, \langle v_{2,1}, dl_{2,1}\rangle, \cdots,\\ &\langle v_{m,n_m}, dl_{m,n_m}\rangle)\end{aligned}$$

where:

- *Subtasks of the same task are a sub-sequence of* ind
- *Subtasks of the same task are ordered in a topological order;*
- $dl_{i,j}$ *represents the local deadline of subtask* $v_{i,j}$*;*
- *All individuals present the same subtasks order.*

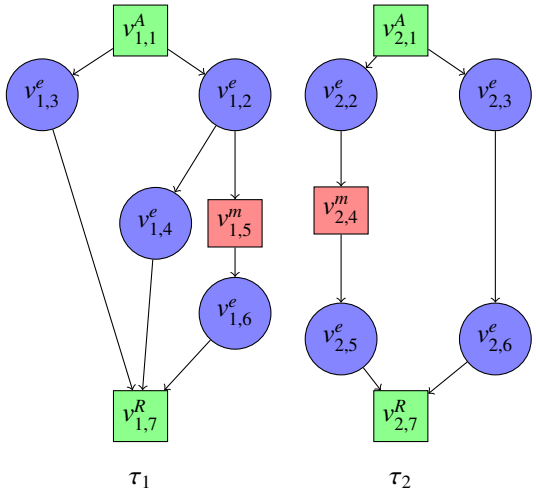


Figure 5: Example of an individual.

Example 2. *In Figure 5, we present a task set composed of two tasks,* τ_1 *and* τ_2*. The former has an end-to-end deadline of 50, while the latter has an end-to-end deadline of 80. Each task is comprised of seven subtasks. Table 1 shows the representation of an individual in our genetic algorithm.*

The subtasks in an individual are sorted according to their topological order. For example, subtask $v^e_{1,3}$ *appears earlier in*

v	dl_v
$v^A_{1,1}$	10
$v^e_{1,3}$	40
$v^e_{1,2}$	25
$v^m_{1,5}$	35
$v^e_{1,4}$	40
$v^e_{1,6}$	40
$v^R_{1,7}$	50
$v^A_{2,1}$	20
$v^e_{2,2}$	30
$v^e_{2,3}$	45
$v^m_{2,4}$	50
$v^e_{2,5}$	70
$v^e_{2,6}$	70
$v^R_{2,7}$	80

Table 1: Example of Individual

the individual than $v^e_{1,4}$, $v^m_{1,5}$, and $v^e_{1,6}$, while $v^e_{1,6}$ appears earlier than $v^R_{1,7}$. It should be noted that we consider local deadlines, and the restitution subtask's local deadline is always equal to the task's end-to-end deadline.

5.2.2. Initial population generation

The first step of a genetic algorithm is the generation of an initial population. It has been recognized that if the initial population provided to the genetic algorithm is of "high quality", the algorithm is more likely to find a sub-optimal solution Zitzler et al. (2000); Burke et al. (2004); Diaz-Gomez and Hougen (2007). Algorithm 2 summarizes the different steps of our initial population generation process.

Algorithm 2 Initial population generation

```
1:  function IP_GENERATION(T̄, p_size)
2:      P_l ← ∅                                          ▷ The initial population
3:      assigned = true
4:      while |P_l| < p_size do     ▷ p_size is the maximum number of generations
5:          S ← ∅                                        ▷ An individual
6:          for τ̄_i ∈ T̄ do
7:              assigned = inter_dline(τ̄_i)              ▷ Assign τ̄_i intermediate
8:                                                       ▷ deadlines
9:              if !assigned then
10:                 break
11:             end if
12:             add τ̄_i intermediate deadlines to S
13:         end for
14:         if assigned then          ▷ If all subtasks have assigned a deadline
15:             if S ∉ P_l then
16:                 add S to P_l
17:             end if
18:         end if
19:     end while
20:     return P_l
21: end function
```

For each task $\overline{\tau_i} \in \overline{\mathcal{T}}$, the algorithm invokes the `inter_dline` function (Line 7) to assign intermediate deadlines to subtasks. Within this function, all task paths are sorted by non-increasing cumulative execution time. For each path, the `random_dlines` function (Algorithm 3) is invoked in its turn to distribute randomly the slack among all subtasks on that path. The `inter_dline` function aborts on FAIL if slack sharing fails on at least one path, otherwise, it ends on SUCCESS. If the intermediate deadline assignment succeeds for all tasks in $\overline{\mathcal{T}}$, the under generation individual denoted as $\mathcal{S}$ is inserted into the population.

The share of each subtask that has not yet been assigned a deadline on a given path is computed in Algorithm 3. We denote the set of subtasks that have not yet been assigned a deadline by $\mathcal{T}_{nd}$. The intermediate deadline assignment is divided into two parts.

In Part 1, Algorithm 3 computes the intermediate deadlines of subtasks that have an offset and where at least one of their immediate successors has been assigned an offset (line 3). If multiple immediate successors are found with different offsets, the algorithm computes the non-null minimum offset. Intermediate deadlines for these subtasks are assigned using Algorithm 4. The intermediate deadline of each subtask is computed as the difference between its offset and the minimum offset of its immediate successor. Each time a deadline is assigned to a subtask, Algorithm 4 modifies: (i) its local deadline, (ii) the offsets of its immediate successors, (iii) and removes it from $\mathcal{T}_{nd}$ (lines 9-11).

Algorithm 3 computes the intermediate deadlines for the rest of the subtasks in $\mathcal{T}_{nd}$ in Part 2. It computes the slack time on the analyzed path, generates a set of random values whose sum is equal to 1 (line 6) and uses each value to compute the intermediate deadline of a subtask as the sum of $\mathsf{SI} \cdot \mathtt{U[v]}$ and the subtask worst-case execution time (line 9). The offset of each subtask is computed as the maximum local deadline of its immediate predecessors (except the offset of the acquisition subtask, which is always equal to 0).

Algorithm 3 Compute the intermediate deadlines on a path

```
1:  function INTER_DLINES(π_i^k)
2:      T_nd ← compute_contributors(π_i^k)
3:      if !non_zero_offset_subtasks_dlines(T_nd, π_i^k) then
4:          return FAIL
5:      end if
6:      SI ← compute_slack(π_i^k)              ▷ Compute the slack in path π_i^k
7:      if SI < 0 then
8:          return FAIL
9:      end if
10:     U ← random_rates()
11:     for v ∈ π_i^k do
12:         if v ∈ T_nd then                                ▷ v has no deadline
13:             dline = C_v + (SI · U[v])
14:             d_v ← dline
15:             dl_v ← φ_v + dline
16:             update_isucc_offset(v)              ▷ update v's immediate
17:                                                 ▷ successors offset
18:             remove(v, T_nd)
19:         end if
20:     end for
21:     return SUCCESS
22: end function
```

Definition 4 (Valid assignment)**.** *An intermediate deadline assignment $\Omega(\tau_i)$ of a task-graph τ_i is valid if:*

1. *the intermediate deadline of each subtask is greater than or equal to its worst-case execution time:* $\forall v \in \mathcal{V}_i,\ d_v \geq \mathsf{C}_v$;
2. *and the local deadline of the restitution subtask is less than or equal to the end-to-end deadline of* τ_i : $dl_{v^R} \leq \mathsf{D_i}$.

Lemma 1. *Consider an AECR-DAG task (resp. AECR-reduced DAG task)* τ_i. *The intermediate deadline assignment* $\Omega(\tau_i)$ *computed by function* random_dlines *is valid.*

Proof. The deadline of a subtask is computed either in Part 1 or in Part 2 of Algorithm 3. In the first part, the deadline of the subtask is assigned only if it is greater than or equal to its worst-case execution time, otherwise, it is computed in the second part of the algorithm as $\mathsf{C}_v + (\mathsf{Sl} \cdot \mathtt{U[v]})$, ensuring Condition 1 of Definition 4. Moreover, the sum of the rates U generated in part 2 of Algorithm 3 is equal to 1; rates are by definition positive values. The slack in its turn, is positive, otherwise, the schedulability fails. The offset of each subtask is computed as the maximum among all predecessors offset, therefore, the local deadline of the restitution subtask cannot be greater than the task end-to-end deadline, confirming Condition 2 of Definition 4. □

Algorithm 4 Non zero offset subtasks deadline assignment

```
1:  function NON_ZERO_OFFSET_SUBTASK_DLINES(&T_nd, p_i)
2:      for v ∈ p_i do
3:          if v ∈ T_nd then                ▷ The subtask deadline is not yet computed
4:              min = min_offset(i_succ(v))     ▷ get the minimum non-null
5:                                              ▷ offset of the immediate successors of v
6:              if φ_v > 0 and min > 0 then
7:                  dline = min − φ_v
8:                  if dline ≥ C_v then
9:                      d_v ← dline
10:                     dl_v ← φ_v + dline
11:                     update_isucc_offset(v)      ▷ update v's immediate
12:                                                 ▷ successors offset
13:                     remove(v, T_nd)
14:                 else
15:                     return FAIL
16:                 end if
17:             end if
18:         end if
19:     end for
20:     return SUCCESS
21: end function
```

5.3. Evaluation Strategy

The most important step of a genetic algorithm is the evaluation of the population. In this section, we assume that all computation subtasks have been allocated (partitioned) on the platform's cores and that subtasks have already been assigned offsets and intermediate deadlines. We apply the processor demand criterion Baruah et al. (1990) to evaluate each individual in the population. Algorithm 5 summarizes and clarifies our evaluation strategy.

The algorithm takes as input the generated population and an empty set $\mathcal{S}$. Each individual ψ_i in the input population needs to be awarded a score to indicate how close it is to meet the overall schedulability. This score is called the *fitness score* and it is calculated by the fitness function (lines 3-4), which will be detailed later in Section 5.3.2.

Algorithm 5 Population evaluation

```
1:  function POPULATION_EVALUATION(P_l, S)          ▷ The population
2:      for s ∈ P_l do
3:          score<A, S2SB, M2SB> = dbf_dag_analysis(s, A)
4:          if (score<A, S2SB, M2SB> == 0) then
5:              return s                            ▷ A solution is found
6:          else
7:              f_score = α_1 · score(A) + α_2 · score(S2SB) + α_3 · score(M2SB)
8:              s.set_score(f_score)                ▷ Set the score of s
9:              S ← s                               ▷ Add s to S
10:         end if
11:     end for
12:     return
13: end function
```

5.3.1. Schedulability of task-graphs

In this paper, we consider a system of sporadic task graphs $\mathcal{T}$. When an instance of a task is activated, its subtasks are activated with an offset relative to the activation of the task-graph. To analyse the schedulability of the system, we proceed by analysing the schedulability on each processor and on the two communication buses. If all subtasks respect their deadlines, then the entire system is schedulable.

Definition 5. *We denote by* $\Lambda_{\mathsf{p}}(\mathcal{T})$ *the subset of subtasks of* $\mathcal{T}$ *allocated on core* p. *By extension,* $\Lambda_{\mathsf{S2SB}}(\mathcal{T})$ *is the set of all communication subtasks, and* $\Lambda_{\mathsf{M2SB}}(\mathcal{T})$ *is the set of memory subtasks.*

To analyse the schedulabity on each processor and on each bus, we use the *Demand Bound Criterion* Baruah et al. (1990). The method consists in computing the *demand bound function* (dbf) in any interval t, and checking that it never exceeds the length of the interval. The dbf is computed as the maximum cumulative worst-case execution time of all jobs (instances of subtasks) having their arrival time and absolute deadline within any interval of time of length L.

The original demand bound analysis of Baruah et al. (1990) only considers sporadic tasks. Instead, subtasks belonging to the same task-graph have offsets with respect to each other. Therefore, we use the approximated method of Pellizzoni and Lipari (2005) to compute the dbf of a task-graph on a given core.

The dbf at L on core $\mathsf{p} \in \mathcal{A}$ of a task-graph can be computed as follows Pellizzoni and Lipari (2005):

$$\mathsf{dbf}\,(\tau_i, \mathsf{L}, \mathsf{p}) = \max_{\forall v_{i,j} \in \Lambda_{\mathsf{p}}(\mathcal{T})} \left\{ \sum_{v_{i,k} \in \Lambda_{\mathsf{p}}(\mathcal{T})} \left(\left\lfloor \frac{\mathsf{L} - \overline{\phi}_{\mathsf{k,j}} - \mathsf{d}_{\mathsf{v_{i,k}}}}{\mathsf{T_i}} \right\rfloor + 1 \right)_0 \cdot \mathsf{C}_{v_{i,k}} \right\} \tag{15}$$

where $\overline{\phi}_{k,j} = (\phi_{v_{i,k}} - \phi_{v_{i,j}}) \mod \mathsf{T_i}$. To understand Equation (15), consider that the subtasks of a task τ_i are activated with an offset relative to the arrival of the first subtask. Therefore, we need to

align the offset of one subtask to the beginning of an interval of length L and compute the workload generated in the interval. We do this for every subtask, and then we take the maximum. The resulting dbf is an upper bound to the actual dbf in that interval.

Theorem 1. *Given a set of subtasks allocated on core* p*, the subtasks are schedulable if their cumulative utilisation is less than 1, and*

$$\forall L \leq L^* \quad \sum_{\tau_i \in \mathcal{T}} \mathrm{dbf}\,(\tau_i, L, p) \leq L \tag{16}$$

where L^* *is the first idle time on core* p.

Proof. The proof descends directly from the proof of Theorem 2 in Pellizzoni and Lipari (2005): it suffices to notice that subtasks belonging to different task graphs have no offset relationship between each other, while subtasks belonging to the same task-graph are subject to offsets. □

Let now consider the schedulability on the buses. In our model, a task-graph may contain memory nodes that represent either the A/R phases or the communication phases, and these nodes are executed non-preemptively: e.g. once a communication subtask starts, it completes the memory transfer without being interrupted by other communication subtasks. Therefore, we have to take into account a blocking time in the schedulability analysis. This can be done by extending Theorem 6 in Pellizzoni and Lipari (2005), which deals with blocking time due to mutually exclusive resources.

Theorem 2. *Given a set of memory subtasks to be scheduled on* bus*, the subtasks are schedulable if their cumulative utilisation is less than 1, and:*

$$\forall L < L^* \quad \sum_{\tau_i \in \mathcal{T}} \mathrm{dbf}\,(\tau_i, L, \mathrm{bus}) + B\,(\tau_i, L, \mathrm{bus}) \leq L \tag{17}$$

where:

$$B\,(\tau_i, L, \mathrm{bus}) = \max\{\, C_{v_{j,k}} \mid \forall j \neq i, v_{j,k} \in \Lambda_{\mathrm{bus}}(\mathcal{T}) \wedge d_{v_{i,j}} > L \,\}. \tag{18}$$

is the blocking time due to non-preemptive scheduling.

Proof. Since subtasks executed completely inside any interval of length L can be blocked only once by subtasks that started before the beginning of the interval and have deadlines after the end of the interval, the maximum blocking time can not be longer than the maximum worst-case execution time among all subtasks having a deadline greater than L.
Now, by contradiction. Let v be the first subtask to miss a deadline at time t_2 and let $t_1 < t_2$ be the last instant before t_2 when there is an idle time or a subtasks with absolute deadline at or before t_2 is executed. Then, from the properties of EDF it follows that :

- in interval $[t_1, t_2]$ there is no idle time;
- at most one subtask with deadline greater than t_2 can execute (the blocking subtask)
- the rest of the interval is executed by subtasks with arrivals no earlier than t_1 and deadline no later than t_2.

Since a subtask has missed its deadline, then the cumulative demand in $[t_1, t_2]$, including the blocking subtask, has exceeded the length of the interval $L = (t_2 - t_1)$ and this in contradiction with the hypothesis. □

Theorem 3. *Given a task set* $\mathcal{T}$*, and a platform* $\mathcal{A}$*, where all subtasks have been assigned to cores, and where:*

- *the computation subtasks are scheduled on their assigned cores by preemptive EDF;*
- *the communication subtasks are scheduled by non-preemptive EDF on the* S2SB *;*
- *the memory subtasks are scheduled by non-preemptive EDF on the* M2SB*;*

the task set is schedulable (i.e. every instance of every task-graph completes before its end-to-end deadline) if:

$$\forall p \in \mathcal{A}, \forall L \leq L^*, \quad \sum_{\tau_i \in \Lambda_p(\mathcal{T})} \mathrm{dbf}\,(\tau_i, L, p) \leq L \tag{19}$$

and

$$\forall L \leq L^*, \quad \sum_{\tau_i \in \Lambda_{\mathrm{S2SB}}(\mathcal{T})} \mathrm{dbf}\,(\tau_i, L, \mathrm{S2SB}) + B\,(\tau_i, L, \mathrm{S2SB}) \leq L \tag{20}$$

and

$$\forall L \leq L^*, \quad \sum_{\tau_i \in \Lambda_{\mathrm{M2SB}}(\mathcal{T})} \mathrm{dbf}\,(\tau_i, L, \mathrm{M2SB}) + B\,(\tau_i, L, \mathrm{M2SB}) \leq L \tag{21}$$

Proof. 1) For each core $p \in \mathcal{A}$, and from Theorem 1, the first condition of the dbf ensures that each computation subtask allocated on p is schedulable (i.e. every instance completes before its intermediate deadline); 2) For both S2SB and M2SB buses, and according to Theorem 2, the second and the third conditions ensure that each memory operation completes before its intermediate deadline.

Therefore: a) the precedence constraints are respected, b) the local deadline of the restitution subtask of every task-graph is less than or equal to its end-to-end deadline. Hence, the system is schedulable. □

5.3.2. Fitness function

The fitness score is an indicator of how "fit" a candidate solution is to meet the overall schedulability condition. The *fitness function* takes as input an individual candidate solution to our problem and specifies how far away this individual is from satisfying the schedulability condition.

We define `score`(r), with $r \in \mathcal{R}$, as the schedulability score of the cores, the inter-core bus (S2SB) and the memory-to-scratchpad bus (M2SB), respectively (line 3).

Definition 6 (Fitness score)**.** *A schedulability score is computed for each resource* $\mathsf{r} \in \mathcal{R}$ *as follow:*

$$\forall \mathsf{r} \in \mathcal{R}, \mathtt{score}(\mathsf{r}) = \max_{0<L\leq L^*} \left(\frac{\overline{dbf}(\mathsf{L}, \mathsf{r}) - \mathsf{L}}{\mathsf{L}}, 0 \right) \tag{22}$$

where $\mathcal{R} = \mathcal{A} \cup \{\mathsf{S2SB}\} \cup \{\mathsf{M2SB}\}$*, and* $\overline{dbf}(\mathsf{L}, \mathsf{r})$ *is the left-hand side expression of equations (19), (20) and (21), respectively.*

The fitness score *of a solution is the weighted sum of the three schedulability scores:*

$$\begin{aligned} \mathsf{f\_score} = \alpha_1 \mathtt{score}(\mathcal{A}) + \alpha_2 \mathtt{score}(\mathsf{S2SB}) \\ + \alpha_3 \mathtt{score}(\mathsf{M2SB}) \end{aligned} \tag{23}$$

and

$$\mathtt{score}(\mathcal{A}) = \frac{\sum_{\forall \mathsf{p} \in \mathcal{A}} \mathtt{score}(\mathsf{p})}{|\mathcal{A}|}$$

Where $\alpha_1 + \alpha_2 + \alpha_3 = 1$.

When the set of subtasks allocated on a given core $\mathsf{p} \in \mathcal{A}$ is schedulable by using preemptive EDF, then the score on p is equal to 0. Similarly, the score on both buses is equal to 0 if the memory subtasks are schedulable by non-preemptive EDF. The blocking time on the platform's cores is set to 0.

Algorithm 5 weights the fitness score for each non-schedulable individual. During our experimentation, it has been noticed that setting the fitness scores to give more importance to the schedulability on the inter-core bus (S2SB) yields better results, as they can vary greatly between individuals. In the experimental settings, we set $\alpha_1, \alpha_2, \alpha_3$ to 0.2, 0.6 and 0.2 respectively. The algorithm adds non-schedulable individuals to S and proceeds to the selection step (lines 7-10).

5.4. Creating the next generation

5.4.1. Selection

Before applying the genetic operators (crossover and mutation), the selection phase must be applied. Different approaches can be used to select the best individuals. A good comparative review of selection techniques in genetic algorithms can be found in Shukla et al. (2015). We use rank selection in this work. One of the advantages of rank selection is its ability to maintain genetic diversity. Therefore, we order individuals by non-increasing order of their fitness score computed in Algorithm 5. Then, we select the 50% best individuals and replace the remaining 50% with new individuals created by applying the genetic operators.

5.4.2. Mutation and crossover

The mutation and crossover operations are applied to the best-selected individuals. The crossover creates new individuals and tries to improve the scheduling objective by exchanging partial information contained in two randomly selected individuals (parents). In our case, each child's individual Ψ_i receives local deadlines from the parents.

Several variants of the crossover are popular Starkweather et al. (1991); Wang and Zheng (2001); Koonce and Tsai (2000). The original approach of the crossover operator is called `one-point` crossover: one crossover point on the two parent individuals is selected, and all local deadlines beyond that point are swapped between the two parents. This approach can be generalized to a multi-point operator, where the number of points is chosen randomly. It can be further generalised by copying local deadlines from the first parent with a probability p and from the second parent with a probability $1 - p$. The case $p = 0.5$ is called `uniform` crossover.

In this work, we use the `one-point` crossover operator. When two individuals are chosen for crossover, a crossover point is randomly selected at the same position for both individuals. This point divides each individual into two distinct parts: a head and a tail. The head of the first individual is combined with the tail of the second individual, and a similar operation is performed for the second individual. This process results in the creation of two new individuals (see Table 2).

The mutation operator is used to introduce genetic diversity in the population. With this operator, either we randomly change the value of a gene or we swap the positions of two of them. For instance, Koonce and Tsai (2000) uses swap, insertion, and inversion mutations. In this work, we randomly select an individual ψ, a DAG task τ_i and a subtask v. The local deadline for v is randomly modified, creating a new child individual. The new deadline is selected randomly in the interval $\mathtt{I} = [\mathtt{b\_inf}, \mathtt{b\_sup}]$ where $\mathtt{b\_inf} = \phi_v + \mathsf{C}(v)$ and $\mathtt{b\_sup}$ is the subtask local deadline. We then derive the new intermediate deadline of v, and we adjust the offset of its immediate successors.

The mutation and crossover rates, η_{mu} and η_{cr} respectively, are used to compute the number of individuals to generate to replace those eliminated by the selection operation.

6. Results and discussions

In this section, we will evaluate the performance of the proposed approaches in comparison to related work. Our contributions include allocation, deadlines assignment, and offsets assignment. First, we will assess the performance of our allocation strategy compared to bin-packing allocation heuristics such as Best Fit (BF) and Worst Fit (WF). Additionally, we will compare our deadline assignment techniques to classical deadline assignment heuristics, specifically *fair* and *proportional* approaches.

To evaluate the performance of these techniques, we measured the schedulability rate and the runtime on two different platforms. The first platform features 4 identical cores, while the second platform features 6 identical cores.

6.1. Task generation

Experiments have been conducted on a large number of randomly generated synthetic task sets. The task generation process takes as input the number of tasks n, the target baseline utilisation of the task set, and graph generation parameters.

First, the algorithm generates n task utilisation using the UUnifast algorithm Emberson et al. (2010) such that their total

τ_i	v	d_v
τ_1	$v^A_{1,0}$	2
	$v_{1,1}$	4
	$v_{1,2}$	4
	$v_{1,3}$	1
	$v_{1,4}$	1
	$v_{1,5}$	5
	$v^R_{1,6}$	2
τ_2	$v^A_{2,0}$	2
	$v_{2,1}$	1
	$v_{2,2}$	1
	$v_{2,3}$	5
	$v_{2,4}$	4
	$v_{2,5}$	4
	$v^R_{2,6}$	2

$\rightleftharpoons$

τ_i	v	d_v
τ_1	$v^A_{1,0}$	2
	$v_{1,1}$	4
	$v_{1,2}$	2
	$v_{1,3}$	2
	$v_{1,4}$	2
	$v_{1,5}$	4
	$v^R_{1,6}$	3
τ_2	$v^A_{2,0}$	2
	$v_{2,1}$	2
	$v_{2,2}$	1
	$v_{2,3}$	4
	$v_{2,4}$	4
	$v_{2,5}$	4
	$v^R_{2,6}$	2

$\Longrightarrow$

τ_i	v	d_v
τ_1	$v^A_{1,0}$	2
	$v_{1,1}$	4
	$v_{1,2}$	4
	$v_{1,3}$	1
	$v_{1,4}$	1
	$v_{1,5}$	5
	$v^R_{1,6}$	2
τ_2	$v^A_{2,0}$	2
	$v_{2,1}$	2
	$v_{2,2}$	1
	$v_{2,3}$	4
	$v_{2,4}$	4
	$v_{2,5}$	4
	$v^R_{2,6}$	2

τ_i	v	d_v
τ_1	$v^A_{1,0}$	2
	$v_{1,1}$	4
	$v_{1,2}$	2
	$v_{1,3}$	2
	$v_{1,4}$	2
	$v_{1,5}$	4
	$v^R_{1,6}$	3
τ_2	$v^A_{2,0}$	2
	$v_{2,1}$	1
	$v_{2,2}$	1
	$v_{2,3}$	5
	$v_{2,4}$	4
	$v_{2,5}$	4
	$v^R_{2,6}$	2

ψ_1 ψ_2 ψ_3 ψ_4

Table 2: Swapping local deadlines after a crossover point (ψ_3 and ψ_4 are new individuals)

sum is equal to the target baseline utilisation. For each task, we randomly select a period from a predefined list of periods: {15000, 12000, 20000, 24000, 30000, 10000, 40000, 60000}. This allows us to control the schedulability analysis complexity and avoid intractable hyper-periods. The task deadline is set equal to $0.8 \cdot \mathsf{T}$.

Next, we compute the utilisation for acquisition and restitution subtasks by inflating the task utilisation by the $\texttt{stall}_{\texttt{A/R}}$ parameter, which is set to 0.05 for each, that is tasks will spend 5% of their worst-case execution time (WCET) receiving and sending data from/to the main memory. We then use the UUnifast-discard algorithm again to distribute the remaining task utilisation among the subtasks of the considered task. We set the number of computation subtasks to 8. The execution time of each subtask is computed by multiplying the subtask utilisation by the task period. Furthermore, we generate precedence constraints between the different subtasks using the *layer-by-layer* method Cordeiro et al. (2010). It is known that the behavior of the classical deadline assignment depends on the graph structure. Therefore, we generate large-DAG or long-DAG. For large DAGs, we randomly select the number of subtasks per layer between 3 and 5. For long DAG tasks, the number of subtasks per layer is either 2 or 3. We consider a probability $\gamma = 0.2$[2] to create a precedence constraint between two subtasks from different layers. Precedence between subtasks belonging to non-consecutive layers is allowed. We guarantee that the task is weakly connected to ensure the absence of isolated subtasks.

Communication subtasks are automatically inserted between every two dependent computation subtasks. The execution time of a communication subtask $\texttt{stall}_{\texttt{m}}$ is set equal to 0.2 of the execution time of its immediate predecessor. This value is subtracted from the predecessor's execution time to maintain the baseline utilisation for the task graph unchanged.

COTS platforms with scratchpads are typically microcontrollers with limited computing capacity. Therefore, they are not able to handle highly complex software composed of hundreds of subtasks, compared to more powerful processors. Consequently, we believe that a setting with hundreds, and thousands of subtasks can not be representative to the applications that can be supported by scratchpad-based platforms.

6.2. Simulation Results and Discussions

We conducted experiments where we varied the baseline utilisation from 0 to the number of cores by a step of 0.4. For each utilisation value, we generated 100 task sets, with each task set containing 8 AECR-DAG tasks. We set the upper bounds for core utilisation as $\mathsf{U}^{\max} = 0.85$ or $\mathsf{U}^{\max} = 0.7$.

For the genetic algorithm, we explored two types of scenarios: a small population class with 50 individuals and a large population class with 150 individuals. The genetic algorithm stops after either 50 generations for small populations or 100 generations for the large population class.

In the experiments, connected computation subtasks of a task-graph that are allocated onto the same core are merged without affecting the precedence constraints and replaced by a single subtask with an execution time equals to the sum of all merged subtasks. This does not affect the results of the schedulability analysis, but allow to reduce the number of subtasks, thus optimizing the performance.

The experiments were conducted on a 12th Gen Intel(R) Core(TM) i7-1255U processor with 16 GB of RAM. We used CPLEX as the ILP solver for our experiments.

6.2.1. Comparison of allocation approaches

First, we conducted a study to analyze the impact of subtask-to-core allocation on the elimination of unnecessary

[2]We used the same value as in the literature Ben-Amor and Cucu-Grosjean (2022).

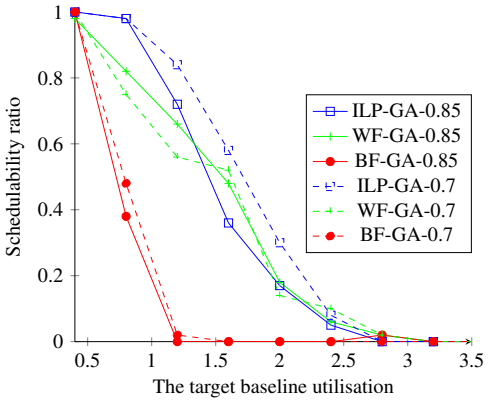


Figure 6: Schedulability for large DAG on Small population: U^{max} = 0.7 vs U^{max} = 0.85

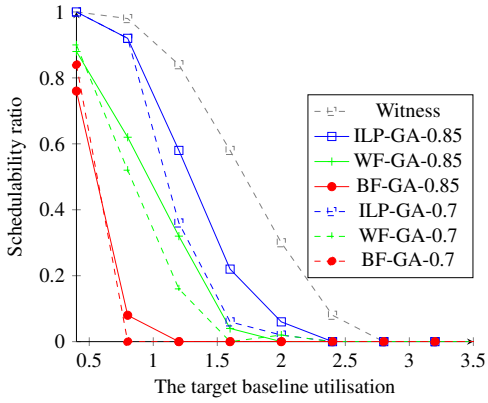


Figure 7: Schedulability for long DAG on Small population: U^{max} = 0.7 vs U^{max} = 0.85

communications. We compared our proposed approach against two well-known allocation heuristics: Worst-Fit (WF) and Best-Fit (BF). The experiments were conducted on a platform with 4 identical cores, using the same settings for the genetic algorithm (mutation probability η_{mu} and crossover probability η_{cr} set to 0.5). The experiments were performed for both small and large populations, as well as for long and large DAGs. Each combination is labeled as ILP, BF, or WF, representing the ILP, Best-Fit, or Worst-Fit allocation strategy, with GA denoting the genetic algorithm. A number represents an upper bound of the processor workload (0.7 and 0.85), and optionally, S or L denotes the small or the large population class.

In Figure 6, we present the results for the schedulability rate of the large graph DAG as a function of the target baseline utilisation. We are specifically interested in examining the impact of the processor workload upper bound on the schedulability ratio.

ILP-based allocation outperforms all other approaches with the same parameters, as it effectively eliminates unnecessary communications compared to the BF and WF approaches. The WF approach itself dominates the BF approach, as the BF approach tends to allocate a maximum number of subtasks on the same core, resulting in a highly loaded core compared to the other approaches. This jeopardizes schedulability, making it weaker compared to the other approaches.

When the processor workload upper bound is set to 0.7, all the allocation strategies outperform the same strategy with the upper bound set to 0.85. This unloads the different cores, providing more laxity to find feasible definitions of the intermediate deadlines. However, the total workload to schedule decreases to 2.8, which is 0.7 multiplied by the number of cores. This reduction is still reasonable since allowing more workload (0.85) per core does not improve schedulability at high workloads.

Figure 7 presents the same experiments as Figure 6, but for long DAGs. To provide an indication, we include the best results from the previous figure in gray[3]. All allocation strategies perform similarly to those in the previous figure. However, the schedulability has slightly decreased. This reduction can be attributed to the presence of long graphs, which offer less flexibility in allocating intermediate deadlines. Although long graphs leverage parallel execution, they significantly impact the schedulability of individual tasks.

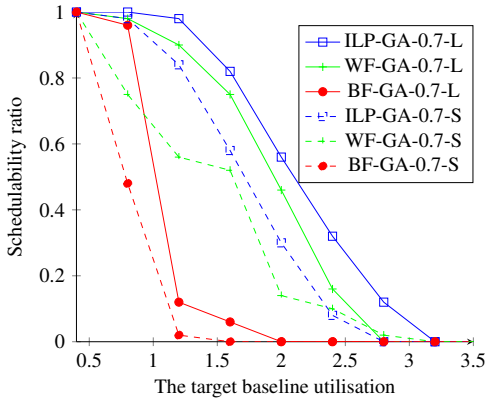


Figure 8: Schedulability for large DAG at U^{max} = 0.7: Small vs Large population

In Figure 8, we examine the impact of increasing the population size on schedulability as a function of the total workload. We compare different approaches with a workload threshold set to 0.7 for all cases. As expected, a larger population size (150 individuals) improves schedulability compared to an equivalent approach with a smaller population size (50 individuals). Even the WF-based allocation with a large population outperforms the ILP-based allocation with a smaller population (compared to the previous experimentations). This is due to the fact that a larger population allows for more diversification, and iterating for 150 generations instead of 50 generations allows for more intensive search. However, the Best-Fit approach still exhibits poor performance because it overly constrains certain cores compared to others.

[3]It should be noted that the previous experiment was conducted on a different task set, so the comparison is made for the sake of indication rather than direct comparability

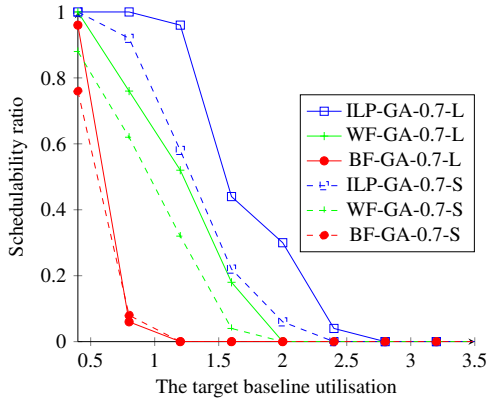


Figure 9: Schedulability for long DAG: $U^{max} = 0.7$ small vs large population

In Figure 9, we present the results of the same experiments as Figure 8, but specifically for long DAGs. The results are comparable to the previous figure, with some notable differences. The ILP approach still outperforms the WF approach, even with a smaller population size. This is explained by the nature of long DAGs in our task set generation algorithm, where we have no more than two subtasks per layer, therefore, the slack time distributed over the different subtasks is smaller compared to large DAGs. In such a highly constrained system, the impact of diversified and intensive search for intermediate deadlines is minimal compared the subtask-to-core allocation, which has a more significant impact in determining schedulability. Therefore, based on the configuration of our tasks, it may be more important to prioritize intensive intermediate deadline search for large DAGs or focus on employing more intensive allocation strategies for long DAGs, to allow the deadline assignment techniques to be more efficient.

In Figure 10, we provide the time required to perform the allocation and schedulability analysis for the previous experiments as a function of total utilisation. We have limited the total utilisation to 2, as having a smaller number of schedulable task sets beyond this limit does not allow drawing statistically significant and meaningful conclusions. Notably, the average time required to assess schedulability for large graphs (the two figures on the left-hand side) is generally shorter than that for the analysis of long DAGs (please note that the scales of the figures are different). On average, the BF heuristic takes more time because schedulability testing fails more frequently compared to the other approaches. Therefore, the analysis continues until the maximum number of iterations is reached, unlike the other approaches, which may terminate earlier if a schedulable solution is found in intermediate generations. Similarly, the ILP approach, which exhibits the best schedulability rates, completes its analysis faster than all the other approaches. Here, the required time to achieve a *good* allocation through the ILP is recovered by the efficiency of the corresponding deadline assignment process.

6.2.2. *Comparison of deadline assignment approaches*

In this section, we evaluate the performance of our genetic algorithm-based deadline assignment approach (Algorithm 1) against two deadline assignment methods: the fair deadline assignment heuristic denoted as FAIR and the proportional deadline assignment heuristic denoted as PROP. The tasks in this experience are allocated using the proposed ILP formulation, since it demonstrates the best performances. Each algorithm is, therefore, labeled by a combination of these techniques. The workload upper bound is set to $U^{max} = 0.7$. We compare the schedulability and the required analysis time for these approaches.

In Figures (11a and 11b), we present the schedulability ratios of different deadline assignment techniques, namely GA, FAIR, and PROP, as a function of total utilisation for large and long DAGs, respectively. Our genetic algorithm-based assignment technique significantly outperforms PROP and FAIR in both cases. The proportional approach assigns slack proportionally to the subtask execution time, which means that subtasks with longer execution times are more likely to receive additional slack compared to the FAIR approaches. As a result, these subtasks have more flexibility in terms of being scheduled without missing their deadlines. It is worth to notice that our approach is sensitive to the DAG topology, meaning that the structure and connections within the DAG can have an impact on the algorithm's performance, similarly to the related work.

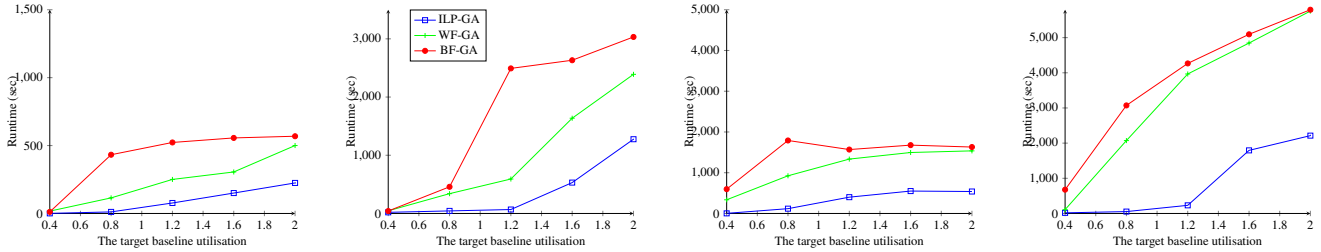
Our approach improves the schedulability performance at an acceptable cost on the required time to achieve the analysis, *i.e.*, t requires more time to complete the analysis compared to the PROP and FAIR approaches (Figures 11c and 11d), which have negligible execution times in comparison to ours.

In Figures (12a and 12b), we compare our genetic algorithm with ILP-based allocation against the PROP and FAIR approaches using WF for subtask-to-core allocation, as a function of total utilisation. In the left-hand side figure, we generate 10 DAGs, each consisting of 8 computational tasks, while in the right-hand side figure, we generate 3 DAGs, each consisting of 25 subtasks. The results indicate that our proposed approach consistently outperforms the approaches found in the literature. Our algorithm demonstrates that its performance is not dependent on the number of DAGs or the number of subtasks.

In Figures (13a, 13b and 13c), we present the schedulability ratios of different deadline assignment techniques as a function of total utilisation for large DAGs when varying the ratios of deadline and period $\frac{D_i}{T_i}$. Our genetic algorithm-based assignment technique significantly outperforms PROP and FAIR in both cases (b) and (c). An improvement in the performance of the genetic algorithm-based approach is noticeable in Figure (13c) since the tasks have larger deadlines. As in the precedent experience 12, the PROP approach outperforms FAIR approaches in all cases. The schedulability rates of the different deadline assignment techniques deteriorate sharply when the ratio is equal to 0.5.

7. Conclusion and future work

For COTS platforms comprising multiple CPU cores, contention for memory accesses can cause a significant decrease of schedulability.



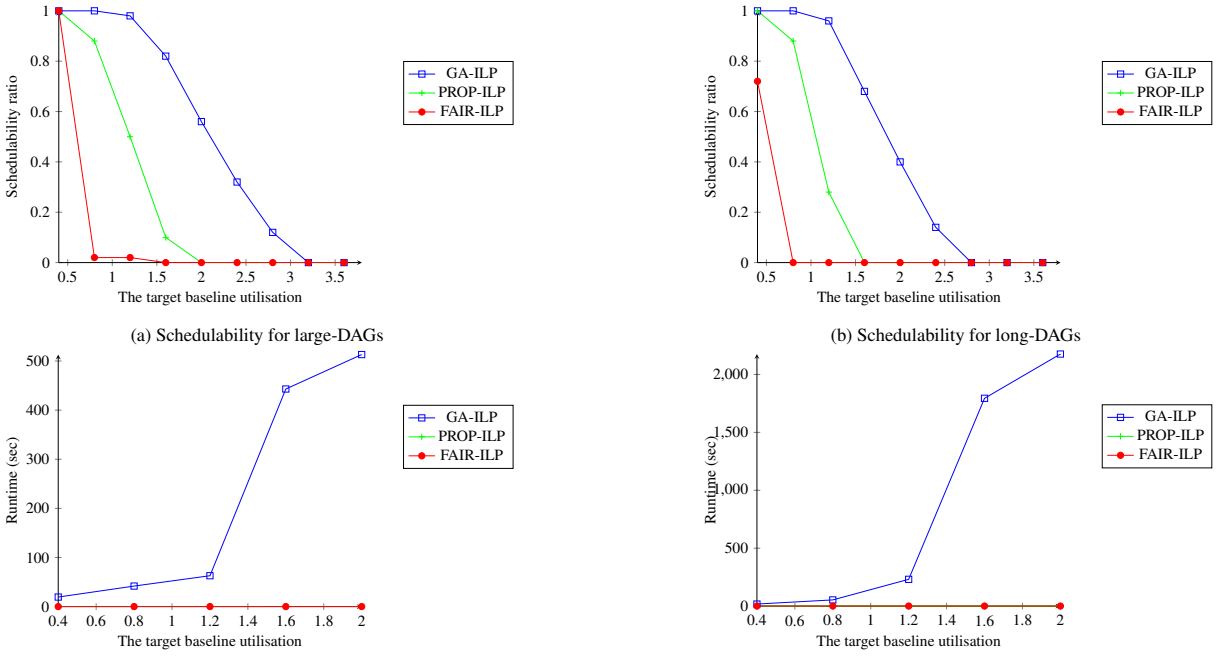
(a) Runtime for large-DAGs on small population(U^{max} = 0.85, 50 generations)

(b) Runtime for large-DAGs on large population (U^{max} = 0.7, 100 generations)

(c) Runtime for long-DAGs on small population (U^{max} = 0.85, 50 generations)

(d) Runtime for long-DAGs on large population(U^{max} = 0.7, 100 generations)

Figure 10: ILP's performances VS (WF and BF) algorithms when using GA for deadline assignment



(a) Schedulability for large-DAGs

(b) Schedulability for long-DAGs

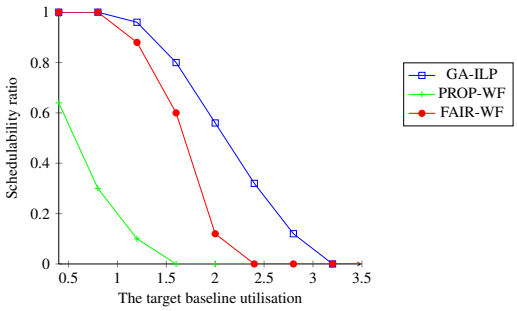
(c) Runtime for large-DAGs

(d) Runtime for long-DAGs

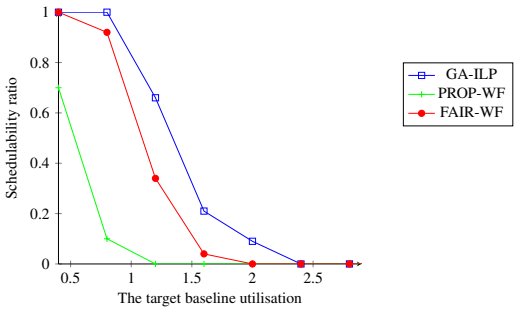
Figure 11: GA's performances VS (FAIR and PROP) when using ILP for task allocation on large population (U^{max} = 0.7)

This paper aims to avoid contention for DAG tasks. In order to achieve this goal, we extended the DAG task model to include memory transfers, and we named it AECR-DAG. This model was used along with scratchpad memories. We proposed an ILP-based allocation strategy for AECR-DAG task sets allocation and a genetic algorithm based technique for task scheduling parameters determination.

Our experiments show a significant improvement in the system performances compared to state-of-the-art. As future work, we plan to extend our techniques to heterogeneous architectures.

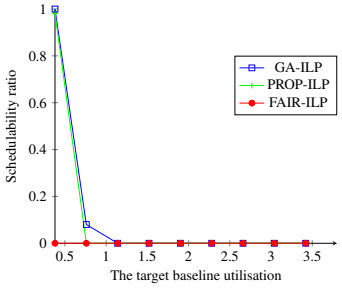


(a) Schedulability for large-DAGs (Scenario 1)

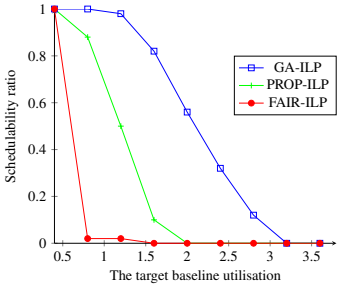


(b) Schedulability for large-DAGs (Scenario 2)

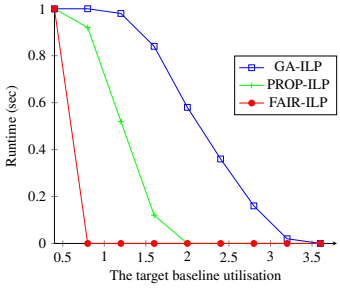
Figure 12: Compare GA's performances when using ILP for large-DAG tasks allocation with the literature (large population for GA, $U^{max} = 0.7$)



(a) Schedulability for large-DAGs: $\frac{D_i}{T_i} = 0.5$



(b) Schedulability for large-DAGs: $\frac{D_i}{T_i} = 0.8$



(c) Schedulability for large-DAGs: $\frac{D_i}{T_i} = 1$

Figure 13: GA's performances VS (FAIR and PROP) when varying the ratios of deadline and period $\frac{D_i}{T_i}$ ($U^{max} = 0.7$)

References

Alhammad, A., Pellizzoni, R., 2014. Time-predictable execution of multithreaded applications on multicore systems, in: 2014 Design, Automation & Test in Europe Conference & Exhibition (DATE), IEEE. pp. 1–6.

Baruah, S., Bonifaci, V., Marchetti-Spaccamela, A., Stougie, L., Wiese, A., 2012. A generalized parallel task model for recurrent real-time processes, in: 2012 IEEE 33rd Real-Time Systems Symposium, IEEE. pp. 63–72.

Baruah, S.K., Mok, A.K., Rosier, L.E., 1990. Preemptively scheduling hard-real-time sporadic tasks on one processor, in: [1990] Proceedings 11th Real-Time Systems Symposium, IEEE. pp. 182–190.

Ben-Amor, S., Cucu-Grosjean, L., 2022. Graph reductions and partitioning heuristics for multicore dag scheduling. Journal of Systems Architecture 124, 102359.

Burke, E.K., Gustafson, S., Kendall, G., 2004. Diversity in genetic programming: An analysis of measures and correlation with fitness. IEEE Transactions on Evolutionary Computation 8, 47–62.

Chekuri, C., Khanna, S., 2005. A polynomial time approximation scheme for the multiple knapsack problem. SIAM Journal on Computing 35, 713–728.

Cordeiro, D., Mounié, G., Perarnau, S., Trystram, D., Vincent, J.M., Wagner, F., 2010. Random graph generation for scheduling simulations, in: 3rd International ICST Conference on Simulation Tools and Techniques (SIMUTools 2010), ICST. p. 10.

Diaz-Gomez, P.A., Hougen, D.F., 2007. Initial population for genetic algorithms: A metric approach., in: Gem, Citeseer. pp. 43–49.

Druetto, A., Bini, E., Grosso, A., Puri, S., Bacci, S., Di Natale, M., Paladino, F., 2023. Task and memory mapping of large size embedded applications over numa architecture, in: Proceedings of the 31st International Conference on Real-Time Networks and Systems, pp. 166–176.

Durrieu, G., Faugère, M., Girbal, S., Pérez, D.G., Pagetti, C., Puffitsch, W., 2014. Predictable flight management system implementation on a multicore processor, in: Embedded Real Time Software (ERTS'14).

Emberson, P., Stafford, R., Davis, R.I., 2010. Techniques for the synthesis of multiprocessor tasksets, in: WATERS.

Hoogeveen, J.A., Lenstra, J.K., Veltman, B., 1996. Preemptive scheduling in a two-stage multiprocessor flow shop is np-hard. European Journal of Operational Research 89, 172–175.

Houssam-Eddine, Z., Capodieci, N., Cavicchioli, R., Lipari, G., Bertogna, M., 2020. The hpc-dag task model for heterogeneous real-time systems. IEEE Transactions on Computers 70, 1747–1761.

IBM, 1987. CPLEX User's Manual. volume 12.

IT, A., 2020. Aurix 32-bit microcontrollers for automotive and industrial applications. Infineon Technologies AG 1.

Kandemir, M., Ramanujam, J., Irwin, J., Vijaykrishnan, N., Kadayif, I., Parikh, A., 2001. Dynamic management of scratch-pad memory space, in: Proceedings of the 38th annual Design Automation Conference, pp. 690–695.

Koonce, D., Tsai, S.C., 2000. Using data mining to find patterns in genetic algorithm solutions to a job shop schedule. Computers & Industrial Engineering 38, 361–374.

Madureira, A., Ramos, C., do Carmo Silva, S., 2002. A coordination mechanism for real world scheduling problems using genetic algorithms, in: Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02 (Cat. No. 02TH8600), IEEE. pp. 175–180.

Maia, C., Nelissen, G., Nogueira, L., Pinho, L.M., Pérez, D.G., 2017. Schedulability analysis for global fixed-priority scheduling of the 3-phase task model, in: 2017 IEEE 23rd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), IEEE. pp. 1–10.

Marinca, D., Minet, P., George, L., 2004. Analysis of deadline assignment methods in distributed real-time systems. Computer Communications 27, 1412–1423.

Martello, S., Toth, P., 1990. Knapsack problems: algorithms and computer implementations. John Wiley & Sons, Inc.

McLean, S.D., Craciunas, S.S., Hansen, E.A.J., Pop, P., 2020. Mapping and scheduling automotive applications on adas platforms using metaheuristics, in: 2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), IEEE. pp. 329–336.

Melani, A., Bertogna, M., Bonifaci, V., Marchetti-Spaccamela, A., Buttazzo, G., 2016. Schedulability analysis of conditional parallel task graphs in multicore systems. IEEE Transactions on Computers 66, 339–353.

Mitra, H., Ramanathan, P., 1993. A genetic approach for scheduling non-preemptive tasks with precedence and deadline constraints, in: [1993] Proceedings of the Twenty-sixth Hawaii International Conference on System Sciences, IEEE. pp. 556–564.

Monnier, Y., Beauvais, J.P., Deplanche, A.M., 1998. A genetic algorithm for scheduling tasks in a real-time distributed system, in: Proceedings. 24th EUROMICRO Conference (Cat. No. 98EX204), IEEE. pp. 708–714.

Oh, J., Wu, C., 2004. Genetic-algorithm-based real-time task scheduling with multiple goals. Journal of systems and software 71, 245–258.

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: 2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium, IEEE. pp. 269–279.

Pellizzoni, R., Lipari, G., 2005. Feasibility analysis of real-time periodic tasks with offsets. Real-Time Systems 30, 105–128.

Qamhieh, M., Fauberteau, F., George, L., Midonnet, S., 2013. Global edf scheduling of directed acyclic graphs on multiprocessor systems, in: Proceedings of the 21st International conference on Real-Time Networks and Systems, pp. 287–296.

Ramamritham, K., 1990. Allocation and scheduling of complex periodic tasks, in: Proceedings., 10th International Conference on Distributed Computing Systems, IEEE Computer Society. pp. 108–109.

Rosen, J., Andrei, A., Eles, P., Peng, Z., 2007. Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip, in: 28th IEEE International Real-Time Systems Symposium (RTSS 2007), IEEE. pp. 49–60.

Senoussaoui, I., Zahaf, H.E., Lipari, G., Benhaoua, K.M., 2022. Contention-free scheduling of prem tasks on partitioned multicore platforms, in: 2022 IEEE 27th International Conference on Emerging Technologies and Factory Automation (ETFA), IEEE. pp. 1–8.

Shukla, A., Pandey, H.M., Mehrotra, D., 2015. Comparative review of selection techniques in genetic algorithm, in: 2015 International Conference on Futuristic Trends on Computational Analysis and Knowledge Management (ABLAZE), pp. 515–519. doi:10.1109/ABLAZE.2015.7154916.

Slim, B.A., Liliana, C.G., Mezouak, M., Sorel, Y., 2020. Probabilistic schedulability analysis for real-time tasks with precedence constraints on partitioned multi-core, in: 2020 IEEE 23rd International Symposium on Real-Time Distributed Computing (ISORC), IEEE. pp. 142–143.

Starkweather, T., McDaniel, S., Mathias, K.E., Whitley, L.D., Whitley, C., 1991. A comparison of genetic sequencing operators., in: ICGA, pp. 69–76.

Suhendra, V., Roychoudhury, A., Mitra, T., 2010. Scratchpad allocation for concurrent embedded software. ACM Transactions on Programming Languages and Systems (TOPLAS) 32, 1–47.

Tabish, R., Mancuso, R., Wasly, S., Pellizzoni, R., Caccamo, M., 2019. A real-time scratchpad-centric os with predictable inter/intra-core communication for multi-core embedded systems. Real-Time Systems 55, 850–888.

Udayakumaran, S., Barua, R., 2003. Compiler-decided dynamic memory allocation for scratch-pad based embedded systems, in: Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems, pp. 276–286.

Wang, L., Zheng, D.Z., 2001. An effective hybrid optimization strategy for job-shop scheduling problems. Computers & Operations Research 28, 585–596.

Wu, Y., Gao, Z., Dai, G., 2014. Deadline and activation time assignment for partitioned real-time application on multiprocessor reservations. Journal of Systems Architecture 60, 247–257.

Yang, Y., Wang, M., Yan, H., Shao, Z., Guo, M., 2010. Dynamic scratch-pad memory management with data pipelining for embedded systems. Concurrency and Computation: Practice and Experience 22, 1874–1892.

Yao, G., Pellizzoni, R., Bak, S., Betti, E., Caccamo, M., 2012. Memory-centric scheduling for multicore hard real-time systems. Real-Time Systems 48, 681–715.

Zahaf, H.E., Benyamina, A.E.H., Lipari, G., Olejnik, R., Boulet, P., 2016. Modeling parallel real-time tasks with di-graphs, in: Proceedings of the 24th International Conference on Real-Time Networks and Systems, pp. 339–348.

Zahaf, H.E., Capodieci, N., Cavicchioli, R., Bertogna, M., Lipari, G., 2019. A c-dag task model for scheduling complex real-time tasks on heterogeneous platforms: preemption matters. arXiv preprint arXiv:1901.02450 .

Zahaf, H.E., Lipari, G., Niar, S., et al., 2020. Preemption-aware allocation, deadline assignment for conditional dags on partitioned edf, in: 2020 IEEE 26th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), IEEE. pp. 1–10.

Zitzler, E., Deb, K., Thiele, L., 2000. Comparison of multiobjective evolutionary algorithms: Empirical results. Evolutionary computation 8, 173–195.