Journal of Systems Architecture 153 (2024) 103198

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

Gradient descent algorithm for the optimization of fixed priorities in real-time systems

Juan M. Rivas [a,*], J. Javier Gutiérrez [a], Ana Guasque [b], Patricia Balbastre [b]

[a] Software Engineering and Real-Time Group, Universidad de Cantabria, Avda. de los Castros 48, Santander, 39005, Spain
[b] Instituto de Automática e Informática Industrial (ai2), Universitat Politècnica de València, Camino de Vera, s/n, Valencia, 46022, Spain

ARTICLE INFO

Keywords:
Real-time
Fixed-priorities
Optimization
Gradient descent

ABSTRACT

This paper considers the offline assignment of fixed priorities in partitioned preemptive real-time systems where tasks have precedence constraints. This problem is crucial in this type of systems, as having a *good* fixed priority assignment allows for an efficient use of the processing resources while meeting all the deadlines. In the literature, we can find several proposals to solve this problem, which offer varying trade-offs between the quality of their results and their computational complexities. *In this paper*, we propose a new approach, leveraging existing algorithms that are widely exploited in the field of Machine Learning: Gradient Descent, the Adam Optimizer, and Gradient Noise. We show how to adapt these algorithms to the problem of fixed priority assignment in conjunction with existing worst-case response time analyses. We demonstrate the performance of our proposal on synthetic task-sets with different sizes. This evaluation shows that our proposal is able to find more schedulable solutions than previous heuristics, approximating optimal but intractable algorithms such as MILP or brute-force, while requiring reasonable execution times.

1. Introduction

Real-time systems, which impose both functional and timing constraints, can be found in many mission-critical applications in domains such as automotive, aerospace and healthcare. These systems are usually composed of a set of tasks that are concurrently scheduled by a scheduler provided by a Real-Time Operating System (RTOS). Although already proposed more than half a century ago [1] in the form of Rate Monotonic Scheduling, Fixed Priority Scheduling (FPS) nowadays remains the most common scheduling policy used in real-time systems [2], and is extensively supported in current RTOSs and programming languages [3]. With FPS, each task is assigned at design time a static fixed priority. At runtime, the scheduler selects for execution the active task with the highest priority.

The assignment of fixed priorities is a vital step in the design of FPS real-time systems. A bad selection of a priority assignment may result in an under-utilization of the resources to be able to meet the timing constraints. On the contrary, a good priority assignment allows for higher utilization of the resources while complying with the timing constraints, and thus reduced costs.

We consider real-time systems characterized by precedence relationships, akin to those encountered in distributed systems. The challenge of finding a fixed priority assignment that meets the timing constraints in this type of systems is known to be NP-hard [4]. Several heuristics have been proposed to work-around this problem offering sub-optimal solutions [3], ranging from the application of general purpose techniques such as Genetic Algorithms [5,6] or Simulated Annealing [4], to tailor-made algorithms such as HOPA [7]. An interesting technique that has been proposed is Mixed Integer Linear Programming (MILP) [8]. While MILP is in theory able to provide optimal solutions, its main drawback lies in its scalability issues, which become apparent when the complexity of the system increases.

Nowadays, the field of Artificial Intelligence, and more specifically Machine Learning, is experiencing the highest rates of research interest and production in the area of Computer Science. This push is specially felt in the advancements reported on areas such as Natural Language Processing, Image Generation or Autonomous Systems. In its most basic building blocks, these systems are usually composed of vast neural networks that must be subject to a computing intensive training process in order to produce useful results. This training is essentially an optimization process, in which the parameters of the neural networks (e.g. weights and biases) are iteratively tuned to minimize a cost function. Currently, Gradient Descent (GD) is the *de facto* algorithm for training such neural networks [9]. GD is a general-purpose optimization algorithm that is used to minimize differentiable mathematical functions. It achieves this by repeatedly making small adjustments to the parameters of the cost function in the opposite

* Corresponding author.
E-mail address: juanmaria.rivas@unican.es (J.M. Rivas).

https://doi.org/10.1016/j.sysarc.2024.103198
Received 31 January 2024; Received in revised form 5 April 2024; Accepted 31 May 2024
Available online 4 June 2024

direction of the gradient. The field of Machine Learning has produced further variants and optimizations to the original GD algorithm, that have demonstrated their effectiveness in locating minima of the cost function in large search spaces, such as deep neural networks [9].

In this paper, we propose exploiting these advancements on efficient training of neural networks, by adapting the Gradient Descent optimization algorithm to the problem of assigning fixed priorities in real-time systems. Additionally, from the literature of Machine Learning we pick two techniques that enhance the behavior of Gradient Descent: the Adam optimizer and Gradient Noise. By employing Gradient Descent and subsequent optimizations, we aim to define a priority assignment algorithm that approximates an optimal assignment, while avoiding the scalability issues of optimal techniques such as MILP.

This paper is organized as follows. In Section 2, we describe the system model for real-time systems that we assume in this paper. In Section 3, we describe how a generic gradient descent algorithm operates, and also we list state-of-the-art algorithms to assign fixed priorities in real-time systems that conform with our model. Section 4 describes the main contribution of this paper, a Gradient Descent-based algorithm to assign fixed priorities. Section 5 proposes an optimization of the previous algorithm to accelerate its execution. In Section 6, we present the results of an exhaustive evaluation of our proposal. Finally, in Section 7, we present the main conclusions of this work.

2. System model

To describe the system model we follow the terminology of the OMG MARTE specification for Schedulability Analysis Modeling (SAM) [10]. An implementation of this specification can be found in the MAST modeling framework [11,12].

We consider real-time systems composed of N steps statically allocated to processing resources. A step can model a task that executes on a CPU processing resource. The steps (i.e. tasks) are grouped into end-to-end (e2e) flows that establish precedence relationships among the steps. For simplifying purposes, in this paper we consider linear e2e flows in which each step may have at most one successor and predecessor step. Therefore, each e2e flow Γ_i is composed of a linear sequence of N_i steps. The *j-th* step of e2e flow Γ_i is denoted as τ_{ij}. A step τ_{ij} has a worst-case execution time (WCET) denoted as C_{ij}, and is statically allocated to processing resource PR_k.

Each end-to-end flow Γ_i is released by a periodic sequence of external events with period T_i. Sporadic events are also supported, in which case the period is considered as the minimum inter-arrival time of the events. We assume that all event sequences that arrive at the system and their worst-case rates are known in advance. The relative phasing of the activations of different end-to-end flows is assumed to be arbitrary and unknown.

Deadlines can be set for individual steps or for the whole end-to-end flow. In this paper, to simplify the notation and without loss of generality, we only consider e2e deadlines. We define D_i as the e2e deadline that flow Γ_i must meet, counting from the release of the e2e flow until its last step finishes its execution. End-to-end deadlines have no restrictions in relation to the periods. Specifically, deadlines can be longer than the periods. The e2e flow Γ_i is released by the arrival of an external event e_i with period T_i and e2e deadline D_i. Assuming that the event e_i arrives at time t, this deadline imposes that the execution of the whole end-to-end flow must finish before $t + D_i$. Fig. 1 shows two simple e2e flows composed of 3 steps, that traverse 3 different processing resources (PR_1, PR_2 and PR_3).

We assume that each processing resource is governed by a fixed priority preemptive scheduler, and that each step is scheduled by their statically assigned fixed priority. The fixed priority of step τ_{ij} is denoted as P_{ij}. Although the values of the fixed priorities are usually restricted to integers, in this paper we relax this restriction to allow any real number: $P_{ij} \in \mathbb{R}$. Moreover, step τ_{ij} is said to have a higher priority than τ_{km} if $P_{ij} > P_{km}$, where P_{km} is the priority of step τ_{km}.

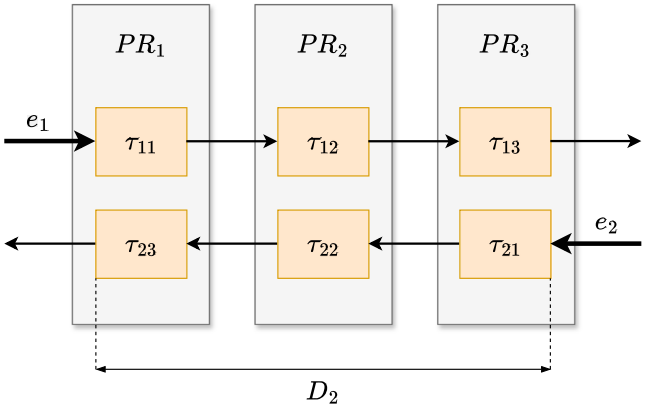

Fig. 1. Two end-to-end flows (Γ_1 and Γ_2), composed of 3 steps, that traverse 3 processing resources (PR_1, PR_2 and PR_3).

We define the worst-case response time (WCRT) of a step as the longest possible (or an upper bound) interval counting from the release of its end-to-end flow till the step's completion. For a step τ_{ij}, we denote its WCRT as R_{ij}. We assume that a step τ_{ij} can have a real response time between 0 and R_{ij}. We define the WCRT of a flow Γ_i as the WCRT of its last step, and denote it as R_i. When every end-to-end flow meets its deadline, that is $R_i \leq D_i \forall i$, we say that the system is *schedulable*.

We allow steps to have input jitter, which models the maximum amount of time the release of a step may be delayed. The jitter of step τ_{ij} is denoted as J_{ij}. Jitters can have any arbitrary positive value. There is an inter-dependency between the jitters and WCRTs because of the precedence constraints: the start time of a step depends on the finish time of its predecessor, which is not constant, as its response time can range from 0 to its WCRT.

The WCRTs of the steps, and by extension of the end-to-end flows, are obtained by applying a WCRT analysis. Since the problem of obtaining exact WCRTs is NP-hard [13], these analyses generally obtain safe upper bounds. In the literature, we can find several such analyses that support this system model, with varying degrees of pessimism and time complexity. For instance, the Holistic analysis [14,15] makes the simplifying but safe assumption that the dependency among tasks in the same e2e flow is only indirectly taken into account by the propagation of their jitters. Offset-Based techniques [16,17] model the in-flow step inter-dependencies more exactly through the use of offsets, resulting in generally less pessimistic WCRTs. Implementations of these techniques can be found in the open source MAST Analysis tool [11,18].

In this paper only tasks executed on CPUs and linear e2e flows are considered, but the MAST model and available WCRT analyses support a wider range of systems. For instance, distributed systems can be modeled by viewing messages as steps transmitted through network processing resources. The analysis of this message traffic on the networks can be carried out using similar techniques to those used in the CPUs, by adding a blocking term that accounts for the non preemptability of the message packets [15,16]. Non-linear end-to-end flows with fork and join events are also supported [19]. More specific and industry-relevant standards, such as Logical Execution Time (LET) in the automotive sector, or ARINC-653 in aerospace applications, can also be supported [20–22].

3. Related work

In this section we provide the context of this work. First, in Section 3.1 we present a review of the state-of-the-art on the assignment of fixed priorities. Then, in Section 3.2, we describe how a generic Gradient Descent algorithm operates.

3.1. Fixed priority assignment

The problem of finding a schedulable fixed priority assignment for a real-time system is known to be NP-hard [4]. The works we can find in the literature [3] offer different balances between their degree of optimality and their computational complexities. Here, we consider an algorithm as sub-optimal if it may be unable to find a schedulable priority assignment when one exists.

One of the first solutions proposed was to leverage the general purpose Simulated Annealing (SA) algorithm [4]. SA is a global optimization technique that attempts to find the lowest point in an energy landscape, by emulating the physical process of heating and controlled cooling of a material to alter its physical properties. This algorithm was proposed to both find schedulable priority assignments and to map steps to processing resources.

Another algorithm found in the literature is HOPA [7]. This iterative technique was created *ad-hoc*, and it is based on the distribution of the end-to-end deadlines among the steps, in the form of virtual deadlines, taking into account the steps' worst-case response times. These virtual deadlines are then transformed into fixed priorities by following a Deadline Monotonic criterion. HOPA has demonstrated to be capable of finding more schedulable priority assignments than SA, in significantly less time [7].

Proportional Deadlines (PD) [23] is another technique that can be used to assign fixed priorities. PD is a non-iterative algorithm that distributes the end-to-end deadlines among the steps proportionally to the WCETs. As in HOPA, these virtual deadlines are then transformed into fixed priorities by applying a Deadline Monotonic principle. As it is non-iterative, PD lacks the capability to improve on the initial priority assignment, and generally is outperformed by iterative algorithms. Nonetheless, it is a very fast algorithm that can be useful as a seed in iterative algorithms, such as HOPA.

Genetic Algorithms are part of the so-called evolutionary algorithms, which imitate biological mechanisms that guide the evolution process in species, and which are used to look for solutions to diverse problems in wide search spaces. In the context of optimizing real-time systems, they have been used as a part of a multi-objective strategy, for example: (1) to allocate tasks in identical processors and to determine cyclic scheduling [24,25]; (2) to allocate independent tasks in heterogeneous distributed real-time systems, to which fixed priorities are assigned following the Rate Monotonic scheme [26]; or (3) to assign priorities to tasks, as well as to determine the timing slots for the messages transmitted through a TDMA network [6]. The work in [5] develops a permutational genetic algorithm for the assignment of priorities to tasks and messages in a distributed real-time system, where the results are compared to those obtained by HOPA [5], showing a slight improvement (up to 4% higher schedulable utilization), but at a much higher cost in computation time.

Similarly, the authors of [27] propose a multi-objective competitive co-evolution algorithm. This work considers tasks that may have precedence constraints, but presents two main incompatibilities with our model, (1) the exact activation instants of the flows are known beforehand (i.e. the relative phasings of the flows are known), and (2) the tasks are dynamically assigned a processor at runtime according to their global priority. The algorithm is evaluated using simulations on a set of industrial examples, and compares positively to relatively simple algorithms (manual expert assignment, random search and sequential search). The execution times range from less than 2 min to 16 h in the more complex example.

Mixed Integer Linear Programming (MILP) is a promising technique that has been applied to both step-to-processor mapping and priority assignment problems [8]. In MILP, the problem is described as a set of linear constraints and a linear objective function, which gets optimized within those constraints. The main benefit of MILP is that, in theory, it is an optimal algorithm, that is, it will find a schedulable solution if one exists. Existing commercial libraries such as Gurobi [28] provide efficient environments to define MILP problems. Nevertheless, we identify that MILP has two main challenges in its applicability to our system model.

First, it is known to be NP-hard itself. This translates into becoming intractable as the search space becomes bigger, which is confirmed in empirical evaluations [8]. Second, MILP requires the definition of linear constraints, which may not be available. For the problem we are tackling in this paper, that is, to find a schedulable priority assignment, we would require linear equations that model a schedulability test compatible with our system model. As far as we know, no such equations have yet been defined that could be feasibly applied to MILP. Generally, simplifying assumptions must be included in the model to obtain feasible equations. For instance, in [8] the distributed model assumes a strictly periodic activation of the steps, with constrained deadlines. This greatly simplifies the problem, and makes that work incompatible with our system model. Recent efforts [29] aim to relax those previous restrictions imposed on the system model, but still include several simplifications that make it incompatible with our model. Namely, it assumes that every step in an e2e flow has the same priority, the end-to-end deadlines are constrained, and that the jitters are assumed to be known before-hand and kept constant. In our model, there is an inter-dependency between jitters and worst-case response times that cannot be solved exactly if jitters are assumed constant.

The scalability issues of MILP were tackled in [30], which proposes a more efficient and near-optimal algorithm that exploits the idea of finding the maximum virtual deadlines that would render the system not schedulable. These values are iteratively computed with an in-loop standard Integer Linear Programming (ILP) optimization with relaxed constraints. Similarly to [8], this paper considers a simpler system model, with constrained deadlines and independent tasks without jitter.

In this paper, we are aiming to improve upon the performance of sub-optimal algorithms such as HOPA, while avoiding the scalability issues that an optimal technique such as MILP suffers.

3.2. Generic gradient descent algorithm

Gradient Descent (GD) is an optimization algorithm for minimizing differentiable mathematical functions. GD is extensively used in the field of Machine Learning to optimize parameters such as coefficients in linear regression problems or weights in neural networks.

The GD algorithm starts with an initial guess for the function input parameters, and then iteratively adjusts them in the direction that reduces the value of the function the most, until a minimum (or some other criteria) is reached. To achieve this behavior, GD employs the gradient of the function. The gradient of a function at a given input is a vector that points in the direction of the steepest increase of the function at said input. GD leverages this observation by updating the current input parameters in the opposite direction of the gradient, as this represents the direction of steepest descent. The size of this update is usually modulated by a factor called *learning rate*.

In formal terms, given an n-dimensional function $f(x_1, \dots, x_n)$, its input can be represented as an n-dimensional point $p = (p_1, \dots, p_n)$. Function f is usually called the *cost function*. Therefore, GD is said to minimize the *cost function*. The gradient of function f at point p is depicted as $\nabla f(p)$, which can be expressed as a vector of the partial derivatives of f at point p as follows:

$$\nabla f(p) = \left[\frac{\partial f}{\partial x_1}(p),\ \frac{\partial f}{\partial x_2}(p),\ \dots\ \frac{\partial f}{\partial x_n}(p) \right] \tag{1}$$

At a given iteration number t, the next point p^{t+1} is calculated by moving the current point p^t in the opposite direction of the gradient, which is scaled by a *learning rate* η:

$$p^{t+1} = p^t - \eta \nabla f\left(p^t\right) \tag{2}$$

Starting from an initial point p^0, Eq. (2) is iteratively applied. If an appropriate η factor is applied, and function f is differentiable

around p^t, the inequality $f(p^{t+1}) \leq f(p^t)$ is respected. Therefore, by repeatedly applying Eq. (2), GD will traverse function f along a path that keeps minimizing f, until a point that produces a minimum of the function is reached. Further stopping criteria could be added, for instance establishing a maximum number of iterations.

Parameters such as the *learning rate* (η) are usually called *hyper-parameters*. A hyper-parameter is defined as a configuration variable that can tweak the behavior of the algorithm, but it is not a parameter that is being optimized. Typically, *known good* values for the hyper-parameters are selected and kept unchanged. For instance, a very low positive value for the *learning rate* (≈ 0.01) is usually considered as a good candidate in the context of training neural networks.

The main challenge for GD is that there is no guarantee that a global minimum will be found: depending on the starting point and the shape of f, following the gradient may lead to different local minima. Furthermore, GD may get stuck in flat areas of the function, as the gradient there evaluates to 0.

The chances of finding a global minimum can be increased by enhancing GD with the idea of *momentum*, which can be intuitively explained if we visualize the Gradient Descent algorithm in the physical world. If we imagine the function f as a 3D shape, and the starting point as the location at which we release a ball, the ball will follow a downward path along the slopes of the shape, i.e. a path opposite to the gradients of the shape. The ball will continue its descent until reaching a resting state at a local minimum. If we add mass to the ball and consider gravity, the ball will accumulate momentum as it accelerates down steep slopes. This momentum may be sufficient for the ball to surpass the first local minimum it encounters, thus increasing the possibility of finding further lower minima.

Several techniques have been proposed that leverage this idea of momentum [9], which have proved to be effective when optimizing vast neural networks comprised of cost functions with millions of parameters. It is worth stating that it is not the objective of this paper to propose new solutions in the field of Machine Learning, but to select and exploit existing and successful techniques, adapting them to the problem of fixed priority assignment, which will be carried out in the next section.

4. Gradient descent priority assignment

The problem we aim to solve can be defined as follows: given a real-time system composed of end-to-end flows as described in Section 2, we want to find a fixed priority assignment for every step in the system in such a way that the system becomes schedulable. That is, we want to find a fixed priority value P_{ij} for every step τ_{ij}, so that the worst-case response times of every end-to-end flow are less than or equal to their end-to end deadlines, $R_i \leq D_i, \forall i$. We assume that every step is already mapped to a processing resource. Any new step-to-processor mapping would require a re-computation of the fixed priority values.

We define Π as a priority assignment, which is a vector containing a particular mapping of a fixed priority value to each step. Therefore, a priority assignment Π is a flat view of the priority values assigned to a system. The ordering inside a priority assignment vector Π follows the ordering of the e2e flows and their steps, as shown in Eq. (3).

$$\Pi = \left[P_{11}, P_{12}, \ldots, P_{21}, P_{22}, \ldots\right] \tag{3}$$

We propose adapting the Gradient Descent (GD) algorithm to assign fixed priorities. The resulting algorithm is called Gradient Descent Priority Assignment (GDPA). As previously described in Section 3.2, the basic idea of the generic Gradient Descent algorithm involves iteratively adjusting the input parameters of a cost function in the direction that makes the function decrease the most. GDPA mirrors this behavior by iteratively adjusting the fixed priority values of every step in the direction that reduces the worst-case response times in relation to the imposed deadlines

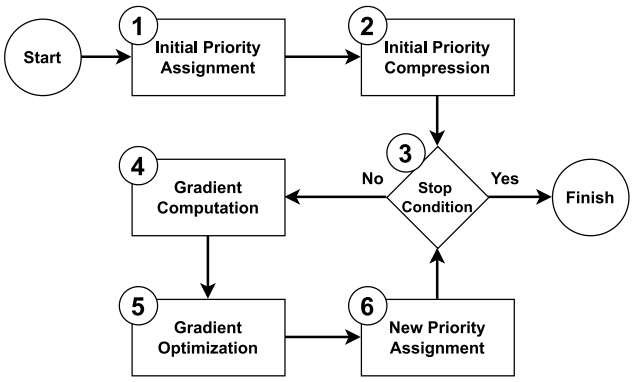

Fig. 2. Gradient Descent Priority Assignment flowchart.

GDPA is an iterative algorithm that will compute and evaluate one priority assignment per iteration. Accordingly, we denote as Π^t the priority assignment evaluated at iteration t. By extension, we define P_{ij}^t as the fixed priority value assigned to step τ_{ij} at iteration t.

Fig. 2 shows a high level overview of the GDPA algorithm. GDPA is composed of 6 main phases: (1) initial priority assignment, (2) initial priority compression, (3) stop condition, (4) gradient computation, (5) gradient optimization, and (6) new priority assignment. These phases will be described in detail in the following sub-sections.

4.1. Initial priority assignment

Any Gradient Descent algorithm requires an initial set of input values from which to start the optimization process. In the case of GDPA, these initial values represent an initial priority assignment, denoted as Π^{ini}. GDPA does not impose any restriction on this initial priority assignment. A technique such as PD [23] is a good candidate, as it is a fast algorithm. A better starting point can be provided by employing a more advanced but slower technique such as HOPA [7]. A completely random priority assignment can also be used.

4.2. Initial priority compression

In any Gradient Descent algorithm, the next candidate solution is always calculated by adding some values to the previous candidate. Consequently, in the case of GDPA, after several iterations there is a risk of inducing runaway priority values that could diverge as the algorithm progresses. To avoid this problem, we add a priority normalization stage by defining a compression function c, which constrains the priority values into the range $[-1, 1]$. The compression function c is defined as follows:

$$c\left(\Pi\right) = \frac{\Pi}{max\left(|\Pi|\right)} \tag{4}$$

where Π is a priority assignment, and $max(|\Pi|)$ represents the maximum of the absolute values of every priority value in Π.

Before feeding the initial priority assignment into GDPA, the compression function c is applied to make sure the priority values get constrained within the expected range $[-1, 1]$. The resulting priority assignment is labeled as Π^0, indicating that this is the first priority assignment evaluated by GDPA, that is, the priority assignment at iteration t=0. Formally:

$$\Pi^0 = c\left(\Pi^{ini}\right) \tag{5}$$

It is worth noting that the compression function c does not modify the actual priority ordering of the steps. Therefore, it has no impact on the results of the schedulability analysis.

4.3. Stop condition

The priority assignment computed at each iteration is evaluated to determine whether the GDPA algorithm should terminate. In the case of the first iteration t=0, the priority assignment Π^0 is evaluated. In GDPA, the algorithm stops if any of the following criteria is met:

1. The current priority assignment Π^t is schedulable.
2. A maximum number of iterations has been reached.

To determine if a priority assignment is schedulable, any schedulability test compatible with the system model described in Section 2 can be employed. Typically, the schedulability of the priority assignment can be determined by applying a worst-case response time analysis, and then comparing the resulting WCRTs with the deadlines. Techniques such as the Holistic analysis [15] or Offset-Based analyses [16,17] can be applied. The selection of which analysis to employ must balance the trade-offs between computing time and the pessimism in the obtained WCRTs.

Regardless of which stopping criteria was met to terminate the algorithm, GDPA will always return the priority assignment that produces the lowest value of the cost function, among all the priority assignments evaluated. The cost function in GDPA is detailed in Section 4.4.1.

The stop condition could be extended by allowing further iterations after a schedulable priority assignment has already been found. This may enable finding solutions with lower cost values. We leave the study of such capability outside of the scope of this work, in order to show the ability of GDPA of finding a schedulable solution more clearly.

4.4. Gradient computation

The objective of this phase is to obtain the gradient of the cost function at the current priority assignment. To achieve this we must first define a suitable cost function. This is described in the following Section 4.4.1. Later, Section 4.4.2 will outline a method to calculate the gradient of the selected cost function.

4.4.1. Cost function

In this section, we define a suitable cost function that can be employed in the GDPA algorithm. In general, the cost function should have as input the parameters we aim to optimize, i.e., we want to find the input at which the cost function is minimized. Moreover, the cost function should represent a metric that we aim to optimize. Accordingly, for the problem of fixed priority assignment, we identify that a cost function f suitable for GDPA should comply with the following 2 requirements:

1. The input of the cost function should be a priority assignment: the cost function maps each priority assignment to a cost value.
2. The cost function should inversely reflect the *schedulability* of the system: lower values of the cost should indicate a better *schedulability* situation. Therefore, by minimizing the cost function we are effectively maximizing the *schedulability*. Although the schedulability status of a system is binary (it is either schedulable or not), here *schedulability* refers to a hypothetical continuous value that quantifies how close (or far) the system is to become schedulable in terms of the distance between its WCRTs and its deadlines.

By applying Gradient Descent with a hypothetical cost function f with the characteristics described above, we would iteratively find new priority assignments that could potentially converge towards a minimum of the cost function, that is, a maximum of the schedulability.

To define the cost function, we will leverage the worst-case response times (WCRT) of the end-to-end flows, as these provide the clearest indication of the *schedulability* of the system. Therefore, we assume the availability of a response-time analysis that is able to calculate the WCRT of every e2e flow.

Given a set of worst-case response times for each e2e flow, $R = [R_1, \dots, R_N]$, a straightforward cost function we could consider is the average WCRT of the system, as this function seems to comply with the 2 requirements set above: (1) for any given priority assignment we get one cost value (i.e. the average WCRT), and (2), lowering the cost function (i.e. lowering the average WCRT) seems to indicate a better *schedulability*. The problem of using the average WCRT as the cost function is that it does a poor job reflecting the overall *schedulability* of the system, as it does not take into account the deadlines. Each iteration of the Gradient Descent algorithm would tend to lower the WCRTs of every e2e flow, regardless of the schedulability status of each particular flow.

Instead, in this paper we use as cost function a metric we call the *inverse slack*, or $invslack$, which we define as follows:

$$invslack(\Pi) = \max_{\forall i} \left(\frac{R_i - D_i}{D_i} \right) \tag{6}$$

where Π is a priority assignment, R_i is the WCRT of flow Γ_i computed for the given priority assignment Π, and D_i is the end-to-end deadline of the same e2e flow.

The main property of $invslack$ is that it focuses on the *worst* flow, that is, the flow with the largest $(R_i - D_i)$ value. Hence, a positive value of $invslack$ indicates that at least one end-to-end flow is not meeting its deadline. On the other hand, a negative value of $invslack$ signals that every flow is meeting its deadline. Additionally, aiming to minimize $invslack$ will tend to increase the schedulability of the worst flow by possibly trading off some of the schedulability of other flows, that were in a better situation. With this approach, at each iteration GDPA will try to improve the worst flow (which may be different each iteration), until the worst flow becomes schedulable, at which point the system as a whole is by extension also schedulable.

4.4.2. Calculating the gradient of the cost function

Once a suitable cost function has been selected, the objective now is to specify a method to calculate its gradients. The cost function has as its input parameter a priority assignment Π, which assigns a priority value to each step in the system. Although this section will focus on calculating the gradient of the $invslack$ cost function, to simplify the notation, in the following we will denote the cost function as f.

As described previously, the gradient of cost function f can be computed as a vector of the partial derivatives of the cost function with respect to each of its parameters. In the case of $invslask$, its parameters are the priorities of each step in the system (P_{ij}). Therefore, the gradient of the cost function f at a given priority assignment Π can be represented as the following vector:

$$\nabla f(\Pi) = \left[\frac{\partial f}{\partial P_{11}}(\Pi), \frac{\partial f}{\partial P_{12}}(\Pi), \dots \right] \tag{7}$$

To calculate these partial derivatives we start by studying the classical definition:

$$\frac{\partial f}{\partial P_{ij}}(\Pi) = \lim_{h \to 0} \frac{f(P_{11}, \dots, P_{ij} + h, \dots) - f(P_{11}, \dots, P_{ij}, \dots)}{h} \tag{8}$$

Eq. (8) implies that to calculate one partial derivative, the cost function must be computed twice: one time with the current priority assignment Π, and another with a different priority assignment in which P_{ij} is increased by an infinitesimal value h. For this equation to be useful, function f must be differentiable around input value Π. Intuitively, this property requires that infinitesimal changes in a priority assignment should induce a change in the output of the cost function. It is trivial to confirm that this property does not hold for $invslack$, as its input (i.e. priorities) have effectively discrete values.

To illustrate the problem, consider a simple system composed of two steps τ_{11} and τ_{21}, located in the same processor. Let us assume a priority assignment $\Pi = [1, 2]$, that is, τ_{11} has a lower priority than τ_{21}.

Let us also assume a cost value for Π equal to X, that is $f(\Pi) = X$. Let us now make an *infinitesimal* change on the priority assignment, obtaining $\Pi' = [1.001, 2]$. Although the priority values have changed, the actual priority ordering remains the same, therefore the cost value also remains unchanged: $f(\Pi') = X$. Consequently, if we were to use Eq. (8) to calculate the partial derivatives, the gradients would probably always be 0, and GDPA would get stuck at the first priority assignment indefinitely.

To circumvent the problem of the non-differentiability of *invslack*, we will approximate its partial derivatives using a non-infinitesimal and constant value for h, which we rename H. We define H as the average priority value separation between consecutive steps inside flat priority vector Π. Formally, H is calculated as follows:

$$H = \lambda \frac{\sum_{i=0}^{N-1} |\Pi_{i+1} - \Pi_i|}{N-1} \tag{9}$$

where N is the number of steps in the system, Π_i is the priority value for the step located at position i in the priority assignment vector Π, and λ is a hyper-parameter to control the size of H.

We modify Eq. (8) to take into account the non-infinitesimal value H. The resulting equation to approximate the partial derivatives is depicted in Eq. (10). By applying a larger non-infinitesimal step size $2H$ ($+H$ to $-H$), this equation will have a greater chance of changing the priority ordering of the steps, and thus providing a non-zero value for the partial derivatives. It is worth noting that, although Eq. (10) is not a formal partial derivative, to simplify the notation we still denote it as such, as it approximately quantifies the slope between two different priority assignments.

$$\frac{\partial f}{\partial P_{ij}}(\Pi) = \frac{f(P_{11}, \ldots, P_{ij} + H, \ldots) - f(P_{11}, \ldots, P_{ij} - H, \ldots)}{2H} \tag{10}$$

According to Eq. (10), to calculate the partial derivative of f with respect to the priority of step τ_{ij}, the cost function must be computed for two different priority assignments: one in which the priority of step τ_{ij} is increased by H, and another in which its priority is decreased by H. Consequently, to calculate the gradient of a system composed of N steps, the cost function must be computed for $2N$ different priority assignments.

It is worth noting that each computation of the cost function requires invoking a response time analysis. Therefore, calculating the gradient of said system composed of N steps requires executing $2N$ analyses. As an example, let us consider a realistic scenario in which 10 iterations of GDPA are executed with a system composed of 100 steps. Under this scenario, considering that in each iteration the gradient will be computed once, in total the response time analysis will be invoked 2000 times.

Any response time analysis compatible with the model presented in Section 2 can be used to compute the cost function, but taking into account that it may potentially be invoked on numerous occasions, it is preferable to select an analysis that tends to be fast, such as the Holistic analysis [15].

It is important to highlight that the response time analysis to compute the gradients, and the response time analysis to determine the stop condition (Section 4.3), do not need to be the same. This property can be exploited by employing a fast analysis for the computation of the gradients, and a slower but more precise analysis to determine the stopping conditions. This can be useful if the results of the fast analysis are correlated with those of the slower but more precise one.

This potential for a high number of invocations of the response time analysis represents the main bottleneck of the GDPA algorithm. To manage this, Section 5 presents a method to accelerate the computation of the gradients by vectorizing a response time analysis technique.

4.5. New priority assignment

In GDPA, instead of utilizing Eq. (2) directly to calculate the next priority assignment Π^{t+1}, we employ the more flexible concept of *update vector*, which abstracts away the *learning rate* and gradient terms. Accordingly, the new priority assignment is calculated by adding the update vector U^t to the current priority assignment Π^t, as shown in Eq. (11).

$$\Pi^{t+1} = c\left(\Pi^t + U^t\right) \tag{11}$$

where t is the current iteration number, Π^t is the current priority assignment, U^t is the *update vector* in the current iteration, and c is the compression function (described in Eq. (4)).

The inclusion of the update vector into the formulation facilitates the incorporation of gradient optimization techniques that will increase the chances of finding the global minimum of the cost function. The field of machine learning, in which the Gradient Descent algorithm is extensively used, has proposed several such optimizations [9]. For this paper we have selected two: Gradient Noise [31], and the Adam optimizer [32]. The update vector U^t is constructed by sequentially applying both techniques. In the following, we provide a more detailed explanation of each technique, and how its notation is adapted to GDPA.

4.5.1. Gradient noise

The Gradient Noise technique adds a Gaussian noise with mean 0 and variance σ_t^2 to the gradient. In a given GDPA iteration t, we denote as G^t the gradient vector with the added noise as follows:

$$G^t = \nabla^t f\left(\Pi^t\right) + \mathcal{N}\left(0, \sigma_t^2\right) \tag{12}$$

where $\mathcal{N}$ denotes a normal or Gaussian distribution.

The variance of the Gaussian noise decays with the iterations of the optimization process, as given in Eq. (13), in which η is the *learning rate*, N is the number of steps in the system, and γ is an additional hyper-parameter to control the noise decay:

$$\sigma_t^2 = \frac{\eta}{(1+N+t)^\gamma} \tag{13}$$

Parameter N in Eq. (13), which was not included in the original formulation of Gradient Noise, is added to modulate the effect of the noise in systems with many steps. In such systems, less noise is required to induce slight variations in the priority ordering of the steps, considering that the priority values of all the steps always get compressed into the range $[-1, 1]$.

4.5.2. Adam optimizer

Adam is a momentum based gradient optimizer that effectively computes specific learning rates for each optimization parameter. It defines two vectors, m and v, which are the first and second moment of the gradient respectively. In a given iteration t, these vectors are defined as follows:

$$\begin{aligned} m^t &= \beta_1 m^{t-1} + (1-\beta_1) G^t \\ v^t &= \beta_2 v^{t-1} + (1-\beta_2)(G^t)^2 \end{aligned} \tag{14}$$

where β_1 and β_2 are hyper-parameters to control the effect of the momentum, and $(G^t)^2$ is the element-wise square of the noisy gradient.

It is worth noting that in this formulation we are directly optimizing the noisy gradient G^t. The vectors m and v are bias-corrected as follows:

$$\begin{aligned} \hat{m}^t &= \frac{m^t}{1-\left(\beta_1\right)^t} \\ \hat{v}^t &= \frac{v^t}{1-\left(\beta_2\right)^t} \end{aligned} \tag{15}$$

Algorithm 1 Gradient Descent Priority Assignment algorithm

Input: Input system **S**, initial priority assignment Π^{ini}, maximum iterations t_{max}
Output: Best priority assignment

```
t ← 0                                          ▷ iteration index
Π ← compress(Π^ini)                            ▷ priority assignment
Π^b ← Π                                        ▷ best priority assignment
best ← ∞                                       ▷ best cost value found
while true do
    value ← cost(Π)                            ▷ current cost value
    if value < best then
        best ← value                           ▷ record best cost value
        Π^b ← Π                                ▷ save best priority assignment
    end if

    schedulable ← analysis(S, Π)               ▷ Schedulability test
    if schedulable or t ≥ t_max then
        break                                  ▷ stop when schedulable or max. iterations
    end if

    ∇(Π) ← gradient(S, Π)                      ▷ gradient
    U ← optimize(∇(Π))                         ▷ update vector
    Π ← compress(Π + U)                        ▷ new priority assignment
    t ← t + 1
end while
return Π^b                                     ▷ return priority assignment with lowest cost
```

Table 1
Parameters of the illustrative example.

	C_{ij}	P_{ij}	*proc.*	T_{ij}	D_{ij}
τ_{11}	5	1	1	30	
τ_{12}	2	2	2		
τ_{13}	20	3	3		35
τ_{21}	5	1	3	40	
τ_{22}	10	2	2		
τ_{23}	10	1	1		45

4.5.3. Update vector

The bias-corrected vectors described in Eq. (15) are combined as in the original formulation of the Adam optimizer [32] to construct the update vector as follows:

$$U^t = -\frac{\eta}{\sqrt{\hat{v}^t} + \epsilon}\hat{m}^t \tag{16}$$

where η is the learning rate, and ϵ is a hyper-parameters to control the update vector.

Finally, given Eqs. (11) and (16), the next priority assignment Π^{t+1} is calculated as follows:

$$\Pi^{t+1} = c\left(\Pi^t - \frac{\eta}{\sqrt{\hat{v}^t} + \epsilon}\hat{m}^t\right) \tag{17}$$

It is important to note that, as in the original Adam formulation, in this equation the vector operations are meant to be performed element-wise.

To summarize, Algorithm 1 shows the pseudo-code of GDPA.

4.6. Illustrative example

We present here a simple example to illustrate how GDPA can be applied on a particular system. The example is adapted from [15], and is composed of 2 e2e flows with 3 steps each, for a total of 6 steps, which traverse 3 processing resources. The basic parameters of the system are shown in Table 1, including the initial priority assignment. For this example we use the Holistic analysis [15] as the WCRT analysis to both determine the schedulability of the system and to compute the gradients. For illustration purposes, in this example we use the following GDPA hyper-parameter values: $\lambda = 1.5$, $\eta = 3$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 0.1$. Section 6.3.1 provides a justification for these values, which were obtained empirically.

GDPA starts by compressing the initial priority assignment. The result of this compression is shown in the second column of Table 2, labeled Π^0. The first column, labeled Π^{ini} shows the uncompressed initial priority assignment.

The next phase of GDPA requires evaluating the Stop Condition, which involves determining whether assignment Π^0 is already schedulable. By applying the Holistic analysis, it is determined that Π^0 is not schedulable, with an initial cost value $invslack = 9.33$. Therefore, GDPA continues to the next phase, the Gradient Computation.

As shown in Eq. (7), the gradient is a vector of the partial derivatives of the priorities of each step (i.e. task). In GDPA, each partial derivative is calculated using Eq. (10), which involves computing the cost function twice, using a non-infinitesimal value H. Eq. (9) is used to calculate H, which is the average separation of the priority values of Π^0, which in this case is $H = 0.6$.

In this example composed of 6 steps, the gradient requires a total of 12 computations of the cost function, each with a different priority assignment. Each of these priority assignments is obtained by summing (or subtracting) value H to the priority of one task. The resulting 12 priority assignments are depicted in the Table 2, from column 3 onwards. The columns labeled $P_{ij} + H$ contain the priority assignment in which the priority of step τ_{ij} is increased by H. Similarly for the columns labeled $P_{ij} - H$.

Next, the cost function is evaluated for each one of those 12 priority assignments. The resulting cost values are depicted in the first 2 columns of Table 3. A cell located at row τ_{ij} and column $f(+H)$ contains the value of cost function *invslack* for a priority assignment in which the priority of τ_{ij} was increased by H. Similarly for the column labeled $f(-H)$. The third column of Table 3 (labeled $\partial f/\partial P_{ij}$), shows the final partial derivative values for each step, obtained by applying Eq. (10). The whole column represents the elements of the gradient $\nabla f(\Pi^0)$.

In a simple Gradient Descent algorithm, the next priority assignment Π^1 would be calculated by adding the gradient (last column of Table 3) scaled by a learning rate to the current priority assignment Π^0. This is represented in Eq. (2). In GDPA, we optimize the gradient by adding a decaying noise and applying the Adam optimizer, as explained in Section 4.5. The results of this process are shown in Table 4, which are decomposed as follows: (1) column Π^0 is the current priority assignment at iteration t=0, (2) column U^0 is the resulting update vector after applying Gradient Noise and Adam, (3) column Π^{1*} is the non-compressed new priority assignment which is computed as the summation of Π^0 and U^0, and (4) column Π^1 is the new priority assignment, which results from compressing column Π^{1*}.

GDPA continues by evaluating the Stop Condition on Π^1. The holistic analysis now deems this new priority assignment as schedulable, with a cost value $invslack = -0.09$. Therefore, GDPA now stops and returns Π^1 as the best priority assignment it has found.

5. Accelerating the gradient computation

In this section, we aim to improve the computation times of the GDPA algorithm. In Section 4 we identified that the Gradient Computation phase of GDPA is its main computational bottleneck, as it requires computing the cost function twice per step in the system, which by extension requires invoking the response time analysis. In total, for a system with N steps, each iteration of GDPA requires $2N$ invocations of a response time analysis.

Also in Section 4.4.2 we observed that each of the invocations of the response time analysis to compute a gradient differs only in the priority assignment it is analyzing. That is, for a system with N steps,

Table 2
Example of the priority assignments involved in one iteration of GDPA.

	Π^{ini}	Π^0	$P_{11}+H$	$P_{11}-H$	$P_{12}+H$	$P_{12}-H$	$P_{13}+H$	$P_{13}-H$	$P_{21}+H$	$P_{21}-H$	$P_{22}+H$	$P_{22}-H$	$P_{23}+H$	$P_{23}-H$
τ_{11}	1	0.33	0.93	−0.27	0.33	0.33	0.33	0.33	0.33	0.33	0.33	0.33	0.33	0.33
τ_{12}	2	0.67	0.67	0.67	1.27	0.07	0.67	0.67	0.67	0.67	0.67	0.67	0.67	0.67
τ_{13}	3	1.00	1.00	1.00	1.00	1.00	1.60	0.40	1.00	1.00	1.00	1.00	1.00	1.00
τ_{21}	1	0.33	0.33	0.33	0.33	0.33	0.33	0.33	0.93	−0.27	0.33	0.33	0.33	0.33
τ_{22}	2	0.67	0.67	0.67	0.67	0.67	0.67	0.67	0.67	0.67	1.27	0.07	0.67	0.67
τ_{23}	1	0.33	0.33	0.33	0.33	0.33	0.33	0.33	0.33	0.33	0.33	0.33	0.93	−0.27

Table 3
Cost values, and coefficients of the resulting gradient.

	$f(+H)$	$f(-H)$	$\partial f/\partial P_{ij}$
τ_{11}	2.82	9.33	−5.43
τ_{12}	3.84	9.33	−4.57
τ_{13}	9.33	9.33	0.00
τ_{21}	9.33	9.33	0.00
τ_{22}	9.33	3.84	4.57
τ_{23}	9.33	2.82	5.43

Table 4
New priority assignment.

	Π^0	U^0	Π^{1*}	Π^1
τ_{11}	0.33	1.27	1.61	0.83
τ_{12}	0.67	1.27	1.94	1.00
τ_{13}	1.00	−1.26	−0.26	−0.13
τ_{21}	0.33	1.27	1.60	0.83
τ_{22}	0.67	−1.27	−0.61	−0.31
τ_{23}	0.33	−1.27	−0.94	−0.48

$2N$ *known* priority assignments are evaluated to calculate a gradient. Here we exploit this observation by proposing a method to effectively calculate the WCRTs of all the required priority assignments at the same time.

We propose a strategy that can be summarized in the following two points:

1. Selection of an efficient response time analysis to calculate the gradients. To our knowledge, the Holistic analysis [15] is the fastest analysis available that is compatible with the system model laid out in Section 2.
2. Vectorization of the Holistic analysis for a faster execution. Moreover, 3D matrices will be employed to effectively analyze several priority assignments at the same time.

A vectorization is a process that involves minimizing the use of loop and conditional operations in a particular piece of code, replacing them with vector and matrix operations. This generally enables a faster execution, as vector operations are typically optimized in modern CPUs, especially when an appropriate library is used. Examples of such libraries implement the BLAS specification [33] such as OpenBLAS [34]. Higher level libraries such as Numpy [35] for Python rely on efficient BLAS libraries.

In this section we simplify the step notation to include just one index, that is, we denote the *i-th* step as τ_i. Following this notation, Eq. (18) depicts the main equation of the Holistic Analysis, for a step τ_a under analysis. Here we follow the formulation presented in Palencia et al. [15].

$$w_a^{n+1}(p) = pC_a + \sum_{\forall b \in hp(a)} \left\lceil \frac{J_b + w_a^n(p)}{T_b} \right\rceil C_b \tag{18}$$

where:

- $hp(a)$ is the set of steps that can preempt step τ_a. A step τ_b can preempt τ_a if both are located in the same processor, and $P_b > P_a$.
- J_b is the jitter of step τ_b, which in the Holistic analysis is simplified as the worst-case response time of the previous step in its e2e flow.
- p is the index of the current instance of the step under analysis, as more than once instance of the same step must be taken into account when deadlines are higher than the periods. The first instance is given a value p=1.
- T_b is the period of step τ_b, which is equal to the period of its end-to-end flow

A typical implementation of the Holistic analysis embeds Eq. (18) inside 3 loops: (1) an inner loop to solve recursive value w_a of the equation for a given p value, (2) a middle loop that iterates the value p and registers the activation number that incurred in the longest response time, and (3) an outer loop that updates the jitters of every step according to the currently found WCRT, and stops when two consecutive outer loops reach the same WCRT. Ideally all these loops should be replaced by pure vector operations, however due to the complexity of this endeavour, in this paper we will focus on vectorizing just Eq. (18), keeping the resulting vectorization inside the same 3 aforementioned loops. In the evaluation section (Section 6), we will show how this limited vectorization still produces sizeable speed-ups.

To vectorize Eq. (18) we employ a two-pronged approach. First, we replace every step attribute in the equation (e.g. C_a, etc.), by an equivalent vector that contains the attribute of every step. Accordingly, we define the following three vectors:

$C = [C_1, C_2, \ldots]$ WCETs vector
$J = [J_1, J_2, \ldots]$ Jitters vector
$T = [T_1, T_2, \ldots]$ Periods vector

Second, we implement the summation in Eq. (18) with a pure matrix multiplication. To achieve this we leverage the concept of a *priority matrix*. We define each element pm_{ab} of a priority matrix PM as follows:

$$pm_{ab} = \begin{cases} 1 & \text{if step } \tau_b \text{ can preempt step } \tau_a \\ 0 & otherwise \end{cases} \tag{19}$$

The idea of the priority matrix is also used in the linearization of schedulability tests, to codify them as linear constraints for MILP [8]. Essentially, a priority matrix contains all the necessary priority ordering information of one priority assignment Π. We propose extending this priority matrix to include the priority information of several priority assignments at the same time.

We define Ψ as a set of M priority assignments as follows:

$$\Psi = \left[\Pi^1, \Pi^2, \ldots, \Pi^M\right] \tag{20}$$

where Π^m is the *m-th* priority assignment.

It is important to note that the set Ψ does not assume or impose any type of relation among the priority assignments it contains. Specifically, the index m is unrelated to the iteration index used in Section 4 to label the priority assignment in a given GDPA iteration.

In this paper, we propose constructing a 3D priority matrix, by stacking together the priority matrices of several priority assignments. We denote this extended priority matrix as HP. For a given set Ψ, we define HP as a binary 3D matrix, in which each of its elements hp_{mab} is defined as follows:

$$hp_{mab} = \begin{cases} 1 & \text{if step } \tau_b \text{ can preempt step } \tau_a \text{ in the} \\ & \text{priority assignment } \Pi^m \\ 0 & \text{otherwise} \end{cases} \quad (21)$$

Intuitively, we can visualize each slice of 3D matrix HP as containing one 2D priority matrix PM.

We can now vectorize Eq. (18) employing the priority matrix HP as follows:

$$W^{n+1}(p) = pC + HP \times \left\lceil \frac{J^T + W^n(p)}{T^T} \right\rceil C \quad (22)$$

where p is the step instance index, J^T and T^T are the transposed Jitter and Period vectors respectively, and W is a matrix in which each of its elements w_{ma} contains the value w for step τ_a (i.e. w_a) associated with priority assignment Π^m.

The recursive Eq. (22) is solved iteratively, with an initial W matrix initialized to zero, stopping when two consecutive iterations yield the same results, that is, when $W^{n+1}(p) = W^n(p)$. As in the original formulation, the p value is initialized to 1.

The original Holistic analysis provides the equations to derive the WCRT of a step from its w value. These equations can be employed directly here too, obtaining the WCRT of every step, and for every priority assignment, at the same time. It is worth noting that Eq. (22) assumes that any matrix shape incompatibilities are automatically solved, for example by employing *broadcasting* [35]. If such capabilities are not available, then the vectors should be manually padded to make their dimensions compatible.

Eq. (22) can be exploited in GDPA to compute a gradient of the cost function with just 1 invocation of the Holistic analysis. This can be achieved by constructing a Ψ set containing all the priority assignments needed to compute a gradient (Eqs. (7) and (8)). As a result of this analysis, the WCRT of every flow in every priority assignment is obtained, which can be used to calculate all the partial derivatives that compose the gradient (Eq. (8)).

To better contextualize the effect of this acceleration method, we can study the computational complexity of GDPA before and after the vectorization process, with respect to the number of tasks n. The complexity of the original Holistic analysis (i.e. before vectorization) can be bounded to $O(n^2)$, as it nests two loops that iterate the whole task-set: one to determine which tasks have higher priority (the summation shown in Eq. (18)), and another outer loop that iterates each task. Considering that GDPA must execute the analysis $2n$ times in each iteration to compute the gradient, this yields a total complexity of GDPA of $O(n^3)$.

Although the vectorized Holistic Eq. (22) has $O(1)$ complexity, there are previous setup stages that constructs set Ψ (Eq. (20)) and the necessary vectors and matrices, which can be computed with linear complexity $O(n)$. Therefore, the whole complexity of GDPA with the vectorized analysis can be bounded to $O(n)$.

It is common for a vectorization process to trade off computational complexity with a space complexity penalty. In our case, GDPA without the vectorization optimization offers a $O(n)$ space complexity, while the vectorized optimization of GDPA has a $O(n^3)$ space complexity, due to 3D matrix HP (Eq. (21)), which has a size $(n \times n \times 2n)$.

6. Evaluation

In this section we present the evaluation results of the GDPA algorithm. This section is organized as follows: in Section 6.1 we provide implementation details of GDPA and other algorithms that we have implemented for this comparison; in Section 6.2 we study the execution time speed-ups obtained by vectorizing the Holistic analysis; and in Section 6.3 we evaluate the GDPA algorithm by analyzing its ability to find schedulable solutions and its computation times.

6.1. Implementation details

We implemented GDPA in Python, with the code publicly available in a Github repository [36]. Although this implementation was created to evaluate GDPA, it is meant to be extensible. As such, it provides classes to model real-time systems and interfaces to implement algorithms other than GDPA.

Two versions of the Holistic analysis were implemented: a standard non-vectorized version directly following the original formulation [15], and another vectorized version adopting the methodology described in Section 5. The latter leverages the Numpy [35] numerical library to efficiently perform the vector operations.

Apart from GDPA, the PD and HOPA priority assignment algorithms were also implemented. Moreover, two optimal fixed priority assignments were implemented: a brute-force algorithm, and another one employing MILP. We consider an algorithm as optimal if it is always capable of finding a schedulable priority assignment if such an assignment exists.

The brute-force algorithm simply evaluates every possible priority ordering. To accelerate this process, it leverages the vectorized Holistic analysis by analyzing at the same time batches of 10000 priority assignments. Nevertheless, this brute-force algorithm is still intractable for all but small systems. For instance, a system composed of 5 processors, with 10 steps mapped to each processor, offers $10!^5 \approx 6.29 \times 10^{32}$ possible priority assignments, which is not feasible even when taking advantage of the vectorized analysis.

For the MILP algorithm we faced the challenge of the unavailability of feasible linear equations to describe a schedulability test for our system model. Because of this, our implementation of a priority assignment algorithm with MILP only defines restrictions that declare when a fixed priority assignment is *valid*. For this, it employs a square binary priority matrix, defined with Eq. (19), with the same linear restrictions as in [8]. Each *valid* priority assignment is evaluated externally using a callback, which invokes a schedulability test to determine whether the *valid* priority assignment is also schedulable. The algorithm is terminated as soon as a schedulable priority assignment is found. This MILP algorithm is implemented with Gurobi [28]. This implementation of MILP is still optimal (i.e. it will find a schedulable solution if one exists), but may incur in a computation time penalty due to the unavailability of feasible linear constraints that completely support our system model. The definition of such linear constraints falls outside the scope of this paper.

To widen the availability of response time analyses, a bridge between Python and the MAST tool [18] was implemented too. This bridge automatically performs the transformation between Python model classes and MAST input and output files, and the execution of the MAST tool executable on those files. This bridge allows any algorithm implemented in MAST to be invoked transparently directly from Python code. This bridge enables an Offset-Based analysis technique implemented in MAST to be used in the evaluation of the stop condition in GDPA.

We also implemented an automatic generator of synthetic systems. It generates random systems given a number of e2e flows, steps per flow, processors, processor utilization, and the selection ranges for the periods and deadlines. To compute the WCETs, it employs the widely used UUNIFAST algorithm [37], and the periods are selected using a log-uniform distribution. The specific characteristics of the synthetic systems are described within each evaluation (Sections 6.2 and 6.3).

The utilization of a processing resource PR_k, which we denote as L_k, is defined with Eq. (23). The system utilization is defined as the average utilization of its processing resources.

$$L_k = \sum_{\forall \tau_{ij} \in PR_k} \frac{C_{ij}}{T_i} \quad (23)$$

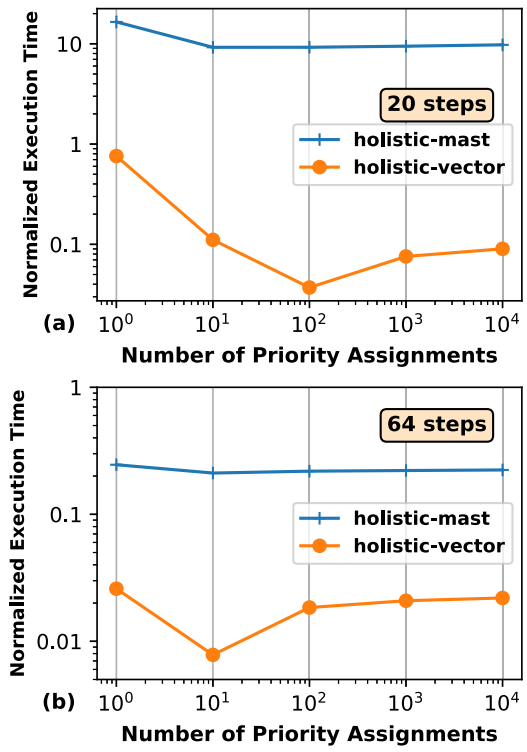

Fig. 3. Total execution times of analyzing a set of priority assignments, normalized to the execution times of *holistic*, for systems with (a) 20 steps, and (b) 64 steps.

6.2. Vectorization speed-ups

The objective here is to measure the execution time speed-ups gained by vectorizing the Holistic analysis as described in Section 5. For this, we measure the total execution times of analyzing 1, 10, 100, 1000 and 10000 priority assignments, comparing three different approaches:

- *holistic*: each priority assignment is analyzed sequentially and independently with the Holistic analysis implemented in Python with no vectorization (i.e. Numpy is not used).
- *holistic-mast*: each priority assignment is analyzed sequentially and independently with the Holistic analysis provided by the MAST tool, leveraging the MAST-Python bridge. The analysis in MAST is written in Ada and compiled into native executables.
- *holistic-vector*: all the priority assignments are analyzed at the same time leveraging the capabilities of the vectorized Holistic analysis described in Section 5, employing the Numpy library.

We generate synthetic systems with two different total numbers of steps: 20 steps and 64 steps. Every system has a utilization of 70% in every processor. The priority assignments are randomized, but the same ones are analyzed in the 3 approaches. The measured execution times are shown in Fig. 3 for both system sizes. The execution times are normalized to those of *holistic*.

For systems with 20 steps (Fig. 3(a)), we can observe that when only 1 priority assignment is analyzed, the vectorized analysis offers a slight speed-up over the non-vectorized version. However, as the number of priority assignments increases, the benefits of the vectorization become clear, with total execution times that are 10 times lower than *holistic*. On the other hand, analyzing with MAST (*holistic-mast*) incurs in execution times 10 times higher than the baseline (*holistic*). These overheads are due to the necessity to write and read a file each time a priority assignment is analyzed with MAST.

For larger systems with 64 steps (Fig. 3(b)), the vectorized analysis is still clearly the faster approach, with execution times that are always more than one order of magnitude faster than *holistic*. Similarly, *holistic-mast* also shows execution times around 0.2 times those of *holistic*. Here the systems are more complex and require more computations to be analyzed. Therefore the native compiled binary of MAST clearly becomes faster than the non-vectorized Python implementation, even considering the necessary file read and write overheads.

6.3. GDPA evaluation

The objective is to compare GDPA against other priority assignment algorithms. We will measure two aspects in this evaluation: the ability of each algorithm to find schedulable solutions, and their respective computation times.

In total the evaluation compares seven priority assignment algorithms, including GDPA with three different initializations:

- *pd*: the non-iterative PD priority assignment [23].
- *hopa*: the HOPA iterative algorithm [7].
- *brute-force*: the brute force algorithm described in Section 6.1.
- *milp*: our *ad-hoc* implementation of a MILP algorithm, described in Section 6.1.
- *gdpa-random*: GDPA with an initial random priority assignment.
- *gdpa-pd*: GDPA with an initial PD priority assignment.
- *gdpa-hopa*: GDPA with an initial HOPA priority assignment.

The evaluation will be performed on a pool of synthetic systems, which were created taking into account the characteristics of publicly available real use cases. For instance, the authors of [38] provide a description of a cruise-control system composed of 11 tasks divided into 2 e2e flows that traverse 2 processing resources. Furthermore, the generic military avionics system described in [39] is composed of 23 tasks running on 2 processing resources. Thus, we generated synthetic systems with 3 different sizes: 16 steps (4 flows with 4 steps each, 4 processors); 30 steps (6 flows with 5 steps each, 5 processors); and 72 steps (12 flows with 6 steps each, 7 processors). Utilizations are swept up from 50% to 90% with 20 intermediate utilizations. The WCETs are generated with UUNIFAST. The step to processor mapping keeps the load balanced, resulting in all processors having the same utilization. Periods are randomly selected in the range $[100, 100000]$ using a log-uniform distribution. The end-to-end deadline of each flow Γ_i is randomly selected in the range $[0.5 \cdot T_i \cdot N_i, T_i \cdot N_i]$. To obtain relevant results, 50 systems were generated for each configuration.

To contextualize the problem, for the systems with 16 steps and 4 processors, there are $4!^4 = 331776$ possible priority assignments. For the systems with 30 steps and 5 processors there are $6!^5 \approx 1.93 \times 10^{14}$ possible priority assignments. Finally, for the largest systems, which contain 72 steps in 7 processors, the search space is composed of $(72/7)!^7 \approx 9.39 \times 10^{47}$ candidate priority assignments. Consequently, the optimal algorithms (*brute-force* and *milp*) were only applied to the systems with 16 steps. For the other system sizes, which offer larger search spaces, these algorithms become intractable.

Unless otherwise specified, we employ the Holistic analysis to determine the schedulability of a priority assignment. In the case of GDPA, we always employ the vectorized Holistic analysis to compute the gradients. The initial priority assignment in HOPA is performed with PD. The maximum number of iterations was set to 100 for GDPA and 120 for HOPA. A common configuration of HOPA was used: $k_a = [1, 1.8, 3, 1.5]$ and $k_r = [1, 1.8, 3, 1.5]$.

This evaluation is organized into 3 sections. Section 6.3.1 performs a preliminary evaluation in which appropriate values for the hyperparameters are determined. Also, the individual contributions to the performance of GDPA of Adam and Gradient Noise are quantified. Section 6.3.2 evaluates the ability of each priority assignment algorithm of finding schedulable solutions. Finally, Section 6.3.3 evaluates the computation times incurred by each of the algorithms under study.

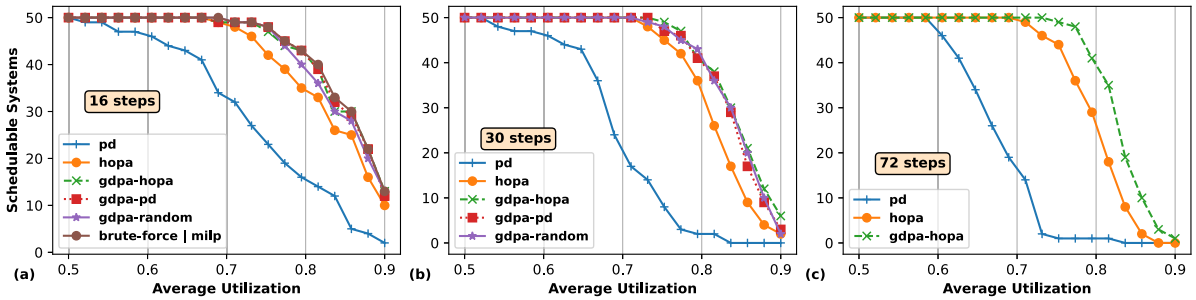

Fig. 4. Number of schedulable solutions found, for systems with (a) 16 steps, (b) 30 steps, and (c) 72 steps.

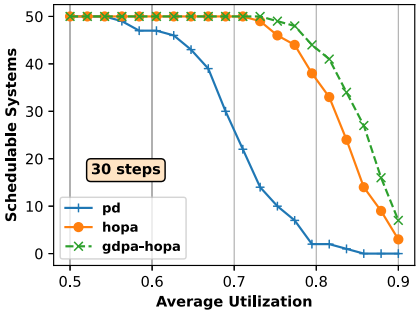

Fig. 5. Number of schedulable solutions found, applying an Offset-Based analysis to evaluate the Stop Condition.

Table 5
Number of schedulable solutions found for different hyper-parameters, with *gdpa-hopa*.

β_1	β_2	η	λ	ϵ	γ	Schedulable solutions
0.9	0.999	3	1.5	0.1	0.9	867
0.7	0.999	3	1.5	0.1	3	867
0.9	0.7	3	1.5	0.1	0.9	861
0.9	0.999	3	1.5	0.1	1.5	860
0.9	0.999	3	2	0.01	0.9	852
0.9	0.999	0.1	2	0.1	3	844
0.7	0.999	0.1	2	0.01	3	820

6.3.1. Preliminary evaluation

As shown in Section 4, GDPA offers 6 hyper-parameters that can be used to tweak its behavior: β_1, β_2, η, λ, ϵ, and γ. Before carrying out the full evaluation of GDPA, we executed a preliminary evaluation to determine appropriate values for these hyper-parameters.

This preliminary evaluation consisted on executing *gdpa-hopa* (i.e. GDPA with a HOPA initial assignment), with different values for the hyper-parameters, on the small synthetic systems (16 steps). In total, 1000 systems were studied, with utilizations ranging from 50% to 90%. For each hyper-parameters configuration, the total number of systems for which a schedulable solution was found was recorded. Table 5 collects a representative selection of the results obtained. From these results, it can be concluded that the hyper-parameters have a clear impact on the performance of GDPA. Furthermore, we determined that the following values provide good results, and were chosen for all subsequent evaluations: $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\eta = 3$, $\lambda = 1.5$, $\epsilon = 0.1$, $\gamma = 0.9$. It is worth indicating that this preliminary evaluation is limited in scope, and a more comprehensive study is needed to precisely determine the effect of each hyper-parameter, including more types of systems. This is left for future work.

Additionally, we performed a further evaluation to quantify the individual contributions that the Adam optimizer and Gradient Noise have on the performance of GDPA. For this, we tested GDPA with 4 different configurations: (1) standard GDPA with Adam + Gradient Noise, (2) GDPA with just Adam, (3) GDPA with just Gradient Noise, and (4) GDPA with neither Adam nor Gradient Noise. Every configuration starts with a PD assignment, and the same pool of synthetic systems as the previous evaluation was used. (i.e. 1000 systems with 16 steps each). The number of systems for which a schedulable solution was found for each configuration is shown in Table 6. For added context, the results for HOPA are also included. From these results, it can be confirmed that each optimizer (Adam and Gradient Noise) positively contributes to the gradient descent algorithm in its search for a schedulable priority assignment, being the combination of both (i.e. Adam + Noise) the scenario that produces the best recorded performance.

Table 6
GDPA optimizers comparison.

GDPA configuration	Schedulable solutions
gdpa-pd (adam + noise)	864
gdpa-pd (adam)	860
gdpa-pd (noise)	854
gdpa-pd (none)	839
hopa	819

6.3.2. Schedulable solutions

Here we measure how many systems each algorithm was able to schedule successfully, for the different system complexities under study. The results are shown in Fig. 4 for each system size, and given as a function of the average processor utilization. It is worth noting that the maximum number of systems each algorithm can schedule is 50, as this is the number of systems that were generated for each utilization level.

For systems with 16 steps (Fig. 4(a)), we observe that both brute-force and MILP dominate, and also were able to schedule the same number of systems. This is expected, as these 2 algorithms are optimal, in the sense that if a schedulable solution exists, they will find it. We also observe that all the GDPA variants closely approximate the optimal algorithms, clearly outperforming HOPA. It is worth bearing in mind that both *gdpa-pd* and HOPA start with the same PD priority assignment. In the case of *gdpa-random*, it shows slightly worse results than the rest of GDPA variants, but still clearly above HOPA. This indicates that GDPA has the capability of correctly exploring the search space, even when starting from very poor priority assignments.

For more complex systems with 30 steps (Fig. 4(b)), the evaluation shows similar results, although here *gdpa-random* has a performance closer to *gdpa-pd*. The GDPA variants clearly outperform HOPA, with a slight advantage of *gdpa-hopa* for higher utilizations. Here brute-force and MILP algorithms were not applied due to their high computation times. Therefore it is not possible to measure whether there is any room to improve above *gdpa-hopa*. However, considering the differences observed in systems with 16 steps, and in general the maximum schedulable utilizations that are reached with fixed priorities scheduling, we expect that here an optimal algorithm would not be able to schedule many more systems than GDPA.

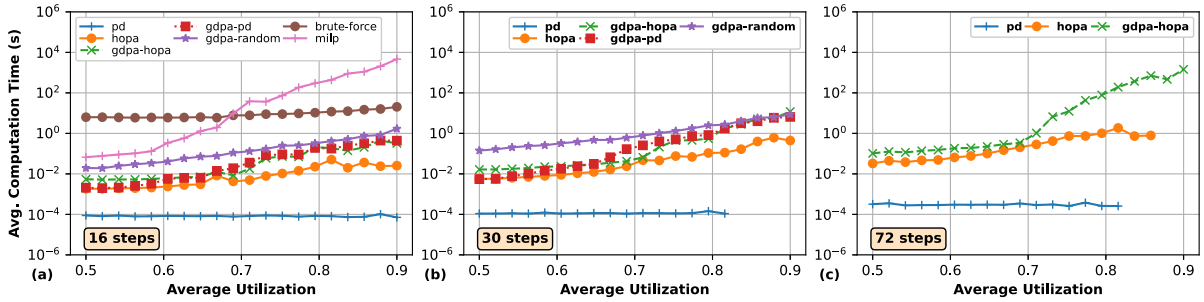

Fig. 6. Average computation time to find a schedulable solution, for systems with (a) 16 steps, (b) 30 steps, and (c) 72 steps.

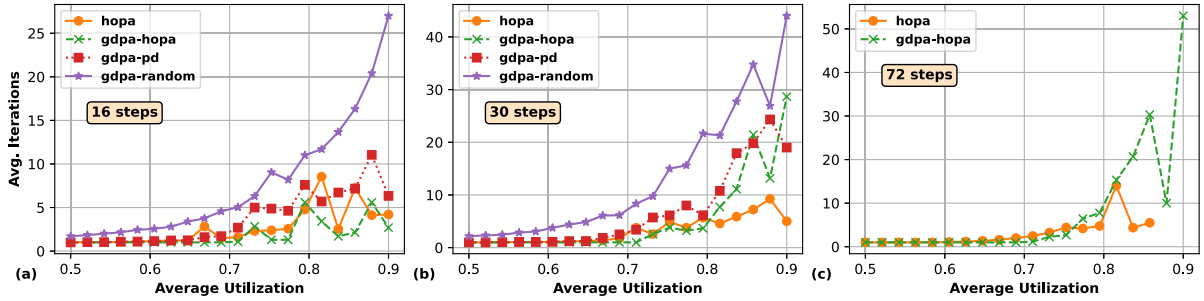

Fig. 7. Average number of iterations required to find a schedulable solution, for systems with (a) 16 steps, (b) 30 steps, and (c) 72 steps.

The same conclusions can be reached for systems with 72 steps (Fig. 4(c)). To bound the computation times of the evaluation, we only applied *gdpa-hopa*, as we have confirmed previously it is the best of the GDPA variants. Here, the difference between *gdpa-hopa* and HOPA is very similar to that observed for the previous system. This indicates that the capability of finding solutions of GDPA is not relatively hindered by increasing the size of the search space.

We now put to test a different configuration of GDPA, in which the stop condition is evaluated using a less pessimistic Offset-Based analysis [16], while keeping the same vectorized Holistic analysis to compute the gradients. The results for systems with 30 steps are shown in Fig. 5. Both PD and HOPA are also using the same Offset-Based analysis. This result depicts a very similar situation to that of Fig. 4(b), with the main difference being that the results are slightly better, equally for all algorithms. This is due to the usage of a less pessimistic analysis. This result validates the idea of employing a fast Holistic-type analysis to direct the optimization process (i.e. to compute the gradients), while using a different less pessimistic but slower analysis to validate the results. Further, this observation also suggests that there is a correlation between the WCRTs of the Holistic analysis and those from the less pessimistic Offset-Based analysis.

6.3.3. Computation times

We have shown that GDPA is able to schedule more systems than HOPA, approximating an optimal algorithm (at least in the situations in which the optimal algorithms are tractable). Here we evaluate the computation times of GDPA in comparison with the other techniques.

Fig. 6 shows the average computation times each algorithm required to find a schedulable solution, for systems with 16, 30 and 72 steps. The computation times of GDPA and HOPA include the times required to calculate the initial priority assignments. For instance, the computation times of *gdpa-hopa* include the computation times required by HOPA to calculate the initial priority assignment.

As a general observation we can confirm that all the GDPA variants required longer computation times than HOPA, specially for systems with utilizations above 70%. This matches the results previously reported in Fig. 4, in which the scheduling improvements of GDPA over HOPA manifested for systems with utilizations higher than 70%. This indicates that the extra computation time required by GDPA is effectively employed in finding more schedulable solutions.

Among the GDPA variants, in Fig. 6(a) we can observe that *gdpa-random* is clearly the slowest. This is expected, as it usually starts with very poor priority assignments, that should require more iterations to improve upon them. Moreover, the reported difference in computation times between *gdpa-pd* and *gdpa-hopa* is not significant.

For systems with 90% utilization, which are the most complex to analyze, GDPA required on average around 1 s, 10 s and 1500 s to find a schedulable solution in the systems with 16, 30 and 72 steps respectively. 90% is a very high utilization that is not usually reached in industrial settings. For the more realistic range up to a utilization of 80%, GDPA required on average 100 s to find a solution in the systems with 72 steps.

Focusing on the available comparison with the optimal algorithms (Fig. 6(a)), we can confirm that all the GDPA variants were significantly faster than both MILP and the *brute-force* algorithms. It is important to note that we have previously showed that for these systems with 16 steps, GDPA demonstrated a capability to find schedulable solutions very close to that of those optimal algorithms (see Fig. 4(a)). Summarizing, although GDPA exhibits scheduling capabilities that are close to optimal, its computational complexity is closer to that of HOPA.

It is important to note that the very high computation times of MILP in this evaluation should not be used to conclude that MILP is not a valid method to assign fixed priorities for systems that follow our system model. Rather, these results signal the need for the definition of linear equations to model a schedulability test with sufficient precision. We believe that, if such equations existed, the performance of MILP here may be greatly improved.

In Fig. 7, we show the average number of iterations each algorithm required to find a schedulable solution. This metric adds more context to the computation times previously reported. The optimal algorithms

are not included here because they are both in essence brute-force algorithms in which the number of iterations required can be considered as arbitrary. Similarly, PD is not included because it is not an iterative algorithm. In general we observe in the figure that GDPA does not need a high number of iterations to find a schedulable solutions. If we focus on the GDPA variants that start with good priority assignments (*gdpa-pd* and *gdpa-hopa*), and on the more complex systems with 90% utilization, we can observe that on average they required less than 5, 30 and 55 iterations for systems with 16, 30 and 72 steps respectively. These results highlight that the current maximum number of iterations set for GDPA (i.e. 100), should not noticeably constrain its ability to find schedulable solutions.

In the Figs. 7(a) and 7(b) (16 and 30 steps respectively), we confirm that *gdpa-random* requires more iterations than the other GDPA variants. This observation reaffirms the conclusion that the longer computation times of *gdpa-random* are due to its need to overcome a worse initial priority assignment.

7. Conclusions and future work

In this paper, we presented a new algorithm to assign fixed priorities in real-time systems, called Gradient Descent Priority Assignment (GDPA). As far as we know, this is the first time a Gradient Descent algorithm has been employed in this particular type of problem.

We evaluated GDPA on a variety of synthetic systems and showed that it is able to find more schedulable solutions than previous custom heuristics such as HOPA. We also showed that GDPA closely approximates the success rate of optimal algorithms, at least in those situations in which those optimal algorithms were tractable. Crucially, the evaluation showed that GDPA achieves this performance while requiring reasonable amounts of computation time. In the more extreme systems tested, with 72 steps and very high processor utilizations of 90%, GDPA was able to find schedulable solutions on average in less than 25 min. For more reasonable utilizations around 80%, which are still considered as high, the computation times were on average less than 100 s for the more complex systems tested with 72 steps.

We are planning to extend this evaluation, to include a deeper study of the effects of the hyper-parameters on a wider set of systems, including more specific system models with multipath e2e flows. Also, the ability of GDPA of optimizing already schedulable solutions will be explored.

As a more general conclusion, this work also hints at the potential of Gradient Descent as an algorithm to optimize real-time systems in general due to its flexibility. In this regard, we plan to generalize the Gradient Descent algorithm, to optimize other parameters such as scheduling deadlines for EDF, or task to processor mapping.

CRediT authorship contribution statement

Juan M. Rivas: Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **J. Javier Gutiérrez:** Writing – review & editing, Writing – original draft, Validation, Methodology, Funding acquisition. **Ana Guasque:** Writing – review & editing, Writing – original draft, Validation, Investigation, Formal analysis. **Patricia Balbastre:** Writing – review & editing, Writing – original draft, Validation, Formal analysis.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was partially supported by MCIN/ AEI/10.13039/ 5011 00011033/ FEDER "Una manera de hacer Europa", Spain under grants PID2021-124502OB-C41 and PID2021-124502OB-C42 (PRESECREL), and by the Vicerrectorado de Investigación de la Universitat Politècnica de Valencia (UPV) "Aid to First Research Projects", Spain under grant PAID-06-23 and PAID-10-20.

References

[1] C.L. Liu, J.W. Layland, Scheduling algorithms for multiprogramming in a hard-real-time environment, J. ACM 20 (1) (1973) 46–61, http://dx.doi.org/10.1145/321738.321743.

[2] B. Akesson, M. Nasri, G. Nelissen, S. Altmeyer, R.I. Davis, A comprehensive survey of industry practice in real-time systems, Real-Time Syst. 58 (2022) 358–398, http://dx.doi.org/10.1007/s11241-021-09376-1.

[3] R.I. Davis, L. Cucu-Grosjean, M. Bertogna, A. Burns, A review of priority assignment in real-time systems, J. Syst. Archit. 65 (2016) 64–82, http://dx.doi.org/10.1016/j.sysarc.2016.04.002.

[4] K.W. Tindell, A. Burns, A.J. Wellings, Allocating hard real-time tasks: An NP-Hard problem made easy, Real-Time Syst. 4 (1992) 145–165, http://dx.doi.org/10.1007/BF00365407.

[5] E. Azketa, J.P. Uribe, M. Marcos, L. Almeida, J.J. Gutierrez, Permutational genetic algorithm for the optimized assignment of priorities to tasks and messages in distributed real-time systems, in: 2011IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications, IEEE, 2011, pp. 958–965, http://dx.doi.org/10.1109/TrustCom.2011.132.

[6] A. Hamann, M. Jersak, K. Richter, R. Ernst, A framework for modular analysis and exploration of heterogeneous embedded systems, Real-Time Syst. 33 (2006) 101–137, http://dx.doi.org/10.1007/s11241-006-6884-x.

[7] J.J. Gutiérrez, M.G. Harbour, Optimized priority assignment for tasks and messages in distributed hard real-time systems, in: Proceedings of Third Workshop on Parallel and Distributed Real-Time Systems, IEEE Comput. Soc. Press, 1995, pp. 124–132, http://dx.doi.org/10.1109/WPDRTS.1995.470498.

[8] Q. Zhu, H. Zeng, W. Zheng, M.D. Natale, A. Sangiovanni-Vincentelli, Optimization of task allocation and priority assignment in hard real-time distributed systems, ACM Trans. Embedd. Comput. Syst. 11 (2013) 1–30, http://dx.doi.org/10.1145/2362336.2362352.

[9] S. Ruder, An overview of gradient descent optimization algorithms, 2016, URL http://arxiv.org/abs/1609.04747.

[10] O.M. Group, UML profile for MARTE: Modeling and analysis of RealTime embedded systems, 2019, OMG Document, v1.2 formal/19-04-01. URL https://www.omg.org/spec/MARTE/1.2.

[11] M.G. Harbour, J.J. Gutiérrez, J.C. Palencia, J.M. Drake, MAST: Modeling and analysis suite for real time applications, in: Proceedings 13th Euromicro Conference on Real-Time Systems, IEEE Comput. Soc, 2001, pp. 125–134, http://dx.doi.org/10.1109/EMRTS.2001.934015.

[12] M.G. Harbour, J.J. Gutiérrez, J.M. Drake, P.L. Martínez, J.C. Palencia, Modeling distributed real-time systems with MAST 2, J. Syst. Archit. 59 (2013) 331–340, http://dx.doi.org/10.1016/j.sysarc.2012.02.001.

[13] F. Eisenbrand, T. Rothvoß, Static-priority real-time scheduling: Response time computation is NP-hard, in: 2008 Real-Time Systems Symposium, IEEE, 2008, pp. 397–406, http://dx.doi.org/10.1109/RTSS.2008.25.

[14] K.W. Tindell, A. Burns, A.J. Wellings, An extendible approach for analyzing fixed priority hard real-time tasks, Real-Time Syst. 6 (1994) 133–151, http://dx.doi.org/10.1007/BF01088593.

[15] J.C. Palencia, J.J. Gutierrez, M.G. Harbour, On the schedulability analysis for distributed hard real-time systems, in: Proceedings Ninth Euromicro Workshop on Real Time Systems, IEEE Comput. Soc, 1997, pp. 136–143, http://dx.doi.org/10.1109/EMWRTS.1997.613774.

[16] J.C. Palencia, M.G. Harbour, Schedulability analysis for tasks with static and dynamic offsets, in: Proceedings 19th IEEE Real-Time Systems Symposium, Cat. No.98CB36279, IEEE Comput. Soc, 1998, pp. 26–37, http://dx.doi.org/10.1109/REAL.1998.739728.

[17] J. Mäki-Turja, M. Nolin, Efficient implementation of tight response-times for tasks with offsets, Real-Time Syst. 40 (2008) 77–116, http://dx.doi.org/10.1007/s11241-008-9050-9.

[18] MAST home page. URL https://mast.unican.es/.

[19] A. Amurrio, J.J. Gutiérrez, M. Aldea, E. Azketa, Priority assignment in hierarchically scheduled time-partitioned distributed real-time systems with multipath flows, J. Syst. Archit. 122 (2022) 102339, http://dx.doi.org/10.1016/j.sysarc.2021.102339.

[20] J.M. Rivas, J.J. Gutiérrez, J.L. Medina, M.G. Harbour, Comparison of memory access strategies in multi-core platforms using mast, in: International Workshop on Analysis Tools and Methodologies for Embedded and Real-Time Systems, WATERS, 2017.

[21] S. Altmeyer, É. André, S. Dal Zilio, L. Fejoz, M.G. Harbour, S. Graf, J.J. Gutiérrez, R. Henia, D. Le Botlan, G. Lipari, J. Medina, N. Navet, S. Quinton, J.M. Rivas, Y. Sun, From FMTV to WATERS: Lessons learned from the first verification challenge at ECRTS, in: A.V. Papadopoulos (Ed.), 35th Euromicro Conference on Real-Time Systems, ECRTS 2023, in: Leibniz International Proceedings in Informatics (LIPIcs), vol. 262, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2023, pp. 19:1–19:18, http://dx.doi.org/10.4230/LIPIcs.ECRTS.2023.19.
[22] J.C. Palencia, M.G. Harbour, J.J. Gutiérrez, J.M. Rivas, Response-time analysis in hierarchically-scheduled time-partitioned distributed systems, IEEE Trans. Parallel Distrib. Syst. 28 (7) (2017) 2017–2030, http://dx.doi.org/10.1109/TPDS.2016.2642960.
[23] J.W.S. Liu, Real-Time Systems, Prentice Hall, Upper Saddle River, NJ, 2000.
[24] J. Oh, C. Wu, Genetic-algorithm-based real-time task scheduling with multiple goals, J. Syst. Softw. 71 (2004) 245–258, http://dx.doi.org/10.1016/S0164-1212(02)00147-4.
[25] M. Yoo, Real-time task scheduling by multiobjective genetic algorithm, J. Syst. Softw. 82 (2009) 619–628, http://dx.doi.org/10.1016/j.jss.2008.08.039.
[26] R. Ayari, I. Hafnaoui, A. Aguiar, P. Gilbert, M. Galibois, J.-P. Rousseau, G. Beltrame, G. Nicolescu, Multi-objective mapping of full-mission simulators on heterogeneous distributed multi-processor systems, J. Def. Model. Simul.: Appl., Methodol., Technol. 15 (2018) 449–460, http://dx.doi.org/10.1177/1548512916657907.
[27] J. Lee, S.Y. Shin, S. Nejati, L.C. Briand, Optimal priority assignment for real-time systems: a coevolution-based approach, Empir. Softw. Eng. 27 (6) (2022) http://dx.doi.org/10.1007/s10664-022-10170-1.
[28] Gurobi Optimization. URL https://www.gurobi.com/.
[29] P. Pazzaglia, A. Biondi, M. Di Natale, Simple and general methods for fixed-priority schedulability in optimization problems, in: 2019 Design, Automation & Test in Europe Conference & Exhibition, DATE, IEEE, 2019, pp. 1543–1548, http://dx.doi.org/10.23919/DATE.2019.8715017.
[30] Y. Zhao, H. Zeng, The virtual deadline based optimization algorithm for priority assignment in fixed-priority scheduling, in: 2017 IEEE Real-Time Systems Symposium, RTSS, 2017, pp. 116–127, http://dx.doi.org/10.1109/RTSS.2017.00018.
[31] A. Neelakantan, L. Vilnis, Q.V. Le, I. Sutskever, L. Kaiser, K. Kurach, J. Martens, Adding gradient noise improves learning for very deep networks, 2015, arXiv preprint arXiv:1511.06807.
[32] D.P. Kingma, J. Ba, Adam: A method for stochastic optimization, in: 3rd International Conference for Learning Representations, San Diego, 2015.
[33] BLAS (Basic Linear Algebra Subprograms). URL https://netlib.org/blas/.
[34] OpenBLAS. URL https://www.openblas.net/.
[35] NumPy. URL https://numpy.org/.
[36] GDPA Repository. URL https://github.com/rivasjm/gdpa.
[37] E. Bini, G.C. Buttazzo, Measuring the performance of schedulability tests, Real-Time Syst. 30 (2005) 129–154, http://dx.doi.org/10.1007/s11241-005-0507-9.
[38] S. Anssi, S. Tucci-Piergiovanni, S. Kuntz, S. Gérard, F. Terrier, Enabling scheduling analysis for AUTOSAR systems, in: 2011 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing, 2011, pp. 152–159, http://dx.doi.org/10.1109/ISORC.2011.28.
[39] C.D. Locke, D.R. Vogel, L. Lucas, J.B. Goodenough, Generic avionics software specification, 1990, Draft Specification for Naval Weapons Center, China Lake, CA. IBM Systems Integration Division, Owego, NY.

Juan M. Rivas is an Assistant Professor in the Software Engineering and Real-Time Group at the University of Cantabria (Spain). He received his B.Sc. degree in Telecommunications Engineering and M.Sc. in Computer Science from the University of Cantabria in 2008 and 2009 respectively. He obtained his Ph.D. degree in Computer Science from the same institution in 2015. He has been involved in several national and European research projects, including industrial collaborations, focusing on topics such as the optimization of distributed hard-real-time systems, modeling, and scheduling in novel platforms such as GPUs.

J. Javier Gutiérrez received his B.S. and Ph.D. Degrees from the University of Cantabria (Spain) in 1989 and 1995 respectively. He is a Professor in the Software Engineering and Real-Time Group at the University of Cantabria, which he joined in the early 90s. His research activity deals with the scheduling, analysis and optimization of embedded real-time distributed systems (including communication networks and distribution middleware). He has been involved in several research projects building real-time controllers for robots, evaluating Ada for real-time applications, developing middleware for real-time distributed systems, and proposing models along with the analysis and optimization techniques for distributed real-time applications.

Ana Guasque was born in Valencia, Spain. She received a B.S. degree in industrial engineering from the Universitat Politècnica de València (UPV) in 2013; and an M.S. degree in automation and industrial computing from the UPV in 2015. She received a Ph.D. degree in industrial engineering from the UPV in 2019. She is currently working as a researcher in the Universitat Politècnica de València. Her main research interests include real-time operating systems, scheduling, and optimization algorithms and real-time control.

Patricia Balbastre is an associate professor of computer engineering at the Universitat Politècnica de València (UPV). She graduated in electronic engineering at the UPV in 1998 and obtained the Ph.D. degree in computer science in 2002. Her main research interests include real-time operating systems, dynamic scheduling algorithms and real-time control.