End-to-End Latency Optimization of Thread Chains Under the DDS Publish/Subscribe Middleware

Gerlando Sciangula[*◇], Daniel Casini[*], Alessandro Biondi[*], Claudio Scordino[◇]
[*]Scuola Superiore Sant'Anna, Pisa, Italy
[◇]Huawei Research Center, Pisa, Italy

***Abstract*—Modern autonomous systems integrate diverse software solutions to manage tightly communicating functionalities. These applications commonly communicate using frameworks implementing the publish/subscribe paradigm, such as the Data Distribution Service (DDS). However, these frameworks are realized with a multi-threaded software architecture and implement internal policies for message dispatching, posing additional challenges for guaranteeing timing constraints.**

This work addresses the problem of optimizing a DDS-based interconnected real-time systems, proposing analysis-driven algorithms to set a vast range of parameters, ranging from classical thread priorities to other DDS-specific configurations. We evaluate our approaches on the Autoware Reference System, a realistic testbed from the Autoware autonomous driving framework.

I. Introduction

Nowadays, autonomous systems are characterized by interconnected and possibly distributed software components, which need to seamlessly integrate and tightly cooperate as if they were executed as a single one. This trend shapes differently in various applications. A case in point is automotive, in which *advanced driver assistance systems* (ADAS) functionalities, ranging from lane-keeping adaptive cruise control to anti-lock braking systems, are distributed across a number of *electronic control units* (ECUs) and interconnected through in-vehicle networks. All these applications are typically implemented as chains of computations to be executed within timing constraints. Looking even further, the software components of future autonomous cars will be distributed in the Edge-to-Cloud continuum, where the vehicle can cooperate with the surrounding infrastructure and leverage the massive computational capacity offered by the Cloud to provide advanced functionality. These developments are centralizing the role of data distribution services, calling for efficient and time-predictable solutions.

Modern automotive applications are more and more on top of frameworks such as AUTOSAR Adaptive [1], ROS 2 [2], and Autoware [3]. All of them leverage the *Data Distribution Service* (DDS) communication middleware [4], which established as the most used solution for spreading data within distributed components [5]. DDS frameworks implement internal policies for message dispatching and are commonly realized with a multi-threaded software architecture. These characteristics need to be taken into account when aiming at guaranteeing timing constraints for DDS-enabled applications, especially when facing with data dependencies between threads, which form the so-called *thread chains*. Accurate system configurations, e.g., in terms of priorities and thread allocation, can significantly improve the timing performance of a system; as such, systematic approaches to set timing-relevant parameters of DDS-enabled applications are a must-have for complex systems.

While previous work [6] proposed a DDS-specific real-time analysis to assess whether a given configuration can allow meeting a set of timing constraints upfront, i.e., without requiring to deploy the application on the target platform, designers are still left with the question of how to optimize the vast set of parameters of DDS-enabled applications to meet timing requirements at best.

Contribution. This paper addresses this open problem by proposing a collection of analysis-driven optimization algorithms leveraging the results from [6]. The FastDDS [7] implementation of the DDS standard is considered. A simulated annealing optimizer is also proposed as a baseline for comparison. The proposed algorithms are evaluated on a case study based on the Autoware Reference System [8] — a realistic testbed from the Autoware autonomous driving framework.

II. Essential Background

DDS enables advanced communication and data-sharing capabilities for distributed and heterogeneous applications. It is a middleware communication protocol standardized by the Object Management Group that builds upon a publish-subscribe model, which decouples data producers (*publishers*) from consumers (*subscribers*). Publishers, subscribers, and topics are the main entities of a DDS domain. A publisher can send information over multiple topics; similarly, a subscriber can receive data from different topics. One of the most popular and efficient C++ implementations of DDS is FastDDS by eProsima [7], which is currently the default DDS implementation in ROS 2. FastDDS provides two modalities for message dispatching: *synchronous* and *asynchronous*. Similarly to [6], we consider the asynchronous mode, which provides more flexibility for real-time messages by means of *flow-controller* threads, which are in charge of the message delivery process. A flow-controller thread is responsible for extracting data from a queue of pending messages (sent by publishers) according to different policies. A *listener* thread is associated with each subscriber and manages the reception of user data. Subscribers can also publish messages, thus originating chains of threads.

III. System Model and Problem Definition

Platform and Thread Model. We consider a distributed system consisting of a set $\mathcal{N}$ of interconnected machines. Each machine $\nu_y \in \mathcal{N}$ comprises a set of homogeneous cores $\mathcal{C}_y$, in which each individual core is denoted with $c_k \in \mathcal{C}_y$. Symbol

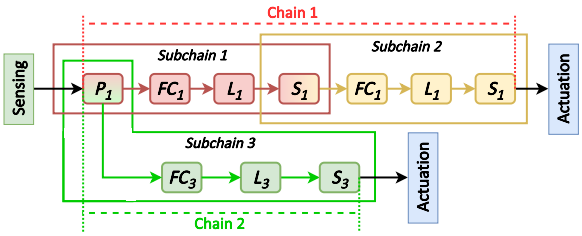

Fig. 1: Two chains activated by the same publisher thread.

$\mathcal{C}$ denotes the set of all cores of all machines. Following prior work [6], we describe DDS-enabled real-time systems as a set of application-level threads (i.e., publishers and subscribers) and middleware-level threads (i.e., flow-controllers and listeners). The set Γ denotes all the threads in the system, irrespective of the type. Threads are grouped into two sets: **(i)** *middleware-level* threads and **(ii)** *application-level* threads. The first set, denoted with Γ_{mw} includes flow-controller and listener threads; the second set, denoted with $\Gamma_{\text{app}} \triangleq \Gamma \setminus \Gamma_{\text{mw}}$, includes all publishers and subscribers. The sets of middleware-level and application-level threads allocated on core $c_k \in \mathcal{C}$ are referred to as Γ^k_{mw} and Γ^k_{app}, respectively. We consider a static DDS system without threads joining or leaving the system at runtime.

Each application-level thread $\tau_i \in \Gamma_{\text{app}}$ is characterized by a *worst-case execution time* (WCET) e_i. Each thread $\tau_i \in \Gamma$ releases a sequence of instances (called jobs): its arrival curve $\eta_i(\Delta)$ bounds the number of release events in any interval of length Δ. Threads are scheduled using a partitioned fixed-priority scheduler, where each thread $\tau_i \in \Gamma$ is statically allocated to a core $c_k \in \mathcal{C}$ and assigned a unique fixed priority Π_i. As a DDS-specific allocation constraint, each publisher (resp. subscriber) thread and the referred flow-controller (resp., listener) threads must be allocated to the same machine.

DDS-based Communication Model. The topics define the logical communication channels between publishers and subscribers. The set of all topics is referred to as Θ. Each topic $\theta_j \in \Theta$ is characterized by a unique priority π_j. Publishers publish messages on the topics: an arbitrary z-th message is denoted by m_z. Each instance of a message m_z, published by a thread $\tau_i \in \Gamma_{\text{app}}$ over the topic θ_j, inherits the priority assigned to the topic θ_j. Similarly to threads, messages are also characterized by a *derived* arrival curve, which is inherited by the producer thread using the arrival-curve propagation process [9] recalled in Section III-A. An application-level thread can be associated with multiple flow-controller threads if it publishes to multiple topics. A subscriber thread is always linked to one listener thread, handling messages from different topics. A message is related to a single flow-controller and a single listener thread. Listener threads manage messages in first-in-first-out (FIFO) order. When a message m_z is managed by a thread $\tau_i \in \Gamma^k_{\text{mw}}$, we say that $m_z \in \tau_i$.

Each flow-controller thread manages a queue of messages to be dispatched according to either the `HIGH_PRIORITY` FastDDS's policy (`HP` for short), i.e., fixed-priority scheduling according to the message priority [6], or `FIFO` (`F` for short).

The parameters $\delta^{\text{f}}(m_z)$ and $\delta^{\text{l}}(m_z)$ denote the worst-case time required to process a message m_z in flow-controller and listener threads, respectively, without the interference of any other message or thread. When distinguishing between flow-controller and listener threads is irrelevant, we simply use $\delta(m_z)$. The symbol $\delta^{\nu_i,\nu_j}_{\text{net}}(m_z)$ denotes the network propagation delay of a message m_z that traverses from a machine ν_i to a machine, with $\nu_j \neq \nu_i$.

Thread chains. We denote by $\mathcal{G}$ the set of all thread chains. A chain $\gamma_i \in \mathcal{G}$ is a sequence of communicating threads. The first thread of a chain is always a publisher; the last is always a subscriber. Within the chain, each publisher-subscriber pair is separated, in order, by the flow-controller thread (FC) associated with the publisher (P) and the listener (L) associated with the subscriber (S), as shown in Fig. 1. The set of all chains can be represented as a directed acyclic graph (DAG) composed of threads (vertexes) and communications (edges). The DAG can have multiple sources and multiple sinks.

The *end-to-end* (E2E) latency of a thread chain γ_i is defined as the longest time span elapsed between the release of the first thread of the chain to when the sink subscriber of γ_i completes. Symbol L_i denotes an upper bound on the end-to-end latency. The end-to-end timing requirement of a chain γ_i is specified as a deadline D_i and is satisfied if $L_i \leq D_i$. In this case, we say that the chain is *schedulable*.

Each chain γ_i comprises one or more subchains. Each subchain is defined as a tuple of four threads: a publisher, a flow-controller, a listener, and a subscriber. Two consecutive subchains of a chain are linked by the subscriber of the first one acting as publisher for the second one, as shown in Fig. 1.

Definitions. The *density* $\sigma(\tau_i, \gamma_h)$ of an application-level thread $\tau_i \in \Gamma_{\text{app}}$ that is part of a chain $\gamma_i \in \mathcal{G}$ is defined as the ratio between the WCET of the thread and the deadline of the chain, i.e., $\sigma(\tau_i, \gamma_h) \triangleq e_i/D_h$. Similarly, the density of a message m_z processed in a flow-controller thread being part of a chain γ_h is defined as the ratio between the WCET of the message and the deadline of the chain, i.e., $\sigma(m_z, D_h) \triangleq \delta(m_z)/D_h$. Finally, we define the *load* of an application-level thread τ_i and a message m_z as $\text{load}(\tau_i, \Delta) \triangleq \eta_i(\Delta) \cdot e_i/\Delta$ and $\text{load}(m_z, \Delta) \triangleq \eta_i(\Delta) \cdot \delta(m_z)/\Delta$, respectively.

A. Data-Delivery Latency Analysis under FastDDS

We briefly recall the most important highlights of the analysis in [6], which bounds the latency of a message between an arbitrary pair of publishers and a subscriber as the sum of the individual *worst-case response times* (WCRT) experienced by the flow-controller and listener threads, also adding the network delay. The WCRTs are subject to different sources of interference: **(i)** *thread-level interference*, which depends on higher-priority application threads, **(ii)** *inter-thread message interference*, which is due to the processing of messages handled in other middleware threads, **(iii)** *intra-thread interference* due to the processing of messages in the same middleware thread, and **(iv)** *self-interference* caused by previously-released instances of the same message or thread. Following Compositional Performance Analysis (CPA) [9], the approach in [6] is

extended by adding the WCRT of publishers and subscribers to quantify the individual latency of subchains, which, in turn, can be summed up to bound the overall end-to-end latency of a thread chain [9].

Arrival curves are externally provided (see Section III) only for source threads; instead, the analysis requires arrival curves for all threads and messages. Arrival curves are derived following the CPA-based propagation process, which uses the activation delays of predecessor threads in a chain as release jitter for the follower threads. This jitter consists of the network propagation delay and the WCRT bound of predecessor threads. This creates a cyclic dependency: to bound WCRTs, arrival curves for all threads and messages are required, but they also require the WCRT bounds. The cycle is broken [6], [9] by initially setting the jitter to zero and computing WCRT bounds iteratively until convergence is reached.

B. Problem Statement

We consider the analysis-based optimization of DDS-based real-time applications with the following objective function:

$$\textbf{minimize} \ \max_{\gamma_i \in \mathcal{G}} L_i. \tag{1}$$

The system configuration consists of the values assigned to all tunable parameters decided by the proposed optimization algorithms. We consider the following tunable parameters:

- **T1** Threads-to-machines and threads-to-core mapping.
- **T2** Number of flow-controllers threads.
- **T3** Messages-to-flow-controllers partitioning.
- **T4** Scheduling policy of flow controllers.
- **T5** Priority of threads.
- **T6** Priority of topics (inherited by the messages).

IV. Optimization Algorithms

This section presents the three approaches we propose to optimize DDS-enabled real-time systems. First, we present an optimization meta-algorithm that can be employed with different combinations of heuristics, which are agnostic on any DDS-specific real-time analysis aspect. Second, we present a specialized algorithm that takes into account the peculiarities of the FastDDS real-time analysis [6]. Finally, we present a simulated annealing optimization algorithm. It is worth noting that this work faces with a complex non-convex optimization problem originating from convoluted real-time analysis equations and algorithms [6]. Although the problem could be linearized to be solved with Mixed-Integer Linear Programming (MILP), since the analysis includes several circular dependencies (see Section III-A), this would require introducing a series of pessimistic overestimation of response-time bounds and a high number of auxiliary variables that would affect both the quality of the solution and the runtime required to find it. For these reasons, we did not consider the use of MILP.

A. Optimize Latency Meta-Algorithm

The Optimize Latency Meta-Algorithm (OL-Alg for short) considers the aforementioned optimization objectives. For simplicity, the number of flow-controller threads is considered to be a constant in input to the problem. OL-Alg is presented in Algorithm 1 and exposes the following configurations:

Algorithm 1 Pseudo-code for OL-Alg

```
1: AP_m: Allocation policy of messages ← (WF, BF, FF)
2: AP_t: Allocation policy of threads ← (WF, BF, FF)
3: SP_m: Sorting policy of messages ← (DW, IW, DL, IL, DD, ID)
4: SP_t: Sorting policy of threads ← (DW, IW, DL, IL, DD, ID)
5: SSP: Sending scheduling policy ← (HP, F)
6: PAP_θ: Priority policy of topics ← (DMW, IMW, DML, IML)
7: function OL-ALG(AP_m, AP_t, PAP_θ, SP_m, SP_t, SPP)
8:     SetSendingPolicy(SSP)
9:     SetTopicsPriority(PAP_θ)
10:    AllocateFCsToMachines()
11:    if AllocateAppThreads(AP_t, SP_t) then
12:        if AllocateMessagesToFCs(AP_m, SP_m) then
13:            if AllocateListeners(AP_t, SP_t) then
14:                if ThreadPrioAssign( SP_t) then
15:                    PerformAnalysis()
16:                else Cannot find a schedulable threads priorities
17:            else Cannot allocate Listener threads
18:        else Cannot allocate Messages
19:    else Cannot allocate Application threads
```

1) the allocation policy used for messages and threads, managed by the set of *allocation policies* AP_m and AP_t of the algorithm (lines 1 and 2), for messages (topics) and threads, respectively.
2) the order in which the algorithm considers the messages and threads, managed by the set of *sorting policies* SP_m and SP_t of the algorithm (lines 3 and 4), for messages (topics) and threads, respectively.
3) the scheduling policy used in flow-controller threads; it can be assigned either to `HP` (fixed-priority scheduling) or `FIFO` (line 5).
4) the priority-assignment policies for the topics, managed by the set PAP_θ of priority-assignment policies for topics of the algorithm (line 6).

For 1), we consider three popular partitioning heuristics: *first-fit*, *best-fit*, and *worst-fit*. When using *first-fit*, each thread or message is placed in the *first* core $c_k \in \mathcal{C}$ that can host it according to a test of the overall load of the core, i.e.,

$$U_k(\Delta) \triangleq \sum_{\tau_i \in \Gamma^k_{\text{app}}} \text{load}(\tau_i, \Delta) + \sum_{\tau_j \in \Gamma^k_{\text{mw}}} \sum_{m_z \in \tau_j} \text{load}(m_z, \Delta) \leq 1. \tag{2}$$

In the above equation, Δ is a parameter of the test, typically set to a large value [10]. Best-fit and worst-fit perform the same check, but after sorting the cores $c_k \in \mathcal{C}$ based on the smallest and largest value of $1 - U_k(\Delta)$, respectively.

For 2), we consider six ordering for threads and messages, namely *increasing/decreasing WCET* (IW/DW), *increasing/decreasing density* (ID/DD), and *increasing/decreasing load* (IL/DL). In the case of messages, the IW/DW policies consider the parameter $\delta^{\text{f}}(m_z)$ as the message WCET. For 4), we similarly consider *increasing/decreasing message WCET* (IMW/DMW) and *increasing/decreasing message load* (IML/DML).

The algorithm works as follows. First, function `SetSendingPolicy()` sets the sending scheduling policy of all flow-controllers of the system equal to the sending policy

passed to the function (chosen from SSP: `HP` or `F`). Then, `SetTopicsPriority()` sets the priority of topics according to the selected priority assignment policies, chosen from the set PAP_θ (IMW/DMW/IML/DML, line 6).

The various steps of the allocation process are then performed: first, flow-controller threads are allocated to machines (line 10) in a round-robin manner; application-level threads are then allocated to cores (using a pair of sorting policies/partitioning heuristics in sets SP_t/AP_t, line 11); messages are mapped to flow-controller threads using policies in the sets SP_m/AP_m and considering that messages need to be allocated on a flow-controller that is within the same machine of the publishers that publish them (line 12); finally, listeners are allocated, following a sorting policy and a partitioning heuristics in sets SP_t/AP_t (line 13).

Priorities for threads are managed similarly to Audsley's Optimal Priority Assignment (OPA) [11]: threads are processed according to the first one assigned to the lowest possible priority, checking whether the corresponding chains are schedulable. If not, the priority is raised. This is implemented in function `ThreadPrioAssign()`. Finally, since the load-based test of Eq. (2) does not guarantee schedulability per se, the function `PerformAnalysis()` performs the actual analysis and retrieves the end-to-end latency bound of each chain, if all chains are schedulable (line 15).

B. Analysis-Aided DDS Meta-Algorithm

This second optimization approach, called Analysis-Aided DDS Meta-Algorithm (AA-Alg for short), extends the previous one by leveraging DDS-specific analysis considerations to assign priorities. In essence, the AA-Alg replaces the threads and topics priority assignments with the following four heuristics. The first one targets the priority assignment of threads *within a chain* (*intra-chain priority assignment*). The second and third ones are two alternatives for the priority assignment of chains (*inter-chain priority assignment*). The last heuristic targets the priority assignment of topics, privileging latency-critical communications.

Intra-Chain Priority Assignment. As discussed in Section III-A, the analysis in [6] leverages arrival-curve propagation to derive the arrival curves of non-source threads, accounting for the delays introduced by predecessor threads in the chain as release jitter. At the same time, release jitter is known to significantly penalize schedulability. Therefore, we make the following key observation: *the less the threads occurring earlier in a chain are interfered with, the less release jitter is propagated to the follower threads, hence tending to reduce their worst-case interference.*

Following this observation, we start assigning the highest priority in the chain to the source thread, and we proceed sequentially in the order they appear in the chain by assigning decreasing priorities. The sink thread is assigned the lowest priority of the chain.

Load-Based Inter-Chain Priority Assignment. This second heuristic uses the following observation: *threads with smaller loads have a lower impact on the WCRT of higher-load threads*. Following this idea and considering threads from different chains, we assign priorities to threads inverse-proportionally to the thread load (i.e., lower load, higher priority) while maintaining the relative-priority order chosen according to the intra-chain rule. This procedure aims at limiting the end-to-end latency of all chains since low-load threads have less impact on the response time than high-load threads and, in general, on the jitter propagated by the chains. Nevertheless, this heuristic does not consider the deadline of each chain; therefore, it could lead to an unschedulable solution in some cases.

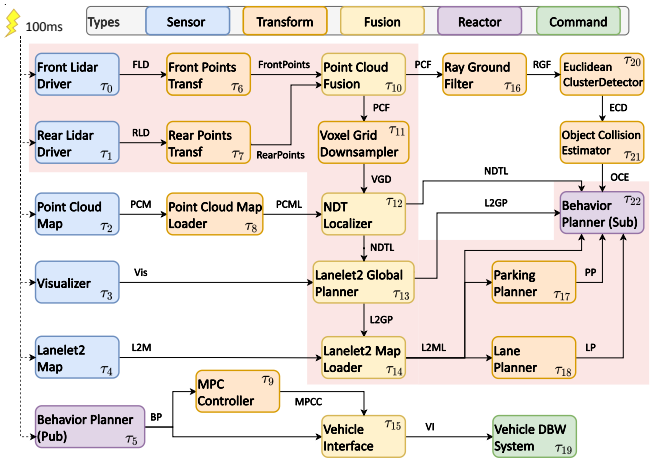

Fig. 2: Overview of the Autoware Reference System. Incoming edges represent the topics subscribed by a thread, while outgoing edges represent the topic on which it publishes. The label attached to each edge represents the topic's name.

Deadline-Based Inter-Chain Priority Assignment. The deadline is instead considered by the *deadline-based inter-chain priority assignment* heuristic, which prioritizes the threads that are part of a chain with tight deadlines. Hence, the priority assignment is inversely proportional to the chain deadline. Internally to each chain, the *intra-chain priority assignment* rule is used.

Prioritizing Critical Communications. Finally, the last heuristic prioritizes communications that involve high-priority application-level threads. The priority of a topic is consequently chosen according to the highest priority of any application-level thread using it. Furthermore, high-priority topics are mapped to higher-priority flow-controllers. The prioritization of critical communications avoids priority inversion due to the processing of low-priority messages by higher-priority flow-controllers.

C. Simulated Annealing Optimizer

To assess the quality of the solutions found by the proposed heuristics, we also developed a *Simulated Annealing (SA)* optimizer as a baseline for comparison. The SA algorithm is implemented without any constraints, allowing the optimizer to evaluate and consider even unfeasible or unschedulable solutions. This is necessary to ensure the exploration of all solutions within the whole design space to avoid sticking to local optima. Restricting the search to only feasible and schedulable solutions may let the algorithm focus on a sub-region of the space, potentially causing the algorithm to miss the global optimum (which, instead, needs to be a feasible solution). The probability of selecting such solutions is minimized by

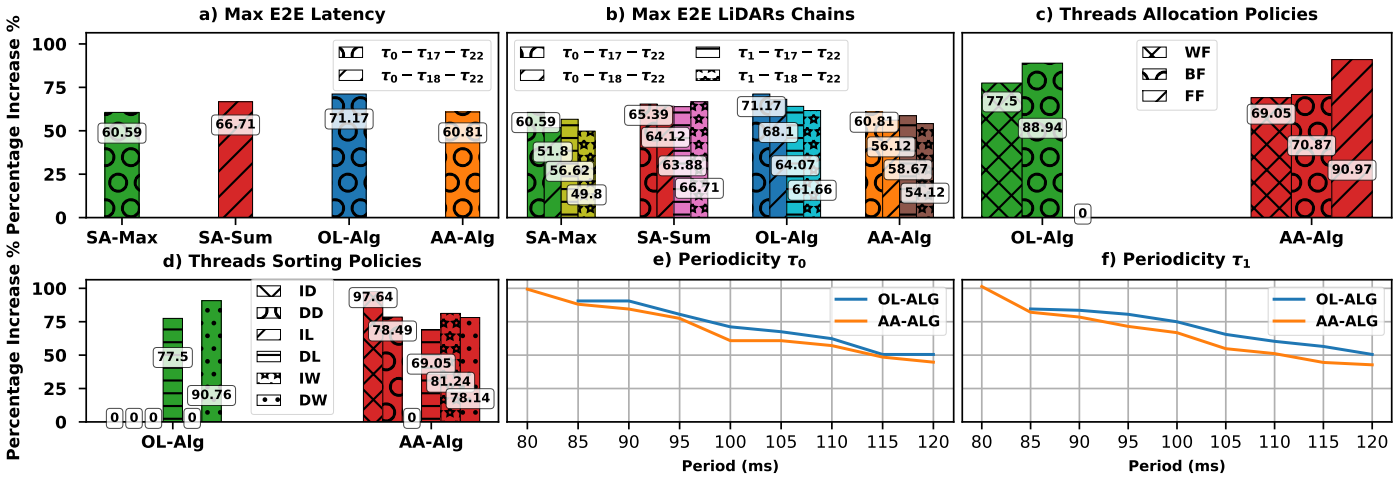

Fig. 3: Percentage-increase to a lower-bound on the end-to-end latency of some representative chains under different algorithms. (a) and (b) try configurations for all optimization parameters. (c) evaluates thread allocation policies for SP_m = DD, SP_t = DL, AP_m = WF, PAP_θ = DML, SSP = HP. (d) evaluates thread sorting policies for SP_m = DD, AP_m = WF, AP_t = WF, PAP_θ = DML, SSP = HP. (e) and (f) show the effect when the periodicity of the source threads is varied while fixing other parameters at SP_m = DD, SP_t = DL, AP_m = WF, AP_t = WF, PAP_θ = DML, SSP = HP.

introducing a penalty factor to inflate the cost value of the system configuration. Nevertheless, the final solution provided in output by the SA is always schedulable (if any is found).

The schedulability of a solution is then checked by using the analysis reviewed in Section III-A. For the design space exploration, we defined transition operators for randomly choosing thread and message assignments and priorities.

V. Experimental Evaluation

Autoware Reference System. We evaluated our approaches on a case study based on the *Autoware Reference System (ARS)* [8], which includes a computation graph from Autoware, a ROS 2-based autonomous driving framework [3]. The application is shown in Fig. 2. We consider a system configuration with one machine ν_i and ten cores. A flow controller is added to each core. The *ARS* comprises five types of threads: *sensor*, *transform*, *fusion*, *reactor*, and *command* threads, with WCETs equal to $0.1ms$, $50ms$, $25ms$, $1ms$, and $0.2ms$, respectively. All source threads are triggered with a period of $100ms$. All publishers publish messages of the same size ($4Kb$). Parameters $\delta^{\mathrm{f}}(m_z)$ and $\delta^{\mathrm{l}}(m_z)$ are set as in [6]. The ARS defines the end-to-end latency of the LiDAR chains, starting from τ_0 and τ_1 (LiDAR threads) and ending with τ_{22} as one of the leading KPIs to evaluate the performance of the system. Chain deadlines are not included in ARS and are obtained by leveraging a lower bound on the end-to-end latency of each chain γ_i in an ideal configuration in which each thread is allocated exclusively to a core, and each message is in a dedicated flow-controller. This eliminates any interference (see Section III-A), except for the self-interference. The deadline of each chain is assigned proportionally to each chain latency lower bound.

Results. Fig. 3(a) illustrates the results, in which OL-Alg, AA-Alg, and SA are compared. The graph shows the percentage increment (y-axis) of the end-to-end latency w.r.t. the lower-bound delay found for a specific chain. For the SA algorithm, we considered two different variants: one uses the objective function in Eq. (1) (SA-Max) and the other minimizes the sum of the latencies $\sum_{\gamma_i \in \mathcal{G}} L_i$ (SA-Sum).

According to SA-Max, OL-Alg, and AA-Alg, the chain τ_0-τ_{17}-τ_{22} results to be the chain with the maximum end-to-end latency within the system. SA-Sum found a different chain, i.e., τ_0-τ_{18}-τ_{22}. From Fig. 3(a), we can observe that the best solution was found by SA-Max, with a maximum end-to-end latency 60.59% higher than the lower-bound. Notably, AA-Alg is almost as good as (in terms of maximum latency) SA-Max (60.81% of increment). SA-Sum tried to balance the latency of all the chains but at the expense of the maximum latency, which has a higher increment of 66.71%. The best configuration found by OL-Alg leads to an increment of 71.17%. Such a clear difference between OL-Alg and AA-Alg lies in the fact that the second algorithm is driven by analysis-related observations, thus trying to prioritize latency-critical communications and minimize the jitter propagated in the chains. Fig. 3(b) shows the percentage increment of the end-to-end latency of the chains highlighted in Fig. 2, i.e., τ_0-τ_{17}-τ_{22}, τ_0-τ_{18}-τ_{22}, τ_1-τ_{17}-τ_{22}, and τ_1-τ_{18}-τ_{22}. Again, in terms of latencies, AA-Alg behaves similarly to SA-Max.

Fig. 3(c) and (d) report the performance of different partitioning heuristics and sorting policies for the threads applied to OL-Alg and AA-Alg, while keeping other parameters fixed (reported in the caption of Fig. 3). In Fig. 3(c), for both algorithms, the worst-fit policy results to be the best. Under the first-fit policy, the OL-Alg could not find a schedulable configuration (represented by a 0 value in the graph). We observed the same trend by also varying the other parameters. There are no observed cases in which OL-Alg can find a schedulable configuration that AA-Alg cannot find.

Fig. 3(d) shows different behaviors for the algorithms. Under several sorting policies (ID, DD, IL, IW), OL-Alg cannot find a schedulable solution, while AA-Alg was not able to develop a schedulable configuration only under the IL sorting policy. This is mainly due to the robustness of AA-Alg in adapting

to any thread allocation or order of allocation since they do not affect the *intra-chain* and *inter-chain* priority assignments. Differently, the priority assignment of OL-Alg is completely dependent on the sorting policy of the threads. Fig. 3(e) and (f) show two scenarios in which the periodicity of τ_0 and τ_1 is varied in $[80, 120]$ ms. The latency of chains τ_0-τ_{17}-τ_{22} and τ_1-τ_{17}-τ_{22} is reported on the y-axis. As in the previous cases, AA-Alg always performs better than OL-Alg, providing smaller increases. When the frequency is higher (i.e., 80 ms), OL-Alg cannot find a schedulable solution.

Runtimes. In Table I we reported the number of tested configurations and the total running times, showing an important take-away message: by leveraging analysis-related considerations instead of blindly trying combinations of heuristics, AA-Alg provides solutions that are almost as good as the simulated annealing, but much faster, reducing the runtime required from 14 hours to 7 minutes for the considered scenarios.

TABLE I: Number of tested configurations and running times.

Algorithm	Configurations	Runtime (minutes)
SA-Max	7328000	846
SA-Sum	7328000	828
OL-Alg	2592	66
AA-Alg	648	7

VI. Related work

The literature related to this paper can be broadly classified into two categories: optimization of real-time systems and real-time analysis of middleware frameworks. Into the first category fall numerous works, ranging from the optimization of hard real-time time-triggered distributed systems [12] to the partitioning and priority assignment of real-time applications with hardware acceleration [13]. Different works leverage different optimization strategies for different problems and optimization objectives, such as simulated annealing [14], mixed-integer linear programming formulations [15], and heuristics [16]. However, none of them targets DDS-based systems.

The second category of related papers, targeting the analysis of real-time properties in middleware frameworks, is less populated but has received considerable attention in recent years. For example, some of the studied frameworks are OpenMP [17], TensorFlow [18], MQTT [19] and ROS 2 [20], [21]. Most related to this paper is the work in [6] targeting a real-time analysis under FastDDS, which, however, does not propose optimization algorithms for DDS-enabled systems.

To the best of our knowledge, there have been no prior attempts at introducing analysis-driven optimization algorithms that can effectively optimize the temporal aspects of thread chains in DDS-based systems.

VII. Conclusion

In this paper, we targeted the optimization of a DDS-enabled real-time system. We tested our approaches on the *Autoware Reference System*, showing that the solutions found by the proposed algorithms are much faster to obtain and can compete with those found by a simulated annealing optimizer, which is able to find quality suboptimal solutions at the cost of higher running times. Furthermore, we demonstrated the benefits of analysis-aided optimization (AA-Alg) by leveraging analytical observations in assigning priorities.

VIII. Acknowledgements

This work has been supported by the project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union – NextGenerationEU and the European Union's Horizon Europe Framework Programme project NANCY under the grant agreement No. 101096456.

References

[1] AUTOSAR, "Specification of Communication Management," 2022.
[2] "Robot operating system (ROS)." https://www.ros.org/.
[3] S. Kato, S. Tokunaga, Y. Maruyama, S. Maeda, M. Hirabayashi, Y. Kitsukawa, A. Monrroy, T. Ando, Y. Fujii, and T. Azumi, "Autoware on board: Enabling autonomous vehicles with embedded systems," in *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)*, pp. 287–296, 2018.
[4] OMG, "Data distribution service (DDS) version 1.4.."
[5] C. Scordino, A. G. Mariño, and F. Fons, "Hardware acceleration of data distribution service (dds) for automotive communication and computing," *IEEE Access*, vol. 10, pp. 109626–109651, 2022.
[6] G. Sciangula, D. Casini, A. Biondi, C. Scordino, and M. Di Natale, "Bounding the Data-Delivery Latency of DDS Messages in Real-Time Applications," in *35th ECRTS 2023*, (LIPIcs), 2023.
[7] eProsima, "Fast-DDS," 2023. https://fast-dds.docs.eprosima.com.
[8] https://github.com/ros-realtime/reference-system.
[9] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst, "System level performance analysis - the SymTA/S approach," *IEEE Proceedings - Computers and Digital Techniques*, March 2005.
[10] J. Diemer, P. Axer, and R. Ernst, "Compositional Performance Analysis in Python with pyCPA," in *In Proceedings of WATERS'12*, 2012.
[11] N. C. Audsley, "Optimal priority assignment and feasibility of static priority tasks with arbitrary start times," Tech. Rep. YCS-164, Department of Computer Science, University of York, 1991.
[12] Q. Zhu, H. Zeng, W. Zheng, M. D. Natale, and A. L. Sangiovanni-Vincentelli, "Optimization of task allocation and priority assignment in hard real-time distributed systems," *ACM Trans. Embed. Comput. Syst.*
[13] D. Casini, P. Pazzaglia, A. Biondi, and M. Di Natale, "Optimized partitioning and priority assignment of real-time applications on heterogeneous platforms with hardware acceleration," *Journal of Systems Architecture*, 2022.
[14] X. He, Z. Gu, and Y. Zhu, "Task allocation and optimization of distributed embedded systems with simulated annealing and geometric programming," *The Computer Journal*, vol. 53, no. 7, pp. 1071–1091.
[15] P. Pazzaglia, A. Biondi, and M. Di Natale, "Optimizing the functional deployment on multicore platforms with logical execution time," in *2019 IEEE Real-Time Systems Symposium (RTSS)*, pp. 207–219, 2019.
[16] M. Saksena and Y. Wang, "Scalable real-time system design using preemption thresholds," in *Proceedings 21st IEEE real-time systems symposium*, pp. 25–34, IEEE, 2000.
[17] M. A. Serrano, A. Melani, R. Vargas, A. Marongiu, M. Bertogna, and E. Quinones, "Timing characterization of OpenMP4 tasking model," in *2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, pp. 157–166, IEEE, 2015.
[18] D. Casini, "A theoretical approach to determine the optimal size of a thread pool for real-time systems," in *2022 IEEE Real-Time Systems Symposium (RTSS)*, pp. 66–78, 2022.
[19] E. Shahri, P. Pedreiras, and L. Almeida, "End-to-end response time analysis for rt-mqtt: Trajectory approach versus holistic approach," in *2023 IEEE 19th International Conference on Factory Communication Systems (WFCS)*, pp. 1–8, 2023.
[20] D. Casini, T. Blaß, I. Lütkebohle, and B. B. Brandenburg, "Response-time analysis of ROS 2 processing chains under reservation-based scheduling," in *31st ECRTS 2019*, 2019.
[21] H. Teper, M. Günzel, N. Ueter, G. von der Brüggen, and J. Chen, "End-to-end timing analysis in ROS2," in *2022 IEEE Real-Time Systems Symposium (RTSS)*, 2022.