Latency Measurement for Autonomous Driving Software Using Data Flow Extraction

Conference Paper · June 2023

DOI: 10.1109/IV55152.2023.10186686

4 authors, including:

Tobias Betz
Technische Universität München
17 PUBLICATIONS **110** CITATIONS
SEE PROFILE

Maximilian Schmeller
Technische Universität München
3 PUBLICATIONS **30** CITATIONS
SEE PROFILE

Johannes Betz
Technische Universität München
132 PUBLICATIONS **1,870** CITATIONS
SEE PROFILE

Latency Measurement for Autonomous Driving Software Using Data Flow Extraction

Tobias Betz, Maximilian Schmeller, Andreas Korb, Johannes Betz

***Abstract*—Real-time capability and robust software behavior have emerged as crucial issues since autonomous vehicles must react reliably to various traffic conditions when operating on our streets. The objective of our work is to understand and examine the processing latency of a software stack for autonomous vehicles. In this paper, we propose a framework based on *ros2_tracing* that automatically extracts implicit and explicit data flow from large-scale ROS 2-based autonomous driving software. It can measure the end-to-end latency and the individual components it is composed of. Using a static analysis, the implicit dependencies can be extracted. The method was used to analyze a software stack for autonomous vehicles. Compared to previous work that requires a manual definition of node-internal data dependencies and often does not follow the data flows completely, this paper provides a more feasible and comprehensive toolkit for analyzing real-world ROS 2 systems.**

I. Introduction

Many companies are working on realizing autonomous driving technology and have already demonstrated initial results [1]. This kind of software can typically be divided into the following modules: perception, planning, and control [2]. Ultimately, these modules consist of serial and parallel computation paths, which result in a complex interleaving of the software. During the development process, the runtimes of the module algorithms must be constantly analyzed and measured, and the corresponding resource requirements must be tracked to ensure the ideal utilization of the available computing capacity. Many software stacks are built with the Robot Operating System 2 (ROS 2) middleware [3], which provides a set of libraries, functions, and tools to build software for autonomous systems. Unfortunately, the complexity has increased dramatically over the last few years due to the integration of a wide variety of complex and interconnected software functions.

Understanding the sources of latency and energy consumption is key to improving the overall autonomous system in both hard- and software. Tracing tools are used to analyze the runtime of an entire software stack. Not only the individual runtimes of the respective algorithms but also the resulting end-to-end latency are of importance. Taking autonomous driving software as an example, the end-to-end latency is the time between new perception data (e.g., LiDAR) being available and the control inputs for steering, acceleration, or brake pedal being produced. The end-to-end latency accumulates from a multitude of algorithm runtimes as well as the time needed to send data between nodes.

T. Betz is with the Institute of Automotive Technology, TUM School of Engineering and Design, Technical University Munich, 85748 Garching, Germany (Corresponding author; email: tobias94.betz@tum.de).
M. Schmeller and A. Korb are with the TUM School of Computation, Information and Technology, Technical University Munich, 85748 Garching, Germany.
J. Betz is with the Professorship of Autonomous Vehicle Systems, TUM School of Engineering and Design, Technical University Munich, 85748 Garching, Germany.

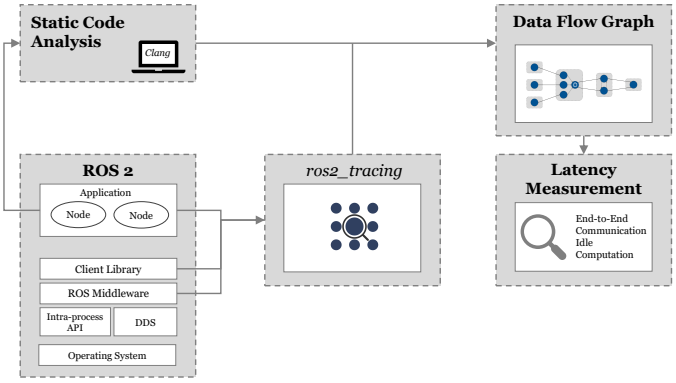

Fig. 1. General overview of the proposed method. We perform a static code analysis for the considered ROS 2 autonomous driving application. Afterward, we construct a data flow graph and annotate it with tracing data from *ros2_tracing* where we can extract different latency variables.

Previous methods for measuring end-to-end latency have the disadvantage that manual annotation of the system's code is necessary to define end-to-end paths. Due to the high complexity and number of algorithms, this annotation is inconvenient and error-prone when performed by humans. Therefore, we present a method where we automatically construct a data flow graph (DFG) for ROS 2 software using tracing data and static code analysis. We use *Clang*-based tooling to analyze intra-node data dependencies in C++ source code in an automated way. Fig. 1 shows the concept of our framework. We demonstrate our approach by analyzing *Autoware.Universe* [4], an open-source autonomous driving software. In summary, this work has three main contributions:

- We present a method to construct DFGs for complex ROS 2 autonomous driving software automatically.
- We present a method to find the intra-node data dependencies of any C++-based ROS 2 system.
- We validate the framework with an open-source autonomous driving stack and evaluate it regarding latency.

The measurement framework will be published as open-source software (https://github.com/TUM-AVS/ros2_latency_analysis).

II. Related Work

The analysis of end-to-end latencies is already receiving much attention in the literature. Especially for safety-critical applications like autonomous driving, a sensor-actuator end-to-end latency of $100\,\mathrm{ms}$ is targeted [5]. It is necessary to analyze the underlying causes of latency to meet such requirements.

Several research clusters are identified and discussed in the following.

ROS 2 Latency Analyses: Reke et al. [6] analyzed the end-to-end latency and the resulting jitter for a ROS 2 software for different real-time kernel setups [7]. Nevertheless, only applying real-time kernels in ROS 2 systems is insufficient to obtain stable runtimes since effects such as scheduling must also be considered. In [8], the end-to-end latency was analyzed for an autonomous race car, focusing on the influence in the application layer of the software. Kato et al. [4] analyzed the end-to-end runtimes for their open-source software using different hardware configurations and adapted the software accordingly to run on embedded platforms. [9] and [10] addressed the Data Distribution Service (DDS) latency issue in ROS 2 and compared the performance with ROS 1 in different use cases. In doing so, the authors explained that this is affected by the size of the data being transferred, among other factors, and can vary depending on the DDS setting. Wang et al. [11] optimized these occurring latencies between nodes by efficient inter-process communication. In [12], occurring communication latencies were included in the end-to-end latency. In doing so, they characterized the causes of overall latency of a LiDAR perception pipeline using architecture-level performance criteria. This will allow bottlenecks to be identified to optimize the software accordingly to the hardware. Overall, some literature deals with the general real-time capability of ROS 2 and scheduling. Casini et al. [13] introduced a corresponding scheduling model for ROS 2 applications and performed response-time analysis for an exemplary application. This can be used to narrow down the worst-case response time. [14] and [15] explicitly addressed the executor behavior of ROS 2. Blaß et al. [16] implemented a custom scheduler for ROS 2 threads that utilizes tracing to gain knowledge of the application structure during runtime and to find processing chains.

Tracing and Performance Frameworks: Several tracing tools for the Linux operating system make it possible to analyze software stacks more precisely. In [17], these tracing tools are benchmarked. The most frequently used tracing tool is the *Linux Trace Toolkit: next generation (LTTng)* [18]. It makes it possible to analyze kernel and user space information to make exact statements about the behavior of the executed software. For analyzing ROS 2 applications, there is already a selection of possible tools. For example, the Apex.AI *performance_test* [19] can be used to benchmark the message transfer latency for different DDS implementations. The Open Robotics *buildfarm_perf_tests* [20] takes this capability and extends it with additional metrics, such as CPU usage, average round trip, and memory usage. Like the *iRobot ros2_performance* toolkit, these frameworks are limited in their metrics or usability to very simple or even idealized publishing/subscription examples. They cannot be applied to complex real-world software. Bédard et al. [21] propose *ros2_tracing*, which is a framework suitable for tracing real-time applications. It is based on the already mentioned *LTTng* tracer. It enables the analysis of ROS 2 trace data and operating system data. In [22], the authors demonstrate the opportunity to optimize real-time applications using *ros2_tracing* tooling. The *ros2_tracing* framework is extended in [23] with the ability to analyze message flows. In doing so, the extension includes the ability to consider various dependencies between input and output messages. Also, [24] developed a framework based on *ros2_tracing*. It allows analyzing the end-to-end latency for corresponding computation paths in *Autoware.Auto*. To identify the dependencies between the callbacks, an architecture document is semi-automatically generated from trace data, with the user having to provide node-internal dependencies manually. [25] have presented an online latency monitoring framework for ROS 2-based applications which requires knowledge of the subscribe-publish relations in the analyzed nodes, as well as small additions to the nodes' and the ROS 2 source code.

ROS Data Flow Graphs: The first publication to represent a ROS 2 application formally as a graph structure is the previously mentioned work by Casini et al. [13]. Their approach is focused on ROS executors and their latency implications; thus, this graph structure is not a DFG but a causality graph. As publish-subscribe relationships are not only causal but also data dependencies, this graph structure also implements a DFG in part. However, intra-node data dependencies are not modeled and thus prevent the graph from being used as such. Choi et al. [26] propose chain-aware priority-based scheduling of ROS callbacks and define metrics on so-called callback chains. These are chains of data-dependent callbacks (publish-subscribe pairs or dependent via node-internal state) that the user manually defines. End-to-end latency is defined without DDS in mind, which makes this approach infeasible for applications with large messages that have a significant DDS latency.

Clang Tooling: LLVM [27] is a compiler toolkit that ships with Clang [28], a C/C++/Objective C compiler that provides APIs to inspect and manipulate its internal abstract syntax tree (AST), called *LibTooling* and *LibASTMatchers*. This makes it possible to find ROS nodes, callbacks and their internal data dependencies in the C++ source code in an automated way. Arroyo et al. [29] used Clang to perform taint-checking, i.e., finding all possible influences of one statement on other places in the code, and Babati et al. [30] used it to perform static analysis on C++ libraries to find portability issues. Malki et al. [31] are using Clang to inject custom statements into the code of micro-ROS applications for benchmarking purposes by detecting special user-inserted code comments. Come et al. [32] proposed a framework to find and report undesirable patterns in the C++ code of ROS and ROS applications using Clang-based tooling. Their approach uses Pangolin, a tool built using Clang's *LibTooling*, to automatically prove/disprove statements about the code which are given as temporal first-order logic formulas.

III. Method

A. Data Flow Graph

To calculate end-to-end latencies, the time between a given data output and the inputs used in its calculation must be found. We define end-to-end latency as the time between a system output and its corresponding newest available system input. It can be the case that a given input message is not the

latest for any output, i.e., when a newer input message arrived before it was processed. Thus, end-to-end latencies have to be calculated backwards, from output to input. While developers could choose to implement and propagate message headers containing this information manually, this is often not feasible in large ROS 2 projects due to a large number of dependencies. Therefore, a large number of nodes would have to be changed manually. The approach presented here builds a DFG from tracing data only, with no modifications to the source code required. Additionally, our method follows the paths through the graph from the system outputs to relevant input messages.

Graph Structure: To build the DFG, the graph-like structure of ROS 2 applications is utilized. A ROS 2 application comprises nodes that can contain callbacks that are either triggered by a timer (timer callbacks) or by messages received on a subscribed topic (subscription callbacks). Additionally, nodes can register publishers which are used to publish messages visible to all subscribers on their topic. Often, a node also has an internal state, i.e., variables that can be accessed and modified by all its callbacks. Publishers and subscriptions are assigned exactly one message type and topic and are owned by exactly one node. Within a node, publishers are accessible to all callbacks; therefore, publications do not need to always come from the same callback. Subscriptions correspond to exactly one callback. Finally, there can be multiple publishers and subscriptions on the same topic, making topics a many-to-many data flow relationship. In our method, external side effects (e.g., file system accesses) are not part of the relevant data flow through the ROS 2 application but rather outputs to the environment and are thus excluded from the DFG. The same holds for ROS 2 clients/services for long-running, non-real-time tasks.

Formal Definition of the DFG: We define the DFG as a directed graph where the vertices are ROS 2 callbacks and the edges are the data flow between them. Vertices are further grouped according to the ROS 2 nodes that their callbacks belong to. As mentioned above, data can either be passed through topics or within a node through an internal state, i.e., variables. We call the former explicit and the latter implicit data dependencies. Fig. 2 shows an example of such a DFG.

Formally, the DFG is represented by the tuple $\mathcal{DFG} = (\mathcal{V}, \mathcal{E}_\mathrm{e}, \mathcal{E}_\mathrm{i})$, with $\mathcal{V}$ being the vertices ($\triangleq$ callbacks), $\mathcal{E}_\mathrm{e} \subseteq \mathcal{V} \times \mathcal{V}$ being the explicit dependencies ($\triangleq$ publish-subscribe relationships), and $\mathcal{E}_\mathrm{i} \subseteq \mathcal{V} \times \mathcal{V}$ being the implicit ones ($\triangleq$ accesses to internal node state). The callbacks' nodes are not directly part of the graph but restrict the possible implicit dependencies: $\forall (v_1, v_2) \in \mathcal{E}_\mathrm{i}.\ v_1 \in N \iff v_2 \in N$, where N is a ROS 2 node. To obtain the real data flow through a ROS 2 application, the actual messages published and callbacks called have to be considered. We thus define the callback (cb) as a set of all its instances i_k, with t_{i_k} and d_{i_k} referring to i_k's timestamp/duration, respectively. The same is defined for publishers (pub), excluding the duration. While in graph theory, a path through a graph is defined as a sequence of edges $(e_1, ..., e_n) \in (\mathcal{E}_\mathrm{e} \cup \mathcal{E}_\mathrm{i})^n$, a data flow in this paper is a sequence of publish instances ($\triangleq$ instances of explicit dependency edges) and callback instances ($\triangleq$ instances of vertices). In such a data flow, implicit dependencies are represented implicitly by two callback instances in a row. Because the DFG contains every possible data flow in the application, all valid instance sequences can be obtained from it.

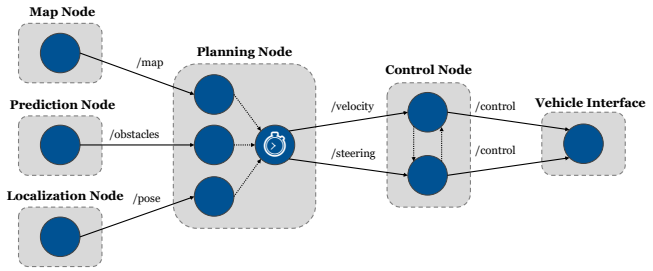

Fig. 2. A basic example of a DFG showing 6 nodes (dashed boxes) with 10 callbacks (circles), of which one is a timer callback (stopwatch icon). Explicit dependencies are represented by solid arrows annotated with a topic name, implicit dependencies are dotted arrows.

Metrics on Data Flows: Given a data flow $f = (i_1, ..., i_n)$, with each i_k being either a publish or callback instance (no consecutive publish instances are permitted), its latency can be computed that can be used for further analysis. Without loss of generality, i_n is assumed to be a publish instance. We first define latency as the difference between the end and start time of the data flow: $\mathrm{L}(f) := t_{i_n} - t_{i_1}$. For two instances directly adjacent in a data flow, the following latency components can be computed: communication ($\triangleq$ time from message publication to subscription callback start), idle ($\triangleq$ time between two callback instances within the same node), and the computation ($\triangleq$ time between callback instance start and the relevant publication within that instance that continues the data flow). Fig. 3 shows all of those latency categories in an example data flow. Formally, the communication latency between a publish instance i_k and a callback instance i_{k+1} is given by $\mathrm{L}_\mathrm{com}(i_k, i_{k+1}) = t_{i_{k+1}} - t_{i_k}$. The idle time between two callback instances i_k and i_{k+1} in the same node is calculated as $\mathrm{L}_\mathrm{idle}(i_k, i_{k+1}) = t_{i_{k+1}} - (t_{i_k} + d_{i_k})$. The computation latency of a callback instance i_k that produces the publish instance i_{k+1} during its execution is defined by $\mathrm{L}_\mathrm{calc}(i_k, i_{k+1}) = t_{i_{k+1}} - t_{k_i}$ with $t_{i_{k+1}} \leq t_{i_k} + d_{i_k}$ since the publication is made during execution. If there is no publication in a callback instance i_k and another callback instance follows it, the computation latency is instead equal to its delay: $\mathrm{L}_\mathrm{calc}(i_k) = d_{i_k}$. These definitions can be combined into a generalistic latency equation:

$$\mathrm{L}(i_k, i_{k+1}) = \begin{cases} \mathrm{L}_\mathrm{com}(i_k, i_{k+1}) & i_k \text{ is pub}, i_{k+1} \text{ is cb}, \\ \mathrm{L}_\mathrm{calc}(i_k, i_{k+1}) & i_k \text{ is cb}, i_{k+1} \text{ is pub}, \\ \mathrm{L}_\mathrm{calc}(i_k) & \\ \quad + \mathrm{L}_\mathrm{idle}(i_k, i_{k+1}) & i_k, i_{k+1} \text{ are cb}. \end{cases}$$

The sum of $\mathrm{L}(i_k, i_{k+1})$ for $k \in 0, \ldots, n-1$ for a dataflow of f of length n equals its full latency $\mathrm{L}(f)$.

B. Tracing and Data Acquisition

This section details the different methods developed for creating a DFG automatically from an unknown ROS 2 system.

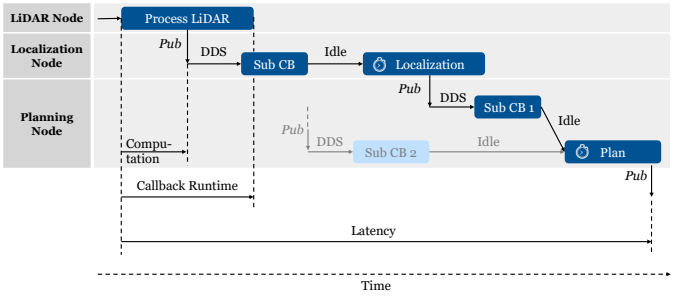

Fig. 3. Simplified data flow through a ROS 2 application. Each rounded box represents a ROS 2 callback and every arrow corresponds to the data flow from one callback to another. The different types of latency (computation, communication, idle) are annotated on those arrows. Communication is represented as DDS in the illustration.

Tracing: The *ros2_tracing* framework is the state-of-the-art tool to capture the structure and callback/publish instances of a ROS 2 application. Notably, some data relevant to the DFG is missing: links between publish instances and their corresponding callback instances and the above-mentioned implicit data dependencies in nodes. The outputs are in CTF format [33] and are converted to a Python-compatible format by the *Tracetools Analysis* library by Bédard et al. [34]. The converted tracing data is stored in a ROS 2 data model [23], from which the vertices and explicit edges of the DFG can be directly extracted.

Caveats: The IDs *ros2_tracing* outputs for ROS 2 objects are based on memory locations in the memory spaces of the applications' processes. This can lead to collisions where two objects have the same ID; for example, when two nodes of the same type are launched. Those objects cannot be told apart and thus either have to be discarded or ignored by the handling code. In the tools developed here, such objects are merged into one. Empirical evidence suggests that there are no two objects within the *Autoware* stack that have ID collisions, but there are some library nodes that do. However, this does not interfere with the DFG construction, as nodes outside the main autonomous driving stack are not of interest.

Clang Tooling: The problem of implicit data dependencies could not be solved through *ros2_tracing* because it does not have insight into the nodes' source code. With access to the source code, the accesses to a node's internal state by all of its callbacks can be analyzed, and dependencies can be added from callbacks that read the state that is modified by other ones. While this cannot generally be done automatically since data dependencies can be arbitrarily complex, an automatic analysis can be performed for simpler cases, e.g., when a variable is written to by callback A and read from by callback B. To analyze the source code of a ROS 2 application, ASTs can be used. An AST is an intermediate, tree-like representation of the program code. For C++, the Clang project [28] provides libraries and tools for working with the AST. To identify patterns relevant to node-internal data dependencies, AST matchers can be used, which are expressions that match desired patterns in the AST. For example, the following AST matcher matches all ROS 2 node definitions: `cxxRecordDecl(hasAnyBase(hasType(cxxRecordDecl(hasName("rclcpp::Node")))))`. The developed tool features AST matchers for ROS 2 nodes, their names and member method bodies, node member variable accesses and their types, calls to the `publish` method of publishers, and timer and subscription callback and publisher registrations. With all occurrences of these patterns collected, these records can be combined to find implicit data dependencies. The first step is to classify node member variable accesses as read or write access. The developed tool cannot always identify a given variable access as a read or write access due to the arbitrary complexity of C++ code. Thus, cases where this is not possible are classified as read+write to preserve all possible data dependencies. All collected accesses are then assigned to the callback they appear in. If a variable access lies in a function `f` which is not a callback, this access is assigned transitively to all functions that call `f`, thus being also assigned to the callbacks that can (transitively) call `f`. If the static analysis fails for a whole node, all callbacks are assumed to be dependent on all of the node's other callbacks. The found intra-node dependencies for each node are then merged with the DFG obtained from the tracing data. This is done by matching the node names and callback signatures.

C. End-to-End Path Extraction

While Section III-A assumes data flows to already be computed, in reality this is a complex process itself, even with an annotated DFG. This is the case because in ROS 2 applications, there is no one-to-one mapping of callback invocations and publications [24]. For example, a localization node could subscribe to IMU messages published at 100 Hz while it only publishes at 10 Hz. Or a callback might decide on its input message what or whether to publish, e.g., publishing a planned path in the normal case but an error message when the input cannot be handled. Thus, the actual dependencies of any given publish or callback instance are unknown without knowledge of the underlying source code. A solution to this problem is to compute the (feasible) set of instances any given instance can depend on and to extract the data flows from these sets. Given that not all input messages to callbacks result in outputs as discussed above, the dependencies have to be computed the other way around: from output to input. A publish instance is always caused by a callback instance, which in turn is always caused by a publish instance or timer. Thus, the proposed procedure for finding data flows is as follows.

Defining Inputs and Outputs: Step one in the above procedure is to choose the ends between which the end-to-end data flows are constructed. In the tool developed here, these ends are given as regular expressions that can match any number of topic names. Since *ros2_tracing* cannot capture input messages from outside its domain, the input topics have to be within the autonomous driving stack. A dependency tree is then built from every publish instance in any of the matched end topics until the earliest match with an input topic pattern, i.e., even when encountering a publish instance from an input topic, the tree continues until there are no inputs further down the path anymore. Furthermore, for the publish instances at

the leaves, their producing callback instance is included in the data flow to get as close as possible to the timestamp where the system is invoked by the environment.

Computing Instance Dependencies: Step two in the procedure is to find the individual dependencies of any given publish/callback instance transitively. Each instance i can theoretically depend on any instance that occured before its start timestamp and which lies on a DFG edge or node from which a path to the DFG element where i is annotated exists. This would mean that, with passing time, each new instance would depend on all past instances on such paths. While it is true that the node state can contain the effects of many past instances, this is not of interest in calculating the end-to-end latency. There, only the newest inputs that led to the output instance are important. To find those inputs, for each instance encountered, only its newest direct dependencies have to be regarded. This transitively leads to the inputs that could have affected the output instance. We define three functions for finding the newest direct dependencies of an instance: Equation (1) is the set of callback instances that a callback instance i directly depends on. Equation (2) is the set of callback instances that could have published a publish instance i. Equation (3) is the set of publish instances that could have triggered callback instance i. We define $\mathrm{cb}(i)$ to be the callback of callback instance i, $\mathrm{p}(i)$ the publisher of publish instance i, $\mathrm{cb}(s)$ the callback of subscription s, $\mathrm{cbs}(p)$ the callbacks that can use publisher p, $\mathrm{subs}(t)$ the subscribers of topic t, and $\mathrm{pubs}(t)$ the publishers of topic t. Π and σ are the projection and selection operator in relational algebra, respectively.

$$\mathrm{D_{cc}}(cb) = \Pi_{v1}(\sigma_{v2=cb}(\mathcal{E}_\mathrm{i}))$$

$$\mathrm{D_{cc}}(i) = \bigcup_{d \in \mathrm{D_{cc}}(\mathrm{cb}(i))} \{i_d \,|\, i_d \in d \wedge t_{i_d} < t_i \wedge \nexists\, i_{d2} \in d \,.\, t_{i_{d2}} > t_{i_d}\} \tag{1}$$

$$\mathrm{D_{pc}}(i) = \bigcup_{d \in \mathrm{cbs}(\mathrm{p}(i))} \{i_d \,|\, i_d \in d \wedge t_{i_d} < t_i \leq t_{i_d} + d_{i_d}\} \tag{2}$$

$$\mathrm{D_{cp}}(cb) = \{p \,|\, \exists\, t \,.\, \exists\, s \in \mathrm{subs}(t) \,.\, cb = \mathrm{cb}(s) \wedge p \in \mathrm{pubs}(t)\}$$

$$\mathrm{D_{cp}}(i) = \max_{t_{i_d}} \bigcup_{d \in \mathrm{D_{cp}}(\mathrm{cb}(i))} \{i_d \,|\, i_d \in d \wedge t_{i_d} < t_i\} \tag{3}$$

These three formulas are used iteratively to build the dependency tree. Additionally, a visited-set is maintained for each instance in the tree, such that no callback or topic can appear twice in any path through the tree. This would correspond to a loop in the DFG, which has been declared invalid in Section III-A. Without these loops and a non-infinite DFG, the dependency tree is also guaranteed to be finite. A constraint is also put on the number of consecutive callback instances in a data flow. Algorithm 1 shows the procedure of assembling a full dependency tree from a given output publish instance under the given constraints. The is_dep_cb and V arguments of that algorithm are to implement the mentioned constraints.

With a generated dependency tree T, all paths that start with a callback instance publishing on a topic matching any of the given input patterns can be extracted via simple traversal of T. Doing this for all dependency trees of all output messages results in a collection of data flows with corresponding paths through the DFG which are uniquely identified by the sequence of callbacks and topics that they comprise. By grouping the data flows by their paths, statistics on the end-to-end latency and latency categories can be computed for each path.

Algorithm 1 Dependency tree generation from a publish/callback instance

Function Signature: $\mathrm{T}(i, V = \emptyset, is\_dep\_cb = false)$

Input: Publish instance i, the set of visited callbacks and topics V, boolean is_dep_cb

Output: A tree $(head, \{subtree_1, ..., subtree_n\})$ with the dependencies of i

$C \leftarrow \emptyset$ is the set of subtrees of the current tree
if i is a publish instance **then**
  $topic \leftarrow$ the topic of i's publisher
  if $topic \in V$ **then**
    return nothing
  end if
  $deps \leftarrow \mathrm{D_{pc}}(i)$
  $C \leftarrow C \cup \{\mathrm{T}(d, V \cup \{topic\}, false) |\ d \in deps\}$
else if i is a callback instance **then**
  $cb \leftarrow i$'s callback
  if $cb \in V$ **then**
    return nothing
  end if
  $deps \leftarrow \mathrm{D_{cp}}(i)$
  $C \leftarrow C \cup \{\mathrm{T}(d, V \cup \{cb\}, false) |\ d \in deps\}$
  if not is_dep_cb **then**
    $deps \leftarrow deps \cup \mathrm{D_{cc}}(i)$
    $C \leftarrow C \cup \{\mathrm{T}(d, V \cup \{cb\}, true) |\ d \in deps\}$
  end if
end if
return (i, C)

IV. Results

A. Case Study: Application of the Method on Autoware

In the following, a real-world example of our method proposed in this paper is demonstrated to show the functionality and its potential for performing more in-depth analyses to identify bottlenecks. We run the *Autoware.Universe* software on an in-vehicle server with an AMD EPYC 7313P CPU (16 x 3.0 GHz) and 4 x 32 GB RAM. A lane driving scenario without traffic was simulated in the *Autoware* simulatior *AWSIM* to obtain sensor data. We record tracing data with *ros2_tracing* and the DFG is constructed with our proposed method. While it is possible to analyze the overall software with respect to different end-to-end paths, we limit ourselves here for clarity to the LiDAR → localization → planning → control data path as in [35], which is also the critical path of the application. This path consists of a total of 15 callbacks and has intra-node dependencies in 3 nodes. First, the end-to-end latency is analyzed. Fig. 4 shows the distribution of the end-to-end runtimes for the exemplary computation chain. An average end-to-end runtime of $152.407\,\mathrm{ms}$ can be seen. The measured maximum is $402.287\,\mathrm{ms}$. If the end-to-end runtime is broken

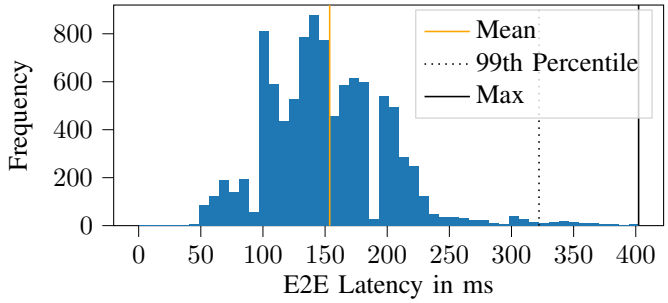

Fig. 4. Histogram of the end-to-end latency of the computation chain.

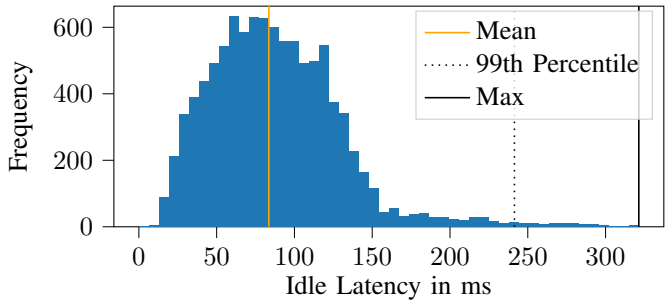

Fig. 5. Histogram of the intra-node idle latency of the computation chain.

down into the components communication, idle, and computation latency, the various influencing variables become clear. The communication latency has an average share of 3.291 ms and a maximum total runtime of 106.295 ms. The computation latency has an average runtime of 61.011 ms and a maximum value of 145.253 ms. The histogram in Fig. 5 shows that the idle latency, which occurs in intra-node dependencies before timer callbacks, takes the largest share as data arrives in subscription callbacks and sits idle until the node's timer callback runs. Table I summarizes all these statistical measures once again. Breaking down the idle runtime to the individual nodes reveals a more in-depth analysis of this part of the end-to-end latency. Table II depicts the statistical description of all intra-node idle times. It can be seen that (2), which corresponds to the intra-node dependency in `trajectory_follower` lingers unprocessed the longest. This is caused by the slow planning timer (10 Hz) and the long computation chain after

TABLE I
STATISTICAL DESCRIPTIVES OF THE END-TO-END, COMMUNICATION, IDLE, AND COMPUTATION LATENCY IN MS.

	End-to-End	Communication	Idle	Computation
Min	41.027	0.929	9.573	8.794
Mean	152.407	3.291	88.105	61.011
Std	49.766	5.732	42.986	23.798
Q25	119.429	2.012	57.556	38.622
Q50	148.706	2.767	83.536	70.051
Q75	180.304	3.263	113.154	79.662
P99	322.157	18.942	241.413	101.305
Max	402.287	106.295	321.376	145.253

it, which itself has a mean delay of circa 50 ms. Conclusions can also be drawn about the communication latency from the transmitted DDS messages. In Table III, it is clear that the transmission of the majority of messages takes less than 1 ms. The outliers regarding maximum communication latency in topics (5) and (8) can be explained through their corresponding callback timings in Table IV: Both callback (7) and (11) have high computation times and are in a single-threaded ROS 2 container with callback (8) and (10), respectively. This means that the execution of the receiving callback is blocked until the publishing one finishes, resulting in high communication latency. Further, callback (7) features a significantly higher callback duration compared to its computation latency from start to publication. The callback thus continues performing calculations irrelevant to the data flow and thus blocks callback (8) for an unnecessarily long time. This can be alleviated by removing the containerization of the nodes, or by using multi-threaded executors. The rest of the callbacks in the table have no or only a slight difference between the two metrics. Callbacks that are followed by an intra-node idle time have equal computation times and callback durations, as they do not publish any message. Multiple callbacks have very short ($\leq$10 µs) durations, as they only set a pointer to the received message and terminate, as confirmed by manual analysis of the source code. Three callbacks have a mean above 5 ms and can therefore be identified as more time-consuming computations.

B. Framework Evaluation

This section evaluates the performance and scalability of the developed method. The impact on the runtime performance of the ROS 2 application is non-significant [21], as only *ros2_tracing* is running beside it. The time required for post-processing and DFG analysis of the traced data is linearly proportional to the timespan of the recorded run. Further, the number of edges in the DFG is an exponential factor for the dataflows that must be explored. Thus, the tool caches subtrees of dependency trees and provides extensive filters to limit the explored subtrees. When filtering for a fully known path through the DFG, the dependency tree for each output message only contains one child for every node in the tree, i.e., each tree contains exactly one dataflow, speeding up processing significantly. For the scenarios analyzed in the following sections, there are 3360 output messages and therefore unique dataflows on the filtered path for which the tools runs a total of 12:11 min. This time can be decomposed into *tracetools_analysis* converting trace data into a Python-compatible format (6:53 min), dependency tree generation (3:43 min), and DFG and index structure generation (1:37 min). Algorithm 1 thus calculates a dependency tree for one output message every 60 ms in this case. If a path is not yet known, full dependency trees need to be generated. With only rudimentary filtering, e.g., excluding debug topics from the DFG, processing the same runs as above takes a total of 19:09 min with 12:56 min being spent in Algorithm 1. Since tree nodes can have more than one child in the unfiltered case, the total number of processed dataflows is 124 573 for the same 3306 output messages, resulting in a computation time

TABLE II
STATISTICAL DESCRIPTION OF ALL INTRA-NODE IDLE LATENCIES IN MS.

ROS 2 Node Name	Min	Mean	Std	P99	Max
(0) /localization/pose_twist_fusion_filter/ekf_localizer	4.643	29.222	5.253	38.076	40.000
(1) /planning/scenario_planning/lane_driving/behavior_planning/behavior_path_planner	15.972	18.784	0.409	19.756	21.191
(2) /control/trajectory_follower/controller_node_exe	0.100	55.158	38.797	213.746	273.247

TABLE III
STATISTICAL DESCRIPTION OF ALL CONSIDERED TOPICS WITH THE CORRESPONDING COMMUNICATION LATENCY IN MS.

ROS 2 Topic Name	Min	Mean	Std	P99	Max
(0) /localization/util/downsample/pointcloud	0.053	0.098	0.082	0.444	1.298
(1) /localization/pose_estimator/pose_with_covariance	0.037	0.159	0.399	2.242	6.652
(2) /localization/pose_twist_fusion_filter/kinematic_state	0.026	0.037	0.024	0.117	0.463
(3) /localization/kinematic_state	0.078	0.216	0.114	0.459	2.638
(4) /planning/scenario_planning/lane_driving/behavior_planning/path_with_lane_id	0.079	0.207	0.069	0.435	0.787
(5) /planning/scenario_planning/lane_driving/behavior_planning/path	0.157	0.749	4.385	0.982	79.325
(6) /planning/scenario_planning/lane_driving/motion_planning/obstacle_avoidance_planner/trajectory	0.157	0.238	0.229	0.639	5.683
(7) /planning/scenario_planning/lane_driving/trajectory	0.056	0.371	0.251	1.304	4.021
(8) /planning/scenario_planning/scenario_selector/trajectory	0.068	0.766	2.772	20.368	29.130
(9) /planning/scenario_planning/trajectory	0.078	0.299	0.422	2.223	4.458
(10) /control/trajectory_follower/control_cmd	0.110	0.270	0.296	1.348	2.324

TABLE IV
STATISTICAL DESCRIPTION OF THE COMPUTATION LATENCIES AND CALLBACK DURATIONS IN MS.

ROS 2 Callback Signature	Computation					Callback				
	Min	Mean	Std	P99	Max	Min	Mean	Std	P99	Max
(0) Filter::(PointCloud2,PointIndices)	0.010	0.016	0.020	0.049	0.419	0.028	0.115	0.392	2.101	5.329
(1) NDTScanMatcher::(PointCloud2)	1.552	9.183	5.019	22.299	35.707	1.584	9.286	4.928	21.971	35.858
(2) EKFLocalizer::(PoseWithCovarianceStamped)	0.001	0.003	0.004	0.022	0.057	0.001	0.003	0.003	0.011	0.099
(3) EKFLocalizer::()	0.533	0.902	0.161	1.323	3.513	0.474	0.940	0.178	1.596	3.530
(4) StopFilter::(Odometry)	0.005	0.007	0.002	0.014	0.021	0.073	0.113	0.027	0.241	0.425
(5) BehaviorPathPlannerNode::(Odometry)	0.000	0.002	0.008	0.004	0.253	0.000	0.002	0.005	0.004	0.253
(6) BehaviorPathPlannerNode::()	1.501	3.920	1.333	6.445	8.146	1.635	4.314	1.467	6.792	14.464
(7) BehaviorVelocityPlannerNode::(PathWithLaneId)	1.004	17.163	11.917	33.603	38.238	1.121	17.423	11.946	34.432	98.398
(8) ObstacleAvoidancePlanner::(Path)	0.235	3.960	4.762	26.782	52.311	0.244	4.031	4.945	27.175	72.601
(9) ObstacleStopPlannerNode::(Trajectory)	0.222	2.373	3.527	11.663	12.002	0.239	4.034	4.614	11.980	21.330
(10) ScenarioSelectorNode::(Trajectory)	0.002	0.006	0.001	0.009	0.013	0.030	0.124	0.048	0.245	0.405
(11) MotionVelocitySmootherNode::(Trajectory)	1.013	20.735	10.630	34.099	45.087	1.194	20.996	10.627	34.451	49.472
(12) Controller::(Trajectory)	0.000	0.001	0.002	0.002	0.041	0.000	0.001	0.003	0.003	0.062
(13) Controller::()	0.909	1.283	0.269	2.194	6.671	1.000	1.390	0.380	2.315	16.305
(14) VehicleCmdGate::(AckermannControlCommand)	0.008	0.017	0.022	0.047	0.859	0.044	0.120	0.058	0.243	3.674

of $8\,\mathrm{ms}$ per dataflow and $231\,\mathrm{ms}$ per dependency tree. This improvement from $60\,\mathrm{ms}$ to $8\,\mathrm{ms}$ comes from the subtree caching mechanism, which provides a 7.5 times speedup over a non-cached version. Finding the path to analyze is thus possible by first outputting all unfiltered paths and narrowing those down through high-level system knowledge, with only minimal or no analysis of the ROS 2 application source code. This is a clear advantage of this approach over the current state of the art, which requires either a manually created architecture file [24] or manually inserted tracepoints with manually extracted information about node-internal dependencies and publisher-callback associations [23]. All processing has been done on a computer with an Intel i7-8550U CPU and $32\,\mathrm{GB}$ of RAM. The static analysis approach provides a reduction in the node-internal dependencies that have to be explored during dependency tree generation. As such, it reduces the number of edges in the `BehaviorPathPlannerNode`, which features 15 subscription callbacks and one timer callback, from 240 to 30. Since the approach cannot discriminate between read and write accesses and thus mostly outputs bidirectional data dependencies between callbacks, this number is still higher than the actual 15 dependencies. Nevertheless, this approach yields a $87.5\,\%$ reduction in edges over the non-static analysis approach in this node alone, compounding over the whole graph.

V. DISCUSSION & CONCLUSION

The method presented allows users to analyze ROS 2-based autonomous driving software regarding its end-to-end latencies along different paths without knowledge of the code. This is done by extracting intra-node data dependencies via static analysis of the C++ source code and relating this information with *ros2_tracing* data in a comprehensive DFG. Paths can be extracted from the DFG using user-defined filters. End-to-end latencies and their breakdown into the comprising callback, communication, and idle latencies can then be visualized for further analysis by the user. Our proposed method cannot detect all node-internal dependencies using Clang AST matchers and thus introduces a small amount of superfluous data dependencies. Additionally, *ros2_tracing* outputs function signatures

without the function names, making collisions of callback signatures possible. While this is rare even in software such as *Autoware.Universe*, those collisions force our implementation to discard potential matches between tracing and Clang data to guarantee correctness. Superfluous dependencies stemming from this approach can, however, easily be identified and discarded manually by the end user through the previously mentioned filters, with high-level knowledge of the ROS 2 application at hand. These metrics increase the understanding in cases where the latency is higher than expected and thus directly help developers trying to optimize their applications. We applied the developed framework to *Autoware.Universe*, an open-source software stack for autonomous vehicles, showing that current state-of-the-art ROS 2-based autonomous driving software exhibits high tail latencies and idle times that are not yet suitable for safe deployment in the real world. In the future, we must therefore investigate how we can optimize and reduce the latencies. We will make future iterations of the presented tools and enhance the *ros2_tracing* output with more data, especially the mentioned function names. In addition, defining more AST matchers to handle more advanced C++ constructs also results in a direct reduction of false-positive data dependencies.

REFERENCES

[1] E. Talpes, D. D. Sarma, G. Venkataramanan, P. Bannon, B. McGee, B. Floering, A. Jalote, C. Hsiong, S. Arora, A. Gorti, and G. S. Sachdev, "Compute solution for tesla's full self-driving computer," *IEEE Micro*, vol. 40, no. 2, pp. 25–35, 2020.

[2] S. D. Pendleton, H. Andersen, X. Du, X. Shen, M. Meghjani, Y. H. Eng, D. Rus, and M. H. Ang, "Perception, planning, control, and coordination for autonomous vehicles," *Machines*, vol. 5, no. 1, 2017.

[3] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall, "Robot operating system 2: Design, architecture, and uses in the wild," *Science Robotics*, vol. 7, no. 66, p. eabm6074, 2022.

[4] 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)*. IEEE, 2018.

[5] S.-C. Lin, Y. Zhang, C.-H. Hsu, M. Skach, M. E. Haque, L. Tang, and J. Mars, "The architectural implications of autonomous driving: Constraints and acceleration," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018, pp. 751–766.

[6] M. Reke, D. Peter, J. Schulte-Tigges, S. Schiffer, A. Ferrein, T. Walter, and D. Matheis, "A self-driving car architecture in ros2," in *2020 International SAUPEC/RobMech/PRASA Conference*, 2020, pp. 1–6.

[7] F. Reghenzani, G. Massari, and W. Fornaciari, "The real-time linux kernel: A survey on preempt_rt," *ACM Computing Surveys*, vol. 52, no. 1, pp. 1–36, 2020.

[8] T. Betz, P. Karle, F. Werner, and J. Betz, "An analysis of software latency for a high-speed autonomous race car - a case study in the indy autonomous challenge," *SAE International Journal of Connected and Automated Vehicles*, vol. 6, no. 12-06-03-0018, 2023.

[9] Y. Maruyama, S. Kato, and T. Azumi, "Exploring the performance of ros2," in *Proceedings of the 13th International Conference on Embedded Software*. New York, NY, USA: ACM, 2016.

[10] T. Kronauer, J. Pohlmann, M. Matthé, T. Smejkal, and G. Fettweis, "Latency analysis of ros2 multi-node systems," in *2021 IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems (MFI)*. IEEE, 2021, pp. 1–7.

[11] Y.-P. Wang, W. Tan, X.-Q. Hu, D. Manocha, and S.-M. Hu, "Tzc: Efficient inter-process communication for robotics middleware with partial serialization," in *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2019.

[12] P. H. E. Becker, J. M. Arnau, and A. Gonzalez, "Demystifying power and performance bottlenecks in autonomous driving systems," in *2020 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2020.

[13] D. Casini, T. Blaß, I. Lütkebohle, and B. B. Brandenburg, "Response-time analysis of ros 2 processing chains under reservation-based scheduling," in *31st Euromicro Conference on Real-Time Systems*, 2019, pp. 1–23.

[14] Y. Tang, Z. Feng, N. Guan, X. Jiang, M. Lv, Q. Deng, and W. Yi, "Response time analysis and priority assignment of processing chains on ros2 executors," in *2020 IEEE 41st Real-Time Systems Symposium*, G. Nelissen, Ed. Piscataway, NJ: IEEE, 2020, pp. 231–243.

[15] T. Blaß, D. Casini, S. Bozhko, and B. B. Brandenburg, "A ros 2 response-time analysis exploiting starvation freedom and execution-time variance," in *2021 IEEE 42nd Real-Time Systems Symposium*. Piscataway, NJ: IEEE, 2021, pp. 41–53.

[16] T. Blaß, A. Hamann, R. Lange, D. Ziegenbein, and B. B. Brandenburg, "Automatic latency management for ros 2: Benefits, challenges, and open problems," in *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium*, G. Nelissen, Ed. Piscataway, NJ: IEEE, 2021, pp. 264–277.

[17] M. Gebai and M. R. Dagenais, "Survey and analysis of kernel and userspace tracers on linux," *ACM Computing Surveys*, vol. 51, no. 2, pp. 1–33, 2019.

[18] M. Desnoyers and M. R. Dagenais, "The lttng tracer: A low impact performance and behavior monitor for gnu/linux," in *OLS (Ottawa Linux Symposium)*, vol. 2006, 2006, pp. 209–224.

[19] Apex.AI, "performance_test." [Online]. Available: https://gitlab.com/ApexAI/performance_test

[20] Open Robotics: buildfarm perf tests. [Online]. Available: https://github.com/ros2/buildfarm_perf_tests

[21] C. Bédard, I. Lütkebohle, and M. Dagenais, "ros2_tracing: Multipurpose low-overhead framework for real-time tracing of ros 2," *IEEE Robotics and Automation Letters*, vol. 7, no. 3, pp. 6511–6518, 2022.

[22] P.-Y. Lajoie, C. Bédard, and G. Beltrame, "Analyze, debug, optimize: Real-time tracing for perception and mapping systems in ros 2." [Online]. Available: https://arxiv.org/pdf/2204.11778

[23] C. Bédard, P.-Y. Lajoie, G. Beltrame, and M. Dagenais, "Message flow analysis with complex causal links for distributed ros 2 systems." [Online]. Available: https://arxiv.org/pdf/2204.10208

[24] Z. Li, A. Hasegawa, and T. Azumi, "Autoware_perf: A tracing and performance analysis framework for ros 2 applications," *Journal of Systems Architecture*, vol. 123, p. 102341, 2022.

[25] J. Peeck, J. Schlatow, and R. Ernst, "Online latency monitoring of time-sensitive event chains in safety-critical applications," in *Proceedings of the 2021 Design, Automation & Test in Europe (DATE 2021)*. Piscataway, NJ: IEEE, 2021, pp. 539–542.

[26] H. Choi, Y. Xiang, and H. Kim, "Picas: New design of priority-driven chain-aware scheduling for ros2," in *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2021, pp. 251–263.

[27] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004*, 2004, pp. 75–86.

[28] "Clang: a c language family frontend for llvm." [Online]. Available: http://clang.llvm.org

[29] M. Arroyo, F. Chiotta, and F. Bavera, "An user configurable clang static analyzer taint checker," in *2016 35th International Conference of the Chilean Computer Science Society (SCCC)*. IEEE, 2016, pp. 1–12.

[30] B. Babati, G. Horváth, V. Májer, and N. Pataki, "Static analysis toolset with clang," in *Proceedings of the 10th International Conference on Applied Informatics (30 January–1 February, 2017, Eger, Hungary)*, 2017, pp. 23–29.

[31] A. Malki, T. Kołcon, and M. Maciaś, "Ofera: Open framework for embedded robot applications," 2020.

[32] D. Come, J. Brunel, and D. Doose, "Improving code quality in ros packages using a temporal extension of first-order logic," *Encyclopedia with Semantic Computing and Robotic Intelligence*, vol. 2, no. 01, p. 1850003, 2018.

[33] M. Desnoyers, "Common trace format (ctf) specification (v1. 8.2)," *Common Trace Format GIT repository*, 2012.

[34] C. Bédard, I. Lütkebohle, and T. Blaß, "Tracetools analysis," https://gitlab.com/ros-tracing/tracetools_analysis, 2019.

[35] T. Betz, M. Schmeller, H. Teper, and J. Betz, "How fast is my software? latency evaluation for a ros 2 autonomous driving software," in *2023 IEEE Intelligent Vehicles Symposium (IV) (IEEE IV 2023)*. IEEE, 2023.