Data-flow Availability: Achieving Timing Assurance in Autonomous Systems

Ao Li and Ning Zhang, *Washington University in St. Louis*

https://www.usenix.org/conference/osdi24/presentation/li

This paper is included in the Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation.

July 10–12, 2024 • Santa Clara, CA, USA

978-1-939133-40-3

Data-flow Availability: Achieving Timing Assurance on Autonomous Systems

Ao Li Ning Zhang
Washington University in St. Louis

Abstract

Due to the continuous interaction with the physical world, autonomous cyber-physical systems (CPS) require both functional and temporal correctness. Despite recent advances in the theoretical foundation of real-time computing, leveraging these results efficiently in modern CPS platforms often requires domain expertise, and presents non-trivial challenges to many developers.

To understand the practical challenges in building real-time software, we conducted a survey of 189 software issues from 7 representative CPS open-source projects. Through this exercise, we found that most bugs are due to misalignment in time between cyber and physical states. This inspires us to abstract three key temporal properties: freshness, consistency, and stability. Using a newly developed concept, Data-flow Availability (DFA), which aims to capture temporal/availability expectation of data flow, we show how these essential properties can be represented as timing constraints on data flows. To realize the timing assurance from DFA, we designed and implemented Kairos, which automatically detects and mitigates timing constraint violations. To detect violations, Kairos translates the policy definition from the API-based annotations into run-time program instrumentation. To mitigate the violations, it provides an infrastructure to bridge semantic gaps between schedulers at different abstraction layers to allow for coordinated efforts. End-to-end evaluation on three real-world CPS platforms shows that Kairos improves timing predictability and safety while introducing a minimal 2.8% run-time overhead.

1 Introduction

Recent advances in artificial intelligence and robotics have promoted the integration of various autonomous cyber-physical systems into society, including self-driving cars [94], drones [31], and home-service robots [32]. Unlike conventional systems, CPS has to sense the physical world, compute for the appropriate control actions, and actuate on the physical world in a timely manner. Therefore, the assurance of temporal properties in autonomous CPS is fundamental to the correctness of the system.

System Challenges in Real-time Cyber-physical Systems. Recognizing its importance, the real-time systems community has devoted significant effort to ensuring the timeliness of computation. However, despite the rich literature on the theoretical foundation of real-time computing, such as schedulability analysis [70], mixed-criticality scheduling [43], and compositional scheduling [51, 83], leveraging these results in the development of CPS software remains quite challenging for non-experts. Furthermore, recent advances in multi-core execution and multi-modal sensing also make the problem challenging even for experts, with plenty of open research questions that are actively being investigated [66, 70]. A recent industrial survey [29] (Question 23) also indicates that only a small fraction (9.38%) of systems are designed with commercial schedulability analysis tools.

Understanding Timing Problems in Real-world CPS. To gain a better understanding of system challenges in CPS, we draw inspiration from the recent survey on concurrency bugs [64, 67], and conducted a systematic study of the 189 timing bugs in 7 mainstream open-source CPS software projects. Our goal is to understand the categories of timing bugs in CPS applications, the root causes of each bug category, as well as the challenges developers face in preventing them. We found that most of the timing bugs are caused by misalignment in time between cyber states and physical states. Therefore, building on top of the cyber-physical control loop abstraction, we extracted three most essential temporal properties: freshness, consistency, and stability. Furthermore, we found that many existing mitigations implemented manual checks for data timestamps, inspiring us to model the problem from a data-flow perspective.

Our solution - Data-flow Availability. Motivated by the findings from the timing bug study, we propose Data-flow Availability, a new concept that achieves timing assurance in autonomous systems. Building on the observation that data flow drives cyber-physical control loops in modern CPS, we

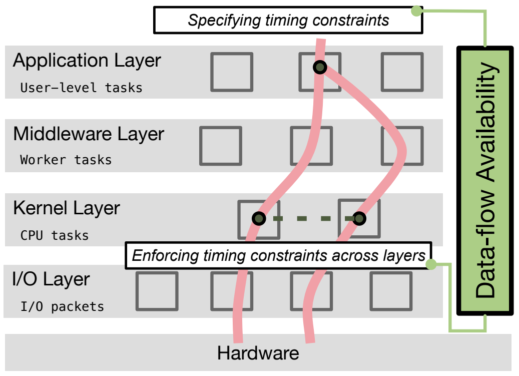


Figure 1: Data-flow Availability in the system stack.

augment data flow with a new temporal dimension, resulting in *Timed Data-flow Graph* (TDFG). Conceptually, each variable (that captures cyber or physical state) would have a time attribute (tag), and information flow among them has to respect the expectation of the software. Therefore, temporal policy is encoded as timing constraints on the edges of the graph.

To realize the concept of data-flow availability in system, we design and develop Kairos, a programming model to enable the automatic detection and mitigation of timing constraint violations. Kairos consists of a DFA-embedding tool for detection of timing constraint violation at run-time and a cross-layer scheduling system for mitigation (as shown in Figure 1). Using the DFA-embedding tool, developers can either manually annotate the source code with APIs provided by Kairos or use the provided dynamic profiler to specify the expected temporal properties. A compiler extension then takes the temporal expectation, expressed as data-flow constraints, and automatically instruments the software to detect timing constraint violations at run-time. However, detection alone does not provide timing assurance. Upon timing constraint violations, actions have to be taken to recover the system. To do so, Kairos builds on the concept of schedulable entity path to construct an association of schedulable entities in different abstraction layers of the operating system for a cyber-physical data flow. This bridges the semantic gap between the abstraction layers, and allows for more effective coordination of schedulers in the system for violation mitigation.

Prototype and Evaluation. To understand the effectiveness of DFA in mitigating the timing bugs, we analytically studied how existing bug fixes can be implemented using Kairos, and found that among the 189 bugs, 111 of them can be mitigated by Kairos. To understand the performance characteristics of Kairos, we built a prototype of Kairos, and evaluated it on three real-world robotic platforms: Autoware [36], Jackal UGV [59], and Turtlebot3 [89], each with distinct workloads and computing power requirements. On these three platforms, we show how TDFG can be constructed and used in Kairos to mitigate the existing timing issues. At runtime, Kairos introduces an average overhead of 2.8% and shows manageable performance under scalability analysis. Under high system overload, Kairos shows a faster and more stable response time in reacting to timing violations compared to other state-of-the-art systems – ROS [80], ERDOS [55], and ghOSt [57]. Furthermore, the end-to-end evaluation shows that Kairos can improve safety under high system overload.

Contributions. We make the following contributions [1]:

- Formulation of Data-flow Availability, a new concept for achieving timing assurance from a data-flow perspective.
- Design and implementation of Kairos, a proof-of-concept realization of DFA. Kairos detects timing violations by embedding a temporal property monitor within the application and mitigates these violations through a cross-layer scheduling infrastructure.
- Evaluation of DFA and Kairos across three real-world robotic platforms, each with distinct workloads and operational domains.

2 Background

Real-time Cyber-physical Systems. A unique characteristic of autonomous cyber-physical systems is their tight connection to physical world processes. Cyber-physical systems software often builds on top of the abstraction of a cyber-physical control loop, which continuously senses the physical world, calculates the appropriate control actions, and then actuates on the system to reach the desired state. The implementation of this control loop is often realized using multiple tasks (processes), where each is modeled as either periodic or sporadic tasks in the real-time models.

Timeliness Abstraction in Cyber-physical Systems. Due to their cyber-physical nature, the correctness of autonomous systems depends on both functional correctness and temporal correctness. To achieve this, real-time schedulability analysis [77] is conducted on each system based on the real-time task models. Meeting deadlines is often considered the most important requirement in real-time systems. Using the task parameters from the schedulability analysis, the scheduler of the system enforces temporal isolation among the tasks, ensuring no task misses its deadline. Based on the ability to tolerate deadline misses, systems can be hard, firm, or soft real-time. Due to various practical challenges, such as difficulty in determining a real-time task model, the efficiency of the processor to achieve system guarantees, and accurate estimation of worst case execution time, many deployed real-time systems are soft real-time systems, according to a recent

[1]The source code, as well as the extended version of this work with additional analysis and experiments is available at https://dataflow-availability.github.io/.

industry survey [29]. Furthermore, timing constraints can manifest in properties other than deadline misses, including but not limited to task response time, execution time, release jitter, and response jitter.

Timeliness Implementation in Cyber-physical Systems. Modern autonomous systems generally involve multiple abstraction layers, as shown in Figure 1. Besides the typical userspace and kernel-space layers, modern CPS software also utilizes middleware, such as the robotic operating system (ROS) [68, 80] to ease programming. Some CPS software even implements their own userspace scheduler within the application, resulting in time management across multiple layers of abstractions. This presents unique challenges for developers in achieving alignment of cyber events with physical world events. Furthermore, there is often a combination of time-driven [33] or event-driven [80] tasks.

3 Real-world Timing Bug Study

Motivation. Real-time theory suggests modeling individual computations as individual tasks. However, developing a real-time task model for modern complex data-driven CPS can be quite challenging for non-experts. Further, the formulation of highly efficient task models often requires deep domain expertise in real-time scheduling. A recent industrial survey [29] (Question 23) also indicates that only a small fraction (9.38%) of systems are designed with commercial schedulability analysis tools. Inspired by existing studies on concurrency bugs [64, 67] that had offered key insights to the community, we conducted a systematic study of timing bugs in 7 open-source robotic software. The goal is to gain a better understanding of the underlying practical challenge faced by developers. As such, the focus of the study is on *timing bugs*, where the bug is caused by non-deterministic timing of data flow within the cyber-physical system.

Methodology. The seven selected open-source GitHub robotic software projects are Autoware [53], MoveIt [2], Google Cartographer [56], Baidu Apollo [37], ORB-SLAM2/3 [3, 4], ROS Navigation [5], and ROS2 rcl [7]. These projects were selected because they represent important subsystems in modern cyber-physical control loops, including perception, localization, planning, and control. Furthermore, they have also been widely adopted [1, 35, 52, 75]. To collect the bugs, a set of keywords (e.g., 'timing', 'sched', 'timestamp', 'temporal', etc.) was used to filter the issues, resulting in a list of 189 bugs.

Summary of Systemization. As shown in Table 1, we find that two categories of root causes account for the majority (169 out of 189) of the collected timing bugs: insufficient specification and enforcement of timing constraints. The rest are design flaws and hardware problems.

Table 1: Timing Bugs in Real-world Applications

Projects	# bugs	Timing Constraint Specification			Timing Constraint Enforcement		Others
		Missing Constraint	False Specification		Missing Constraint	False Enforcement	
			Expressibility	Parameter			
Cartographer [47]	34	14	12	1	1	4	2
Apollo [37]	49	11	23	2	3	0	10
MoveIt [2]	23	4	7	2	2	5	3
ORB-SLAM [4]	6	1	1	0	1	2	1
Autoware [53]	16	6	5	0	1	0	4
Navigation [5]	15	3	3	1	2	4	2
ROS rcl [7]	46	3	3	1	3	35	1
Total	189	42	54	7	13	50	23
Scope of This Work		✓	✓	✓	✓		

3.1 Timing Specification Bugs

The most common cause of timing assurance failure is incorrect specification of timing constraints (103/189 bugs). As discussed earlier, though real-time theory provides a sound foundation for assuring timing behavior, there remains a gap in transitioning the theory into practice for developers without expertise in real-time computing. Without the formal guarantees provided by real-time theory tools (such as schedulability analysis), current practice adopted by developers to mitigate this involves developers tagging data with timestamps when data is created or transferred, and then using these timestamps to check the data's validity (e.g., freshness) when it is used. This data-centric approach to timestamp checking for specifying timing constraints is ubiquitous in the codebases we investigated. For example, Autoware and Google Cartographer use timestamp checks in over 340 and 110 places, respectively, to determine the execution logic. Additionally, state-of-the-art middleware such as ROS [80], ROS2 [68], and ERDOS [55] also incorporate built-in timestamps on data transferred between tasks.

3.1.1 Missing Time Constraints (What to Check)

Figuring out where and how to add the timestamp checks manually is quite challenging due to the complex dependency among data from different tasks [8, 12, 14, 17, 19–21, 23, 50, 54]. Naively, one can simply add timing checks on all instructions. However, that will introduce prohibitive overhead to the system, leading to adverse physical outcomes.

Implication - It is essential to understand not only which program statements need to be checked but also which aspects of temporal properties should be verified, in order to minimize the performance impact of the protection.

To further dive into the root cause of the problem in a principled approach, we went back to the basic abstraction of a cyber-physical control loop to ask the question of what properties are these timing bugs violating. Through the lens of physical world impact, three key properties arise during the analysis of the timing violations.

Freshness - *describes the latency between the occurrence of a physical phenomenon and the consumption of its cyber rep-*

resentation. While data should be as fresh as possible, there will always be some delay due to sensing and computation.

The key is to ensure that the freshness of the particular data is acceptable by the control implementation. Figure 2 shows an example from Cartographer-Pull-153. Cartographer [47] uses a queue to manage and process sensor data streams from multiple sources in a coordinated, time-ordered manner. It then uses the data to construct a robot's trajectory for localization. The code snippet checks if the incoming data is older than the start time of the current trajectory and discards the outdated data if it is.

```
void OrderedMultiQueue::Dispatch() {
  // We take a peek at the time after next data. If it
  // is not beyond 'common_start_time' we drop it
  std::unique_ptr<Data> next_data = next_q->queue.Pop();
  if (next_q->queue.Peek()->time > common_start_time) {
    last_dispatched_time_ = next_data->time;
    next->callback(std::move(next_data));
  }
  // else: drop the data
```

Figure 2: Freshness check where outdated data is dropped. Simplified code snippet from Cartographer-Pull-153.

Consistency - *describes the temporal alignment of the physical world observations in the data flows converging at a specific statement of the program. Ideally, the physical events captured by these cyber states should be as synchronized as possible.*

Figure 3 shows an example from ROS Navigation [5] (Navigation-Pull-1121), where the control task retrieves the robot pose using the `tf_` buffer, which maintains historical poses. In the original code (highlighted in red in line 1), it directly uses the latest pose. However, since the `tf_` buffer is dynamically updated by other tasks, the timestamp of the current map used by the control task (`time`) might be older than the latest pose in `tf_`. This could result in using a pose that is ahead of the current map in time, causing motion planning to produce incorrect paths. To fix this, the code highlighted in green (in line 3) adopts a timestamp-based check that compares the timestamp of `tf_` with the control task's timestamp. If `time` is not newer than the latest in `tf_`, the `lookupTransform()` function is called to interpolate the pose that temporally aligns with the current map.

```
 1 tf_.transform(robot_pose, global_pose, global_frame);
 2  // check if curr_time is less than latest update time of tf_
 3 if (tf_.canTransform(global_frame, robot_base_frame, curr_time)) {
 4   // if so, transfomr at the time point of curr_time
 5   transform = tf_.lookupTransform(global_frame,
 6                                   robot_base_frame,
 7                                   current_time);
 8   tf2::doTransform(robot_pose, global_pose, transform);
 9 } else {
10   // use the latest otherwise
11   tf_.transform(robot_pose, global_pose, global_frame_);
12 }
```

Figure 3: Consistency check detecting temporal alignment between the data from two tasks (Navigation-Pull-1121).

Stability - *describes the variation in freshness. This is similar to the concept of jitter in the real-time and control domains, and ideally jitter should be minimized.*

Many control algorithms and systems are designed to have an implicit assumption of not only the boundary of the freshness but also its variation (often relatively small) from loop to loop. In essence, it is about the consistency of data flow in the temporal dimension, as compared to the spatial dimension (consistency as discussed above). Figure 4 shows a code snippet from AutowareAuto-Pull-980, where a timer is added to ensure the stability of control output. The timer checks the elapsed time in a polling loop to trigger the control output function at expected intervals.

```
NERaptorInterface::NERaptorInterface(. . .){
  /* Use a ROS timer to ensure the stability */
  m_timer = node.create_wall_timer(m_pub_period,
                                   std::bind(&cmdCallback, this));
 }

/* In implementation of ROS timer */
while (rclcpp::ok()) {
// Use elasped time to check if timer is ready via a polling loop
  rcl_timer_get_time_until_next_call(m_timer, &time_until_next_call);
  if (time_until_next_call <= 0)    m_timer->call();
}
```

Figure 4: Stability check using a ROS timer to minimize control jitters (AutowareAuto-Pull-980).

Summary - These three key properties present a unique opportunity to address a large number of bugs with a small amount of temporal property checks.

3.1.2 Inadequate Timing Constraints (How to Check)

Even after solving the challenge of what to check, developers also have to tackle the challenge of how to check. There are 61 bugs caused by inadequate specification of timing constraints; among these, we found two common causes. The first category is that some of the hard-coded time constraints may not be appropriate for the deployment. This often happens due to insufficient testing or changing software/hardware [11] or operating environment [55] of the system. The second category is less straightforward. In real-time cyber-physical systems, there are other essential timing dimensions beyond latency (maps to freshness discussed earlier), such as alignment (maps to consistency discussed earlier) and jitter (maps to stability discussed earlier). For examples, issues can arise on arrival jitter [10], detection of data loss [24], processing data time ratios [13, 27], and requests of development of timing utilities [9, 16]. Figure 5 shows a simplified bug example from Google Cartographer [47] (Cartographer-Issue-242) that spans multiple patches before being finally fixed. The code snippet estimates the robot's velocity by dividing the difference in positions between two adjacent frames by the time interval. In this case, the freshness of data is a problem because if the incoming LiDAR frame is older than the latest one (i.e., out-of-order), it causes the time difference (`delta_t`) to be negative. The freshness check was added in patch [79] (highlighted in yellow on line 3 of the figure). However, another problem beyond data freshness persists even if `delta_t` is positive. The irregular timing may result in two LiDAR frames being too close in time, causing `delta_t` to be too short. In such cases, the position difference is divided by a very small `delta_t` value, which can significantly magnify any estimation errors, potentially causing the velocity to become infinitely large. This issue is finally fixed by inserting a check (on line 8) that the frames with intervals less than 1 ms are dropped.

```
 1 // Estimate the velocity estimate.
 2 if (time >  common::Time::min()
 3     && time > last_scan_match_time) {
 4
 5   // Prevent out-of-order data
 6   double delta_t = common::ToSeconds(time - last_scan_match_time);
 7
 8   if (delta_t < 1e-6) return;
 9   // Prevent too short intervals
10   velocity_estimate_ += (pose_estimate_.translation() -
11                          model_prediction.translation()) /
12                          delta_t;
13 }
```

Figure 5: Timestamp checking is incomplete in semantic.

Implication - Given the dynamic range of timing expectations in different platforms and physical environments, it is important to develop a mechanism that simplifies the configuration of these ranges for developers. Ideally, this mechanism should also enable the automatic discovery of the necessary ranges to maintain system safety.

3.2 Timing Enforcement Bugs

There are 62 bugs stemming from inadequate enforcement of timing constraints. Most of these (50/62) are due to conventional software bugs, such as memory corruption in the enforcement infrastructure with schedulers and timers. Another 13 of these bugs are caused by timing constraints not being delivered to the enforcement mechanisms. These are primarily caused by the fact that the specified timing constraints are limited to userspace applications and are not propagated to other scheduling layers. Figure 1 schematically illustrates the scheduling layers involved in designing and deploying autonomous systems. Due to the inadequate support for delivering scheduling contexts across schedulers, timing constraints (or scheduling decisions) specified at one scheduler fail to propagate to others. This type of problem can be observed in issues where priorities are inverted across layers [22], leading to critical tasks not being reliably triggered [26], executing at varying periods [25], or executing out of order [15]. As a result, mitigation methods to maintain relative priority at the user level or middleware alone are often quite challenging if not impossible.

```
void SchedulerChoreography::CreateProcessor() {
  proc->BindContext(ctx);
  /* Reserve a set of CPU cores for the tasks */
  SetSchedAffinity(proc->Thread(), pool_cpuset_, pool_affinity_, i);
  SetSchedPolicy(proc->Thread(), pool_processor_policy_,
                 pool_processor_prio_, proc->Tid());
}
```

Figure 6: Mitigating disconnection across scheduling layers.

Figure 6 is the mechanism adopted to handle Apollo-Issue-9433. It introduces a new scheduling strategy that reserves a set of CPU cores for middleware tasks, allowing them to be directly scheduled on these cores and avoiding disconnection between layers. However, implementing this scheduling strategy necessitates a thorough understanding of the tasks, including their dependencies and execution times.

Implication - Assurance of timing expectations is more effective when scheduling contexts are visible across all layers of abstraction.

3.3 Summary

Table 1 shows the timing bugs we studied, and the scope of the proposed mechanism DFA. Based on the study, we summarize the opportunities and insights that inform the design of DFA:

- Timing expectations are often added by programmers by checking the age of data, hinting at the potential to use information(data)-flow as a mechanism to capture the programmer's intention.
- There are three types of key temporal properties we systematized based on the cyber-physical control loop abstraction, freshness, consistency, and stability.
- Timing enforcement would often benefit from visibility across different layers of abstractions in the OS.

4 Data-flow Availability

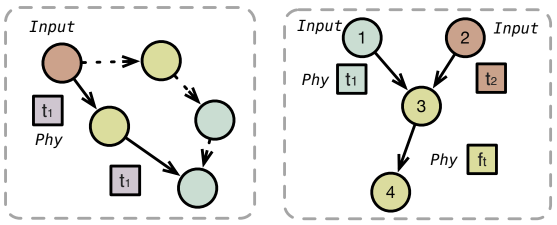
Motivated by the challenges in Section 3, this paper introduces ***Data-Flow Availability*** (DFA), which approaches the policy definition of temporal property from a data-flow perspective.

4.1 Timed Data-flow Graph

A *Timed Data-flow Graph* (TDFG) is a representation of a DFA-enabled program, extended from the program's data-flow graph.

Graph Definition. TDFG is a directed graph $\mathbf{G} = (\mathbf{V}, \mathbf{E}, \mathbf{T}, \mathbf{C})$ constructed from program's intermediate representation:

- *Vertex:* Each vertex v in the set $\mathbf{V}$ corresponds to a statement in the intermediate representation.
- *Edge:* Edges $\mathbf{E} \subseteq \mathbf{V} \times \mathbf{V}$ represent data dependencies between vertices. An edge is added if the corresponding statements have a data dependency.
- *Timing Tag:* A timestamp $t_{phy} \subseteq \mathbf{T}$ is generated with `def` of memory SSA (Static Single Assignment) in the graph and propagated along edges at runtime. It includes two types of timing, either the physical world sensor reading or the range of a derived value from the timed sensor values.



(a) Sequential propagation. (b) Merge from different inputs.

Figure 7: Two cases of timing propagation in data flows.

```
void EvaluatorManager::DumpCurrentFrameEnv() {
  FrameEnv curr_frame;
  auto obstacles = ContainerManager()->Get(PERCEPTION_OBSTACLES);
  curr_frame.set_timestamp(obstacles->timestamp());
}
```

(a) The timestamp propagates on a single data flow (Apollo-Pull-8503).

```
bool Fusion::GenerateMsg(Obstacles* obstacles) {
  common::Header * header;
  header->set_lidar_stamp(lidar_timestamp * 1e9);
  header->set_camera_stamp(camera_timestamp * 1e9);
  header->set_radar_stamp(radar_timestamp * 1e9);
  /* Processing */
```

(b) The timestamps of multiple data flows merge (Apollo-Pull-5459).

Figure 8: Code examples for timing propagation in data flows.

- *Timing Constraints:* The edges can be assigned timing constraints $C \in \mathbf{C}$. These constraints define the temporal properties that the edge is expected to meet within specified tolerance thresholds. The constraints are evaluated upon information flow. These temporal properties are defined by DFA's metrics, which are detailed in Section 4.2. Note that some of the temporal properties require analyzing the statistics of data flows into a vertex over time/iterations.

Timing Information Propagation. Timing tags can be propagated along edges E at runtime. There are two forms of timing tag propagation patterns (shown in Figure 7) that are common in a cyber-physical system:

- *Propagation of Timing Tag in Single Data Flow.* The timing information is propagated along a single data flow (Figure 7(a)), where edges inherit the timing tag from the predecessor edge, unless the data flow comes from a new sensor reading. Since the timing tag represents the time when the physical world observation is made, the data flow within cyber space does not change the tag. This is the most common case. In practice, developers programmatically add the timestamps to the variables according to data received from the predecessor tasks. Figure 8(a) shows an example from the Baidu Apollo self-driving car project (Apollo-Pull-8503). The prediction task inherits the timestamp of obstacles from the object detection task. The timestamp is then used to calculate the data age of the currently perceived environment upon which the prediction is based.

- *Merging Timing Tag from Multiple Data Flows.* This category (Figure 7(b)) involves merging multiple data flows at a vertex. This is typically required for tasks that fuse information from different sensors. In this case, the resulting memory SSA from the statement inherits the timestamps from its incoming edges, and maintains $t_{phy} = f_v(t^1_{phy}, \ldots, t^n_{phy})$, where f_v is the merging function for the vertex v. While the figure shows only two data flows, there could be more than two. Note that there is a one-size-fits-all solution in how timing tags can be merged, since it is effectively merging observations on the physical world from different time instances. One common approach is to keep the range of the time tags. The code snippet in Figure 8(b) depicts a fusion task in Apollo, added in Apollo-Pull-5459. Since this task fuses detection results from LiDAR, cameras, and radar, it also incorporates their timestamps to check the temporal alignment later.

4.2 Timing Constraints in TDFG

Based on the timing bug study, three essential temporal properties were formulated based on the cyber-physical control loop abstraction: freshness, consistency, and stability. In the following, we will show how they can be captured using TDFG in the form of *Timing Correctness*.

Freshness focuses on the difference between the time when a physical observation is made and the time when this observation is used by the control system. In cyber-physical systems, this difference often has to be bounded, as any latency increases the temporal gap between the cyber and the physical world, as previously discussed in Section 3. As a result, given an edge e with a maximum tolerable timing threshold θ_f, its freshness is calculated by:

$$C_f = \theta_f - (t_{-} - t_{\text{phy}}) \qquad (1)$$

where t_{-} is the current time.

Consistency concerns the time differences between the timing tags from different data flows into a vertex, which intuitively indicates the differences in the physical world status at different times. Generally, the smaller it is, the closer the time stamps are, and the more consistent the physical world observations should be. Consider n edges that have the same egress vertex: $T_{\text{def}} = \langle t^1_{\text{def}}, \ldots, t^n_{\text{def}} \rangle$. Their temporal consistency can be checked by:

$$C_c = \theta_c - \max_{i,j \le n}(t^i_{\text{phy}} - t^j_{\text{phy}}) \qquad (2)$$

where θ_c is the tolerable threshold (or range).

Stability captures differences in timing characteristics of data flows into/out of a vertex temporally. Many tasks in real-time systems are implemented as periodic workloads and thus some underlying algorithms/models are designed with the assumption of periodicity, which necessitates periodicity in data usage, such as input (e.g., sensor input) or output data (e.g., actuation command) [71]. For w edges belonging to a

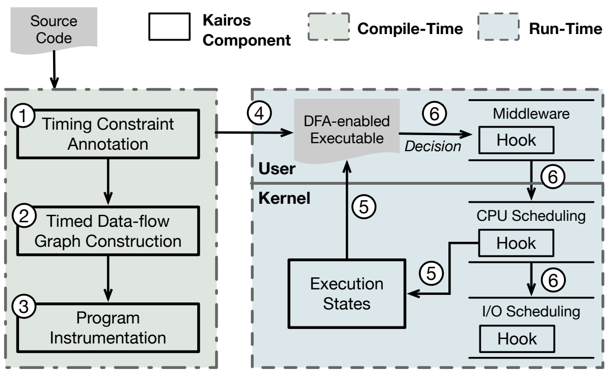
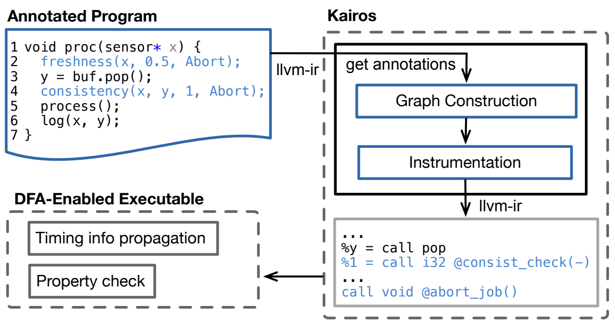


Figure 9: Workflow of Kairos.



```
void proc(sensor* x) {
  freshness(x, 0.5, Abort);
  y = buf.pop();
  consistency(x, y, 1, Abort);
  process();
  log(x, y);
}
```

```
...
%y = call pop
%1 = call i32 @consist_check(-)
...
call void @abort_job()
```

Figure 10: Pipeline of DFA-enabled application construction.

set of data flows from sequential loops via the same program point, they have $T = \langle t^1_{\text{phy}}, \dots, t^w_{\text{phy}} \rangle$. One typical way to check stability is by measuring jitters:

$$C_s = \theta_s - \max_{i,j \leq w-1} |D_i - D_j|, D_i = \Delta_i - I, \Delta_i = t^{i+1}_{\text{phy}} - t^i_{\text{phy}}. \quad (3)$$

where Δ represents the interval between two iterations and I is the expected interval. In practice, the form of stability can vary based on the design of the target systems, with alternatives potentially being variations of freshness.

An edge e is evaluated upon the program's execution reaching its egress vertex, and it is considered compliant with timing correctness if all its added metrics meet $C > 0$, namely $(C_f > 0 \land C_c > 0 \land C_s > 0)$.

5 Design and Implementation of Kairos

Kairos is a proof-of-concept realization of Data-flow Availability. There are two main components, the temporal policy definition using TDFG and the mitigation of policy violation. Kairos is composed of a compiler extension and a run-time system. Figure 9 outlines its key components and workflow. At compile-time, Kairos leverages program analysis and user annotation/automatic annotation ① from profiling to construct the TDFG of the target application ②; Utilizing the TDFG, it instruments code to perform timing information propagation ③; At run-time, tasks update timing information and evaluate timing correctness ④; Upon timing constraint violation, the task triggers a handler to execute the pre-defined policy ⑤; The scheduling decisions from handling policy are then shared with schedulers across different layers ⑥.

5.1 DFA-enabled Application

As shown in Figure 10, there are two main steps in the construction of DFA-enabled application, the construction of TDFG, which defines the temporal properties the application has to follow and the instrumentation of the application to enable detection and mitigation of the property violation.

TDFG Construction. TDFG captures the expected temporal properties of the developers. Upon extraction of value-flow graph [49, 86], the timing constraints in TDFG are expressed either manually via developer annotations or automatically via dynamic profiling.

Table 2: Kairos API

Function Name	Arguments				Description
	Targets	Tolerance	Window	Handling Policy	
freshness	var	threshold	-	abort	Checks the expected properties. If violated, triggers the handling policy function.
consistency	var, ...		-	prioritize	
stability	var		size	skip-next	

To facilitate manual annotation, three APIs are provided for annotating the source code to express timing constraints over the three key properties (freshness, consistency, and stability) that were previously discussed in Section 3. As shown in Table 2, the functions take four types of arguments: target variables, tolerance threshold, window size (for stability only), and handling policy. The threshold and window size parameters enable the check of timing correctness. The handling policy argument specifies the function to be invoked if timing correctness checks fail.

However, manual annotation often requires strong domain knowledge not only of the physical system but also of the computing stack, which may not always be available. To tackle this, Kairos also provides an option to extract the timing constraint using performance profiles from dynamic analysis. To do so, Kairos needs two key components: first, an oracle (criteria) to determine if the timing behavior of the software needs to be corrected or not; second, inputs to instrument the system such that all potential behaviors can be observed. For the oracle, Kairos borrows existing practice in CPS evaluation, where safety (often measured as control state deviation) is used as the metric. When physical safety (such as vehicles crashing into pedestrian or drones falling from the sky) is compromised by the violation of a specific temporal property, Kairos considers this temporal property to be essential and has to be monitored and checked at runtime. Inputs to the system, i.e., the physical scenarios, to test the system is an open challenge in CPS testing [72]. In Kairos, in addition to relying on the user to supply scenarios that might reveal temporal property violations, we use the performance interference tool [65] to probe the system with different potential timing impacts. To minimize the impact on the timing behavior of the software due to the profiling system, hardware performance monitors and debug functions are used. Among all flows that cause the same property violation, Kairos only adds the check on the first occurrence. It is important to note that dynamic profiling is much more effective in finding the acceptable range of the constraints rather than finding where to add the constraint (which has a much larger search space).

TDFG Embedding. Before constructing the TDFG, Kairos analyzes the source code to identify statements that receive sensor inputs, and automatically instrument them to extract timestamps t_{phy} from the sensing or input payloads. To begin the construction of TDFG, Kairos builds on top of the value-flow analysis in the SVF [86] tool with LLVM-IR [63], then uses a set of python scripts to add the timing constraints and annotation. Additionally, a set of LLVM compiler passes is also developed to instrument the necessary code for timing information propagation and checking. Kairos also leverages several heuristics to reduce the performance overhead. First, to avoid instrumenting every instruction for timing metadata propagation, Kairos automatically bypasses the timed data flows with the same time tag t_{phy} (sensor timestamps). Without loss of generality, a vertex is selected to be in the TDFG based on three criteria: *(i)* it is either a physical input or physical output vertex, *(ii)* it merges multiple data flows, or *(iii)* it is annotated with timing constraints as a vertex of interest.

5.2 Timing Constraint Violation Mitigation

Timing Constraint Violation Handler Policies. Mitigation of timing constraint violations often requires consideration of the physical components, and there is no one-size-fits-all solution. Drawing inspiration from our bug study, prior works in real-time computing [39, 46, 69, 91] and current industrial practices [29], Kairos offers three policies: *abort*, *prioritize*, and *skip-next*. More specifically, *abort* discards the task instances with timing constraint violations. *prioritize* switches the system into a different set of task models, often involving raising the priority of the task. Lastly, *skip-next* allows the delayed task to continue but skips its next instance to recover.

It is important to note that individually, these policies may give rise to further timing constraint violations in a cascading effect. For example, prioritizing a task that has missed its deadline might prevent other tasks from making progress, resulting in subsequent deadline misses. However, if correctly composed, these policies support existing adaptive real-time scheduling paradigms, e.g., elastic scheduling [46] and mixed-criticality scheduling [90].

Under *elastic scheduling*, task utilizations are decreased (typically by increasing the periods at which they are invoked)

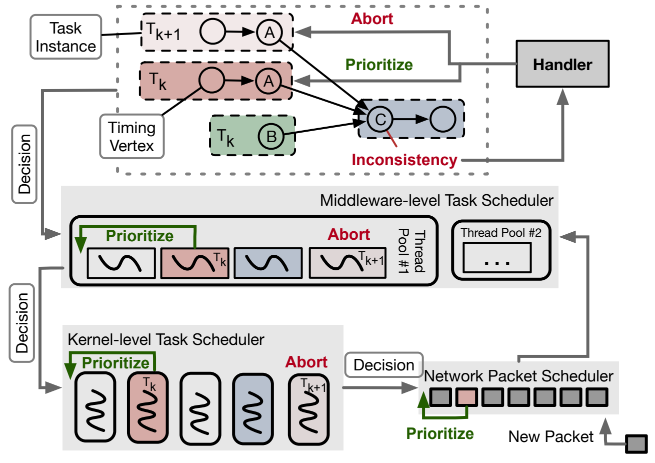


Figure 11: An illustrative case of Path across layers.

to avoid deadline misses. Though originally proposed in [46] as a mechanism to adapt to system overload, elastic scheduling has since evolved as a means by which systems can adapt to unexpectedly long task execution times [45] or interference from other tasks [85]. In response to a violation of timing constraints, Kairos can use the algorithm from [84] to quickly recompute task periods, then enforce this with multiple *prioritize* policies to change task priorities or `SCHED_DEADLINE` attributes accordingly.

In *mixed criticality systems*, non-critical task instances may be dropped in response to timing anomalies in critical tasks. Earliest-deadline first (EDF) scheduling with virtual deadlines (EDF-VD) is an optimal scheduling algorithm for non-clairvoyant mixed-criticality systems (i.e., those for which timing anomalies can't be predicted a priori, but are only identified when they occur) [39]. Under EDF-VD, each critical task is prioritized according to its virtual deadline, which is assigned as a constant parameter. When a critical task overruns its expected execution time, instances of non-critical tasks are dropped to maintain guarantees to critical tasks, and critical tasks are re-prioritized according to their absolute deadlines [39]. Kairos's handler supports this mode switch via a combination of its *abort* and *prioritize* policies (applied to the non-critical and critical tasks, respectively). While developing more sophisticated policies presents intriguing research opportunities, it is left for future exploration.

Implementation. Handlers can be implemented in individual layers or across multiple layers. In our prototype, we implement it as a modification to the kernel schedule where the mitigation mechanism is invoked before the built-in scheduler for proof-of-concept. For task abort, our prototype instruments code to enable early return. However, it is important to note that resource deallocation and inconsistent state removal often require sophisticated management [82]. The skip-next is demonstrated in the middleware by dropping the next task invocation message at ROS.

Cross Layer Scheduling Association. While the handling policies for timing constraint violation are relatively well understood under the real-time task problem formulation at the task level. Existing software ecosystems come with schedulable entities at different layers of architectural abstractions from I/O layer (such as network packet) and operating system (real-time process) to middleware (ROS component) to application (application-specific schedulers). This presents non-trivial system challenges in realizing consistency in the handler policy due to missing semantics across the abstraction layers.

This problem can be observed in 13 bugs [18, 38] from our earlier bug study. For example, the handling of a consistency violation often needs to adjust threads on sensor processing rather than the fusion process, necessitating the correlation of a subgraph of TDFG to the corresponding schedulable entities such that the timing constraint violation handler knows which one to intervene on [41]. Another example that commonly occurs in time-sensitive networking is the need to prioritize specific items in the network queue due to reprioritization of tasks [95], which can be part of the handling process.

To mitigate this, we propose to bridge the semantic gap by associating schedulable entities to data flows in TDFG. This not only allows the handler to know the schedulable entity to operate on, but also allows the other abstraction layers to respond to a handling mechanism much more effectively. To ensure the association is complete, Kairos draws inspiration from the Path concept from Scout system [73], where Path is used to track the components a packet travels through (e.g., network devices or protocol layers) on network appliance systems. In Kairos, upon dispatching, Path is updated to reflect the chain of schedulable entities that leads to execution of the application along a particular path, as shown in Figure 11.

Implementation. Our prototype modifies the data structure of native scheduling entities to store the Path to which they belong. This information is updated in a shared buffer accessible to four layers, i.e. user space, kernel, middleware, and network stack. The method of incrementing Path varies: in the kernel and network stack, it occurs where new tasks or packets are created; while in ROS middleware, it happens as threads are dispatched to execute callback functions.

6 Evaluation

This section seeks to answer the following questions: *(i)* What is the capability of DFA in addressing real-world timing bugs? – Section 6.1; *(ii)* What is the cost and efficacy of Kairos? – Section 6.2; *(iii)* How do DFA and Kairos improve performance/safety in abnormal timing situations? – Section 6.3.

Experimental Setup. The evaluations were performed on synthetic workloads and the workloads of three real-world autonomous systems: (1) Autoware.Auto [36] – an open-source full-stack autonomous driving project, which presents a high-end real-time autonomous system. (2) Jackal UGV – an unguided ground vehicle that represents mid-end autonomous system. It uses Google cartographer [47] for vehicle localization and ROS navigation [6] for path planning and control. (3) Turtlebot3 – a low-end indoor robot that relies on ROS navigation [6] for localization, planning, and control. Given that the software stacks of each system require distinct computing power, we used three different computing units that align with (or are similar to) the official recommendations to better emulate abnormal timing situations. The experimental hardware settings are listed in Table 3. Autoware and Jackal UGV were evaluated using hardware-in-the-loop simulations, while Turtlebot3 was also evaluated using a real robot.

Table 3: Evaluation Platforms

Platforms	Software Stack	Computing	Cores	RAM	Kernel
Autoware	Autoware.Auto [36, 61]	AMD 9 3900X RTX 3070 Ti	12	128GB	Linux 5.11
Jackal	Cartographer [47] & Navigation [5]	Intel Nuc 8	4	16GB	Linux 5.11
Turtlebot3	Navigation [5]	RPi 4B	4	4GB	RPi 5.15
Microbenchmark	ORB-SLAM3 [4]	Intel i9-12900K	12	128GB	Linux 5.11

Table 4: Root Cause Analysis of Bug Fix Capability

Category		**Description**	**Number**
Fixable	Non refactoring	Inadequate timing information/constraints/propagation	104
	Refactoring	Remove built-in conflict logics	5
		Adapt with software semantic	7
Unfixable	Out-of-Scope	Hardware-related timing faults	6
		Algorithm-related timing bugs	8
		Infrastructure bugs (e.g., scheduler crash)	41
	Limited	Concurrency bugs	12
		Performance issue	6

6.1 DFA in Solving Real-world Bugs

A key question to answer in evaluating the efficacy of Kairos is its ability to address the timing problems. To do so, our evaluation leverages the collection from the bug study and analyzes if Kairos can detect the timing problems (through temporal policy defined in TDFG) and mitigate the timing problems (using the cross-layer temporal policy violation handler). Due to the need to use physical system or emulation to exercise the system, most of the results from this evalua-

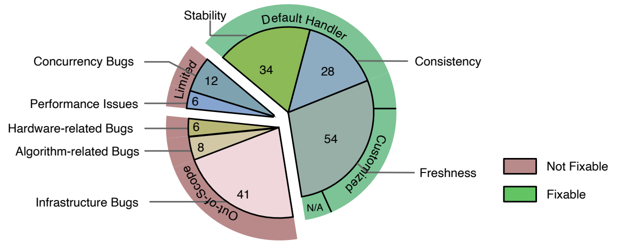


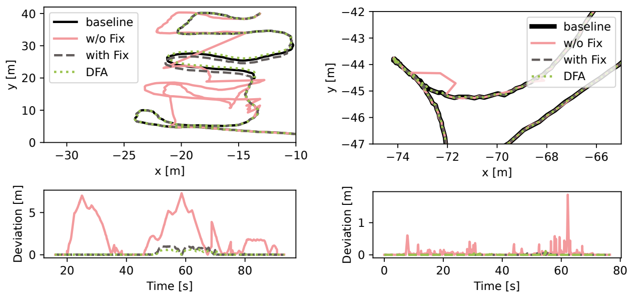
Figure 12: Statistics on bug fixability and root causes.

tion item are acquired through manual inspection by three cyber-physical system developers with 6, 10, and 17 years of experience respectively. A bug is considered fixable if all three developers agree that the mitigation can be expressed using primitives in Kairos. Furthermore, we've also conducted two case studies to demonstrate that Kairos can be used to detect and mitigate violations of the key temporal properties.

The results of the manual inspection are shown in Figure 12. From our earlier case study, there are 189 timing bugs. 116 of them can be detected by DFA, and 23 are not detectable, because they are caused by underlying infrastructure bugs, design flaws, or hardware issues (beyond the scope of Kairos). 76 bugs can be mitigated directly using default mitigation handling policies, while 35 bugs can only be mitigated using customized temporal violation handlers. 5 bugs cannot be mitigated even with customized handlers, because they require adaptation in the underlying design model or algorithms, necessitating a complete software redesign. Table 4 summarizes the reasons on all the bugs that Kairos cannot address. To further understand how Kairos can be used to address real-world bugs, we reproduce two bugs violating stability and consistency respectively, since freshness is often easier to handle.

Case-1: Abnormal timing of LiDAR Pointcloud in Cartographer. This case study demonstrates the effectiveness of Kairos in identifying and mitigating violations of stability timing constraint. Specifically, we evaluate Kairos on issue Cartographer-Issue-242 (code snippet shown in Figure 5) in Cartographer [56], a widely used localization package. According to the original issue report, the LiDAR pointcloud data, which is assumed to arrive at periodic intervals, sometimes arrives more closely than expected, violating the system's stability timing constraints. To reproduce the same impact of the issue, we modified the driver code to induce the same abnormal timing patterns, specifically manipulating the time between two pointclouds to be below 1 ms. Such timing patterns cause the vehicle to deviate from the baseline at most 10.3 m, as shown in Figure 13(a). To solve this issue, the patch in the codebase checks the timestamp of each point and removes abnormal ones with intervals of less than 1 ms. With this removal, the produced localization results are comparable to the baseline (deviation at 0.20 m). With Kairos, we specify stability timing constraint on *ScanMatch()* statement which consumes variable *msg* in task *HandleLaserScanMessage* that receives the point cloud with the annotation API *stability()*. The tolerance threshold argument in the API, which is set between 22 and 33 ms, is obtained through dynamic profiling of the ranges of intervals that do not incur adverse control outcomes. This process takes 18.3 minutes. We use *abort* as the default policy to mitigate the violation. The produced localization result aligns with the baseline at 0.19 m, which is comparable to the official patch as shown in Figure 13(a).

Case-2: Latency in Updating Location. This case study show-



(a) Jackal case study. (b) Autoware case study

Figure 13: Case studies on fixing timing bugs using DFA.

cases the effectiveness of Kairos in detecting and mitigating violation of consistency timing constraints. Specifically, we evaluated Kairos on the issue Autoware-Issue-458 in Autoware (code snippet shown in 14). According to the original issue report, the timestamps of the generated LiDAR data and odometer data used by the optimization procedure (line 31), which aims to perform the localization, should be within a threshold but are sometimes misaligned, violating the system consistency timing constraint. Since the mechanism to trigger the bug was not discussed in the original issue submission, we inject an intermittent CPU overload at 60 % level using stress-ng [62] on the cores running localization-related tasks to introduce inconsistency between LiDAR and odometer data. Such inconsistency causes the vehicle to produce a trajectory that deviates from the ground truth by 1.88 m, as indicated in Figure 13(b). To solve this issue, the patch in the codebase tags the LiDAR and odometry data with timestamps. It then compares these timestamps. If the difference between them is more than 1 second, the results are discarded. With this removal, the produced localization results align with the baseline at 0.12 m. To solve this problem with Kairos, we use annotation API *consistency()* (line 31 in Figure 14) to specify the consistency timing constraint on *transform_tree* and *msg_ptr*. The tolerance threshold argument in the API, which is set between 1.2 s, is obtained through dynamic profiling of *the difference between timestamps of two variable generations* that do not incur adverse control outcomes. This process takes 8.5 minutes. We use *prioritize* as the default policy to mitigate the timing constraint violation. The produced localization result aligns with the baseline at 0.091 m, which is comparable to the official patch.

A Programming Example. We use case-2 (Autoware-Issue-458) as an example to showcase how Kairos reduces the effort in programming timing constraints. Figure 14 shows simplified code snippets from Autoware's localization component. In this component, incoming LiDAR pointclouds, HD maps, and the transformation tree (extrapolated pose based on past information) are used jointly, so their timestamps should

```
  void observation_callback(typename ObservationMsgT::ConstSharedPtr msg_ptr){
    // Get the timestamp of new coming LiDAR message
    const auto observation_time = get_stamp(*msg_ptr);
    // Get global variable transformation tree
    const auto & transform_tree = xxx;
    // Get global variable map
    const auto & map = xxx;
    const auto &initial_guess = m_pose_initializer.guess(
                                        transform_tree, observation_time);
-   initial_guess.header.stamp = transform_tree.stamp;
    if (m_external_pose_available){
        initial_guess = m_external_pose;
-       initial_guess.header.stamp = get_stamp(*msg_ptr);}
    // Assign timestamp
-   const auto message_time = msg.header.stamp;
    // Validate timestamp (Map shouldn't be newer than a measurement)
-   if (message_time < map.timestamp()){
-       return ERROR;}
    // Assign timestamp
-   const auto guess_scan_diff = initial_guess.header.stamp - message_time;
-   const auto stamp_tol = m_config.guess_time_tolerance();
    // Validate timestamp (Backwards extrapolation is not supported)
-   if (initial_guess.header.stamp < message_time){
-       return ERROR;}
    // Validate timestamp
-   if (guess_scan_diff.count() > std::abs(stamp_tol.count())){
-       return ERROR;}
+  CONSISTENCY(guess, transform_tree, THRESHOLD, PRIORITIZE);
+  CONSISTENCY(map, msg_ptr, THRESHOLD, ABORT);
    NDT_optimizer.solve(initial_guess, msg, map);...}
```

Figure 14: Simplified code for temporal consistency checks in Autoware. '-' (red) represent built-in checks, while '+' (green) are checks via Kairos's API.

be checked as aligned. The standard checking mechanism (red lines marked by '-') requires developers to identify data provenance, label timestamps, and verify them before use. This often involves frequent jumps to other functions in different contexts, necessitating a deep understanding of data-flow relationships, which is both time-consuming and error-prone. In contrast, by using Kairos's APIs, users can omit all timestamp assignments and checks and simply add two statements before using the LiDAR point cloud and map (marked by two green lines with '+' in the figure).

Overall, Kairos eases programming with timing constraints in three ways. First, it removes the requirement of programmatically assigning timestamps to variables. Second, it avoids unnecessary or repeated timestamp checks. Third, it does not require developers to thoroughly understand the temporal relationships between different data flows in the source code.

6.2 Cost and Efficacy of Kairos

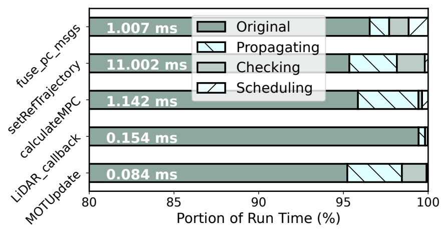
Runtime Overhead on Real-world Applications. The runtime overhead of Kairos stems from three aspects: timing information propagation, timing correctness checking, and the added logic in schedulers. We separately measured the overhead for each aspect on five representative tasks (or functions) per platform, averaging execution times over 100 runs. The results, shown in Figure 15, include original times and proportional increases.

The largest overheads observed in these tasks are *MOTUpdate* in Autoware (4.77%), *UpdateVelsPoses* in Jackal UGV (4.69%), and *getOdomPose* in Turtlebot3 (2.74%). Overall, the increased percentage of execution time is highly related to the number of edges in the task dependency graph (TDFG). Typically, tasks that involve more sensor inputs will introduce more edges. For example, the *MOTUpdate* task has a high overhead percentage because it is the multiple object tracking task in Autoware that fuses multiple pointcloud inputs. Furthermore, the tasks that maintain more historical timing states will also have a higher overhead. An example is the task *AddImuData* in Jackal UGV, which stores hundreds of inertial data frames in a queue, inducing a 4.21% overhead. Breaking it down, most of the overhead comes from propagating timing information along edges which can reach up to 4.39%. We found that Kairos's add-on logic on schedulers introduces negligible overhead, where the largest overhead is 0.84% from task *AddImuData*. Besides the individual execution times, we also measure the end-to-end latency from sensor input reading to actuation output. The overhead on end-to-end latencies for Autoware, Jackal UGV, and TurtleBot3 are 3.24%, 2.44%, and 2.75%, respectively.

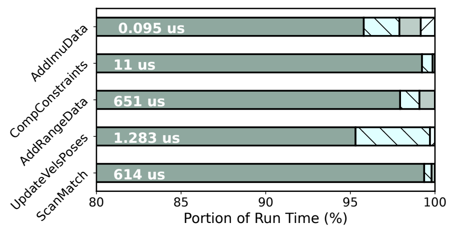
Scalability Analysis. The sensor reading rate, the number of edges in TDFG, and the number of timing tags in timing constraint checking affect the scalability of Kairos. Thus, we evaluate the scalability of Kairos by measuring the runtime overhead with respect to these three factors. We create synthesized workloads with varying scalability impact factors by modifying the original workload of ORB-SLAM3. Specifically, (1) to emulate varying sensor reading rates, we change the replay speed of the recorded sensor data from the original ORB-SLAM3 workload. (2) To adjust the number of edges, we duplicate tasks and annotate timing constraints on their shared data flows. (3) Since only *stability* performs timing constraint checks over multiple timestamps, we adjust its window size to assess the impacts of timing tag size.

Sensor Reading Rate. The sensor reading rate impacts runtime overhead. Figure 16(c) shows a linear increase in execution times for updating and checking timing tags as input frequency rises. The execution time for a single vertex increases from 37.537 μs at a frequency of 20 to 4087.95 μs at a frequency of 500Hz, primarily due to timing checks. A higher sensor reading rate also significantly increases lock wait times, increasing 100 times from 10 Hz to 500 Hz. Yet, CPU usage on a single core remains at just 0.23% even under high input frequency.

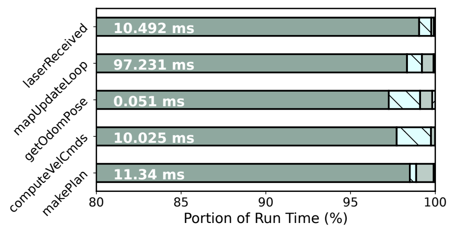
Number of Edges in Timing Propagation. We use the number of Paths created per second as a proxy for the size of TDFG. Figure 16(b) presents the runtime and memory overheads. The duplicated tasks are callback workers in the ROS middleware, thus increasing the number of tasks in the middleware but not significantly affecting the kernel scheduler. We observe that middleware scheduling time increases with the number of tasks. CPU usage on a single core peaks at 0.42% with



(a) Autoware

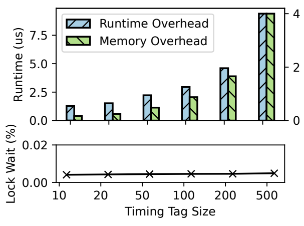


(b) Jackal UGV

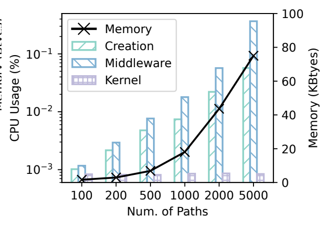


(c) Turtlebot3

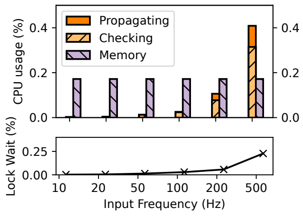
Figure 15: Run-time overhead breakdown. The execution time of the original task, the time spent logging timing information, online checking of timing correctness, and extra scheduling are shown.



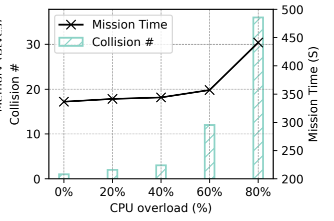
(a) Timing tag Size.



(b) Number of Paths.



(c) Input frequency.



(d) Different CPU overloads.

Figure 16: Scalability analysis (16(a), 16(b), and 16(c)) and control impact of CPU overload 16(d).

5000 Paths, while memory overhead reaches 75 KB, which is relatively low given that the target platform typically has over ten GB of RAM.

Number of Timing Tags. Figure 16(a) shows the runtime and memory overhead induced by a single edge with different numbers of timing tags used during timing constraint checks. We can observe that the checking time increases proportionally with the number of timing tags. With a tag size of 100, the average overhead is 2.945 μs of runtime and 896 bytes of memory; increasing to 500, it reaches 9.4 μs and 4096 bytes, respectively. Given that the number of edges typically remains below a few hundred, the total overhead is low. The figure also shows that as the number of timing tags increases, lock wait time slightly rises but remains below 0.05% of CPU usage on one core.

Invocation Latency in Delivering Execution Decision. In this experiment, we compared Kairos's efficacy to ROS [80],

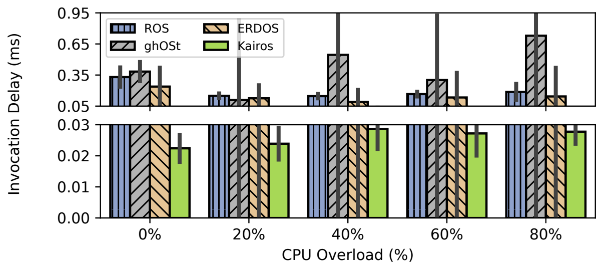


Figure 17: Invocation delays of handlers in different systems.

ghOSt [57] and ERDOS [55] in delivering scheduling decisions. ROS features the actionLib library [28], which supports preemptible tasks. ghOSt [57] is a userspace-informed kernel scheduling system that allows scheduling decisions to be made from userspace. EDROS is a robotic middleware that provides programming interfaces to deploy deadline miss handlers. We measure the delay from when the decision is made to the targeted task being executed. The experiments are conducted under different CPU overloads. We use *stress-ng* [62] tool to inject overloads, from 0% to 80%, then compare the increases and variations of latencies in Figure 17.

Under 80% overload, the response times for ROS, ghOSt, ERDOS, and Kairos are 0.186 ms, 0.72 ms, 0.14 ms, and 0.027 ms, respectively. Kairos has the fastest response time in fulfilling an execution decision, at least 2.16× faster than the others under high system overload. To be fair, these systems do not aim to achieve performance under high overload. Kairos has the cooperation in prioritizing the target task across scheduling layers. In contrast, ROS and ERDOS enforce scheduling decisions only at the middleware layers. Similarly, ghOSt enforces them solely through its own scheduler, which is a sub-scheduler with a lower priority than Linux's CFS scheduler. All four systems have stable invocation times while CPUs are idle, with variations of 0.095 ms, 0.094 ms, 0.18 ms, and 0.004 ms, respectively. However, we observe that the invocation latency and variations on ghOSt and ERDOS increase significantly (up to 0.81 ms under 80 % CPU overload) as the system overhead increases. This is because they

Table 5: DFA on Different Platforms

Platforms	Loc	# Inputs	# Tasks	# Vertices	Deviation (m)		# Collisions	
					Native	DFA	Native	DFA
Autoware	92 k	8	16	14	0.67	0.13	45	16
Jackal	68 k	4	6	6	0.27	0.09	12	4
Turtlebot3	34 k	3	4	3	0.87	0.21	35	13

are highly affected by Linux's underlying native scheduler (CFS). We conclude that Kairos takes faster and more stable countermeasures when mitigating a timing violation.

6.3 Effectiveness in Improving Safety

This section evaluates the capability of DFA model and Kairos in improving performance/safety in abnormal timing situations. We performed dynamic profiling on three platforms to identify which code regions required annotated timing constraints and to determine the expected temporal properties. Regarding handling policies, they also require understanding the task model and semantics of the target program. To mitigate the impact of this subjectivity, we employed an automatic strategy to apply three default policies for these timing constraints accordingly. Specifically, we model the target application's tasks as a directed graph. We adopted the *prioritize* policy for tasks on the critical path since aborting them will significantly increase end-to-end response time. For the tasks on the non-critical path, we apply the edges with *freshness* constraints and the *abort* policy for the remaining tasks. This is because violations of *consistency* and *stability* often lead to erroneous computation results, which should be prevented from propagating to downstream tasks. Table 5 shows the number of inputs and tasks on the three platforms as well as the number of edges annotated with timing constraints in TDFG. In generating the timing thresholds for those timing constraints, we observed an average variation of 8.82 ms.

We generated 100 trajectories in each scenario (Autoware in Parking Lot [34], Jackal UGV in Office [48] and Turtlebot3 in House [81] scenarios.)) for the vehicles to follow. During navigation, we injected CPU overload using the *stress-ng* tool [62] to emulate abnormal timings. We selected a 60% overload, as this condition typically triggers notable degradation in control performance. Figure 16(d) shows the number of collisions of Jackal UGV in 100 runs under overloads. We observed a significant increase in collisions at 60 %. Additionally, mission time increased with CPU overload because higher overload often triggered fail-safe, stopping the vehicle during the mission. At 80 % overload, vehicles typically halted, requiring manual intervention to continue.

Control Performance Improvements. The control performance is quantified by metrics (1) the distance vehicles deviate from the reference mission trajectories and (2) the number of collisions. The results are shown in Table 5. We observe that Kairos can considerably reduce control deviations across all three platforms. The lowest improvement of deviation is 2.97× on the Jackal UGV. As to collisions, Kairos reduces the collision by 64.4%, 83.3%, and 62.9% on Autoware, Jackal UGV, and Turtlebot3 respectively. Upon further investigation, the improvement is mainly due to proactively aborting false computational results to prevent the vehicle from outputting erroneous actuation commands. However, this approach will slow down the vehicle, which increases the mission time.

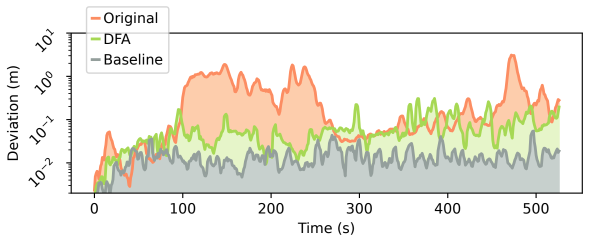


Figure 18: Comparison of control performance on Jackal UGV, with the deviation averaged over a window of 10.

Figure 18 plots the localization errors over time of one test run on Jackal UGV. We see that Kairos's abnormal timing detection and handling mechanism can significantly reduce the magnitude of intermittent computational errors, from ∼ 1m level to ∼ 10cm level. In this case, nearly one-third of the frames are dropped under high system overload, preventing the software from using data with abnormal timings and avoiding erroneous computational results. Aborting or skipping data also slows the robot's movement, reducing collision risks but increasing mission time by 56.8%. This policy may not suit hard real-time systems with strict deadline requirements. However, it effectively reduces adverse control outcomes in soft real-time systems.

7 Discussion and Limitation

Expressing Real-time Computing Constructs using DFA. DFA approaches timing assurance from a data-flow perspective, providing a more intuitive mechanism to express, detect, and mitigate timing constraint violations. However, to leverage this system coherently with existing constructions of real-time systems, DFA has to be able to express traditional real-time primitives. This will allow developers to build on top of the extensive advancements in real-time theory from the past several decades using the DFA-expressed real-time primitives. There are two categories of real-time computing primitives. The first category is execution time constraints, which specify the bounds of the execution period between two statements in the program. With Kairos, developers can use the statements *freshness(var, delay, abort)* and *freshness(var, delay, policy)* as firm and soft deadline specifications, where *var* is the task's output and *delay* is the deadline. The second category is synchronization primitives, which specify the order of shared data accesses among multiple real-time threads. This can be expressed by strictly ordering the timing of two data flows using statements such as *consistency(write_var, read_var, 0, policy)*, where *read_var* and *write_var* are SSA variables in read and written by different concurrent threads. This annotation will ensure that the write operation occurs before the read operation.

Manual Efforts. While Kairos provides tools for profiling through dynamic analysis, the search space for where and what temporal constraints to include is often prohibitively large. As a result, the automated tool may take a long time to identify the appropriate temporal policy. Developer guidance with some manual annotations can quickly narrow down this search space. Furthermore, once a policy is found, deploying it in a safety-critical system may require re-validation or even re-certification of the target system.

Multiple System Components in Violation Handling. Kairos requires seamless collaboration between the software instrumentation and multiple components across different scheduling layers for effective violation mitigation. This interdependency poses two limitations. First one is on reliability, as failures in one part can affect the entire system. Second one is maintainability, as migrating to different platforms may require substantial engineering efforts. However, the modularized design of detection and mitigation of temporal violations allows Kairos to integrate with other existing detection or mitigation techniques. Additionally, the infrastructure that bridges the semantic gap between different abstraction layers also reduces the engineering effort needed to build cross-layer timing mitigation.

Generality of DFA. While DFA is designed for cyber-physical systems, the concept of imposing temporal expectation on data flow generally applies to broader classes of computing, including conventional cyber-only environments such as data centers. For example, DFA's timing constraints on data usage can be adapted to systems with non-determinism to ensure logical correctness, such as the order of input events in distributed systems [74]. It is also possible to leverage DFA to track computation progress through the lens of data flow in distributed workloads.

8 Related Work

Timing Semantics in Programming Model. In data streaming systems, there have been efforts that incorporate timing information into the programming model to represent logical points, such as logic timestamps or watermarks [30,74,88,92]. This facilitates the coordination of computation among distributed nodes. Such extension of timing information on data-flow graphs inspired our design. However, these systems are designed for massive parallel data processing, rather than the cyber-physical timing alignment.

In real-time computing, several programming models have been proposed to react to timing violations [42,44,55,76,87]. In particular, Timed C [76] is a dialect of C that allows the specification of soft and firm real-time constraints. However, compared to these works, DFA introduces a design approach that focuses on the temporal policy on data flows, which builds on top of the cyber-physical control loop abstraction, allowing the detection and mitigation of cyber-physical state (data) misalignment.

Cross-layer Scheduling. There is a large body of work that focuses on cross-layer scheduling. However, existing works often target specific hardware [40,58,78,95], such as NICs. Furthermore, many target server platforms have abundant computation power, thus these solutions may not translate well to resource-constrained embedded systems. Notably, similar to Kairos, Syrup [60] offers programmable abstractions and interfaces for custom scheduling policies. However, it focuses on the rapid deployment of customized schedulers, rather than on enabling cross-layer scheduling actions.

Cross-layer scheduling has also been studied in the real-time community in the context of compositional scheduling [51, 83, 93]. However, deployment of these techniques often requires the target system to be rigorously modeled and deployed as real-time tasks, which may not always fit some of the existing software architectures for CPS.

9 Conclusion

In this paper, we presented data-flow availability, a concept that aims to define temporal policy for data-flow in real-time safety-critical cyber-physical systems. Through a bug study of 189 issues over 7 representative CPS software, three key temporal properties were extracted concerning the alignment of cyber states and physical states in time. To allow for the concrete expression of temporal expectation, we augment data-flow with timing constraints, captured by TDFG. To realize the concept in system, we design and develop Kairos that detects temporal violations by embedding the policy as checks in the application and mitigates them via a cross-layer scheduling infrastructure. Lastly, the system is evaluated on three CPS platforms for feasibility.

Acknowledgment

We express our gratitude to the anonymous reviewers and shepherd for their insightful feedback. We also thank Sanjoy Baruah and Ron Cytron for their valuable discussions. This work was partially supported by the NSF (CNS-2238635) and Intel.

References

[1] Google cartographer ros for the toyota hsr. https://google-cartographer-ros-for-the-toyota-hsr.readthedocs.io/en/latest/. Accessed: 2024-04-18.

[2] Moveit. https://github.com/ros-planning/moveit. Accessed: 2023-04-15.

[3] Orb-slam2. https://github.com/raulmur/ORB_SLAM2. Accessed: 2023-04-15.

[4] Orb-slam3 github. https://github.com/UZ-SLAMLab/ORB_SLAM3. Accessed: 2023-04-15.

[5] Ros navigation. https://github.com/ros-planning/navigation. Accessed: 2023-04-15.

[6] Ros navigation stack. https://github.com/ros-planning/navigation. Accessed: 2023-10-04.

[7] Ros rcl. https://github.com/ros2/rcl. Accessed: 2023-04-15.

[8] Cartographer #153 compute the common start time per trajectory. https://github.com/cartographer-project/cartographer/pull/153, 2016. Accessed: 2023-5-13.

[9] Cartographer #8 adds rate timer. https://github.com/cartographer-project/cartographer/pull/8, 2016. Accessed: 2023-05-22.

[10] Cartographer #242 improve 2d velocity estimation to be less fragile to poor data timing. https://github.com/cartographer-project/cartographer/issues/242, 2017. Accessed: 2023-05-24.

[11] Apollo #4492 timestamp correction in conti_radar. https://github.com/ApolloAuto/apollo/issues/4492, 2018. Accessed: 2024-05-20.

[12] Cartographer #1033 store timestamp of the latest range data in submap*d. https://github.com/cartographer-project/cartographer/pull/1033, 2018. Accessed: 2023-05-22.

[13] Cartographer #1275 add metrics: real time ratio and cpu time ratio. https://github.com/cartographer-project/cartographer/pull/1275, 2018. Accessed: 2023-05-13.

[14] Cartographer #1495 add serialization for timestampedtransform. https://github.com/cartographer-project/cartographer/pull/1495, 2019. Accessed: 2023-05-13.

[15] Moveit #1299 preempt trajectory execution if one controller aborts. https://github.com/ros-planning/moveit/issues/1299, 2019. Accessed: 2023-05-22.

[16] Ros 2 rclcpp #694 fixup time. https://github.com/ros2/rclcpp/pull/694, 2019. Accessed: 2023-11-22.

[17] Autoware.auto #1002 add predict/update functions receiving timestamp to kalman filter. https://gitlab.com/autowarefoundation/autoware.auto/AutowareAuto/-/issues/1002, 2020. Accessed: 2023-05-22.

[18] Autoware.auto #65 ros 2 and real-time. https://gitlab.com/autowarefoundation/autoware.auto/AutowareAuto/-/issues/65, 2020. Accessed: 2023-05-22.

[19] Moveit #232 fix race conditions when updating planningscene. https://github.com/ros-planning/moveit/pull/232, 2020. Accessed: 2023-05-22.

[20] Moveit #2395 fix pose tracking race condition. https://github.com/ros-planning/moveit/pull/2395, 2020. Accessed: 2023-05-22.

[21] Orb-slam2 #946 time stamps are not used in motion model part of tracking. https://github.com/raulmur/ORB_SLAM2/issues/946, 2020. Accessed: 2023-05-22.

[22] Ros 2 rclcpp #1121 lock-order-inversion (potential deadlock). https://github.com/ros2/rclcpp/issues/1121, 2020. Accessed: 2023-05-23.

[23] Autoware.auto #605 record replay_planner does not continuously update the trajectory. https://gitlab.com/autowarefoundation/autoware.auto/AutowareAuto/-/issues/605, 2021. Accessed: 2023-05-22.

[24] Autoware.auto #821 detect when nodes' incoming messages are skipped. https://gitlab.com/autowarefoundation/autoware.auto/AutowareAuto/-/issues/821, 2021. Accessed: 2023-05-22.

[25] Autoware.auto #980 updated ne raptor interface to send messages periodically. https://gitlab.com/autowarefoundation/autoware.auto/AutowareAuto/-/merge_requests/980/diffs, 2021. Accessed: 2023-05-22.

[26] Ros 2 rclcpp #1679 action client feedback callback does not reliably trigger. https://github.com/ros2/rclcpp/issues/1679, 2021. Accessed: 2023-05-22.

[27] Ros 2 rcl #967 problems with arguments in rcl_timer_exchange_period api. https://github.com/ros2/rcl/issues/967, 2022. Accessed: 2023-05-22.

[28] Ros actionlib. http://wiki.ros.org/actionlib. Accessed: 2022-10-10.

[29] Benny Akesson, Mitra Nasri, Geoffrey Nelissen, Sebastian Altmeyer, and Robert I Davis. An empirical survey-based study into industry practice in real-time systems. In *2020 IEEE Real-Time Systems Symposium (RTSS)*, pages 3–11. IEEE, 2020.

[30] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, et al. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. 2015.

[31] Amazon airprime. https://www.aboutamazon.com/news/transportation/amazon-prime-air-drone-delivery-mk30-photos. Accessed: 2023-11-30.

[32] Amazon astro. https://www.aboutamazon.com/news/devices/meet-astro-a-home-robot-unlike-any-other. Accessed: 2022-01-10.

[33] Björn Andersson, Sanjoy Baruah, and Jan Jonsson. Static-priority scheduling on multiprocessors. In *Proceedings 22nd IEEE Real-Time Systems Symposium (RTSS 2001)(Cat. No. 01PR1420)*, pages 193–202. IEEE, 2001.

[34] Autoware Foundation. Autonomous valet parking demonstration. https://autowarefoundation.gitlab.io/autoware.auto/AutowareAuto/avpdemo.html, 2020. Accessed: 2023-12-05.

[35] Autoware Foundation. Past, present, and the future of autoware. https://autoware.org/past-present-and-the-future-of-autoware/, 2023. Accessed: 2024-04-18.

[36] Autoware.auto project. https://autowarefoundation.gitlab.io/autoware.auto/AutowareAuto/. Accessed: 2022-08-15.

[37] Baidu. Apollo self-driving project. https://github.com/ApolloAuto/apollo. Accessed: 2022-08-15.

[38] Baidu. Apollo #9433: Cyberrt, coroutine to thread mapping, 2019. Accessed: 2023-05-22.

[39] S. Baruah, V. Bonifaci, G. DAngelo, H. Li, A. Marchetti-Spaccamela, S. van der Ster, and L. Stougie. The preemptive uniprocessor scheduling of mixed-criticality implicit-deadline sporadic task systems. In *2012 24th Euromicro Conference on Real-Time Systems*, pages 145–154, 2012.

[40] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. {IX}: a protected dataplane operating system for high throughput and low latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 49–65, 2014.

[41] Tobias Blass, Arne Hamann, Ralph Lange, Dirk Ziegenbein, and Björn 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 (RTAS)*, pages 264–277. IEEE, 2021.

[42] Gregory Bollella and James Gosling. The real-time specification for java. *Computer*, 33(6):47–54, 2000.

[43] Alan Burns. Mixed criticality systems-a review.

[44] Alan Burns and Andrew J Wellings. *Real-time systems and programming languages: Ada 95, real-time Java, and real-time POSIX*. Pearson Education, 2001.

[45] Giorgio Buttazzo and Luca Abeni. Adaptive workload management through elastic scheduling. *Real-Time Systems*, 23:7–24, 2002.

[46] Giorgio C. Buttazzo, Giuseppe Lipari, and Luca Abeni. Elastic Task Model for Adaptive Rate Control. In *IEEE Real-Time Systems Symposium*, 1998.

[47] Google cartographer. https://github.com/cartographer-project/cartographer. Accessed: 2022-11-21.

[48] Clearpath Robotics. Additional simulation worlds. Jackal Tutorials 0.6.0 Documentation, 2020. Accessed: 2023-12-05.

[49] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991.

[50] davetcoleman. Moveit #294 isvalidvelocitymove() for checking maximum velocity between two robot states. https://github.com/ros-planning/moveit/pull/294, 2016. Accessed: 2023-11-22.

[51] Arvind Easwaran, Madhukar Anand, and Insup Lee. Compositional analysis framework using edp resource models. In *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*, pages 129–138. IEEE, 2007.

[52] Facebook Engineering. Slam: Bringing art to life through technology. https://engineering.fb.com/2017/09/21/virtual-reality/slam-bringing-art-to-life-through-technology/, September 2017. Accessed: 2024-04-18.

[53] Autoware Foundation. Autoware.auto. Accessed: 2023-04-15.

[54] Gaschler. Cartographer #936 gracefully handle time-overlapping point clouds. https://github.com/cartographer-project/cartographer/pull/936, 2018. Accessed: 2023-11-22.

[55] Ionel Gog, Sukrit Kalra, Peter Schafhalter, Joseph E Gonzalez, and Ion Stoica. D3: a dynamic deadline-driven approach for building autonomous vehicles. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 453–471, 2022.

[56] Google. Cartographer. https://github.com/googlecartographer/cartographer. Accessed: 2023-04-15.

[57] Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakh, Paul Turner, and Christos Kozyrakis. ghost: Fast & flexible user-space delegation of linux scheduling. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 588–604, 2021.

[58] Stephen Ibanez, Alex Mallery, Serhat Arslan, Theo Jepsen, Muhammad Shahbaz, Changhoon Kim, and Nick McKeown. The nanopu: A nanosecond network stack for datacenters. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, pages 239–256, 2021.

[59] Jackal ugv. https://clearpathrobotics.com/jackal-small-unmanned-ground-vehicle/. Accessed: 2021-07-30.

[60] Kostis Kaffes, Jack Tigar Humphries, David Mazières, and Christos Kozyrakis. Syrup: User-defined scheduling across the stack. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 605–620, 2021.

[61] Shinpei Kato, Shota Tokunaga, Yuya Maruyama, Seiya Maeda, Manato Hirabayashi, Yuki Kitsukawa, Abraham Monrroy, Tomohito Ando, Yusuke Fujii, and Takuya Azumi. Autoware on board: Enabling autonomous vehicles with embedded systems. In *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)*, pages 287–296. IEEE, 2018.

[62] Colin King. stress-ng. https://wiki.ubuntu.com/Kernel/Reference/stress-ng. Accessed: May 20, 2022.

[63] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[64] Tanakorn Leesatapornwongsa, Jeffrey F Lukman, Shan Lu, and Haryadi S Gunawi. Taxdc: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. In *Proceedings of the twenty-first international conference on architectural support for programming languages and operating systems*, pages 517–530, 2016.

[65] Ao Li, Marion Sudvarg, Han Liu, Zhiyuan Yu, Chris Gill, and Ning Zhang. Polyrhythm: Adaptive tuning of a multi-channel attack template for timing interference. In *2022 IEEE Real-Time Systems Symposium (RTSS)*, pages 225–239. IEEE, 2022.

[66] Ao Li, Jinwen Wang, Sanjoy Baruah, Bruno Sinopoli, and Ning Zhang. An empirical study of performance interference: Timing violation patterns and impacts. In *2024 Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2024.

[67] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pages 329–339, 2008.

[68] Steven Macenski, Tully Foote, Brian Gerkey, Chris Lalancette, and William Woodall. Robot operating system 2: Design, architecture, and uses in the wild. *Science Robotics*, 7(66):eabm6074, 2022.

[69] Martina Maggio, Arne Hamann, Eckart Mayer-John, and Dirk Ziegenbein. Control-system stability under consecutive deadline misses constraints. In *32nd euromicro conference on real-time systems (ECRTS 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[70] Claire Maiza, Hamza Rihani, Juan M Rivas, Joël Goossens, Sebastian Altmeyer, and Robert I Davis. A survey of timing verification techniques for multi-core real-time systems. *ACM Computing Surveys (CSUR)*, 52(3):1–38, 2019.

[71] Pau Marti, Josep M Fuertes, Gerhard Fohler, and Krithi Ramamritham. Jitter compensation for real-time control systems. In *Proceedings 22nd IEEE Real-Time Systems Symposium (RTSS 2001)(Cat. No. 01PR1420)*, pages 39–48. IEEE, 2001.

[72] Till Menzel, Gerrit Bagschik, and Markus Maurer. Scenarios for development, test and validation of automated vehicles. In *2018 IEEE Intelligent Vehicles Symposium (IV)*, pages 1821–1827. IEEE, 2018.

[73] David Mosberger and Larry L Peterson. Making paths explicit in the scout operating system. In *OSDI*, volume 96, pages 153–167, 1996.

[74] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 439–455, 2013.

[75] Picknik robotics wins space force, nasa contracts. https://www.therobotreport.com/picknik-robotics-wins-space-force-nasa-contracts/. Accessed: 2024-04-18.

[76] Saranya Natarajan and David Broman. Timed c: An extension to the c programming language for real-time systems. In *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 227–239. IEEE, 2018.

[77] José Carlos Palencia and M González Harbour. Schedulability analysis for tasks with static and dynamic offsets. In *Proceedings 19th IEEE Real-Time Systems Symposium (Cat. No. 98CB36279)*, pages 26–37. IEEE, 1998.

[78] George Prekas, Marios Kogias, and Edouard Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 325–341, 2017.

[79] Cartographer Project. Fix division by velocity. https://github.com/cartographer-project/cartographer/commit/b4b83405ce4009ea0c1ac22c7ab9edeeb9d48a42, 2017. Commit b4b834.

[80] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, Andrew Y Ng, et al. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, Japan, 2009.

[81] ROBOTIS. Turtlebot3 simulation. https://emanual.robotis.com/docs/en/platform/turtlebot3/simulation/#gazebo-simulation. Accessed: 2024-05-20.

[82] Utsav Sethi, Haochen Pan, Shan Lu, Madanlal Musuvathi, and Suman Nath. Cancellation in systems: An empirical study of task cancellation patterns and failures. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 127–141, 2022.

[83] Insik Shin and Insup Lee. Compositional real-time scheduling framework. In *25th IEEE International Real-Time Systems Symposium*, pages 57–67. IEEE, 2004.

[84] Marion Sudvarg, Chris Gill, and Sanjoy Baruah. Linear-time admission control for elastic scheduling. *Real-Time Systems*, 57(4):485–490, 10 2021.

[85] Marion Sudvarg, Ao Li, Daisy Wang, Sanjoy Baruah, Jeremy Buhler, Chris Gill, Ning Zhang, and Pontus Ekberg. Elastic Scheduling for Harmonic Task Systems. In *2024 Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2024.

[86] Yulei Sui and Jingling Xue. Svf: interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th international conference on compiler construction*, pages 265–266, 2016.

[87] Milijana Surbatovich, Limin Jia, and Brandon Lucia. Automatically enforcing fresh and consistent inputs in intermittent systems. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 851–866, 2021.

[88] Peter A. Tucker, David Maier, Tim Sheard, and Leonidas Fegaras. Exploiting punctuation semantics in continuous data streams. *IEEE Transactions on Knowledge and Data Engineering*, 15(3):555–568, 2003.

[89] Turtlebot3. https://emanual.robotis.com/docs/en/platform/turtlebot3/overview/. Accessed: 2022-09-10.

[90] Steve Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *28th IEEE international real-time systems symposium (RTSS 2007)*, pages 239–243. IEEE, 2007.

[91] Nils Vreman, Anton Cervin, and Martina Maggio. Stability and performance analysis of control systems subject to bursts of deadline misses. In *33rd Euromicro Conference on Real-Time Systems (ECRTS 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

[92] Guozhang Wang, Lei Chen, Ayusman Dikshit, Jason Gustafson, Boyang Chen, Matthias J Sax, John Roesler, Sophie Blee-Goldman, Bruno Cadonna, Apurva Mehta,

et al. Consistency and completeness: Rethinking distributed stream processing in apache kafka. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2602–2613, 2021.

[93] Jinwen Wang, Ao Li, Haoran Li, Chenyang Lu, and Ning Zhang. Rt-tee: Real-time system availability for cyber-physical systems using arm trustzone. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 352–369. IEEE, 2022.

[94] Waymo driveless service in phoenix. https://blog.waymo.com/2020/10/waymo-is-opening-its-fully-driverless.html. Accessed: 2022-01-10.

[95] Chuanyu Xue, Tianyu Zhang, Yuanbin Zhou, Mark Nixon, Andrew Loveless, and Song Han. Real-time scheduling for 802.1qbv time-sensitive networking (tsn): A systematic review and experimental study. In *2024 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2022.