Probabilistically Guaranteeing End-to-End Latencies in Autonomous Vehicle Computing Systems

Hyoeun Lee, Youngjoon Choi, *Graduate Student Member, IEEE*, Taeho Han, *Graduate Student Member, IEEE*, and Kanghee Kim, *Member, IEEE*

Abstract—Good responsiveness of autonomous vehicle computing systems is crucial to safety and performance of the vehicles. For example, an autonomous vehicle (AV) may cause an accident if the end-to-end latency from sensing a pedestrian to emergency stop is too high. However, the AV software stacks are too complex to probabilistically analye the end-to-end latency on a multi-core system. They consist of a graph of tasks with different periods, and have a large variability in the task execution times, which may lead to the maximum core utilization $U^{\max}$ greater than 1.0 on some cores. This paper proposes a novel stochastic analysis of the end-to-end latency over the AV stacks that allows $U^{\max}$ to exceed 1.0 on each core. The proposed analysis models the entire stack as a graph of task graphs under a multi-core partitioned scheduling and provides a probabilistic guarantee that the analyzed latency distribution upper-bounds the one observed from a real system under the assumption of independent task execution times. Using the Autoware stack with inter-task dependent execution times, it is shown that our analysis, combined with a task grouping to mitigate the inter-task correlations, can give a latency distribution for each task path that almost upper-bounds the observed one.

Index Terms—Autonomous vehicles, autoware, end-to-end latency, probabilistic guarantee, response time analysis

1 Introduction

Recently, autonomous vehicles (AVs) are being increasingly deployed in the market in various forms such as ADAS (Advanced Driver Assistance Systems)-equipped cars, self-driving cars, and autonomous racecars. These AVs are equipped with a computing system that reads data from various sensors, performs computations for self-localization, obstacle detection and tracking, path planning, trajectory planning and following, and finally outputs vehicle control signals. In such a computing system, end-to-end latency from sensing to vehicle control is significant for safety and performance of the vehicle, since it directly relates to time to collision with other cars or pedestrians in front of it. For example, when a vehicle drives at 72 km/h (= 20 m/s), an end-to-end latency of 100 ms from sensing a pedestrain to controlling the vehicle for emergency stop allows the vehicle to continue toward the pedestrian by 2 meters without proper control, thus may cause an accident if the remaining distance is not enough. If we know an accurate value of the end-to-end latency, we can calculate an accurate safety margin of the control parameters of the AV stack and synchronize the tasks in the stack in terms of timing. For example, the control signal generated with object state estimations made 100 ms before, becomes inaccurate at the time of actuation. Thus, given the latency value of 100 ms, the AV stack can be instrumented so that every state estimation may be projected into the 100 ms future.

However, analyses of the end-to-end latency for real AVs are hardly found in the literature. The first reason is that access to the full software stack running in the AVs is limited due to the closed source strategy of AV companies. Recently, the accessibility is getting gradually better with emerging open-source stacks such as Autoware [1] and Apollo [2]. The second reason is that the AV stacks are too complex to analyze due to a graph structure of tasks communicating with messages. There are a few studies that try to analyze such a structure in the context of safety critical systems. In [3], a probabilistic analysis is presented to analyze a single DAG (Directed Acyclic Graph) of tasks under EDF scheduling on a uniprocessor system. In [4], another analysis is proposed to address multiple DAGs in a drone system under fixed-priority scheduling on a multi-core system. The above analyses give safe probabilistic response times for the DAG(s), but do not deal with a graph of task graphs with different periods, thus cannot provide a probabilistic guarantee on the end-to-end latencies. In [5], a deterministic analysis is proposed that only gives the worst-case end-to-end latency in a robotic system, thus limited to the case where $U^{\max} \leq 1.0$ for every core. Recently, in [6], a probabilistic worst-case execution time (pWCET) of each task in the Apollo stack has been analyzed by using the

• *Hyoeun Lee, Youngjoon Choi, and Taeho Han are with the Department of Intelligent Systems, Soongsil University, Seoul 06978, Korea. E-mail: {dlgydms412, venbo12, sethut1224}@gmail.com.*
• *Kanghee Kim is with the School of Artificial Intelligence Convergence, Soongsil University, Seoul 06978, Korea. E-mail: khkim@ssu.ac.kr.*

Manuscript received 1 May 2021; revised 20 January 2022; accepted 12 February 2022. Date of publication 16 February 2022; date of current version 11 November 2022.
This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) under Grants NRF-2021R1A2C2012837 and NRF-2021R1A4A1032252.
(Corresponding author: Kanghee Kim.)
Recommended for acceptance by M. Annavaram.
Digital Object Identifier no. 10.1109/TC.2022.3152105

Extreme Value Theory. This theory [7] allows us to derive a safe pWCET distribution that upper-bounds the real distribution, while considering architectural complexities such as caches and memory buses. The analysis of pWCET, however, does not deal with how to analyze the end-to-end latencies for a graph of tasks, which possibly include idle CPU times due to the underlying task model.

This paper presents a novel stochastic analysis of the end-to-end latency over AV stacks that allows the maximum core utilization $U^{\max}$ to exceed 1.0 for each core. The proposed analysis provides a probabilistic guarantee that an observed latency distribution from a real system will be upper-bounded by the analyzed distribution under the assumption of independent task execution times. It helps reduce the implementation cost of the computing system, which is necessary for the cost-competitive automotive industry. This is possible by allowing tasks to utilize CPU cores higher than the existing analyses [3], [5], which require to over-provision the cores against the worst-case demand of the tasks. To make the system analyzable, we model the entire stack as a graph of disjoint subgraphs of tasks, and build an isolated environment for each subgraph so that one subgraph cannot affect the timing behavior of the others. This is enabled by the multi-core partitioned scheduling and non-blocking inter-graph communication. Then, each subgraph can be independently analyzed in terms of the steady-state behavior, and the end-to-end latency from a subgraph to another can be analyzed by combining the analysis results of the individual subgraphs. Using an AV stack called Autoware with inter-task dependent execution times, we show that our analysis, combined with a task grouping to mitigate the inter-task correlations, gives a latency distribution for each task path that almost upper-bounds the observed one. The contribution of the paper can be summarized as follows:

- To the best of our knowledge, it is the first attempt to probabilistically analyze end-to-end latencies from sensing to control for a real AV stack.
- The proposed analysis is safe in that the observed latency distribution from a real system will be upper-bounded by the one obtained by the analysis.
- It provides a solid base on evaluating the safety and performance of AVs, which is necessary in issuing AV licenses.

The organization of the paper is as follows. In Section 2, we describe the system model and the problem formulation, and in Section 3, explain the proposed analysis in detail. In Section 4, the experimental results are given, and in Section 5, the related work is reviewed. Finally, in Section 6, we conclude the paper with future research directions.

2 Preliminaries

An AV is a cyber-physical system that is equipped with various sensors to perceive the surrounding environment and monitor the status of the vehicle and with several actuators to control the maneuver. The sensors include LiDARs, Radars, cameras, GNSS, IMU and odometer, which are used for self-localization (LiDARs and GNSS), object detection (LiDARs, Radars, and cameras), and dead reckoning (IMU and odometer). Depending on the state of the art, each type of sensors generates sampled data at its own period, for example, LiDARs at 50−200 ms [8], cameras at 33.3−100 ms, GNSS at 10−100 ms [9], and so on. The actuators include motors to manipulate the steering wheel and throttle/brake pedals, which are controlled by the automotive electronic (AE) system with a common control period, e.g., 10 ms. Thus, the AE system with the actuators can collectively be treated as a single virtual actuator that accepts a tuple of control signals, e.g., (target acceleration a, target steering angle δ). Fig. 1 shows a high-level view of a general AV stack. It consists of five core functions: self-localization, obstacle detection and tracking, path planning, trajectory planning and following, and vehicle control.

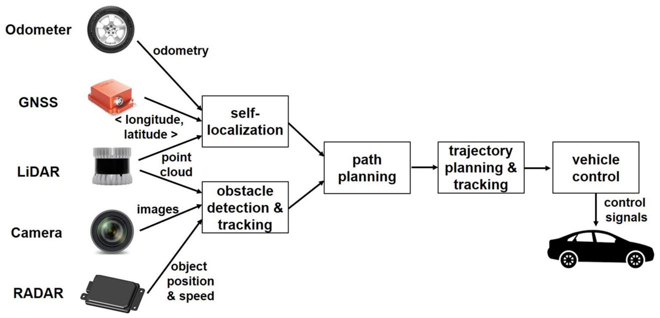


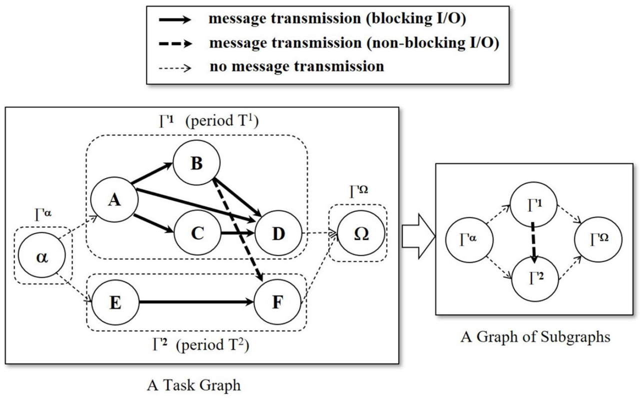
Fig. 1. An overview of the AV software stack.

We assume that the above AV stack is implemented in a graph of tasks communicating with messages. The task graph consists of source tasks that read data from sensors, sink tasks that generate control signals, and intermediate tasks between both. This task model is widely used in ROS (Robot Operating System)-based open-source AV stacks [1], [2] and AutoSAR [10]. In ROS, an execution unit called node can be modeled as a task as long as it explicitly communicates with ROS messages, regardless of whether it is a process or a thread. It is also assumed that the AV stack runs on a single-board computer with multi-core processors [11], [12].

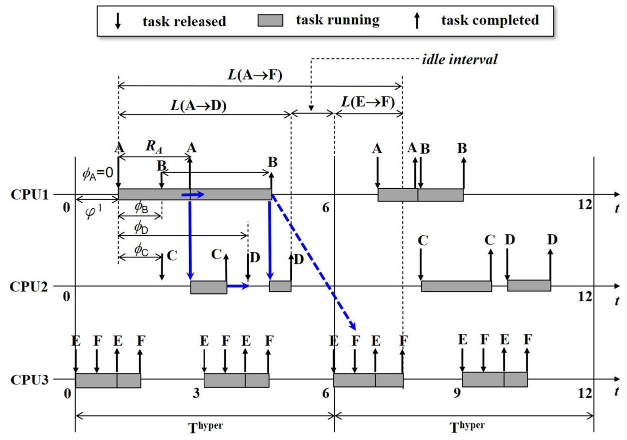
2.1 Task Model

For the proposed analysis, we model the task graph of the AV stack as a directed acyclic graph (DAG), as shown in Fig. 2a. If a task τ_i directly sends a message to τ_j, we denote it by $\tau_i \rightarrow \tau_j$ and call τ_i an immediate predecessor of τ_j and τ_j an immediate successor of τ_i. For ease of explanation, to the graph we add two dummy tasks, τ_α and τ_Ω: the former points to all the source tasks (i.e., τ_A and τ_E) while the latter is pointed to by all the sink tasks (i.e., τ_D and τ_F). If there is a task that feeds a message back into a predecessor task in the graph, which creates a cycle in the original graph, we remove the cycle by creating a virtual source task that periodically produces the feedback to the predecessor task.

Then the modeled graph can be decomposed into several disjoint subgraphs, each consisting of tasks periodically released with the same period. As mentioned above, in the AV stack, each task has a period T that is defined in association with a sensor, an actuator or for its own need. For the example of the task graph in Fig. 2a, it is divided into two subgraphs, i.e., Γ^1 with period T^1 and Γ^2 with T^2, thus understood as a graph of the two subgraphs plus the two



(a) A graph of tasks and the abstracted graph of subgraphs



(b) A possible CPU schedule

Fig. 2. An example of the task graph.

dummies, Γ^α and Γ^Ω, shown at the right side of Fig. 2a. This abstracted graph is also a DAG. If Γ^k directly sends a message to Γ^l, it is denoted by $\Gamma^k \rightarrow \Gamma^l$.

Throughout the paper, we consider such a DAG that, for any path $\Gamma^1 \rightarrow \Gamma^2 \rightarrow \cdots \rightarrow \Gamma^L$ found in the abstracted graph, the involved subgraphs have a non-increasing order of periods, i.e., $T^1 \geq T^2 \geq \cdots \geq T^L$. This condition is required by the proposed analysis, since it assumes no message dropping in inter-graph communication. We make this assumption because the end-to-end latency cannot be defined for dropped messages. If the condition does not hold for $\Gamma^l \rightarrow \Gamma^{l+1}$, i.e., $T^l < T^{l+1}$, this means that Γ^l generates messages faster than Γ^{l+1} consumes. Thus, every instance of Γ^{l+1} should consume multiple input messages from Γ^l at once for the computation of a single output message. This may be accompanied by dropping some input messages if they are redundant or outdated from the viewpoint of Γ^{l+1}. For example, in the AV stacks, for state estimations required by self-localization and object tracking, only the latest input message is used in general. According to our observation, the above condition can easily be accepted in the AV stack for the following reasons. First, as mentioned above, the period of sensors is greater than that of actuators in general. Thus, there exists such tendency that each subgraph gives an increasingly shorter period as a message flows into the sequence of subgraphs. Second, this tendency becomes obvious in sensor fusion. For example, in our customized Autoware, later shown in Fig. 6a, the RVF task fuses at every 50 ms a Lidar-originated message with a period of 100 ms and a camera-originated message with a period of 50 ms. For any path of subgraphs that violates the condition, the subgraphs can be rewritten to satisfy it simply by adjusting their periods.

In our model, every task does not become eligible for execution as soon as it releases. A source task becomes runnable as soon as it releases. An intermediate or a sink task (e.g., τ_F) becomes runnable immediately after it has received messages from all immediate predecessors in the same subgraph (i.e., τ_E), regardless of whether or not it receives a message from a predecessor in a different subgraph (i.e., τ_B). In other words, τ_F performs *blocking I/O* on the intra-graph communication of $\tau_E \rightarrow \tau_F$, which is represented by a bold edge in Fig. 2a, but performs *non-blocking I/O* on the inter-graph communication of $\tau_B \rightarrow \tau_F$, which is represented by a dashed edge. The rationale behind this assumption is that it is not possible to allow blocking I/O between two tasks (or subgraphs) with different periods in order to keep the period of each task (or subgraph). Regarding when a task sends a message to the successors, we assume that the task does so at the time when it finishes execution. Moreover, it is also assumed that each task reads messages as soon as it starts execution, not in the middle, regardless of the I/O mode. Note that both assumptions are safe in that it always makes our analysis pessimistic.

Regarding the release times of subgraphs, we define φ^k as a relative offset of the release time of (the first released task within) the subgraph Γ^k to a reference time origin ($0 \leq \varphi^k < T^k$). It is called the *graph phase* of Γ^k. Then the relative offset ϕ_i of each task τ_i within a subgraph, called *task phase*, is defined as to φ^k. That is, the absolute task phase Φ_i^k of τ_i^k in a subgraph Γ^k is equal to $\varphi^k + \phi_i^k$. Thus, the jth instance $\tau_{i,j}^k$ releases at $\varphi^k + \phi_i^k + (j-1) \times T^k$.

Each task τ_i^k has a probabilistic execution time C_i^k, which is regarded as a discrete random variable. The probability distribution of C_i^k, i.e., the execution time distribution (ETD), can be obtained by profiling the task with an extremely large number of incoming messages generated from real representative sensor workloads, or by probabilistically reasoning from a relatively small number of incoming messages with the Extreme Value Theory [6] and/or the Extended Path Coverage [7]. Thus, the response time $R_{i,j}^k$ of $\tau_{i,j}^k$ is also a random variable, defined as $\eta_{i,j}^k - \lambda_{i,j}^k$ where $\eta_{i,j}^k$ and $\lambda_{i,j}^k$ are the completion and release times of $\tau_{i,j}^k$, respectively. It may be affected by late completion of a predecessor on a different CPU core or the immediate preceding task on the same core. Thus, the response time distribution (RTD) of τ_i^k is defined with interferences from those tasks. For the stochastic analysis to be possible, we assume that the execution time of a task is identically distributed and independent of those of other tasks in the same subgraph and those of tasks from other subgraphs.

In our task model, we allow such tasks that use both CPU and GPU. In this case, the GPU codes of such a task may be executed asynchronously to the CPU codes of the same task. However, we can safely assume that parallel executions of CPU and GPU codes are all synchronized before the task finishes. The reason is that every task (except for the sink task) ends with CPU codes to send a message to a

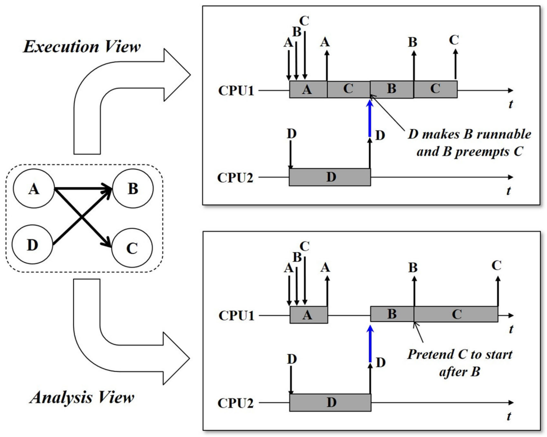

Execution View
Analysis View
A
B
C
D
CPU1
CPU2
t
D makes B runnable and B preempts C
Pretend C to start after B

Fig. 3. A preemption scenario of a task graph.

successor. This means that all asynchronous executions of GPU codes should be finished before the last CPU codes generate an output message. These CPU codes serve as a synchronization point between CPU cores and the GPU. Note that the entire execution time of a task is simply defined as the difference between the start time and the finish time of the task, regardless of whether there are contentions for the GPU between the tasks or not. It is out of the scope of the paper to accurately model the pure GPU execution time.

The objective of our analysis is to compute a safe end-to-end latency distribution (LD) from sensing to control in the AV stack. For example, Fig. 2b shows an end-to-end latency $L(\tau_A \rightarrow \tau_F)$ under a possible CPU schedule for the task graph in Fig. 2a. By $L(\tau_{src} \rightarrow \tau_{sink})$ or $L^{\tau_{src}}_{\tau_{sink}}$, we denote the end-to-end latency from the release of τ_{src} to the completion of τ_{sink}. Note that the end-to-end latency is not merely the sum of the execution times of all the tasks involved in the path. As shown in Fig. 2b, for $L(\tau_A \rightarrow \tau_F)$, it includes an idle time between the completion of τ_D and the release of τ_E. Thus, in our analysis, the effect of task or subgraph phasing will also be modeled. Also, note that there may be several paths of interest in a DAG.

2.2 Scheduling Model

We use fully partitioned EDF (Earliest Deadline First) scheduling. Each subgraph Γ^k is assigned a partitioned set P^k of CPU cores, thus given K subgraphs, we need K cores at minimum. In each partition P^k, every task τ_i^k is statically assigned to a certain core, and no migration is allowed between the cores within P^k nor across P^k's. Then, on every core, we use EDF scheduling assuming that the relative deadline D_i^k of τ_i^k is equal to T^k. That is, the absolute deadline $d_{i,j}^k$ of $\tau_{i,j}^k$ is $\lambda_{i,j}^k + T^k$, and a smaller value means a higher priority. If two tasks have the same absolute deadline, we break the tie by always selecting the one with the smaller task index of i.

To understand how EDF scheduling works for a DAG, let us consider an example shown in Fig. 3. In this example, priority(τ_A) $>$ priority(τ_B) $>$ priority(τ_C) because the release order of τ_A, τ_B, and τ_C immediately becomes the absolute deadline order because they have the same period. According to the graph, when τ_A finishes execution, τ_C immediately becomes runnable, but τ_B does not if τ_D is not completed. In this case, τ_C will take CPU 1 and start execution. However, if τ_D finishes before τ_C finishes, τ_B will immediately become runnable, thus preempt τ_C and take back CPU 1. Then τ_C will be resumed after τ_B finishes. In our analysis, we will safely model such preemption scenarios by assuming that τ_C always starts execution after τ_B finishes, as shown in the bottom half of Fig. 3. From the analysis perspective, this makes the EDF scheduling degenerate to FCFS (First Come First Served) with no preemptions. Although this assumption seems to be a pessimistic artifact to the analysis, we can reduce the negative effect of the artifact simply by making τ_D released much earlier in the task schedule so that τ_D may not cause any delay to τ_B.

2.3 Problem Statement

From the above task and scheduling models, we can formulate our problem as follows. First, a task τ_i^k in subgraph Γ^k is defined as (ϕ_i^k, a_i^k, C_i^k) where ϕ_i^k is the task phase of τ_i^k, a_i^k is the ID of a CPU core assigned to τ_i^k, and C_i^k is a probabilistic execution time of τ_i^k on the assigned core. Then, a subgraph Γ^k is defined as $(T^k, \varphi^k, V^k, E^k)$ where T^k is the period of Γ^k, φ^k is the graph phase, V^k is the set of tasks serving as the vertices, i.e., $\{\tau_1^k, \ldots, \tau_{n_k}^k\}$, and E^k is the set of directed edges between the tasks. Then, the entire graph $\boldsymbol{\Gamma}$ is defined as $\{\Gamma^1, \ldots, \Gamma^K\}$. Thus, our goal is to obtain an end-to-end latency distribution for each path S of tasks that upper-bounds the one observed in a real system. We denote by $S_j^{<h>}$ the jth instance of path S released within the hth hyperperiod, and define $L(S^{<h>})$ as a random variable describing the end-to-end latency of all S's instances released within the hth hyperperiod. A hyperperiod is a duration whose length is equal to the least common multiple of all periods, i.e., T^1, T^2, $\cdots$, T^K. Since we consider even the cases where the maximum core utilization may exceed 1.0 for each core, we try to find a stationary distribution of $L(S^{<h>})$, which is obtained when the system reaches a steady state. This distribution is defined as a limiting distribution obtained by approaching $h \rightarrow \infty$.

3 Proposed Analysis

The proposed analysis is divided into two stages: intra-graph analysis and inter-graph analysis. In the first stage, we compute the waiting time distribution (WTD) and the RTD of every task within each Γ_k, independently of other subgraphs. In the second stage, from the resulting task RTDs, we derive the end-to-end LD for each path S.

3.1 Intra-Graph Analysis

The intra-graph analysis consists of three steps: (1) graph transformation, (2) computation of task RTDs, and (3) computation of intra-graph end-to-end LDs.

Graph transformation: The intra-graph analysis is independently applied to each Γ^k obtained by decomposing the entire task graph. All tasks in Γ^k have the same period T^k and use blocking I/O on inter-task communication. For the intra-graph analysis, we first transform each Γ^k to such a graph with no task preemptions. This transformation is possible by serializing all the tasks in Γ^k as described in the following lemma. The resulting graph can be analyzed by extending the analysis proposed in [13]. While the previous analysis addresses only a pipeline of tasks, our extended analysis deals with an arbitrary DAG.

Lemma 1. *(Serialization) For a graph Γ of tasks with the same period, let L_m be a list of tasks assigned on core m. It is assumed that on L_m there exists no such edge that $\tau_i \rightarrow \tau_j$ with $\lambda_j < \lambda_i$ since it is meaningless that a task τ_j releases earlier than the predecessor τ_i. Then, according to the increasing order of the release times, we number the tasks so that $\lambda_i \leq \lambda_j$ if $i < j$. If any $\tau_i \rightarrow \tau_j$ with $\lambda_i = \lambda_j$, i should be smaller than j. In this ordered list, if there exist any two consecutive tasks, τ_i and τ_{i+1}, with no edge between them, we add an edge such that $\tau_i \rightarrow \tau_{i+1}$. Let Γ^{serial} be the resulting graph from augmenting the ordered task list on every core. Then, the response time of each task in Γ^{serial} is always greater than or equal to the one obtained in Γ.*

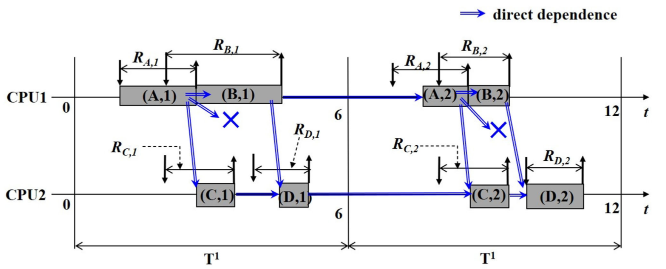
Proof. For any graph Γ assumed in the lemma, the above augmentation leads to a graph Γ^{serial} that has no cycles and no preemptions on every core m. First, Γ^{serial} has no cycles because the augmentation rule does not create any such edge that $\tau_j \rightarrow \tau_i$ when $i < j$. Second, Γ^{serial} has no preemptions because the resulting list on each core guarantees that at most one task is runnable on the core at any time. Recall that preemption is only possible when more than one task is runnable at the same time. This augmentation always makes R_j^{serial} equal to or worse than R_j because it removes the possibilities that τ_j may be executing before τ_i for $i < j$ while τ_i is not runnable due to some unfinished predecessor of τ_i. Thus, the above lemma holds. □

Then, we transform the serialized task-level graph to a task instance-level backlog dependence graph. This means that, in the resulting graph, each instance $\tau_{i,j}$ only has immediate predecessors that directly affect its response time $R_{i,j}$. This instance-level graph can be directly used in computing the RTDs of the tasks. It is constructed as follows.

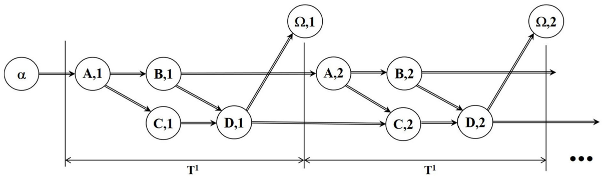
First, the original Γ^k is again augumented with two dummy tasks, τ_α and τ_Ω in the same way as described earlier. We assume that both tasks have zero execution times and that $\phi_\alpha = 0$ and $\phi_\Omega = T^k$. That is, in the example of Γ^1 in Fig. 2a, $\tau_\alpha \rightarrow \tau_A$ and $\tau_D \rightarrow \tau_\Omega$.

Second, in the dummy-augmented graph, we remove any redundant edges in terms of task instance-level backlog dependence. In Fig. 4a, the edge of $\tau_{A,1} \rightarrow \tau_{D,1}$ is removed because the associated dependence is made indirect by alternative sequences of direct dependence, i.e., $\tau_{A,1} \rightarrow \tau_{B,1} \rightarrow \tau_{D,1}$ or $\tau_{A,1} \rightarrow \tau_{C,1} \rightarrow \tau_{D,1}$. That is, when $\tau_{D,1}$ becomes runnable, it is guaranteed that $\tau_{A,1}$ is already completed with both $\tau_{B,1}$ and $\tau_{C,1}$, thus an output message from $\tau_{A,1}$ is always available to $\tau_{D,1}$. Any edge of such indirect dependence should be removed to obtain a graph of direct dependences.

Third, on every core, we add an edge directed from the last task instance in the jth period to the first task instance in the $(j+1)$th period. That is, in Fig. 4a, edges are added between $\tau_{B,1}$ and $\tau_{A,2}$ and between $\tau_{D,1}$ and $\tau_{C,2}$, respectively. The reason is that there exists the backlog dependence between the consecutive instances executing on the same core, despite no inter-task communication.



(a) The response times defined in Γ_1



(b) The backlog dependence graph obtained from Γ_1

Fig. 4. An example of the intra-graph analysis.

As a result, we finally obtain the task instance-level backlog dependence graph, shown in Fig. 4b. This expands over an infinite sequence of periods, but we use a reduced form of it by virtually creating an edge from the last task to the first task on each core, i.e., $\tau_B \rightarrow \tau_A$ and $\tau_D \rightarrow \tau_C$.

Response time computation: To compute the limiting RTDs for all tasks at once, we start to traverse from τ_α the task instance-level backlog dependence graph obtained above and follow the edges to compute the RTD of every task instance. For each $\tau_{i,j}$ visited, starting from the first period, we compute the WTD of $\tau_{i,j}$, i.e., a probability distribution on the waiting time of $\tau_{i,j}$ between the release and the execution start, from the RTDs of all the directly dependent predecessors. Then the RTD of $\tau_{i,j}$ is easily calculated from its WTD and ETD. Once the RTD of every task instance is calculated for the jth period, we increment j and repeat the same process and so on. This process is repeated until we observe that the RTD of $\tau_{\Omega,j}$ converges. It is intuitive that, if the RTD of $\tau_{\Omega,j}$ converges towards a limiting distribution, all the other tasks, which are predecessors to $\tau_{\Omega,j}$, also have limiting RTDs.

In the RTD computation, we need three types of operators on probability distributions: *shrinking* [14], *convolution* and *maximum* [4].

Definition 1. *(Shrinking [14]) For a random variable X, a variable Y obtained by shrinking X by an integer of d, denoted by $X \ominus d$, is defined with the following function:*

$$\mathbf{P}(Y = t) = \begin{cases} 0 & \text{if } t < 0 \\ \sum_{s=-\infty}^{s=d} \mathbf{P}(X = s) & \text{if } t = 0 \\ \mathbf{P}(X = t + d) & \text{if } t > 0 \end{cases}$$

Given the RTD of a directly dependent predecessor τ_X, the shrinking operator can be used to compute the WTD of the successor τ_Y. In this case, the operand d is set to $\lambda_Y - \lambda_X$, thus $WTD_Y = RTD_X \ominus d$.

Definition 2. *(Convolution [4]) For two independent random variables X and Y, the sum Z of X and Y, denoted by $X \otimes Y$, is defined with the following function:*

$$\mathbf{P}(Z=t)=\sum_{s=-\infty}^{s=\infty}\mathbf{P}(X=s)\mathbf{P}(Y=t-s)$$

The convolution operator can be used to compute the RTD of τ_i by convolving the WTD and the ETD of τ_i, i.e., $RTD_i = WTD_i \otimes ETD_i$.

Definition 3. *(Maximum [4]) For two independent random variables X and Y, the maximum Z of X and Y, denoted by $X \bowtie Y$, is defined with the following function:*

$$\mathbf{P}(Z=t)=\sum_{\max(x,y)=t}\mathbf{P}(X=x)\mathbf{P}(Y=y)$$

The maximum operator is used to compute the WTD of τ_Z that has more than one directly dependent predecessor. If τ_Z has two predecessors, i.e., τ_X and τ_Y, $WTD_Z = WTD_{X\to Z} \bowtie WTD_{Y\to Z}$ where $WTD_{X\to Z} = RTD_X \ominus (\lambda_Z - \lambda_X)$ and $WTD_{Y\to Z} = RTD_Y \ominus (\lambda_Z - \lambda_Y)$.

Applying these three operators to the backlog dependence graph, we can compute the limiting RTD for each task if it exists. Let us take the example of Fig. 4. From now on, we denote a probability distribution for random variable X as an ordered list of probability values, i.e., [$\mathbf{P}(X=0)$, $\mathbf{P}(X=1)$, $\mathbf{P}(X=2)$, $\cdots$, $\mathbf{P}(X=X^{\max})$]. Suppose that $T^1 = 6$, $ETD_A = ETD_B = ETD_C = ETD_D = [0, {}^1/_3, {}^1/_3, {}^1/_3]$, and $\Phi_{A,1}$ = 1, $\Phi_{B,1} = \Phi_{C,1}$ = 2, and $\Phi_{D,1}$ = 4. Then, for the first period, the RTD of each task is computed as follows.

$$\begin{aligned}
RTD_{A,1} &= WTD_{\alpha\to A,1}\otimes ETD_A\\
&= [1]\otimes[0,{}^1/_3,{}^1/_3,{}^1/_3] = [0,{}^1/_3,{}^1/_3,{}^1/_3]\\
RTD_{B,1} &= WTD_{A,1\to B,1}\otimes ETD_B\\
&= (RTD_{A,1}\ominus 1)\otimes ETD_B\\
&= [{}^1/_3,{}^1/_3,{}^1/_3]\otimes[0,{}^1/_3,{}^1/_3,{}^1/_3]\\
&= {}^1/_3\times[0,{}^1/_3,{}^1/_3,{}^1/_3] + {}^1/_3\times[0,0,{}^1/_3,{}^1/_3,{}^1/_3]\\
&\quad + {}^1/_3\times[0,0,0,{}^1/_3,{}^1/_3,{}^1/_3]\\
&= [0,{}^1/_9,{}^2/_9,{}^3/_9,{}^2/_9,{}^1/_9]\\
RTD_{C,1} &= RTD_{B,1}\\
RTD_{D,1} &= WTD_{D,1}\otimes ETD_D\\
&= (WTD_{B,1\to D,1}\bowtie WTD_{C,1\to D,1})\otimes ETD_D\\
&= ((RTD_{B,1}\ominus 2)\bowtie(RTD_{C,1}\ominus 2))\otimes ETD_D\\
&= ([{}^3/_9,{}^3/_9,{}^2/_9,{}^1/_9]\bowtie[{}^3/_9,{}^3/_9,{}^2/_9,{}^1/_9])\otimes ETD_D\\
&= [{}^9/_{81},{}^{27}/_{81},{}^{28}/_{81},{}^{17}/_{81}]\otimes[0,{}^1/_3,{}^1/_3,{}^1/_3]\\
&= [0,{}^9/_{243},,{}^{36}/_{243},{}^{64}/_{243},{}^{72}/_{243},{}^{45}/_{243},{}^{17}/_{243}]
\end{aligned}$$

Then, for the second period, we can compute the RTD of each task in the same way as above, but have to consider the backlog dependence of $\tau_{B,1} \to \tau_{A,2}$ on CPU 1 and $\tau_{D,1} \to \tau_{C,2}$ on CPU 2. $WTD_{B,1\to A,2}$ = $RTD_{B,1} \ominus (\lambda_{A,2} - \lambda_{B,1})$ = $RTD_{B,1} \ominus 5$ = [1], which is a one-point distribution meaning no backlog, thus not affecting $RTD_{A,2}$. On the contrary, $WTD_{D,1\to C,2}$ = $RTD_{D,1} \ominus (\lambda_{C,2} - \lambda_{D,1})$ = $RTD_{D,1} \ominus 4$ = [${}^{181}/_{243}$, ${}^{45}/_{243}$, ${}^{17}/_{243}$], which is not a one-point distribution, thus affecting $RTD_{C,2}$.

To compute the limiting distribution of $WTD_{D,j\to C,j+1}$, we repeat the above computation process while incrementing the period index of j until we observe that WTD_Ω converges. If it converges, the WTD of the first task on each core, and in turn, the RTD of every task converges too. To see if WTD_Ω converges, we perform the Kolmogorov-Smirnov test between $WTD_{\Omega,j}$ and $WTD_{\Omega,j+1}$.

Notice that it is counter-intuitive that $WTD_{D,1\to C,2}$ is not null if we consider the maximum core utilization of CPU 2, i.e., $U^{\max} = C_C^{\max}/T^1 + C_D^{\max}/T^1 = {}^3/_6 + {}^3/_6 \le 1.0$. On a single-core system, if $U^{\max} \le 1.0$, it is guaranteed no backlog occurs to the first task instance in the next period. However, on multi-core systems, it does not hold because an idle time may occur on CPU 2 between η_C and λ_D when $\eta_C < \eta_B$. Since this idle time cannot be utilized by any other tasks on CPU 2, it has the effect of making the $U^{\max}$ of CPU 2 exceed 1.0.

The procedure to compute the RTD of τ is pseudo-coded in Table 1, and the procedure to compute the RTDs of all tasks in the period index of j is pseudo-coded in Table 2.

TABLE 1
RTD Computation of a Task Instance τ

```
{ $RTD_\tau$ on success } = ComputeRTD($\tau$)

{ // $ipred(\tau)$ = immediate predecessors of $\tau$
    for (each $\tau_{ipred}$ of $ipred(\tau)$) {
        if ($RTD_{ipred(\tau)}$ is not yet computed)
            return False;
        $IRT = \lambda_\tau - \lambda_{ipred(\tau)}$; // inter-release time
        $WTD_\tau^{ipred(\tau)}$ = $RTD_{ipred(\tau)} \ominus IRT$;
    }
    $WTD_\tau$ = $WTD_\tau^{ipred1(\tau)} \bowtie \cdots \bowtie WTD_\tau^{ipred\_n_\tau(\tau)}$;
    $RTD_\tau$ = $WTD_\tau \otimes ETD_\tau$;
    return True;
}
```

Theorem 1. *Let Γ^{serial} be a graph obtained from Γ^{orig} by the serialization defined in Lemma 1, and Γ^{bdep} be the task instance-level backlog dependence graph obtained from Γ^{serial}. We define the RTD_i^{serial} as the limiting $RTD_{i,j}^{bdep}$ obtained when $j \to \infty$. If the RTD_i^{serial} exists, it upper-bounds the RTD_i^{orig} of τ_i in Γ^{orig} in that $\mathbf{P}(R_i^{orig} \le t) \ge \mathbf{P}(R_i^{serial} \le t)$ for any $t > 0$.*

Proof. According to Lemma 1, in the first period, the $RTD_{i,1}^{bdep}$ for each $\tau_{i,1}$ in Γ^{bdep} upper-bounds the $RTD_{i,1}^{orig}$ of $\tau_{i,1}$ in Γ^{orig} due to the serialization. Then, let us consider the $WTD_{1,2}^{bdep}$ of the first task instance $\tau_{1,2}$ in the second hyperperiod. This $WTD_{1,2}^{bdep}$ is obtained by applying $\ominus$ to the $RTD_{n,1}^{bdep}$ of the last task instance $\tau_{n,1}$ on the core in the first hyperperiod, since $\tau_{n,1}$ is the only predecessor of $\tau_{1,2}$ in terms of the backlog. It is clear that $WTD_{1,2}^{bdep}$ also upper-bounds $WTD_{1,2}^{orig}$ because $RTD_{n,1}^{bdep}$ upper-bounds $RTD_{n,1}^{orig}$. Then, the $RTD_{i,2}^{bdep}$ for each $\tau_{i,2}$ in turn in the second hyperperiod upper-bounds the $RTD_{i,2}^{orig}$ of $\tau_{i,2}$ due to both the bounding $WTD_{1,2}^{bdep}$ and the serialization. By induction, the $WTD_{1,j}^{bdep}$ is obtained by applying $\ominus$ to the $RTD_{n,j-1}^{bdep}$ of the last task instance on the core in the $(j-1)$th hyperperiod, and it follows that the $RTD_{i,j}^{bdep}$ for every $\tau_{i,j}$ upper-bounds the $RTD_{i,j}^{orig}$ of $\tau_{i,j}$. Thus, we can conclude that, for each $\tau_{i,j}$ in a steady-state period $j \to \infty$, the limiting $RTD_{i,j}^{serial}$ upper-bounds the limiting $RTD_{i,j}^{orig}$. Therefore, the above theorem holds. □

Intra-graph latency computation: Once the limiting RTD is computed for every task in a subgraph, for any path of tasks within the subgraph, we can easily determine the end-

TABLE 2
Intra-Graph Analysis for a Period Index of j

```
{ $RTD_\tau$ for every $\tau$ } = AnalyzeGraphInPeriod($j$)
    {
        $tasks\_to\_visit$ = { $\tau_{\alpha,j}$ } ;
        $tasks\_visited$ = { } ;
        $tasks\_to\_compute$ = { } ;
        while ($tasks\_to\_visit$ is not empty) {
            $\tau$ = dequeue($tasks\_to\_visit$);
            if ($\tau$ belongs to $tasks\_visited$) continue;
            enqueue($tasks\_visited$, $\tau$);

            if ($\tau$ == $\tau_{\Omega,j}$) continue; // skip the closing dummy task
            else if ($\tau$ == $\tau_{\alpha,j}$)
                $RTD_{\alpha,j}$ = [ 1.0 ]; // assign the initial one-point distribution.

            for (each $\tau_{isucc}$ of $isucc(\tau)$) {
                enqueue($tasks\_to\_visit$, $\tau_{isucc}$);
                if ($RTD_{\tau_{isucc}}$ is already computed) continue;
                if (computeRTD($\tau_{isucc}$)) {
                    if ($\tau_{isucc}$ belongs to $tasks\_to\_compute$)
                        remove($tasks\_to\_compute$, $\tau_{isucc}$);
                }
                else
                    enqueue($tasks\_to\_compute$, $\tau_{isucc}$);
            }
        }

        while ($tasks\_to\_compute$ is not empty) {
            $\tau$ = dequeue($tasks\_to\_compute$);
            if (!computeRTD($\tau$))
                enqueue($tasks\_to\_compute$, $\tau$);
        }
    }
```

to-end latency distribution. For the example of Fig. 4, $L(\tau_A \rightarrow \tau_D) = (\phi_D - \phi_A) + R_D$. That is, for any path $\tau_X \rightarrow \tau_Y$, the end-to-end latency can be expressed as follows:

$$L(\tau_X \rightarrow \tau_Y) = (\phi_Y - \phi_X) + R_{Y,j} \text{ where } j \rightarrow \infty. \tag{1}$$

Thus, the LD can be obtained simply as $RTD_Y \ominus (\phi_X - \phi_Y)$ since $\ominus$ can handle the negative value according to the definition.

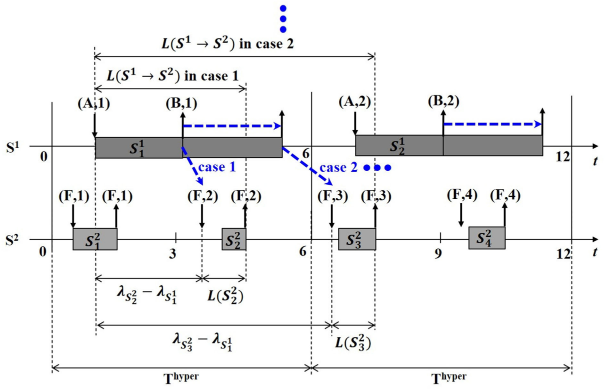
Theorem 2. *Given the limiting RTD_τ^{serial} for every τ in Γ^{serial} described in Theorem 1, for any path S in Γ^{serial}, the limiting latency distribution LD^{serial} of S upper-bounds LD^{orig} of S in Γ^{orig}.*

Proof. This is straightforward from Theorem 1 and Equation 1. □

3.2 Inter-Graph Analysis

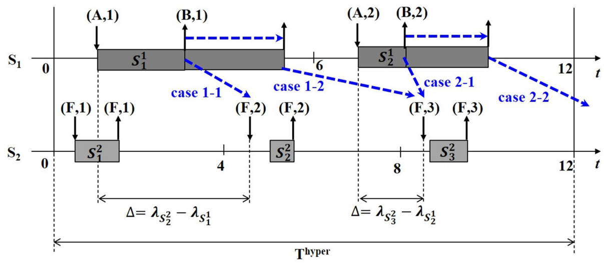
Given the limiting RTD of every τ in each Γ^{serial}, we can now compute the end-to-end latency of a path S from a task in a subgraph to another task in a different subgraph. For each Γ through which S penetrates, let us number it in the order of appearance as $\Gamma^1, \Gamma^2, \cdots, \Gamma^L$. Recall that the same subgraph does not appear twice on S because the original graph is a DAG. Then S can be decomposed into path segments S^1, $S^2, \cdots, S^L$ where S^l is a subsequence of tasks that belongs to Γ^l. Thus, for the example of $S = \tau_A \rightarrow \tau_B \rightarrow \tau_F$ in Fig. 2a, it can be rewritten as $S^1 \rightarrow S^2$ where $S^1 = \tau_A \rightarrow \tau_B$ and $S^2 = \tau_F$.

The complexity of the inter-graph analysis lies in between two consecutive path segments. By definition, the two segments, say S^1 and S^2, have different period values, i.e., $T^1 \neq T^2$, thus there are possibilities that an output message generated from an instance S_m^1 may reach different instances of S^2, depending on the completion time of S_m^1. For the example of $S = S^1 \rightarrow S^2$, depending on the completion time of S_1^1, the message sent by S_1^1 may be received by S_2^2, S_3^2, $S_4^2, \cdots$, released in the subsequent invocations of T^2. Fig. 5a shows this scenario when $T^1 = T^{hyper}$.



$L(S^1 \rightarrow S^2)$ in case 2
$L(S^1 \rightarrow S^2)$ in case 1
(A,1)
(B,1)
(A,2)
(B,2)
S^1
S_1^1
S_2^1
case 1
case 2
(F,1)
(F,2)
(F,3)
(F,4)
S^2
S_1^2
S_2^2
S_3^2
S_4^2
$\lambda_{S_2^2} - \lambda_{S_1^1}$
$L(S_2^2)$
$\lambda_{S_3^2} - \lambda_{S_1^1}$
$L(S_3^2)$
T^{hyper}
t

(a) A case of harmonic periods ($T^1 = 6$ and $T^2 = 3$)



(b) A case of non-harmonic periods ($T^1 = 6$ and $T^2 = 4$)

Fig. 5. Examples of the inter-graph analysis.

Lemma 2. *For an instance of $S = S^1 \rightarrow S^2$, let us suppose that S_m^1 is completed between the release of S_{n-1}^2 and that of S_n^2. Then, the end-to-end latency from S_m^1 to S^2 can be expressed as follows:*

$$L(S_m^1 \rightarrow S^2) \leq \lambda_{S_n^2} - \lambda_{S_m^1} + L(S_n^2) \quad \text{if } \lambda_{S_{n-1}^2} < \eta_{S_m^1} \leq \lambda_{S_n^2} \tag{2}$$

Proof. Consider a list of pending instances of S^2, i.e., instances whose execution is yet to be started, at time $\lambda_{S_n^2}$. If only S_n^2 is pending, the message from S_m^1 is received by S_n^2, thus $L(S_m^1 \rightarrow S^2) = L(S_m^1 \rightarrow S_n^2) = \lambda_{S_n^2} - \lambda_{S_m^1} + L(S_n^2)$. However, if S_{n-1}^2 is not completed at time $\lambda_{S_n^2}$, thus both S_{n-1}^2 and S_n^2 are pending, the message is received by S_{n-1}^2, thus $L(S_m^1 \rightarrow S^2) = L(S_m^1 \rightarrow S_{n-1}^2) = \lambda_{S_n^2} - \lambda_{S_m^1} + L(S_{n-1}^2) \ominus T^2$. By induction, if $S_{n-k}^2, \cdots, S_{n-1}^2$ and S_n^2 are pending, the message is received by S_{n-k}^2, thus $L(S_m^1 \rightarrow S^2) = L(S_m^1 \rightarrow S_{n-k}^2) = \lambda_{S_n^2} - \lambda_{S_m^1} + L(S_{n-k}^2) \ominus k \times T^2$. Note that $L(S_{n-k}^2) \ominus k \times T^2 = \eta_{S_{n-k}^2} - \lambda_{S_n^2}$, thus $L(S_m^1 \rightarrow S_{n-k}^2) = \eta_{S_{n-k}^2} - \lambda_{S_m^1} < L(S_m^1 \rightarrow S_n^2) = \eta_{S_n^2} - \lambda_{S_m^1}$ since $\eta_{S_{n-k}^2} < \eta_{S_n^2}$. Therefore, for any $k \geq 0$, the above lemma holds. □

At this point, recall that $L(S_m^1)$ and $L(S_n^2)$ are already computed in the previous intra-graph analysis as the limiting distributions by $m \rightarrow \infty$ and $n \rightarrow \infty$. Moreover, in the steady state, we can safely assume that $LD(S_{n-k}^2) = \cdots = LD(S_n^2)$ when $n \rightarrow \infty$.

TABLE 3
Computation of LD From LD^{head} and LD^{tail}

```
{ $LD$ } = ComputeLD($LD^{head}$, $LD^{tail}$, $T^{tail}$, $\Delta$)  where $\Delta \geq 0$
{
    if ($\Delta$ == 0) $\Delta$ = $T^{tail}$;
    $\delta$ = $T^{tail} - \Delta$;
    $LD$ = [ 0, 0, 0, ... ]; // initialized as an empty distribution
    $count$ = 0;
    for ($idx$ = 1; $idx$ < len($LD^{head}$); $idx$ += $T^{tail}$) {
        $s\_idx$ = max(0, $idx - \delta$);
        $e\_idx$ = $T^{tail} + idx - \delta$;
        $prob\_cond$ = sum($LD^{head}[s\_idx \leq L < e\_idx]$);
        $condLD$ = $prob\_cond \times LD^{tail}$;
        $condLD$ = $condLD \ominus -(\Delta + T^{tail} \times count)$;
        $LD$ = $LD \oplus condLD$; // coalesced LD
        $count$ += 1;
    }
    return $LD$;
}
```

TABLE 4
End-to-End Latency Analysis of Path S

```
{ $LD(S)$ } = AnalyzePath( $S = S^1 \rightarrow \cdots \rightarrow S^L$ )
{
    $T^{hyper}$ = LCM($T^1$, $T^2$, $\cdots$, $T^L$);
    $M$ = $T^{hyper}/T^1$;
    for ($m$ = 1; $m$ <= $M$; $m$ += 1) {
        $\varphi^1_m$ = $\varphi^1$ + $(m-1) \times T^1$; // graph phase of $S^1_m$
        $LD_m$ = $LD(S^1)$;
        for ($l$ = 2; $l$ <= $L$; $l$ += 1) {
            $\varphi^l_n$ = $\varphi^l$ + $\left\lceil \frac{\varphi^1_m - \varphi^l}{T^l} \right\rceil \times T^l$; // graph phase of $S^l_n$
            $\Delta$ = $\varphi^l_n - \varphi^1_m$; // $\Delta \geq 0$
            $LD_m$ = ComputeLD($LD_m$, $LD(S^l)$, $T^l$, $\Delta$);
        }
    }
    $LD(S)$ = $\frac{1}{M} \times (LD_1 \oplus LD_2 \oplus \cdots \oplus LD_M)$;
    return $LD(S)$;
}
```

Using Equation 2, we can condition on $L(S^1_m)$ so that we can enumerate all possible scenarios that may happen between S^1 and S^2. This conditioning process should be started from a minimum value of the index n of S^2_n that the minimum possible value of $L(S^1_m)$ may lead to and should be finished to a maximum value of n that the maximum possible value of $L(S^1_m)$ may lead to. This process is pseudo-coded in Table 3 as `ComputeLD`, which computes the end-to-end LD over two consecutive path segments, *head* and *tail*. In this procedure, note that the argument of Δ should be given as $\lambda_{S^{tail}} - \lambda_{S^{head}}$, and that an operator of $\oplus$, called *coalescion* [3], is used to merge all conditional distributions into one distribution. The definition of coalescion is as follows:

Definition 4. *(Coalescion [3]) For two independent conditional random variables X and Y, the coalesced variable Z of X and Y, denoted by $X \oplus Y$, is defined with the following function:*

$$\mathbf{P}(Z = t) = \mathbf{P}(X = t) + \mathbf{P}(Y = t)$$

Note that $\sum_t \mathbf{P}(X = t) < 1$ and $\sum \mathbf{P}(Y = t) < 1$ since X and Y are conditional. If X and Y covers all the sample space of Z, $\sum \mathbf{P}(Z) = 1$.

This idea of conditioning on the end-to-end latency of the head segment can be generalized to cover all the subsequent segments of the path. For this generalization, we first consider the harmonic case where $T^1 = T^{hyper}$. In this case, there exists only one instance of S^1 in the hyperperiod, thus $LD(S^1 \rightarrow \cdots \rightarrow S^L)$, abbreviated as $LD^{S^1}_{S^L}$, can be computed as follows:

$$\begin{aligned}
LD^{S^1}_{S^2} &= \texttt{ComputeLD}(LD(S^1), LD(S^2), T^2, \Delta_2),\\
LD^{S^1}_{S^3} &= \texttt{ComputeLD}(LD^{S^1}_{S^2}, LD(S^3), T^3, \Delta_3),\\
&\vdots\\
LD^{S^1}_{S^L} &= \texttt{ComputeLD}(LD^{S^1}_{S^{L-1}}, LD(S^L), T^L, \Delta_L)
\end{aligned}$$

where each $LD(S^l)$ is defined as its limiting distribution, and $\Delta_l = \lambda_{S^l_n} - \lambda_{S^1_1}$, which assumes that S^l_n is the first instance of S^l released since $\lambda_{S^1_1}$.

Next, consider the case where $T^1 < T^{hyper}$, as shown in Fig. 5b. Since we have more than one instance of S^1 in the hyperperiod, using `ComputeLD`, we compute $LD_m = LD(S^1_m \rightarrow \cdots \rightarrow S^L)$ for every instance S^1_m in the hyperperiod, and then average all the computed LDs for $m = 1, \ldots, M$ ($M = T^{hyper}/T^1$). The reason why we perform the computation for each instance of S^1 is that, for the example of Fig. 5b, the inter-graph phasing Δ between S^1_1 and S^2_2 and the one between S^1_2 and S^2_3 are different. To generalize this idea, $LD(S) = \frac{1}{M} \times (LD_1 \oplus LD_2 \oplus \cdots \oplus LD_M)$. This computation process is pseudo-coded in Table 4 as `AnalyzePath`.

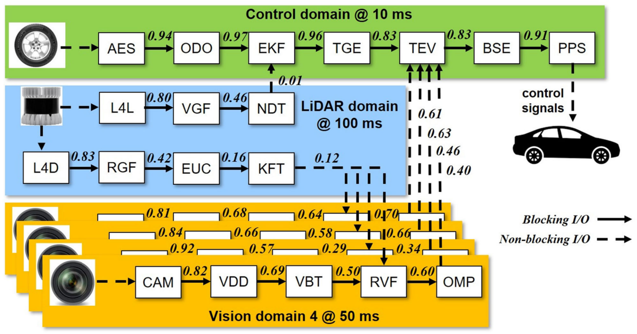
Note that there may be the low chances that S^{l+1} receives more than one message from S^l. In Fig. 5b, we can see that S^2_3 may receive two messages from S^1_1 (case 1-2) and S^1_2 (case 2-1), respectively, for the case where S^1_1 completes between $\lambda_{S^2_2}$ and $\lambda_{S^2_3}$, and S^1_2 completes before $\lambda_{S^2_3}$. Since we do not allow message dropping in our model, even if S^2_3 generates only one output message in response to the two input messages, we virtually treat the single output message as two in order to complete the end-to-end latency of both input messages. Finally, the following theorem states that the inter-graph analysis described above is safe.

Theorem 3. *Given the limiting RTD^{serial}_τ for every τ in each Γ of $\mathbf{\Gamma^{serial}}$, for any path $S = S^1 \rightarrow \cdots \rightarrow S^L$, the limiting LD^{serial} of S upper-bounds the LD^{orig} of S in $\mathbf{\Gamma^{orig}}$.*

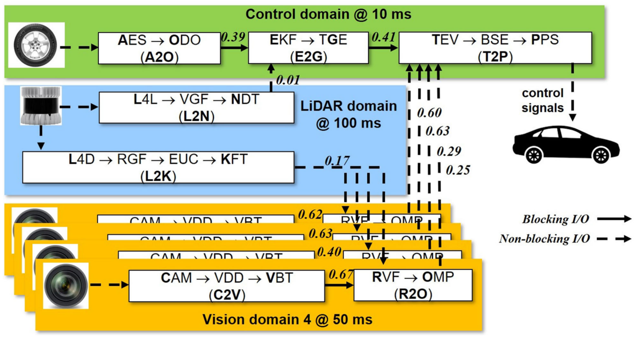
Proof. First, for $S = S^1 \rightarrow S^2$, this theorem holds by Theorem 2 and Lemma 2. Second, for $S = S^1 \rightarrow S^2 \rightarrow S^3$, it still holds because $S^1 \rightarrow S^2$ can be treated as one virtual segment from the viewpoint of S^3. By induction, for $S = S^1 \rightarrow \cdots \rightarrow S^L$, it still holds because $S^1 \rightarrow \cdots \rightarrow S^{L-1}$ can be treated as one virtual segment from the viewpoint of S^L. Thus, the above lemma holds. □

3.3 Task Grouping

The proposed analysis is based on the assumption of independent execution times between tasks, but it may not hold in reality. To deal with inter-task dependent execution times, one possible approach is to derive a joint distribution of task execution times for any two consecutive tasks and revise the proposed analysis in order to be based on the joint distributions. However, it is clear that such an approach will increase the analysis complexity significantly, because it will split a single execution time distribution into multiple pieces and create a huge number of



(a) The graph of Autoware tasks



(b) The graph of Autoware task groups

Fig. 6. The Autoware task graph (arrows annotated with Pearson Correlation Coefficients).

combinations of the pieces from different tasks for the analysis.

In the paper, we take the approach of task grouping, which groups as many dependent tasks as possible into a single virtual task. In this grouping, a group can only be composed of tasks running on the same core, every source task and every task with an immediate predecessor in a different subgraph should open a new group so that every task path starting from such a task can have the corresponding group path.

Fig. 6a shows the entire task graph of our customized Autoware, divided into six disjoint subgraphs: one control graph Γ^C, one LiDAR graph Γ^L, and N vision graphs $\Gamma^{V(n)}$. It is slightly modified in order to comply with our assumption of blocking intra-graph and non-blocking inter-graph communications. Table 5 gives a detailed description about the task set.

To evaluate the degree of the inter-task dependence in the execution times, we analyze the Pearson Correlation Coefficient (PCC) between any two consecutive Autoware tasks. The numbers annotated to the arrows in Fig. 6a are the PCCs. A PCC value of 1.0 means a strong positive correlation, a value of -1.0 means a strong negative correlation, and a value of zero means no correlation in general. From the figure, we can see that many pairs of Autoware tasks have a higher PCC value than 0.7, which may compromise the assumption of our analysis. To mitigate the strong inter-task correlations, we perform task grouping and analyze the inter-group PCC by profiling the group-level execution times. The analyzed PCCs are shown in Fig. 6b. On the one hand, this task grouping has the main effect of hiding the inter-task correlations within each group. On the other hand, it produces the lowered inter-group PCCs than the associated inter-task PCCs for some cases. For example, PCC(ODO → EKF) = 0.97 decreases to PCC(A2O → E2G) = 0.39, PCC(TGE → TEV) = 0.83 decreases to PCC(E2G → T2P) = 0.41, PCC(OMP(3) → TEV) = 0.46 decreases to PCC (R2O(3) → T2P) = 0.29, and PCC(OMP(4) → TEV) = 0.40 decreases to PCC(R2O(4) → T2P) = 0.25. As another example, PCC(VBT(3) → RVF(3)) = 0.29 increases to PCC(C2V(3) → R2O(3)) = 0.40, and PCC(VBT(4) → RVF(4)) = 0.50 increases to PCC(C2V(4) → R2O(4)) = 0.67. In the other cases, the inter-group PCC is almost equal to or slightly higher than the associated inter-task PCC.

4 Experimental Results

In this section, we evaluate the proposed analysis in terms of probabilistic guarantee and analysis accuracy. For experiments, we use a real AV of our own [15], that runs our customized Autoware stack. A sensor workload is recorded while the AV is driving at 30 to 40 km/h for 15 minutes in K-City [16].

4.1 Experimental Setup

The computing system used in the experiments executes Ubuntu Linux 18.04 and Autoware 1.13 in a docker container on top of the Linux. For the experiments, we use a plain Linux kernel v5.4.0 without any real-time extension such as PREEMPT_RT since the Autoware is executed with the pre-recorded sensor workload and no sensor devices attached. It is equipped with an Intel i7 processor with 6 cores (12 virtual cores) operating at 2.6 GHz, main memory of 32 GB, and a GPU of RTX 2070. Our Autoware stack is instrumented to collect timing information. To profile the task (group) ETDs, for every instance of each task, which corresponds to an iteration of the main loop of the task, the start time and the finish time are recorded with a Linux system call `clock_gettime`. This profiling overhead is measured as less than 50 us for each task instance[1]. To profile the end-to-end LDs for each task path, the entire Autoware stack is instrumented so that every sensor message includes a unique sequence number, and a sequence of tasks executed in response to the message relays the sequence number in their output message. Thus, the end-to-end latency is measured as the difference between the instant when the first (source) task receives a sensor message and the instant when the last (sink) task outputs a control message with the same sequence number.

Table 5 shows the parameters of the Autoware tasks. The CPU assignment and the task phases are determined so that each subgraph is serialized. In the table, note that the VDD tasks share the single GPU to execute a deep neural network called YOLOv3, which may make strong inter-task correlations. From the table, it can be seen that the entire task graph is partitioned into 4 + N cores, which are roughly balanced in terms of $\bar{U}$ and that core 0 and 2 have $U^{\max} > 1.0$. To

1. To evaluate the profiling overhead, we instrument an Autoware task so that it can get a timestamp for every iteration of the main loop. The total execution time of the instrumented VDD task for 1000 iterations is 5896.840 ms with profiling and 5872.829 ms without profiling. Thus, the net profiling overhead of a single call of `clock_gettime` is $\frac{5896.840-5872.829}{1000} = \frac{24.011}{1000} = 0.024$ ms.

TABLE 5
The Autoware Task Set Used in the Experiments ($n = 1, 2, \ldots, N$)

Group	Task	Description	T(ms)	Φ (ms)	$\bar{C}$ (ms)	$C^{\max}$ (ms)	CPU ID	$\bar{U}$	$U^{\max}$
A2O	AES	AE System Driver	10	0	1.00	2	0	0.27	1.10
	ODO	Odometer Driver							
E2G	EKF	Extended Kalman Filter		0	1.67	9			
	TGE	Trajectory Generator							
T2P	TEV	Trajectory Evaluator		0	2.23	10	1	0.22	1.00
	BSE	Behavior Selector							
	PPS	Pure Pursuit							
L2N	L4L	LiDAR Driver for Localization	100	0	32.20	112	2	0.32	1.12
	VGF	Voxel Grid Filter							
	NDT	NDT-based Localization							
L2K	L4D	LiDAR Driver for Detection		0	18.21	73	3	0.18	0.73
	RGF	Ray Ground Filter							
	EUC	Euclidean Clustering							
	KFT	Kalman Filter Contour Track							
C2V(n)	CAM(n)	Camera Driver	50	$\frac{50}{N} \times (n-1)$	10.75$\sim$13.12	31$\sim$39	3+n	0.23$\sim$0.28	0.70$\sim$0.86
	VDD(n)	Vision Darknet Detect							
	VBT(n)	Vision Beyond Track							
R2O(n)	RVF(n)	Range Vision Fusion		$\frac{50}{N} \times (n-1)$	1.00	2$\sim$4			
	OMP(n)	Open Motion Predictor							

determine all task ETDs, the execution times of each task are profiled in unit of microseconds and discretized in unit of millisecond.

To the Autoware task graph shown in Fig. 6, we feed a sensor workload that is recorded from our vehicle with the following sensors: an odometer, a 64-channel Ouster LiDAR installed at the roof, and a full-HD Sekonix camera installed behind the rear-view mirror. To see the effect of multiple cameras, we replicate images from the single camera and feed them into the multiple vision graphs.

4.2 Probabilistic Guarantee

To evaluate the proposed analysis, we profile the latency distribution of each possible path from a source group to the sink group. In Fig. 6b, there are ten paths of task groups with four vision graphs, whose LDs are shown in Fig. 7. This figure shows the CDF form of the profiled LDs in green and the analyzed LDs in black. In the CDF, if the analyzed graph is always below the profiled graph, it means that the former upper-bounds the latter. From the figure, we can see that the analyzed graph has a strong tendency to upper-bound the profiled graph for all paths. There are, however, cases where the starting part of the analyzed graph is above the profiled graph in Figs. 7e, 7f, 7i, and 7j. These are the paths that share the single GPU, which show a strong inter-task dependence in the execution times. At this point, it is worth noting that the inter-group PCCs associated with the vision graphs are relatively high and not uniform in Fig. 7b. For example, PCC(C2V(n) $\rightarrow$ R2O(n)) = 0.62, 0.63, 0.40, 0.67 for $n = 1, 2, 3, 4$, respectively. All the latency graphs in Fig. 7, except for Fig. 7a, have staircase shapes due to the latency conditioning explained in Fig. 5.

Table 6 shows the tail latencies TL of 99.9999% percentile obtained from the profiled and analyzed graphs for each path. From this table, for both cases of $N = 4$ and $N = 6$, we can see that the analyzed tail latency TL_{az} is larger than the profiled TL_{pf} for every path. This is because the analysis covers an infinite sequence of periods, thus tends to have a longer tail than the profiling in a finite-time duration. In the paper, the main interest is such a tail latency, which explains a probabilistic worst-case responsiveness of the vehicle. Recall that any existing deterministic analysis to compute the single worst-case latency value cannot be applied to our multi-core system with $U^{\max} > 1.0$ for some core because it gives an infinite latency value. Also, it is important that the TL's of C2V(n) $\rightarrow$ R2O(n) $\rightarrow$ T2P with $N = 6$ tends to be greater than those with $N = 4$. Since more tasks use the GPU, the GPU utilization gets higher, thus the possibility increases that the C2V tasks are executed in an interleaved manner within the GPU. When such interleaved executions occur, the execution time of the task is measured as larger than the one obtained when it runs alone. Table 7 shows the average execution times of the C2V tasks while varying N. From the figure, we can see that the average execution times obtained when $N = 6$ clearly increase compared with those of the smaller N. This implies more interleaved executions.

4.3 Analysis Accuracy

In order to evaluate the accuracy of our analysis over a serialized graph $\boldsymbol{\Gamma}^{serial}$, we generate synthetic task graphs. Every synthetic task graph is decomposed into five subgraphs, which are connected in series, that is, $\Gamma^1 \rightarrow \Gamma^2 \rightarrow \cdots \rightarrow \Gamma^5$. The five subgraphs have a set of random phases, (φ^1, φ^2, $\cdots$, φ^5), and harmonic periods of (T^1, T^2, T^3, T^4, T^5) = ($16T$, $8T$, $4T$, $2T$, T). Each Γ^k consists of the same number N of tasks, which are also connected in series, that is, $\tau_1^k \rightarrow \tau_2^k \rightarrow \cdots \rightarrow \tau_N^k$. All the tasks of Γ^k is assigned to CPU k with zero task phases. Each task τ_i^k has a uniform ETD ranging from 1 to $2 \times \bar{C}^k - 1$. In each Γ^k, every τ_i^k has the same average execution time $\bar{C}^k$, thus the average core utilization $\bar{U}$ of core k is $\frac{N \times \bar{C}^k}{T^k}$. We also assume that every core has the same average utilization $\bar{U}$. Note that this synthetic task graph is

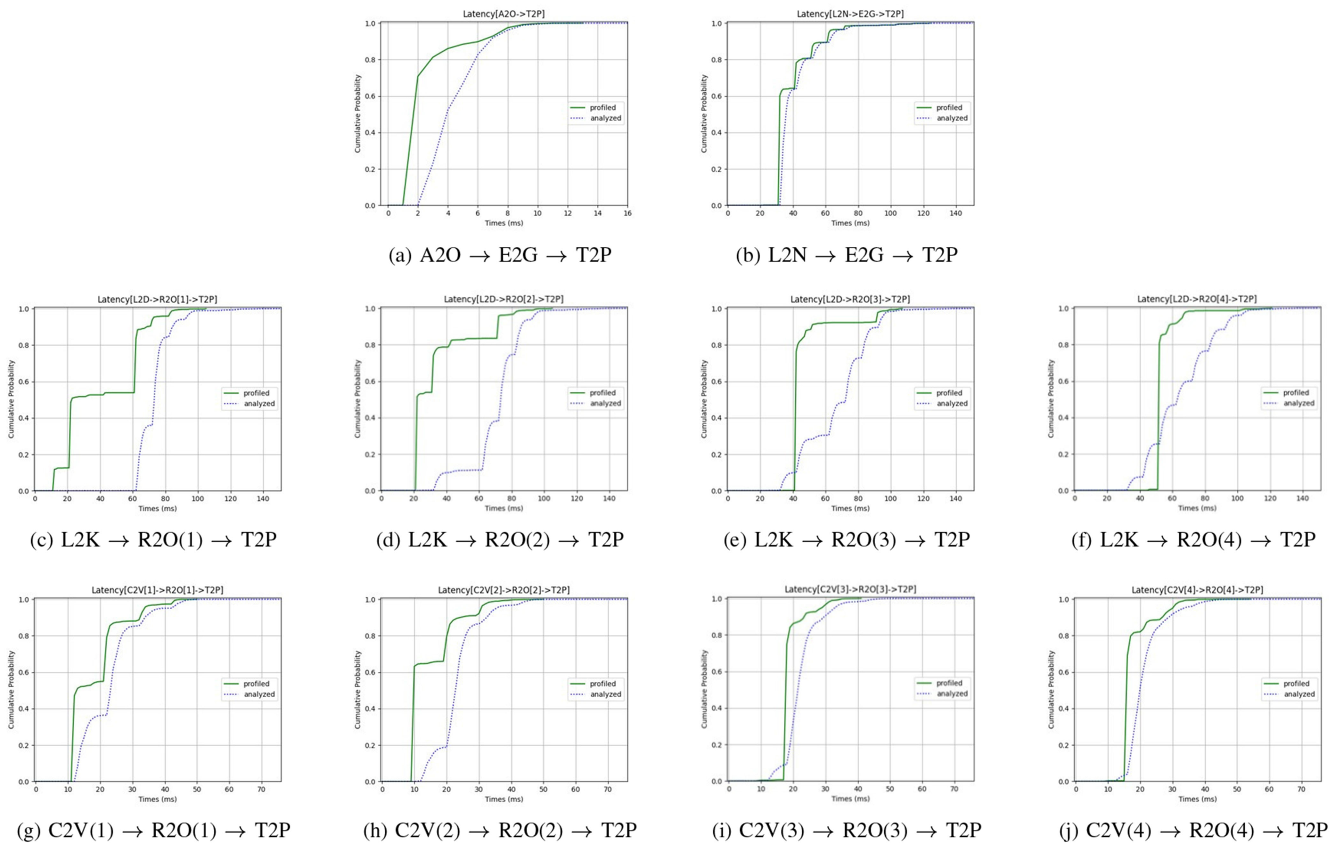


Fig. 7. Comparison between the profiled LD and the analyzed LD for each task path in Autoware.

general enough to create various cases that our analysis should encounter. That is, if we consider an arbitrary task graph, the inter-graph analysis analyzes each path from a source to a sink, and each single path under the analysis can be regarded as a sequence of subgraphs with sequential tasks. The above synthetic graph represents one such path.

For the experiments, 100 synthetic graphs are generated with random phase combinations while varying $\bar{U}$. The average core utilization $\bar{U}$ is chosen among 40%, 60% and 80%. Recall that $U^{\max}$ is almost equal to $2 \times \bar{U}$ since every task has a maximum execution time of $2 \times \bar{C}^k - 1$. For each synthetic graph, we analyze $TL^{99.9\%}$ and $TL^{99.9999\%}$ for each path $S^1 \rightarrow S^k$. To factor out the effect of the subgraph periods, we normalize the tail latency by dividing it by T^1, and take the average of the 100 normalized tail latencies obtained for each $L_{S^k}^{S^1}$. For comparison purposes, we also perform simulations for the 100 synthetic graphs for a duration of $160,000T$ with our own discrete-event simulator. It is written in Python language and only mimics the lifecycle events of tasks (i.e., task arrival, running, waiting, preemption, and completion) and those of messages (i.e., creation, transmission, and destruction) according to the task graph and the resource assignment. From the simulations, we take the average of the 100 normalized tail latencies obtained, and compare it with the averaged tail latency obtained from the analyses. Fig. 8 shows the averaged normalized tail latency obtained for each path with $N = 8$, while varying $\bar{U}$. From the figure, it is clear that the averaged $TL^{99.9999\%}$ (or $TL^{99.9\%}$) from the analyses is always higher than that from the simulations. Finally, note that, when $\bar{U}$=80%, the overhead in the analysis accuracy for $TL^{99.9999\%}$ is merely $\frac{3.19-2.87}{2.87} = 11.1\%$.

TABLE 6
The Profiled Tail Latency TL_{pf} and the Analyzed Tail Latency TL_{az} (In Ms) of 99.9999 Percentile

Task path	$N = 4$		$N = 6$	
	TL_{pf}	TL_{az}	TL_{pf}	TL_{az}
A2O → E2G → T2P	13	16	15	17
L2N → E2G → T2P	124	134	142	150
L2K → R2O(1) → T2P	105	150	123	150
L2K → R2O(2) → T2P	105	150	104	150
L2K → R2O(3) → T2P	107	150	105	150
L2K → R2O(4) → T2P	121	150	115	150
L2K → R2O(5) → T2P	-	-	115	150
L2K → R2O(6) → T2P	-	-	105	150
C2V(1) → R2O(1) → T2P	48	54	53	62
C2V(2) → R2O(2) → T2P	49	59	52	61
C2V(3) → R2O(3) → T2P	41	59	51	61
C2V(4) → R2O(4) → T2P	50	58	51	60
C2V(5) → R2O(5) → T2P	-	-	51	60
C2V(6) → R2O(6) → T2P	-	-	50	59

4.4 Analysis Time

In order to understand the complexity of our analysis, we generate three synthetic task graphs. Each task graph is a sequence of three subgraphs, i.e., $\Gamma^1 \rightarrow \Gamma^2 \rightarrow \Gamma^3$, each consisting of four tasks, with a period combination of (T^1, T^2, T^3), shown in Table 8.

From the table, we can see that the analysis time does not significantly increase as T^{hyper} increases. It requires a longer time with a higher $\bar{U}$ for the task RTDs to converge.

TABLE 7
The Average Execution Times (In Ms) of C2V Tasks

Task	$N = 1$	$N = 2$	$N = 4$	$N = 6$
C2V(1)	12.08	12.54	12.98	15.46
C2V(2)	-	12.49	11.04	15.10
C2V(3)	-	-	10.75	14.14
C2V(4)	-	-	10.83	14.55
C2V(5)	-	-	-	13.20
C2V(6)	-	-	-	14.14

TABLE 8
Analysis Times With Different Period Combinations

$T = (T^1, T^2, T^3)$	T^{hyper}	Analysis time (sec)		
		$\bar{U}$=0.7	$\bar{U}$=0.8	$\bar{U}$=0.85
(30, 50, 100)	300	0.71	15.30	66.96
(30, 60, 100)	600	0.80	16.18	68.19
(30, 70, 100)	2100	0.83	16.86	68.92

5 Related Work

Over decades, many end-to-end latency analyses have been performed in context of general distributed systems [17], [18], [19] or multi-resource systems [20], [21], [22], [23], [24]. In the former context, a real-time transaction is defined as a sequence of tasks with maximum execution times and minimum inter-release times. Then the maximum interference experienced by each task in every transaction is analyzed to derive the worst-case response times of the tasks, and in turn, the worst-case end-to-end delay of the transaction. In the latter context, a real-time task is modeled as a graph of subtasks and an upper bound on the end-to-end latency from a source to a sink is derived while modeling the effect of overlapped executions on different resources. In either context, since each task in the transaction or each node in the graph is assumed to have a worst-case execution time (WCET), the analyses give a deterministic guarantee on meeting an end-to-end deadline.

In the context of safety critical systems, several studies have been performed to analyze a graph of tasks. In [3], a probabilistic analysis is presented to analyze a single DAG (Directed Acyclic Graph) of tasks under EDF scheduling on a uniprocessor system. In [4], another analysis is proposed to address multiple DAGs under fixed-priority scheduling on a multi-core system. The above analyses give safe probabilistic response times for the DAG(s), but do not deal with a graph of task graphs with different periods, thus cannot provide a probabilistic guarantee on the end-to-end latencies. In [5], a deterministic analysis is proposed that only gives the worst-case end-to-end latency in a robotic system, thus limited to the case where $U^{\max} \leq 1.0$ for every core. Recently, in [6], a probabilistic worst-case execution time (pWCET) of each task in the Apollo stack has been analyzed by using the Extreme Value Theory. This theory [7] allows us to derive a safe pWCET distribution that upper-bounds the real distribution, while considering architectural complexities such as caches and memory buses. To guarantee the safeness, it requires that the considered input data or execution path inside the task should be general enough to represent all possible execution paths. In [7], to achieve the representativeness, the EPC (Extended Path Coverage) method is proposed that estimates and convolves the pWCETs of basic blocks within each task. In [25]. the inter-core interferences are estimated by analyzing the task graph and monitoring the usage of memory bandwidth. The analysis of pWCET, however, does not deal with how to analyze the end-to-end latencies for a graph of tasks, which possibly include idle CPU times due to the underlying task model.

Among the above studies, it is worth to more deeply review the probabilistic schedulability analysis (PSA) [4], which is similar to our analysis. The task model assumed by

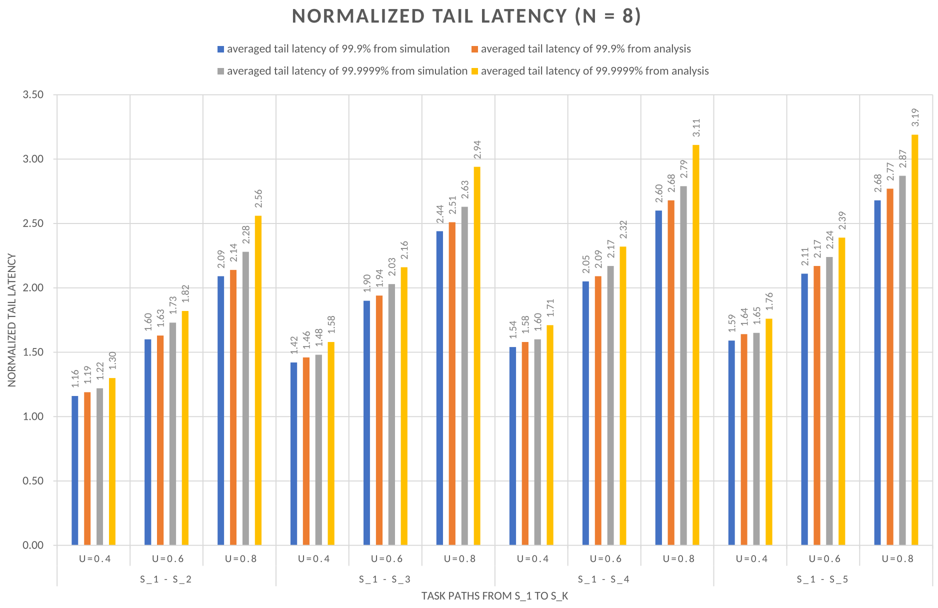


Fig. 8. Comparison between the simulated and the analyzed tail latencies from 100 random phase combinations ($N = 8$).

the PSA is a set of independent multi-rate tasks (comparable to our subgraphs), each consisting of subtasks (comparable to our tasks) with the same rate and all zero phases. The tasks are scheduled with task-level fixed priorities while allowing subtasks from different tasks with different rates to share the same CPU core. The goal of the PSA is to analyze the RTDs of individual subtasks by modeling the maximum interference caused by other subtasks from the same task and from other tasks, under the assumption of the so-called critical instant. That is, all the subtasks from different tasks release at the same time. For each task, the task-level RTD is defined as the RTD of a single sink subtask in the task.

Our analysis differs from the PSA in two aspects. First, our analysis covers an infinite sequence of the periods of the subgraphs to obtain the limiting backlog distribution on every core while the PSA does not. The PSA only computes the RTD of each subtask that assumes no backlog at the critical instant and the maximum interferences from a possibly infinite sequence of higher-priority subtasks. Thus, the resulting RTD is not guaranteed to upper-bound the limiting RTD obtained when the hyperperiod index $h \to \infty$. It is obvious that the latter upper-bounds the former because the limiting RTD reflects the limiting backlog distribution. Second, our analysis gives the limiting LD for every task path penetrating through multiple subgraphs with different periods while the PSA does not. The PSA does not consider inter-task communication. Our graph serialization and transformation to a backlog dependence graph makes it possible to accurately analyze interactions between the subgraphs.

6 Conclusion

In the paper, a stochastic analysis has been proposed that analyzes the end-to-end latency over an AV software stack running on a multi-core system. The proposed analysis is distinguished in that it addresses a graph of task graphs possibly with the maximum core utilization greater than 1.0 on each CPU core, and under the assumption of independent task execution times, it is proved that the analyzed latency distribution upper-bounds the real distribution.

To apply the analysis to an AV stack called Autoware, we performs task grouping to mitigate the effect of inter-task dependent execution times, and conducts the analysis over the graph of task groups. Through the experiments, it is shown that our analysis gives a latency distribution for each task path that almost upper-bounds the observed one from our customized Autoware with a real sensor workload. In future work, it will be addressed how to further reduce the effect of shared resources and to synthesize an optimal resource schedule for an arbitrary task graph.

References

[1] S. Kato *et al.*, “Autoware on board: Enabling autonomous vehicles with embedded systems,” in *Proc. ACM/IEEE 9th Int. Conf. Cyber-Phys. Syst.*, 2018, pp. 287–296.

[2] Apollo: Autonomous driving solution, Baidu Inc., 2017. [Online]. Available: https://apollo.auto

[3] S. Ben-Amor, D. Maxim, and L. Cucu-Grosjean, “Schedulability analysis of dependent probabilistic real-time tasks,” in *Proc. 24th Int. Conf. Real-Time Netw. Syst.*, 2016, pp. 99–107.

[4] S. Ben-Amor, L. Cucu-Grosjean, M. Mezouak, and Y. Sorel, “Probabilistic schedulability analysis for precedence constrained tasks on partitioned multi-core,” in *Proc. 25th IEEE Int. Conf. Emerg. Technol. Factory Automat.*, 2020, pp. 345–352.

[5] D. Casini, T. Blaß, I. Lütkebohle, and B. B. Brandenburg, “Response-time analysis of ROS 2 processing chains under reservation-based scheduling,” in *Proc. 31st Euromicro Conf. Real-Time Syst.*, 2019, pp. 1–23.

[6] M. Alcon, H. Tabani, L. Kosmidis, E. Mezzetti, J. Abella, and F. J. Cazorla, “Timing of autonomous driving software: Problem analysis and prospects for future solutions,” in *Proc. 2020 IEEE Real-Time Embedded Technol. Appl. Symp.*, 2020, pp. 267–280.

[7] L. Cucu *et al.*, D3.8 final report on probabilistic analysis for mixed criticality on multicore version 1.0, PROXIMA consortium, 2018 [Online]. Available: https://cordis.europa.eu/docs/projects/cnect/5/611085/080/deliverables/001-D38.pdf

[8] *VLP-16 User Manual*. San Jose, CA, USA: Velodyne LiDAR, 2018.

[9] N. Shen *et al.*, “A review of global navigation satellite system (GNSS)-based dynamic monitoring technologies for structural health monitoring,” *Remote Sens.*, vol. 11, no. 9, Apr. 2019, Art. no. 1001.

[10] Specification of operating system, AUTOSAR, 2015 [Online]. Available: https://www.autosar.org/fileadmin/user_upload/standards/classic/4–2/AUTOSAR_SWS_OS.pdf

[11] E. Talpes *et al.*, “Compute solution for tesla’s full self-driving computer,” *IEEE Micro*, vol. 40, no. 2, pp. 25–35, Mar. 2020.

[12] NVIDIA DRIVE AGX developer kit, NVIDIA corporation, 2018. [Online]. Available: https://developer.nvidia.com/drive/drive-agx

[13] H. Kang, K. Kim, and H.-W. Jin, “Real-time software pipelining for multi-domain motion controllers,” *IEEE Trans. Ind. Informat.*, vol. 12, no. 2, pp. 705–715, Apr. 2016.

[14] K. Kim, J. L. Díaz, L. LoBello, J. M. López, C.-C. Lee, and S. L. Min, “An exact stochastic analysis of priority-driven periodic real-time systems and its approximations,” *IEEE Trans. Comput.*, vol. 54, pp. 1460–1466, Nov. 2005.

[15] Soongsil university’s ichthus participated in 2019 autonomous vehicle competition, 2019 [Online]. Available: https://youtu.be/CXIlXaGeKZo.

[16] K-city: Pilot city for autonomous vehicles, korea transportation safety authority, 2018. [Online]. Available: https://youtu.be/uts6n8go1Q0.

[17] K. Tindell and J. Clark, “Holistic schedulability analysis for distributed hard real-time systems,” *Euromicro J. Microprocessing Microprogramming*, vol. 40, no. 2/3, pp. 117–134, Apr. 1994.

[18] K. Tindell and A. Burns and A. Wellings, “Calculating controller area network (CAN) message response times,” *IFAC J. Control Eng. Pract.*, vol. 3, no. 8, pp. 1163–1169, Aug. 1995.

[19] R. I. Davis and A. Burns and R. J. Bril and J. J. Lukkien, “Controller area network (CAN) schedulability analysis: Refuted, revisited and revised,” *TCCS J. Real-Time Syst.*, vol. 35, no. 3, pp. 239–272, Apr. 2007.

[20] P. Jayachandran and T. Abdelzaher, “Delay composition algebra: A. reduction-based schedulability algebra for distributed real-time systems,” in *Proc. IEEE Real-Time Syst. Symp.*, 2008, pp. 259–269.

[21] A. Saifullah, D. Ferry, J. Li, K. Agrawal, C. Lu, and C. D. Gill, “Parallel real-time scheduling of DAGs,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 12, pp. 3242–3252, Jan. 2014.

[22] J. Fonseca, G. Nelissen, and V. Nélis, “Improved response time analysis of sporadic DAG tasks for global FP scheduling,” in *Proc. 25th Int. Conf. Real-Time Netw. Syst.* 2017, pp. 28–37.

[23] M. Nasri, G. Nelissen, and B. B. Brandenburg, “Response-time analysis of limited-preemptive parallel DAG tasks under global scheduling,” in *Proc. 31st Euromicro Conf. Real-Time Syst.*, 2019, pp. 21:1–21:23.

[24] J. Sun, N. Guan, F. Li, H. Gao, C. Shi, and W. Yi, “Real-time scheduling and analysis of OpenMP DAG tasks supporting nested parallelism,” *IEEE Trans. Comput.*, vol. 69, no. 9, pp. 1335–1348, Sep. 2020.

[25] F. Guet, L. Santinelli, J. Morio, G. Phavorin, and E. Jenn, “Toward contention analysis for parallel executing real-time tasks,” in *Proc. 18th Int. Workshop Worst-Case Execution Time Anal.*, 2018, pp. 4:1–4:13.

Hyoeun Lee received the BS degree from the Department of Smart Systems Software and the MS degree from the Department of Intelligent Systems, Soongsil University, South Korea, in 2019 and 2021, respectively. Her current research interests include autonomous vehicle software and real-time systems.

Youngjoon Choi (Graduate Student Member, IEEE) received the BS degree in 2021 from the Department of Smart Systems Software, Soongsil University, South Korea, where he is currently working toward the MS degree with the Department of Intelligent Systems. His current research interests include autonomous vehicle software and high-definition maps.

Taeho Han (Graduate Student Member, IEEE) received the BS degree in 2021 from the Department of Smart Systems Software, Soongsil University, South Korea, where he is currently working toward the MS degree with the Department of Intelligent Systems. His current research interests include autonomous vehicle software and automotive embedded systems.

Kanghee Kim (Member, IEEE) received the BS, MS, and PhD degrees from the Department of Computer Engineering, Seoul National University, South Korea, in 1996, 1998, and 2004, respectively. He is currently an associate professor with the School of Artificial Intelligence Convergence, Soongsil University, South Korea. From 2004 to 2009, he was a senior engineer with Mobile Communication Division, Samsung Electronics Co., Ltd. His current research interests include real-time and embedded systems, operating systems, and autonomous vehicle computing systems.