



End-To-End Timing Analysis in ROS2

Harun Teper, Mario Günzel, Niklas Ueter, Georg von der Brüggen, Jian-Jia Chen
TU Dortmund, Department of Computer Science, Dortmund, Germany

Citation:

Preprint Version. Citation Info:

©2022 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

End-To-End Timing Analysis in ROS2

Harun Teper, Mario Günzel, Niklas Ueter, Georg von der Brüggen and Jian-Jia Chen

TU Dortmund University, Germany

{harun.teper, mario.guenzel, niklas.ueter, georg.von-der-brueggen, jian-jia.chen}@tu-dortmund.de

Abstract—Modern autonomous vehicle platforms feature many interacting components and sensors, which add to the system complexity and affect their performance. A key aspect for such platforms are end-to-end timing guarantees, which are required for safe and predictable behavior in every situation.

One widely used tool to develop such autonomous systems is the Robot Operating System 2 (ROS2), which allows creating robot applications composed of several components that communicate with each other to form complex systems. Furthermore, it guarantees real-time constraints and provides reliable timing behavior using a custom scheduler design that manages the execution of all components. These components and their data propagation form multiple cause-effect chains that can be analyzed to determine two key metrics: maximum reaction time (which is the maximum time for the system to react to an external input) and maximum data age (which equals the maximum time between sampling and the output of the system being based on that sample). However, an end-to-end analysis for cause-effect chains in ROS2 systems has not been provided yet.

In this paper, we provide a theoretical upper bound for the end-to-end timing of a ROS2 system on a single electronic control unit (ECU). Additionally, we show how to simulate a ROS2 system to get a lower bound for the timing analysis and introduce an online end-to-end timing measurement method for existing ROS2 systems. We evaluate our methods with a basic autonomous navigation system and determine the timing behavior for different components and sensor configurations.

Index Terms—End-to-End Timing, Maximum Reaction Time, Maximum Data Age, Robot Operating System 2

I. INTRODUCTION

Modern autonomous vehicles include an increasing number of components, which must comply with timing constraints to ensure correct functionality and safety of both the vehicle and its environment. They include components like smart lights, ACC, and ABS on different electronic control units (ECU) that interact over an intra-vehicle network. Additionally, there are many system designs that feature different hardware and software solutions. As the complexity of these systems increases, guaranteeing timing constraints becomes more difficult.

One widely used software framework to develop autonomous systems is the Robot Operating System 2 (ROS2) [13], which is a set of software libraries and tools to develop robot software. ROS2 allows freely creating arbitrarily complex robot systems, as it provides tools to create simple components that can communicate with each other. It also provides a customizable scheduler abstraction, called an *executor*, which processes the time-triggered and event-triggered functions of each system component. The autonomous driving project Autoware.Auto [11], [19] represents one advanced open-source solution that is based on ROS2 and features autonomous valet-parking and cargo delivery.

One promise of ROS2 is the possibility of providing real-time guarantees, which was not considered in the original Robot Operating System (ROS) [15]. The ROS2 executor has been recently analyzed and optimized [2]–[5], [18]. Specifically, the execution of ROS2 components can be modeled as a directed acyclic graph (DAG), whose worst-case response time can be derived by extending existing timing analysis methods in real-time systems [2], [4]. Moreover, Choi et al. [5] proposed a priority assignment approach for the ROS2 executor to improve the worst-case response time. Instead of modeling the ROS2 components as a DAG, Tang et al. [18] proposed to model them as processing chains to perform worst-case response time analysis and optimize the component priorities.

If all components actively trigger the components they are connected to, then the communication between ROS2 components can be described by the worst-case response time of the DAG [2], [4], [5]. However, ROS2 also features components where the data propagation and execution order are independent from other tasks. For example, a chain of tasks can include multiple intermediate timers that are sporadically executed according to their period, or include a chain whose tasks are also triggered by tasks of other chains. Such chains can in fact be more properly modeled as a cause-effect chain.

A cause-effect chain models a sequence of reactions from the cause (e.g., sensing) to an effect (e.g., actuation). The definition of cause-effect chains for a ROS 2 system is inspired by the event-chains of the AUTOSAR Timing Extensions [1], as well as the definition of cause-effect chains by Günzel et al. [10] for periodic and sporadic task systems. Two types of end-to-end latencies are of most interest in the literature: the maximum reaction time (which is the maximum time for an external event to be processed by the system), and the maximum data age (which is the maximum duration between sampling and the output being based on that sample).

However, existing timing analyses for cause-effect chains, e.g., [6], [7], [10], [12], are only valid for periodic and sporadic task systems and cannot be directly applied to the event-triggered and time-triggered components of ROS2.

Contributions: We provide, to the best of our knowledge, the first end-to-end timing analysis for ROS2 cause-effect chains. We assume that our system has one ECU, whose components are scheduled by one single-threaded ROS2 executor. We detail the ROS2 architecture in Section II and introduce the task model, cause-effect chains, and analyzed timing values in Section III. Related work and a delimitation and discussion are presented in Sections IV and V, respectively.

This paper provides the following contributions:

- Section VI provides an upper bound analysis for the end-to-end latency of cause-effect chains in ROS2 systems.
- In Section VII, we introduce a simulation method for the ROS2 scheduler to replicate the timing behavior of ROS2 systems. Additionally, we provide an online end-to-end measurement method that can be applied to existing ROS2 systems and our simulation method to measure the end-to-end timing latencies of the system.
- To compare different system configurations, we evaluate the end-to-end latencies of different systems in a case study in Section VIII. Furthermore, in Section IX, we examine the timing behavior of a basic autonomous driving system that includes a variable number of sensors.

II. ROS2 SYSTEM MODEL

ROS2 is a set of software libraries for building robot systems and applications. In this section, we explain its architecture, including the main components and interconnects in Section II-A, and the ROS2 scheduler (called ROS2 executor) in Section II-B, as described by previous works [2], [4]. Section II-C explains how to compose a robot system based on the components detailed in Section II-A.

A. Main Components and Interconnects

We consider a ROS2 system, composed of two types of components, *nodes* and *topics*.

- A *node* represents one component of the system that receives, processes, and forwards data. It consists of timers and subscriptions, as well as the functions that are executed by them. Each timer and subscription is assigned to one specific function of the node, which are called callback functions or *callbacks*.
 - Timers define *time-triggered* callbacks, which access data in the node or receive data from an external interface to process and publish messages to subscriptions or to the underlying robot platform via an external interface.
 - Subscriptions define *event-triggered* callbacks that process received messages. They may store the result in the node or publish the result to subscriptions of other nodes or to the underlying robot platform via an external interface.

We note that a node could also consist of other services, which are irrelevant to our study.

- A *topic* is used as a means of communication among different nodes. It is implemented via data-distribution services (DDS), which provide a publish-subscribe architecture for message transfer. In ROS2, a node can publish a message to a topic, so that all nodes that are subscribed to that topic receive the message.

B. ROS2 Executor Scheduling

In this paper, we consider the execution of the system on one ECU with one single-threaded *ROS2 executor* that schedules all timer and subscription callbacks of the system nodes.

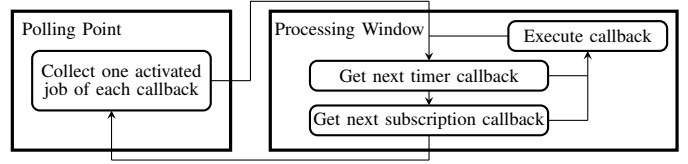


Fig. 1: ROS2 Executor Model

The ROS2 executor implements a custom scheduling policy. It stores all timers and subscriptions of all registered nodes and updates their state during the schedule. Relating this to the classical real-time scheduling theory, a callback function can be seen as a recurrent task and each execution of a callback function is a task instance (or job). A job can reside in one of the following states: *idle*, *activated*, *sampled*, and *running*. There can be several idle jobs of each task being initiated and then being activated by external events. For timers, one job is activated each time its period elapses, while a subscription is activated when it receives a new message. Each timer and subscription has a buffer with a maximum buffer size that stores the currently activated jobs of the task. We note that the jobs of a subscription process the messages in its buffer in a FIFO order. In case the buffer size is not sufficient, under-sampling or message loss may happen. We assume that the buffers are sufficiently large to avoid under-sampling or message loss in all cases.

The scheduling mechanism of the executor is designed to sample and execute jobs with a two-phase design that consists of polling points and processing windows (see Figure 1):

- At each *polling point*, the executor samples the oldest activated job of each task, which will be executed in the next processing window.
- During each processing window, the sampled jobs are executed in the order specified by the executor's scheduling policy. We assume that each timer and each subscription has a unique priority. In ROS2-Foxy (the currently supported version of ROS2), timers have higher priority than subscriptions, and the priorities of timers and subscriptions individually follow the registration order.

After all sampled callbacks are executed, the executor runs the next polling point. If there are no activated jobs at a polling point, the executor spins until an external event takes place, such as a timer period elapsing or message being received.

C. System Composition

In this subsection, we distinguish several node classes depending on the timers and subscriptions and how they communicate. In addition, we introduce how data is propagated by timers and subscriptions in and between nodes.

A ROS2 system contains nodes that include timers and subscriptions with their corresponding callbacks. Specifically, we consider six node classes, namely *sensor*, *filter*, *subscription actuator*, *timer actuator*, *subscription fusion*, and *timer fusion*. They are classified based on the means of triggering (i.e., timer, callbacks, or a mixture of them) and whether they publish to topics. Table I shows this classification.

TABLE I: Classification of ROS2 Node Classes

	Topic Publication(s)	
	YES	NO
1 Timer	Sensor	(not used here)
1 Subscription	Filter	Subscription Actuator
>1 Subscription	Subscription Fusion	(not used here)
1 Timer + ≥1 Subscription	Timer Fusion	Timer Actuator

These six classes are general enough to compose complex systems. Although we focus on these six classes, our timing analyses can be extended to further classes.

We further classify the communication and the timer and subscription composition of the node classes:

- Three types of communication through callbacks:
 - **O: Inter-node** communication takes place when a callback publishes a message to a topic.
 - **I: Intra-node** communication happens if a callback stores data in the node for other callbacks to access.
 - **E:** For **External** communication, the callback sends (or receives) data to (or from) an external interface.
- The number of timers and subscriptions can be arbitrarily chosen by the user. We assume that nodes can be comprised of the following types of components:
 - **T:** One timer.
 - **S:** One or more subscriptions.

We combine the communication types and node component types. The format represents the components of one node with one or more names that are separated by dashes. Each abbreviation XYZ includes three parts: the incoming communication type X , the node component type Y , and the outgoing communication type Z . Specifically, we define sensor, filter, and subscription actuator as follows:

- **Sensor (ETO):** A sensor receives data from an external interface, processes it, and publishes a message. It is the first element of the system that generates data.
- **Filter (OSO):** A filter receives messages, processes them, and publishes the result as a message.
- **Subscription Actuator (OSE):** A subscription actuator receives messages, processes them, and sends the result to the robot via an external interface. It is the last element of the system that processes the data.

Timer fusion, subscription fusion, and timer actuator classes include two steps with intra-node communication:

- **Timer Actuator (OSI-ITE):** A timer actuator includes one subscription that receives messages, processes them, and saves the result in the node. The timer accesses the data, processes it, and sends an output signal to the robot at the end of the execution. The timer is the last element of the system that processes the data.
- **Timer Fusion (OSI-ITO):** A timer fusion class features multiple subscriptions that receive messages, process them, and save the result in the node. The timer accesses and combines the data of all callbacks, processes it, and publishes the result as a message.

- **Subscription Fusion (OSI-ISO and OSO):** A subscription fusion class includes multiple subscriptions that receive messages, process them, and either save the result in the node or publish a message. We differentiate between two types of subscriptions, passive and trigger subscriptions. A subscription fusion class includes exactly one trigger subscription. A passive subscription receives a message, processes it, and saves the result in the node. The trigger subscription receives a message, processes it, and triggers the fusion; that is, combining the data of all passive subscriptions and its own data and publishing the result as a message. The fusion is triggered by every execution of the trigger subscription.

Based on these classes, many existing systems can be modeled and analyzed. The analysis does not depend on these specific node classes and instead only analyzes the end-to-end timing behavior of the timers and subscriptions depending on their communication type. As a result, the timing analysis is applicable for any system and the defined node classes only abstract the timers and subscriptions of the node and the intra and inter-node communication for the data propagation between the components in the node and the nodes themselves.

III. TIMING MODEL

We define the task model in Section III-A. In Section III-B, we introduce the cause-effect chains and job chains in ROS2, for which we specify the end-to-end latencies in Section III-C.

A. Task Model

The runtime of the system is managed by an executor that is responsible for dispatching and executing callbacks of a set of nodes that are registered with that executor. Each node can include several timers and several subscriptions. We assume that all nodes are registered to the same executor. This executor is running on a single ECU and schedules all callbacks. In the following we define the tasks of a ROS2 system, which are specified by timers, subscriptions, and their callbacks.

A *timer* is defined by the tuple $tmr_i = (\tau_i, k_i, pubT_i, sd_i)$, where the callback task $\tau_i = (C_i, T_i)$ is specified by its period $T_i > 0$ and worst-case execution time (WCET) $C_i \geq 0$.¹ A timer has a maximum buffer size k_i , and we assume $k_i > 1$ for all buffers. We define the current number of elements in the buffer at time t as $k_i(t) \leq k_i$, which is the number of activated jobs of task τ_i at time t . For inter-node communication, the publishing topic $pubT_i$ specifies to which topic messages are published. It can be *Null*, if the timer does not publish data; for example, if it sends the data to the robot platform instead. For intra-node communication, the subscription dependency sd_i , which is a subset of the tasks on the same node as τ_i , corresponds to the tasks that save data in the node for the callback to access. If $\tau_j \in sd_i$, then there is intra-node communication from τ_j to τ_i . As an example, sd_i can specify the subscriptions accessed by the timer in a timer fusion class.

¹We omit the phase, which determines the first activation time, as we always assume the worst case pattern for the upper bound analysis and the online end-to-end measurement method includes the effect of the phase by design.

A *subscription* is defined by the tuple $sub_i = (\tau_i, k_i, subT_i, pubT_i, sd_i)$, where the callback task $\tau_i = (C_i)$ is specified by its WCET C_i . The current number of messages in the buffer is given by $k_i(t)$, which is limited by the maximum buffer size k_i , with $k_i > 1$. The subscription subscribes to the topic $subT_i$ and may publish to the topic $pubT_i$ for inter-node communication. We assume that $\forall i, j : pubT_i \neq pubT_j$, that is, there can be only one publisher per subscription. If $pubT_i$ is *Null*, the subscription can either send the data to the robot platform or save it in the node for other callbacks. Please note that $subT_i$ is required for the triggering and therefore cannot be *Null*. Subscriptions can also include subscription dependencies sd_i ; for example if they are part of a subscription fusion class.

We denote the finite set of all callbacks included in nodes registered to the executor under analysis as \mathbb{T} . It includes the set of all timers and subscriptions registered to the executor. We denote the sum of the WCET of all tasks as $C_{sum} = \sum_{\tau_i \in \mathbb{T}} C_i$. Each task has its unique priority that is given by the function $\pi(\cdot)$. To compare the priorities between callbacks, we use the Iverson Bracket $[\cdot]$, which returns 1 or 0 if the condition is true or false, respectively. For example, we determine whether τ_i has a higher priority than τ_j with $[\pi(\tau_i) > \pi(\tau_j)]$. Each task can be scheduled in each processing window and the execution of the task τ_i in the k -th processing window is defined by the job $j_{i,k}$. Note that $j_{i,k}$ does not necessarily exist for all k , as a callback may not be included in every processing window. We denote a job's start time by $s_{i,k}$ and the finish time by $f_{i,k}$, with $f_{i,k} \leq s_{i,k} + C_i$ due to the non-preemptive execution of callbacks in ROS2.

The task model in ROS2 is different from standard periodic and sporadic task models. Specifically, tasks do not have a deadline parameter and the total utilization of all executor callbacks is not limited. For example, if the period T_i is less than the WCET C_i for a timer, the callback is processed in every processing window and the system is never idle.

B. Cause-Effect Chains

In this subsection, we introduce cause-effect chains that describe how data propagates through the components of the ROS2 system. The definition of cause-effect chains is inspired by the event-chains of the AUTOSAR Timing Extensions [1], as well as the definition of cause-effect chains by Günzel et al. [10] for periodic and sporadic task systems.

A cause-effect chain $E = (\tau_1, \dots, \tau_n)$ is a sequence of tasks, with n tasks in the chain. For the simplicity of presentation, we only focus on one cause-effect chain. For a system with more than one cause-effect chain, each chain should be analyzed and indexed independently. We note that C_{sum} is the sum of the workload of the whole system independent from the cause-effect chain under analysis. The chains of the system are formed by the communication between the callbacks. In ROS2, inter-node communication takes place if a callback publishes a message to a subscription of another node, while intra-node communication happens if callbacks of the same node access and modify the same data in the node.

The callback τ_i sends data via **inter-node communication** to the subscription callback τ_j if $pubT_i = subT_j$, so that τ_j is the successor callback of τ_i in the cause-effect chain E . In this case, each message sent by a job of τ_i triggers exactly *one* job of τ_j , and this job of τ_j processes the message in a future processing window after the message has been published.

The callback τ_j accesses data from the subscription callback τ_i via **intra-node communication** if $\tau_i \in sd_j$, so that τ_j is the successor callback of τ_i in the cause-effect chain E . In this case, the same data can be accessed *several times* by different jobs of τ_j . Additionally, the callback τ_j can access data that is modified by τ_i in the same processing window if their jobs are executed in the same processing window and $[\pi(\tau_j) < \pi(\tau_i)]$.

For each job $j_{i,k}$, we define the read event $re_{i,k}$ to be at the start time $s_{i,k}$, while the write event $we_{i,k}$ is at the finish time $f_{i,k}$. Then for ROS2 cause-effect chains, the following properties define the data propagation for the jobs:

Definition 1. (Job Chain) A job chain of E is a sequence $jc = (j_{1,\rho(1)}, \dots, j_{n,\rho(n)})$ of data dependent jobs of tasks in \mathbb{T} with the following properties:

- The entry $j_{i,\rho(i)}$ is a job of τ_i for all $i \in \{1, \dots, n\}$.
- For each inter-node dependency ($pubT_i = subT_{i+1}$), $j_{i+1,\rho(i+1)}$ is the (unique) job that processes the message from $j_{i,\rho(i)}$, i.e., $we_{i,\rho(i)} \leq re_{i+1,\rho(i+1)}$ holds.
- For each intra-node dependency ($\tau_i \in sd_{i+1}$), $j_{i+1,\rho(i+1)}$ is a job with $we_{i,\rho(i)} \leq re_{i+1,\rho(i+1)}$.

Please note that $\rho(i) < \rho(i+1)$ for inter-node dependencies, whereas $\rho(i) \leq \rho(i+1)$ for intra-node dependencies.

Analogous to Günzel et al. [10], we define forward and backward job chains to determine the end-to-end timing behavior of the system. However, since for inter-node communication there is no choice of the jobs in a job chain, only the choice of jobs contributing to intra-node communication are of special interest.

Definition 2. (Immediate Forward Job Chain) An immediate forward job chain is a job chain $jc = (j_{1,\rho(1)}, \dots, j_{n,\rho(n)})$ where $\forall i \in \{1, \dots, n-1\}$ the following applies: If $\tau_i \in sd_{i+1}$, then the job $j_{i+1,\rho(i+1)}$ is the *earliest* with $we_{i,\rho(i)} \leq re_{i+1,\rho(i+1)}$, i.e., $\rho(i+1) = \arg \min_{k \geq \rho(i)} re_{i+1,k} \geq we_{i,\rho(i)}$.

Definition 3. (Immediate Backward Job Chain) An immediate backward job chain is a job chain $jc = (j_{1,\rho(1)}, \dots, j_{n,\rho(n)})$ where $\forall i \in \{n, \dots, 2\}$ the following applies: If $\tau_{i-1} \in sd_i$, then the job $j_{i-1,\rho(i-1)}$ is the *latest* with $we_{i-1,\rho(i-1)} \leq re_{i,\rho(i)}$, i.e., $\rho(i-1) = \arg \max_{k \leq \rho(i)} we_{i-1,k} \leq re_{i,\rho(i)}$.

Next, we augment these chains to include external events, so that we can capture the duration between an external event and the sensor sampling, as well as the time between the actuation and the output of the system.

For an immediate forward job chain we denote by z the time of the external activity and by z' the time at which the data is processed. We utilize the definition by Günzel et al. [10] to augment the immediate forward job chain and define the job chain whose length corresponds to the reaction time:

Definition 4. (*Immediate Forward Augmented Job Chain*) An immediate forward job chain jc is the unique augmented job chain $(z, j_{1,\rho(1)}, \dots, j_{n,\rho(n)}, z')$, such that:

- The external activity z takes place directly after the sampling of the previous job of τ_1 before $j_{1,\rho(1)}$.
- The sampling happens at the next read-event $re_{1,\rho(1)}$.
- The sequence $(j_{1,\rho(1)}, \dots, j_{n,\rho(n)})$ is an immediate forward job chain.
- The data is processed at time $z' = we_{n,\rho(n)}$.

For an immediate backward job chain we denote by z the time of the sensor sampling and by z' the time at which an system output is based on the actuation. We augment the immediate backward job chain to determine the chain whose length corresponds to the data age:

Definition 5. (*Immediate Backward Augmented Job Chain*) An immediate backward job chain jc is the unique augmented job chain $(z, j_{1,\rho(1)}, \dots, j_{n,\rho(n)}, z')$, such that:

- The output z' takes place directly before the write event of the next job of τ_k after $j_{n,\rho(n)}$.
- The actuation happens at the previous write-event $j_{n,\rho(n)}$.
- The sequence $(j_{1,\rho(1)}, \dots, j_{n,\rho(n)})$ is an immediate backward job chain.
- The sampling occurs at time $z = re_{1,\rho(1)}$.

C. End-To-End Latencies

In this subsection, we define the end-to-end latencies that are analyzed in this paper, namely maximum reaction time and maximum data age for a specific cause-effect chain, based on the job chains introduced in Section III-B.

In ROS2, each cause-effect chain (or, in short, chain) starts with a sensor node and ends with an actuator node. Hence, the first task of the chain is a timer callback, while all other tasks are either timer or subscription callbacks. Job chains represent specific instances of a cause-effect chain; that is, the timing of processing one specific external signal.

The maximum reaction time is the maximum latency for an external signal to be processed by the actuator. For example, it is the largest interval between a user pressing a button to lock the car's doors and them actually being locked. The maximum data age corresponds to the maximum duration between a sensor sampling and an output being based on that sample. For example, it is the maximum length between the current camera image sampling and the latest time at which the steering controls are based on that camera sample. The maximum reaction time and maximum data age of a cause-effect chain correspond to the supremum over all immediate forward and backward augmented job chains, respectively.

An example system is shown in Figure 2. It includes two sensor nodes, which publish data to the corresponding sensor topics. The fusion node features one subscription per sensor and a timer that publishes data to the fusion topic. The actuator includes one subscription that subscribes to the fusion topic. The system includes two cause-effect chains, each of which consists of one sensor timer, one fusion subscription, one fusion timer, and one actuator subscription.

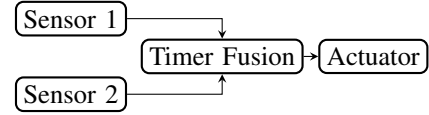


Fig. 2: A fusion system that includes two sensors, one timer fusion and one actuator node. The timer fusion consists of one subscription per sensor and one timer to publish the fusion results. The system includes two cause-effects chains.

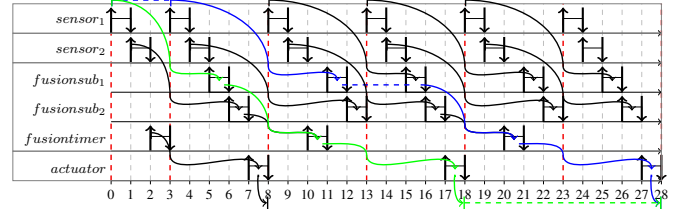


Fig. 3: Example schedule for the system shown in Fig. 2. Each box represents one node and its callbacks. The schedule includes polling points (red dashed lines) and processing windows. The arrows represent the data propagation between jobs. The reaction time (blue) and data age (green) are shown for the chain start starts from the first sensor, and includes the fusion subscription and timer, and the actuator subscription.

Figure 3 shows an example schedule for the fusion system in Figure 2. Each box specifies one node and the callbacks that are part of it. The schedule includes the polling points, which are indicated by the red dashed vertical lines, while each processing window is given by the interval between two successive polling points. In addition, the release time and finish time of each job is indicated by the upwards and downwards pointing arrows, respectively. In this schedule, the sensor timers are executed in every processing window, while the fusion timer is executed in every second processing window. For ease of presentation, all callbacks execute as long as their worst-case execution time $C_i = 1$ in this schedule.

The data propagation between the callbacks is shown by the black, green, and blue arrows. The inter-node communication takes place between the sensor and fusion subscriptions, as well as the fusion timer and actuator subscription, while the intra-node communication takes place between the fusion subscriptions and fusion timer. In this schedule, callbacks always process messages from previous processing windows.

The blue and green paths represent the immediate forward and backward augmented job chains for the reaction time and data age, respectively. As introduced in Section III-B, the blue path is an immediate forward augmented job chain $(z, j_{1,\rho(1)}, \dots, j_{n,\rho(n)}, z')$. The external event z happens right after time 0 and is sampled at time 3. The reaction time is the difference between the external event and the actuator processing it at time 28, i.e., it is $28 - 0 = 28$ in this case. Likewise, the data age corresponds to the green path, which is an immediate backward augmented job chain $(z, j_{1,\rho(1)}, \dots, j_{n,\rho(n)}, z')$. For this chain, the data is sampled at time 0 and processes new data at time 28, i.e., the data age is $28 - 0 = 28$.

IV. RELATED WORK

The Robot Operating System (ROS) [15], which is a set of software libraries for building robot applications, was released in 2007. It provides functionalities to implement basic components as nodes that communicate via topics. Additionally, there are many available packages that provide basic components of robot systems for a variety of applications, such as warehouse robots and autonomous driving. However, ROS does not natively support real-time programming. Although some extensions, such as RT-ROS [20] and ROSCH [16] extend the ROS architecture to support some real-time features, they are not widely adopted as they are not part of the core architecture.

In 2017, the first version of ROS2 was released, which built upon the core concepts of ROS, such as nodes and topics, but improved upon many aspects, such as the support for real-time code and the use of DDS [14] for secure and real-time inter-node communication. In addition, it includes a new scheduler designs for the executor that manages the execution of the time-triggered and event-triggered function of the system.

In the following years, the ROS2 executor was analyzed, including response time analysis when modeling the ROS2 components as a DAG [2], [4], or as processing chains [18]. For such systems, each component directly triggers the components they are connected to. In addition, optimization for the priority assignment of tasks was proposed by Choi et al. [5]. However, these results do not explicitly consider the data propagation between the sampling and the actuation when modeling the tasks as cause-effect chains.

End-to-end timing analysis of cause-effect chains for real-time systems have been studied over decades. In 2009, Feiertag et. al. [8] proposed the first end-to-end latency semantics to define the maximum reaction time and maximum data age. The subsequent work can be classified into two categories. So-called active approaches control the release of jobs in the chain to ensure the correctness of data reading and writing, e.g., [9], [17]. The upper bound analysis in this work is a passive approach that analyzes the end-to-end latency based on a given set of tasks and dependencies. For sporadic and periodic task systems, multiple end-to-end analysis can be found in the literature [6], [7], [10], [12]. Since these approaches are designed for periodic or sporadic task systems, they are not directly applicable to ROS2 systems, which feature a combination of time-triggered and event-triggered functions. Thus, our work is the first to consider end-to-end timing analysis for cause-effect chains in ROS2.

V. DELIMITATION AND DISCUSSION OF RELATED WORK

In this section, we emphasize and discuss the differences between our work and previous research results in the domain of end-to-end latency analyses for ROS2 systems. A summary is provided in Table II, that includes the execution model, timing metric, analyzed chain structure, communication types, number of executors, and the OS overhead. Note that SBF refers to supply bound function for the OS overhead. Additionally, we use regular expressions for the chain structure that include timers and subscriptions as T and S , respectively.

Casini et al. [4] and Blass et al. [2] both provide a worst-case response time analysis for processing chains, considering multiple executors and operating system overheads using supply-bound functions. They analyze processing chains consisting of one triggering timer at the start and multiple chained callbacks. Each callback is directly triggered by the finishing of the preceding callback, and each callback is allowed to have multiple predecessors and successors. Notably, their proposed model of computation implies that the callbacks only communicate via the publish-subscribe architecture of DDS, which corresponds to the inter-node communication in our work. The analyzed timing metric is the *maximum response time*, which is the maximum time between release of the timer at the beginning of the chain and the completion of all chain callbacks.

In contrast, the focus of our work is the temporal behavior of data propagation through the system, considering different end-to-end semantics. In particular, we guarantee that each callback execution processes the data of all chains that it is part of. In [2], [4] the maximum response time specified is the maximum time until a callback is executed. However, a subscription job only processes the oldest message in the buffer, so that a processed message may not originate from the analyzed chain if the buffer includes multiple messages from different publishers. As a result, messages may be processed in a later processing window or may be lost due to buffer overflow. To circumvent this, we limit the number of predecessors to one per subscription. We consider a single executor and do not consider any operating system overheads, as we focus on the data propagation between callbacks, including inter- and intra-node communication. On the upside, our work is more general with respect to the admissible chain structures. That is, only the first callback is limited to a timer and each subsequent callback can be either a timer or a subscription, while we do not allow consecutive timers in a cause-effect chain. This allows us to consider chains that include multiple individually triggered timers and chains with callbacks that propagate data without triggering the successor callback directly. Furthermore, the metrics are different, as our analysis does not only specify the worst-case response time, which corresponds to the maximum difference between the start time of the first chain callback and the finish time of the last chain callback. In contrast, we consider the MRT and MDA, which include the additional time from the external activity until the data sampling and the time between an actuation and the output. In comparison, the maximum response time corresponds to the maximum *reduced* reaction time as defined in [10].

In consequence of the different focuses, systems that can be analyzed by both [2], [4] and our work would consider: (1) a single executor without operating system overheads, and (2) a single chain that only includes a timer and multiple subscriptions with inter-node communication. In such a system, neither the strengths and focus of our work (i.e., different message transmission, system structure, and end-to-end semantics) nor of [2], [4] (i.e., the supply-bound function and the multiple executors) are considered. Therefore, a direct comparison would focus on a degenerate case of both analyses.

TABLE II: Related Works Contribution Overview

Related Work	Exec. Model	Timing Metric	Chain Structure	Communication	No. Executors	Overhead
Casini et al. [4]	ROS2	Max. Response Time	TS*	Inter-Node	Multiple	SBF
Blass et al. [2]	ROS2	Max. Response Time	TS*	Inter-Node	Multiple	SBF
Tang et al. [18]	ROS2	Max. Response Time	TS*	Inter-Node	Single	SBF
Several [6], [7], [10], [12]	Periodic	MRT & MDA	T*	Inter-Task	–	–
Our	ROS2	MRT & MDA	T(S ST)*	Inter-/Intra-Node	Single	–

Tang et al. [18] analyze the worst-case response time of processing chains for a model similar to the once considered by Casini et al. [4] and Blass et al. [2]. The main difference is that they consider the impact of the priority assignment for callbacks to analyze the worst-case response time. As a result, a comparison would result in the same simplification and problems as for Casini et al. [4] and Blass et al. [2].

Davare et al. [6], Dürr et al. [7], Günzel et al. [10] and Kloda et al. [12] provide a *maximum reaction time* and/or *maximum data age* analysis of cause-effect chains consisting of periodic tasks in a uniprocessor system. Their system model is incompatible with our work. In particular, only a set of periodic tasks that is scheduled in a preemptive fixed-priority manner is considered. Event-triggered execution, which is an essential component of the ROS2 architecture, is not considered in their model. Moreover, Günzel et al. [10] requires fixing the execution time to the worst case, and the worst-case response-time analysis, that is required for the maximum reaction time and maximum data age analysis in [6], [7], [12], is not applicable to the ROS2 executor/scheduler.

VI. CAUSE-EFFECT CHAIN UPPER BOUND ANALYSIS

In this section, we present how to calculate an upper bound on the maximum reaction time and maximum data age given a chain $E = (\tau_1, \dots, \tau_n)$. The system consists of the node classes, timers, and subscriptions that are introduced in Section II-C. We assume a DDS with synchronous message passing, where the execution time includes the time for publishing a message. As a result, all messages are guaranteed to be transferred when a callback finishes. Since we assume that there is only one publisher per subscription, there is at most one message published to each topic per processing window. This message is directly processed (and removed) in the *subsequent* processing window. Please note that because of $k_i > 1$ no buffer overflow can occur.

We first provide an upper bound on the maximum reaction time in Theorem 1 and that on the maximum data age in Corollary 1. Specifically, Table III presents the callbacks that are added to the corresponding cause-effect chain by the node types and which upper bound type they contribute to the total upper bound of the cause-effect chain.

Theorem 1. *Under the assumption of a DDS with synchronous message passing, τ_1 is a sensor timer task and τ_n is an actuation task, the maximum reaction time of a cause effect chain $E = (\tau_1, \dots, \tau_n)$, is at most $\sum_{i=1}^n ub(\tau_i)$, where*

TABLE III: Classification of ROS2 Callback Classes

Node class	Chain Callbacks	Upper Bound Types
Sensor	tmr_i	$ub_{st}(\tau_i)$
Filter	sub_i	$ub_{as}(\tau_i)$
Subscription Actuator	sub_i	$ub_{as}(\tau_i)$
Timer Fusion	$sub_i + tmr_j$	$ub_{ps}(\tau_i) + ub_{tt}(\tau_j)$
Subscription Fusion	sub_k or $sub_i + sub_j$	$ub_{ts}(\tau_k)$ or $ub_{ps}(\tau_i) + ub_{ts}(\tau_j)$
Timer Actuator	$sub_i + tmr_j$	$ub_{ps}(\tau_i) + ub_{tt}(\tau_j)$

$$ub(\tau_i) = \begin{cases} ub_{st}(\tau_i) & \text{if } \tau_i \text{ a sensor timer} \\ ub_{tt}(\tau_i) & \text{if } \tau_i \text{ an trigger timer} \\ ub_{as}(\tau_i) & \text{if } \tau_i \text{ an active subscription} \\ ub_{ps}(\tau_i) & \text{if } \tau_i \text{ a passive subscription} \\ ub_{ts}(\tau_i) & \text{if } \tau_i \text{ a trigger subscription} \end{cases}, \quad (1)$$

and $ub_{st}(\tau_i)$, $ub_{tt}(\tau_i)$, $ub_{as}(\tau_i)$, $ub_{ps}(\tau_i)$, and $ub_{ts}(\tau_i)$ are defined in Eqs. (4), (5), (6), (7), and (8), respectively.

Proof. As defined in Section III-B, let $j_{i,\rho(i)}$ be the job of task τ_i that is executed in the $\rho(i)$ -th processing window. Recall that $re_{i,\rho(i)}$ is the moment of its read event and also its start time, whereas $re_{i,\rho(i)} + C_i$ is the upper bound on the moment of its write event. We know that $re_{i,\rho(i)} + C_i$ is *no more* than the finishing time of the $\rho(i)$ -th processing window of the executor. There are two cases due to the non-preemptive schedule of the ROS2 executor:

- If the data of $j_{i,\rho(i)}$ is sent to the next task τ_{i+1} via inter-node communication, then the next job in the immediate forward augmented job chain is processed in the $(\rho(i) + 1)$ -th processing window. That is, $\rho(i + 1)$ is $\rho(i) + 1$.
- If the data of $j_{i,\rho(i)}$ is sent to the next task τ_{i+1} via intra-node communication, then the next job in the immediate forward augmented job chain is processed in the $\rho(i + 1)$ -th processing window, with $\rho(i + 1) \geq \rho(i)$.

Suppose θ_i is the finishing time the $\rho(i)$ -th processing window for job $j_{i,\rho(i)}$. By definition, the length of an immediate forward augmented job chain jc is

$$z' - z = \theta_1 - z + \left(\sum_{i=2}^n \theta_i - \theta_{i-1} \right) + z' - \theta_n \quad (2)$$

In the followings, we provide bounds on each of these terms:

Bound on $z' - \theta_n$: Since τ_n is an actuator by assumption, we know that z' is equal to $we_{n,\rho(n)} \leq \theta_n$. Therefore, $z' - \theta_n \leq 0$.

Bound on $\theta_1 - z$: Due to the assumption that there is always a timer at the beginning, E starts with a sensor timer task τ_1 .

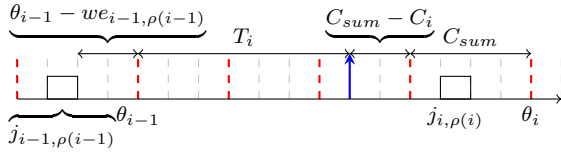


Fig. 4: Example schedule for Lemma 1. The blue arrow marks release of job $j_{i,\rho(i)}$, dotted red lines represent polling points.

For the upper bound of $\theta_1 - z$, the external event z happens right after the last sensor timer job $j_{1,k}$ before $j_{1,\rho(1)}$ finishes. After that, the timer callback is sampled after its period T_1 elapses. In the worst case, its period elapses right after a processing window starts that includes every callback except τ_1 , resulting in an additional delay of $C_{sum} - C_1$. Afterwards, the timer is executed together with every callback, resulting in a delay of C_{sum} . As a result, $\theta_1 - z$ is upper bounded by:

$$\theta_1 - z \leq T_1 - C_1 + 2 \cdot C_{sum} \quad (3)$$

We denote this as the **Sensor Timer** upper bound for task τ_i :

$$ub_{st}(\tau_i) = T_i - C_i + 2 \cdot C_{sum} \quad (4)$$

Bound on $\theta_i - \theta_{i-1}$: To improve readability, we show in Lemmas 1 to 4, that $\theta_i - \theta_{i-1}$ is upper bounded by $ub_{tt}(\tau_i)$, $ub_{as}(\tau_i)$, $ub_{ps}(\tau_i)$, and $ub_{ts}(\tau_i)$ when τ_i is a trigger timer, an active subscription, a passive subscription, and a trigger subscription, respectively. Combining these results with the argumentation so far proves the theorem. \square

Lemma 1 (Trigger Timer). *If τ_i is a trigger timer task, then $\theta_i - \theta_{i-1}$ in Eq. (2) is at most*

$$ub_{tt}(\tau_i) = T_i - C_i + 2 \cdot C_{sum} \quad (5)$$

Proof. Due to the ROS2 executor design, timers are executed before subscriptions in a processing window. As specified in Table III, the job $j_{i-1,\rho(i-1)}$ is a subscription job. Therefore, the trigger timer job $j_{i,\rho(i)}$ can not be executed before θ_{i-1} . Additionally, the job $j_{i-1,\rho(i-1)}$ is guaranteed to finish before θ_{i-1} , so that $we_{i-1,\rho(i-1)} \leq \theta_{i-1}$. At the earliest, the job $j_{i,\rho(i)}$ is sampled at θ_{i-1} , so that it is guaranteed to finish no later than $\theta_{i-1} + C_{sum}$. At the latest, as illustrated in Figure 4, the timer elapses after $\theta_{i-1} + T_i$ right after a processing window starts that includes every callback except τ_i , resulting in an additional delay of $C_{sum} - C_i$. Therefore, the job $j_{i,\rho(i)}$ is sampled no later than $\theta_{i-1} + T_i + C_{sum} - C_i$. The processing window $\rho(i)$ of job $j_{i,\rho(i)}$ has a maximum length of C_{sum} . As a result, $\theta_i - \theta_{i-1} \leq T_i - C_i + 2 \cdot C_{sum}$. \square

Active subscriptions receive messages, process them, and send the result to the robot platform or publish a message. This case only covers the subscription of the subscription actuator class and filter classes, but not fusion or timer actuator classes.

Lemma 2 (Active Subscription). *If τ_i is an active subscription task, then $\theta_i - \theta_{i-1}$ in Eq. (2) is at most*

$$ub_{as}(\tau_i) = C_{sum} \quad (6)$$

Proof. Since the subscription always processes a message in the following processing window in the ROS2 system model, $\rho(i)$ is equal to $\rho(i-1) + 1$ in this case. Therefore, by definition, each processing window takes at most C_{sum} of time and $\theta_i - \theta_{i-1} \leq C_{sum}$ for this case. \square

For timer fusion, subscription fusion, and timer actuators, such passive subscriptions only receive, process, and save the result in the node, so that it can be accessed by the timers and the trigger subscription of these classes.

Lemma 3 (Passive Subscription). *If τ_i is a passive subscription task, then $\theta_i - \theta_{i-1}$ in Eq. (2) is at most*

$$ub_{ps}(\tau_i) = C_{sum} \quad (7)$$

Proof. The proof is identical to the proof of Lemma 2. \square

Trigger subscriptions combine the data of all subscriptions in the node and send the result as a message. A subscription fusion node consists of multiple subscriptions, including one trigger subscription. The fusion is triggered every time the trigger subscription is executed, which happens every time it processes a received message of its predecessor task, which may be part of the chain E or a different chain in the system.

Lemma 4 (Trigger Subscription). *If τ_i is a trigger subscription task, then $\theta_i - \theta_{i-1}$ in Eq. (2) is at most*

$$ub_{ts}(\tau_i) = \begin{cases} C_{sum} & \text{if } \tau_{i-1} \notin sd_i \\ \sum_{\bar{\tau}_k \in \bar{E}} ub(\bar{\tau}_k) + C_{sum} & \text{if } \tau_{i-1} \in sd_i \end{cases} \quad (8)$$

where the chain $\bar{E} = (\bar{\tau}_1, \dots, \bar{\tau}_m)$ ²³ triggers task τ_i i.e., task $\bar{\tau}_1$ is a timer, $(\bar{\tau}_2, \dots, \bar{\tau}_m)$ are subscriptions, and $\text{pub}T_m = \text{sub}T_i$.

Proof. We consider two cases, depending on whether the task τ_{i-1} is in sd_i : the task τ_i accesses data of τ_{i-1} via intra-node communication if $\tau_{i-1} \in sd_i$, or τ_i receives data via inter-node communication if $\tau_{i-1} \notin sd_i$.

Case 1: Inter-node: If $\tau_{i-1} \notin sd_i$, τ_{i-1} and τ_i communicate via inter-node communication and the subscription is processed in the next processing window, i.e., $\rho(i)$ is equal to $\rho(i-1) + 1$, leading to $\theta_i - \theta_{i-1} \leq C_{sum}$.

Case 2: Intra-node: If $\tau_{i-1} \in sd_i$, then there is intra-node communication between τ_{i-1} and τ_i , and $j_{i,\rho(i)}$ is not directly triggered by $j_{i-1,\rho(i-1)}$ but by a job of $\bar{\tau}_m$.

To find such a job that triggers $j_{i,\rho(i)}$ (if it was not triggered before), let $t \in \mathbb{N}$ such that $\bar{j}_{1,t}$ is the first job of $\bar{\tau}_1$ executed after θ_{i-1} . Then the following holds:

- $\bar{re}_{1,t-1} \leq \theta_{i-1}$
- The immediate forward chain starting at $\bar{j}_{1,t}$ ends at a job $\bar{j}_{m,q}$ of $\bar{\tau}_m$, with $q \in \mathbb{N}$, that triggers $j_{i,\rho(i)}$ (if it was not triggered before).

Hence, $j_{i,\rho(i)}$ is executed in the $(q+1)$ -th processing window.

²Please note that the indices of this chain \bar{E} are unrelated to the chain E under analysis and that they do not share callbacks.

³Each callback in \bar{E} is triggered by inter-node communication. Starting from τ_i , \bar{E} can be constructed by adding the preceding publisher callback to the head of the chain until a timer has been added.

The maximum reaction time of the chain \bar{E} covers the time from θ_{i-1} until $\overline{we}_{m,q}$. The upper bound $\sum_{\bar{\tau}_k \in \bar{E}} ub(\bar{\tau}_k)$ covers also the time from θ_{i-1} until the end of the processing window of job $\bar{j}_{m,q}$. Moreover, the length of the $(q+1)$ -th processing window is upper bounded by C_{sum} . Therefore, $\theta_i - \theta_{i-1} \leq \sum_{\bar{\tau}_k \in \bar{E}} ub(\bar{\tau}_k) + C_{sum}$. \square

Corollary 1. *Under the same assumption of Theorem 1, the maximum data age is upper bounded by $\sum_{i=1}^n ub(\tau_i)$, where $ub(\tau_i)$ is defined in Eq. (1).*

Proof. Let τ_k be the last timer task in the cause-effect chain. By definition, such a task τ_k exists, as the first task τ_1 is a sensor timer task. Therefore, $1 \leq k \leq n$. By the definition of an immediate backward augmented job chain, z' happens directly before the next write event of the next job of τ_n after $\bar{j}_{n,\rho(n)}$. By definition, the length of an immediate backward augmented job chain jc is

$$z' - z = \psi_1 - z + \left(\sum_{i=2}^k \psi_i - \psi_{i-1} \right) + (z' - \psi_k)$$

with ψ_i being the beginning of the $\rho(i)$ -th processing window. A similar proof as for Theorem 1 can be done to show that $\psi_1 - z + \left(\sum_{i=2}^k \psi_i - \psi_{i-1} \right) \leq \sum_{i=1}^{k-1} ub(\tau_i)$.

Here, we only sketch the proof of $z' - \psi_k$. Since τ_k is the latest timer of the cause-effect chain, the time between the write event $we_{k,\rho(k)}$ and the next write event of τ_k is upper bounded by $T_k - C_k + 2 \cdot C_{sum}$. Furthermore, $\tau_{k+1}, \tau_{k+2}, \dots, \tau_n$ are only subscription tasks. Therefore, the jobs of $\tau_{k+1}, \tau_{k+2}, \dots, \tau_n$ in a backward augmented job chain are executed in consecutive processing windows. Hence, $z' - \psi_k \leq (n-k)C_{sum} + T_k - C_k + 2 \cdot C_{sum} = ub_{tt}(\tau_k) + \sum_{i=k+1}^n ub(\tau_i)$, where $ub(\tau_i)$ is either $ub_{as}(\tau_i)$ or $ub_{ts}(\tau_i)$ for $i = k+1, k+2, \dots, n$. Thus, we reach the conclusion. \square

VII. ONLINE END-TO-END ANALYSIS

The upper bound analysis in Section VI is pessimistic compared to the real end-to-end latencies of the system, as it assumes the worst-case execution pattern in all cases. In Section VII-A, we introduce a method to simulate ROS2 systems with a single executor to determine a lower bound on the timing values.⁴ Additionally, we introduce an online end-to-end timing measurement method in Section VII-B to determine the maximum reaction time and the maximum data age of all cause-effect chains. This method can be applied to existing ROS2 systems and our simulation approach to compare our lower bound and the real observed values.

A. Executor Simulation

This subsection introduces how to simulate the executor and callback execution of a ROS2 system on a single ECU.

⁴We only consider the execution scenario by fixing the execution time of each task to its worst-case execution time. For this scenario we observe the exact maximum reaction time and maximum data age. However, since this does not cover all possible execution scenarios, the measured values are just lower bounds for the general case.

Algorithm 1 Executor Simulation

```

1: procedure SIMULATEEXECUTORPROCESSINGWINDOW
2:   updateTimerBuffers()
3:   readyTimers = getReadyTimers()
4:   readySubscriptions = getReadySubscriptions()
5:   pastPollingPoints.append(now)
6:   executeTimers(readyTimers)
7:   executeSubscriptions(readySubscriptions)
8:   if length(getReadyCallbacks()) == 0 then
9:     skipToNextPollingPoint()

```

For the simulation, we define a system state that consists of the global time, the past polling points, and the nodes that consist of multiple callbacks. During the simulation, we simulate the execution of callbacks by elapsing the global time by the execution time of each callback, and updating the buffer states of all timer and subscription callbacks.

The simulation of the system is detailed in Algorithm 1. We simulate the schedule considering the global time and past polling point times. Line 2 updates the activation buffer $k_i(t)$ for each timer. The function determines the difference in activations between the previous and current polling point and adds it to the timer buffer. Lines 3-4 collect all callbacks with a non-empty buffer, after which Line 5 adds the current time to the past polling points. Lines 6-7 execute all timer and subscription callbacks, given their priority order. For each callback, the global time is advanced by its execution time. Lines 8-9 skip to the next polling point and advance the global time to the next callback activation if no job is activated.

B. Online Timing Measurement

In this subsection, we introduce an online end-to-end timing measurement method to determine the end-to-end timing latencies of existing ROS2 systems and of the simulation approach that is presented in the previous subsection. For the timing measurement, we introduce a new message header, which is a message itself, that can be added to existing message types in ROS2 and stores the timing information to calculate the reaction time and data age of all job chains. A message contains a list of entries, each of which represents one job chain. Each entry contains a unique identifier, the reaction time origin date, and the data age origin date. The unique identifier is a list of callbacks that processed the message so far. Each callback adds its unique identifier, such as a name, to the history during its execution. The origin dates for the reaction time and data age are required to determine the maximum reaction time and maximum data age and are set once by the sensor timers during their execution. For each sensor timer execution $\bar{j}_{i,k}$, the data age origin date corresponds to the start time $s_{i,k}$. The reaction time origin date is the start time of the previous execution $s_{i,k-1}$, as shown in Figure 3.

With this approach, each callback determines the end-to-end latencies of all chains that end with the callback itself. As a result, the maximum reaction time and maximum data age of all sub-chains of the system are measured during execution. Furthermore, the end-to-end latencies of the complete chains are stored in the actuator callbacks of the system.

Algorithm 2 Message Creation

```

1: procedure CREATEMESSAGE
2:   if type == Timer then
3:     msg = Message()
4:   else
5:     msg = DDSBuffer.pop()
6:   for forwardMessage  $\in$  forwardBuffer do
7:     for entry  $\in$  forwardMessage do
8:       msg.addEntry(entry)
9:   forwardBuffer.clear()
10:  return msg

```

Algorithm 3 Subscription Execution

```

1: procedure EXECUTESUBSCRIPTION
2:   now += wcet
3:   updateAnalysisDataAge()
4:   msg = createMessage()
5:   updateLatestMessage(msg)
6:   updateMessageRegisters()
7:   updateAnalysisRegisters()
8:   forwardMessage(msg)
9:   sendMessage(msg)

```

For each callback, we add two register types, namely the message and analysis registers for each timing value, resulting in four different registers, as well as a forward buffer that stores messages of subscription dependencies for subscription fusion, timer fusion, and timer actuator classes.

The message registers store the entries of processed messages. Each time a callback is executed and processes a message, it updates the origin dates of existing entries with the message entries. The analysis register saves the maximum reaction time and maximum data age of all entries, which is the maximum difference between the current execution time and the origin dates of the processed entries. During each callback execution, the analysis register values are updated with the entries that are stored in the message register, including new entries that are stored in the processed message. The forward buffer is used for intra-node communication and stores messages that are forwarded by subscription dependencies. In general, reaction time is updated before a send operation, while data age is updated before receive and send operations.

Algorithm 2 details the message creation for all callbacks. For timers, Lines 2-3 create a new empty message, while subscriptions take the newest message in the buffer in Lines 4-5. For fusion nodes and timer actuators, Lines 6-9 collect all entries from forwarded messages of subscription dependencies, after which the forward buffer is cleared.

The next part is the subscription execution, which is detailed in Algorithm 3. First, Line 2 executes the function of the subscription. We assume each callback executes according to its WCET. Line 3 updates the maximum data age of the analysis registers with the entries that are saved from the previous executions. Lines 4-5 create and update the latest message. After that, Line 6 updates the message registers with the entries of the new message. Then, Line 7 updates the analysis registers for both the maximum reaction time and maximum data age with the new message register contents.

Algorithm 4 Timer Execution

```

1: procedure EXECUTETIMER
2:   activationBuffer--
3:   msg = createMessage()
4:   if sdi == None then
5:     if hasLatestMessage() then
6:       msg.addEntry(last, now, [id])
7:     else
8:       msg.addEntry(now, now, [id])
9:   now = now+wcet
10:  updateAnalysisDataAge(now)
11:  updateLatestMessage(msg)
12:  updateMessageRegisters()
13:  updateAnalysisRegisters(now)
14:  sendMessage(msg)

```

Line 8 forwards the message to all callbacks that include the subscription as a subscription dependency. Additionally, it adds the forward callback to the history of all entries of the forwarded message. Finally, Line 9 sends a message so that the subscription with sub_j with $subT_j = pubT_i$ receives it.

Next is the execution of timers, which are part of sensors, timer fusion, and timer actuators. Timers cannot directly process received messages, but instead create new messages from sensor data or process data from subscription dependencies.

Algorithm 4 details the timer execution and the updating process for the timing values. Line 2 decreases the activation buffer, as the timer is executed. Lines 3-8 create the message. For fusion and actuator timers, Line 3 collects the messages from all subscription dependencies that are stored in the forward buffer. For sensor timers, Lines 4-8 create the first entry of the message. For the first timer execution, the current time now is used for both reaction time and data age origin dates. For all other executions, the data age origin date is the current time now , while the reaction time origin date is the start time of the previous execution $last$. Line 9 represents the callback execution. We assume each callback executes as long as its WCET. After that, the timing analysis values are updated for the analysis and message registers.

The data age analysis register is updated with the previous message register entries in Line 10. Then, the latest message is updated in Line 11. The new message is used in Line 12 to update the entries in the message registers. After that, Line 13 updates the analysis registers, including the maximum reaction time and maximum data age using the new values in the message register. Finally, Line 14 publishes the message, so that subscriptions sub_j with $subT_j = pubT_i$ receive it.

With this method, existing ROS2 systems can measure the end-to-end timing values of all chains. After the execution, each callback includes an entry for each chain in its analysis register that ends with the callback itself. The maximum reaction time and maximum data age of each complete cause-effect chain is stored in the analysis registers of each actuator callback. In addition, each entry includes a list of the callbacks that are part of the chain. This method can be applied to the executor simulation in Subsection VII-A and to existing ROS2 systems to measure the end-to-end timing latencies.

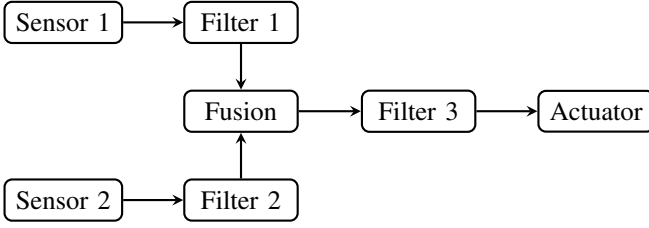


Fig. 5: Fusion system with two chains as case-study.

TABLE IV: Analysis for over-utilized system

System	Chain	C_{chain} [ms]	Upper bound [ms]	Lower Bound [ms]	Observed [ms]
subscription fusion + subscription actuation	Chain1	110.0	1160.0	1080.0	1090.0
	Chain2	160.0	1950.0	1070.0	1080.0
subscription fusion + timer actuation	Chain1	140.0	1797.5	1320.0	1330.0
	Chain2	190.0	2722.5	1310.0	1320.0
timer fusion + subscription actuation	Chain1	140.0	1797.5	1470.0	1480.0
	Chain2	160.0	1787.5	1460.0	1470.0
timer fusion + timer actuation	Chain1	170.0	2570.0	1770.0	1780.0
	Chain2	190.0	2560.0	1760.0	1770.0

VIII. CASE STUDY

We examine the end-to-end latencies of the system in Figure 5. Specifically, we calculate the upper bounds based on the analysis in Section VI, and apply the simulation according to Section VII-A as well as the online measurement method from Section VII-B to the real system in ROS2.

The system in Figure 5 consists of two sensors, three filters, one fusion class, and one actuator class. The sensors and filters are implemented with the sensor timer and filter classes from Section II-C. For the fusion and actuator node, we compare the timing behavior between the subscription-based and timer-based types of each class. As a result, we evaluate four different system types. For each class, the number of callbacks per chain changes, which we specify for each chain in the analysis. For example, a subscription actuator only includes one callback, while a timer actuator includes two for each chain that it is part of. We further configure the subscription fusion class to utilize the subscription of *filter1* as the trigger subscription, while *filter2* is a passive subscription.

For each component, we set the execution time of the callbacks and the period of the timers. We consider two different cases, specifically an *over-utilized system* and an *under-utilized system*. Let C_{sum} be the sum of all callback execution times. For an over-utilized system, $C_{sum} > T_i$ for all timers tmr_i , while $C_{sum} < T_i$ for an under-utilized system. For an over-utilized system, the periods of the timers are set to $T_i = \frac{C_{sum}}{4}$ for sensor class nodes and to $T_i = \frac{C_{sum}}{2}$ for the timer fusion and timer actuator. For an under-utilized system, we set the period to $T_i = C_{sum} \cdot 2$ for the sensors and $T_i = C_{sum} \cdot 4$ for timer fusion and timer actuators. For each sensor and filter, we set the execution time such that $C_{sensor1} = C_{filter1} = 10$ ms, $C_{sensor2} = C_{filter2} = 20$ ms, and $C_{filter3} = 30$ ms. For the fusion block, we set the execution time of all timer and subscription callbacks τ_i to $C_i = 30$ ms. We assume each callback executes as long as its WCET and all callbacks have a buffer size of $k_i > 1$.

TABLE V: Analysis for under-utilized system

System	Chain	C_{chain} [ms]	Upper bound [ms]	Lower Bound [ms]	Observed [ms]
subscription fusion + subscription actuation	Chain1	110.0	1430.0	540.0	550.0
	Chain2	160.0	2490.0	530.0	540.0
subscription fusion + timer actuation	Chain1	140.0	2900.0	1320.0	1330.0
	Chain2	190.0	4140.0	1310.0	1320.0
timer fusion + subscription actuation	Chain1	140.0	2900.0	1470.0	1480.0
	Chain2	160.0	2890.0	1460.0	1470.0
timer fusion + timer actuation	Chain1	170.0	4730.0	2490.0	2500.0
	Chain2	190.0	4720.0	2480.0	2490.0

Table IV and Table V display the timing values of the system types for the over-utilized and under-utilized case, respectively. We further denote the sum of the execution times of the tasks along a chain as C_{chain} in Tables IV and V. For each chain, the C_{chain} column specifies the sum of its callback execution times. The upper bounds and lower bounds are derived based on the methods presented in Sections VI and VII-B, respectively. For the lower bound and observed values, the last callback of the actuator stores the timing values of both chains and are read out after the timing values remain static. In addition, the timing values of the lower bound and observed columns correspond to the maximum reaction time and the maximum data age, as we observed these values to be equal for every system and for all executions.

As shown in Table IV, the system types have different timing behaviors, with the subscription-based system having the best and the timer-based systems having the worst performance for the lower bound and observed values. The observed timing values of both chains are very similar and the lower bound of the simulated system is almost identical to the lower bound, but includes the overhead of running ROS2. For every system type, the upper bound exceeds the observed values by a large margin, as it assumes the worst-case execution pattern in every case. Additionally, the second chain that includes a passive subscription of a subscription fusion class has a very pessimistic upper bound compared to timer fusion classes. We could not observe a relation between the chain execution time and the timing behavior, as the chains directly effect each other when running on the same system.

Table V shows that an under-utilized system has a similar behavior, with timer-based systems performing worse than subscription-based ones. In addition, the margin between the upper bound and lower bound is much larger for each chain compared to the over-utilized system. This is due to the idle time of timer callbacks, which could result in large delays between message transfers. Similar to an over-utilized system, the observed values of an under-utilized system are close to the lower bound of the simulated system, and subscription fusion classes have a high upper bound for the second chain.

IX. EVALUATION

As shown by the case study in Section VIII, the timing behavior of the under-utilized and over-utilized system is very different. In this evaluation, we further investigate this behavior and also determine the timing behavior of ROS2 systems when changing the number of components.

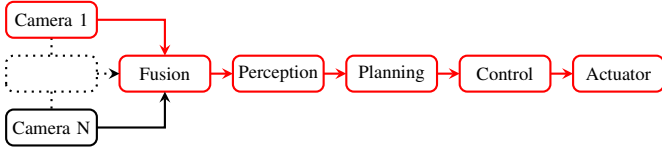


Fig. 6: Navigation system with variable number of cameras to evaluate the timing behavior of the chain highlighted in red.

In this section, we consider the basic autonomous driving system shown in Figure 6. It includes N cameras, a fusion, a perception, a planning, a control, and an actuator node. The cameras are sensor nodes, while the fusion node is implemented with a subscription fusion class. The actuator is a subscription actuator, while the perception, planning, and control nodes are implemented as filters. We determine the end-to-end timing latencies for the chain that is highlighted in red. It includes a static amount of callbacks, which include the sensor timer, one passive fusion subscription, one fusion timer, three filter subscriptions and one actuator subscriptions. We observe the timing behavior of this chain while changing the number of cameras of system. Each camera adds one sensor timer node and one passive fusion subscription to the system.

The system includes N cameras for each case. For the sensors, we use a timer with a period of $100ms$ and a WCET of $5ms$. The sensor fusion includes one subscription per camera, each with a WCET of $5ms$. For each camera, the total WCET of the added callbacks, consisting of the the sensor node timer callback and fusion node passive subscription, is $10ms$. For the perception, the planning, the control, and the actuator subscriptions, the respective WCET is $10ms$. In total, the WCET of the filters and actuators is $40ms$. As a result, the system is under-utilized if the number of cameras is less than six, perfectly utilized if six cameras are included, and over-utilized if there are more than six cameras.

As shown in Figure 7, the timing behavior is very different for the under-utilized and over-utilized cases when changing the number of cameras. Over all cases, the upper bound is linearly increasing with the number of cameras, while the lower bound has two distinct patterns. If $N \leq 5$, the system is under-utilized and for each added camera an additional latency of $10ms$ is added, which corresponds to the added WCET per camera, including the sensor timer callback and passive fusion subscription. For $N = 6$, the system is perfectly utilized and the observed lower bound is much smaller than the observed values. This can be explained by the overhead when running the system, which leads to the system being over-utilized and performing like an over-utilized system. For $N \geq 7$, the observed values and lower bound perform like an over-utilized system. In addition, the lower bound and observed values are very similar to the upper bound values for this chain. For the over-utilized chain, the timing value increases by $70ms$ per added camera, as every callback is executed in every processing window and the chain consists of six callbacks and also needs to process the external event with one additional processing window.

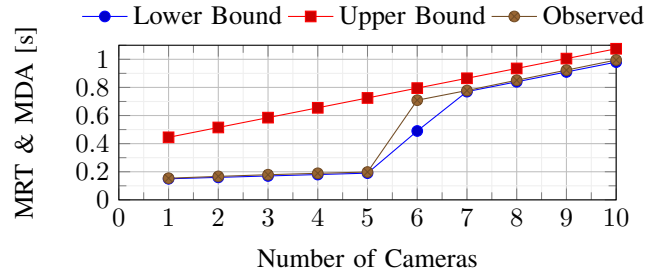


Fig. 7: End-to-end latencies of the system in Figure 6 when varying the number of cameras. The graphs show the values for both maximum reaction time and maximum data age, as these are equal for this system.

As shown in the evaluation, the under-utilized system performs much better than the over-utilized system. Therefore, it is important to consider the periods of all timers and the execution times of all callbacks that are registered to the executor. Additionally, we observe that our system does not leave the over-utilized case once it has reached it; hence, it is important that this state is never reached. For example, this can be done by guaranteeing that the sum of all execution times does not exceed the periods of each timer or by determining the execution patterns so that it does not transition to this state.

X. CONCLUSION

In this paper, we explored the end-to-end timing analysis of ROS2 systems that are executed on a single ECU with a single-threaded executor. We introduce the main components that are part of the ROS2 architecture and form the systems. Based on these components, we introduce node classes, which specify the composition of these components and how they propagate data through the system. After that, we provide an upper bound analysis to determine the maximum end-to-end latencies of these systems. Additionally, we provide a simulation method to replicate the schedule when executing the system on a single executor. We introduce a measurement method to determine the end-to-end timing behavior when running ROS2 systems and when simulating the systems with our method.

The analysis shows that ROS2 systems perform vastly different depending on the utilization of the system. An under-utilized system performs much better than an over-utilized system and there is large margin between the observed values and the upper bound for the under-utilized case. In addition, ROS2 systems need to be guaranteed to never reach an over-utilized state, as they do not leave that state once it has been reached. The results demonstrate that the end-to-end analysis is an important aspect that needs to be considered when designing systems in ROS2. As the user can freely configure each system in ROS2, this analysis can provide insights on the expected end-to-end behavior for each configuration.

We plan to extend our analysis to systems that consist of multiple ECUs. Furthermore, we want to improve the scheduler design and provide guidelines for creating ROS2 systems that improve the general timing behavior.

ACKNOWLEDGMENTS

This work has received funding by the German Federal Ministry of Education and Research (BMBF) in the course of the 6GEM research hub under grant number 16KISK038.

REFERENCES

- [1] AUTOSAR. Specification of timing extensions, November 2020. release R20-11.
- [2] T. Blass, D. Casini, S. Bozhko, and B. B. Brandenburg. A ros 2 response-time analysis exploiting starvation freedom and execution-time variance. In *Proceedings of the 42nd Real-Time Systems Symposium (RTSS)*, 2021.
- [3] T. Blass, A. Hamann, R. Lange, D. Ziegenbein, and B. B. Brandenburg. Automatic latency management for ros 2: Benefits, challenges, and open problems. In *Proceedings of the 27th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2021.
- [4] D. Casini, T. Blass, I. Lütkebohle, and B. B. Brandenburg. Response-time analysis of ros 2 processing chains under reservation-based scheduling. In *Proceedings of the 31st Euromicro Conference on Real-Time Systems (ECRTS)*, 2019.
- [5] H. Choi, Y. Xiang, and H. Kim. Picas: New design of priority-driven chain-aware scheduling for ROS2. In *27th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2021, Nashville, TN, USA, May 18-21, 2021*, pages 251–263. IEEE, 2021.
- [6] A. Davare, Q. Zhu, M. Di Natale, C. Pinello, S. Kanajan, and A. Sangiovanni-Vincentelli. Period optimization for hard real-time distributed automotive systems. In *Proceedings of the 44th Annual Design Automation Conference*, page 278–283, 2007.
- [7] M. Dürr, G. von der Brüggen, K.-H. Chen, and J.-J. Chen. End-to-end timing analysis of sporadic cause-effect chains in distributed systems. *ACM Trans. Embed. Comput. Syst.*, 18(5s), oct 2019.
- [8] N. Feiertag, K. Richter, J. E. Nordlander, and J. Å. Jönsson. A compositional framework for end-to-end path delay calculation of automotive systems under different path semantics. In *RTSS 2009*, 2008.
- [9] A. Girault, C. Prévot, S. Quinton, R. Henia, and N. Sordon. Improving and estimating the precision of bounds on the worst-case latency of task chains. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2578–2589, 2018.
- [10] M. Günzel, K. Chen, N. Ueter, G. von der Brüggen, M. Dürr, and J. Chen. Timing analysis of asynchronized distributed cause-effect chains. In *27th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2021, Nashville, TN, USA, May 18-21, 2021*, pages 40–52. IEEE, 2021.
- [11] S. Kato, S. Tokunaga, Y. Maruyama, S. Maeda, M. Hirabayashi, Y. Kit-sukawa, A. Monrroy, T. Ando, Y. Fujii, and T. Azumi. Autoware on board: Enabling autonomous vehicles with embedded systems. In *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPs)*, pages 287–296, April 2018.
- [12] T. Kloda, A. Bertout, and Y. Sorel. Latency analysis for data chains of real-time periodic tasks. In *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*, volume 1, pages 360–367, 2018.
- [13] Open Robotics. Ros2: Foxy, May 2022. <https://docs.ros.org/en/foxy>.
- [14] G. Pardo-Castellote. Omg data-distribution service: architectural overview. In *23rd International Conference on Distributed Computing Systems Workshops, 2003. Proceedings.*, pages 200–206, 2003.
- [15] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Ng. Ros: an open-source robot operating system. volume 3, 01 2009.
- [16] Y. Saito, F. Sato, T. Azumi, S. Kato, and N. Nishio. Rosch:real-time scheduling framework for ros. In *2018 IEEE 24th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 52–58, 2018.
- [17] J. Schlatow and R. Ernst. Response-time analysis for task chains in communicating threads. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–10, 2016.
- [18] Y. Tang, Z. Feng, N. Guan, X. Jiang, M. Lv, Q. Deng, and W. Yi. Response time analysis and priority assignment of processing chains on ROS2 executors. In *41st IEEE Real-Time Systems Symposium, RTSS 2020, Houston, TX, USA, December 1-4, 2020*, pages 231–243. IEEE, 2020.
- [19] The Autoware Foundation. Autoware, 2022. <https://www.autoware.org/>.
- [20] H. Wei, Z. Shao, Z. Huang, R. Chen, Y. Guan, J. Tan, and Z. Shao. Rt-ros: A real-time ros architecture on multi-core processors. *Future Gener. Comput. Syst.*, 56:171–178, 2016.

NS-FTL: Alleviating the Uneven Bit-Level Wearing of NVRAM-based FTL via NAND-SPIN

Wei-Chun Cheng, Shuo-Han Chen, Yuan-Hao Chang, Kuan-Hsun Chen*, Jian-Jia Chen*,
Tseng-Yi Chen[†], Ming-Chang Yang[‡], Wei-Kuan Shih[§]

Institute of Information Science, Academia Sinica, Taipei, Taiwan

*TU Dortmund, Department of Computer Science, Dortmund, Germany

[†]Department of Computer Science and Information Engineering, National Central University, Taoyuan, Taiwan

[‡]The Chinese University of Hong Kong, Department of Computer Science and Engineering, Hong Kong

[§]National Tsing Hua University, Department of Computer Science, Hsinchu, Taiwan

e-mail: {weichun, qoolili, johnson}@iis.sinica.edu.tw, *{kuan-hsun.chen,jian-jia.chen}@cs.uni-dortmund.de,

[†]tychen@g.ncu.edu.tw, [‡]mcyang@cse.cuhk.edu.hk, [§]wshih@cs.nthu.edu.tw

Abstract—Non-Volatile random access memory (NVRAM) has been regarded as a promising DRAM alternative with its nonvolatility, near-zero idle power consumption, and byte addressability. In particular, some NVRAM devices, such as Spin Torque Transfer (STT) RAM, can provide the same or better access performance and lower power consumption when compared with dynamic random access memory (DRAM). These nice features make NVRAM become an attractive DRAM replacement on NAND flash storage for resolving the management overhead of the flash translation layer (FTL). For instance, when adopting NVRAM for storing the mapping entries of FTL, the overheads of loading and storing the mapping entries between the non-volatile NAND flash and the volatile DRAM can be eliminated. Nevertheless, due to the limited lifetime constraint of NVRAM, the bit-level update behavior of FTL may lead to the issue of uneven bit-level wearing and the lifetime capacity of those less-worn NVRAM cells could be underutilized. Such an observation motivates this study to utilize the emerging NAND-like Spin Torque Transfer memory (NAND-SPIN) for alleviating the uneven bit-level wearing of NVRAM-based FTL and making the best of the lifetime capacity of each NAND-SPIN cell. The experimental results show that the proposed design can effectively avoid the uneven bit-level wearing, when compared with page-based FTL on NAND-SPIN.

Index Terms—NAND-SPIN, FTL, NAND Flash, NVRAM

I. INTRODUCTION

Flash translation layer (FTL) has been used extensively to manage the constraints of NAND flash memory on NAND flash storage devices. FTL remaps write requests to writable free space and records the mapping between logical address space and physical address space as logical-to-physical mapping entries. These mapping entries are loaded into dynamic random access memory (DRAM) for updates and stored back to NAND flash for persistence. Vulnerabilities of DRAM-based FTL include load/store overhead and possible data loss during sudden power outages. To resolve these issues, non-volatile random access memory (NVRAM) has been investigated to replace DRAM for hosting FTL with its nonvolatility and similar-to-DRAM access performance [3, 4, 9]. Examples of NVRAM include Phase Change Memory (PCM) and Spin Torque Transfer (STT) RAM. The major design consideration of NVRAM-based FTL is the lifetime constraint of NVRAM. In particular, the bit-level update pattern during mapping entry updates could cause uneven bit-level wearing to NVRAM

cells. When part of the cells in a mapping entry wear out prematurely, the lifetime capacity of other less-worn cells in the same entry is underutilized. Even though previous researchers [4] have tried to distribute the updates of mapping entries evenly across the whole NVRAM for avoiding uneven inter-entry wearing, *the issue of uneven intra-entry bit-level wearing receives much less attention.*

To alleviate the issue of uneven bit-level wearing, this study proposes to utilize the emerging NAND-like Spin Torque Transfer memory (NAND-SPIN) [10] as the underlying NVRAM for FTL. NAND-SPIN, which is evolved from STT-RAM and Spin Orbit Torque (SOT) RAM, has shorter access latency and provides higher read/write throughput, when compared with DRAM. In addition, NAND-SPIN has a higher areal density than both STT-RAM and SOT-RAM for providing larger capacity with the same die area. NAND-SPIN allows bit-level programs¹ to alter individual cells from 0 to 1. However, unlike previous NVRAMs, changing NAND-SPIN cells from 1 to 0 needs to be carried out in the unit of string and is referred to as the *string-based erase*. Note that a NAND-SPIN string is composed of multiple NAND-SPIN cells, each of which stores one bit.

The unique string-based erase of NAND-SPIN provides the possibility of alleviating the uneven bit-level wearing issue of NVRAM-based FTL. This is because, according to our investigation, the uneven bit-level wearing is mainly attributed to the unpredictability of bit-level patterns during FTL mapping updates, and string-based erases can partially resolve the unpredictability by ensuring cells of the same string receive the same number of erases. Nevertheless, since changing a cell from 1 to 0 requires whole-string erases, directly deploying conventional FTL [1] on NAND-SPIN could lead to excessive erases and cause uneven wearing between mapping entries. In summary, the technical difficulty of hosting FTL on NAND-SPIN lies in *how to alleviate both the uneven inter- and intra-entry wearing while minimizing the number of string-based erases.*

To utilize the nice features of NAND-SPIN for NVRAM-based FTL, this study proposes a NAND-SPIN FTL (NS-

¹In this study, the term “program” and “write” are used interchangeably for referring to the operation of changing the value in an NVRAM cell.

FTL) to alleviate both the uneven inter-entry and intra-entry bit-level wearing issue of NVRAM-based FTL and make the best of the lifetime capacity² of every NAND-SPIN cell. NS-FTL alleviates the uneven bit-level wearing by trying to ensure cells of a NAND-SPIN string can all be programmed through bit-level programs before string-based erases. Such an approach makes cells of the same NAND-SPIN string have similar numbers of programs and erases. In addition, this approach also allows mapping entries to be updated multiple times by changing bits from 0 to 1 with bit-level programs and avoids 0-to-1 updates to minimize the number of string-based erases. NS-FTL introduces three different components: (1) the inverted mapping strategy, (2) the zig-zag space allocator, and (3) the dual rotation wear leveler. The inverted mapping strategy is firstly used to minimize the number of ‘1’ bits during updates by flipping the bits of to-be-program mapping entry, when the number of to-be-program ‘1’ bits is more than that of ‘0’ bits. Next, the zig-zag space allocator is included to manipulate the allocation strategy of NAND flash storage for enhancing the possibility of the 0-to-1 update pattern. Finally, the dual rotation wear leveler is utilized to alleviate the condition of uneven inter-entry wearing.

The contributions of this study can be highlight as follows:

- This study is a pioneer design proposed to resolve the uneven bit-level wearing issue of NVRAM-based FTL.
- The proposed NS-FTL achieves its design goals by allowing an FTL mapping entry to receive multiple updates through the bit-level program feature of NAND-SPIN via maximizing the possibility of the 0-to-1 update pattern with the proposed components.
- Experimental results show that the proposed NS-FTL can effectively reduce the standard deviation of the bit-level program/erases by up to 88.86%, when compared with naively deploying page-based FTL [1] on NAND-SPIN.

The rest of this study is organized as follows. The background and motivation are described in Section II. The design of NS-FTL is then introduced in Section III. Next, the evaluation results are explained in Section IV. Finally, Section V presents the concluding remarks of this study.

II. BACKGROUND AND MOTIVATION

A. NVRAM-based FTL

NAND flash memory has become the mainstream storage medium in computer systems with its high access (read and write) speed and shock resistance. Nevertheless, NAND flash has several management constraints, including the erase-before-write property, asymmetric program/erase (P/E) units, and limited P/E cycles. For instance, a NAND flash block is the minimum erase unit and consists of a fixed number of pages, while a NAND flash page is the smallest read/write unit. To manage the constraints of NAND flash, FTL is utilized to redirect write requests to the free (erased) area and maintains logical-to-physical mapping entries. As shown in Figure 1, when NAND flash storage revives data updates,

²Lifetime capacity refers to the total number of writes that an NVRAM cell can endure before worn out.

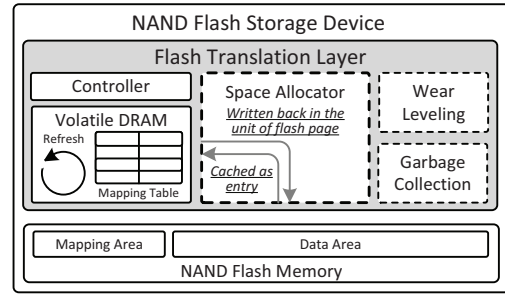


Fig. 1. **Conventional DRAM-based FTL**, in which mapping entries are loaded onto DRAM for updates and stored back to NAND flash for persistence. Notably, the modules of space controller, wear leveling, and garbage collection are used to manage the constraints of NAND flash.

mapping entries are loaded from NAND flash pages into DRAM for mapping entries updates. Then, those updated entries are stored back to NAND flash pages for persistence.

These load and store operations of mapping entries induce internal management overhead on NAND flash devices. In addition, due to the size difference between a mapping entry and the minimal access unit of NAND flash, storing mapping entries back to NAND flash also induces the problem of write amplification. For instance, even when only few bits are updated for a mapping entry update, a whole flash page needs to be written for preserving those updated bits. To avoid the write amplification and lower the management overhead of FTL, NVRAM has been considered to replace DRAM and used to store mapping entries of FTL.

As the primary design issue of NVRAM-based FTL is the lifetime constraint of NVRAM, studies have been proposed to avoid intensive mapping updates. For instance, Liu et al. [4] proposed a write-activity-aware PCM-assisted flash memory management scheme, also known as PCM-FTL, to reduce the maximum number of bit flips on the PCM. PCM-FTL employs page-based mapping for random writes, block-based mapping for sequential writes, and an inter-entry wear leveler. Nevertheless, the bit-level update behaviors of FTL are not considered within the design of PCM-FTL. In particular, *uneven bit-level wearing may lead to premature wearing to some of the NVRAM cells in a mapping entry and leave the lifetime capacity of the rest NVRAM cells underutilized.*

B. NAND-SPIN

NAND-SPIN is regarded as a promising successor to STT-RAM and SOT-RAM with comparable access latency and

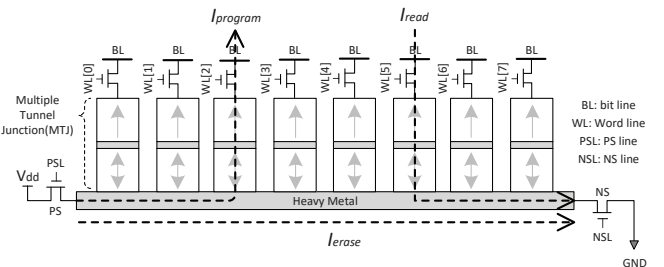


Fig. 2. **NAND-SPIN Memory**, in which 2^n MTJs are integrated onto the same heavy-metal strip and share the same PMOS (PS) transistor. Compared with STT-RAM and SOT-RAM, NAND-SPIN can achieve higher density, better energy efficiency and faster access speed.

better energy efficiency, as summarized in Table I. NAND-SPIN resolves the source degeneration³ issue of STT-RAM via unidirectional program/erase current and achieves higher density than SOT-RAM via sharing the heavy metal between cells. As shown in Figure 2, NAND-SPIN connects 2^n Magnetic Tunnel Junction (MTJ) onto one heavy-metal strip. MTJ is the basic storage unit of NAND-SPIN memory and consists of 2 ferromagnetic layers, which are the *reference layer* with fixed magnetization direction and the *free layer* with alterable magnetization direction. The value stored in the MTJ cell is determined by the parallelity between the fixed and free layers. In this study, we define the parallel state as the value of 1 and the antiparallel state as the value of 0.

TABLE I
COMPARISON OF STT-RAM, SOT-RAM, AND NAND-SPIN [10].

Parameters	STT-RAM	SOT-RAM	NAND-SPIN
Size (F^2)	40.3	9*2	7.95
Read Latency (ns)	1.62	1.63	1.65
Write Latency (ns)	'0' to '1': 6 '1' to '0': 4	'0' to '1': 0.7 '1' to '0': 1	Erase: 1 Program: 4
Read Energy (fJ)	'0': 15.336 for '1': 16.285 for	'0': 15.987 '1': 16.78	'0': 19.173 '1': 20.134
Write Energy (fJ)	'0' to '1': 627 '1' to '0': 1387	'0' to '1': 178.6 '1' to '0': 127.2	Erase: 30.91 /bit Program: 369.7 /bit

On NAND-SPIN, the parallelity of an MTJ cell is changed through either erase operation or write operation. For erase operation, NAND-SPIN cells need to be erased to antiparallel state in the unit of string. Erase operation is achieved by activating the selection transistors (PS and NS) of the selected string and passing a charge current through the heavy-metal strip to generate the SOT (I_{erase} in Figure 2). Erased cells are in the antiparallel state and represent the value of 0. On the other hand, program operation switches bit-cells to parallel state with the value of 1 by activating the access transistors and PS transistors of those bit-cells to be switched and grounding bit-lines (BLs) for a current, which induces the STT, to flow from the free layer to reference layer of MTJs ($I_{program}$ in Figure 2). Meanwhile, read operations require the access transistors and NS transistors to be activated. Then, BL is connected to the sensing amplifier for sensing the resistance level of MTJs for determining the stored data.

Based on NAND-SPIN, previous studies [11, 12] mainly focus on reducing the number of erases when NAND-SPIN is utilized as the processor cache. For instance, Wu et al. [11] propose an adaptive buffer entry (ABE) write policy to adaptively extend the write data length under a fixed maximum supply current. Meanwhile, a non-volatile look-up table is proposed by Zhang et al. [12] to enable a shared random access module for fast configuration and readout operations. *The applicability of alleviating the uneven bit-level wearing on NVRAM via NAND-SPIN receives much less attention.*

C. Motivation

Designs of NVRAM-based FTL have been proposed to store the management data of FTL on NVRAM. However,

³Source degeneration refers to the degraded drivability of the access transistor in STT-RAM due to the smaller gate-source bias when current flows from the source line (SL) to bit line (BL) [10].

most of previous NVRAM-based FTL designs [4] focus on reducing write traffic of FTL to reduce the number of overall mapping entry updates on NVRAM. *The problem of uneven bit-level wearing is neglected in previous studies.* The bit-level uneven wearing on NVRAM is induced by the fact that updates to mapping entries are conducted in bit level, and the number of writes to each NVRAM cell is not balanced.

With the emergency of NAND-SPIN memory, we observe the opportunity of alleviating the uneven bit-level wearing condition of NVRAM-based FTL via the string-based erases. String-based erases can naturally ensure that cells in the same string receive the same number of erases. In other words, if a mapping entry is stored on a single NAND-SPIN string, cells of a mapping entry will have the same number of erases. Nevertheless, string-based erases can not solely resolve the uneven bit-level wearing issue because the content of mapping entry updates is unpredictable. Thus, the number of programs on each cell can not be guaranteed. On the other hand, directly deploying conventional FTL [1] on NAND-SPIN could lead to excessive erases because updating a cell from 1 to 0 requires the string-based erase. In summary, *the major issue of alleviating the uneven bit-level wearing via NAND-SPIN lies in how to alleviate uneven intra-entry wearing and avoid uneven inter-entry wearing by minimizing the number of string-based erases.*

III. NAND-SPIN-BASED FLASH TRANSLATION LAYER

A. Overview

To revolve the uneven bit-level wearing of NVRAM-based FTL, this section presents a NAND-SPIN-based FTL (NS-FTL) to ensure bits of a NAND-SPIN string can revive similar numbers of programs and erases. NS-FTL realizes this design goal by allowing a mapping entry to receive multiple mapping updates with bit-level programs to change bits from 0 to 1 before string-based erases are used to change all bits in a NAND-SPIN string from 1 to 0. Therefore, as most bits are programmed before erased, the uneven wearing condition is diminished. To the best of our investigation, few studies have investigated the applicability of utilizing NAND-SPIN to revolve the uneven bit-level wearing issue of NVRAM-based FTL. The system architecture of the proposed NS-FTL is illustrated in Figure 3.

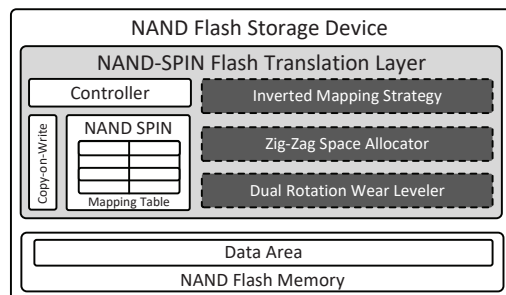


Fig. 3. System architecture of the proposed NS-FTL, in which mapping entries of the FTL are stored on NAND-SPIN and a small copy-on-write buffer is utilized to prevent inconsistency during updating a mapping entry.

The proposed NS-FTL is designed as an FTL to manage both the mapping entries of FTL and the storage space of

NAND flash memory. NS-FTL is a page-based FTL and records the logical-to-physical mapping for each flash page. Each mapping entry is stored on a single NAND-SPIN string. In other words, a mapping entry can be erased independently without affecting other entries. During updating an existing entry, a small copy-on-write buffer is used to prevent inconsistency if a power failure happens during mapping entry updates. The core concept of NS-FTL is to allow a mapping entry to receive multiple updates without erases. To enforce this concept, NS-FTL first minimizes the number of '1' bits in mapping entry updates by the inverted mapping strategy (see Section III-B) to flip the bits of a mapping entry if the number of to-be-program '1' bits is larger than that of '0' bits. Secondly, to update a mapping entry multiple times without string-based erases, the zig-zag space allocator (see Section III-C) is included to maximize the possibility of the 0-to-1 update pattern. With the first two components, NS-FTL aims to alleviate the uneven intra-entry wearing. Finally, to avoid uneven inter-entry wearing, the dual-rotation wear leveler (see Section III-D) is included to periodically rotate the mapping entries for avoiding excessive updates to a small region of NAND-SPIN strings.

B. Inverted Mapping Strategy

Due to the unique physical layout of NAND-SPIN, updating a NAND-SPIN cell from 1 to 0 requires the string-based erase, which consumes the valuable lifetime of NAND-SPIN cells. To avoid excessive string-based erases, NS-FTL minimizes the number of '1' bits during updating mapping entries by flipping the mapping entry when the number of '1' bits is larger than that of '0' bits. Then, future mapping entry updates can be achieved via altering '0' bits to '1' bits with the bit-level programs of NAND-SPIN, thus lowering the number of string-based erases. The mechanism of inverted mapping strategy can be summarized as Figure 4. Note that an extra invert bit is included in each mapping entry to record the flip status and is set to 1 if the entry is inverted when stored on the NAND-SPIN string.

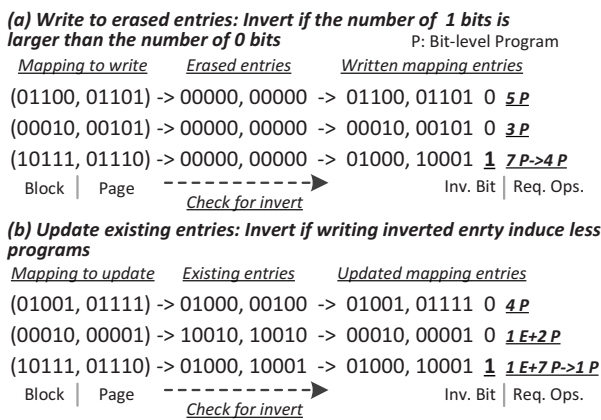


Fig. 4. **The inverted mapping strategy**, in which mapping entry updates are checked for counting the number of to-be-programmed '1' bits and inverted if the number of to-be-programmed '1' bits is larger than the number of to-be-programmed '0' bits.

As shown in Figure 4, the inverted mapping strategy categorizes mapping entry updates into two different scenarios.

The first scenario is writing to erased mapping entries. As shown in Figure 4(a), mapping entries are inverted if the number of to-be-programmed '1' bits is larger than that of to-be-programmed '0' bits. For instance, when writing the mapping entry (10111, 01110) to an erased NAND-SPIN string, the mapping will be inverted for inducing less 0-to-1 programs. Such an approach makes most of the bits to be '0' after initial entry updates and lowers the possibility of the 1-to-0 update pattern during future mapping entry updates for avoiding string-based erases, as most of the bits are left as 0.

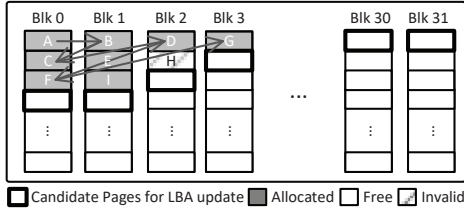
Leaving most of the bits in an entry as 0 also provides the possibility of updating the entry multiple times via bit-level programs without string-based erases. For instance, as shown in Figure 4(b), writing a mapping update (01001, 01111) into an existing entry (01000, 00100) requires only 0-to-1 programs and induce no erases. On the other hand, when writing updated mapping to an existing entry, bits of the to-be-updated entry are flipped if the flipped entry induces less 0-to-1 programs or no erases, which is the most common condition at runtime of NS-FTL. In Figure 4(b), bits of the updated mapping (10111, 01110) are flipped when written to the existing entry (01000, 10001) and thus only induce one bit-level program. *Updating entries via bit-level programs without string-based erases provides a higher chance for every bit to be programmed before being erased, thus enabling NS-FTL to alleviate the uneven bit-level wearing.*

C. Zig-Zag Space Allocator

As NS-FTL is designed to store the logical-to-physical mapping of NAND flash, the bit pattern of each mapping entry is dependent on the allocation strategy of NAND flash pages. In other words, inappropriate allocation strategy may induce frequent string-based erases and aggravate the issue of uneven bit-level wearing. For instance, if the allocation strategy is unaware of the string-based erases and forces a mapping to be erased during updates with the 1-to-0 update pattern, bits with the value of '0' may get less wearing when compared with bits with the value of '1'. This is because bits with the value of '1' are programmed after erased, while bits with the value of '0' are not programmed after erased. As this variance accumulates during each entry update, the issue of uneven bit-level wearing becomes significant. To alleviate above condition, the zig-zag space allocator is included to maximize the chance of updating a mapping entry without string-based erases, while complying with the allocation restriction of NAND flash. The allocation strategy of the included allocator can be summarized as Figure 5.

As shown in Figure 5(a), the allocation of NAND flash pages are performed in a zig-zag pattern across multiple blocks. The zig-zag allocator allows pages with different block and page numbers to become allocatable at the same time and provides multiple *candidate pages* for a logical page update. Candidate pages with different block and page numbers also have different bit patterns in their logical-to-physical mappings. Such an approach increases the possibility of updating mapping entry via bit-level programs because there are multiple pages of different bit patterns in their logical-to-physical mapping to choose from during logical

(a) Zig-Zag Space Allocation



(b) Example of allocating page for a logical page update

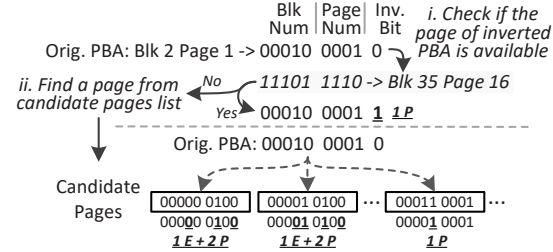


Fig. 5. The zig-zag space allocator, which allocates NAND-flash pages for logical page updates in a zig-zag pattern to maximize the chance of updating mapping entry via bit-level programs only and with no string-based erases.

page updates. Notably, because NAND flash only allows pages to be allocated in ascending order, each block has only one available candidate page at a time.

The detailed workflow of the zig-zag allocator can be summarized as Figure 5(b). While updating a mapping entry for a logical page update, the allocator first checks if the physical page with inverted mapping is available or not. If it is available, the updated data content will be stored into the physical page with inverted logical-to-physical mapping. Then, the mapping entry update can be achieved by setting the inverted bit to 1. In contrast, if the physical page with inverted mapping is not available, NS-FTL will go through candidate pages for finding a physical page with the mapping that induces the minimal number of programs or erases. It is worth noting that the zig-zag pattern also makes most of the bits in the mapping entries become zero with smaller block and page numbers. During future updates, these entries have greater chance of being updated by bit-level programs to increase block or page numbers without being erased.

D. Dual Rotation Wear Leveler

In addition to alleviating the uneven bit-level wearing by exploiting the features of NAND-SPIN, the dual rotation wear leveler is designed to resolve the issue of uneven inter-entry wearing by referring to previous hardware-based wear leveler [6]. Uneven inter-entry wearing happens when a few mapping entries are updated multiple times due to intensive logical page updates. As shown in Figure 6, to resolve this issue, NS-FTL divides NAND-SPIN into multiple regions, each of which is composed of a fixed number of NAND-SPIN strings, and designates a *pivot* and an extra string as *interval* in each region. For performing inter-entry wear leveling, the *interval* will be moved forward after a region revives N entry updates. After the *interval* reaches the location of the *pivot*, the *pivot* will be moved backward and the *interval* will be reset to the end of the region again. Above operations are referred to as the in-region rotation and aim to quickly spread

the intensive logical page updates across different NAND-SPIN strings without inducing high movement overhead.

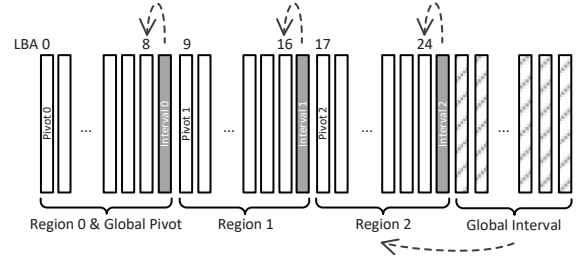


Fig. 6. The dual rotation wear leveler, in which the rotation-based wear leveling is performed at both the in-region and inter-region levels. After a fixed number of entry updates are received in one region or on the NAND-SPIN, the in-region or global interval will be moved forward.

On the other hand, rotation is also performed at the region level, and it is referred to as inter-region rotation. The goal of inter-region rotation is to avoid intensive mapping updates to a region, owing to strong locality in data update patterns. Similarly, a region is set as the *global pivot*, and one extra region is included as the *global interval*. As the global rotation induces larger overhead when compared with in-region rotation, it is performed less frequently than in-region rotation and is triggered when the NAND-SPIN received M entry updates, where M is larger than N . The value of N and M can be set according to the lifetime of NAND-SPIN. Based on previous study [7] and assuming NAND-SPIN has the same MTJ size as STT-RAM, the lifetime is predicted to be 4×10^{12} . We then refer to 10^3 and 10^6 as the value of N and M by referring to PCM-based wear leveling study [6].

IV. PERFORMANCE EVALUATION

A. Experiment Setup

In this section, evaluations are conducted by using the Microsoft Research Cambridge (MSR) [5] traces to evaluate the proposed NS-FTL in terms of the bit-level wearing, entry-level wearing, and the throughput of accessing mapping entries on NAND-SPIN. Meanwhile, a self-collected I/O trace is also utilized to assess the effectiveness of NS-FTL with the workload of one-month period personal computer usages. In the experiments, NS-FTL is implemented based on a flash simulator [2] and compared with (1) page-based FTL [1] on NAND-SPIN and (2) page-based FTL with hardware-based wear leveler [6] on NAND-SPIN. These two configuration are referred to as *FTL* and *FTL-WL* in this section. Notably, similar to the dual-rotation wear leveler, the hardware-based wear leveler also includes a pivot and an interval on NVRAM and performs entry-level rotation when the NVRAM receive a fixed number of writes, which is set as 10^3 in this evaluation. The latency parameters of NAND-SPIN can be found in Table I and the size of the NAND flash storage is 64 GB with 16 KB pages [8].

B. Experimental Results

As the main goal of NS-FTL is to mitigate the uneven bit-level wearing of NVRAM-based FTL, Figures 7 and 10 first show the reduced amount of standard deviation and arithmetic mean for bit-level program/erase count. The results show that NS-FTL can effectively reduce the bit-level standard

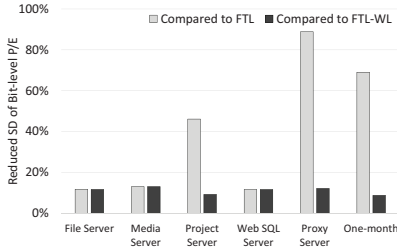


Fig. 7. Standard deviation comparison of bit-level program/erase count.

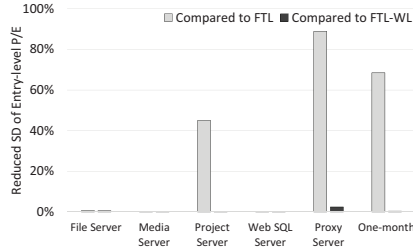


Fig. 8. Standard deviation comparison of entry-level program/erase count.

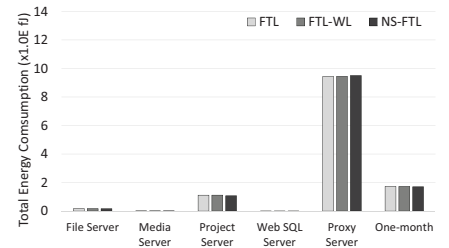


Fig. 9. Total Energy Consumption.

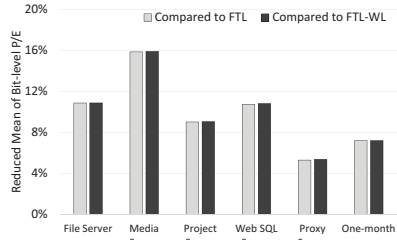


Fig. 10. Arithmetic mean comparison of bit-level program/erase count.

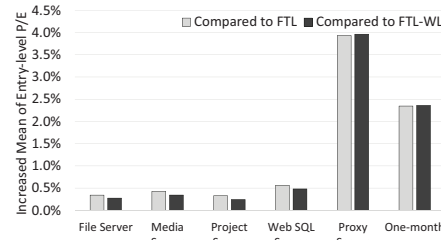


Fig. 11. Arithmetic mean comparison of entry-level program/erase count.

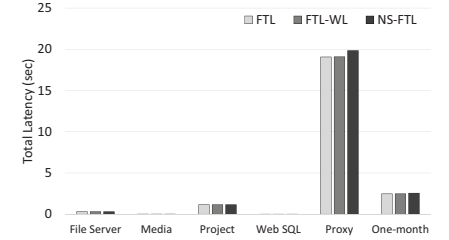


Fig. 12. Total latency.

deviation and arithmetic mean by up to 88.86% and 15.84%, when compared with page-based FTL. The average reductions are 40.11% and 9.86% for bit-level standard deviation and arithmetic mean. These reductions are achieved by both the inverted mapping strategy and zig-zag space allocator. In addition, comparing with FTL-WL, the reductions of bit-level standard deviation and mean are 9.86% and 11.14%, which suggests that NS-FTL can outperform FTL-WL in terms of bit-level wear leveling.

For entry-level comparison, NS-FTL can reduce the standard deviation by 33.88% on average and up to 2.45%, comparing with FTL and FTL-WL, as shown in Figure 8. The reduction of entry-level standard deviation suggests that NS-FTL can effectively utilize NAND-SPIN to alleviate uneven bit-level wearing, while avoiding excessive string-based erases for preventing uneven entry-level wearing. Meanwhile, to further avoid excessive string erases, the arithmetic mean of entry-level program/erase count is slightly increased by 1.30%, as shown in Figure 11, due to the inter-region rotation of the dual rotation wear leveler.

The comparisons of NAND-SPIN energy consumption and latency are reported. Figures 9 suggests that the energy consumption can actually be decreased by 5.16% on average, due to the reduced number of the 0-to-1 update pattern during mapping entry updates. On the other hand, Figures 12 shows that the latency of updating and accessing mapping entry is similar to the page-based FTL, and the difference is only 1.58%, even with the mechanisms of NF-FTL.

V. CONCLUSION

To resolve the uneven bit-level wearing of NVRAM-based FTL, this study firstly presents the NAND-SPIN FTL (NS-FTL) to utilize the unique bit-level program and string-based erase features of NAND-SPIN. NS-FTL firstly introduces the inverted mapping strategy to minimize the number of '1' bits during mapping entry updates for avoiding excessive string-based erases. Then, the zig-zag space allocator is included

to alter the allocation of NAND flash pages for further the probability of the 0-to-1 update pattern. Finally, to avoid uneven inter-entry wearing, the dual rotation wear leveler is utilized to periodically rotate the physical location of each mapping entry. Experimental results show that the standard deviation of bit-level program/erase count can be reduced up to 88.86%, when compared with the page-based FTL.

REFERENCES

- [1] A. Ban. Flash file system, U.S. 5404485 A, Apr. 1995.
- [2] Y.-H. Chang and T.-W. Kuo. A management strategy for the reliability and performance improvement of mlc-based flash-memory storage systems. *IEEE Transactions on Computers*, Mar 2011.
- [3] J. K. Kim, H. G. Lee, S. Choi, and K. I. Bahng. A pram and nand flash hybrid architecture for high-performance embedded storage subsystems. In *Proceedings of the 8th ACM International Conference on Embedded Software*, 2008.
- [4] D. Liu, T. Wang, Y. Wang, Z. Qin, and Z. Shao. Pcm-ftl: A write-activity-aware nand flash memory management scheme for pcm-based embedded systems. In *IEEE 32nd Real-Time Systems Symposium*, 2011.
- [5] D. Narayanan, A. Donnelly, and A. Rowstron. Write off-loading: Practical power management for enterprise storage. In *ACM Transactions on Storage (TOS)*, 2008.
- [6] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali. Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2009.
- [7] K. Rani and H. K. Kapoor. Write variation aware buffer assignment for improved lifetime of non-volatile buffers in on-chip interconnects. *IEEE Transactions on Very Large Scale Integration Systems*, 2019.
- [8] Samsung. Samsung v-nand@ONLINE, <http://www.samsung.com/semiconductor/products/flash-storage/v-nand/>, 2017.
- [9] Y. SONG, S. LEE, D. H. KANG, and Y. I. EOM. Nvram-aware mapping table management for flash storage devices. *IEICE Transactions on Information and Systems*, 2019.
- [10] Z. Wang, L. Zhang, M. Wang, Z. Wang, D. Zhu, Y. Zhang, and W. Zhao. High-density nand-like spin transfer torque memory with spin orbit torque erase operation. *IEEE Electron Device Letters*, 2018.
- [11] B. Wu, P. Dai, Z. Wang, C. Wang, Y. Wang, J. Yang, Y. Cheng, D. Liu, Y. Zhang, W. Zhao, and X. S. Hu. Bulkyflip: A nand-spin-based last-level cache with bandwidth-oriented write management policy. *IEEE Transactions on Circuits and Systems*, 2020.
- [12] H. Zhang, W. Kang, Z. Wang, E. Deng, Y. Zhang, and W. Zhao. High-density and fast-configuration non-volatile look-up table based on nand-like spintronic memory. In *IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*, 2018.