SPEED: Scalable and Predictable EnhancEments for Data Handling in Autonomous Systems

Dongjoo Seo*[1], Changhoon Sung*[2], Junseok Park[3], Ping-Xiang Chen[1], Bryan Donyanavard[2], Nikil Dutt[1]
[1] *University of California, Irvine,* [2] *San Diego State University,* [3] *Kookmin University*
{dseo3, p.x.chen, dutt}@uci.edu, {csung7167, bdonyanavard}@sdsu.edu, 20191271@kookmin.ac.kr

***Abstract*—Scalable and predictable disk I/O management is critical for autonomous applications that must handle realtime sensor data streams. Existing ROS-based architectures for end-to-end autonomous compute pipelines suffer from lack of scalability and performance predictability that can compromise safety. We present SPEED, an approach leveraging a multi-queue architecture and a context-aware I/O scheduler to achieve scalability and performance predictability. We demonstrate SPEED's ability to reduce single I/O latency in *rosbag* by 84%, and improve by over 1.5× the scalability of existing camera sensors. Furthermore, SPEED's context-aware I/O scheduler improves the predictability of multi-task application pipelines by over a 3.08× reduction in performance standard deviation. Our results demonstrate significant improvements in I/O management, making SPEED well suited for the stringent I/O demands of modern end-to-end autonomous applications that must execute sensor-to-actuator compute pipelines while meeting realtime performance requirements.**

I. Introduction

Autonomous systems, such as autonomous vehicles (AVs), rely heavily on data from various sensors to make critical decisions in real-time for safe navigation of physical environments [1]. The efficient storage and retrieval of sensor data are crucial to ensure that information from sensors such as LiDAR, radar, and cameras is available for processing, not only with minimal delay, but also in a timely manner [1]–[8]. Figure 1 illustrates a cooperative system in an AV environment [1], where operations such as object detection and path planning interact with various sensors such as LiDAR, radar, GPS, and cameras in an end-to-end (sensors-to-actuators) computational pipeline that must meet timing constraints. I/O performance is a crucial factor in minimizing the end-to-end delay of the software pipeline to provide road safety control. All mobile autonomous systems generally require efficient data collection and sharing at runtime, e.g., a robot dog [5] or an autonomous vehicle [2] that uses multiple sensors and subsystems to navigate unfamiliar terrain.

Frameworks such as the Robot Operating System (ROS) are widely adopted to manage multiple autonomous tasks in complex sensor-driven autonomous systems, including in AVs [7]–[9]. ROS operates as a middleware layer atop the operating system (e.g., Linux), relying on OS-level mechanisms for I/O and scheduling, which can introduce unpredictability. ROS uses a publish-subscribe architecture that manages communications to ensure efficient data flow among applications, supporting multiple data streams through uniquely named topics. Each topic in ROS corresponds to an independent application flow characterized by its specific inputs and outputs. This model allows application developers to focus on individual tasks (e.g., AV computational pipeline blocks) tailored to each topic. Data associated with each topic can be archived locally using a storage engine known as *rosbag* [10]. The current *rosbag*, a vital component within the ROS ecosystem, is designed to efficiently handle the storage and retrieval of message data. *rosbag* operates by subscribing to specific topics and collecting incoming messages, then serializes the messages into a structured binary data format that, *in principle*, is portable and scalable [10].

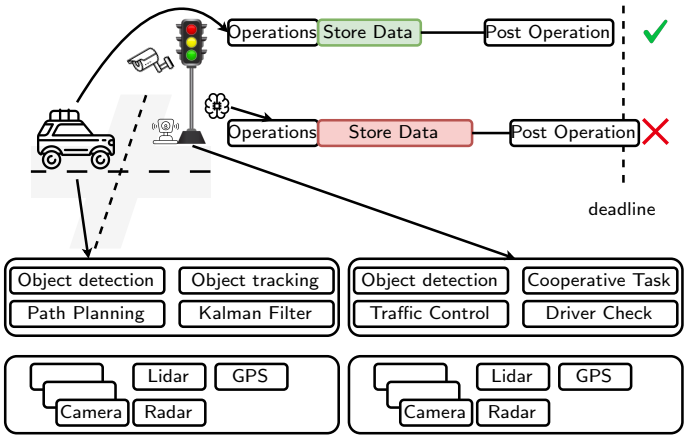


Fig. 1: Example cooperative AV system: different operations (e.g., object detection and path planning) interact with various sensors in an end-to-end pipeline. The I/O performance plays a critical role in end-to-end delays.

However, the current *rosbag* architecture's single-queue buffer system introduces significant performance degradation as the number of sensors and concurrent datastreams grow [3], limiting scalability to meet the increasing data throughput demands in modern autonomous systems. In particular, the current *rosbag* single-queue architecture poses two challenges: 1) *Maintaining performance as number of sensors grows*, since data logging/sharing become bottlenecks, leading to degraded performance in storage operations, and 2) *Reducing the standard deviation of end-to-end latency for realtime AV pipelines*, that affects the predictability (i.e., reduced variability) of realtime sensor data to meet strict timing constraints for safety-critical applications. Indeed, the timing of data writes to disk is controlled by the operating system [11], that can result

Dongjoo Seo and Changhoon Sung are co-first authors.

in unpredictable delays which can compromise consistent and timely performance paramount for ensuring safety.

These two challenges highlight the pressing need for a more advanced *rosbag* I/O framework that can guarantee scalable and predictable data management in the face of increasing sensor data inputs for contemporary autonomous systems. Our paper addresses these limitations through the following contributions:

- We propose SPEED: Scalable and Predictable Enhancements Data Handling for *rosbag*. SPEED is a multi-queue architecture that increases the performance and scalability of *rosbag*. Our evaluation in emulated sensors for Tesla AV scenarios [3] show an increase in the number of camera sensors supported by 1.5×.
- We present a context-aware I/O scheduler for SPEED that increases predictability by reducing the variability in performance standard deviation with *rosbag* for autonomous systems. Our evaluation demonstrates a 3.08× reduction in performance standard deviation for an autonomous vehicle application pipeline.
- Our experiments are carried out with an extensive simulated end-to-end autonomous vehicle pipeline using ROS.

II. Background and Challenges

We begin by outlining the operational dynamics of ROS-integrated applications. Then, we detail the mechanisms by which data is stored utilizing *rosbag* [10], highlighting the specific functionalities and challenges faced by this approach.

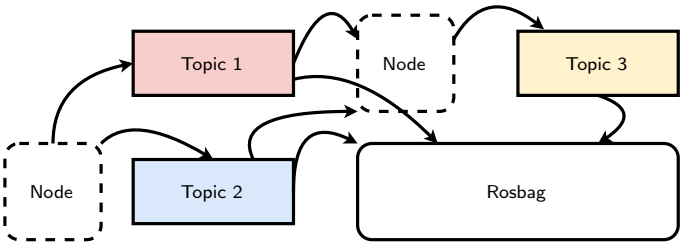


Fig. 2: Example data flow of multi-node ROS application storing data in a `rosbag`

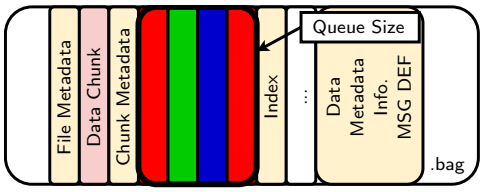


Fig. 3: Layout of data file in *rosbag* 2.0

ROS is based on the core architectural principle of distributed microservices, where the system is divided into distinct compute elements known as *nodes* [9]. These nodes interact over shared communication channels termed *topics*. Each node has the ability to publish messages to a topic, and nodes can subscribe to topics to receive relevant messages. This decentralized communication model is integral to ROS's flexibility and scalability in handling diverse robotic and AV tasks. In *rosbag*, the data-writing process begins with nodes publishing serialized message payloads to topics. *rosbag* captures these messages from the ROS network, appending meta-information including timestamps and topic names, and writes the data sequentially to a .bag file (see Figure 2). This format was sufficient for early versions of *rosbag*. As illustrated in Figure 3, while the data structure of *rosbag* can vary between versions, it generally involves combining the data from multiple randomly-ordered topics, with each topic containing a series of timestamped message payloads in a unified file structure. Importantly, as ROS is a middleware rather than a standalone OS, file I/O operations in *rosbag* ultimately rely on underlying OS-level calls. These calls often involve buffered I/O, which can block unpredictably and introduce variable delays that undermine real-time data logging requirements.

To address different operational needs, *rosbag* offers two primary storage methodologies. The first method uses a database-based system, where data is managed through structured databases such as SQLite [12]. This technique ensures strong data integrity and supports intricate query capabilities, which are crucial for thorough analysis and extraction of large datasets. However, given that end-to-end latency is a critical factor in autonomous systems, this method is undesirable [12]–[14]. The second method, exemplified by newer implementations such as MCAP (Message Capture Format) [15], stores data directly in the filesystem. This approach is designed for high-throughput writing and reading, ensuring minimal I/O completion latency in data handling. MCAP, in particular, improves the efficiency of data storage and access by leveraging a format optimized for rapid serialization and deserialization of ROS messages [10].

While MCAP addresses the latency concerns, ROS workloads commonly consist of multi-node applications (e.g., autonomous vehicles that execute multiple pipelined applications such as object detection, tracking, and planning) and can also comprise multiple collaborative applications (e.g., cooperative AVs, smart cities) that share and store data on-device at runtime [1], [3]–[5]. Such applications require numerous sensors and datastreams and must meet end-to-end latency constraints for safety. When these applications include *rosbag* on each platform to record data for logging, playback, or post-processing, *rosbag* poses two challenges: scalability and predictability.

Scalability is a major concern when dealing with the large volumes of data generated by numerous sensors in autonomous systems. The current *rosbag* architecture relies on a single-queue-based buffer system to handle data input, which becomes a bottleneck as the number of sensors increases. This bottleneck leads to increased latency and poorer performance when I/O operations are integrated into the pipeline, particularly in high-throughput environments [10]. As autonomous systems continue to evolve, the need for a more scalable I/O management system that can efficiently handle multiple concurrent data streams becomes increasingly evident.

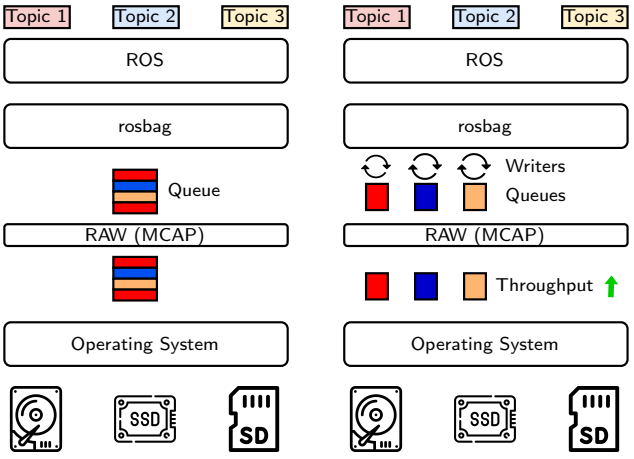
Predictability – manifested by minimizing timing variability – is another critical issue, especially given the safety-critical nature of autonomous applications like AVs. The timing of data writes to the actual disk device in the existing *rosbag*

system is highly dependent on the buffered I/O mechanisms of the underlying operating system [11]. This dependency introduces unpredictability in data storage, as the timing of when data is written to disk can vary significantly, leading to potential resource contention when saving data and ultimately less reliable runtime performance. Furthermore, the timing variation (via minimizing deviation) of end-to-end performance is also critical for ensuring autonomous system safety.

Given these two challenges, it is clear that common I/O engines, especially the current *rosbag* architecture, are insufficient for current autonomous systems, specifically suffering from inefficiencies in terms of scalability and predictability. We present SPEED: a novel approach that addresses these limitations by improving the scalability and predictability of *rosbag*, better aligning it with the needs of realtime autonomous systems.

III. SPEED Architecture and Scheduler

SPEED consists of a multi-queue architecture and context-aware I/O scheduler for *rosbag* file storage.



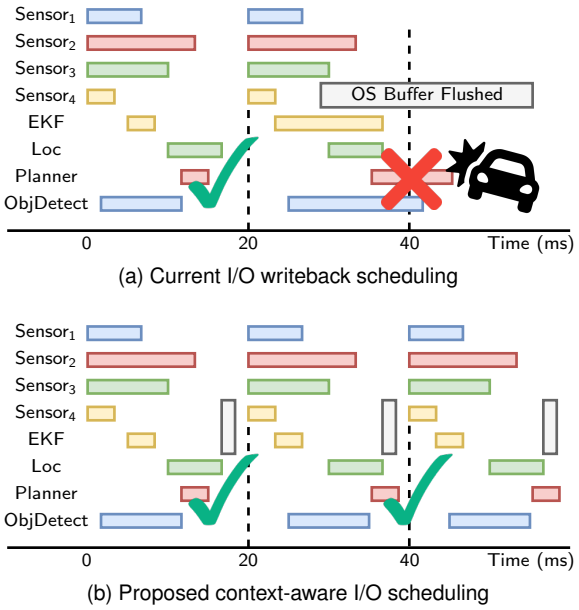
(a) Current I/O stack. (b) SPEED architecture.

Fig. 4: Existing I/O stack compared to SPEED architecture

A. Multi-Queue Architecture

In the traditional *rosbag* architecture (Figure 4a), data from various sensors is processed through a single-queue system, creating a bottleneck as the number of topics and amount of data stored increases, increasing end-to-end latency. We propose the SPEED architecture (Figure 4b), a multi-queue system that allocates a dedicated queue for each topic, allowing each sensor's data to be processed separately in parallel, thereby mitigating the inherent bottleneck in single-queue systems. Data from each sensor is dispatched to its respective queue based on the topic identifier. Queue sizes are determined based on the expected data size during system initialization. The SPEED architecture facilitates simultaneous data writing processes, thereby boosting overall application scalability up to the maximum capability of the operating system.

SPEED mitigates *rosbag* buffer overflow that occurs in single-queue architectures by utilizing asynchronous I/O and dynamic memory management. In the current *rosbag* implementation, a swapping buffer, referred to as a *cache*, is employed to allow message reception while performing synchronous write operations. Existing synchronous blocking system calls ensure data integrity while the kernel completes the write [16], but they involve a blocking wait until I/O completion. If data ingestion is faster than synchronous I/O processing, this can lead to buffer overflow, resulting in data loss. SPEED leverages `io_uring` [17], which enables efficient non-blocking I/O by queuing and processing asynchronous requests independently. Switching to asynchronous non-blocking methods can lead to data invalidation if the data is modified or deallocated before the write request is completed, resulting in corrupt data being recorded on disk. To avoid this potential data corruption during non-blocking asynchronous I/O operations, SPEED dynamically allocates memory and retains it until the completion event is confirmed via the completion queue event (CQE). This design reduces *rosbag cache* overflow significantly but increases the working set size of the application, as each asynchronous I/O request requires its own buffer.



(a) Current I/O writeback scheduling

(b) Proposed context-aware I/O scheduling

Fig. 5: Comparison of I/O scheduling for sensor data in AV examplar

B. Context-Aware I/O Scheduler

In Linux-based operating systems, I/O scheduling is typically managed through the block layer [18], which handles buffered writes and coordinates I/O requests to storage devices. Although effective in conventional computing, this approach introduces significant predictability challenges in safety-critical applications such as AVs. Buffered writes introduce unpredictability by delaying data flushes to disk based on operating system conditions, which may interfere with realtime requirements in autonomous systems. This unpredictability leads to resource contention, particularly during processing-critical moments, which compromises the end-to-end latency

essential for AV operations. Figure 5a illustrates how delayed writebacks of buffered data can cause predictability issues. Initially, I/O operations are buffered by the OS, and actual disk writes occur later when the OS decides to flush the buffer, potentially holding locks and blocking synchronous *rosbag* operations. The inherent latency and nondeterministic behavior of the traditional block layer exacerbate these issues, compromising system predictability.

To address these challenges, we introduce a context-aware I/O scheduler that leverages the end-to-end application cycles of autonomous systems. As shown in Figure 5b, our hint-based scheduler optimizes the timing of writeback operations by using specific hints from the application when it starts and completes its periodic tasks. By modifying the application to trigger the scheduler based on regular events, the scheduler detects cycle completions and triggers a writeback. The trigger mechanism employs a heuristic that balances writeback with periods of system idleness to minimize interference with ongoing processing. The heuristic assigns static computation weights during initialization, considering both the combined computation weight of each application and the amount of data accumulated in the buffer, and triggers flushes based on an empirical value that varies depending on the autonomous system.

The scheduler flushes buffered data at the ideal time, such as after the compute-intensive phase of the autonomous pipeline, enhancing predictability by writing back data when it is least intrusive. For example, in Figure 5b, when applications and sensor operations complete every 20 ms, *rosbag* flushes data to disk, minimizing the impact on application execution. This method differs from conventional OS writeback handling [19], where buffered writes are delayed and batched without considering the application's realtime requirements, potentially causing performance variability.

IV. Evaluation

A. Environment Setup

CPU	AMD Ryzen™ 7 5800X
GPU	Nvidia Geforce RTX 3060 12GB
Memory	DDR4 64GB
OS	Ubuntu 22.04.4
Kernel	6.5.0-45-generic
liburing	2.6
ROS2	Humble Hawksbill
Rosbag2	0.15.10

TABLE I: Environment Specification

Table I outlines our setup specification, where we conducted our evaluation on a single machine, following a setup comparable to Tesla's autonomous vehicle configuration [3]. Note that while previous work [10] recorded ROS data to evaluate the ROS-based storage scenario, it is hard to reproduce OS behavior (especially the effect of buffered writes from *rosbag* on recorded data), hence our current experimental setup.

In ROS2 Humble, the default storage format for *rosbag* is MCAP, and both the default *rosbag* and SPEED use MCAP as the storage format. However, for simplicity, from this point onward, we will refer to the *rosbag* default behavior and architecture collectively as MCAP when making comparisons with our proposed approach.

B. Workloads

Name	Msg Size (Byte)	Msg rate (Hz)	Msg Count
Camera	5529600	36	360
Radar	1600	50	500
Main LiDAR	640000	25	250
Secondary LiDAR	160000	25	250
GPS	200	25	250
IMU	400	100	1000
Ultrasonic	150	40	400

TABLE II: Microbenchmark Workloads

Initially, to isolate and evaluate the performance of the storage engine more accurately for our benchmarks, we bypass the ROS transport layer based on the Data Distribution Service (DDS) for microbenchmarks. If the input stream exceeds the *rosbag*'s processing speed, the recorder's *cache* may overflow, causing message drops. Therefore, a high percentage of recorded messages indicates sufficient performance. We use bpftrace, a tracing tool based on extended Berkeley Packet Filter (eBPF) [20], to measure latency in the *rosbag* and related kernel subsystems.

Table II details the workloads used in this section. We create three workloads for evaluation. The first workload is used for scalability evaluation. We simulate automotive sensors using the rosbag2 performance benchmarking tool with Tesla HW 3.0 camera specs (1280x960 HDR12 @ 36fps) [3]. Each sensor node can send messages to multiple topics, and benchmarks are measured five times and averaged.

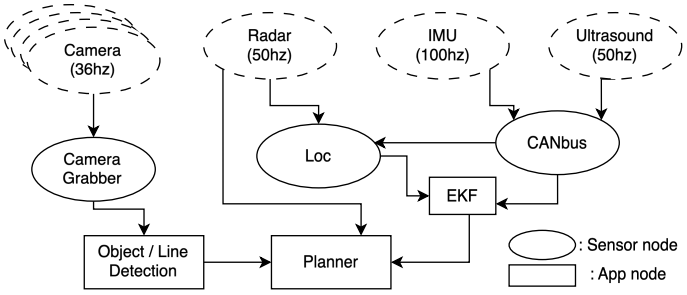


Fig. 6: Autonomous Vehicle end-to-end pipelined applications

The second workload is used for variability evaluation. For variability evaluation in a more realistic environment, we implement a comprehensive end-to-end autonomous vehicle workload. Figure 6 shows the workload running on the system with ROS. We use four different types and frequency of sensors, which is important for functional work on the autonomous vehicle scenario with DDS [21]. Due to the overhead of supporting multiple real cameras, we use images from one real camera sensor and the remaining camera sensors are emulated using images from the KITTI dataset [22] time-aligned with the real camera. While we emulate some sensors to generate consistent and repeatable workloads, we ensure that the data

rates and message sizes closely match those found in real-world scenarios.

C. Scalability

We evaluate the SPEED architecture compared to the state-of-the-art *rosbag* architecture, MCAP, in three dimensions: 1) single-topic I/O completion latency; 2) single-topic multi-sensor throughput; and 3) multi-topic multi-sensor throughput.

1) Single ROS Topic I/O Performance: We first analyze the latency involved in processing messages within the *rosbag*. This latency is broken down into three key stages: (1) message handling, (2) I/O preprocessing, and (3) performing the operating system write operation. We execute the workload using the rosbag2 performance benchmarking tool [23]. The workload consists of a single-camera node generating images at 36HZ [3] running for 10 seconds. Each data point in Figure 7 represents the cumulative latency of I/O operation in *rosbag*. As shown in Figure 7, MCAP spends the majority of its time waiting for the system call (syscall) to complete during I/O operations, making it a significant contributor to the overall latency. Specifically, a large portion of the total time is spent in the syscall phase, which creates a bottleneck that slows down data processing. In contrast, SPEED significantly reduces this waiting time by handling I/O asynchronously. By moving the blocking I/O calls out of the critical execution path, SPEED allows other operations to continue without being delayed by I/O tasks in *rosbag*. This asynchronous handling reduces both I/O preprocessing and syscall latency by 84%, contributing to an overall reduction of 79% in total latency.

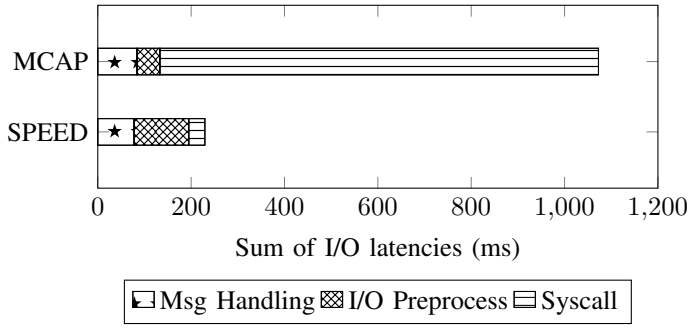


Fig. 7: Breakdown of Sum of I/O Latencies in MCAP and SPEED with Camera Data Saving Workload

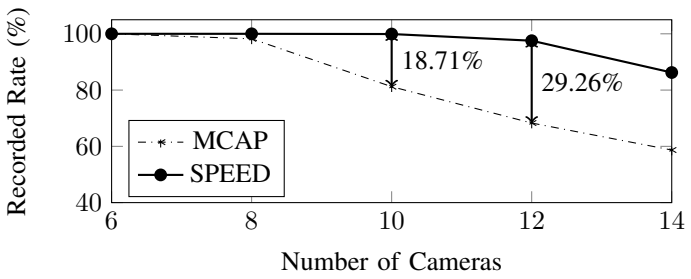


Fig. 8: Recorded Success Rate by Number of Camera Count

2) Multiple ROS Sensor I/O Scalability: After establishing that SPEED significantly reduces I/O latency, we now turn to scalability. We evaluate how well *rosbag* performs when we increase the number of camera nodes publishing to the same topic. Figure 8 depicts the number of cameras on the x-axis, and the data recorded rate of the number of camera generated on the y-axis. Figure 8 shows that with 10 cameras, the MCAP data recorded rate drops to 81% from 100%, while SPEED maintains a rate over 97% with 12 cameras, approximately 1.5× more sensor capacity. As anticipated, MCAP's single queue buffer suffers when multiple concurrent inputs share the same buffer. Overall, SPEED's proposed multi-queue architecture efficiently utilizes the CPU asynchronously, ensuring that the overall CPU usage remains optimized even under high sensor load conditions.

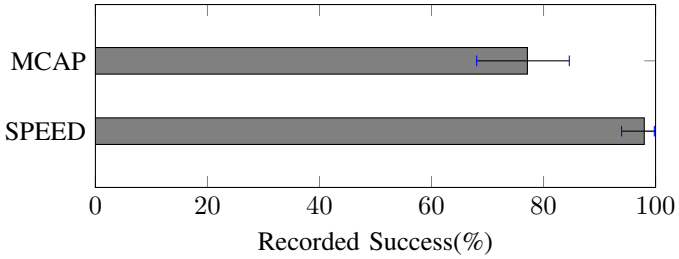


Fig. 9: Comparison of Data Recorded Success Rates of Multiple Different Types of Sensors

3) Multiple ROS Topic I/O Scalability: Lastly, we evaluate a workload involving multiple sensors streaming and saving data concurrently across several topics without computation applications. Extending the previous 8-camera benchmark, we add 1 Main LiDAR, 4 Secondary LiDAR, 5 Radar, 1 GPS, 1 IMU, and 12 Ultrasonic sensors (details in Table II). Sensors of the same type share the same topic. As shown in Figure 9, MCAP's *rosbag* recording rate drops from 100% to 77.1%, while SPEED maintains 97.9%, showing almost no performance degradation.

Our evaluation shows that SPEED not only reduces single I/O submission latency by 84% but also increases system scalability by 1.5×, enabling the system to handle more sensors concurrently without performance degradation. This increased scalability is crucial in autonomous systems where multiple sensors generate vast amounts of data simultaneously, and processing delays can impact the system's overall performance and safety.

D. Variability

Although we successfully improved the I/O performance and scalability of *rosbag*, it is essential to evaluate the variability of these improvements and their applicability in systems with DDS and computational workloads. To assess variability, we conduct two experiments: (1) first we compare the performance of the context-aware scheduler and MCAP/OS for a simple two-node application, and (2) then we evaluate the variability of our context-aware scheduler when applied to a real-world AV scenario.

1) Variability of I/O Task Completion: To demonstrate the variability of the existing *rosbag*'s performance, we implement a simple two-node application where one is a computation

node and the other is a I/O submission node to *rosbag*. Figure 10 demonstrates the variability of CPU-bound task execution (10 ms) over time under different I/O scheduling strategies (MCAP/OS vs. SPEED+sched). The y-axis represents the number of CPU tasks completed per time interval(33 ms), highlighting performance consistency. The MCAP/OS (blue line) shows significant performance variability due to unpredictable I/O submission and writeback operations, leading to inconsistent CPU availability. In contrast, the SPEED+sched (orange line) demonstrates less variability, maintaining a stable task completion rate over time by scheduling I/O during CPU idle periods. Performance variability, as measured by standard deviation, is 3.08× higher for MCAP/OS, indicating less predictable execution.

2) Variability of End-to-End AV pipeline performance: We construct an end-to-end AV workload that involves the saving of large amounts of sensor data and the execution of various application functions [3], [21]. Figure 11 compares three driving scenes: a 106-second passage through a shaded commercial alley with pedestrians and bicycles (Scene A), a 48-second right turn and signal wait on a quiet road with parked cars (Scene B), and a 66-second drive through a residential area with parked cars and garbage bins (Scene C). The figure shows the normalized standard deviation of the end-to-end operation completion latencies for each scene. MCAP is excluded due to significant data loss (10-30%), likely caused by its inability to handle concurrent I/O requests at high number of sensor loads. In contrast, our SPEED configuration successfully saved all sensor data without any loss.

Figure 11 demonstrates that context-aware scheduling improves predictability by reducing variability on the standard deviation of end-to-end latencies in all three scenarios, compared to the baseline SPEED configuration. This improvement is primarily due to the context-aware I/O scheduler's ability to align I/O with idle CPU periods, minimizing resource contention between I/O and computation tasks. However, this decrease in variability does not directly result in overall performance improvements in the AV pipeline, as seen in Figure 10. A likely reason for this is that while I/O scheduling enhances reliability, inconsistencies in the application and network resource usage persist, limiting overall performance gains. Further optimization, particularly in harmonizing I/O scheduling with application resource allocation, will be crucial for achieving more substantial performance improvements in future work.

V. Related Works

Efficient I/O management is essential for autonomous systems that handle large volumes of real-time sensor data [1]–[8]. The ROS framework and its *rosbag* tool are widely used but suffer from scalability and predictability limitations due to its single queue architecture, causing scalability and latency problems with concurrent sensor streams [18], [24], [25]. Zhang et al. proposed BORA to improve data acquisition by reorganizing data by topic [10], but focuses on post-processing rather than realtime I/O management, making it less suitable for autonomous systems [11].

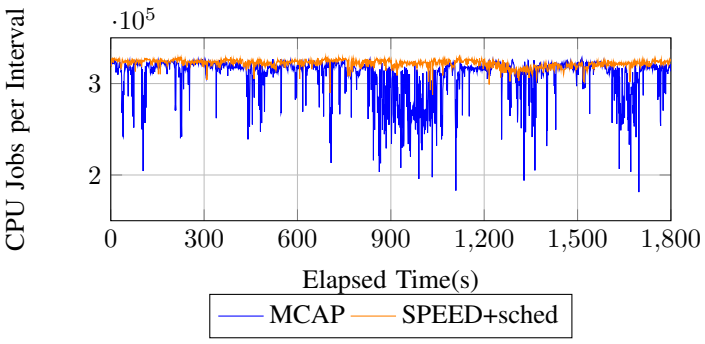


Fig. 10: Performance Variability Comparison in Connected I/O Submission and CPU-Bound Applications Over Time

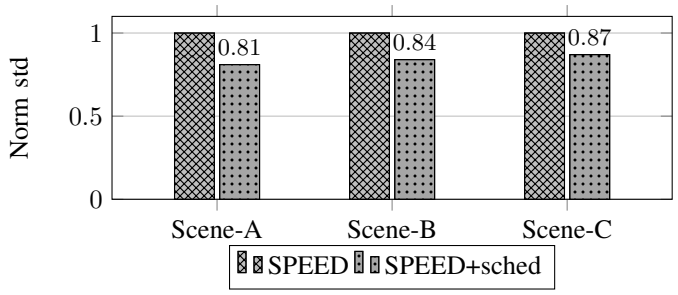


Fig. 11: End-to-End Latency Variability Comparison Between SPEED with or without Context-Aware Scheduling In Three Different Road Scene Scenario

Jiang [26] introduced a real-time I/O system for many-core embedded systems, addressing the need for predictable I/O performance in systems with stringent real-time requirements. Their work emphasizes the importance of coordinating I/O operations and prioritizing tasks based on realtime constraints, which is relevant to autonomous systems. Incorporating real-time I/O scheduling techniques from Jiang's research complements our context-aware I/O scheduler by enhancing timing predictability and reducing latency variability.

Recent research has also improved I/O performance using multi-queue architectures [18] and Asynchronous I/O mechanisms like io_uring [17] effectively reducing bottlenecks in concurrent data environments. While application-level scheduling has been used to enhance AV system predictability [27]–[32], I/O scheduling remains less explored [33], [34]. SPEED builds on these advancements by introducing a multi-queue architecture and a context-aware I/O scheduler tailored to the realtime, high-throughput requirements of autonomous systems, addressing current limitations in handling large amounts of sensor data.

VI. Conclusion

We presented SPEED: a novel I/O management architecture for autonomous systems, addressing the limitations of the scalability and predictability of *rosbag*. The SPEED architecture mitigates single-queue bottlenecks, improving an autonomous vehicle system's ability to handle high-throughput sensor inputs and reducing I/O latency by 84%. Our empirical results demonstrate support for 1.5× more sensors while maintaining

a 97% recording success rate. Additionally, by integrating a context-aware I/O scheduler, we align writeback operations with application cycles and improve predictability by reducing the variance in standard deviation of pipelined application performance by 3.08×. These enhancements significantly improve *rosbag*'s scalability and predictability, which are critical for safety and performance in autonomous operations. While our multi-queue architecture enhances throughput and scalability, it also poses additional challenges such as increased memory overhead and scheduling complexity. In future work, we plan to explore dynamic or adaptive queue allocation strategies that better balance resource usage and preserve predictable performance, especially in large-scale, distributed autonomous environments with numerous sensors. We will also validate our system under real-world conditions and incorporate advanced resource management techniques to further improve overall system performance and predictability.

REFERENCES

[1] Xin Xia, Zonglin Meng, Xu Han, Hanzhao Li, Takahiro Tsukiji, Runsheng Xu, Zhaoliang Zheng, and Jiaqi Ma. An automated driving systems data acquisition and analytics platform. *Transportation research part C: emerging technologies*, 151:104–120, 2023.

[2] Yuxin Wang, Yuankai He, Ruijun Wang, and Weisong Shi. Quantitative analysis of storage requirement for autonomous vehicles. In *Proceedings of the 16th ACM Workshop on Hot Topics in Storage and File Systems*, pages 71–78, 2024.

[3] Emil Talpes, Debjit Das Sarma, Ganesh Venkataramanan, Peter Bannon, Bill McGee, Benjamin Floering, Ankit Jalote, Christopher Hsiong, Sahil Arora, Atchyuth Gorti, et al. Compute solution for tesla's full self-driving computer. *IEEE Micro*, 40(2):25–35, 2020.

[4] Pratik Vyavahare, Sivaranjani Jayaprakash, and Krishna Bharatia. Construction of urdf model based on open source robot dog using gazebo and ros. In *2019 Advances in Science and Engineering Technology International Conferences (ASET)*, pages 1–5. IEEE, 2019.

[5] Mohsen Sombolestan, Yiyu Chen, and Quan Nguyen. Adaptive force-based control for legged robots. In *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 7440–7447. IEEE, 2021.

[6] Siqi Yi, Stewart Worrall, and Eduardo Nebot. A persistent and context-aware behavior tree framework for multi sensor localization in autonomous driving. *arXiv preprint arXiv:2103.14261*, 2021.

[7] Miguel Alcon, Hamid Tabani, Jaume Abella, and Francisco J Cazorla. Dynamic and execution views to improve validation, testing, and optimization of autonomous driving software. *Software Quality Journal*, 31(2):405–439, 2023.

[8] Hamid Tabani, Roger Pujol, Miguel Alcon, Joan Moya, Jaume Abella, and Francisco J Cazorla. Adbench: benchmarking autonomous driving systems. *Computing*, 104(3):481–502, 2022.

[9] Anis Koubâa et al. *Robot Operating System(ROS).*, volume 1. Springer, 2017.

[10] Jian Zhang, Tao Xie, Yuzhuo Jing, Yanjie Song, Guanzhou Hu, Si Chen, and Shu Yin. Bora: a bag optimizer for robotic analysis. In *IEEE SC20*, pages 1–15, 2020.

[11] Teng Wang, Kathryn Mohror, Adam Moody, Kento Sato, and Weikuan Yu. An ephemeral burst-buffer file system for scientific applications. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 807–818. IEEE, 2016.

[12] Alexander J Fiannaca and Justin Huang. Benchmarking of relational and nosql databases to determine constraints for querying robot execution logs. *Computer Science & Engineering*, pages 1–8, 2015.

[13] Young-Kuk Kim and Sang H Son. Predictability and consistency in real-time database systems. *Advances in real-time systems*, pages 509–531, 1995.

[14] André Dietrich, Siba Mohammad, Sebastian Zug, and Jörg Kaiser. Ros meets cassandra: Data management in smart environments with nosql. In *Proc. of the 11th Intl. Baltic Conference (Baltic DB&IS)*. Citeseer, 2014.

[15] Markus Schratter et al. From simulation to the race track: Development, testing, and deployment of autonomous racing software. In *2023 IEEE International Automated Vehicle Validation Conference (IAVVC)*, 2023.

[16] Young Jin Yu, Dong In Shin, Woong Shin, Nae Young Song, Jae Woo Choi, Hyeong Seog Kim, Hyeonsang Eom, and Heon Young Yeom. Optimizing the block i/o subsystem for fast storage devices. *ACM Transactions on Computer Systems (TOCS)*, 32(2):1–48, 2014.

[17] Kanchan Joshi, Anuj Gupta, Javier González, Ankit Kumar, Krishna Kanth Reddy, Arun George, Simon Lund, and Jens Axboe. {I/O} passthru: Upstreaming a flexible and efficient {I/O} path in linux. In *22nd USENIX Conference on File and Storage Technologies (FAST 24)*, pages 107–121, 2024.

[18] Matias Bjørling, Jens Axboe, David Nellans, and Philippe Bonnet. Linux block io: Introducing multi-queue ssd access on multi-core systems. In *Proceedings of the 6th international systems and storage conference*, pages 1–10, 2013.

[19] Viktor Leis, Adnan Alhomssi, Tobias Ziegler, Yannick Loeck, and Christian Dietrich. Virtual-memory assisted buffer management. *Proceedings of the ACM on Management of Data*, 1(1):1–25, 2023.

[20] Alastair Robertson, Brendan Gregg, et al. bpftrace - high-level tracing language for linux. https://github.com/bpftrace/bpftrace.

[21] Biswadip Maity, Saehanseul Yi, Dongjoo Seo, Leming Cheng, Sung-Soo Lim, Jong-Chan Kim, Bryan Donyanavard, and Nikil Dutt. Chauffeur: Benchmark suite for design and end-to-end analysis of self-driving vehicles on embedded systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 20(5s):1–22, 2021.

[22] Andreas Geiger, Philip Lenz, Christoph Stiller, and Raquel Urtasun. Vision meets robotics: The kitti dataset. *The International Journal of Robotics Research*, 32(11):1231–1237, 2013.

[23] Michael Orlov et al. Rosbag2 writer benchmarking, 2021. https://github.com/ros2/rosbag2/tree/humble/rosbag2_performance/rosbag2_performance_benchmarking.

[24] Wonse Jo, Shyam Sundar Kannan, Go-Eum Cha, Ahreum Lee, and Byung-Cheol Min. Rosbag-based multimodal affective dataset for emotional and cognitive states. In *2020 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 226–233. IEEE, 2020.

[25] Yongbon Koo and SungHoon Kim. Distributed logging system for ros-based systems. In *2019 IEEE International Conference on Big Data and Smart Computing (BigComp)*, pages 1–3. IEEE, 2019.

[26] Zhe Jiang. *Real-time i/o system for many-core embedded systems*. PhD thesis, University of York, 2018.

[27] Saehanseul Yi, Tae-Wook Kim, Jong-Chan Kim, and Nikil Dutt. Easyr: E nergy-efficient a daptive sy stem r econfiguration for dynamic deadlines in autonomous driving on multicore processors. *ACM Transactions on Embedded Computing Systems*, 22(3):1–29, 2023.

[28] Nora Sperling, Alex Bendrick, Dominik Stöhrmann, Rolf Ernst, Bryan Donyanavard, Florian Maurer, Oliver Lenke, Anmol Surhonne, Andreas Herkersdorf, Walaa Amer, et al. Information processing factory 2.0-self-awareness for autonomous collaborative systems. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6. IEEE, 2023.

[29] Haohao Zhang, Kyosuke Watanabe, Kazuhiro Motegi, and Yoichi Shiraishi. Ros based framework for autonomous driving of agvs. *Proceedings of the IPS6-04, ICMEMIS, Kiryu, Japan*, pages 4–6, 2019.

[30] Yasuhiro Nitta, Sou Tamura, and Hideki Takase. A study on introducing fpga to ros based autonomous driving system. In *2018 International Conference on Field-Programmable Technology (FPT)*, pages 421–424. IEEE, 2018.

[31] Dongjoo Seo, Biswadip Maity, Ping-Xiang Chen, Dukyoung Yun, Bryan Donyanavard, and Nikil Dutt. Proswap: Period-aware proactive swapping to maximize embedded application performance. In *2022 IEEE International Conference on Networking, Architecture and Storage (NAS)*, pages 1–4. IEEE, 2022.

[32] Dongjoo Seo, Ping-Xiang Chen, Changhoon Sung, Quang Anh Hoang, Adam Manzanares, and Nikil Dutt. Memscape: Sculpting tiered memory management for autonomous vehicles. *ACM Transactions on Embedded Computing Systems*.

[33] Luca Belluardo, Andrea Stevanato, Daniel Casini, Giorgiomaria Cicero, Alessandro Biondi, and Giorgio Buttazzo. A multi-domain software architecture for safe and secure autonomous driving. In *2021 IEEE 27th international conference on embedded and real-time computing systems and applications (RTCSA)*, pages 73–82. IEEE, 2021.

[34] Julius Ziegler and Christoph Stiller. Fast collision checking for intelligent vehicle motion planning. In *2010 IEEE intelligent vehicles symposium*, pages 518–522. IEEE, 2010.