Programming Abstractions & Systems for Autonomous Vehicles

By

Sukrit Kalra

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Ion Stoica, Chair
Professor Joseph E. Gonzalez
Professor Prabal Dutta
Professor Alexey Tumanov

Summer 2024

Programming Abstractions & Systems for Autonomous Vehicles

Abstract

Programming Abstractions & Systems for Autonomous Vehicles

by

Sukrit Kalra

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Ion Stoica, Chair

Autonomous Vehicles (AVs) have the potential to revolutionize transportation through their significant safety, environmental and mobility benefits. However, despite their benefits and significant investment spanning over a decade, AVs remain restricted to locations with favorable driving conditions, due to the complications arising from the long-tail of complex driving scenarios. Most research has sought to address this critical challenge through the design of robust algorithms and machine learning (ML) models that underpin the various decision making components of a modern AV computational pipeline. In contrast, there has been relatively little focus on the software systems that must orchestrate an efficient, real-time execution of these components on the AV's constrained, heterogeneous hardware.

This dissertation examines the often-overlooked design of such software systems and presents a clean slate approach to developing AVs. We introduce D3, a novel programming model for AVs that enables the computation to proactively adjust to *dynamically-varying* deadlines and models missed deadlines as *exceptions*. We realize D3 in our open-source system, ERDOS, whose novel extensions to concepts from streaming data systems enable it to speculatively execute computation and enforce deadlines between an arbitrary set of events. ERDOS's efficient execution of AV pipelines is further enabled by two key scheduling contributions of this dissertation: SuperServe and DAGSched. SuperServe unlocks a resource-efficient serving of the entire range of ML models spanning the latency-accuracy tradeoff space, enabling AV pipelines to quickly adjust to dynamically-varying deadlines. In addition, DAGSched efficiently multiplexes the available compute resources in an AV amongst the decision making components, with an aim to maximize the ability of the AV pipeline to meet dynamically-varying deadlines. Finally, we address the crucial lack of AV benchmarks by providing the first completely open-source AV pipeline, Pylot, and use it to evaluate the positive effects of D3 and ERDOS on the driving safety of AVs. Together, these systems span the entire workflow of developing and evaluating AVs, and we believe are crucial to bridging the gap towards achieving "fully autonomous vehicles".

To my family.

Contents

List of Figures

List of Tables

List of Algorithms

Acknowledgments

This dissertation is the culmination of the collective efforts of many individuals, whose invaluable support, astute guidance and unwavering belief have made it possible. I am profoundly grateful for everyone who has contributed in their own unique ways during the rewarding process of this dissertation.

This work would not have been possible without my advisor, Ion Stoica, whose foresight initiated the research that culminated in this dissertation. Throughout my PhD journey, Ion has constantly supported every endeavor of mine with patience. He has given me space to pursue my own ideas, and provided invaluable insight when I needed. Ion's relentless pursuit of simplicity and his supportive style of advising has positively inspired the way I approach research. I am profoundly thankful for his guidance and mentorship.

I would like to especially thank Joseph Gonzalez and Alexey Tumanov, whose constant enthusiasm for my ideas and invaluable insights made this research possible. I have greatly enjoyed working closely with them and shared many joyous moments debating with them over small yet significant topics. I am also immensely grateful to Raluca Ada Popa, whose belief in me as a first-year PhD student provided me the strength to take on challenging projects, while adjusting to a new environment. Her enthusiasm and deep understanding of her research topics consistently lead to engaging conversations, and I gained invaluable insights while teaching under her guidance. Finally, I have had many insightful conversations with Joseph Hellerstein, Hong Zhang, Ken Goldberg, Prabal Dutta, Dave Culler, Natacha Crooks and Koushik Sen, all of whom have made my time at Berkeley memorable.

I was fortunate to find an amazing group of long-term collaborators in the ERDOS team, whose shared values and enthusiasm made the PhD process much easier. Thank you, Ionel Gog, for your mentorship and friendship (and for introducing me to British panel shows!). I am also grateful to Peter Schafhalter for sharing the highs and lows of the entire process with me, and for always bringing his optimism about our work. A special mention to Rishabh Poddar, whose support both before and during my PhD was immeasurable. Thank you for taking me under your wing, I have greatly enjoyed your friendship and working with you!

The PhD process would have been immeasurably harder without the support of the RISE, Sky and EECS staff, who made sure that we could focus on our research and took care of everything else. Thank you, Jean Nguyen, Kattt Atchley, Boban Zarkovich, Ivan Ortega, Jon Kuroda, Shane Knapp, Kailee Truong and Dave Schonenberg!

I would like to convey my sincerest gratitude to my mentors at IBM Research and IIIT-

Delhi, who profoundly shaped my research interests. Thank you, Mohan Dhawan, for your continued mentorship. I will always cherish the time we spent iterating on papers for days on end! I would also like to thank Rahul Purandare, Subodh Sharma and Vinayak Naik for giving me the opportunities and support that I needed to succeed as an undergraduate.

The amazing people of RISE and Sky lab have provided laughter, friendship and unwavering support throughout my PhD. First and foremost, I will always cherish the friendship of Romil Bhardwaj. His calm and collected approach to life is an inspiration, and I could not have found a better person to share an apartment with during my PhD (especially during the pandemic!) I would also like to thank Tianjun Zhang, Shishir Patil and Jiwon Park, who shared my enthusiasm for meals, and provided comedic relief at a moment's notice! Thank you, Conor and Laura Power, for your friendship. We all appreciate your ability to form an inclusive community wherever you go! Thank you, Lisa Dunlap, for teaching me something new and exciting in every conversation. I would also like to thank Shreya Shankar for her willingness to go on long walks and provide effortless conversation throughout. Additionally, I am also grateful to Moustafa AbdelBaky, Weikeng Chen, David Chu, Chris Douglas, Vivian Fang, Anand Iyer, Alind Khare, Shadaj Laddad, Richard Liaw, Kevin Lin, Pratyush Mishra, Charles Packer, Suzanne Petryk, Deevashwer Rathee, Mayank Rathee, Daniel Rothchild, Eyal Sela, Manish Shetty, Stephanie Wang, Jean-Luc Watson, Michael Whittaker, Justin Wong, Sarah Wooders, Wen Zhang and Wenting Zheng for their friendship!

A special mention to my school friends who made sure that I never felt too far away from home: Nishchay Budhiraja, Ila Jain, Rishabh Jetly, Raveesh Kalra, Prateek Kohli, Tarun Monga, Gaurav Nanda, and Pulkit Verma. Thank you for your lifelong friendship!

I could not have done this without my partner and best friend, Carson Fox Young. Your selfless love and support has given me the strength to tackle bigger challenges, and taught me how to be a better person in immeasurable ways. You're the reason I look forward to coming home every day. Thank you for being who you are, saying yes to adventures, for standing for hours in the rain to watch Bruce Springsteen, and for going through life's ups and downs with me. A special thanks for bringing Sky into my life, who wags her tail everytime she sees me, and who turned me into a dog person. I love you, I hope I can reciprocate all that you've given me and I look forward to sharing the rest of my life with you!

I am also immensely grateful to Carson's parents: Robert and Colleen, who welcomed me into their home and their hearts, and who continue to positively shape my approach to life, while providing laughter, love (and food!) along the way. Thank you for making sure I have a home away from home. Finally, I want to thank my parents, Namita and Sudeep Kalra. You have been the best role models I could have asked for in my life, and have always spurred me to follow my dreams while providing a safe place to call home. I would not be who I am without you! Thank you to my sister, Stuti, who graciously took on all the responsibilities that I left behind. I could not have asked for a more understanding and caring sibling. Thank you to my grandparents, who raised me and instilled their values in me, and who cheer me on in everything I do.

Chapter 1

Introduction

Autonomous Vehicles (AVs) are poised to revolutionize transportation through their significant safety, environmental, economic and societal benefits [208, 211]. In the United States alone, the National Highway Traffic Safety Administration (NHTSA) expects AVs to: (*i*) reduce human error from traffic accidents, which made up for 94% of the 42,514 vehicle related deaths in 2022 [212, 209], (*ii*) increase traffic flow, which could free upto 50 minutes per person per day [202, 208], and (*iii*) provide employment to around 2 million people with disabilities [76]. Despite these potential benefits and significant investments [56, 164, 89], "fully autonomous vehicles" that are able to drive themselves without human intervention remain restricted to locations with favorable driving conditions [299, 227, 325, 163], due to the complications arising from the long-tail of complex driving scenarios [26, 171, 251].

The complexity of the wide-range of driving scenarios encountered in production presents unique challenges across the three major layers of an AV's computational stack (illustrated in Fig. 1.1 and detailed in §1.1). Most research addresses these challenges at the top of the stack, focusing on designing robust algorithms and machine-learning (ML) models that underpin various components of a modern AV pipeline [289, 147, 320, 257, 276]. However, there has been comparatively little focus on the software systems that orchestrate an efficient, real-time execution of these components. This dissertation examines the often-overlooked design of such software systems and argues that it both presents novel distributed systems challenges and is crucial to bridging the gap towards achieving "fully autonomous vehicles".

1.1 Computational Stack of an Autonomous Vehicle

We begin our discussion by providing a high-level overview of the computational stack that enables the decision making in an AV, and consists of three layers (see Fig. 1.1):

1. **Hardware:** This layer forms the foundation of the AV stack, facilitating the sensing of the driving environment. AVs are equipped with a large number of sensors (e.g., dozens of cameras, several LIDARs and radars), all of which help improve algorithmic accuracy and increase sensor redundancy in case of hardware failures [150, 145, 146]. These

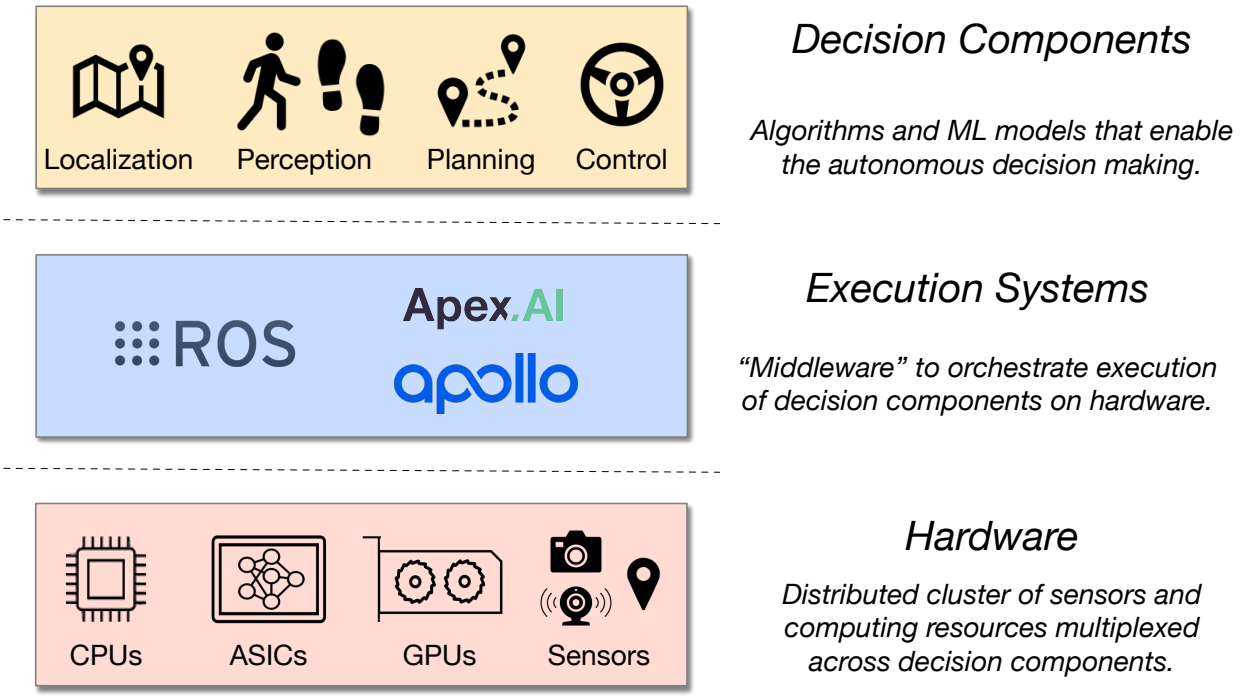


Figure 1.1: Computational Stack of an AV is made up of three layers: (*i*) a distributed cluster of heterogeneous resources, (*ii*) the middleware that orchestrates execution of decision components, and (*iii*) the algorithms and ML models that drive the decision making.

sensors generate 1–2 GB/s of data [113, 312, 22] (see Table 1.1). This data must be processed in real-time through multiple algorithms and ML models that drive decision-making components at the top of the stack, leading to computation requirements that are at least 100× higher than those of the vehicles in production before [217].

To support the execution of these algorithms and ML models, the hardware layer provides a distributed cluster of heterogeneous computing resources [186, 200]. The heterogeneity aids in the low-latency execution of decision-making components by utilizing specialized hardware (e.g., GPUs, ASICs etc.). The distributed nature of the hardware layer enables AVs to address hardware failures effectively (e.g., by deploying critical components on independent power systems, and using redundant hardware to perform emergency maneuvers in the event of a failure [103, 318, 113]).

2. **Execution Systems:** The hardware resources in an AV are inherently limited by physical factors such as weight distribution, cooling, and power constraints [186]. As a result, these resources must be efficiently multiplexed across the various decision-making components at the top of the stack. The execution system layer provides the "middleware" that is used to orchestrate the execution of the various components of an AV's computation across its distributed, heterogeneous computing resources.

Hardware	Quantity	Frequency (Hz)	Bandwidth (MB/s)
Backfly Camera	8	22	365
IMX390CQV Camera	8	60	960
VLS-128 LIDAR	5	10	75
Radar	21	40	8

Table 1.1: An AV's sensor suite contains several types (e.g., Cameras, LIDARs etc.) of sensors executing at varying frequencies that generate 1–2 GB of raw data per second.

The current state-of-the-art AV pipelines (e.g., Autoware [35], Cruise [312], BMW [8] etc. [106, 311]) are built atop the Robot Operating System (ROS) [248]. ROS was designed as an execution platform for enabling robotics research, and achieves its key goal of supporting execution of complex computational stacks (such as the one visualized in Fig. 1.1 and Fig. 1.2) through its modular publisher-subscriber communication paradigm [7] and best-effort execution of components.

The goal of this dissertation will be to delve deeper into the requirements imposed on these execution systems by the other layers of an AV's computational stack and propose and evaluate the design of an ideal execution system.

3. **Decision Components:** The critical driving decisions of an AV are determined by a suite of five components: *perception*, *localization*, *prediction*, *planning*, and *control*. These components process the camera, radar, and LIDAR data, and generate steering and acceleration commands to physically operate the vehicle. Fig. 1.2 provides a detailed architecture of a representative state-of-the-art AV pipeline, and illustrates how these components are interconnected to transform sensor input into control output.

   Below, we discuss the roles and utility of each of the the aforementioned components. We also highlight the multitude of solutions available for these components that are specialized towards deployment scenarios, available compute resources or runtime:

   - **Perception:** To ensure safe navigation, the AV must accurately detect obstacles and track their movement in real-time. To achieve this, the perception component is split across: (*i*) *detection*, which applies ML models to detect lanes, obstacles, and traffic lights etc. [289, 147], and (*ii*) *tracking*, which uses algorithms and ML models to assign identifiers to detected obstacles [346, 326, 48].
   - **Localization:** The localization module is tasked with triangulating a decimeter-level accurate location of the AV on a high-definition map stored offline on the AV. While individual sensors are prone to occlusion and drift, the module achieves its goal by fusing data from multiple sensors (e.g., GNSS, IMU, LiDAR etc.) using algorithms such as Extended Kalman Filtering [262, 345] and Particle Filters [33, 243]. The output from localization and perception is fused to create an abstract representation of the real-world, which is then provided as input to prediction.

- **Prediction:** Upon the abstract representation of the real-world, the AV deploys ML models to predict the future trajectories of detected obstacles. The prediction component uses the past trajectory of each obstacle to propose a set of predicted future trajectories along with the probability of taking a trajectory. Various solutions account for more complex interaction amongst the obstacles at the expense of increased runtime [180, 256, 67, 291]. Moreover, distinct solutions cater to the prediction of trajectories for specific agents (e.g., pedestrians [344, 14, 258], vehicles [157, 81]) and scenarios (e.g., intersections [107, 24], highways [320, 97]).
- **Planning:** The abstract representation of the real-world around the AV is annotated by the prediction component with the predicted future trajectories for obstacles, and then used by planning algorithms in order to produce a *safe* and *comfortable* motion plan consisting of decimeter-level waypoints [230]. Although optimal yet tractable planning algorithms remain unavailable [252], the multitude of available algorithms [329, 169, 268] enable developers to optimize the accuracy of the plan with respect to the available runtime. Moreover, "anytime algorithms" [170] that iteratively refine their results enable this choice to occur at runtime by allowing their executions to be interrupted in order to retrieve the most accurate motion plan available by that time.
- **Control:** Finally, specific control algorithms tailored to particular speeds and conditions [230, 109, 78, 167, 265] stabilize the computed waypoints to generate steering and acceleration commands for the underlying machinery in the AV.

The computational complexity of the decision components, coupled with the relative scarcity of the underlying hardware, presents unique challenges on the execution systems that lie in the middle layer. These challenges necessitate a synergistic development approach between the decision components and the underlying execution systems. In §1.2, we provide a taxonomy of the distinct types of requirements that AV computational pipelines impose on these software systems. Additionally, §1.3 discusses the contributions of this dissertation in meeting those requirements.

1.2 Requirements from AV Execution Systems

As discussed in §1.1, an AV pipeline is structured as a distributed graph of components, where each component may have an arbitrary number of statically defined task dependencies with sporadic input arrivals. The execution of this pipeline presents unique requirements on the set of *programming abstractions* that are provided for the development of decision-making components and their efficient *execution* to ensure the operational safety of the vehicle. We now discuss these two set of requirements separately in §1.2.1 and §1.2.2.

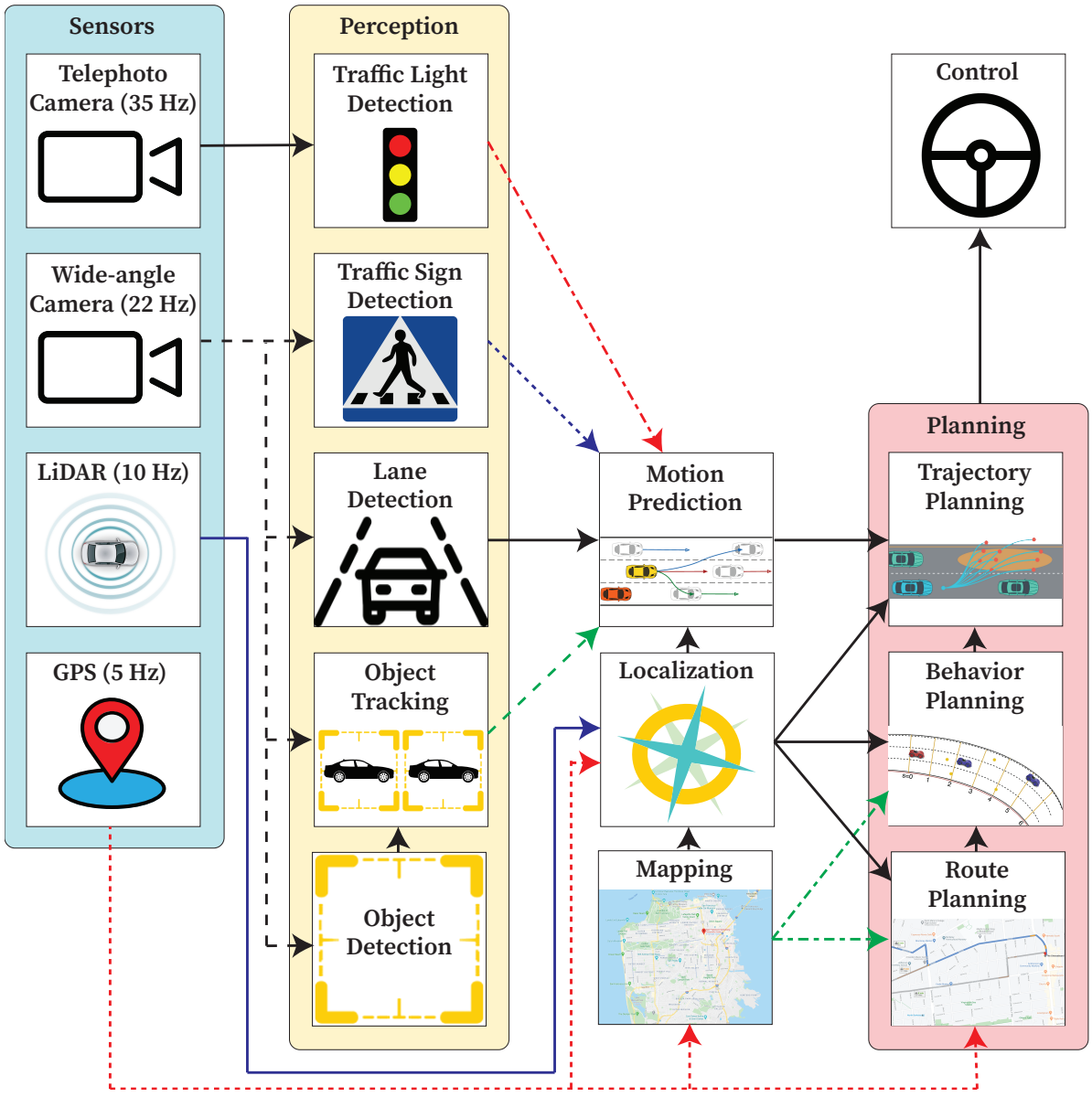


Figure 1.2: A detailed architecture of a modular, state-of-the-art AV computational pipeline. A modern AV uses multiple sensors to perceive the environment around it. These sensor readings are used by the *perception* module to detect other agents, and by the *localization* module to compute the location of the AV itself. The *prediction* module uses their output to predict the future trajectories of other agents, and the *planning* module computes a safe and feasible trajectory for the AV using these predictions. Finally, the *control* module produces steering and acceleration commands for the underlying machinery.

1.2.1 Programming Abstraction Requirements

AVs have reliability, safety, and real-time requirements similar to hard real-time systems (e.g., fighter jets), and processing and throughput demands similar to those found in big-data systems. Simultaneously satisfying these demands in the context of AVs presents two unique programming requirements that are often at odds with each other:

- **Deadline Support:** The safety-critical nature of an AV requires the execution systems to provide primitives to ensure the fulfillment of the end-to-end deadline by allowing each component (or a set of components) to specify deadlines that: (*i*) trigger the start of computation even in the presence of arbitrary delays or failures in upstream components, and (*ii*) bound the response time of components with unpredictable runtimes.

- **Rapid Innovation:** The components that underpin AVs are undergoing innovation at a breathtaking pace, with the top 10 entries in the KITTI detection challenge constantly evolving [111], and Tesla constantly updating their AutoPilot software [295, 151]. While the hardware on an AV remains fixed due to the laborious processes required to meet the automotive regulations [186], the resource requirements of the components constantly evolve [147, 289]. We expect this trend to continue due to the need to update algorithms to better handle the increasingly-diverse scenarios encountered in production, and thus require a constant reconfiguration of the computation to achieve the highest accuracy within the limits of the AV's available hardware.

  This constant reconfiguration of the AV pipeline requires *modularity* of each component as the addition of new algorithms might require reallocation of response times and computational resources. In some cases, innovation in one component (e.g., path planning) might shift the optimal algorithm for another component (e.g., object detection) towards a faster yet less-accurate one, thus requiring the pipeline to be restructured.

The rapid innovation in AV's components is at odds with the rigorous engineering practices used to develop prior secure and reliable hard real-time systems (e.g., nuclear plants, fighter jets etc.) that provide support for deadlines in several ways: (*i*) AVs depend on external large codebases that are not hard real-time (e.g., TensorFlow [3], PyTorch [236]), (*ii*) new algorithms are often validated on the public infrastructure since testing all possible behaviors in simulation is impossible given the multitude of situations an AV can encounter, and (*iii*) unlike airplanes that rely on human pilots as fallback mechanisms, fully autonomous vehicles must handle all edge cases safely without any support.

1.2.2 Execution System Requirements

The software systems must orchestrate the execution of the AV pipeline that is programmed within the context of the requirements in §1.2.1 while ensuring the operational safety of the AV. This leads to the following three requirements:

- **Performance:** The vast amount of data generated by the sensors coupled with the need to quickly compute a control decision [186, 311, 330] requires distributed execution across machines and accelerators to enable the *high throughput* data to be processed with *low latency*. Crucially, this low-latency execution must adhere to an end-to-end deadline decided by the components.
- **Determinism:** Vendors have underscored the need to enable a *deterministic replay* of the execution of an AV pipeline [318, 312] in order to allow a reproduction of complex scenarios. This property aids: (*i*) *development* and *testing* of algorithms by replaying scenarios in simulation and preventing regressions [266, 255, 226], and (*ii*) *verification* of an AV to prevent bugs [168].
- **Fault Tolerance:** Regulations mandate that hardware and software are tolerant to failures [278, 149]. While hardware components achieve fault-tolerance through redundancy [156, 15], the software systems must provide a range of fault-tolerance mechanisms (e.g., active replication [271, 43, 127], checkpoint-replay [101, 9]) that conform to the state and runtime properties of each component.

1.3 Contributions

This dissertation focuses on the end-to-end lifecycle of developing an AV's computational pipeline (see Fig. 1.3). We redesign the often-overlooked layer of "*execution systems*" in Fig. 1.1, and propose a synergistic approach of developing AVs to address the requirements of §1.2. Specifically, the dissertation makes the following concrete contributions:

(**1**) We propose a new programming model, **D3** (**D**ynamic **D**eadline-**D**riven) in Chapter 2 for developing AVs. D3 departs from the conventional publisher-subscriber based model of development in modern AVs and envisions an AV pipeline as a static, directed graph of decision components. Moreover, D3 centralizes the management of deadlines into a *deadline policy*, and enables decision making components to exploit the multitude of available algorithms and ML models catering to different response times and driving environments (described in §1.1) to meet their assigned deadlines. Finally, D3 models missed deadlines as "*exceptions*" and proposes *exception handlers* to quickly react to these events.

(**2**) We present techniques to efficiently realise the D3 model in an open-source, high-performance execution system called **ERDOS** (Chapter 3). ERDOS provides novel extensions to the concepts of *watermarks* from streaming systems to expose fine-grained execution events for the specification of dynamically-varying deadlines that restrict the wall-clock time elapsed between such events. Moreover, ERDOS' speculative execution mechanisms aid the decision making components in fulfilling these deadlines by executing the appropriate implementation automatically. Finally, if deadlines are missed, ERDOS' automatic state management significantly eases the execution of D3's exception handlers that allow components to convey intermediate results and unlock the execution of downstream components.

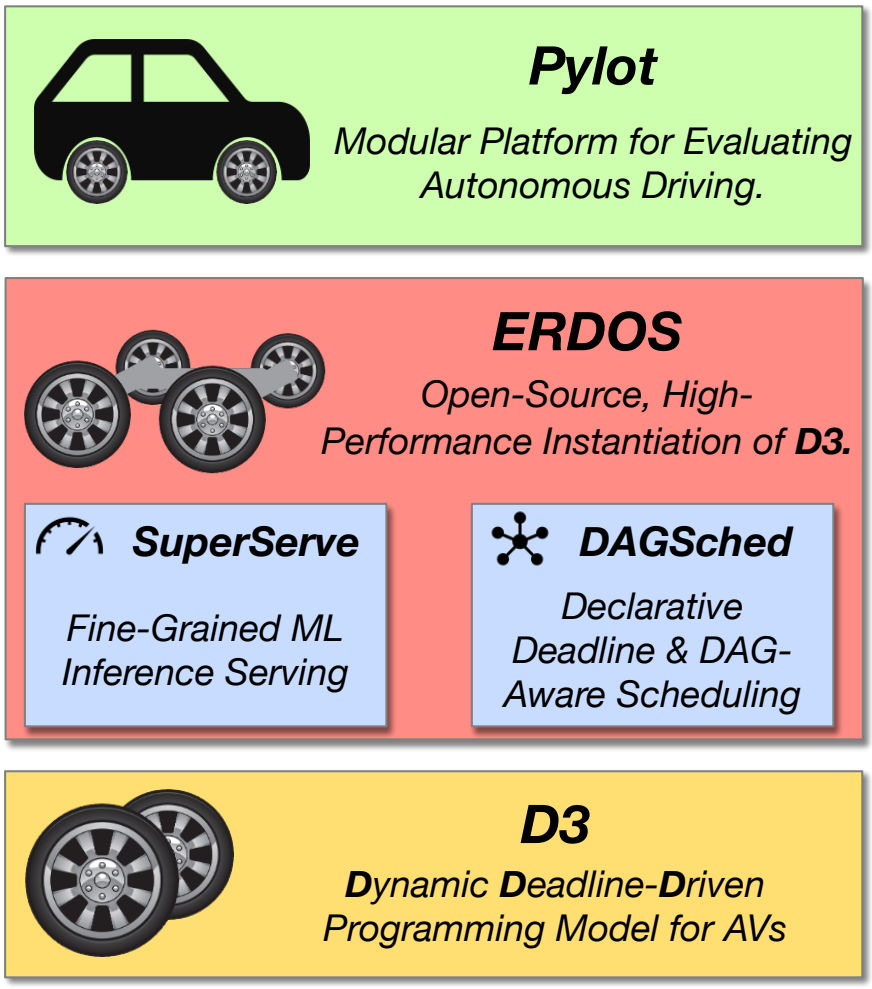


Figure 1.3: This dissertation's contributions span the entire workflow of developing and evaluating AVs: (*i*) D3 provides a new programming model to significantly ease AV pipeline development while providing support for dynamic deadlines, (*ii*) ERDOS provides an efficient, open-source realization of D3 using novel extensions to concepts from streaming systems, (*iii*) SuperServe and DAGSched efficiently schedule the AV's computational pipeline developed using D3, and (*iv*) Pylot provides a modular, open-source platform for evaluating the efficacy of components and execution systems on the end-to-end driving behavior of AVs.

(**3**) We address the crucial lack of AV benchmarks by providing the first completely open-source AV pipeline, **Pylot** (Chapter 4). Pylot's design goals of *modularity*, *portability* and *debuggability* enable it to work across simulators and real-vehicles, and provide a state-of-the-art platform for exploring latency-accuracy tradeoffs in decision making components and execution systems. We evaluate the efficacy of the dynamic deadline-driven execution of D3 and ERDOS by driving Pylot across 50kms of challenging driving scenarios, and observe a 68% reduction in collisions as compared to prior state-of-the-art programming models.

(**4**) The increasing proliferation of ML models in AV's decision making components at the top of the stack coupled with the relative scarcity of resources at the hardware layer on the bottom (Fig. 1.1) requires an efficient utilization of the available resources. This tension is exacerbated by the speculative execution mechanisms enabled by D3 and ERDOS that crucially rely on being able to efficiently serve ML models at low latency to enable components to meet their deadlines. To address this challenge, we propose **SuperServe** (Chapter 5) that dynamically inserts novel control-flow and slicing operators into SuperNet neural architectures for ML models. This allows SuperServe to dynamically route requests within one SuperNet deployment with negligible overhead, enabling near-instantaneous *actuation* of different ML models. We find that SuperServe achieves 4.67% higher accuracy for the same deadline attainment, and 2.85× higher deadline attainment for the same accuracy requirements, while requiring upto 2.6× lower memory to serve the ML models.

(**5**) An AV's sensor suite generates data at a higher frequency (see Table 1.1) than the processing rate of the computational pipeline shown in Fig. 1.2. As a result, an execution system like ERDOS must efficiently multiplex the available compute resources at the hardware layer amongst the various invocations of an AV's computational pipeline executing concurrently. The AV pipelines present three key scheduling requirements: (*i*) *precedence constraints*, due to the DAG-based structure of D3, (*ii*) *placement preferences*, to efficiently exploit the heterogeneous resources available in the AV, and (*iii*) *timing constraints*, that require the computational pipeline to finish within a deadline. To address these requirements, we propose **DAGSched** (Chapter 6), a framework for simplifying the development of efficient, solver-based schedulers through a declarative specification of job requirements in its novel language, **STRL++**. We evaluate DAGSched and achieve a 43.75% improvement in the number of concurrently executing jobs that achieve a 99% deadline attainment, and upto 6.42× increase in deadline attainment under high load, all while reducing the p90 latency of the scheduler by 60× over carefully hand-crafted mathematical models.

Finally, Chapter 7 concludes by discussing the key takeaways of our work and enumerates the lessons learnt during the process. In addition, we discuss future directions that remain unexplored and provide research avenues to enhancing the support for the next-generation of cyber-physical systems. Overall, this dissertation highlights the efficacy of a clean-slate approach to building the next-generation of cyber-physical systems by innovating through the entire workflow of developing a key example of such systems, *autonomous vehicles*.

Chapter 2

D3: Programming Abstractions for Autonomous Vehicles

To safely drive in complex environments, AVs must ensure highly-accurate results by executing complex pipelines with hundreds of computationally-intensive algorithms and ML models [312] using multiple parallel processors and hardware accelerators [186]. As discussed in §1.2.1, the software systems for AVs must support a *deadline-driven* execution of their computational pipelines. Achieving this goal in the presence of *rapidly evolving* components is complicated by the following two unique characteristics of AVs (discussed in §2.1):

C1: Environment-dependent deadlines. AVs need to *automatically* complete their computation at varying timescales to safely drive across the wide array of scenarios in the real-world. For example, navigating a crowded urban street requires different algorithms and can tolerate longer computation times than swerving in response to an obstacle on the freeway [75, 17]. While traditional hard real-time systems that interact with the environment appear to have similar requirements, they crucially rely on humans to initiate mode changes for different scenarios and ensure safety. For example, flight controllers [181, 1] rely on pilots to infrequently transition between takeoff, cruising, landing etc. [31].

C2: Environment-dependent runtimes. The runtime of various stages of an AV pipeline like pedestrian tracking vary with the input (e.g., the number of pedestrians). As a result, these stages exhibit *runtime-accuracy* tradeoffs that must be addressed *dynamically* according to the environment [74, 317]. Moreover, the reliance of various components on large external codebases that are not real-time [30] (e.g., Tensorflow [3], CUDA [218] etc.), and their execution on non real-time hardware such as GPUs, ASICs etc. [186, 85, 215, 238, 331, 100, 99], further present runtime variations that must be addressed dynamically.

The current state-of-the-art systems for autonomous driving (e.g., Autoware [35], Cruise [312], BMW [8] etc. [106, 311]) are built atop the Robot Operating System (ROS) [248]. ROS was designed as an execution platform for enabling robotics research, and achieves its key goal of supporting the construction of complex pipelines through its modular design [7] and best-effort execution of the stages. However, these systems lack mechanisms to specify

and enforce deadlines on the computation thus precluding a deadline-driven execution of an AV pipeline (**C1**), which is critical for vehicle safety.

Conversely, decades of work in cyber-physical systems has produced sophisticated techniques for safety-critical applications that ensure the fulfillment of strict deadlines [46, 47, 54, 123, 196, 92]. However, these techniques require a comprehensive, time-consuming analysis of the schedulability of the stages driven by estimates of their worst-case runtimes. Since various stages of the pipeline exhibit environment-dependent runtimes (**C2**) and rapidly evolve, there exists a wide variance between their average and worst-case runtimes. Thus, any schedulability analysis driven by the latter is overly-conservative and leads to an under-utilization of the compute resources, which could be used to execute higher-accuracy algorithms and optimize the runtime-accuracy tradeoff [74, 317] (elaborated in §2.2.1).

We posit that a new class of systems is required to enable a deadline-driven execution of applications that must interact with a continuously-evolving environment (e.g., robotics, AVs), and exhibit **C1-C2**. Such systems must combine the ease-of-development of state-of-the-art robotics platforms with the deadline specification and enforcement mechanisms of cyber-physical systems. Specifically, such systems must enable applications to specify deadlines that evolve with the environment (**C1**), and adapt their computation to such deadlines to maximize the runtime-accuracy tradeoff (**C2**).

Our work seeks to provide a general execution model for such applications. Thus, we propose D3 (**D**ynamic **D**eadline-**D**riven), an execution model for applications that interact with a continuously-evolving environment, and exhibit **C1-C2**. D3 decomposes the application as a graph of computation along with a *deadline policy*, which determines the deadline according to the environment (**C1**). While applications proactively try to meet deadlines, D3 models missed deadlines due to **C2** as *exceptions* and allows the execution of *reactive measures*. Further, D3 notifies downstream computation about missed deadlines, allowing it to *eagerly execute* on incomplete input or adjust to fit in the reduced time upon arrival of the input (see §2.2). This chapter elaborates on the two key contributions made by D3:

- We introduce and underscore the importance of the two characteristics of applications that interact with a continuously-evolving environments (**C1-C2**) by analyzing data collected from our own AV and the sensor data of a state-of-the-art AV vendor (§2.1).
- We elaborate on D3, a novel execution model that enables such applications to maximize their accuracy in the presence of the continuously-evolving environment (§2.2).

2.1 Motivation: Defining Characteristics of AVs

We have previously discussed the computational stack that drives an AV's decision making in §1.1. We begin here by noting that it is imperative that while the pipeline shown in Fig. 1.2 produces accurate results, it also computes them within a specific environment-dependent deadline (**C1**) in order to prevent collisions or unnecessary emergency maneuvers that affect the comfort of the passengers [186]. However, these two requirements are often

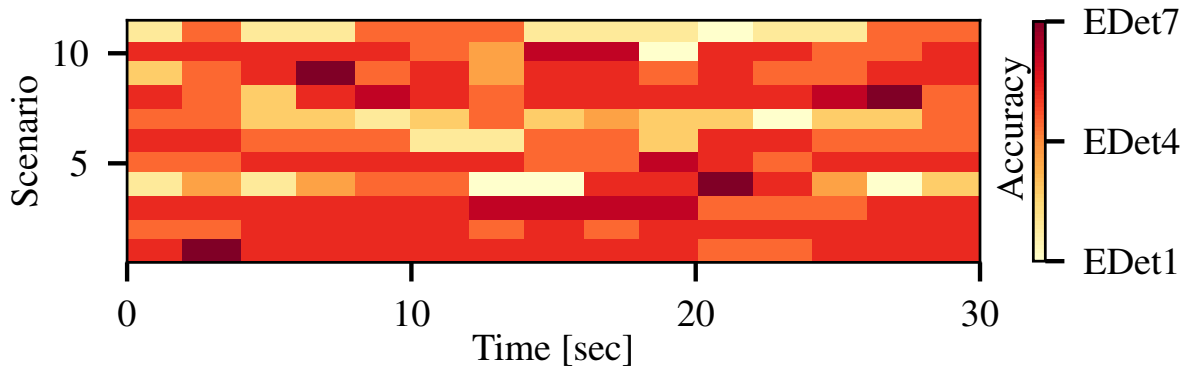


Figure 2.1: No silver bullet. We underscore the need for dynamically-varying deadlines by showing that: the choice of the optimum object detector can vary widely within and across driving scenarios proving that AVs benefit from being able to *dynamically adapt* to the given environment. We divide a set of 12 real-world driving scenarios [32] into 2 second intervals, and plot the model with the best accuracy [182] from the EDet family [289].

at odds since higher-accuracy components typically incur an increased response time, and the optimization of this runtime-accuracy tradeoff is further complicated by **C1-C2**. In the remainder of the section, we analyze these two unique characteristics using data collected from both our own real AV and the sensor data released by state-of-the art vendors.

2.1.1 C1: Environment-Dependent Deadlines

Ensuring safety across the wide-range of complex scenarios encountered in general driving requires an AV to dynamically change its response time to meet the varying deadlines demanded by the environment. To demonstrate this, we divide 12 driving scenarios from a real-world dataset [32] into 2 second intervals, and plot the object detection model with the highest accuracy (adjusted by its runtime [182]) from the EfficientDet family [289], which provide multiple points in the runtime-accuracy tradeoff curve. Fig. 2.1 shows that models with differing runtimes and accuracies perform better at different times, which renders the selection of a static point on the tradeoff curve during development inadequate.

To further support this, we develop a scenario using our real vehicle where a replica of a pedestrian walks out in front of the AV, and requires the AV to brake upon its detection[1]. In order to check if the AV can safely stop in time, we measure the *stopping sight distance* [277], which is the sum of the distance traveled by the AV during the detector's response time and the distance required to come to a halt (i.e., braking distance). To explore the tradeoff, we choose detectors EDet6 and EDet2 from the EfficientDet family where EDet6 is accurate at the expense of a higher response time, and EDet2 is faster but less accurate. Hence, while EDet6 can detect the pedestrian 72m away, EDet2 can only do so at a distance of 40m.

[1]A simulation of this scenario can be found at https://tinyurl.com/j4mhezze

Driving Speed (m/s)	Mean stopping sight distance (m)			
	Human	EDet2	EDet4	EDet6
7	24.65	7.66	8.71	11.14
12	51.02	21.89	23.69	27.87
17	84.69	43.43	45.98	51.89

Table 2.1: Stopping-sight distances with different EDet models and driving speeds. Time-sensitive scenarios at higher driving speeds make the fast low-accuracy models safer than the slow high-accuracy models, which perform better at slower driving speeds.

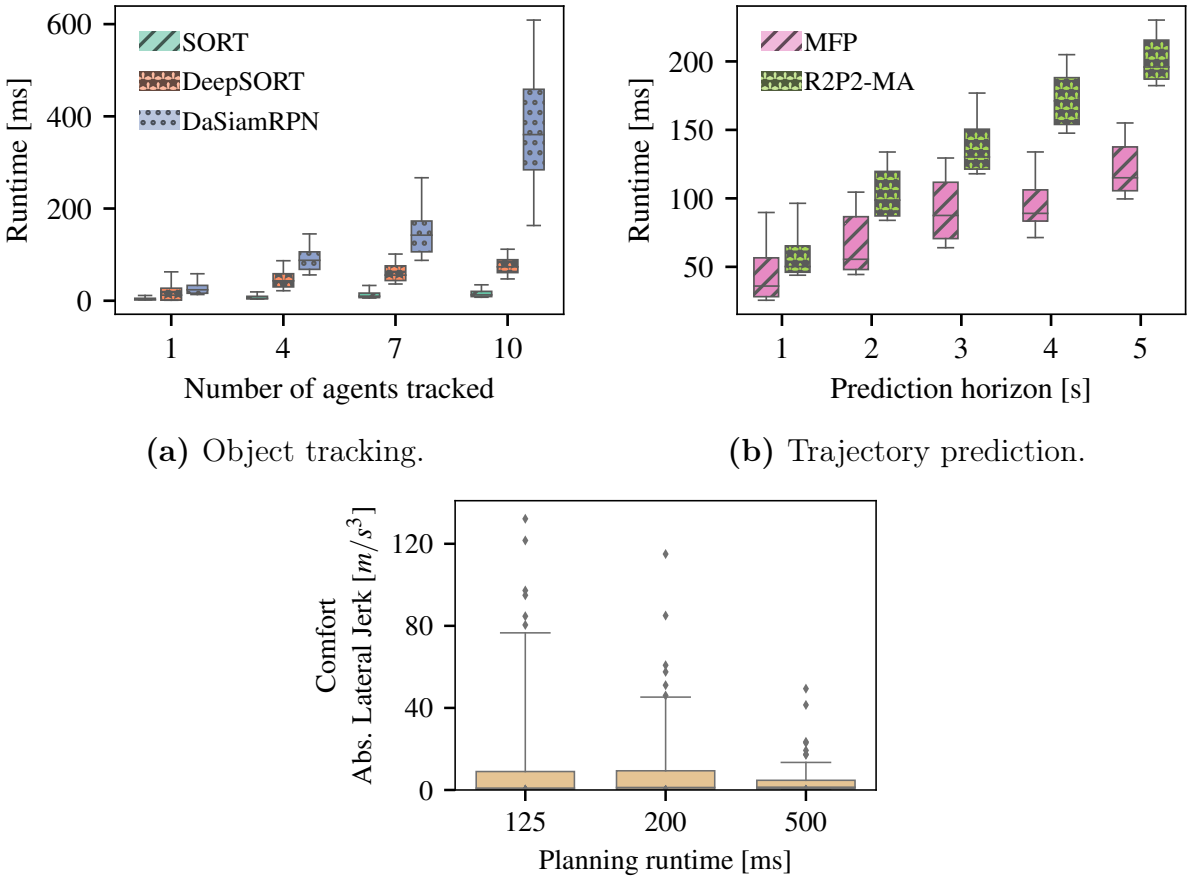
As a result, the AV must ensure safety by dynamically choosing between the two detectors based on its speed and the distance to the pedestrian (see Table 2.1). Specifically, an AV driving at 7m/s requires 7.66m to stop with EDet2 and 11.14m with EDet6, and hence must use EDet2 if the pedestrian walks out 12m away from the AV to be able to stop in time. Conversely, an AV driving at 17m/s requires 43.43m to stop with EDet2, while it can detect the pedestrian at a distance of 40m, which requires the AV to use EDet6 to stop safely.

2.1.2 C2: Environment-Dependent Component Runtime

Meeting constantly-evolving deadlines imposed by **C1** is complicated by the impact of the environment on the runtimes of AV components. For example, the number of agents (i.e. vehicles or pedestrians) in the scene affects the runtime of the perception module. Quantifying this impact, Fig. 2.2a plots how increasing the number of agents changes the runtimes of several object trackers, which are critical components of the perception module that track the trajectories of detected objects. To obtain these results, we drive an AV in the CARLA simulator [96] while increasing the number of agents, and observe an increase in the median runtime for all object trackers. Note that while SORT [48] provides a lower runtime, both DeepSORT [326] and DaSiamRPN [346] offer high accuracy.

In addition, the runtime of the prediction module depends on the AV's driving speed. An AV driving at a greater speed requires a higher prediction horizon i.e., it must forecast the trajectories of agents for longer into the future in order to ensure safety of the vehicle. Many prediction approaches (e.g., MFP [291], R2P2-MA [256]) use recurrent neural networks, which have a linear runtime dependence on the prediction horizon as shown in Fig. 2.2b.

The compounding of the runtime variability of individual components leads to a large skew between the mean and the maximum response time of the AV pipeline, which renders worst-case execution time analysis inefficient [17, 288]. To demonstrate this, Fig. 2.3 analyzes sensor data from Baidu's Apollo AV [42] that drove over 108,000 miles [315, 281]. Specifically, we focus on the traffic light detector [29], a key part of the perception module, that relies on the map and the vehicle's location to choose between multiple cameras in order to obtain bounding box proposals, which are individually refined and classified by multiple neural networks. We find that the response time of the traffic light detector depends on both the choice of the camera and the number of lights in the environment. As a result, the p99

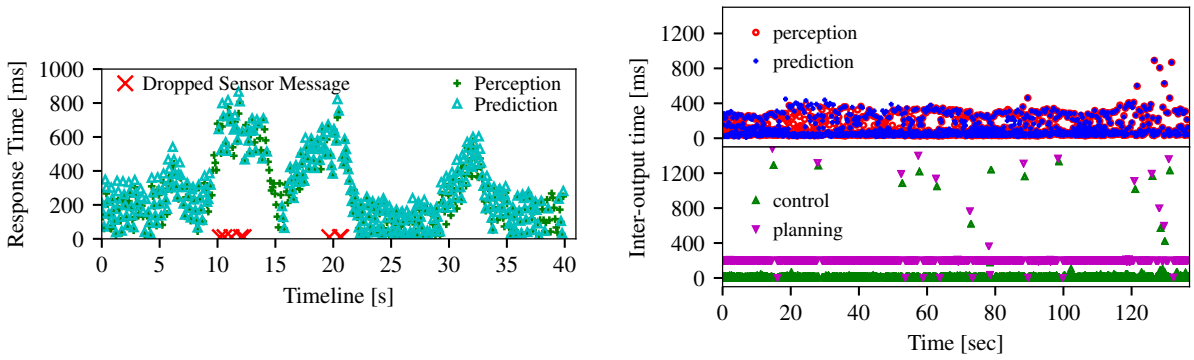


(a) Object tracking.

(b) Trajectory prediction.

(c) Trajectory planning.

Figure 2.2: Environment-Dependent Component Runtimes. Components benefit from an increased allocation of time as the complexity of the environment increases, which leads to better tracking accuracy, higher prediction horizons and more comfortable rides.



(a) Response time of components. **(b)** Inter-output time of components.

Figure 2.3: Response time variability. Baidu's Apollo production-grade components suffer from response time variability leading to delays and dropped sensor messages.

response time latency of perception is 3.3× higher than the mean, which further increases the response time of the downstream prediction component. Moreover, an increase in the response time keeps resources busy, thus forcing the pipeline to drop sensor messages.

2.2 D3: *D*ynamic *D*eadline-*D*riven Execution

The execution of applications that interact with a continuously-evolving environment (e.g. robots, AVs) requires a careful orchestration of their components. To enable such applications to optimize their accuracy under dynamically-varying deadlines (**C1**), we propose D3, an execution model that centralizes the management of deadlines. D3 models deadlines missed due to runtime variability (**C2**) as exceptions, and enables components to reactively adjust their computation.

To ease development, D3 structures its application as a directed operator graph along with a *deadline policy* π_{DP} (see Fig. 2.4). π_{DP} receives the environment's state (e.g., distance to obstacles) and computes an end-to-end deadline $\mathcal{D}$ that ensures safety and prevents unnecessary emergency maneuvers i.e., $\mathcal{D}$ bounds the wall-clock time that can elapse between an input to the graph and its corresponding output (Step 1). Further, π_{DP} splits $\mathcal{D}$ across operators and assigns a per-operator deadline $\mathcal{D}_i$ which aims to maximize the runtime-accuracy tradeoff based on the accuracy and pre-computed runtimes (Step 2).

In order to ensure *accurate* results, each operator requires its inputs from its multiple sources to be *synchronized* i.e., there must be a bounded skew between its earliest and its last arriving input for each invocation of the computation. By handling the synchronization of inputs automatically, D3 aids the development of AV pipelines in a decentralized manner. Each operator (of a component) can be easily replaced by a new version (with a potentially different runtime profile) without cascading changes throughout the pipeline that require downstream operators to synchronize on its input correctly.

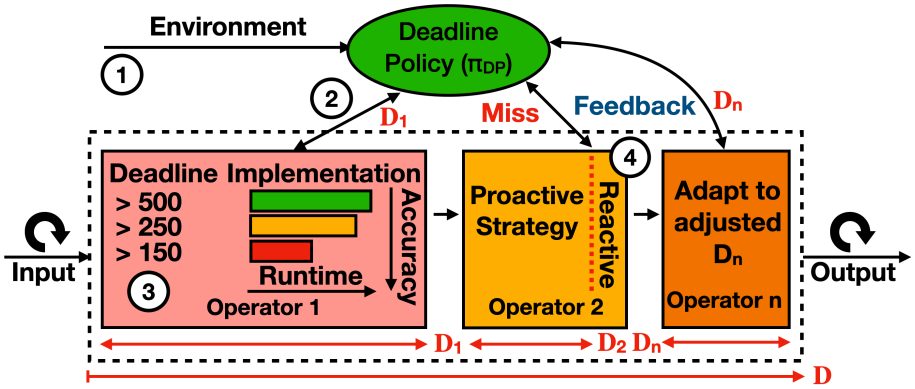


Figure 2.4: D3 Model structures an application as an operator graph with a policy π_{DP} that decides the deadline $\mathcal{D}$ as per the environment (1), and assigns a $\mathcal{D}_i$ to each operator (2). The operators *proactively* try to meet $\mathcal{D}_i$ (3). However, if $\mathcal{D}_i$ is missed, D3 executes reactive measures (4), and adjusts downstream $\mathcal{D}_i$s using a *feedback* loop.

Once an operator receives both its *synchronized* input and its allocated deadline $\mathcal{D}_i$, D3 expects operators to meet their allocated deadline $\mathcal{D}_i$ by using *proactive strategies* (e.g., running faster models under reduced deadline allocations; Step 3 of Operator 1 in Fig. 2.4). D3 models $\mathcal{D}_i$s missed due to runtime variability (**C2**) as exceptions and notifies operators to undertake reactive measures to quickly release output (e.g., quickly amending and releasing previous results; Step 4). D3 also notifies the downstream operators of the missed upstream deadline, allowing them to either: (*i*) eagerly execute with incomplete inputs due to a lack of output from the upstream operator that missed its $\mathcal{D}_i$, or (*ii*) reason about the reduction in their available time once the upstream operator's reactive measures release output and modify their computation accordingly to meet the updated deadline.

Crucially, D3 models missed deadlines as *exceptions*. D3 conveys the occurrence of the missed deadline events to π_{DP} using a *feedback loop*. Upon notification, π_{DP} may adjust the deadline for both downstream operators and future executions of the application. In extreme cases where the application is unable to perform its intended function due to multiple missed deadlines, π_{DP} can choose to execute a safety backup mode that performs simple maneuvers (e.g., braking or pulling over) to ensure a minimal risk condition [278].

2.2.1 Comparison with Related Execution Models

We now highlight D3's ability to maximize accuracy under dynamic deadlines by comparing it to two key works: data-driven execution models and periodic execution models.

Data-driven execution models employed by Service Level Objective-based (SLO) robotics platforms (e.g., ROS [248, 66], Cyber RT [41]) trigger computation on the arrival of input data and hence preclude the initiation of downstream computation in the absence of inputs due to a missed upstream deadline (**C1**). Moreover, such platforms execute computation on a best-effort basis and lack mechanisms to reason about changes in deadlines or variability

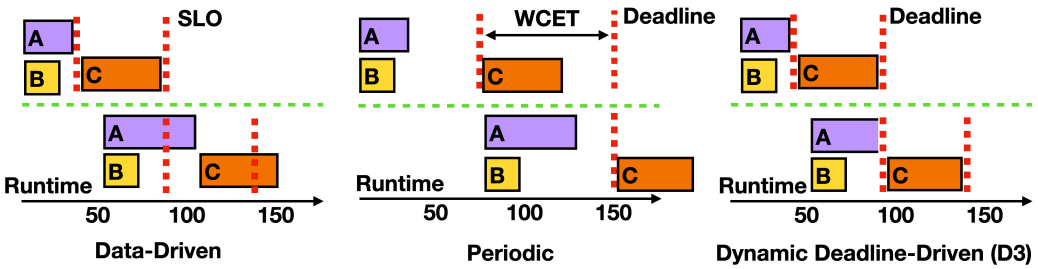


Figure 2.5: Timeline of execution models when C executes upon receipt of input from A and B. Data-driven models do not enforce deadlines and delay C's execution until both inputs are available. Periodic models use WCET to execute components at a fixed interval that is unable to adjust to slacks or delays, and fails to maximize the runtime-accuracy tradeoff. D3 achieves this by enabling components to either adjust to a constrained deadline or wait for delayed inputs.

in runtime [312]. As a result, components are unable to adjust their execution to varying deadlines leading them to miss their $\mathcal{D}_i$ in the presence of runtime variability (**C2**). For example, the lack of mechanisms to reason about a changed deadline coupled with the runtime variability in the second execution of A in Fig. 2.5 leads to a missed deadline under a data-driven execution model. Further, since downstream components can only trigger computation upon receiving of the input from upstream components, the effects cascade and the second execution of C misses its deadline. Conversely, D3's π_{DP} notifies A of the change in deadline allowing it to proactively modify its computation to meet the new deadline, or reactively release output quickly in case it is missed. D3 also enables C to execute its computation without the input from A, or wait until the input is available and modify its computation to fit within the reduced time.

Periodic execution models, which underpin hard real-time systems [191, 123], use conservative worst-case execution time (WCET) estimates to execute computation at a fixed, periodic interval. While this approach precludes a lack of input from upstream components or a reduction in computation time due to **C2**, it fails to maximize the runtime-accuracy tradeoff due to the large skew between the mean and maximum runtime of computation in an AV pipeline (see §2.1.2). For example, the WCET-driven periodic execution of C in Fig. 2.5 leaves plenty of slack in the average case, which could be used to execute a higher-accuracy computation. Further, the inflexibility of these models has led to applications adjusting their computation to meet environment-dependent deadlines (**C1**) by defining a fixed set of *mode changes*. However, executing the various modes requires the components to either transition to SLO-based execution [51, 176, 58], or undergo a time-consuming schedulability analysis for each possible deadline and mode transition [54, 46, 47, 250]. By contrast, D3's proactive strategies and reactive measures enable the computation to forego this expensive analysis, and still adjust itself to meet dynamic deadlines (**C1**). We emphasize that D3's execution model subsumes such coarse-grained mode changes by allowing the deadline policy π_{DP} to perform mode changes on either deadline misses or specific environment conditions (e.g., change in the vehicle speed) (see Chapter 3).

Chapter 3

ERDOS: Elastic Robot Dataflow Operating System

We now elaborate on our design and implementation of ERDOS, a proof-of-concept realization of D3 built specifically for AV pipelines. ERDOS exposes fine-grained execution events to the application and provides abstractions for the specification of dynamically-varying deadlines that restrict the wall-clock time elapsed between such events. ERDOS' speculative execution mechanism then aims to fulfill a deadline by executing the appropriate implementation automatically (see Chapter 3). However, if deadlines are missed due to **C2**, ERDOS executes *exception handlers* that allow computation to convey intermediate results to enable the execution of downstream computation. This chapter is organized as follows:

- We introduce the key concepts from prior streaming systems [121, 62, 205, 337] that are critical in understanding how ERDOS' novel extensions enable D3 (§3.1).
- We present the techniques that enable ERDOS to support D3 (§3.3-§3.4) and exemplify ERDOS' key concepts through the implementation of a `Planner` component (§3.3).
- We elaborate on the open-source implementation of ERDOS, a deadline-driven system for AVs (§3.6). The artifacts are available at `https://github.com/erdos-project/`.
- We address the crucial lack of AV benchmarks by providing the first open-source state-of-the-art AV, Pylot (§3.7.1) (see Chapter 4 for a detailed discussion). Pylot works across simulators and real-vehicles, and achieved the top score in an AV challenge.
- We evaluate the efficacy of the dynamic deadline-driven execution enabled by D3 and ERDOS by driving Pylot across 50 km of challenging driving scenarios in simulation (§3.7), and observe a 68% reduction in collisions as compared to the execution model of state-of-the-art robotics platforms.

3.1 Primer on Streaming Systems

Streaming systems (e.g. Cloud Dataflow [121], Flink [62]) structure applications as a directed graph of computational operators, which contains a set of source operators that generate the input and a set of sink operators that consume the output. The sources annotate the inputs with a timestamp derived from an ordered time domain, and notify their downstream operators when they have finished sending all the input for a given timestamp. These messages and notifications cascade along the directed edges of the graph, with each operator potentially transforming its input messages M_t timestamped with t and received along an edge e before sending them along e'.

Further, the operators are notified of the receipt of all messages with time $t' \leq t$ using a watermark message W_t. A watermark [280, 308, 11] informs the operators of the availability of all the inputs required for a computation across all its edges, and thus ensures accurate computation upon synchronized data [13]. While the computation registered with a watermark notification is executed sequentially according to the the timestamp order, the computation that acts on messages is allowed to execute out-of-order, which allows the operators to prevent stragglers while ensuring correctness [13].

3.2 Computation Structure of an ERDOS Application

ERDOS instantiates D3 by modeling the application as a directed graph composed of multiple subgraphs representing the modules (e.g., perception), with each module containing operators representing the components (e.g., lane detection) connected by typed streams. A source of the graph reads data from a sensor and uses an output `WriteStream` to inject it into the graph, while a sink extracts data from the graph using an input `ReadStream`, and sends commands to the vehicle.

Each operator must implement an interface that specifies both the number and types of its input and output streams. This static registration of the input and output allows the system to ensure that the computation graph is well-formed at compile-time, and reduces the runtime errors. Moreover, the static registration enables the system to optimize the allocation of operators to hardware (e.g., colocate operators).

The typed streams allow communication through timestamped messages i.e. a stream s of type $\mathcal{T}$ can carry: (i) a `DataMessage` (M_t), with a payload of type $\mathcal{T}$ and a timestamp t, and (ii) a `WatermarkMessage` (W_t), with a timestamp t that represents the completion of all incoming messages for $t' \leq t$. Corresponding to the type of message received and the input stream it is received upon, the interface implemented by each operator defines the callbacks that are invoked by ERDOS (see §3.3).

The timestamp t generated by the source operators consists of

$$t = \left(l \in \mathbb{N}, \hat{c} : \langle c_1, \ldots, c_k \rangle \in \mathbb{N}^k\right)$$

where l represents a logical time (see §3.4.1), and $\hat{c}$ conveys application-specific information (elaborated in §3.4.3). This abstraction enables applications to seamlessly work across both

real-world and simulation, by using l to represent the wall-clock time in real AVs, and simulation time when using a simulator, the latter of which may advance at a different rate than real-time. While a simulator provides a consistent notion of time, ERDOS exploits the presence of a local high-speed network in real AVs to precisely synchronize clocks in order to correctly reason about the wall-clock time across multiple machines [128, 114].

3.3 ERDOS' API

We now provide an overview of the API with the help of a simplified `Planner` (see Lst. 3.1) that receives the `Obstacles` and `TrafficLights` from perception through `DataMessage`s. It then computes a motion plan and returns a set of `Waypoints` i.e., fine-grained points on the road that characterize the trajectory of the AV. To register its input and output, the `Planner` implements the `TwoInOneOut` interface where the `ReadStream`s are typed by `Obstacles` and `TrafficLights`, and the `WriteStream` by `Waypoints`.

Further, to invoke the computation, the `Planner` implements `on_msg` (lines 13-17) that convert the coordinate system of each `Obstacle` and `TrafficLight`, a task that can be executed out-of-order for each timestamp. However, in order to compute a safe plan, the `Planner` requires a synchronized and complete set of all the obstacles and traffic lights from perception, and hence, waits for a `WatermarkMessage` from both the upstream operators signifying the receipt of all incoming messages for each timestamp. It then uses the converted obstacles and lights to compute the `Waypoints` for the AV in `on_watermark` (line 18).

Moreover, the `Planner` must complete its computation within a deadline, and thus restricts its runtime from the time of the receipt of the input to a dynamically-varying deadline retrieved from the `deadline_stream` provided by the deadline policy π_{DP} (line 11). §3.4.1 and §3.4.2 further elaborate on the specification and dynamic-variation of deadlines. ERDOS automatically exposes the deadline for the timestamp computed by π_{DP} to each of the callbacks via the `Context`, allowing the operators to employ proactive strategies to meet deadlines and vary their computation accordingly (see §3.4.3).

However, to meet its deadline in the presence of a delay of more than 30ms in the receipt of the `TrafficLights`, the `Planner` chooses to eagerly initiate the computation with partial input, and computes the plan using just the obstacles (line 6). Finally, the deadline specification also requires an exception handler that invokes reactive measures to quickly releases output upon a missed deadline (`on_deadline`). The handler receives a `Context` containing information useful to mitigate the deadline miss (e.g., timestamp, deadline) along with the state of the operator, and can be used to output the previous computed plan offset from the AV's current location.

3.4 Achieving Dynamic End-to-End Deadlines

We now discuss ERDOS' core contributions that enable it to address the following challenges posed by realizing D3 in an efficient system:

```
impl TwoInOneOut<Obstacles, TrafficLights, State<PlanningState>, Waypoints>:

  fn setup(objects: ReadStream<Obstacles>,
           lights: ReadStream<TrafficLights>,
           plan: WriteStream<Waypoints>,
           deadlines: ReadStream<Deadline>) {
    // Call ‘on_watermark‘ even in the absence of traffic lights.
    FrequencyDeadline::new(PlanningOp::on_watermark)
        .with_static_deadline(30).on_stream(lights);

    // Constrains the completion of local computation.
    TimestampDeadline::new(PlanningOp::on_deadline)
        .with_end_condition( // and a default start condition
          |sent_msg_cnt: usize, watermark_status: bool| sent_msg_cnt > 0)
        .with_dynamic_deadline(deadlines).on_stream(plan);
  }

  fn on_left_msg(ctx: Context,
                 objects: Message<Obstacles>,
                state: State<PlanningState>) {
    // Change coordinate system of objects and add to state.
  }

  fn on_right_msg(..., lights: Message<TrafficLights>, ...) {...}

  fn on_watermark(ctx: Context,
                  state: State<PlanningState>,
                  plan: WriteStream<Waypoints>) {
    // Computes a plan upon receiving obstacles and traffic lights.
  }

  fn on_deadline(ctx: Context,
                 state: State<PlanningState>,
                 plan: WriteStream<Waypoints) {
    // Invoked when a deadline is missed.
  }
```

Listing 3.1: A simplified `Planner` that computes a trajectory for the AV using the `Obstacles` and `TrafficLights`, and specifies deadlines on its execution and response time.

- ERDOS must initiate computation upon the availability of all required input, and allow components to bound their response time from that event. In addition, components must be allowed to initiate computation in the presence of partial input if upstream components miss their deadline $\mathcal{D}_i$ (§3.4.1).
- ERDOS must allow the deadline policy π_{DP}, which dynamically varies $\mathcal{D}_i$, to meet strict safety, adaptivity and modularity constraints, owing to its critical effects on the latency-sensitive computation (§3.4.2).
- ERDOS must provide efficient mechanisms to enable components to utilize different strategies to *proactively* output the highest-accuracy results possible within $\mathcal{D}_i$ (§3.4.3).
- In case of a missed $\mathcal{D}_i$, ERDOS must enable execution of *reactive measures* that quickly release output, and allow downstream computation to begin. (§3.4.4).

3.4.1 Deadline Specification

In an effort to meet both its individual deadline $\mathcal{D}_i$, and the end-to-end deadline $\mathcal{D}$, each component of a D3 application must be able to: (i) bound the execution time from the receipt of the inputs and the generation of the corresponding output (**C1**), and (ii) bound the time between the invocation of the computation on inputs of successive timestamps in the presence of runtime variability in upstream components (**C2**). While (i) ensures that a component adheres to its allocated deadline $\mathcal{D}_i$ for time t, (ii) allows components to eagerly initiate their computation on incomplete input for time $t' > t$ to ensure that the end-to-end deadline $\mathcal{D}$ is met for time t' in case the upstream components miss their $\mathcal{D}_i$ for t'.

To achieve these goals, ERDOS must track the initiation and completion of computation for every time t. ERDOS accomplishes this by automatically capturing fine-grained execution events from the components at the granularity of the logical time l. Specifically, for each logical time l of the timestamp t, ERDOS maintains counters of the number of incoming and outgoing messages annotated with t (M_t), and boolean variables indicating the receipt and generation of the watermark for t (W_t) across all the input streams for each component. This allows ERDOS to automatically initiate computation once all required inputs are available (denoted by the receipt of a W_t across all the input streams), and register the computation's completion for the logical time l once the watermark W_t for t is sent on the output streams.

In order to enable flexibility in the events that are constrained by a deadline, ERDOS exposes these events to the components, and allows them to specify relative deadlines, which limit the amount of wall-clock time that can elapse between any two events. Specifically, components can register two boolean functions: *deadline start condition* (D_{SC}) and *deadline end condition* (D_{EC}), which return `True` to signify the initiation and completion of the computation for a logical time l respectively. D_{SC} and D_{EC} are evaluated at the receipt and generation of every message, and receive a tuple ($n \in \mathbb{N}, w \in \{\texttt{True}, \texttt{False}\}$), where n denotes the number of messages received or sent for that logical time, and w depicts the receipt or generation of the watermark. ERDOS maps the relative deadline $\mathcal{D}_i$ to an absolute deadline

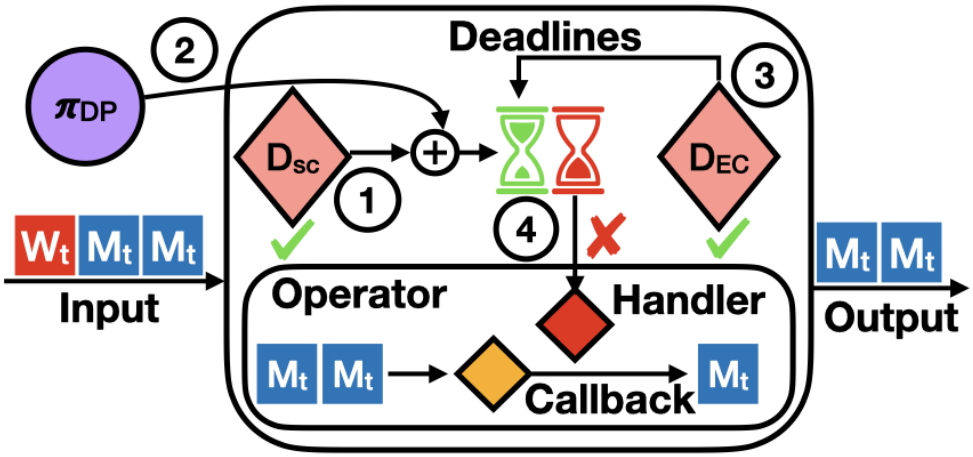


Figure 3.1: Environment-dependent deadlines. ERDOS evaluates D_{SC} for every message (1). If satisfied, it initiates an absolute deadline according to π_{DP} (2). Similarly, ERDOS evaluates D_{EC} upon generation of messages, and removes any satisfied deadlines (3). If a deadline is missed, ERDOS invokes an exception handler (4).

by automatically capturing the wall-clock time at which D_{SC} is satisfied (❶ in Fig. 3.1), and offsetting it by $\mathcal{D}_i$ (❷ in Fig. 3.1). ERDOS then tracks the passage of wall-clock time and ensures that D_{EC} is satisfied before the absolute deadline (❸ in Fig. 3.1).

Further, to simplify the specification of the relative deadline $\mathcal{D}_i$ for the enforcement of the response time deadlines (i) and (ii), ERDOS provides the following two general deadline abstractions that constrain a default set of events:

Timestamp deadlines (lines 7-11 in Lst. 3.1) bound the execution time. Components define a relative deadline $\mathcal{D}_i$ that constrains the wall-clock time elapsed between a default D_{SC} that specifies the receipt of the first message timestamped with t (M_t), and a default D_{EC} that specifies the output of the first watermark timestamped with $t' \geq t$ $(W_{t'})$. If D_{EC} is not satisfied before $\mathcal{D}_i$ expires, ERDOS invokes the exception handler (`on_deadline` on line 22 in Lst. 3.1), which releases output to initiate downstream computation (see §3.4.4).

Frequency deadlines (lines 4-6 in Lst. 3.1) allow a precise invocation of the computation in the presence of runtime variability. To achieve this, components define a relative deadline $\mathcal{D}_i$ that constrains the maximum wall-clock time that may elapse between a default D_{SC} that specifies the receipt of the watermark timestamped with t (W_t), and a default D_{EC} that specifies the receipt of the first watermark for $t' > t$ $(W_{t'})$. If D_{EC} is not satisfied for t' before $\mathcal{D}_i$ expires, ERDOS automatically inserts $W_{t'}$ on the given input stream to simulate the arrival of all incoming data for t', and invokes the computation with the partial input (see §3.4.3). For example, if $W_{t'}$ does not arrive on the `lights` stream within 30ms of the receipt of W_t (as specified on line 6 in Lst. 3.1), ERDOS automatically inserts $W_{t'}$ and invokes `on_watermark` with partial input.

We emphasize that the ability to tailor the above general abstractions using the fine-grained execution events exposed by ERDOS, enables components to specify the full spectrum of deadline constraints discussed in prior work [84]. To exemplify this ability, the `Planner` in Lst. 3.1 uses this control to tailor the `TimestampDeadline` constraint with a custom D_{EC} (lines 9-10 in Lst. 3.1) that is satisfied as soon as the first message for a timestamp t is output. Coupled with the default D_{SC}, this constraint allows the `Planner` to bound the time between the receipt and generation of the first message timestamped with t (M_t). This deadline can be used by the `Planner` to quickly release a coarse-grained plan before refining it, thus enabling downstream computation to begin.

3.4.2 Environment-Dependent Deadlines

Components may use the abstractions discussed in §3.4.1 to specify static deadlines that do not evolve over time by using static values for $\mathcal{D}_i$ (e.g., 30 ms for the `FrequencyDeadline` on line 6 in Lst. 3.1). However, D3 requires that these abstractions support dynamic deadlines determined by a deadline policy π_{DP}, which evolve according to the environment (**C1**).

We emphasize that the centralization of π_{DP}, which dynamically determines the end-to-end deadline $\mathcal{D}$ and the individual deadline $\mathcal{D}_i$ for each component, is a novel contribution of the D3 execution model. While the development of such a policy raises interesting research challenges orthogonal to this work (we explore such a policy in deeper detail in [270]), this section concerns itself with the following key systems challenges that its placement on the critical path of the computation presents to the design of ERDOS:

Safety. The presence of π_{DP} on the critical path of affecting what computation runs in each component requires it to meet strict deadline constraints. In addition to being able to initiate a safety backup mode that performs simple maneuvers if multiple component deadlines are missed (see §2.2), π_{DP} must also ensure safety by executing the backup mode if it misses its own deadline (due to delayed inputs or runtime variability).

Adaptivity. To reduce π_{DP}'s effect on the latency of the critical path, ERDOS must allow applications to adapt the frequency at which the deadline allocations are recomputed according to the *dynamicity* of the environment. For example, a π_{DP} may change the allocations less frequently on highways than in cities, owing to the infrequent change in environment.

Modularity. Individual modules (e.g., perception) may exploit expert knowledge to specify policies that split deadlines across their components (e.g., detection, tracking) more efficiently than a centralized policy. Thus, ERDOS must enable the decomposition of monolithic policies such that high-level policies provide coarser-grained deadlines to module-specific policies, which further split them across their components.

To achieve these goals, ERDOS executes π_{DP} as a subgraph of operators which receive information about the environment from components on its input streams. π_{DP} processes this information to compute an end-to-end deadline $\mathcal{D}$ and decomposes into individual deadlines $\mathcal{D}_i$, which are sent to components via its output streams. Specifically, lines 8-11 in Lst. 3.1

show how an operator can adjust its `TimestampDeadline` according to π_{DP} by registering on the `deadlines` stream provided by ERDOS. π_{DP} utilizes the state of the environment to dynamically compute the relative deadline $\mathcal{D}_i$ for each logical time l, and communicates it to the operator using the `deadlines` stream. ERDOS automatically synchronizes the computation for l with the corresponding $\mathcal{D}_i$ provided by π_{DP}, and utilizes it to compute the absolute deadlines for the computation (see Fig. 3.1). To enable components to adjust their computation to meet the changing deadlines (§3.4.3), ERDOS exposes the absolute deadlines via the `Context` data structure (§3.3).

Executing π_{DP} as a subgraph enables the policy to exploit ERDOS' graph abstraction to achieve *modularity* by splitting itself across operators, and benefit from co-location with components that share the state of the environment with them (see §3.4.4). Moreover, π_{DP} can use ERDOS' timestamping mechanism to achieve *adaptivity*. Specifically, a π_{DP} can send a $\mathcal{D}_i$ in a message M_t followed by a watermark $W_{t'}$, where $t' \geq t$, and adaptively evolve the delta between t' and t according to the environment. $W_{t'}$ signifies the completion of all outputs from π_{DP} until timestamp t', and specifies the relative deadline $\mathcal{D}_i$ for the next timestamps from t to t'. Further, π_{DP} can ensure *safety* by using ERDOS' static deadlines (§3.4.1) to enforce strict constraints on its execution, and invoke the safety backup mode (available in various production AVs) in case it misses a deadline (see §3.4.4).

3.4.3 Meeting Deadlines

ERDOS exposes the deadline $\mathcal{D}_i$ (allocated by π_{DP}) to the operators via the `Context` (lines 13, 18 in Lst. 3.1). We now discuss some general proactive strategies that operators may use to meet $\mathcal{D}_i$ (by satisfying D_{EC} before it expires) below:

Executing anytime algorithms [169, 324, 329] that maximize the accuracy for a given $\mathcal{D}_i$ through iterative refinement [347], and provide a continuous runtime-accuracy tradeoff curve by monotonically increasing accuracy with increasing deadlines. Such algorithms can be interrupted when $\mathcal{D}_i$ expires and ensure the highest-accuracy results possible within the time. Moreover, components can choose to release lower-accuracy results (before $\mathcal{D}_i$ expires) to downstream operators, allowing them to begin computation early and iteratively refine their results. For example, the `Planner` in Lst. 3.1 could execute an anytime planning algorithm [169, 324, 329] in its `on_watermark` method. The algorithm would release coarse-grained waypoints and iteratively refine them, to allow the downstream control operator to begin generating commands for the underlying machinery and continuously updating them.

Changing the implementation based on the most accurate algorithm that typically completes within $\mathcal{D}_i$ (e.g., mean or p99 runtime is less than $\mathcal{D}_i$). This is facilitated by the existence of multiple algorithms for the components, that enable a tradeoff between runtime and accuracy [147, 289] (see Chapter 1). We show a latency-sensitive, resource-efficient technique to serve multiple ML models that enable this strategy in Chapter 5.

Executing multiple versions of components to ensure that at least one completes before $\mathcal{D}_i$ expires (similar to [284]). In addition to choosing the highest-accuracy algorithm that fits

within $\mathcal{D}_i$, components can execute faster algorithms that are guaranteed to finish execution before $\mathcal{D}_i$ expires, thus maximizing the runtime-accuracy tradeoff, while still meeting deadlines in the presence of runtime variability (**C1**). For example, a detector can run in parallel: (i) the most-accurate model that *typically* completes within $\mathcal{D}_i$, and (ii) a fast, low-accuracy model, and return results from (ii) if (i) does not meet $\mathcal{D}_i$. Similar to anytime algorithms, components can release the lower-accuracy results to unblock downstream operators, or wait until $\mathcal{D}_i$ expires, and return the highest accuracy results available.

Skipping the execution of an algorithm in case of small $\mathcal{D}_i$. Unlike load shedding [65, 294, 293] that does not generate results, AV components can quickly release reduced-accuracy results to unblock downstream computation by amending prior results. For example, the `Planner` in Lst. 3.1 can release its last computed plan offset to the AV's current location.

Eagerly executing with partial input if upstream operators cannot meet their $\mathcal{D}_i$ due to runtime variability (**C2**). While previous strategies require the input to be available and must adjust the computation to a reduced deadline in case of upstream runtime variability (**C2**), this strategy allows components to eschew input from certain upstream components in order to maintain its initially allocated $\mathcal{D}_i$. For example, the `Planner` in Lst. 3.1 eagerly executes without `TrafficLights`, and plans a trajectory using `Obstacles` if the upstream component experiences a runtime variability of more than 30ms.

To ease the use of these strategies, ERDOS allows the specification of multiple implementations of components along with their runtimes. ERDOS then chooses to either change the implementation, speculatively execute multiple versions or skip the execution based on $\mathcal{D}_i$, and provides two novel mechanisms to enable the efficient realization of these strategies:

Intermediate Results. ERDOS' extension of timestamps provides first-class support for anytime algorithms, speculative execution of multiple versions, and enables eager execution with partial input. Specifically, anytime algorithms and different versions can annotate outputs with $t = (l, \hat{c})$, and increase the value of $\hat{c}$ to notify downstream computation of the accuracy of the results as they become available (with an increased value of $\hat{c}$ signifying increased accuracy of the results). ERDOS orders the execution of computation using $\hat{c}$ and automatically prioritizes computation on higher-accuracy inputs, thus maximizing the accuracy of results. Similarly, upon expiration of a `FrequencyDeadline`, ERDOS automatically inserts a W_t (with a low value of $\hat{c}$) on the stream that failed to generate the required input within the deadline. The computation conveys the accuracy of these results downstream, and refines its results as missing inputs from upstream components become available.

For example, in the absence of `TrafficLights` or when using anytime algorithms, the `Planner` can output a coarse-grained plan and annotate its accuracy using $t_1 = (l, \hat{c}_1)$. This allows the downstream control operator to generate commands using the coarse-grained plan, and refine them after a fine-grained plan is available. If multiple plans are available, ERDOS automatically eschews the control operator's execution with a coarse-grain plan tagged with t_1 in favor of a fine-grained plan tagged with $t_2 = (l, \hat{c}_2)$, where $\hat{c}_2 > \hat{c}_1$.

Speculative Execution. ERDOS automatically chooses to change the implementation,

execute multiple versions or skip the execution based on $\mathcal{D}_i$. To achieve this, it requires components to decouple their state from the implementation of the computation, and specify multiple implementations along with their runtime profiles. Specifically, a component must register its state (e.g., `PlanningState` on line 1 in Lst. 3.1) with ERDOS. By assuming control of the state, ERDOS' speculative execution mechanism achieves an efficient execution of different implementations for successive timestamps by automatically providing access to the state to different callbacks (lines 14, 18, 22 in Lst. 3.1). Further, the mechanism enables the parallel execution of multiple versions by providing each implementation with a view of the state without requiring operators to synchronize updates to the state (see §3.4.4).

3.4.4 Handling Deadline Misses

If the strategies discussed in §3.4.3 fail to meet the allocated $\mathcal{D}_i$, D3 requires components to undertake reactive measures, whose execution presents the following challenges:

Fast Invocation of the measures upon expiration of $\mathcal{D}_i$ so as to quickly unblock downstream computation and minimize the reduction of available time for downstream operators.

Access to the state of the partially-executed proactive strategies to enable the measures to quickly release results, and ensure its correctness for executions of future timestamps.

Parallel execution of the measures and the proactive strategies to enable components to quickly unblock downstream computation with lower-accuracy results while using higher-accuracy computation for state updates. For example, if the `Planner` misses $\mathcal{D}_i$ while running a higher-accuracy algorithm, it can release the last computed plan offset to the current location of the AV as its reactive measure, while continuing to release the higher-accuracy plan and updating the state with the high-accuracy computation for future executions.

To address these challenges, ERDOS enables components to accompany the specification of timestamp deadlines (from §3.4.1) with *deadline exception handlers* (D_{EH}) (lines 22-25), which execute the reactive measures if $\mathcal{D}_i$ is not met. ERDOS orchestrates the execution of the specified D_{EH} alongside the proactive strategies through the following execution policies:

Abort. Terminates the execution of the proactive strategy for time t, and requires D_{EH} to notify the computation's completion by sending a watermark W_t and ensure the correctness of state for t. To achieve this, D_{EH} receives a view of the state for $t' < t$ along with the *dirty state* for t (i.e., mutations made to the state by the partially-executed proactive strategy for t). D_{EH} uses these views to quickly release output and amend the dirty state to ensure its correctness. For example, a D_{EH} in a `Planner` could either use the dirty state at t to output and save the best plan found by the deadline if the operator is anytime, or amend the plan computed for t' otherwise.

Continue. Executes the proactive strategy for t in parallel with the D_{EH}. The latter unblocks downstream computation by releasing output for t (M_t), while the former notifies the computation's completion (W_t) and ensures state consistency. To achieve this, D_{EH}

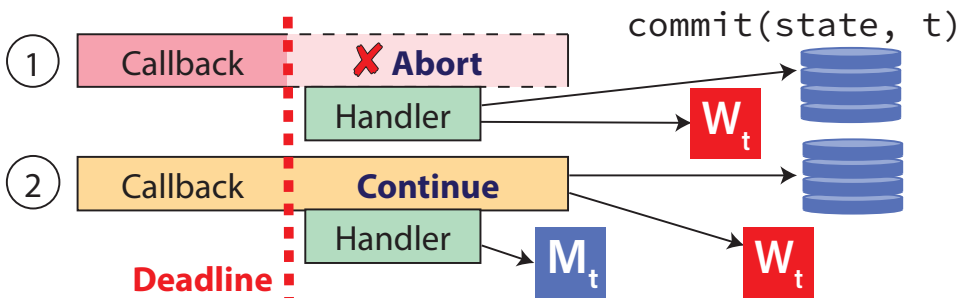


Figure 3.2: Handling missed deadlines. When a deadline is missed, handlers are invoked to mitigate the consequences. Callbacks which miss their deadline may **Abort** to let the handler rapidly update operator state, or **Continue** to ensure more accurate state updates.

receives a view of the state for $t' < t$, and executes a fast algorithm to quickly release output. In parallel, the proactive strategy continues releasing output for t, thus providing the downstream computation a choice of more accurate results. Moreover, allowing the proactive strategy to save the state for t enables the computation for $t'' > t$ to use the high-accuracy results, and prevents a cascade of low-accuracy results across time. For example, a D_{EH} in a `Planner` can amend the plan computed for t', while the proactive strategy releases and saves a more-accurate plan.

A seamless execution of D_{EH} under the *Abort* and *Continue* policies requires a careful management of the component's state in order to ensure its consistency. To aid the components in this endeavor, ERDOS provides **system-managed state.** Specifically, ERDOS assumes control over the state of the components decoupled from their implementation (see *Speculative Execution* in §3.4.3), and enables the state to meet the challenges of executing D_{EH} discussed earlier by ensuring the following two key properties over it:

Transactional Semantics. In order to ensure a fast invocation and parallel execution of π_{DP}, D_{EH} and multiple versions of the computation, ERDOS must provide them with a view of the state and ensure that it is saved from either the D_{EH} or the proactive strategy according to the execution policy. ERDOS achieves this by enforcing transactional semantics on the state at the granularity of a timestamp, and provides the proactive strategy with a view of the last committed state, and automatically commits any mutations made by them upon the successful release of the watermark for the currently executing timestamp. In case of a missed $\mathcal{D}_i$, ERDOS invokes D_{EH} and shares the dirty state along with a view of the last committed state, and automatically commits the changes made to the dirty state by D_{EH} upon its completion. These semantics cleanly enforce the consistency of the state when using the *Abort* policy by discarding any mutations made to the state by the proactive strategy.

Time-Versioning. To further ensure the execution of π_{DP}, D_{EH} and proactive strategies across multiple timestamps, ERDOS maintains a version of the state for each timestamp t. For example, multiple executions of the `Planner` for different timestamps (each corresponding to a different set of `Obstacles` and `TrafficLights`) can be executed in parallel with their

computed plans being saved in different versions. In case a deadline is missed for t, the D_{EH} gets access to the *committed* state for all timestamps $t' < t$ and can send M_t to unblock downstream computation. Meanwhile, the proactive strategies can continue in parallel for timestamps $t'' \geq t$ and commit state mutations by releasing $W_{t''}$.

Moreover, while ERDOS provides a default `State` implementation with the properties discussed above, it allows components to provide their own states. These states implement an interface that customizes both transactional semantics (through `commit`) and time-versioning (through `get_committed` to retrieve a view of the state at t), and may use techniques such as CRDTs [274]. For example, the `Planner` could implement the interface for `PlanningState`, instead of using `State` (as shown in line 1 of Lst. 3.1). In such a case, the `PlanningState` could maintain a vector of waypoints for timestamp $t = 0$, and log additions of future waypoints in `commit`, instead of saving the entire set of waypoints for each timestamp $t' > t$.

3.5 Achieving Determinism

To support rapid innovation, it is paramount to be able to deterministically reproduce failures in order to debug and compare components in identical scenarios [266]. While a dataflow system orders the messages arriving on a stream, the behavior of the system (e.g., increased network load and different application and system scheduling decisions) can introduce variability in the arrival of messages across multiple streams. This becomes problematic when a single operator shares `State` across those streams, since non-commutative updates to the state may result in differences depending on message arrivals. ERDOS automatically ensures a deterministic execution by enforcing that all modification to the shared `State` occur exclusively within a strict execution lattice that is enforced by the watermark callbacks.

Processing on watermarks. To avoid non-determinism due to reordering of messages, ERDOS does not allow message callbacks to either read from or write to the state for any timestamp. The message callbacks can only append results to an unordered set for the current timestamp, which is utilized by the watermark callback to compute the final results from any partial computations by the message callbacks. Upon completion of the watermark callback, the final result is timestamped and committed to the time-versioned state. Subsequent accesses to committed state are read-only. By deterministically processing unordered sets of messages on watermark callbacks, we transform the dataflow into a Kahn process network [116], which guarantees determinism.

Callback execution lattice. To ensure that updates to the state remain deterministic, ERDOS defines a partial order over the different callbacks of an operator. Since the message callbacks are only allowed commutative appends to an unordered set, ERDOS allows message callbacks to be run in parallel. However, a watermark callback for timestamp t on a stream is only allowed to execute after all watermark callbacks for timestamp $t' < t$ and all message callbacks for the time t on that stream have finished executing.

We note that while ERDOS's default partial order provides deterministic execution of

events, its design makes it easy to relax these constraints and allows advanced developers to choose the required point in the tradeoff between determinism and performance. For example, developers can choose to run watermark callbacks for multiple timestamps in parallel by defining them to be equivalent in their partial order.

Reproducible deadlines. A deadline miss and the subsequent handler invocation introduces non-determinism because the handler invocation depends on physical time, as opposed to the logical time of the dataflow. To ensure deterministic replay, ERDOS logs the timestamp of the message for which the handler was invoked, the physical time at which the handler was invoked and restricts access to only the committed state inside the handler.

3.6 ERDOS' Implementation

ERDOS is an open-source distributed system implemented in $\sim$ 13k lines of Rust, whose type safety and memory semantics are essential for safety-critical applications. Further, to interact with ML frameworks [3] and enable prototyping with simulators [96], ERDOS provides a Python interface. We now elaborate on its design by discussing how operators execute (§3.6.2), how they communicate (§3.6.1), and how they enforce deadlines (§3.6.3).

ERDOS' distributed nature is enabled by a leader-worker architecture where the leader manages a set of worker processes running across several machines. The leader partitions the operator graph and schedules operators to workers, which are responsible for exchanging data along streams (§3.6.1), executing callbacks (§3.6.2), managing deadlines and executing their exception handlers (§3.6.3). We choose the leader-worker architecture due to its implementation simplicity, and ensure its scalability by keeping the leader off the critical path. We now elaborate on the implementation techniques that, along with the novel deadline specification and enforcement semantics (§3.4), enable ERDOS to efficiently support D3.

3.6.1 Communication

ERDOS initializes itself by constructing a control plane between the leader and the workers, which is used by the leader to schedule operators to workers and synchronize their initialization, thus ensuring that all operators are ready to execute before transmitting any messages. The workers construct a data plane amongst themselves atop TCP sessions, which is used to communicate the messages sent between the operators. This allows ERDOS to keep the leader off the critical path, while still enabling centralized scheduling decisions.

ERDOS provides a rapid communication of messages by choosing the underlying communication channel based on whether it connects operators: (*i*) on the same worker, or (*ii*) on different workers. While the communication for (*ii*) is multiplexed atop the data plane among the workers, operators on the same worker store data on the heap and communicate a reference to it over Rust's inter-thread channels, enabling rapid delivery of large messages and safe zero-copy communication using Rust's compile-time mutability checks.

3.6.2 Operator Execution

Workers execute computation by maintaining an execution lattice, a dependency graph of callbacks which guarantees the processing of message and watermark callbacks in timestamp order, thus providing lock-free access to state. Upon receiving a message, a worker retrieves a view of the state using `get_committed` (§3.4.4), and inserts into the lattice a *bound callback*, consisting of the state, the `Context`, the callback, and the received message. Similarly, upon the receipt of a watermark, the worker verifies if it acts as a low watermark across the operator's input streams, and inserts a callback that commits the state upon completion.

This execution lattice serves as a run queue for a worker's multi-threaded runtime. A set of threads retrieve and execute the callbacks, and notify the lattice upon their completion to unlock further dependencies (e.g., callbacks with higher timestamps). ERDOS allows operators to override the ordering semantics of the lattice to fine-tune the parallelism and state-management. For example, an operator may manually synchronize updates to its state, and ask ERDOS to execute all its callbacks in parallel by specifying that all timestamps are equivalent, and thus ready to execute concurrently.

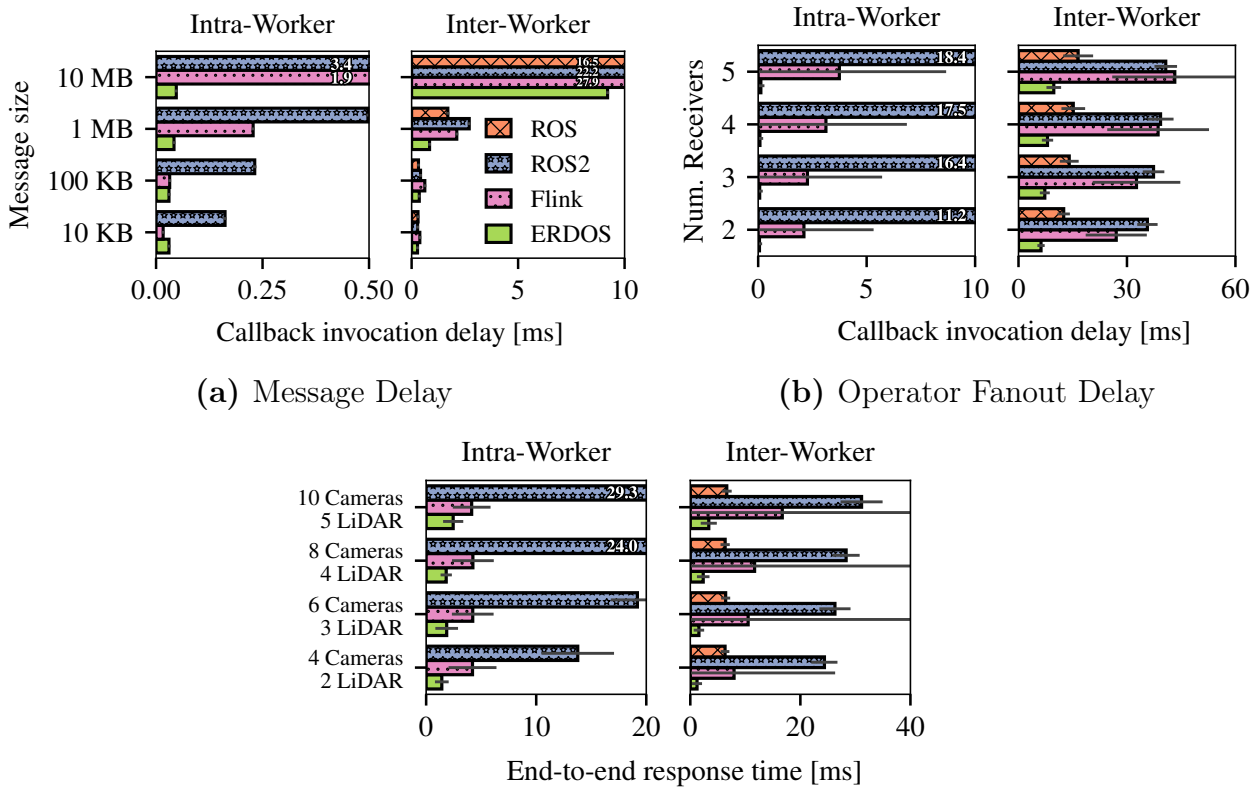
3.6.3 Deadline Management

The worker also ensures that the deadlines are initialized, and execute the exception handlers in case they are missed. To initialize a deadline, workers maintain per-stream statistics on the receipt of messages and the watermark status for each time t. Upon receipt of a message (M_t) or watermark (W_t), the worker updates the statistics, and invokes D_{SC}. If satisfied, the worker synchronizes the relative deadline $\mathcal{D}_i$ for t sent by the `deadline_stream` (line 3 in Lst. 3.1), and computes the absolute wall-clock time at which it expires. The deadlines, along with their handlers (D_{EH}), are maintained by the worker in a priority queue ordered by the expiration of the absolute deadline.

A deadline is removed from the queue when the operator satisfies D_{EC} or misses the deadline. Workers maintain per-stream statistics of the transmission of messages and watermarks, and remove the deadline and the D_{EH} from the queue upon satisfaction of D_{EC}. Further, workers poll the queue, and invoke the D_{EH} according to either the *Abort* [55] or *Continue* policy upon the expiration of a deadline. Similarly, at the transmission of each message, ERDOS checks D_{EC}, and stops the deadline upon satisfaction, by removing the reference to the handler from the priority queue.

3.7 Evaluation

Open-source AV pipelines (e.g., Autoware [35], Apollo [40]) do not include models and lack feature-complete integration with realistic open-source simulators, which are required to measure the efficacy of D3. Thus, we developed Pylot, a state-of-the-art AV that achieves a competitive score on the map track of a simulated AV challenge and drives real AVs. We use Pylot to evaluate D3 and ERDOS, and seek to answer:



(a) Message Delay

(b) Operator Fanout Delay

(c) Sensor Scaling

Figure 3.3: Messaging Performance. We evaluate the response time of ERDOS for **(a)** varying message sizes, **(b)** operator fanout, and **(c)** pipeline sizes for intra-worker and inter-worker communication. In all cases, we find that ERDOS' optimized implementation and D3's operator model helps achieve better performance.

1. How does ERDOS compare with other systems? (§3.7.2)
2. Does ERDOS enable the fulfillment of deadlines? (§3.7.3)
3. Do D3's dynamic deadlines improve safety? (§3.7.4)

Experimental Setup. We perform all our experiments on a machine having 2× Xeon Gold 6226 CPUs, 128GB of RAM, and 2× Titan-RTX GPUs, running Linux Kernel 5.3.0. This configuration closely reflects the hardware used in our AVs.

3.7.1 Pylot: An AV Development Platform

The construction of Pylot was a multi-year effort leading to approximately 28k lines of code, with an additional 434 lines required to port it to a real AV[1]. Pylot contains dozens

[1]A demo of one of our test drives: https://tinyurl.com/yaumb4sn

of components and is, to the best of our knowledge, the most comprehensive open-source AV pipeline with trained models. We now briefly describe a few components relevant to our evaluation (see Chapter 4 for an extended discussion).

Pylot's perception module comprises of components that perceive objects, lanes, and traffic lights using multiple cameras. While Pylot provides several implementations for each component (suited for different driving environments), our experiments use EDet2 to EDet6 from the EfficientDet family [289] in the order of increasing accuracy and runtime. This enables us to experimentally evaluate the runtime-accuracy tradeoff as accuracy varies from 39.6 mAP (EDet2) to 51.7 mAP (EDet6), and the runtime varies from 20ms to 262ms.

The Pylot planning component contains implementations of Hybrid A* [95], RRT* [169] and FOT [324, 329] that perform best under different driving scenarios [178, 172, 230]. Since we execute our experiments in an urban environment, we utilize the FOT planner that discretizes the configuration space, and is fast if coarse discretizations are chosen, with poor discretizations producing infeasible plans. We create configurations of the planner by varying the space discretization from 0.3m to 0.7m, and evaluate them in Fig. 2.2c, which plots the lateral jerk while performing a swerving maneuver. We observe that configurations with longer deadlines, and lower space and time discretization provide increased comfort.

Methodology. The tight coupling between the existing open-source AV pipelines and their underlying systems (e.g., Autoware and ROS [248], Apollo and CyberRT [41]) makes porting these pipelines to ERDOS a time-consuming engineering effort. Similarly, migrating Pylot to the underlying systems used by these pipelines is a challenging undertaking. As a result, our evaluation follows a two-pronged approach. First, we measure low-level system metrics (e.g. callback invocation delay) to show a lack of regression in ERDOS' realization of D3 as compared to the other systems (§3.7.2). Second, we extend the CARLA challenge [298] to construct a challenging benchmark for AV systems spanning 50km of simulated driving. We port the execution models used by the underlying systems to ERDOS and use the benchmark to highlight the efficacy of D3 when compared to these models (apart from any engineering benefits that come from ERDOS) (§3.7.3, §3.7.4).

3.7.2 ERDOS' Performance vs. Other Systems

We evaluate the latency of ERDOS with respect to message size, operator fanout, and pipeline complexity. We compare against (*i*) ROS, a widely used platform for AVs [312, 35, 106, 8], (*ii*) ROS2, which provides more real-time guarantees [94, 199], and (*iii*) Flink [62] a data-driven streaming system that is closest to ERDOS due to its operator-centric programming model and usage of watermarks for unlocking computation.

Microbenchmarks. We measure the delay incurred by sending messages of increasing sizes between two operators and invoking a callback upon receipt of the message. By measuring the callback invocation delay, we compare how different systems contribute to AV pipeline's response time via the implementation of the communication stack and the scheduling of callbacks. We send messages at 30Hz, the frequency at which AVs process data [28]. Fig. 3.3a

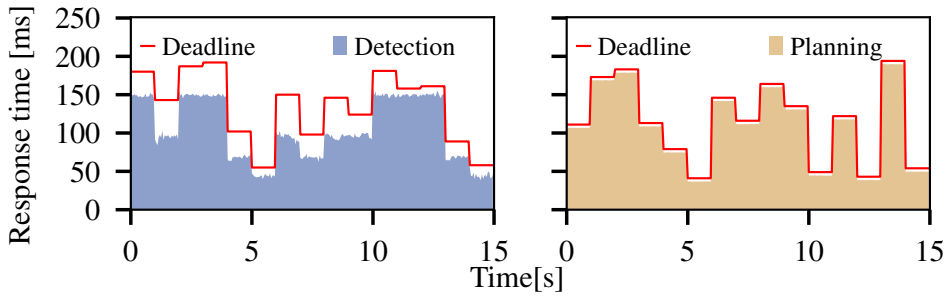


Figure 3.4: Meeting Deadlines. We vary the deadline every second and show how the modules respond to the new deadlines. Both detection and planning adapt to meet the deadline and the more adaptive planning module is better at using its time allotment.

shows the results across both intra-worker and inter-worker placements of the operators. ERDOS' intra-worker callback invocation delay remains constant across message sizes due to its zero-copy communication. Further, ERDOS' inter-worker implementation performs 2.0× better than ROS, and 3.2× better than ROS2, and 2.5× better than Flink when sending 1MB messages. We analyze the systems to attribute the overhead, and find that Flink and ROS have additional data copies and a more inefficient networking path accounting for 80% of the overhead, and slower serialization/deserialization responsible for 20% of the overhead. Moreover, we attribute ROS2's overhead to its use of the Data Distribution Service, which incurs additional costs for data conversion [199].

Next, we compare ERDOS' callback invocation delay to other systems' when broadcasting the output of an operator (e.g., camera image) to multiple operators (e.g., perception components), which is a common pattern in AVs. Fig. 3.3b shows that ERDOS sends a typical camera image message of 6MB to 5 operators in the same worker at a median latency of 0.12ms, 150× faster than ROS2 and 30× faster than Flink. When communicating across workers, ERDOS broadcasts to 5 operators at a median delay of 9.76ms, which is 1.7×, 4.2×, and 4.4× faster than ROS, ROS2, and Flink.

Response Time Benchmarks on Synthetic Pipelines. In order to measure the scalability to complex AV pipelines [312, 150, 145], we emulate Pylot with an increasing number of sensors sending data at 30Hz. We first instrument Pylot, and retrieve the mean size of each message type. Based on these measurements, we emulate a pipeline with an increasing number of sensors and operators, which sends messages totalling 925MB/s when processing 10 cameras and 5 LiDARs across 75 operators. Moreover, for a worst-case estimate of system overheads, we assume each operator has a 0ms runtime.

Fig. 3.3c compares the end-to-end response time of the pipeline when executed within a worker and across workers. We find that for 10 cameras and 5 LiDARs, ERDOS' intra-worker implementation exhibits a median response time of 2.5ms, which is 12× and 1.7× better than ROS2 and Flink. When placing each operator in its own worker, ERDOS exhibits a median response time of 3.4ms, which is 2.0×, 9.3×, and 5.0× faster than ROS, ROS2, and Flink.

Note that a realistic deployment of Pylot would colocate operators in workers, and thus the worst-case latency would be similar to that observed in the intra-worker graph.

Takeaway: *ERDOS' efficient implementation scales to large pipelines and enables AVs to meet more deadlines by minimizing the amount of time lost to system overheads when invoking computation due to message arrivals.*

3.7.3 Efficacy of ERDOS' Deadline Mechanisms

We evaluate the latency overhead introduced by the mechanism for implementing π_{DP} policies, the ability of components to proactively meet dynamic deadlines, and the effect of reactive measures in meeting end-to-end deadlines.

Latency Added by the Policy Mechanism. To achieve dynamic deadlines, applications define π_{DP}, which computes deadlines using pipeline data and sends deadlines to components (§3.4.2). We now investigate the latency overhead of π_{DP}. In order to isolate the latency of the mechanism from the latency of the π_{DP} logic, we use a *no-operation* π_{DP} that receives data from Pylot's components and sends static deadlines to the components. We measure Pylot's response time without and with the no-operation π_{DP} during a 35km drive, and we find that the policy mechanism increases the response time by less than 1%. The median and 90$^{\text{th}}$ percentile response times increase by 0.9ms and 2.3ms respectively.

Meeting Deadlines. We evaluate ERDOS' support for fine-grained changes in deadline allocations (§3.4.3) and if Pylot's components adapt to meet these changes. In the experiment, we use a policy that randomly changes deadline allocations every second. Fig. 3.4 shows the response time of Pylot's detection and planning during a short drive. We observe that while detection meets its deadline, it fails to utilize its entire time allocation. This is because the EfficientDet [289] family provides 8 models with different runtimes, and ERDOS chooses the model with the highest runtime that fits within the allocated deadline, which may be significantly higher. By contrast, the planning component fully utilizes its time allocation because it executes an anytime algorithm [324, 329].

Handling Deadline Misses. Deadline exception handlers (D_{EH}) ensure that a missed deadline does not delay downstream components (§3.4.4). In this experiment, we compare against a D_{EH} implementation based on ROS' actionlib, a preemptible task library. Fig. 3.5 (left) shows that ERDOS invokes D_{EH} 0.1ms after a deadline is missed, and it is 5× faster than ROS. This delay is acceptable for Pylot, as Fig. 3.5 (right) shows the per-component and end-to-end response time without D_{EH} (i.e., the data-driven execution model described in §2.2.1) and with D_{EH} during a 50km drive in simulation. Pylot without D_{EH} has a 0.6% end-to-end deadline miss ratio, whereas with D_{EH} it always meets the end-to-end deadline.

Takeaway: *ERDOS implements D3 by swiftly executing π_{DP}, enabling proactive strategies to meet deadlines, and rapidly taking reactive measures when deadlines are missed.*

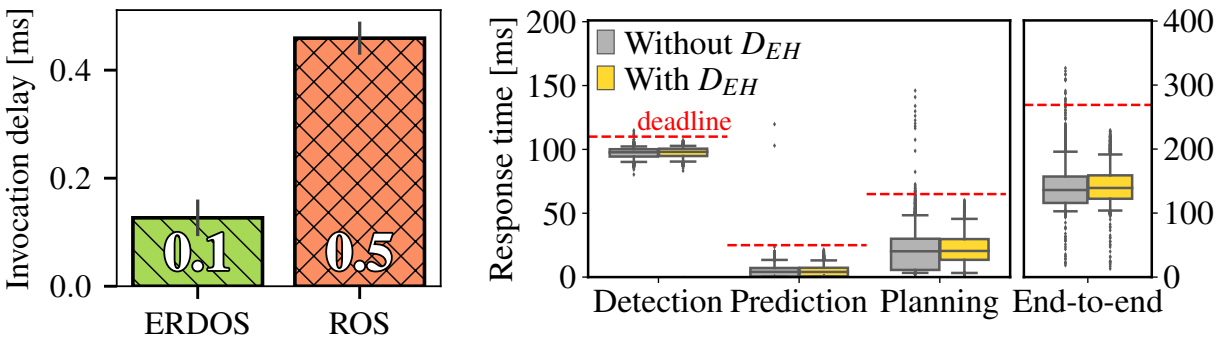


Figure 3.5: Impact of Exception Handlers. ERDOS supports fast invocation of handlers (left), and enables quick reactions to missed deadline (right), ensuring timely responses.

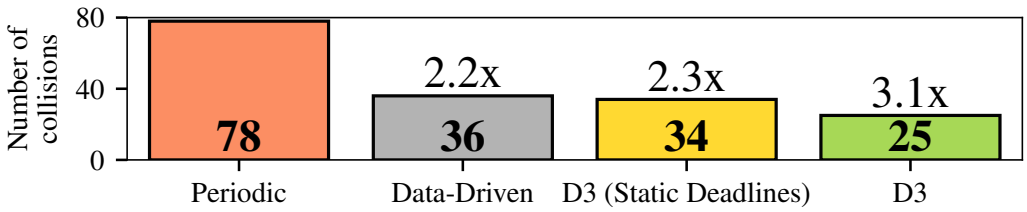


Figure 3.6: D3 Reduces Collisions. In a challenging 50km drive, ERDOS' realization of D3's dynamic deadlines reduces collisions by 68% over the periodic execution model.

3.7.4 Efficacy of the D3 Execution Model

We evaluate the efficacy of D3 by exploring a deadline allocation policy that adjusts the end-to-end deadline to avoid collisions in challenging scenarios. The focus of our work is not the design of policies, but to provide the mechanisms to implement such policies. Therefore, we present a baseline policy that adapts deadlines as a function of the AV speed and the trajectories of other agents. Our policy computes *reaction time*, defined as the sum of time to receive 8 sensor readings, which are sufficient to build a trajectory prediction for the agents, and the end-to-end runtime of the current configuration. The policy uses the reaction time and the AV's driving speed to estimate the AV's *stopping distance*. It then adjusts the end-to-end deadline depending on how close to other agents the AV will be at the end of its stopping distance. We compare Pylot's performance under dynamic deadlines to five static deadlines ranging from 125ms to 500ms.

Aggregate Study. We explore if our policy adjusts the deadlines to avoid collisions during a challenging 50km CARLA Challenge drive [298]. In this experiment, we adapt the detector in response to shorter deadlines, but keep all the other components fixed in order to limit the experiment duration (exploring all tradeoffs required 100 days of simulation).

Fig. 3.6 highlights the efficacy of D3 apart from the engineering of ERDOS by run-

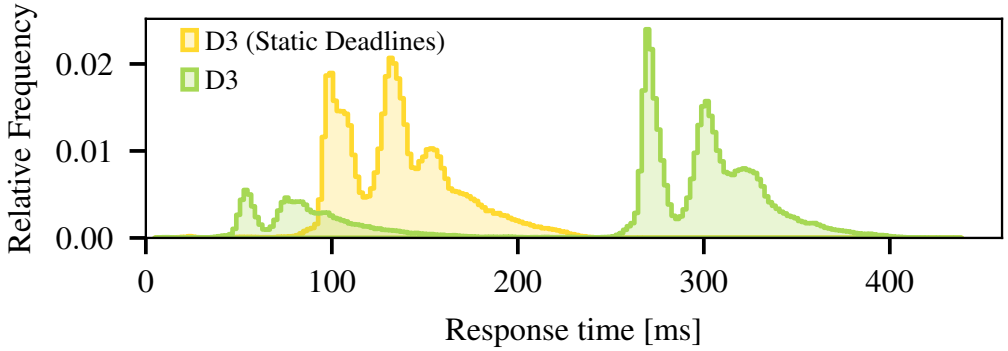


Figure 3.7: Response Time Histogram. D3 (Static Deadlines) enforces the static deadlines that perform the best during the drive and the variability is due to **C2**. By contrast, D3 with dynamic deadlines offers faster responses when needed, and executes more accurate computation during normal driving scenarios.

ning Pylot atop ERDOS using four execution models: (*i*) a periodic execution derived from WCET estimates (similar to Apollo [28] and Autoware [35], which execute most components periodically), (*ii*) the best data-driven configuration that executes each component upon receipt of all input data (similar to some ROS deployments, see §2.2.1), (*iii*) the best configuration with static deadlines enforced by D3's D_{EH}, and (*iv*) a D3 execution with dynamic deadlines enabled by our deadline allocation policy and ERDOS. The execution with our policy (D3) reduces collisions by 68% over a periodic execution, and by 26% over the best configuration with static deadlines because the policy reduces the deadlines in challenging scenarios. Finally, we compare the end-to-end response times of Pylot's D3 execution with dynamic deadlines with Pylot's best configuration with static deadlines. Fig. 3.7 shows that in most situations D3's Pylot execution runs a slow, high-accuracy configuration, but adapts to fast configurations when the environment demands it.

Takeaway: *ERDOS' dynamic deadlines result in significantly fewer collisions compared to the periodic execution and static deadlines used in state-of-the-art platforms.*

In-depth Study of Scenarios. We study the benefits of ERDOS' realization of D3 using two challenging scenarios that require the AV to adapt in order to avoid collisions.

Person Behind Truck. This scenario simulates a person illegally entering the AV's lane (see video[2]). The scenario is complicated by a truck that occludes the person until they enter the AV's lane. Thus, the AV cannot stop in time and must perform an emergency swerving maneuver. Since this maneuver requires a knee-jerk reaction, we expect the configurations that minimize the response time to perform better.

Traffic Jam. This scenario simulates merging into a traffic jam. The AV is required to come to a halt behind a vehicle and a motorcycle, while the other lane is lined up with vehicles. The motorcycle complicates this scenario as it requires the AV to perceive the object from

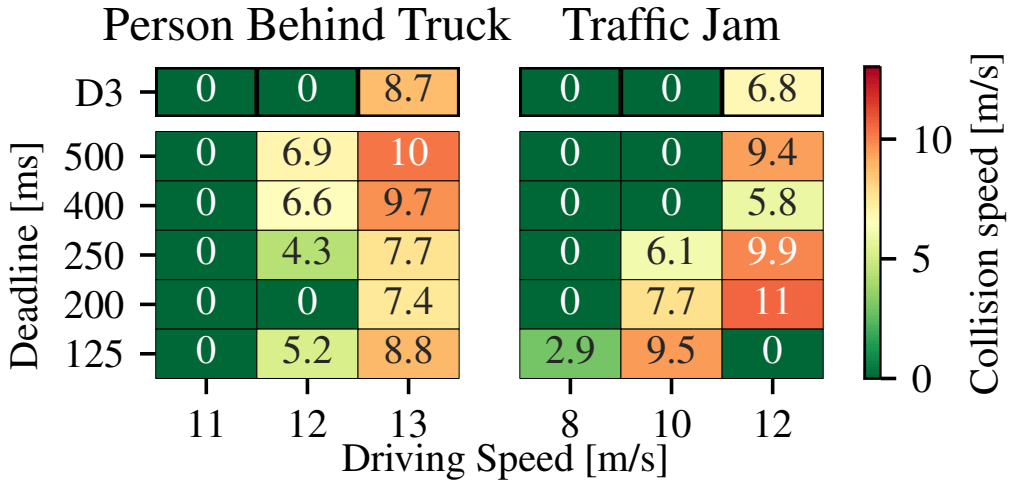


Figure 3.8: Versatility of D3's Dynamic Deadlines. Configurations with short deadlines reduce collision speed in the *person behind truck* scenario (left), but increase it in the *traffic jam* scenario (right). By contrast, D3 adapts Pylot's deadlines depending on the driving speed and scenario complexity resulting in fewer collisions.

afar in order to prevent a collision. Moreover, the vehicles on the other lane prevent the AV from performing an emergency swerve. While the previous scenario requires a fast response, this scenario needs consistent high-quality responses from the AV in order to prevent an otherwise-safe scenario from turning into an emergency.

In the experiment, we drive the AV at a fixed speed using a fixed set of hardware resources (see §3.7). We execute Pylot's five configurations with static deadlines (§3.7.4) and Pylot's D3 execution with dynamic deadlines computed by our policy. We use the driving speed of the AV at the time of the collision (i.e., *collision speed*) as a proxy for the impact of the collision, where a speed of 0m/s shows that Pylot avoided a collision.

In Fig. 3.8 we plot the collision speed across varying speeds. As expected, at a speed of 12m/s, the probability of successfully handling the *person behind truck* scenario increases with a decrease in response time. In this scenario all but the fastest configuration detect the person, which is visible 20m away. Thus, the configuration with the lowest response time (200ms) that detects the person 20m away prevents a collision, while configurations with higher response times collide with the person at collision speeds that increase with the response time. We note that the configuration with the lowest response time (125ms) collides as it detects the person too late (12m away) due to its low perception accuracy. On the contrary, in the *traffic jam* scenario, the slower, more accurate configurations allow the AV to reliably stop at 10m/s. This is because the motorcycle is partially occluded, and thus faster, less-accurate models perform poorly. Fig. 3.9 shows how Pylot adapts as our policy reduces the deadline once the person is visible in the *person behind truck* scenario.

Takeaway: *ERDOS' deadlines adapt in both scenarios, and avoid more collisions than*

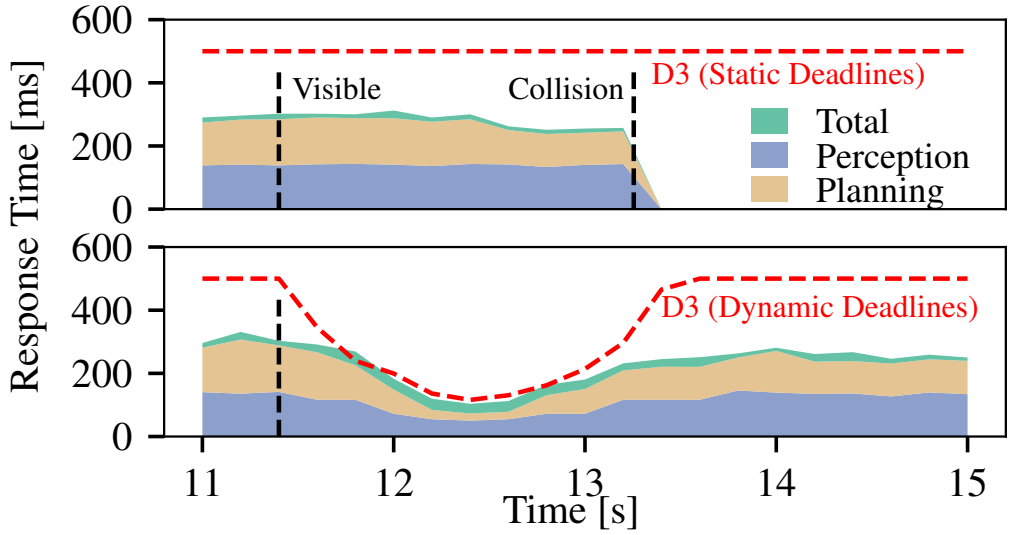


Figure 3.9: Adapting to Deadlines. D3 enables Pylot's components to meet dynamic deadlines and avoid a collision.

any static configuration[2].

3.8 Related Work

Data-Driven Execution Model. Vendors [312, 35, 311, 106, 8] are developing AV pipelines atop robotics platforms that provide a modular design and best-effort execution of the components (e.g., ROS [248], ROS2 [140], CyberRT [41]). As a result, vendors execute these pipelines as SLO-based best-effort applications that attempt to meet an environment-agnostic end-to-end deadline [312, 35, 40]. The AV pipelines are deployed as ROS/CyberRT processes, that either use the data-driven execution model to run each component to completion upon receiving all input (§2.2.1), which may delay downstream components due to runtime variability, or run components periodically [244, 28], which preclude adaptations to meet dynamic deadlines. Moreover, these platforms do not offer a system-managed consistent view of time, and thus lack mechanisms to specify and enforce deadlines, or reason about the available execution time [312]. By contrast, ERDOS enables the development of D3 applications by offering a consistent view of time via logical times, an automatic mapping of logical time to wall-clock time, which components can use to reason about deadlines and available execution time, and apply reactive measures to mitigate missed deadlines.

Stream processing systems such as Flink [62], Cloud Dataflow [121], MillWheel [9], and Naiad [205] also utilize the data-driven execution model. Although, these systems inspired elements of our design (e.g., logical time [205, 62, 121], watermarks [13, 62, 308], intermediate

[2] *Static vs dynamic deadlines in Pylot:* https://tinyurl.com/y24p4g8d

results [2]), these systems are designed for massively parallel data processing, and embed architectural and implementational decisions that make them unconducive to the development of AVs. For example, Naiad paralellizes an application by partitioning data across workers, which each execute an entire copy of the dataflow computation (AV sensor data is not partitionable). Moreover, these systems are unable to realize the D3 model because, unlike ERDOS, they lack APIs to specify environment-dependent deadlines, implement proactive strategies to meet these deadlines, and apply reactive measures when deadlines are missed.

Periodic Execution Model. Hard real-time systems conduct schedulability analyses driven by WCET estimates to guarantee that deadline constraints are met [189, 303, 54, 46, 123, 92, 196]. However, AV components preclude the accurate estimation of WCETs due to environment-dependent runtimes and large input spaces [288, 16, 17], or the non-deterministic nature of the algorithms they use [169, 17, 72, 317]. Thus, developing AVs as such systems requires use of conservative WCETs to derive the periodicity of execution for each component [87]. However, periodic executions cannot meet dynamic deadlines, and trade accuracy to ensure that components with a large gap between mean and worst-case execution time meet deadlines [17, 74, 51]. To address the former, real-time applications implement *mode changes* [250, 71, 20], which depend on WCETs to verify if transitions between modes lead to deadline misses [250, 303, 283, 189]. By contrast, *adaptive real-time systems* [51, 176, 197, 263] support the execution of components without WCET. These systems minimize deadline misses by using feedback-based policies to choose the best service level from multiple application-defined levels (similar to §3.4.3's changing implementations), but lack mechanisms to enforce deadlines and mitigate deadline misses.

D3 subsumes prior systems by allowing the execution of both mode changes and adaptive real-time applications. The developers of D3 applications can specify mode changes using the deadline policy (π_{DP}) and trigger them to perform graceful degradation (on deadline misses or environment changes) using D3's feedback loop (§2.2). Furthermore, ERDOS's use of timestamps and watermarks helps with tracking the causality of messages back to sensor data, and along with system-managed state makes ERDOS more amenable to analysis and verification than current AV platforms. Thus, applications can exploit prior certification and predictability analysis by restricting themselves to a limited set of mode changes.

3.9 Conclusion

We highlight two key characteristics of AVs, and introduce D3, an execution model for applications that must maximize accuracy in the presence of dynamic deadlines, and demonstrate how existing solutions fall short. We realize D3 in ERDOS, atop which we build an AV, Pylot, and find that D3 reduces collisions by 68%.

Chapter 4

Pylot: A Modular Platform for Exploring Latency-Accuracy Tradeoffs

Chapter 3 briefly introduced the key components of Pylot that enabled the evaluation of the efficacy of D3 under driving environments that required a dynamic tradeoff between the runtime of the components and their accuracy. More broadly, Pylot is built as a platform for AV research and development with the goal to allow researchers to study the effects of the latency and accuracy of their models and algorithms on the end-to-end driving behavior of an AV. This is achieved through a modular structure enabled by our high-performance dataflow system, ERDOS, that represents AV software pipeline components (object detectors, motion planners, etc.) as a dataflow graph of operators which communicate on data streams using timestamped messages. Pylot readily interfaces with popular AV simulators like CARLA, and is easily deployable to real-world vehicles with minimal code changes.

To reduce the burden of developing an entire pipeline for evaluating a single component, Pylot provides several state-of-the-art reference implementations for the various components of an AV pipeline. Using these reference implementations, a Pylot-based AV pipeline is able to drive a real vehicle, and attains a high score on the CARLA Autonomous Driving Challenge. In this chapter, we provide an overview of the design decisions that underlie the development of Pylot and present several case studies enabled by Pylot, including evidence of a need for context-dependent components, per-component time allocation, and the ability to exploit cloud resources for safer driving. We open-source Pylot for community benefit and make it available at `https://github.com/erdos-project/pylot`.

4.1 Introduction

Over the past decade, advances in application areas such as object detection and localization [254, 192, 289], vehicle motion planning and control [169, 95, 329, 269, 131], and behavior and motion prediction [179, 256, 203] have enabled rapid advances in AV technology. Reciprocally, the promise of AV technology has motivated many recent advances in

emerging technologies like artificial intelligence, with, e.g. driving-based datasets becoming fundamental baselines for modern computer vision [158, 112, 77, 285, 32].

Various vendors [113, 103, 216, 35, 40] realize the computation that underlies an AV as a pipeline similar to the one shown in Fig. 4.1. This modular approach has been the standard for AV pipelines since at least the DARPA Grand Challenge of the mid-2000s [301].

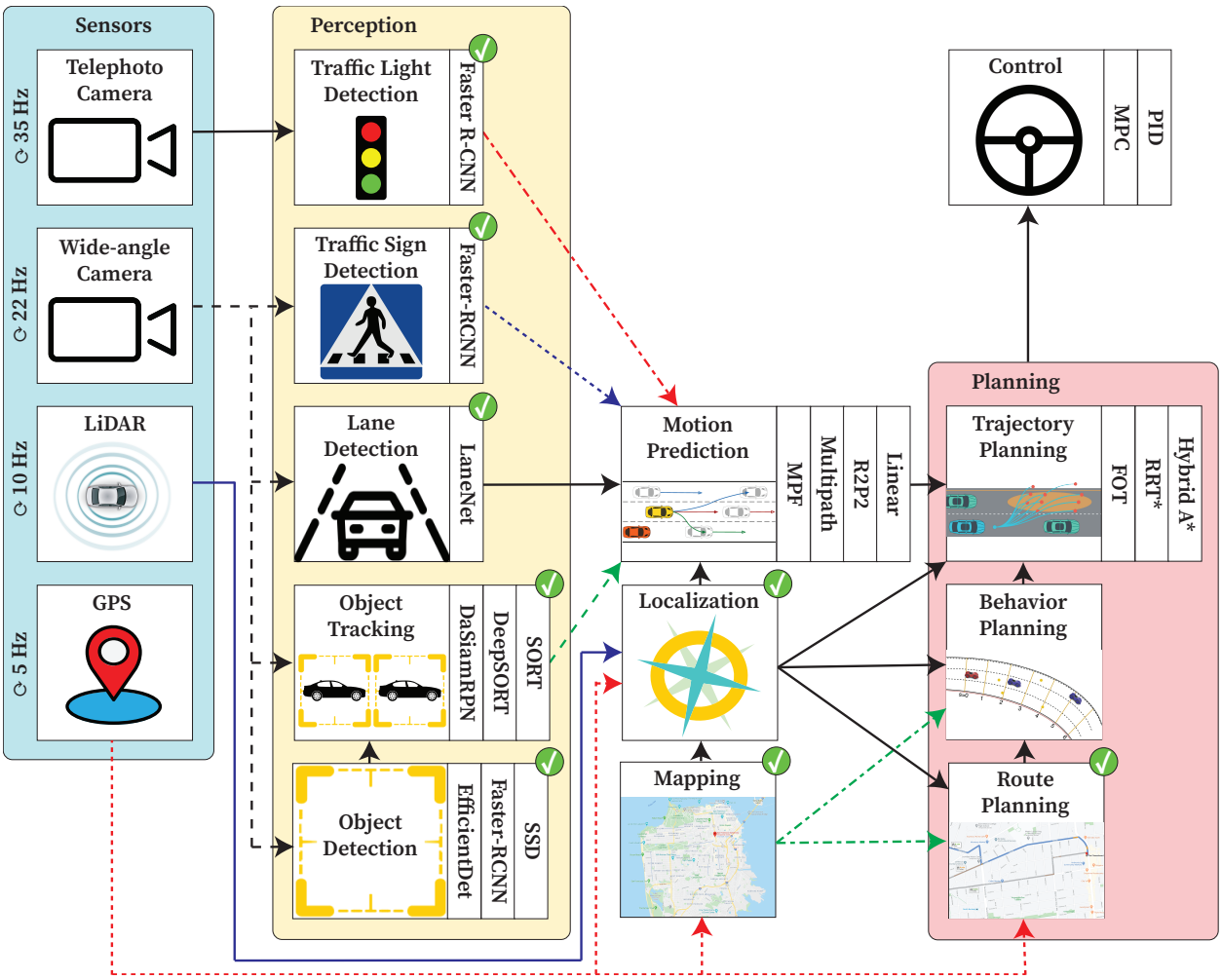


Figure 4.1: Pylot's AV pipeline consists of several interconnected modules (e.g. perception, planning). For each component in these modules, Pylot provides reference implementations along with "perfect" implementations (for those with a green check mark) that access ground truth data from the simulator.

While this decomposition of the driving task into multiple components has enabled researchers to innovate independently on each component, it has also led to the development of problem-specific evaluation metrics that fail to account for the end-to-end driving behavior of the AV [242]. For example, even driving-based datasets like KITTI [112] and Cityscapes [77],

which are used to develop ML models for the perception component, utilize static evaluation metrics such as average precision [110] that fail to account for the runtime of the model. Exploring the tradeoff between the latency of a component and its accuracy on offline datasets is paramount for safety-critical applications such as AVs where correctness is defined as a function of both the accuracy of the algorithms and their end-to-end runtime [182].

In this chapter we introduce Pylot, a modular platform that enables the study of this critical tradeoff between the accuracy of a module and the effects of its runtime on the safety of the AV. Pylot is built on top of ERDOS, our high-performance, deterministic dataflow system [102], and provides: (*i*) state-of-the-art reference implementations for components allowing researchers to evaluate their algorithms or models in the context of a realistic AV pipeline, (*ii*) "ground truth" implementations that allow the development of components and their debugging in the context of an idealized AV pipeline, and (*iii*) a portable interface that enables seamless transition between a simulator and a real vehicle. Pylot is open source, and is the top submission on the CARLA Autonomous Driving Challenge HD map track [1] [298], thus granting the AV community a platform comparable to proprietary pipelines.

The remainder of this chapter is organized as follows: §4.2 gives an overview of existing open AV platforms, §4.3 presents the critical design goals of Pylot, and §4.4 discusses how our implementation achieves these goals. Further, §4.5 describes (*i*) the reference component implementations, (*ii*) a prototype AV pipeline built with these implementations, and (*iii*) our experience of porting this pipeline to a real-world AV. Finally, §4.6 presents several case studies enabled by Pylot that evaluate in-context performance metrics of AV pipeline components, and §4.7 concludes and discusses future work.

4.2 Related Work

Recent work in object detection has emphasized the need to achieve a balance between the latency and accuracy of an ML model for a given application [147]. While there have been efforts to both define evaluation metrics that integrate latency and accuracy of the perception module in the context of AVs [182, 242], and develop flexible backbones for object detection models that enable developers to choose an optimum point in the latency-accuracy curve [289, 52], they fail to account for the effect of the runtime on the end-to-end driving behavior. Moreover, to the best of our knowledge, such metrics do not exist for other modules in the AV pipeline.

On the other hand, open-source implementations of AV pipelines lack significantly in their ability to allow developers to explore the accuracy of individual components in the context of an end-to-end pipeline. For example, both Autoware [35] and Baidu's Apollo [40] provide limited interfaces to freely-available simulation platforms. Specifically, neither of these AV pipelines allow photo-realistic simulation of cameras, thus failing to account for the accuracy and runtime of the perception module. Moreover, they omit pre-trained models

[1] A demo video of Pylot running on the CARLA Challenge is available at https://tinyurl.com/y6ozzpwd.

for other components specific to any simulation platform which raises the bar for testing a component. Finally, the debugging ability of these platforms is also limited by their choice of the underlying publisher-subscriber communication paradigm which complicates the deterministic replay of driving scenarios [312].

4.3 Design Goals

The central design goal of Pylot is to support the study of the tradeoff between the runtime of components and their accuracy in the context of the end-to-end driving behavior. To achieve this goal, Pylot must fulfill three key requirements:

Modularity. To enable rich evaluation of new models and algorithms, Pylot must provide modular components that can be swapped out for alternate implementations. This allows developers to compare their components with state-of-the-art implementations on similar driving scenarios in addition to evaluating their components using standard metrics on offline datasets. Moreover, a "plug-and-play" architecture supports the future introduction of new modules and components without requiring a tedious overhaul of the entire pipeline.

Portability. Developers must be able to transition between different simulators and real-world vehicles in order to evaluate their components across various driving environments. For example, a developer should be able to ensure that their planning algorithm provides similar behavior on a traffic simulator such as SUMO [175], a dynamic world simulator such as CARLA [96] or AirSim [272], and real-world vehicles. A key stipulation of portability is that Pylot must be highly performant. This allows components developed in Pylot to be tested in simulation, and effortlessly deployed to a real vehicle without any additional changes that might affect the tradeoff between latency and accuracy. On the other hand, the runtimes of components observed in a real vehicle can be faithfully reproduced in order to ensure that the latency-accuracy tradeoff can be correctly explored in simulation.

Debuggability. Ensuring the safety of models and algorithms requires extensive testing across various scenarios. Thus, Pylot must provide developers with tools that allow them to understand and easily debug their components when they exhibit abnormal or unsafe behavior. In order to reproduce unsafe behaviors, the software system must be output deterministic (i.e. produce the same output given the same inputs) [21], and the pipeline must enable seamless logging of the data necessary to reconstruct the AV's behavior.

4.4 Achieving Design Goals

We realize the design requirements outlined in §4.3 in our implementation of Pylot, which consists of approximately 28,000 lines of Python code. While Pylot executes atop our low-overhead open-source streaming dataflow system implemented in Rust [102], we chose to implement Pylot itself in Python in order to enable faster prototyping and easier interfacing

to both simulators such as CARLA [96] and deep learning frameworks such as PyTorch [236] and Tensorflow [3]. The remainder of this section discusses the design of Pylot by focusing on how it achieves *modularity* (§4.4.1), *portability* (§4.4.2) and *debuggability* (§4.4.3).

4.4.1 Modularity

Pylot's modular structure is achieved through a dataflow programming model, where the AV pipeline is structured as a directed graph in which vertices, also known as *operators*, perform computation (e.g. running object detection) and edges, also known as *streams*, enable communication through timestamped messages (e.g. transmitting bounding boxes of detected objects). This inter-component communication style is reminiscent of the familiar "publisher-subscriber" model with the operators akin to ROS nodes, and streams akin to the ROS publishers and subscribers. Similar to the publisher-subscriber model, the dataflow programming model limits interactions between operators to streams thus allowing them to be swapped as long as they conform to the same interface (e.g. an object detection component based on Faster-RCNN [254] can be swapped for one based on EfficientDet [289]).

However, unlike the publisher-subscriber model, the dataflow programming model allows swapping of components that differ in their runtimes and resource requirements without requiring cascading changes throughout the entire pipeline. Specifically, while a ROS node synchronizes incoming data from multiple sources by fetching data from the required publishers at a fixed frequency, a dataflow system allows developers to register callbacks that get invoked upon the arrival of synchronized data across the requested streams. The dataflow system underneath Pylot seamlessly synchronizes data across multiple streams by requiring the operators that publish on those streams to send a special *watermark* message upon completion of the outgoing data for a specific timestamp [308, 62, 205, 121]. Hence, while swapping a component with a different runtime in the publisher-subscriber model requires fine-tuning the frequency at which the downstream operators invoke their computation, a dataflow system is robust to runtime variabilities and enables highly modular applications.

4.4.2 Portability

In order to allow pipelines developed in simulation to be ported to real-world vehicles, Pylot must support high-throughput processing of the data generated by a vehicle's sensors [113, 312, 22]. This is enabled through our custom high-throughput and low-latency dataflow system that outperforms ROS, a commonly used robotics middleware, by 30% in terms of communication latency [102] while providing a shim layer that allows *operators* to interface with legacy ROS code. The underlying system *transparently* schedules the parallel execution of these operators across machines according to their resource requirements, and provides collocated modules with zero-copy communication via shared memory queues. Moreover, this transparent scheduling and communication does not require any code changes, and coupled with the shim layer for legacy ROS code allows seamless and piecemeal porting of Pylot to different hardware platforms.

Conversely, Pylot also enables the transition from real-world vehicles to simulation by enabling the precise replication of the runtimes of components in a simulator. We achieve this through Pylot's integration with the CARLA simulator. CARLA provides two modes of execution: (*i*) *synchronous*, where the simulator allows the instantaneous application of control commands by pausing the simulator after sending sensor inputs to the client, and (*ii*) *asynchronous*, where the simulator moves forward and applies the control command when Pylot finishes execution. While CARLA's asynchronous mode should allow control commands to be precisely applied when the computational pipeline finishes executing, it lags behind due to the high rendering cost of the underlying graphics engine. This results in the imprecise application of control commands from the pipeline to the simulated vehicle.

To enable a timely application of the commands, we developed a *pseudo-asynchronous* execution mode that allows Pylot to maintain tight control over the simulation loop. In this mode, we run CARLA *synchronously* with a high frequency (over 200Hz), and attach a synchronizer between the control module and the simulator. This synchronizer tracks the runtime of the pipeline for each sensor input, and buffers the control command until the simulation time is at least the sum of the sensor input time and the runtime of the pipeline.

Both the synchronizer and Pylot can be ported to other simulators [272, 261] with minimal effort. Developers are only required to implement drivers to extract sensor data from the simulator (e.g., camera frames, LiDAR point clouds), and an operator to send control commands to the simulator (as evidenced by our port of Pylot to a real vehicle in §4.5.6).

4.4.3 Debuggability

Pylot integrates with CARLA's ScenarioRunner [267], and provides a suite of test scenarios based on the National Highway Traffic Safety Administration's pre-crash scenarios [210]. In contrast to other AV pipelines implemented atop a publisher-subscriber model [35, 312, 40], Pylot's underlying dataflow programming model enables deterministic replay of such scenarios, thus allowing easier debugging of components. This deterministic execution is achieved by performing computation on the receipt of *watermark* messages [205, 62, 12].

To aid in debugging the latency and accuracy of components, Pylot provides extensive logging of both output data and the runtime of each component. Specifically, Pylot can be configured to log sensor data, and outputs of internal modules such that scenarios can be reproduced and debugged offline. Moreover, Pylot provides fine-grained logs for the execution time of each component that can be visualized using trace profiling tools [300] and replayed in order to study the latency-accuracy tradeoffs in deterministic scenarios.

To aid in development, Pylot provides "perfect" modules that obtain ground-truth data from CARLA, which can be used to test other modules in isolation (e.g., planning using perfect perception), and to generate new training data sets. Moreover, Pylot can be deployed with different sensor setups (e.g., cameras only, cameras and LiDARs), and can be used to study the end-to-end performance of each setup, and as well the robustness of solutions when only partial or noisy sensor data is available [241].

In addition, Pylot provides visualizations for important information, such as detected ob-

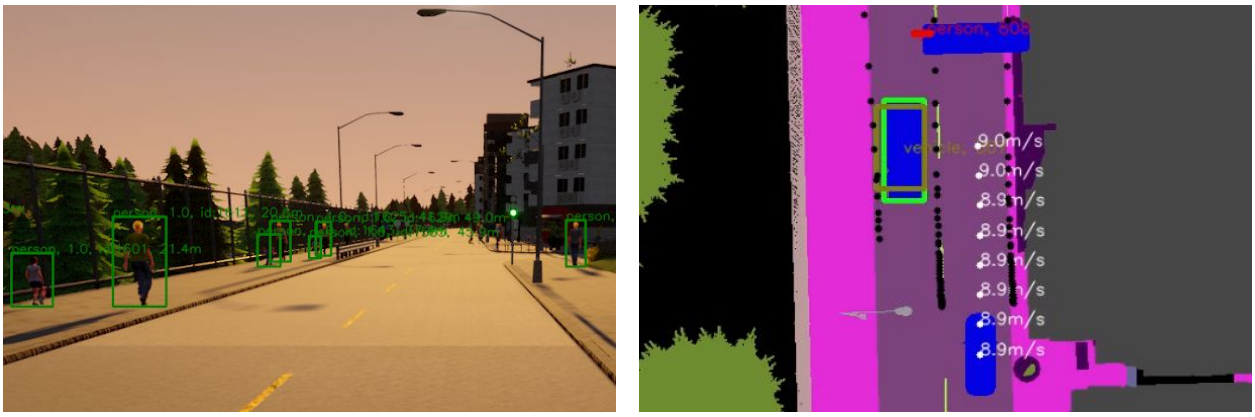


Figure 4.2: Pylot's visualizations for critical components. The object tracker view (left) shows the bounding boxes and the identifiers of detected agents in the camera frame. A bird's eye view of the planning module (right) includes lanes, predictions for agents, and waypoints computed by the planner.

jects, lanes, and traffic lights; sensors including cameras, LiDARs, and inertial measurement units; and algorithmic outputs for depth perception, pose estimation, behavior prediction, semantic segmentation, and planning. Fig. 4.2 exemplifies two such visualizations: (*i*) the output of one of Pylot's reference object trackers, and (*ii*) a bird's eye view of the information used by Pylot's planners to compute trajectories (detected obstacles, lanes, traffic signs, predicted trajectories, and proposed waypoints).

4.5 The Internals of Pylot

The goal of Pylot is to accelerate research into new algorithms and models for autonomous driving as well as the design of the underlying software systems. To support this goal, Pylot provides: (*i*) state-of-the-art algorithms, pre-trained models, and evaluation metrics, (*ii*) the ability to selectively replace components and modules with ground-truth data from simulators, (*iii*) an easily extensible deterministic runtime environment, and (*iv*) a range of challenging driving scenarios. In the rest of this section, we describe the key components and modules in Pylot (c.f. Fig. 4.1), and our experience of porting Pylot from a simulator to a real Lincoln MKZ test vehicle and driving it on a test track[2].

4.5.1 Object Detection

The object detection component comprises of operators that process camera and LiDAR data to detect, and localize objects, lanes, and traffic lights. These operators communicate their results via an `ObstacleMessage` that contains the bounding box of the detected object

[2]A demo video of Pylot running on the MKZ is available at https://tinyurl.com/y5vsly3f.

along with the confidence score returned by the ML model. Users may run multiple versions of these models in parallel in order to benchmark against each other and against perfect detection by using the standard accuracy metrics (e.g. mAP) provided by Pylot.

Pylot allows drop-in replacements of models conforming to the Tensorflow Object Detection API [147], and provides several configurations of Faster-RCNN [254] and SSD [192] models trained on data collected from the CARLA simulator using Pylot. In addition, we also utilize the EfficientDet [289] family of models which are built atop of a variable-sized network backbone. This property of EfficientDet along with the different configurations of Faster-RCNN and SSD allow the exploration of the runtime-accuracy tradeoff space.

4.5.2 Object Tracking

The tracking component estimates the bounding boxes of objects over time, and maintains consistent identifiers for the objects across detections. Pylot provides three object trackers that cover the two main tracking approaches: (*i*) *tracking-by-detection* continuously updates tracker state with bounding boxes received from object detection, and (*ii*) *detection-free tracking* follows a fixed number of objects over time. These trackers utilize the bounding boxes from the `ObstacleMessage` and communicate their results via an `ObstacleTrajectoryMessage` that contains the identifier for each obstacle.

We find that each tracker has context dependent performance and accuracy tradeoffs. For example, the SORT [48] tracker is a lightweight multiple object tracker of the tracking-by-detection variant that uses Kalman filters to estimate object positions between frames assuming a constant linear velocity model, and the Hungarian algorithm to match bounding boxes upon detection updates. While SORT is fast, it offers low accuracy in the presence of object occlusions or camera motion. However, DeepSORT [326] improves on these limitations by incorporating appearance information for each tracked object, at the cost of an increased runtime from executing a convolutional feature extraction model.

On the other hand, DaSiamRPN [346] is a detection-free single object tracker that leverages a siamese feature extraction network to learn distractor-aware features. The tracker relies on neural network inference to track an object thus providing more accurate estimations between detections at the cost of increased runtime. However, the "best" choice of tracker depends on the situation. For example, SORT performs best under emergency when low runtime is critical and DeepSORT is best in regular driving, while DaSiamRPN excels in scenarios that demand high accuracy for a small number of objects (e.g. urban driving).

4.5.3 Prediction

The prediction modules uses the tracked history of nearby agents (vehicles, pedestrians, etc.), along with scene context from LIDAR, to predict future agent behavior. Specifically, it receives the bounding box and the identifier of each obstacle at every time instant through the `ObstacleTrajectoryMessage`, and generates an `ObstaclePredictionMessage` containing the past and the predicted trajectory for each obstacle.

Pylot contains multiple implementations of the prediction module trained on CARLA data that represent the major classes of ML based approaches for trajectory prediction. As a baseline, Pylot also provides a linear predictor that utilizes a linear regression model that assumes that each agent will travel at a constant velocity and predicts their forward position based on their past trajectory. While this predictor is fast and hence ideal for emergency collision-avoidance scenarios, it does not account for the behavior of multiple agents.

Additionally, Pylot provides R2P2 [256], a state-of-the-art single-agent trajectory forecasting model which learns a distribution over potential future trajectories that is parameterized by a one-step policy using a gated recurrent unit, and attempts to optimize for both quality and diversity of samples. To extend R2P2 to the multi-agent setting, Pylot runs R2P2 on every agent by rotating the scene context and past trajectories of other agents to the ego vehicle coordinate frame. Finally, Multipath [67] uses a lightweight neural network to obtain a useful representation of the scene and then applies a smaller network features for each agent to output predictions. Because the per-agent computation can be batched, Multipath is fast but less accurate owing to its inability to explicitly consider agent interactions. In contrast, Multiple Futures Prediction (MFP) [291] jointly models agent behavior, leading to increased runtime but more accurate predictions.

4.5.4 Planning

The goal of the planning module is to produce a safe, comfortable, and feasible trajectory that accounts for the present and future possible states of the environment. To achieve this, the planning module in Pylot synchronizes the output from all the other modules to construct a `World` representation that contains the past and future trajectories of all agents in the scene, along with the location and state of static objects such as traffic lights, traffic signs etc. Crucially, this allows Pylot to selectively utilize ground truth information for any of the previous components or modules and ascertain the effects of the accuracy and runtime of a selected set of components on the end-to-end driving behavior of the AV.

The planning module is comprised of three components: route, behavioral, and motion planners, with the latter having the greatest effect on the comfort and the end-to-end runtime of the AV. Hence, Pylot provides implementations for each of the three main classes of motion planners: graph search, incremental search, and trajectory generation [178, 172, 230].

Graph-based search planners (e.g. Hybrid A* [95]) discretize the configuration space as a graph and provide fast results at lower discretizations. However, a poor choice of the discretization value may produce infeasible paths. On the other hand, *incremental search planners* (e.g. RRT* [169]) gradually build a path by sampling the configuration space instead of precomputing a fixed set of configuration nodes. They are not limited by the initial graph construction, and thus provide the ability to fine-tune the accuracy of the result for any given computation time. Finally, *trajectory generation planners* (e.g. Frenet Optimal Trajectory [324, 329]) construct a set of candidate paths, which they validate for collisions and physical constraints. While the resulting paths are smoother than their counterparts, trajectory generation may omit feasible paths if the discretization is too coarse.

4.5.5 Control

The control module receives waypoints and target speeds from the planning module. It aims to closely follow the provided waypoints while maintaining its target speed. Pylot offers two control options: a Proportional-Integral-Derivative (PID) controller, and a Model Predictive Control (MPC) controller. Both options compute commands that adjust brakes, steering, throttle, and send these to the simulator or a real-world vehicle's drive-by-wire kit.

While Pylot currently comprises of the modules described above, developers can integrate multi-functionality modules (e.g. MPC for both motion planning and control [23, 188]) or replace entire subgraphs of the pipeline with end-to-end solutions [184, 339, 319].

4.5.6 Deploying Pylot on an Autonomous Vehicle

In order to port Pylot from the CARLA simulator to a real Lincoln MKZ AV test vehicle, we made the following minimal changes:

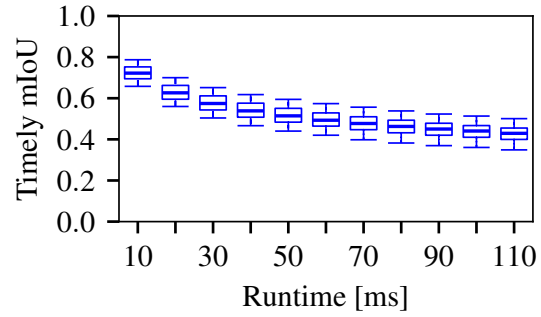
1. **Sensors:** We utilized the shim layer that injects data from ROS topics exposed by the drivers of the AV's sensors.
2. **Control:** We modified Pylot to send control commands to the ROS node exposed by a drive-by-wire kit [86].
3. **Localization:** We used the Normal Distributions Transform (NDT) algorithm [50] implemented in Autoware [35].
4. **Model replacement:** We replaced our CARLA-trained detection and tracking models with off-the-shelf models trained on real-world datasets.

As mentioned previously, porting from simulation to a real vehicle required only 434 lines of Python code. This efficiency demonstrates the flexibility of our interfaces.

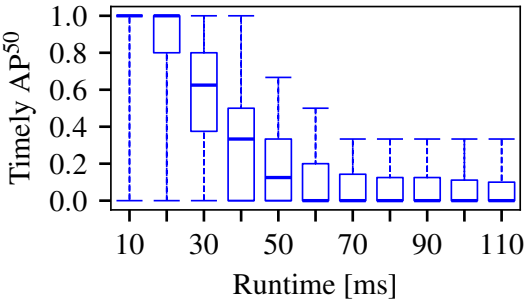
4.6 Case Studies

We now show how Pylot enables researchers to study: (*i*) the trade-off between accuracy and runtime for different components (§4.6.1), (*ii*) the effects that changes in a single component have on an end-to-end driving metric (§4.6.2), and (*iii*) the ability of Pylot to exploit remote resources in the cloud to increase driving safety (§4.6.3).

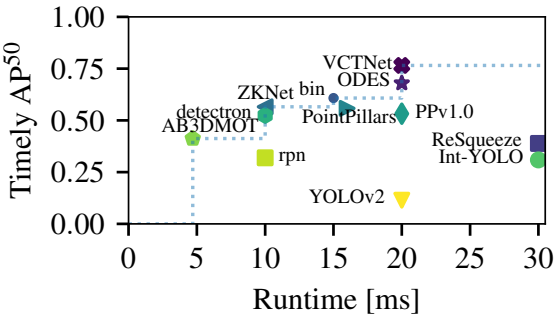
Experimental Setup. The experiments were performed atop CARLA on a machine having 2×Xeon Gold 6226 CPUs, 128GB RAM and 2×Titan-RTX 2080 GPUs. This configuration closely resembles the hardware used in the AV that Pylot was deployed to in §4.5.6.



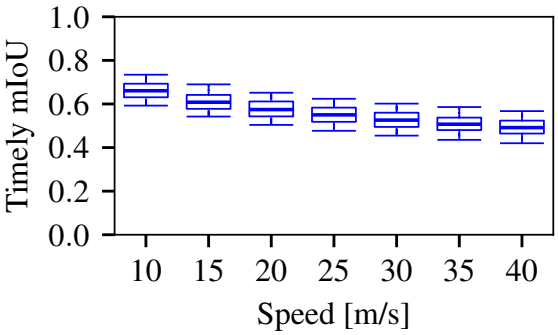
(a) Semantic segmentation timely mIoU.



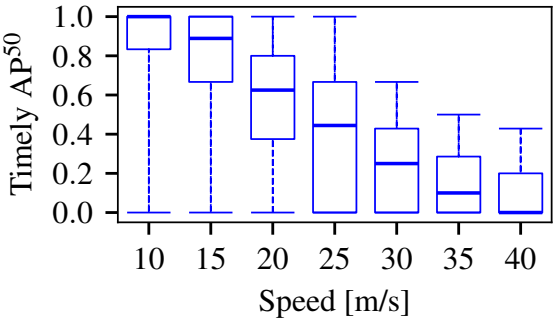
(b) Pedestrian detection timely AP^{50}.



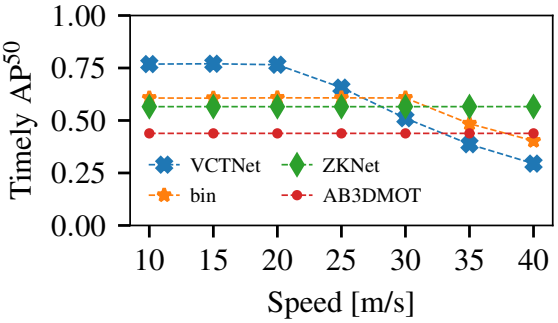
(c) Choice of best detector changes.



(d) Degradation of Segmentation mIoU



(e) Pedestrian detection timely AP^{50} degrades faster at high speeds.



(f) Different object detectors are best at different driving speeds.

Figure 4.3: Timely mIoU and AP^{50} degrade when runtime and driving speed increase. Thus, accuracy, runtime, and driving speed are all important when making model decisions (Fig. 4.3c and Fig. 4.3f).

4.6.1 Accuracy vs. Runtime Trade-off

We introduce a new family of metrics, **timely accuracies**, in order to capture the impact of runtime on accuracy of modules. The timely accuracy is a metric of how accurate a module's results are with respect to the present world, and not the past world captured by the input on which the result was based. The key idea is to evaluate a module's output performed on inputs at time t_1 with the ground truth labels at $t_2 = t_1 + l_t$, where l_t is the module's runtime. Thus, timely accuracy captures how accuracy degrades as a function of runtime in dynamic worlds.

In the experiment, we attached a camera to a simulated AV, and set up a scenario in which the AV drove in a city at 20 m/s. For each frame we collected the ground-truth semantic segmentation and computed the *timely* (i.e. time-delayed) mean Intersection over Union (mIoU) by emulating different prediction runtimes. Fig. 4.3a shows that a runtime of 10ms is sufficient to reduce mean *timely* mIoU of a perfect semantic segmentation component to approximately 0.75, which is less than the mIoUs of the top three Cityscapes submissions [77]. Similarly, we measured the tradeoff between accuracy and runtime for pedestrian detectors by computing *timely* Average Precision at IoU 50% (AP^{50}) for a perfect detection component (i.e. with 1.0 AP^{50}). Fig. 4.3b shows that mean *timely* AP^{50} halves with a runtime of 35ms. Thus, an object detector with long runtimes must be accompanied by a tracker or a prediction component that can predict accurate trajectories for longer than the detector's runtime.

The timely accuracy does not only depend on runtime, but also on the AV's driving speed. The faster an AV drives, the quicker the world changes. To show the effect of an AV's speed on timely accuracy, we emulated a 20ms runtime for the perfect semantic segmentation and detection components, and varied the driving speed. In Fig. 4.3d, we illustrate that the *timely* mIoU for semantic segmentation decays as speed increases: median *timely* mIoU is 28% smaller at 40m/s than at 10m/s. In the case of object detection, speed has a bigger effect on *timely* AP^{50}: median *timely* AP^{50} is 1.0 when driving at 10m/s, but it decreases to 0 at 40m/s (Fig. 4.3e). In Fig. 4.3f we illustrate the trade-off between model accuracy and runtime using models from the KITTI pedestrian detection challenge [111]. We observe that fast, low-accuracy detectors obtain higher timely accuracy than slow, high-accuracy detectors when driving at high speeds.

4.6.2 Effects of Component Changes on End-to-end Driving

We now leverage Pylot's *pseudo-asynchronous* execution mode to study both the effect of component changes and model hyperparameter tweaks on end-to-end driving. For this experiment, we developed a scenario which simulates the illegal crossing of the AV's lane by a person, as shown in Fig. 4.2. The scenario is further complicated by the presence of a truck on the opposite lane, which occludes the person until it reaches the AV's lane (20 meters away). This presents an imminent threat to the AV, and requires the AV to perform an emergency maneuver to avoid a collision.

In this experiment, we omit the perception models and use ground truth to compare

Planner [P_{99} runtime] \ Speed [m/s]	16	18	20	22
FOT [P_{99} = 30 ms]				
FOT [P_{99} = 550 ms]				
RRT* [P_{99} = 15 ms]				
RRT* [P_{99} = 76 ms]				
Hybrid A* [P_{99} = 25 ms]				
Hybrid A* [P_{99} = 760 ms]				

Table 4.1: Study of Runtime-Accuracy Tradeoff of Planner. Configurations that avoid a collision are marked in green, while failing configurations are marked in red.

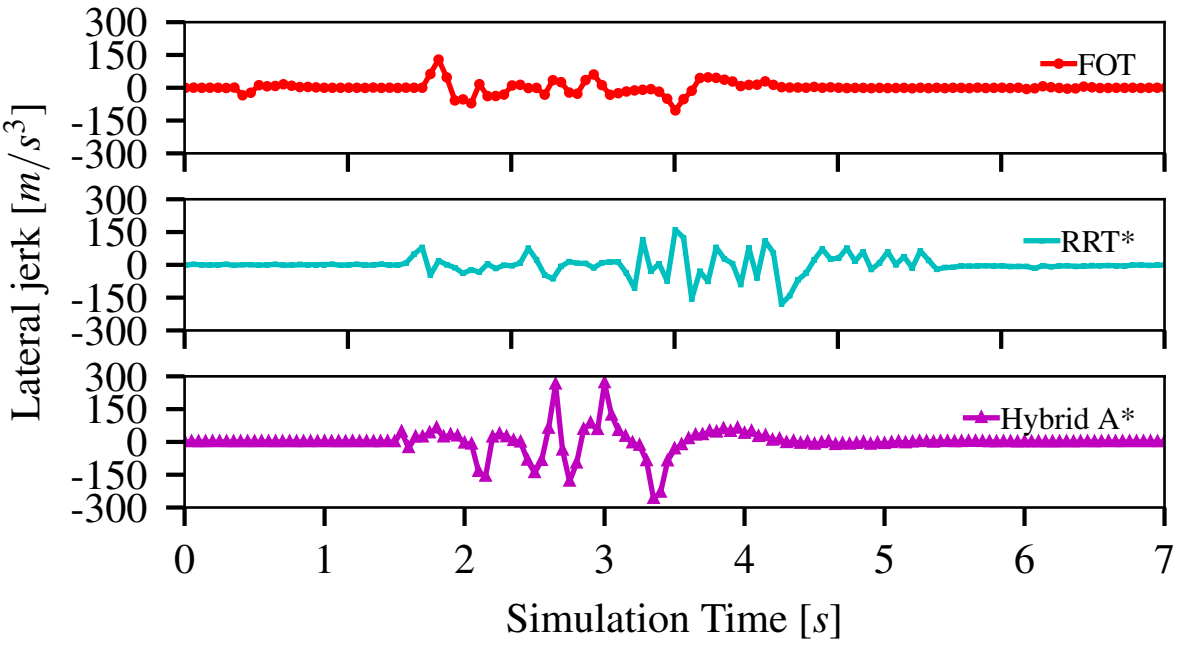


Figure 4.4: Comparison of the ride comfort offered by the planners that avoid the collision when driving at 16 m/s target speed.

each planner in isolation[3]. For the Frenet Optimal Trajectory (FOT) planner [324, 329] we executed a fast and a slow configuration with 0.3 seconds time discretization and 0.5 meters space discretization, respectively 0.1 space and time discretization. Similarly, for RRT* [169] we experimented with 0.1 and 0.5 meters step sizes Lastly, we explore Hybrid A* [95] with step sizes of 3.0 and 6.0, and radian step discretizations of 0.25 and 0.75.

In §4.6.2 we show which configurations avoided a collision with the person and the truck for different *target speeds* the AV drove at before it detected the person. Since avoiding a

[3]Visualizations are available at https://tinyurl.com/y3coq57r.

collision requires a fast, *swerving* maneuver from the AV, the planner configurations that minimized runtime performed better. Furthermore, Fig. 4.4 compares the three configurations that succeed at 16m/s target speed by plotting the lateral jerk (i.e. ride comfort) while performing the *swerving* maneuver to avoid a collision. We see that the three planners have markedly different jerk profiles. While FOT avoids a collision in fewer cases than RRT*, it provides a more comfortable ride. This experiments illustrates the realistic end-to-end "A/B testing" of AV components enabled by Pylot.

4.6.3 Leveraging Cloud Computing for Safer Driving

To drive safely, AVs must produce accurate and timely results using state-of-the-art algorithms and models that consume high-fidelity sensor data. The combination of these characteristics requires AVs to exploit the compute capabilities of cutting-edge hardware. However, the deployment of such hardware in an AV is constrained by its cooling, power, and stability requirements [186]. For example, the DRIVE platform [221], NVIDIA's flagship hardware for AVs, is updated every 3 years. Its most recent iteration, the DRIVE Orin [224], was put into production vehicles in 2023 and its successor, the DRIVE Atlan [305], is slated for release in 2026 [222]. Moreover, upgrading the hardware of previously-deployed AVs is often infeasible due to the cost and complexity involved with a recall [297]. As a result, modern AVs are forced to trade-off accuracy, and hence, safety, for computational resources and timely results, by either reducing the amount of data fused from multiple sensors, or deploying algorithms and models that require a lower number of parameters and FLOPs [117].

In light of this fundamental mismatch between the pace of development of compute technologies with the update cycles of vehicles [34], we study the augmentation of the computational resources in an AV with the compute capabilities of the cloud. Cloud computing platforms provide the illusion of infinite computing resources [104], and enable low-cost access to state-of-the-art hardware [223, 122, 37]. In contrast to the 3-year update cycle in AVs, the hardware and software in the cloud is frequently updated [36, 57]. For AVs, the cloud enables the deployment of compute-intensive, rapidly-evolving algorithms and models which can exploit state-of-the-art hardware without requiring complex recalls for hardware and software updates [296, 307].

For example, the NVIDIA DRIVE Orin platform [224], the latest revision supported by Baidu's Apollo AV [136], uses NVIDIA's Ampere microarchitecture to deliver a performance of 5.2 FP32 TFLOPs [220]. The equivalent Ampere cloud GPU is the NVIDIA A100, which was released in 2020 and delivers a performance of 19.5 FP32 TFLOPs [219], a 3.75× increase over the DRIVE Orin. Table 4.2 measures the effects of this disparity by executing open-source implementations of DETR [64, 93] and SWIN [194, 340], two state-of-the-art vision transformers which have been adapted for object detection, atop both the NVIDIA Orin and the A100. We observe a significant speedup of up to 8.4× by executing the same model on the same input on a cloud GPU. Similar trends have been shown by prior work for other safety-critical algorithms applicable to AVs, such as motion planning [68].

Conventional wisdom suggests that the latency and availability of cellular network con-

Model	Runtime [ms]		Speedup
	Orin	A100	
DETR-ResNet-50	301.7	102.2	2.95×
DETR-ResNet-101	407.7	118.2	3.45×
DETR-ResNet-101-DC	859.2	146.6	5.86×
DINO-SWIN-Tiny	722.1	90.1	8.01×
DINO-SWIN-Small	903.5	107.1	8.43×
DINO-SWIN-Large	1529.9	180.8	8.46×

Table 4.2: Runtime disparity between hardware on the AV and the cloud for various state-of-the-art object detection models.

nections makes using the cloud on the critical-path of the computation infeasible [173]. However, Table 4.2 presents an arbitrage opportunity whereby an AV could potentially return more accurate results faster by exploiting the computational power of the hardware in the cloud. We now seek to exploit the best-effort speculative execution approach enabled by ERDOS (discussed in §3.4.3) to leverage the cloud when it is available and utilize reliable fallback mechanisms that use on-board computation when the cloud is not immediately accessible. We discuss three mechanisms that enhance the AV's safety below:

Higher-Accuracy Models. AVs can selectively offload data from their input sensors to the cloud, allowing higher accuracy models to be executed in the cloud. For example, in Table 4.2, an AV can choose to execute DETR-ResNet-101-DC in the cloud on its camera data, which provides a ∼3-point increase in average precision over DETR-ResNet-50 [64] and executes faster. Thus, while an AV with an end-to-end deadline of 500 ms can only execute DETR-ResNet-50 on its local hardware, it can optimistically exploit the accuracy of all models in Table 4.2 when the latency to the cloud is low.

Accurate Environment Representation. While the earlier mechanism significantly enhances an AV's ability to process sensor data and understand its surroundings, this mechanism does not apply to obstacles obscured by the AV's blind spots. To improve safety in these scenarios, AVs can share their locations computed by the localization module to the cloud and subsequently retrieve the locations of other nearby vehicles. Integrating the locations of nearby vehicles via the cloud allows the planning module to generate safer trajectories which avoid collisions with vehicles located in blind spots.

Contingency Planning. During the course of computation, AVs must make probabilistic decisions which affect the outputs of the modules. For example, object detectors are configured with a confidence threshold to filter out misdetections from the model's outputs [93]. Similarly, the prediction module generates several possible trajectories for nearby obstacles, and ranks them based on their probability of occurring [256, 257]. The planning module then uses the most likely trajectory of the obstacles to plan a trajectory for an AV to follow.

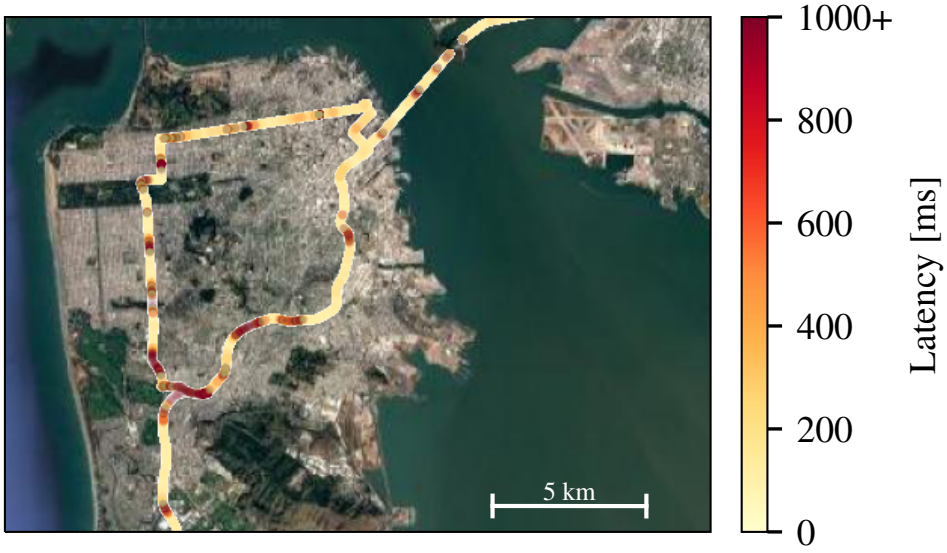


Figure 4.5: Cellular network latency of a 5G connection while driving through a route in San Francisco frequented by Waymo and Cruise AVs.

However, AVs can offload the computation of plans that handle unlikely object trajectories to the cloud. When an object takes an unlikely trajectory, AVs can access the corresponding cloud-computed plan, enabling quick reactions to any sudden changes in the environment.

We now evaluate the efficacy of ERDOS in augmenting the safety of cloud-assisted AVs under complex, real-world scenarios while maintaining the accuracy of an AV using only on-board computation. Specifically, we seek to answer the following questions:

1. Do current technologies support a low-latency and reliable connection to the cloud?
2. Does ERDOS enable the three techniques to enhance AV safety?

Feasibility of Cloud Access. We investigate whether modern cellular networks are able to provide the speed and bandwidth necessary to execute data-intensive AV operators. To measure network performance in realistic setting, we conduct a field test by following a route in San Francisco where Cruise and Waymo already provide fully autonomous rides [321, 322]. The route contains both urban and highway driving and is visualized in Fig. 4.5.

In our vehicle, we connect a Lenovo ThinkPad P1 Gen 2 laptop to an Inseego MiFi X PRO 5G hotspot on the Verizon network via USB-C. The laptop executes a multithreaded gRPC [129] client which sends 33.3 KB messages at 30 Hz to a server to match the bitrate of 30 FPS HD camera footage [333]. We establish a connection to a Google Cloud Platform `n1-highmem-8` instance in the `us-west2-a` zone which executes a gRPC server that responds to messages from the client with 1 KB acknowledgments. The client measures the round-trip latency of sending a message to receiving an reply.

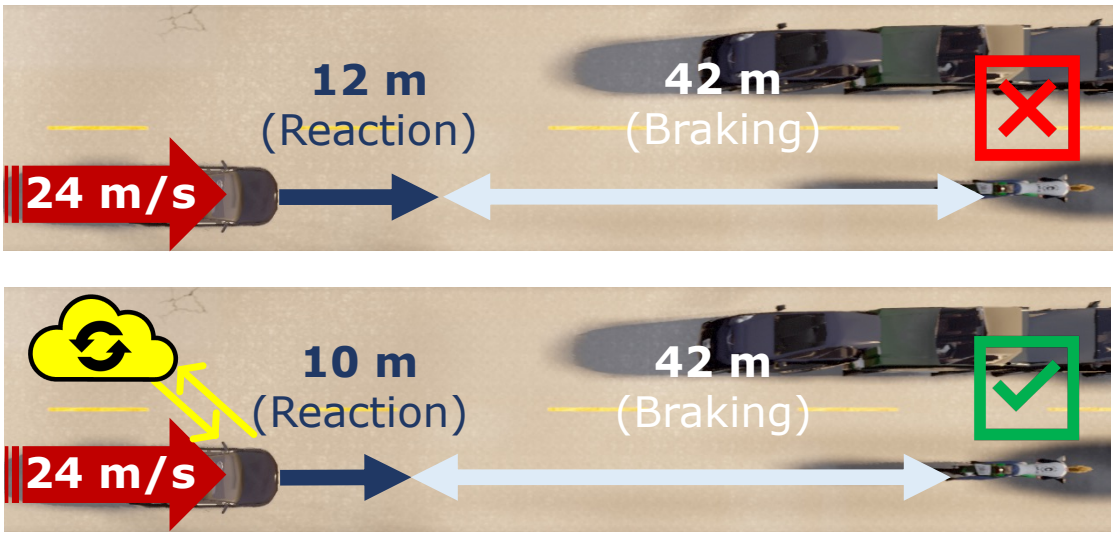


Figure 4.6: Traffic Jam Scenario leverages the cloud's ability to run *higher-accuracy models* at a reduced latency to reduce the AV's response time, thus minimizing its reaction time and avoiding a collision with the motorcycle.

We find that the 5G network in San Francisco frequently provides latencies that enable cloud execution. We measure a median round-trip-latency of 68 ms (Fig. 4.5) which demonstrates an opportunity to take advantage of the hundreds of milliseconds in runtime disparity between cloud and AV hardware (Table 4.2). In addition, the long tail of network latencies from 336 ms at the 90th percentile to 3027 ms at the 99th percentile substantiates the need to manage network delays.

Study of Complex Scenarios. We now study of the safety benefits of ERDOS and how it enables the three mechanisms which use the cloud to improve AV safety. For each mechanism, we demonstrate its efficacy under a complex, real-world scenario executed using the CARLA simulator [96]. We use the pseudo-asynchronous mode of execution from the Pylot AV platform [119] to simulate the delay of calculating a demand for different end-to-end deadlines, retrieved from different end-to-end deadlines, retrieved from Fig. 4.5[4].

Traffic Jam. This scenario from [117] simulates merging into a traffic jam, visualized in Fig. 4.6. The AV drives at a high speed on a two lane undivided road and must come to a halt behind the motorcycle stopped in the distance. The motorcycle complicates this scenario because the AV must perceive the stopped obstacle from afar to prevent a collision. Moreover, the AV cannot swerve to avoid a collision due the vehicles in the opposite lane. Since this scenario requires both far-away detections and rapid responses due to the high driving speed, the technique of exploiting the cloud to run *higher-accuracy models* quickly ensures maximum safety. To evaluate safety, we use a simple planner that brakes once the

[4]Videos of scenarios are available at https://tinyurl.com/26fzrabu

Speed [m/s]	Approach	Cloud Response Time [s]					
		0.5	0.75	1.0	1.25	1.5	3.0
11	**Local**						
	Cloud						
	Ours						
18	**Local**						
	Cloud						
	Ours						
20	**Local**						
	Cloud						
	Ours						
22	Local						
	Cloud						
	Ours						
24	Local						
	Cloud						
	Ours						

Table 4.3: Efficacy of leveraging cloud resources for Traffic Jam scenario. Configurations that avoid a collision are marked in green, while configurations that collide are marked in red.

AV's object detection operator identifies the obstacle on three consecutive camera frames. We then investigate the following three operator configurations (Table 4.3):

1. **Local** which executes DETR-ResNet-50 on a local NVIDIA Orin GPU. We choose DETR-ResNet-50 since it is the only model that provides response times required to ensure human-level safety on the local GPU (Table 4.2).

2. **Cloud** which executes DETR-ResNet-101 on an NVIDIA A100 GPU running on a Google Cloud `a2-highgpu-1g` instance. We choose DETR-ResNet-101 to ensure that the local and cloud models belong to the same architecture.

3. **Ours** which enables operators to specify deadlines on response time from the cloud and fall back to local results when the cloud is unable to meet the deadline.

We sweep the entire range of driving speeds in California (i.e., 25 mph to 65 mph), and simulate cloud response times up to the p99 latency collected from our drive through San Francisco (Fig. 4.5). We note that the higher cloud response times do not apply to the *Local* approach. We find that at higher speeds (e.g., 22 m/s), the lower-latency access to *higher-accuracy models* afforded by the cloud is critical in ensuring the safety of the AV. However, without appropriate mechanisms to fall back to local computation using ERDOS'

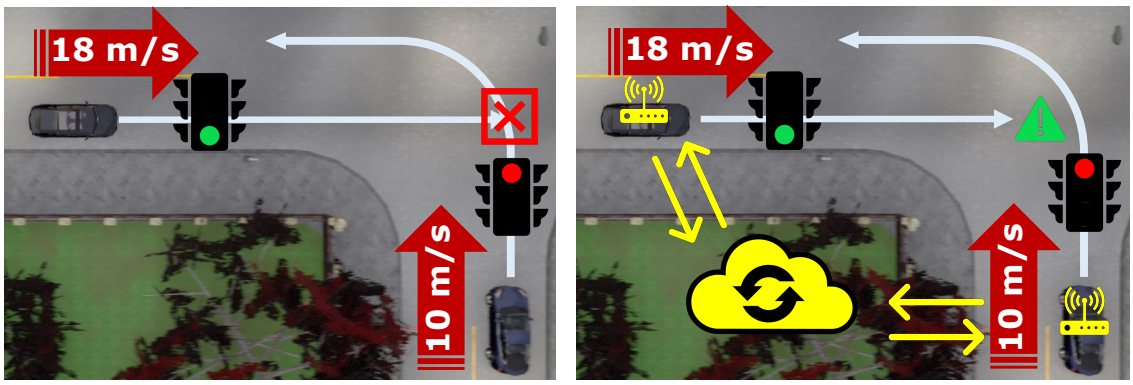


Figure 4.7: Running a Red Light Scenario leverages the cloud's ability to build *accurate environment representations* to detect the occluded vehicle running the red light.

deadlines, the *Cloud* approach incurs more safety violations than the *Local* approach when network latency is high (e.g., a 3 second cloud response time when the AV drives at 18 m/s).

Fig. 4.6 investigates how executing a higher-accuracy object detector locally affects collision-avoidance in high-speed scenarios. We compare a *Local* execution of DETR-ResNet-101 to a *Cloud* execution with median network latency. We find that the local execution fails to break in time due to its 222 ms longer response time, resulting in a collision.

Running a Red Light. This scenario from the NHTSA pre-crash typology [214, 246] (depicted in Fig. 4.7) simulates a vehicle running a red light which forces the AV to perform a collision avoidance maneuver. The AVs can only perceive the other vehicle just before a potential collision, challenging its ability to respond in time. We evaluate three variations of this scenario with different intersections, and find that neither local nor cloud executions of object detectors are able to avoid a collision. However, using the cloud's ability to build an *accurate environment representation* (e.g., by sharing location data with nearby AVs), allows the AV to plan its trajectory with other vehicles in its blind spots. This enables the AV to brake early and avoid a collision in this scenario.

Person Jaywalking. This scenario simulates a person unexpectedly entering the street, requiring the AV to quickly respond in order to avoid a collision. Fig. 4.8 evaluates the scenario in which an AV executes its planning module locally, which has a 500 ms end-to-end response time. Because the pedestrian enters the street when the AV is only 10 m away, the AV cannot generate an emergency swerving maneuver or stop in time, resulting in a collision with the pedestrian.

However, we perform *contingency planning* using the cloud which generates a plan for the low-likelihood case that the pedestrian enters the street, downloads the plan to the AV, and caches the plan in the AV's planner. When the cloud-assisted AV detects the pedestrian entering the street, the AV enacts the cached contingency plan and bypasses the local planner, lowering the response time to 400 ms. We observe that the cloud-computed contingency plan enables the AV to swerve in time and successfully avoid a collision.

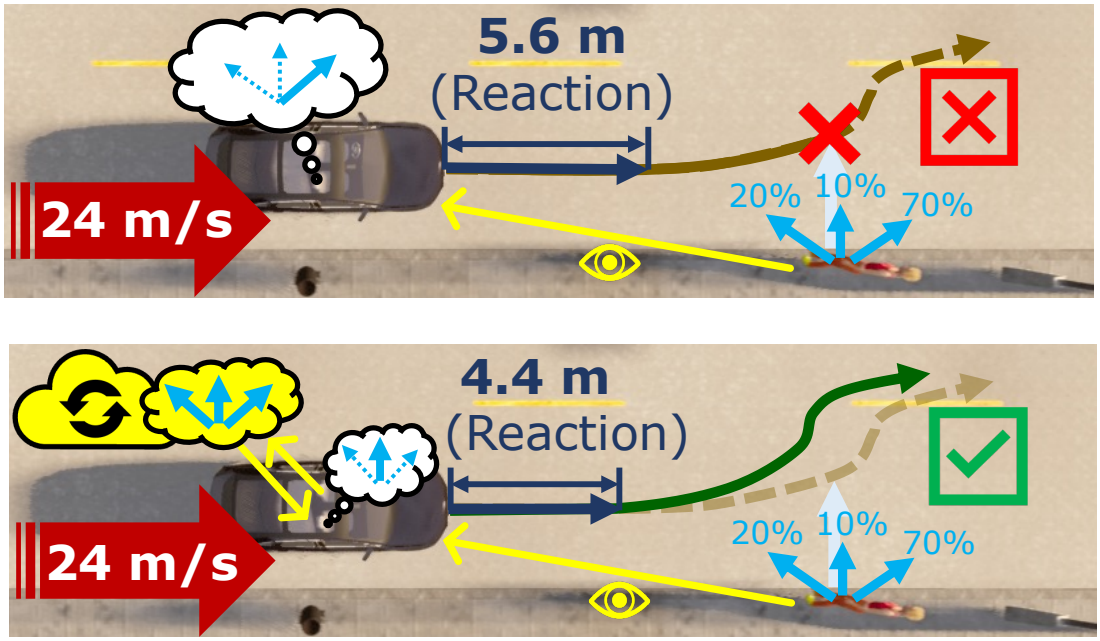


Figure 4.8: Person Jaywalking Scenario leverages the cloud's ability to do *contingency planning* for the unlikely case that the pedestrian enters the street, allowing the AV to use the cached plan quickly and avoid a collision.

4.7 Conclusion

We have presented an open-source platform for AV research. Pylot's modular and portable structure enables iterative development and evaluation of components in the context of an entire AV pipeline. This approach to AV development supports the study of interactions between modules, resulting in a better understanding of the effects of runtime and accuracy on end-to-end driving behavior.

Pylot provides reference and ground-truth implementations for several components of an AV pipeline. These state-of-the-art implementations have been used to both drive a real-vehicle and attain a high score on the CARLA Challenge. In addition, we explore various novel scenarios to enhance end-to-end driving safety of AVs using Pylot and ERDOS.

Chapter 5

SuperServe: Fine-Grained Inference Serving for ML Models

The increasing proliferation of ML models in AV's decision making components at the top of the stack coupled with the relative scarcity of resources at the hardware layer on the bottom (Fig. 1.1) requires an efficient utilization of the available resources. This tension is exacerbated by the speculative execution mechanisms enabled by D3 and ERDOS that crucially rely on being able to efficiently serve multiple ML models at low latency to enable components to meet their deadlines. However, modern inference serving systems either choose a static point in the latency-accuracy tradeoff space to serve all requests, leading to degraded accuracy under normal operating conditions, or load specific models on the critical path of request serving, leading to missed deadlines.

In this chapter, we instead resolve this tension by simultaneously serving the entire range of models spanning the latency-accuracy tradeoff space. Our novel mechanism, SubNetAct, achieves this by carefully inserting specialized control-flow operators in pre-trained, weight-shared super-networks. These operators enable SubNetAct to dynamically route a request through the network to actuate a specific model that meets the request's latency and accuracy target. Thus, SubNetAct can serve a vastly higher number of models than prior systems while requiring upto 2.6× lower memory. More crucially, SubNetAct's near-instantaneous actuation of a wide-range of models unlocks the design space of fine-grained, reactive scheduling policies. The remainder of this chapter explores the design of SuperServe, an inference serving system that instantiates SubNetAct and explores the efficacy of various scheduling policies to serve requests for ML models available in the AV.

5.1 Introduction

Recent advancements in machine learning (ML) techniques have unlocked vast improvements in both accuracy and efficiency of a wide-variety of tasks [69, 229, 18, 108, 161, 286]. As a result, ML models have been quickly deployed across a wide-range of applications in

both datacenters [343, 279, 233, 138, 228] and the edge [25, 231, 117, 187], where they are subjected to the stringent requirements of production applications.

Notably, ML models on the critical path of these applications must deal with *unpredictable request rates* that rapidly change at a sub-second granularity. For example, web applications in datacenters increasingly rely on ML models [343, 279], and have extremely bursty request rates, with a 50× higher peak demand than average [183]. Similarly, as we discussed in Chapter 2 and Chapter 3, D3 achieves major end-to-end driving safety benefits from rapidly changing request rates for ML models (as well as the ML models themselves) as a function of the terrain (city vs. freeway driving), time of the day etc. [117].

In the presence of these unpredictable request rates, ML inference serving systems that cater to production applications must strike a careful balance between three requirements:

R1: Latency. Production applications have extremely stringent latency requirements, quantified through a Service-Level Objective (SLO) [166]. For example, both web serving [279, 233, 132] in datacenters and AVs [117, 187] on the edge must maximize the number of requests completed within a specified SLO ranging from $10 - 100$ ms [117, 341].

R2: Accuracy. Production applications demand the highest-accuracy results possible within the latency targets of their requests. For example, higher accuracy has been intricately tied to a better user experience for web applications [327, 138]. Similarly, the safety of an AV heavily relies on the accuracy of its different ML models [117, 119].

R3: Resource-Efficiency. Web applications at Facebook process 200 trillion ML model requests daily [314] – a significant fraction of datacenter usage [233]. The proliferation of ML models and their reliance on expensive resources such as GPUs, TPUs [5], AWS Inferentia [38] etc. has led to resource tensions in both datacenters and the edge [231]. Thus, inference serving systems must make judicial use of these resources.

The first-generation of inference serving systems [79, 228, 80, 165, 225, 132, 341] resolve this tension by choosing a static point in the tradeoff space between **R1**-**R3** and serving all requests of an application using the chosen model. As a result, applications either miss their SLO targets (**R1**) under bursty request rates or suffer degraded accuracy (**R2**) under normal conditions. More recently, state-of-the-art inference serving systems [260, 342] enable applications to register multiple ML models spanning the entire pareto frontier of latency (**R1**) and accuracy (**R2**) targets, and *automatically* choose a model based on the incoming request rates. To achieve this, these systems must either keep the entire set of models in memory or rely on *model switching* to load the required models at runtime [342]. As GPU memory remains the key bottleneck in both datacenter and edge [183, 231], these systems choose between **R3** – effectively utilizing the resources (by incurring the prohibitive latency penalties of switching models), or **R1** – meeting SLO targets under bursty request rates.

Conventional wisdom in inference serving literature touts the "*non-negligible provisioning time* [for ML models due to switching], *which can exceed the request processing times*" as a "*key characteristic of ML workloads*", and "*rules out reactive techniques*" for responding to bursty request rates [133]. This wisdom has been widely accepted [260, 79, 132] leading to

the development of coarse-grained scheduling policies for inference serving that must account for the enormous latency penalty of switching models when reacting to bursty request rates. As a result, these coarse-grained policies typically avoid or minimize switching models by design [260], and are hence, unable to optimally navigate the tradeoff space between **R1**-**R3** under rapidly-changing, unpredictable request rates.

In this chapter, we challenge this conventional wisdom that forces a choice between **R1** and **R3**. We describe a mechanism, SubNetAct, to simultaneously serve the entire range of models spanning the latency-accuracy tradeoff space (**R1**-**R2**) in a resource-efficient manner (**R3**). At the core of our mechanism are novel control flow operators that SubNetAct carefully inserts into trained super-network [61, 264, 334] (SuperNet) neural architectures. SuperNets enable a fine-grained latency-accuracy tradeoff (**R1**-**R2**) by training a set of shared model weights for many neural networks, without duplication. Prior works [61, 264] propose efficient mechanisms for training SuperNets for both vision and NLP tasks, but require each model instance to be individually extracted for inference. This leads inference serving systems to a similar choice as before – either load all individual models or switch between them at runtime. However, SubNetAct's novel operators obviate the need to extract individual models and load them at runtime. Instead, SubNetAct dynamically routes requests within one SuperNet deployment with negligible overhead, enabling near-instantaneous *actuation* of different models. Thus, it unlocks orders of magnitude improvements in the navigation of the latency-accuracy tradeoff space (**R1**-**R2**), while substantially reducing the memory footprint (**R3**) (see §5.2).

In addition to being resource-efficient (**R3**), SubNetAct's agility in navigating the latency-accuracy tradeoff space (**R1**-**R2**) fundamentally changes the design space of scheduling policies. Instead of complex scheduling policies that must reason about future request rates in a bid to avoid paying the latency of switching ML models dynamically under bursts, SubNetAct enables the specification of simple policies that directly optimize for the key success metrics: **R1**-**R3**. While conventional wisdom deems such reactive policies infeasible, we explore one example point in this design space with a simple, yet effective policy that we call SlackFit. SlackFit is a reactive scheduling policy that exploits the near-instantaneous actuation property of SubNetAct to make fine-grained decisions about how many requests to serve in a batch, and which latency/accuracy choice to select for serving in real-time.

We summarize the contributions of this chapter as follows:

- We introduce SubNetAct (§5.3), a novel mechanism that enables a resource-efficient, fine-grained navigation of the latency-accuracy tradeoff space. SubNetAct achieves this by carefully inserting novel control flow operators that dynamically route requests through a single SuperNet.

- We unlock the design of fine-grained, reactive scheduling policies and provide a mathematical formulation of their objective (§5.4.1). We then propose SlackFit (§5.4.2), a simple, yet effective greedy heuristic scheduling policy and show how it accurately approximates the optimal objective.

- We instantiate SubNetAct and SlackFit in a real-world system, SuperServe, a real-time asynchronous model serving system with pluggable scheduling policies (§5.5).
- We extensively evalute SuperServe with both SlackFit and several state-of-the-art scheduling policies (§5.6). We find that SuperServe achieves 4.67% higher accuracy for the same SLO attainment and 2.85× higher SLO attainment for the same accuracy on the real-world Microsoft Azure Functions trace.

5.2 Motivation

We first motivate the need for a reactive, fine-grained scheduling policy for handling unpredictable, bursty request rates (§5.2.1). To meet the request SLO while maximizing the response accuracy, we motivate the use of SuperNets (§5.2.2) that enable a fine-grained exploration of the latency-accuracy tradeoff space (**R1**-**R2**) for multiple latency targets.

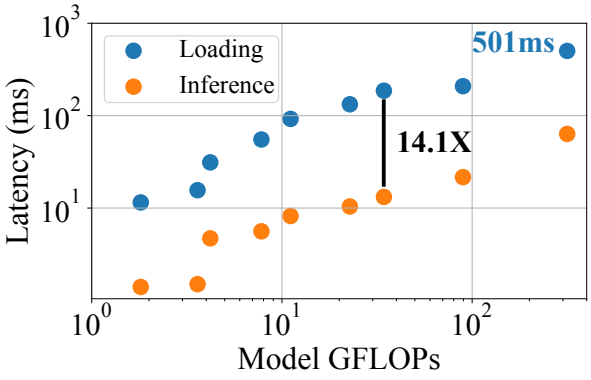
5.2.1 Reactive, Fine-Grained Scheduling

Prior works in inference serving systems [341, 183] have exhaustively analyzed both production traces from Microsoft Azure Functions (MAF) [273] and synthetic application traces with a goal of highlighting their bursty request arrival patterns. For example, Zhang et al. [341] underscore the high coefficient of variance in request arrivals in production traces [273]. Further, the authors claim that the bursty "*sub-second request arrival patterns* [are] *nearly impossible to predict*", thus frustrating the goal of meeting the stringent SLO requirements (**R1**) of requests in an ML-based production applications.

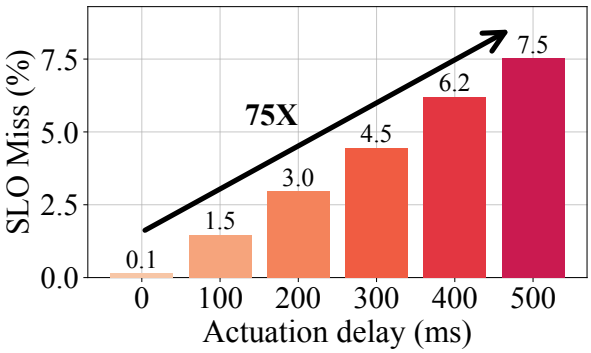
A strawman solution to meeting SLOs under bursty request rates requires these systems to load the entire set of models spanning the latency-accuracy tradeoff space into GPU memory and switch between them as request rate fluctuates. While this reduces the latency of switching models, allowing serving systems to rapidly degrade accuracy (**R2**) under bursts to meet SLO targets (**R1**), it is resource-inefficient (**R3**).

As a result, state-of-the-art inference serving systems [342, 260] page models in and out when required, to efficiently utilize GPU memory (**R3**). However, Fig. 5.1a shows that the loading time of ML models into GPU memory is vastly more than the inference time, and the gap widens as the model sizes increase. Thus, reactive approaches that either provision resources or switch models upon arrival of bursty request rates must incur an *actuation delay* (i.e., the latency penalty of loading ML models) on the critical path of request serving, leading to an order-of-magnitude increase in missed SLOs (see Fig. 5.1b). In a bid to offset this latency penalty, inference serving systems rely on *predictive* scheduling policies that make coarse-grained estimations of future request arrival patterns [133]. Such policies are bound to be suboptimal owing to the difficulty of predicting the short bursts in request arrival rates coupled with their stringent SLO requirements [341].

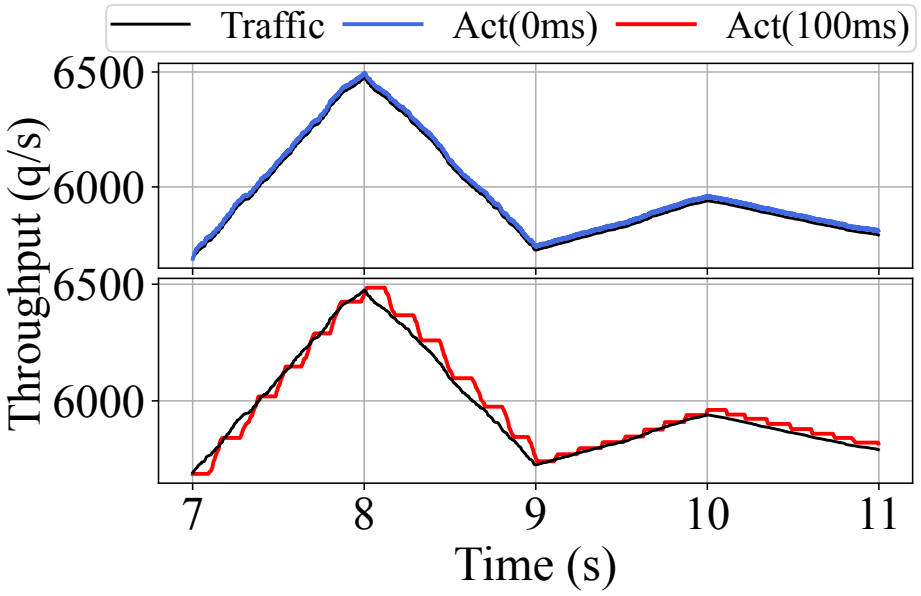
We believe that the key to optimally serving bursty request rates instead lies in the ability to rapidly switch between ML models thus obviating the need for coarse-grained predictive



(a) Model switching is expensive



(b) SLO misses wrt. actuation delay



(c) Fine-grained scheduling

Figure 5.1: Fine-grained scheduling policies are beneficial. (a) The latency of loading convolutional neural networks [139, 335, 195] and transformer-based neural networks [193] is greater than their inference latency across multiple batch sizes, making model switching expensive. This gap widens as model sizes increase, with a peak difference of up to 14.1×. (b) The higher actuation delay (due to model loading) leads to up to 75× higher SLO misses while serving the entire real-world, bursty MAF [273] trace. (c) A high actuation delay on a snapshot of the MAF trace shows 2% of the requests missing their SLO (**R1**) as request rates increase, and an inefficient utilization of resources (**R3**) as request rates decrease.

scheduling policies. To validate our hypothesis, we simulate a coarse-grained policy with an actuation delay of 100ms and an ideal fine-grained policy with an actuation delay of 0ms. Fig. 5.1c plots the effects of these policies on a small bursty subtrace from the MAF trace. We observe that the coarse-grained policy leads to higher SLO misses (**R1**) under increasing

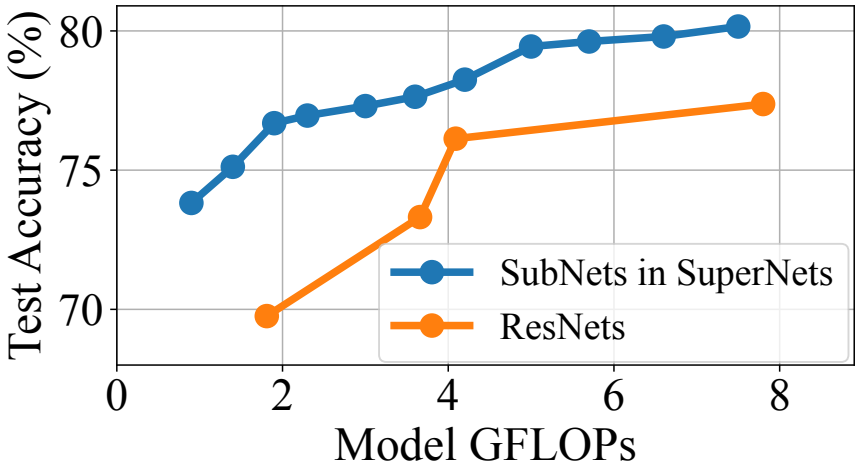


Figure 5.2: Enhanced, fine-grained latency-accuracy tradeoff with SuperNets. The accuracy of SubNets extracted from OFAResNet SuperNet [61] is vastly superior to the hand-tuned ResNets from Fig. 5.1a for the same FLOP requirements (**R1**). Moreover, SuperNets can instantiate vastly higher number of points in the space.

request rates and wasted resources (**R3**) under decreasing request rates. Conversely, the fine-grained policy is able to instantaneously adjust to the request rates leading to no missed SLOs and effective utilization of the GPU.

5.2.2 Weight-Shared SuperNets

The problem of navigating the pareto-optimal frontier of the latency-accuracy tradeoff space (**R1-R2**) by finding highest accuracy ML models for a specific latency target is well studied in ML literature. Conventional Neural Architecture Search (NAS) [349, 348, 290, 59, 190] couple the search and training of ML models to produce a single architecture with the highest accuracy for a particular latency target. To satisfy multiple latency targets, these approaches must repeat the prohibitively expensive train and search procedure.

To solve this issue, recent NAS works [334, 61, 264, 144] decouple the search and training procedures by allowing multiple architectures to share their weights while training. These approaches first train one SuperNet and then extract subsets of its layers to form multiple *SubNets*, without requiring any further retraining. These SubNets are extracted to target vastly superior points in the latency-accuracy tradeoff space (**R1-R2**), as compared to hand-tuned ML models. For example, Fig. 5.2 highlights the accuracy benefits of SubNets extracted from a ResNet-based SuperNet when compared to the hand-tuned ResNets for an equivalent number of FLOPs.

SuperNets enable a fine-grained exploration of the latency-accuracy tradeoff space (**R1-R2**) by yielding specialized ML model architectures for a wide-variety of latency targets. This is enabled by a search for multiple architectures, which relies on the following parameters: (*i*) *Depth* ($\mathbb{D}$), which describes the depth of a SubNet, (*ii*) *Expand Ratio* ($\mathbb{E}$), which describes

layer-wise ratio of output to input channels of a convolution or number of heads of a multi-head attention layer, and (*iii*) *Width Multiplier* ($\mathbb{W}$), which describes layer-wise fraction of input and output channels/heads to be used. These parameters ($\mathbb{D}$, $\mathbb{E}$, $\mathbb{W}$) create a combinatorially large architecture space, Φ ($|\Phi| \approx 10^{19}$) [61], from which individual SubNets are extracted *statically* for inference. However, this static extraction in prior work [334, 61, 264, 144] again yields individual models that must either be simultaneously deployed (wasting resources; **R3**) or paged in as request rates fluctuate (missing SLOs; **R1**).

5.3 SubNetAct: Instantaneous ML Model Actuation

Motivated by §5.2, we seek to exploit SuperNets' fine-grained exploration of the latency-accuracy tradeoff to unlock the development of reactive scheduling mechanisms. To achieve our goal, we introduce SubNetAct, which addresses the challenges posed by static extraction of individual models in SuperNets, which force a choice between **R1** and **R3**.

Key Idea. To resolve this fundamental tension, we make the key observation that by virtue of performing architectural search *post* training, a SuperNet subsumes the entire architectural space of SubNets. Specifically, we follow the training procedure of prior NAS approaches [334, 61, 264, 144] to retrieve the architecture of the SuperNet, $\mathcal{M}$, along with its weights W[1]. The architecture of the SuperNet, $\mathcal{M}$, captures the set of all layers and weights that can be used in inference. Thus, instead of performing the search procedure and extracting individual SubNets, SubNetAct automatically modifies $\mathcal{M}$ to introduce its novel control flow operators. This allows SubNetAct to deploy the entire SuperNet $\mathcal{M}$, and *dynamically route* requests to the appropriate SubNet. §5.3.1 provides an overview of SubNetAct's novel control flow operators, and §5.3.2 describes the procedure by which SubNetAct automatically inserts these operators into $\mathcal{M}$.

SubNetAct exploits the weight-sharing among the SubNets of a SuperNet to enable a memory-efficient model *actuation* mechanism (**R3**). Its fine-grained control flow operators near-instantaneously switch between SubNets in order to pick the optimal point in the latency-accuracy tradeoff space (**R1**-**R2**).

5.3.1 SubNetAct's Operators

SubNetAct's key insight lies in the introduction of the three novel operators (Fig. 5.3) that enable it to route requests to the required subnet dynamically and selectively use the trained supernet's weights. SubNetAct works on both convolution [61] and transformer-based [144] supernets. The three operators introduced in SubNetAct are as follows:

LayerSelect operates at a block level in $\mathcal{M}$, where each block is a collection of multiple layers. The LayerSelect operator enforces control flow by either passing the input activation

[1]We highlight that many NAS approaches make the trained $\mathcal{M}$ and W publicly available. Our evaluation uses these trained models, and does not require any further re-training.

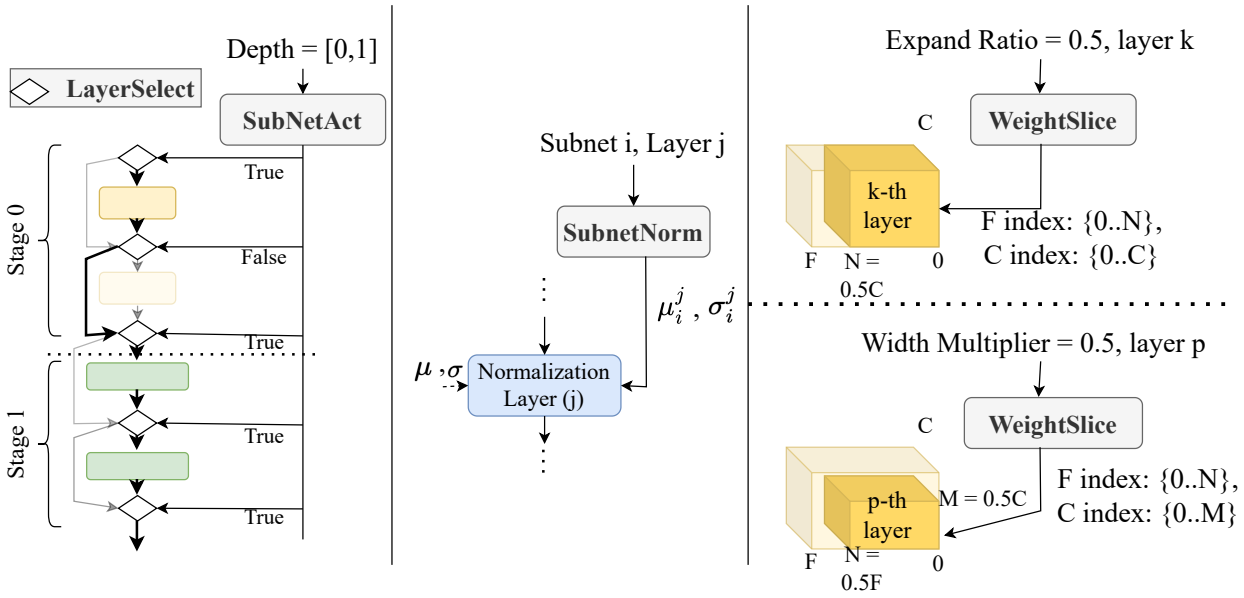


Figure 5.3: SubNetAct's novel operators (LayerSelect, SubnetNorm, WeightSlice) dynamically actuate SubNets by routing requests through weight-shared layers and non weight-shared components.

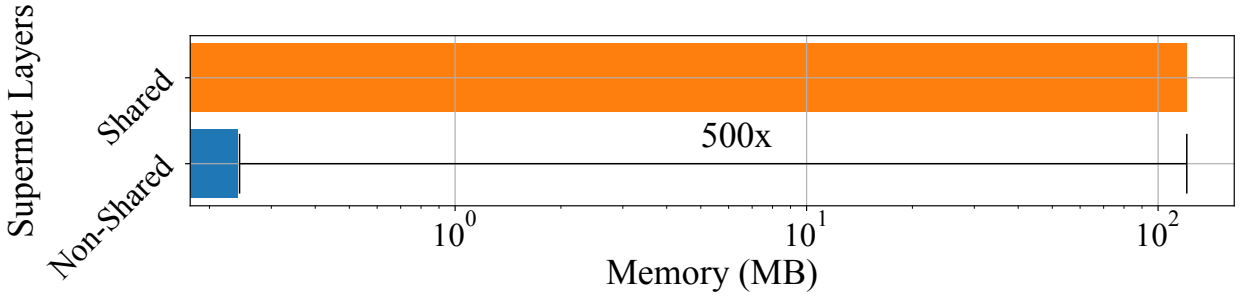
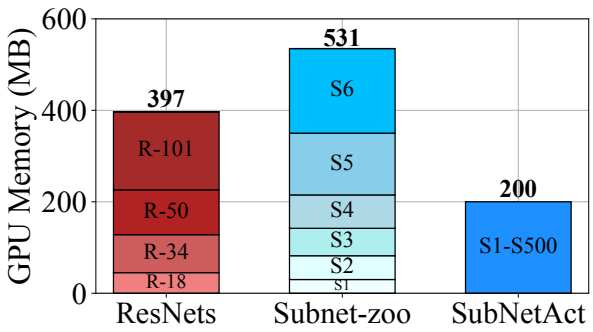


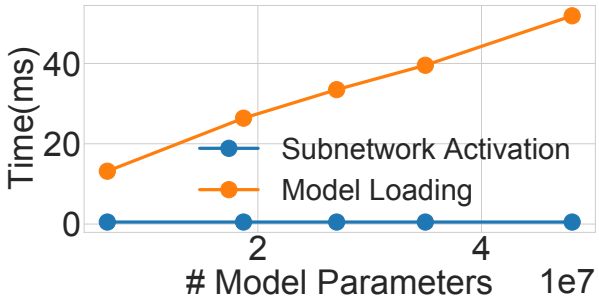
Figure 5.4: SubNetAct's memory savings. The memory used by the normalization statistics is 500× smaller than the non-normalization layers. SubnetNorm decouples the normalization statistics for each SubNet and provides accurate bookeeping thus enabling high accuracy (**R1**), with minimal increase in memory consumption (**R3**).

to the wrapped layers in the block or skipping the block and directly forwarding the input to the next block. In convolution-based SuperNets like OFAResNets [60], the LayerSelect operator (de)-selects the bottleneck blocks [139]. In transformer-based SuperNets like DynaBert [144], the LayerSelect operator (de)-selects the transformer blocks (that consist of both multi-head attention and feed-forward layers [313]).

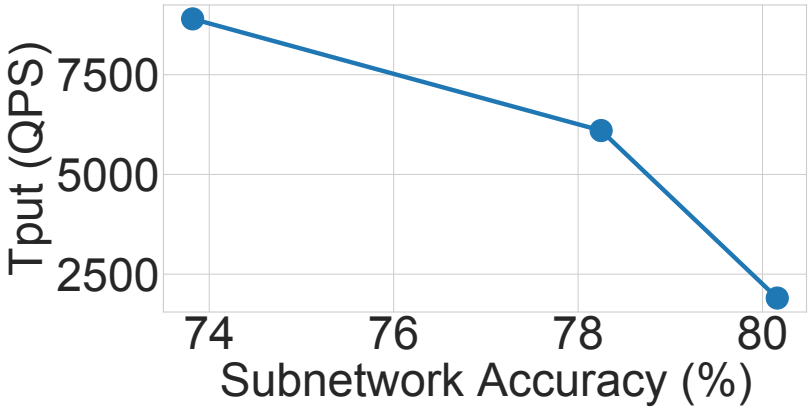
SubNetAct takes as input the depth $\mathbb{D}$, and dynamically executes blocks of $\mathcal{M}$ based on $\mathbb{D}$. Overall, this operator enables layer-sharing among subnets that differ in depth ($\Phi_{\mathbb{D}} \subset \Phi$), which reduces the GPU memory consumption (**R3**). For instance, for a SubNet with depth $\mathbb{D} = 1$, the blocks selected for execution by SubNetAct are shared with the SubNet with



(a) Reduced Memory Requirements.



(b) Instantaneous Model Actuation.



(c) High dynamic throughput range.

Figure 5.5: Efficacy of SubNetAct. (a) SubNetAct requires upto 2.6× lower memory to serve a higher-range of models when compared to the ResNets from Fig. 5.1a and six individual SubNets extracted from SuperNet [61] (b) SubNetAct actuates different SubNets near-instantaneously (< 1ms), which is orders of magnitude faster than the model switching time. (c) SubNetAct's instantaneous actuation of models enables it to sustain higher ingest rates thus inducing a wide dynamic throughput range ($\approx 2 - 8k$ queries per second) within a narrow accuracy range.

depth $\mathbb{D} = 2$. Moreover, through skipping blocks based on $\mathbb{D}$, LayerSelect enables near-instantaneous (**R1**) switching of the SuperNet's accuracy (**R2**) under bursty request rates.

WeightSlice operates at each layer in a block of $\mathcal{M}$. For each layer, it dynamically selects the slice of the SuperNet's trained weights that participate in inference. SubNetAct determines the number of channels in the Convolution or the number of heads in Multi-head Attention layer based on the expand ratio ($\mathbb{E}$) and width multiplier ($\mathbb{W}$). The operator enables partial layer-sharing among SubNets ($\{\Phi_{\mathbb{E}} \cup \Phi_{\mathbb{W}}\} \subset \Phi$), thus increasing the number of available SubNet architectures. For example, for a SubNet with $\mathbb{W} = 0.5$, WeightSlice selects the first 50% channels or heads and shares its weights with the SubNet with $\mathbb{W} = 0.75$. The

combination of LayerSelect and WeightSlice enable SubNetAct to dynamically actuate the entire set of latency-accuracy options (**R1**-**R2**).

SubnetNorm is specifically implemented for convolution-based SuperNets. We observe that naively introducing the LayerSelect or WeightSlice operator leads to a significant drop in SubNet accuracy (as low as 10%) in the convolution-based SuperNet. This is due to the incorrect tracking of the mean (μ) and variance (σ) in normalization layers such as BatchNorm [152]. The transformer-based SuperNet uses LayerNorm [39] which doesn't require tracking of mean and variances, and hence doesn't face this issue. To account for the discrepancy in BatchNorm layers, SubNetAct introduces the SubnetNorm operator that precomputes and stores μ and σ for each possible SubNet by performing forward pass inference on the training data. SubnetNorm takes as input a unique SubNet ID (i) and a layer ID (j) and outputs the pre-computed normalization statistics $\mu_{i,j}$ and $\sigma_{i,j}$. The layer j then uses the provided statistics to perform normalization of activations, effectively specializing j for each SubNet i. Although this bookkeeping increases the memory requirements of deploying the SuperNet, Fig. 5.4 shows that the overhead of these non-shared normalization statistics is 500× smaller than the memory requirement of the shared layers. SubNetAct can host thousands of SubNets in memory by only keeping the statistics unique to each subnet and sharing the non-normalization weights amongst all the SubNets.

Finally, we note that the input to these control flow operators (i.e., $\mathbb{D}$, $\mathbb{E}$, $\mathbb{W}$) remains similar to the inputs for architectural search in NAS [61]. Moreover, these inputs are independent from the request served by the actuated SubNets, and are declaratively specified by a scheduling policy (§5.4). Given the arrival rate, the scheduling policy chooses a specific SubNet for a request (by specifying the control tuple $\mathbb{D}$, $\mathbb{E}$ and $\mathbb{W}$), which is then actuated by SubNetAct near-instantaneously.

5.3.2 SubNetAct: Automatic Operator Insertion

We introduce SubNetAct's control flow operators automatically. Specifically, the LayerSelect operator is introduced at every stage of $\mathcal{M}$, and each block within the stage (such as Bottleneck in OFAResNets [61] or TransformerBlock in Dynabert [144]) is converted to a boolean module whose boolean handle is tracked by the LayerSelect operator. Each convolution or attention layer of $\mathcal{M}$ is modified by wrapping it with the WeightSlice operator. Finally, all the batchnorm layers in $\mathcal{M}$ are converted to the SubnetNorm operator with additional information provided to it about each SubNet's tracked statistics. The algorithm to automatically enable control flow operations in M is provided in Algorithm 1.

5.3.3 Discussion: Efficacy of SubNetAct

We now highlight SubNetAct's efficacy in achieving key application requirements (**R1**-**R3**) under bursty request rates.

Reduced Memory Requirements. SubNetAct's novel operators enable SubNets to share

Input: Supernet Arch. $\mathcal{M}$, Supernet Weights W, Tracked Mean and Variances $TrackedStats$

```
newOperators = {}
for s ∈ STAGES(M) do
    // layerSelect operator selects layers within each stage
    ls = LayerSelect()
    newOperators[s] = {}
    for m ∈ GetModules(M, s) do
        if m.type == Bottleneck || m.type == TransformerLayer then
            bool select_m // boolean switch for layer
            m_new = toBoolModule(m, select_m)
            // layerSelect controls boolean of stage's layers
            ls.registerBool(select_m)
        end
        else if m.type == Attention // m.type == Conv then
            // WeightSlice applied to attn or conv layers
            m_new = WeightSlice(m.type, W[m.id])
            newOperators[s][m.id] = m_new
        end
        else if m.type == BatchNorm then
            // SubnetNorm only applied to BatchNorm
            m_new=SubnetNorm(W[m.id], TrackedStats)
            newOperators[s][m.id] = m_new
        end
        ModifyModule(M, m, m_new)
        newOperators[s]["layerSelect"] = ls
    end
end
RegisterControlFlowOps(M, newOperators)
```

Algorithm 1: Introducing SubNetAct Operators in Supernets. The algorithm introduces control-flow operates to enable SubNetAct for latency/accuracy navigation. The pre-requisites to enable SubNetAct are trained weights and architecture of the supernet that are obtained from existing NAS approaches [334, 61, 264, 144].

layers in place and dynamically route requests to the appropriate SubNet based on the control tuple $(\mathbb{D}, \mathbb{E}, \mathbb{W})$. Thus, SubNetAct can simultaneously serve the entire range of models spanning the latency-accuracy tradeoff space while drastically reducing memory requirements. Fig. 5.5a demonstrates the requirements of serving the same accuracy range by comparing: (i) four different hand-tuned ResNets [139] (publicly available from prior literature), (ii) six uniformly sampled individual SubNets [61] from a ResNet-based SuperNet, and (iii) SubNetAct that enables dynamic actuation of 500 SubNets. We highlight that SubNetAct reduces

memory usage by up to 2.6×, while serving vastly more latency-accuracy tradeoff points.

Near-Instantaneous Model Actuation. While switching between individual models requires loading their weights to the GPU, SubNetAct's operators enable scheduling policies to actuate any SubNet *in place* without incurring additional loading overhead. Fig. 5.5b compares the time taken to perform on-demand loading of individual SubNets versus in-place actuation of a SubNet in SubNetAct. We highlight that SubNetAct's model actuation is orders-of-magnitude faster than on-demand loading of ML models. This allows scheduling policies that use SubNetAct to rapidly actuate lower-accuracy models under bursty conditions (**R1**) and switch to higher-accuracy models under normal load (**R2**), without coarse-grained predictions about future request rates.

Increased Throughput & Accuracy. Through its instant model actuation, SubNetAct allows scheduling policies to rapidly scale the throughput of the system, thus inducing a broad throughput range within a narrow range of accuracy to help meet SLOs (**R1**-**R2**). Fig. 5.5c compares the maximum sustained ingest throughput for a point-based open-loop arrival curve for serving the largest, smallest, and a median SubNet on 8 GPUs. We observe that SubNetAct can serve a wide throughput range from 2000-8000 QPS, while being able to increase accuracy between 74% to 80%.

5.4 Fine-Grained Scheduling Policies

SubNetAct's resource-efficient (**R3**) near-instantaneous actuation of the entire latency-accuracy tradeoff space unlocks the development of fine-grained, reactive scheduling policies. These policies can quickly scale an inference serving system's throughput upon arrival of bursty request rates to ensure that the requests meet their SLO (**R1**) with the maximum possible accuracy (**R2**). Specifically, upon a *query*'s[2] arrival to an inference serving system, it invokes the scheduling policy, which must decide the following four decision variables:

- **SubNet** ϕ from the set of all possible SubNets Φ available for actuation by SubNetAct. As discussed in §5.3, a $\phi \in \Phi$ is uniquely identified by the control tuple ($\mathbb{D}$, $\mathbb{E}$, $\mathbb{W}$).
- **Batch** B of size $|B|$ which groups the queries that are executed together on a GPU using the SubNet ϕ.
- **GPU** n upon which the batch B is executed.
- **Time** t at which the batch B must be executed on GPU n.

In this section, we first start with the mathematical formulation of an optimal scheduling policy that decides the above parameters (§5.4.1). We then describe our proposed policy SlackFit that approximates the optimal policy and aims to achieve both high accuracy and SLO attainment (§5.4.2).

[2]We use the term *query* and *request* interchangeably.

5.4.1 Optimal Scheduling Policy

We formulate an optimal scheduling policy with an oracular knowledge about all queries as a Zero-One Integer Linear Program (ZILP). The policy's decision is captured by the variable $I(\phi, B, n, t) \in \{0, 1\}$, which represents the decision to execute all queries $q \in B$ (from the set of all possible batches $\mathcal{B}$) on GPU n at time t with the SubNet ϕ. The SubNet ϕ has an accuracy $Acc(\phi)$ and a latency profile $l_\phi(|B|)$, which is the latency of ϕ on the batch size $|B|$. We use $a(B)$ to refer to the earliest arrival time of all the queries $q \in B$, and $d(B)$ to refer to the earliest deadline. Intuitively, the policy's goal is to maximize the accuracy of the responses (to queries) within their SLO (**R1**-**R2**). This is represented in ZILP as follows:

$$\text{maximize} \sum_{t} \sum_{n} \sum_{\phi \in \Phi} \sum_{B \in \mathcal{B}} \text{Acc}(\phi) \cdot |B| \cdot I(\phi, B, n, t) \tag{5.1}$$

$$\text{s.t.} \sum_{t} \sum_{n} \sum_{\phi \in \Phi} \sum_{\{B | q \in \mathcal{B}\}} I(\phi, B, n, t) \leq 1, \qquad \forall q \tag{5.1a}$$

$$\sum_{B \in \mathcal{B}} \sum_{\{t' \leq t \leq t' + l_\phi(|B|)\}} I(\phi, B, n, t^{'}) \leq 1, \qquad \forall n, t, \phi \tag{5.1b}$$

$$a(B) \cdot I(\phi, B, n, t) \leq t, \qquad \forall n, t, B, \phi \tag{5.1c}$$

$$\sum_{\phi \in \Phi} I(\phi, B, n, t) \leq 1, \qquad \forall n, t, B \tag{5.1d}$$

$$\sum_{\phi \in \Phi} (l_\phi(|B|) + t) \cdot I(\phi, B, n, t) \leq d(B), \qquad \forall n, t, B \tag{5.1e}$$

$$I(\phi, B, n, t) \in \{0, 1\}, \qquad \forall n, t, B, \phi \tag{5.1f}$$

The ZILP maximizes the number of queries that satisfy their latency SLOs with the highest possible accuracy across all the selected query batches i.e., $\forall(\phi, B) : I(\phi, B, n, t) = 1$, $Acc(\phi) \cdot |B|$ is maximized. The constraints of the ZILP denote:

(1a) A query q can be assigned to at most one batch B.

(1b) A GPU n can only execute a single SubNet ϕ on a single batch B at a time t.

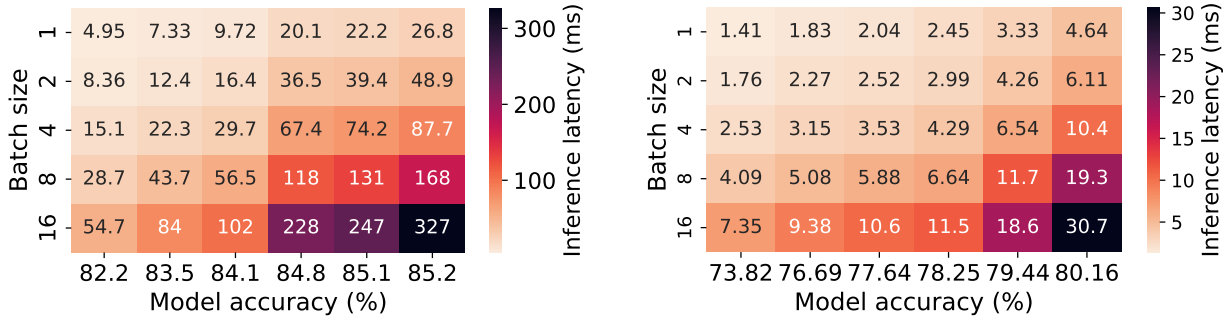
(1c) Batch B can only execute after its arrival time $a(B)$.

(1d) Batch B can be served with a maximum of one SubNet ϕ on a GPU n at a time t.

(1e) The batch B should complete before its deadline $d(B)$.

(1f) The choice variable $I(\phi, B, n, t)$ is a boolean indicator.

Given that solving the above ZILP is NP-Hard [232, 341] and it is impractical to expect oracular query arrival knowledge, it cannot be used to serve models online. Instead, we approximate its behavior with a heuristic, *online* scheduling policy.



(a) Transformer-Based SuperNet **(b)** Convolution-Based SuperNet

Figure 5.6: Latencies of SlackFit's Control Parameter Space. Latencies for six different (uniformly sampled wrt. FLOPs) pareto-optimal SubNets in SubNetAct as a function of accuracy (x-axis) and batch size (y-axis) shown for transformer and convolution-based SuperNet. The latency increases monotonically with batch size (**P1**) and accuracy (**P2**).

5.4.2 SlackFit: Online Scheduling Policy

We introduce SlackFit—a simple yet effective scheduling policy that aims to maximize the accuracy (**R1**) with which the requests meet their SLO (**R2**). SlackFit approximates the ILP-based policy in Eq. (5.1) and enables tractable decision making through two phases:

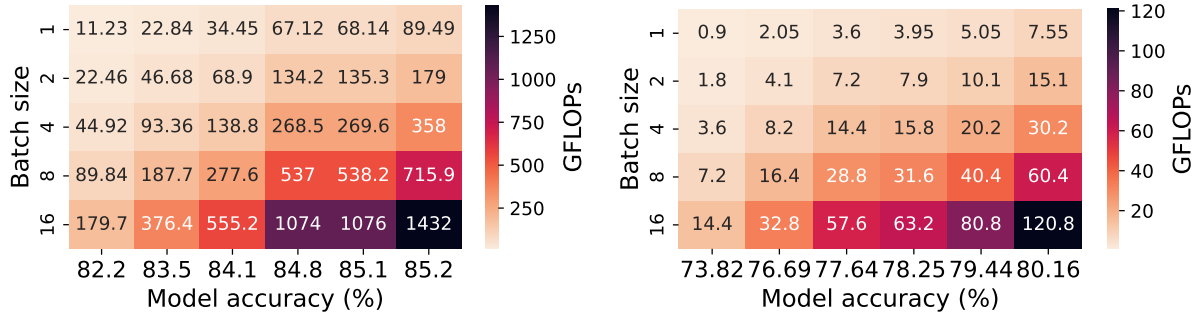
Offline Phase triggered upon the registration of a SuperNet $\mathcal{M}$ modified by SubNetAct (§5.3.2) that the inference serving system must serve. This phase consists of two stages:

1. **Profile pareto-optimal SubNets Φ_{pareto}:** To make SubNet choices in reasonable time, SlackFit makes the design decision to operate on Φ_{pareto} instead of Φ. Φ_{pareto} is the set of pareto-optimal SubNets with respect to latency and accuracy obtained by using the search stage of prior NAS methods [61][3]. The size of $|\Phi_{\text{pareto}}| \approx 10^3$ is orders of magnitude smaller than $|\Phi| \approx 10^{19}$, contributing to rapid scheduling decisions.

2. **Bucketize SubNet ϕ and batch size $|B|$ choices:** As discussed in §5.4, SlackFit must decide the SubNet $\phi \in \Phi_{\text{pareto}}$ and the batch size $|B|$ for the incoming queries. To reduce the search space for this decision, SlackFit relies on three key properties of SubNets in Φ_{pareto} (visualized in Fig. 5.6): **P1**: the latency increases monotonically with batch size as observed by prior works [79, 132, 80], **P2**: the latency increases monotonically with accuracy due to the choice of SubNets retrieved by NAS in Φ_{pareto}, and **P3**: lower accuracy SubNets can serve higher batch sizes at similar latencies to lower batch sizes in higher accuracy SubNets, due to the SubNet's FLOPs distribution shown in Fig. 5.7.

   These properties enable SlackFit to reduce the search space of choices for ϕ and $|B|$ to a single dimension – *batch latency*. Thus, SlackFit constructs evenly-spaced buckets between the minimum and maximum latency of all SubNets in Φ_{pareto} (i.e., $l_{\phi_{min}}(|B| = 1)$

[3]It takes $\leq$ 2 minutes to perform this NAS profiling on SuperNets.



(a) Transformer-Based SuperNet **(b)** Convolution-Based SuperNet

Figure 5.7: FLOPs for SlackFit's Control Parameter Space. FLOPs for six different pareto-optimal SubNets in SubNetAct as a function of accuracy (x-axis) and batch size (y-axis) shown for both transformer and convolution-based supernet. The FLOPs are *monotonic* with batch size and accuracy. This trend in FLOPs forms the analytical basis of the trend in the inference latency of these models (as shown in Fig. 5.6).

and $l_{\phi_{max}}(|B| = 16)$ respectively, where ϕ_{min} and ϕ_{max} are the lowest and highest accuracy SubNets; using properties **P1**-**P2**). Within each bucket, SlackFit chooses the (ϕ', $|B'|$) with the highest $|B'|$ such that $l_\phi(|B'|)$ is less than the bucket's latency. By **P3**, low latency buckets contain lower accuracy ϕ, higher $|B|$ (leading to higher throughput), and higher latency buckets contain higher accuracy ϕ and lower $|B|$ (leading to lower throughput).

Online Phase of SlackFit is triggered upon the arrival of queries or the availability of a GPU n to the serving system. The key insight of the online phase is that the remaining slack of the query with the earliest deadline provides a proxy to changes in the traffic. Specifically, bursts in traffic increase queuing delays which reduces the available slack, while slack remains high under normal conditions.

Thus, SlackFit chooses a bucket $(\phi, |B|)$, where $l_\phi(|B|)$ is closest to but less than the slack of the query with the earliest deadline. It then packs $|B|$ queries with the earliest deadline into a batch B and executes it on an available GPU n at time t.

By making decisions based on the minimum remaining slack, SlackFit can automatically adjust accuracy (**R2**) and throughput of the system by choosing appropriate latency bucket on variable arrival traffic to maintain high SLO attainment (**R1**). Under normal conditions, a higher slack leads to the choice of buckets with higher $l_\phi(|B|)$, which is strongly correlated with the choice of higher accuracy models (**P2**). Conversely, bursty request arrivals lead to buckets with lower $l_\phi(|B|)$, as SlackFit operates under reduced latency slack. These buckets maximize $|B|$ (due to **P3**), thus opportunistically maximizing accuracy while satisfying SLO.

SlackFit's Approximation of Optimal Offline ZILP

We now provide insights on how SlackFit emulates behavior of the optimal offline ZILP. To understand the behavior of ZILP, we formulate a proxy utility function that captures the inner-term of the ZILP objective function in Eq 5.1, the utility function is defined for a SubNet ϕ, batch size $|B|$ and the earliest deadline d_B among all queries:

$$\mathbb{U}(\phi, |B|, d_B) = \begin{cases} \text{Acc}(\phi) \cdot |B|, & \text{if } l_\phi(|B|) < d_B \\ 0, & \text{otherwise} \end{cases} \tag{5.2}$$

This utility is non-zero iff SubNet ϕ performs inference on batch size $|B|$ within the deadline d_B, and is zero otherwise. This maximizes both the number of queries processed within their SLO (**R1**) and the accuracy of their responses (**R2**).

A. ZILP and SlackFit prefer pareto-optimal SubNets. SlackFit's key design choice is to operate on pareto-optimal SubNets with respect to latency, accuracy (Φ_{pareto}) (§5.4.2). We claim that the ZILP also tends towards pareto-optimal SubNets (with respect to latency, accuracy), as these SubNets yield higher utility.

Lemma 5.4.1 *The utility of pareto-optimal SubNets is higher than non pareto-optimal SubNets if they have similar inference latency for a batch of queries.*

$$\mathbb{U}(\phi_p, |B|, d_B) > \mathbb{U}(\phi_q, |B|, d_B), \qquad \forall B, d_B$$
$$s.t.\ \ \phi_p \in \Phi_{pareto},\ \phi_q \in \{\Phi \setminus \Phi_{pareto}\},\ l_{\phi_p}(|B|) \approx l_{\phi_q}(|B|)$$

Proof By Contradiction. Assume a non-pareto optimal SubNet (ϕ_q) such that it has higher utility than pareto optimal SubNet (ϕ_p) for a batch B and $l_{\phi_p}(B) \approx l_{\phi_q}(B)$ i.e., $\mathbb{U}(\phi_p, B, d_B) < \mathbb{U}(\phi_q, B, d_B)$.

Now, due to the pareto optimal property $Acc(\phi_p) > Acc(\phi_q)$, this implies $Acc(\phi_p) \cdot |B| > Acc(\phi_q) \cdot |B|$ which implies $\mathbb{U}(\phi_p, B, d_B) \geq \mathbb{U}(\phi_q, B, d_B)$ for any delay d_B as $l_{\phi_p}(B) \approx l_{\phi_q}(B)$. This is contradiction. Hence Proved.

This validates SlackFit's design choice to operate on pareto-optimal SubNets only.

B. ZILP and SlackFit prioritize lower accuracy & higher batch sizes under bursts. We make a key observation that the utility of a lower accuracy, higher batch size ($\phi_{low}, |B_{high}|$) configuration is higher than a higher accuracy, lower batch size ($\phi_{high}, |B_{low}|$) configuration in Φ_{pareto}. This is because the factor difference in accuracy of SubNets in Φ_{pareto} (< 1) is less than the factor differences of batch sizes as seen in Fig. 5.6 i.e., $\frac{\text{Acc}(\phi_{high})}{\text{Acc}(\phi_{low})} \leq \frac{|B_{high}|}{|B_{low}|} \Rightarrow \text{Acc}(\phi_{high}) \cdot |B_{low}| \leq \text{Acc}(\phi_{low}) \cdot |B_{high}|$. Therefore, $\mathbb{U}(\phi_{low}, |B_{high}|, d_q) \geq \mathbb{U}(\phi_{high}, |B_{low}|, d_q)$ may hold true under bursts, in cases where the query q with the earliest deadline in a batch of k queries ($q \in B_k$) can be served either by: a) low accuracy model (ϕ_{min}) with batch size $|B_k|$ or b) higher accuracy model (ϕ_{max}) on a subset of queries (say m, $q \in B_m$) with remaining queries ($B_k \setminus B_m$) missing the deadline due to high load. In such cases, the optimal offline ILP will tend to option (a), similar to SlackFit.

C. ZILP and SlackFit prefer higher accuracy under normal conditions. We make another observation from the latency profiles of SubNets from Φ_{pareto} in Fig. 5.6. For a batch size $|B|$, such that $|B| = |B_1| + |B_2|$ where $|B_1| > |B_2|$, the following holds true in many cases - $B_1 \cdot \text{Acc}(\phi_{high}) + B_2 \cdot \text{Acc}(\phi_{low}) > B \cdot \text{Acc}(\phi_{mid})$. Therefore, $\mathbb{U}(\phi_{high}, B_1, d_q) + \mathbb{U}(\phi_{low}, B_2, d(B_2)) \geq \mathbb{U}(\phi_{mid}, B, d_q)$, may hold true under low load, where the query q in batch B can be served by either: a) mid accuracy model (ϕ_{mid}) with batch size B, or b) high accuracy model (ϕ_{high}) with larger partition B_1 ($q \in B_1$) with rest of the queries in batch B_2 served with the low accuracy model (ϕ_{low}) and meeting deadline $d(B_2)$. In such cases, ILP will tend to option (b) i.e., an option with higher average accuracy, similar to SlackFit (as described in §5.4.2).

5.5 SuperServe: System Implementation

SuperServe is a system that instantiates both the SubNetAct mechanism and the SlackFit policy. SuperServe's architecture is illustrated in Fig. 5.8. Clients first register the SuperNet that they want SuperServe to serve, which invokes SubNetAct to automatically insert the control flow operators for dynamic actuation of SubNets (§5.3.2). SuperServe then profiles the SuperNet to enable SubNetAct to operate on the pareto-optimal SubNets.

SuperNet Profiler. The profiler employs neural architecture search (NAS) [61] to find pareto-optimal SubNets from the SuperNet for each latency target. The latency of each SubNet is a function of the batch size and the environment of execution (i.e., the GPUs on the available workers). We emphasize that the NAS and the model profiling is efficient, taking ≤ 2 minutes to complete, and providing significant benefits for the online phase of SlackFit. Moreover, state-of-the-art systems perform similar model profiling for non SuperNet models (e.g., ResNets [139], Wide-ResNets [335], ConvNeXt [195] etc.).

Post SuperNet registration, the clients submit queries to the SuperServe router with a deadline via RPC asynchronously. These queries are enqueued to a global earliest-deadline-first (EDF) queue (❶). As soon as any worker becomes available, SuperServe's fine-grained scheduler is invoked (❷). It decides on the query-batch (B) and the subnet (ϕ) which are then dispatched to the worker (❸). Upon receiving this query-batch, the worker that instantiates the supernet instantaneously actuates the chosen subnet in-place on the GPU using SubNetAct (❹), performs inference (❺), and returns predictions for the query-batch (❻). The router redirects these predictions back to the client (❼). We discuss the critical components of SuperServe below:

Router. The router runs the fine-grained scheduling policy and interacts with workers via RPCs. All queries are received, enqueued, and dequeued asynchronously in the router. It maintains pending queries in a global EDF queue, ordered by query deadlines (SLOs). The router invokes the scheduler whenever (a) a worker signals availability and (b) the EDF queue is not empty. It sends query-batches decided by the scheduler to workers and returns the predictions to the clients.

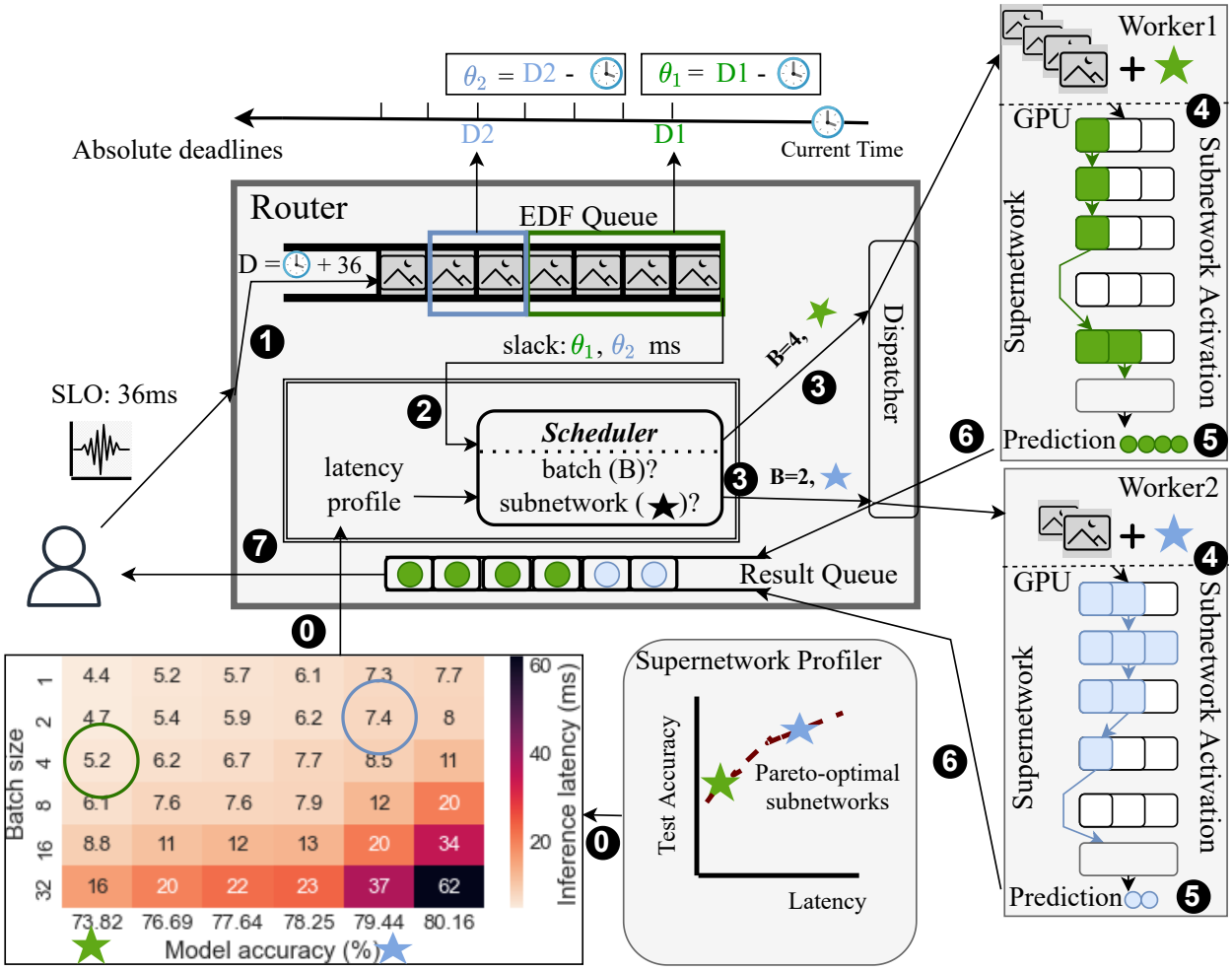


Figure 5.8: SuperServe's Architecture comprises of a SuperNet profiler, a router, a fine-grained scheduler (SubNetAct), and GPU-enabled workers. Clients register SuperNets for ERDOS to serve, whose profiling and insertion of control-flow operators is done before queries arrive. Clients submit queries to the router with a specified SLO asynchronously. The query follows the critical path ❶ - ❼.

Fine-Grained Scheduler. The scheduler's control decision is a batch-size and subnet ($\phi = (\mathbb{D}, \mathbb{E}, \mathbb{W})$). The scheduler provide pluggable APIs for different policy implementations. SlackFit is one such policy implemented in the scheduler. All policies in scheduler leverage two key properties to make control decisions: (a) predictability of DNN inference latency, (b) fast actuation of SubNetAct on the query's critical path.

Worker. The DNN worker employs the SubNetAct mechanism to host a SuperNet (**R3**). SubNetAct's operators are implemented in TorchScript's intermediate representation (IR) [306]. After receiving a query-batch and SubNet ($\mathbb{D}$, $\mathbb{E}$, $\mathbb{W}$) from the router, the worker

actuates the desired SubNet using SubNetAct. A forward pass on the actuated SubNet produces predictions that are returned to the router. The router is notified about worker availability on receiving the predictions.

5.6 Evaluation

We assess SuperServe's end-to-end performance i.e., its ability to maximize SLO attainment (**R1**) and accuracy (**R2**) under a variety of traffic conditions, including synthetic traces (§5.6.2) and a real-world Microsoft Azure Functions trace (§5.6.3). SuperServe is resource-efficient (**R3**) due to the use of SubNetAct mechanism, already established in §5.3.3. We conclude with microbenchmarks (§5.6.4) that show SuperServe linearly scaling to 33,000 qps and providing transparent fault tolerance.

5.6.1 Experimental Setup

Success Metrics. *SLO attainment* is defined as the fraction of the queries that complete within the latency deadline (**R1**). The *mean serving accuracy* is calculated for the queries that satisfy the SLO and is the average of models' profiled accuracy that were used to serve the queries (**R2**).

Traces. We evaluate SuperServe on three sets of traces: bursty, time-varying, and real-world. Bursty and time-varying traces are synthetic, similar to those used in InferLine [80]. We construct the *bursty traces* by starting with a base arrival with mean ingest rate λ_b (with $CV^2 = 0$) and add a variant arrival trace with mean ingest rate λ_v drawing inter-arrival times from a gamma distribution (Fig. 5.11a). We vary λ_b, λ_v and CV^2. *Time-varying* traces differ from bursty by varying the mean ingest throughput over time. We change the mean from $\mu = 1/\lambda_1$ to $\mu = 1/\lambda_2$ at rate τ q/s^2 with a fixed CV_a^2. Higher ingest acceleration τ q/s^2 corresponds to faster change from λ_1 to λ_2. All synthetic trace generation is seeded. Lastly, we use a MAF trace [273] for evaluation on a real-world workload.

Baselines. We compare SuperServe with the single model serving systems that don't perform accuracy trade-offs (and the models are manually selected by users, non-automated serving systems in §5.7). These systems are represented as Clipper$^+$ baseline and include systems like Clipper [79], Clockwork [132], and TF-serving [228]. Clipper$^+$ is manually configured to serve six different accuracy points (SubNets) that uniformly span the SuperNet's accuracy range and result in its six different versions. We also compare SuperServe with INFaaS and note that INFaaS is designed to "pick the most cost-efficient model that meets the [specified] accuracy constraint" [105, 260, 259]. However, in the presence of unpredictable, bursty request rates, the choice of the model accuracy to serve in order to meet the SLO requirements is unknown. Since, unlike SuperServe, INFaaS does not automatically discover the accuracy of the model to serve under unpredictable request rates and instead requires queries to be hand-annotated with accuracy thresholds, we choose to run INFaaS with no

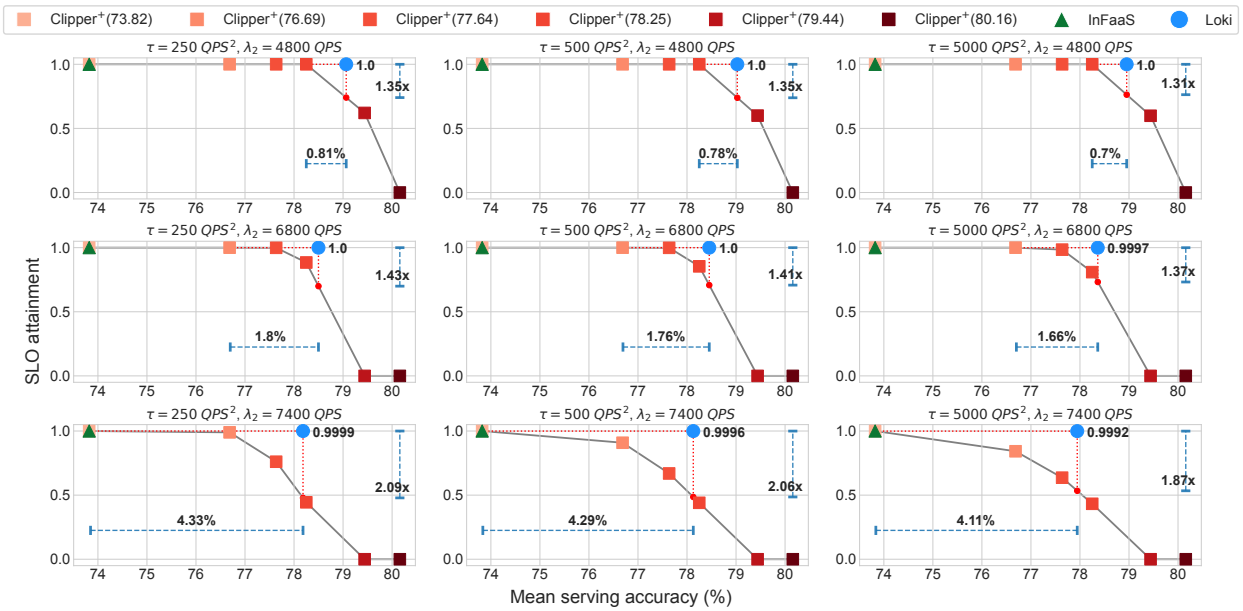


Figure 5.9: SuperServe with variable burstiness. SuperServe outperforms Clipper$^+$ and INFaaS baselines by finding better tradeoffs and consistently achieving > 0.999 SLO attainment on bursty traces. Variable ingest rate $\lambda_v = \{2950, 4900, 5550\}$ q/s increases vertically (down). $CV_a^2 = \{2, 4, 8\}$ increases horizontally (across). SuperServe achieves a better trade-off in SLO attainment (y-axis) and mean serving accuracy (x-axis) in all cases. SuperServe consistently achieves high SLO attainment > 0.999.

accuracy thresholds provided (§5.6.2, §5.6.2). In such a scenario, INFaaS reduces to serving the most cost-efficient model (which is the model with the minimum accuracy). We confirmed this behavior with the INFaaS authors, who agree that "[our] representation of INFaaS as a baseline that always chooses the same model is correct in the absence of an accuracy threshold, or a fixed (never changing) accuracy threshold." [105].

SubNet-Profiling. We use a ResNet-based SuperNet trained on ImageNet [91] released by [61] and enable SubNetAct in it. We extract pareto-optimal SubNets (Φ_{pareto}) by running NAS (publicly released by [61]) on the trained SuperNet. The pareto-optimal SubNets in the SuperNet span $0.9 - 7.5$ GFLOPs range and an accuracy range of $73 - 80\%$. The SubNets are profiled with varied batch sizes on NVIDIA RTX2080Ti GPU.

Test-Bed. SuperServe is implemented in 17.5k lines of C++. gRPC [129] is used for communication between the client, the router and workers. The experiments use 8 RTX2080Ti GPUs and 24 CPU cores, with each worker assigned one GPU.

5.6.2 End-to-End: Synthetic

We aim to answer the following questions, whether SuperServe (a) *automatically* serves queries using appropriate models (accuracy) for different traces (**R2**), (b) achieves a better

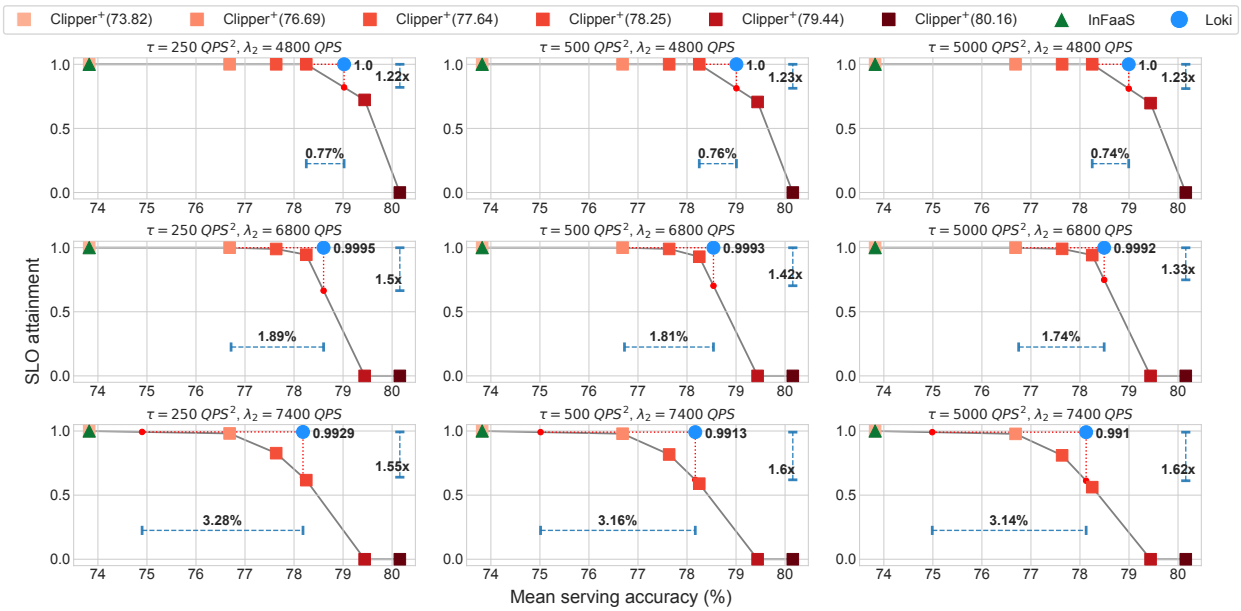


Figure 5.10: SuperServe with arrival acceleration. SuperServe outperforms Clipper$^+$ and INFaaS baselines by finding better tradeoffs on time varying traces. Mean ingest rate accelerates from λ_1 to λ_2 q/s with τ q/s^2. $\tau = \{250, 500, 5000\}$ increases horizontally (across), while $\lambda_2 = \{4800, 6800, 7800\}$ increases vertically (down) with $\lambda_1 = 2500$ q/s and $CV_a^2 = 8$ staying constant. SuperServe finds a better trade-off in SLO attainment (y-axis) and mean serving accuracy (x-axis).

trade-off with respect to the success metrics (**R1-R2**), (c) withstands sharp bursts while maintaining high SLO attainment (**R1**) and (d) *instantaneously* changes system throughput where mean ingest rate changes over time. To answer these questions, we evaluate SuperServe on the bursty and time-varying traces (§5.6.1).

Baseline comparison with burstiness

Fig. 5.9 compares SuperServe with the baselines over a range of traces increasing mean ingest rate λ_v across and CV_a^2 down. All traces are configured with 36ms SLO. Achieving high SLO attainment (**R1**) and high mean serving accuracy (**R2**) is desirable, which implies the *best trade-off is in the top-right corner of the graph.* We demonstrate that no single choice of a model is *sufficient* for different mean arrival rates and CV_a^2. For instance, the SLO attainment of Clipper$^+$(76.69) decreases as the CV_a^2 increases for $\lambda_v = 5550$ (row 3). Similarly, the SLO attainment of Clipper$^+$(78.25) decreases with increase in λ_v for $CV_a^2 = 2$ (column 1). We draw the following takeaways: **(1)** SuperServe achieves a significantly better trade-off between SLO attainment and accuracy (**R1-R2**) than the baselines (Clipper$^+$ and InFaaS). It is 4.33% more accurate than the baselines at an SLO attainment level of 0.9999 and 2.06x higher SLO attainment at the same accuracy level. SuperServe is consistently at the top-right corner in Fig. 5.9 across all the traces. **(2)** SlackFit *automatically* selects

appropriate models for sustaining different traffic conditions. As λ_v increases, SuperServe reduces serving accuracy while maintaining high SLO attainment (columns).

Note that, across all the traces, InFaaS achieves an optimal SLO attainment but with a significantly smaller mean serving accuracy (by up to 4.33%) than SuperServe. InFaaS's policy serves the min-cost (and hence min accuracy) model for the trace without accuracy constraints. Whereas, SuperServe achieves a better trade-off between the success metrics because (a) SubNetAct allows in place activation of different SubNets *without* affecting SLO attainment (**R1**); (b) SlackFit opportunistically selects higher accuracy models based on query's slack (**R2**). Also, the difference between SuperServe and Clipper$^+$ narrows with respect to accuracy as CV_a^2 increases, since SlackFit switches to lower accuracy models more frequently with burstier traffic.

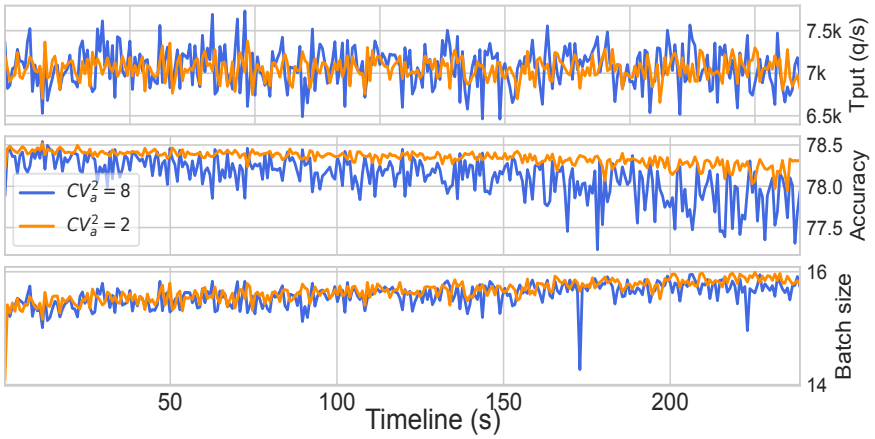
System Dynamics

We also derive key observations from the dynamics to understand how SuperServe achieves high SLO attainment and better trade-offs (**R1-R2**) for synthetic traces.

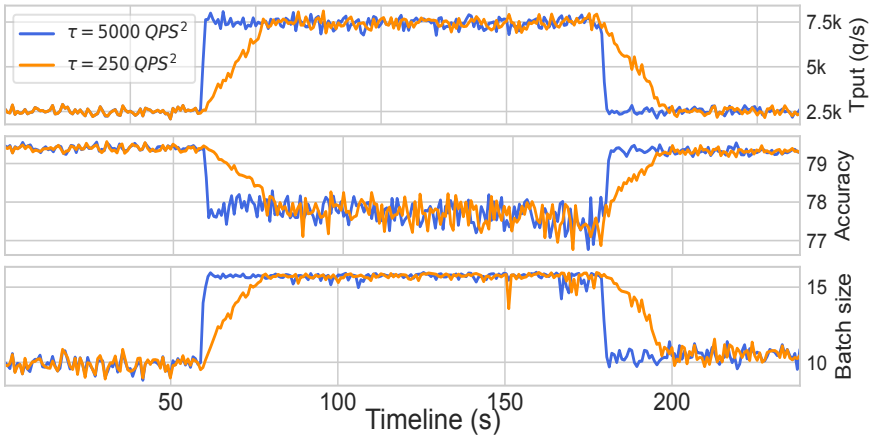
Fig. 5.11 shows the system dynamics of SuperServe for both bursty and time-varying traces. The mean ingest rate of the bursty traces is 7000 qps and they vary in $CV_a^2 = \{2, 8\}$. Similarly, in case of the time-varying traces, the ingest rate is increased from λ_1 qps to λ_2 qps at varying accelerations $\tau = \{250\ q/s^2, 5000\ q/s^2\}$. The control decisions made by SlackFit (subnetwork (accuracy) and batch size) are shown over time.

Fig. 5.11a shows system dynamics for the bursty traces. The trace with $CV^2 = 8$ (blue line) has higher spikes than the trace with $CV^2 = 2$ (orange line). First, note that SuperServe operates at an accuracy range of $76 - 78\%$ and never selects a higher accuracy subnetwork such as the subnetwork of 80.16% accuracy. This is because the subnetwork of 80.16% accuracy diverges at the mean ingest rate of 7000 qps (also seen in Fig. 5.9 last row). Hence, SuperServe *automatically* selects appropriate subnetworks for different mean ingest rates. Moreover, SuperServe uses lower accuracy models more frequently with increasing CV_a^2. This is because increased jitter reduces query slack, causing SlackFit to pick lower latency buckets more often. This corroborates the trend seen in Fig. 5.9 where the mean serving accuracy of SuperServe monotonically decreases as CV_a^2 increases. Lastly, during the load spikes, SlackFit usually selects control parameters with high batch size and smaller subnetwork (§5.4.2). This control decision allows SuperServe to drain the queue faster, resulting in a high SLO attainment on the traces (**R1**).

Fig. 5.11b shows the system dynamics for the time-varying traces. $\tau = 5000\ q/s^2$ (blue line) increases the ingest rate from 2500 qps to 7400 qps faster than $\tau = 250\ q/s^2$. For both the traces, SuperServe dynamically changes the accuracy from ≈ 79.2 to ≈ 77.5 as mean ingest rate increases. SuperServe's ability to dynamically adjust accuracy helps it achieve a higher mean serving accuracy (**R2**) compared to serving a single model statistically. Moreover, for $\tau = 5000\ q/s^2$, SuperServe jumps to lower accuracy and higher batch size control parameters quickly. While, for for $\tau = 250\ q/s^2$, SuperServe uses intermediate models to serve the intermediate ingest rate during $\approx 60 - 80$ seconds. A higher τ value



(a) Dynamic accuracy and batch size control: bursty traces



(b) Dynamic accuracy and batch size control: time-varying

Figure 5.11: System Dynamics on Synthetic Traces. Accuracy and batch size control decisions shown over time in response to ingest throughput (q/s). (a) bursty traces $\lambda = 7000 = (\lambda_b = 1500) + (\lambda_v = 5500)$ with burstiness of $CV_a^2 = 2$ (orange) and $CV_a^2 = 8$ (blue). (b) time varying traces accelerate from $\lambda_1 = 2500$ q/s to $\lambda_2 = 7400$ q/s with acceleration $\tau = 250q/s^2$ (orange) and $\tau = 5000q/s^2$ (blue). Batch size and subnetwork activation control choices over time show how SuperServe reacts to each of the four plotted traces in real time. This illustrates dynamic latency/accuracy space navigation.

forces query's slack to reduce drastically. Hence, SlackFit rapidly switches to selecting control parameters of smaller subnetwork and higher batch size from the low latency buckets (§5.4.2) to satisfy deadlines (**R1**). Therefore, increase in τ decreases mean serving accuracy (a trend observed in Fig. 5.10 across the rows).

Baseline comparison with arrival acceleration

Fig. 5.10 evaluates SuperServe's performance at different levels of arrival rate change (i.e., arrival *acceleration*). Traces start at λ_1 and increase to λ_2 with acceleration τ. Traces

fix $\lambda_1 = 2500qps$ and $CV_a^2 = 8$ but change λ_2 and acceleration τ .

The τ and λ_2 are chosen to demonstrate that single, pre-configured model choices are inadequate to sustain different rates of arrival (mean λ) and acceleration (τ). Clipper$^+$(79.44) starts diverging as τ increases (λ_2 is 6800 *qps* (row 2)). Similarly, Clipper$^+$(79.44) starts diverging with increase in λ_2 (τ = 250 q/s^2 (column 1)). The key takeaways from this experiment are as follows:

- SuperServe *rapidly* scales system throughput and achieves a high SLO attainment (0.991-1.0) even with high values of τ (5000 q/s^2). The experiment demonstrates two key properties of SuperServe— (a) the *actuation delay* in SuperServe is indeed negligible, (b) the lower *actuation delay* helps achieve higher SLO attainments for time-varying traces (**R1**). SuperServe empirically demonstrates "agile elasticity" (§5.2), and withstands high acceleration in arrival rate (τ).

- SlackFit *dynamically* adjusts the serving accuracy over time (**R2**) and achieves a better trade-off between success metrics (**R1-R2**). When the mean ingest throughput is low (λ_1), SuperServe uses higher accuracy models. It quickly switches to lower accuracy models when mean arrival rate is high (λ_2), as evident in system dynamics Fig. 5.11b.

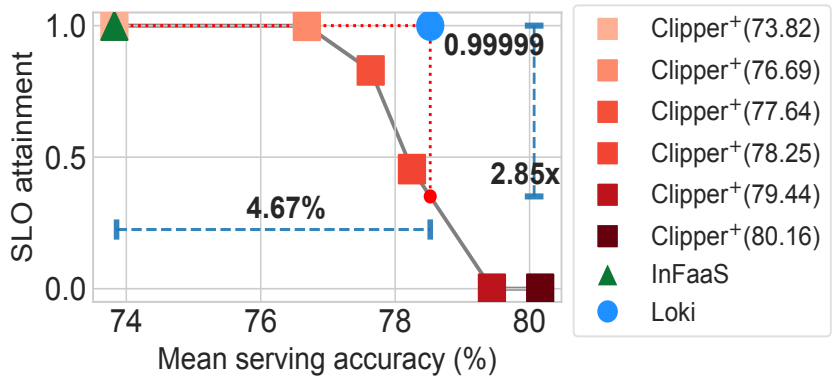
Fig. 5.10 experiments exhibit interesting trends. As the τ increases, the gap between SuperServe and Clipper$^+$ with respect to mean serving accuracy narrows. This is because SlackFitselects smaller accuracy sooner with the increase in τ. Lower τ values give enough time to SuperServe to serve intermediate mean arrival rates with higher accuracy models while gradually moving to lower accuracy models as mean ingest rate increases to λ_2 qps. Whereas, InFaaS continues to serve min accuracy model for all traces as its policy doesn't maximize accuracy by design.

5.6.3 End-to-End: Real Workloads

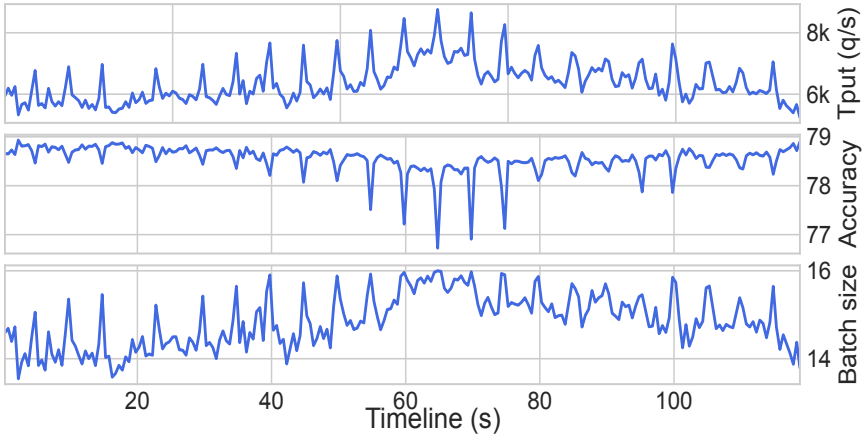
We investigate if: (a) SlackFit is capable of achieving a better trade-off between SLO attainment and mean serving accuracy on real workloads (**R1-R2**), and (b) SubNetAct contributes to serve highly unpredictable workloads at high SLO attainment.

We use the MAF trace [273] to evaluate SuperServe (similar to Clockwork [132]). The trace is collected on Microsoft's serverless platform and serves as a reasonable workload to evaluate SuperServe as serverless ML inference is an active research area [332, 155]. It consists of number of invocations made for each function per minute and contains nearly 46,000 different function workloads that are bursty, periodic, and fluctuate over time. We use 32,700 function workloads from the MAF trace, resulting in a mean arrival rate of 6400 qps. The 24 hour long trace is shrunk to 120 seconds using *shape-preserving* transformations to match our testbed.

Result. Fig. 5.12a compares SuperServe with Clipper$^+$ and InFaaS on the real-world MAF trace. SuperServe achieves an SLO attainment (**R1**) of 0.99999 (five '9's). Compared to Clipper$^+$ and InFaaS , SuperServe is 4.65% more accurate (**R2**) at the same level of SLO



(a) SuperServe achieves best tradeoffs w.r.t. SLO attainment and accuracy on a real trace.

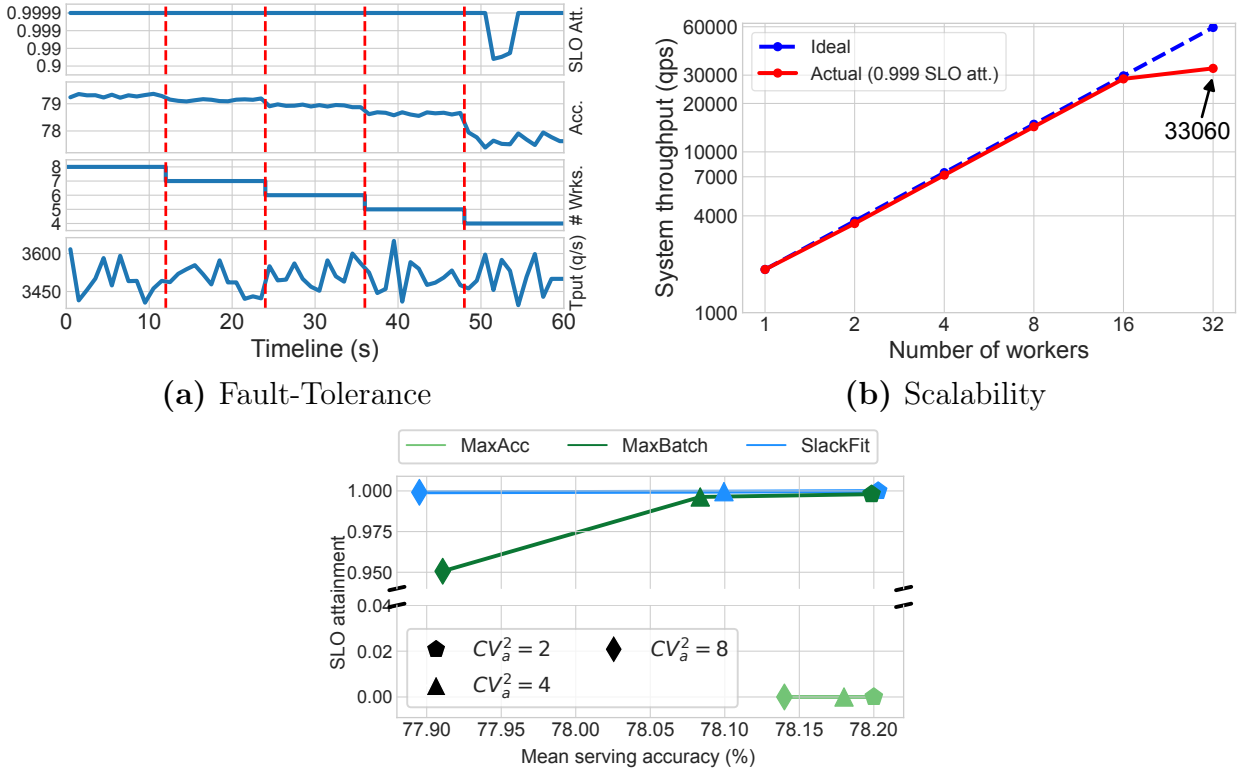


(b) SuperServe's system dynamics

Figure 5.12: SuperServe on a Real World Trace. SuperServe on Microsoft Azure Functions (MAF) [273] trace. (a) SuperServe is compared with Clipper$^+$ and INFaaS baselines, reaching 4.67% higher accuracy at same SLO attainment and 2.85x higher SLO attainment at same accuracy than any fixed accuracy point that can be served by INFaaS (in the absence of accuracy constraints) and Clipper$^+$. (b) System dynamics w.r.t. batch size and SubNet activation control decisions over time in response to ingest rate in the top graph.

attainment. It also achieves a 2.85x factor improvement in SLO attainment at the same mean serving accuracy. Moreover, Clipper$^+$(79.44, 80.16) diverges on the MAF trace.

System Dynamics. Fig. 5.12b shows the ingest throughput (qps), serving accuracy and batch size control decisions (made by SlackFit) for the MAF trace. As seen in the figure, the trace contains periodic short-interval spikes that reach upto 8750 qps, demonstrating the agility of the system. SlackFit selects both smaller accuracy model and higher batch size during the load spikes to meet the deadline (**R1**). SlackFit makes such control decisions because it uses query's slack as a signal to maximize batch size. As the query slack decreases,



(a) Fault-Tolerance **(b)** Scalability

(c) Policy Space Exploration

Figure 5.13: SuperServe's Micro-benchmarks. **(a)** SuperServe resiliency to faults. SuperServe maintains high SLO attainment in the system by dynamically adjusting served accuracy as workers drop out over time. The trace stays statistically the same ($\lambda = 3500$ qps, $CV_a^2 = 2$ (last row)). **(b)** SuperServe scales linearly with the number of workers, achieving up to 33000 qps (orders of magnitude higher than published SotA systems) while maintaining high .999 SLO attainment. **(c)** SlackFit finds the best tradeoff on the SLO attainment/accuracy maximization continuum automatically (§5.6.4).

it selects maximum batch size control parameters in the lower latency buckets. Furthermore, these control decisions increase the system throughput instantly through SubNetAct. Lastly, SlackFit serves higher accuracy models when the ingest rate is low and hence, achieves better mean serving accuracy (**R2**).

5.6.4 Microbenchmarks

Fault Tolerance. SubNetAct mechanism provides an additional advantage of transparent fault tolerance. We run SuperServe with 100% capacity (8 workers) with a bursty traffic trace ($\lambda = 3500$ qps, $CV_a^2 = 2$) for 60 seconds and gradually kill a worker every 12 seconds to simulate faults. SuperServe shows resilience to decreases in system throughput to as low as 50% by maintaining SLO attainment as high as 0.999 for the unchanging trace as it leverages

subnetwork activation to serve lower accuracy models automatically. Similar methodology was used in [316].

Fig. 5.13a shows SLO attainment as a function of time (along with other system dynamics). As the faults occur (workers killed, dotted red lines), SuperServe *automatically* transitions to lower accuracy models to maintain high SLO attainment. We attribute SuperServe's fault tolerance to (a) a wide-dynamic throughput range afforded by SubNetAct (Fig. 5.5c) that allows SuperServe to serve the workload even with 50% capacity, and (b) SubNetAct's low actuation delay that provides *agility* to rapidly increase system-throughput (during faults) without sacrificing SLO attainment (**R1**).

Scalability. We assess if SuperServe reaches high SLO attainment at scale. To show this, we scale the number of workers and observe the maximum throughput SuperServe sustains to reach SLO attainment of 0.999. We serve ResNet-18 [139] across all the workers with clients providing a batch of 8 images[4]. Scalability experiments are conducted with $CV_a^2 = 0$. Fig. 5.13b shows sustained ingest throughput with the increase in workers. In this experiment SuperServe achieves an SLO attainment of 0.999 while reaching throughputs as high as $\approx$ 33000 qps.

Policy Space Exploration. We now compare the following different scheduling policies:

- **MaxBatch Policy:** This policy first maximizes the batch size and then the accuracy. It greedily finds a maximal batch size (b) for the smallest accuracy subnetwork that fits within latency slack θ.Within the chosen batch size MaxBatch finds the maximum accuracy subnetwork (s) such that the profiled latency $L(b, s) < \theta$. It returns the control choice (b, s). This policy leverages insights (**I1**) and (**I2**). It takes O($log(B)$) operations to find b and O($log(S)$) operations to find s (binary search on monotonically increasing latency with respect to batch size and accuracy). As a result, this lightweight policy scales well with the profile table, taking only O($log(B) + log(S)$) operations to make control decisions.

- **MaxAcc Policy.** MaxAcc first maximizes the accuracy and then the batch size. Mirroring MaxBatch, MaxAcc performs a binary search for the largest accuracy (s') with $L(1, s') < \theta$ first. Then, it finds the maximal batch size (b') keeping the subnetwork choice fixed to the chosen s', such that $L(b', s') < \theta$ ms. Similarly to MaxBatch policy, it leverages insights (**I1**) and (**I2**) and takes O($log(B) + log(S)$) operations to return the control choice (b', s').

- **SlackFit Policy.** This is our best performing policy. At a high level, SlackFit partitions the set of feasible profiled latencies into evenly sized latency buckets. Each bucket consists of control tuples (b, s) with $L(b, s)$ within the range of bucket width. Then the policy chooses a bucket with latency $\leq \theta$. Finally, from the choices within the selected bucket, it picks the control choice that maximizes batch size. Intuitively,

[4]We don't perform adaptive batching for this experiment

selecting control parameters closest to slack θ configures the system to operate as close to capacity as possible.In other words, choices with latency less than that either reduce the throughput capacity or the serving accuracy, eventually lowering system's SLO attainment and accuracy. This draws on the monotonicity insights (**I1**) and (**I2**). SlackFit's novelty is in insight (**I3**). We observe that SlackFit *dynamically* detects and adapts to the runtime difficulty of the trace. A well-behaved trace (e.g., low ingest rate, variation, acceleration) results in higher θ. Higher θ leads to the choice of higher latency buckets. And higher latency buckets are correlated strongly with fewer control tuple choices (Fig. 5.6), maximizing the probability of choosing higher accuracy models. Conversely, mal-behaved traces (higher ingest rate, variation, acceleration) lead to lower latency bucket choices, as the scheduler is operating under much lower θ conditions. There are more control choices in lower latency buckets, which leads to control tuples within those buckets to favor higher batch sizes. This leads to processing the queue faster.

In Fig. 5.13c we show that SlackFit achieves the best tradeoff with respect to our success metrics compared to both MaxAcc – a policy that greedily maximizes accuracy and MaxBatch — a policy that greedily maximizes batches. The traces used mean $\lambda = 7000$ qps $((\lambda_b = 1500) + (\lambda_v = 5550))$ and $CV_a^2 \in \{2, 4, 8\}$. SlackFit reaches the highest SLO attainment(0.999) for all CV_a^2. MaxBatch starts under performing with respect to SLO attainment with CV_a^2 increase. The SlackFit and MaxBatch difference is most pronounced at the highest CV_a^2, eventually causing a significant 5% drop in the SLO attainment. Both policies maximize the batch size within latency slack θ when operating under small θ. When θ increases, however, MaxBatch continues to maximize the batch size unconditionally—a greedy choice that leads to packing larger batches. This greedy decision causes more time to be spent in a worker compared to SlackFit, which adaptively shifts to higher accuracy models under larger θ conditions with compound effect on queued queries, eventually missing their SLOs. maxAcc is unable to keep up with this trace. It never switches to policy decisions that process the queue faster. This policy comparison shows a continuum between faster queue processing and serving higher accuracy, with SlackFit automatically finding the best point in this continuum.

5.7 Related Work

Training SuperNets was first proposed by OFA [61]. Recent works such as CompOFA [264] and BigNAS [334] propose improvements to the SuperNet training. CompOFA makes the training of SuperNets faster and more accurate by training a fewer number of SubNets simultaneously. On the other hand, BigNAS trains the SuperNet in one-shot with a wider range of SubNets. DynaBERT [144] trains a SuperNet based on the Transformer architecture for text datasets. Similarly, AutoFormer [70] trains SuperNets derived from vision transformers. NasViT [120] trains the SuperNet for semantic segmentation tasks and achieves a better

trade-off between accuracy and latency at fewer FLOPs. SuperServe provides system support for serving SuperNets trained using any existing technique.

Model Serving Systems can be divided into two categorizes — a) Non-Automated, and b) Automated. *Non-automated serving system* expect developers to provide the prediction models and make explicit choices in the accuracy-latency trade-off space. TensorFlow Serving [228] serves the models trained in TensorFlow framework while Clipper [79] and Triton [225] support models trained from multiple frameworks. Clockwork [132] guarantees predictable tail latency for DNN inference by making cross-stack design decisions explicitly for worst case predictability. Inferline [80] provides support for provisioning inference pipelines that consist of multiple models, but the models are still hard coded in the pipeline vertices. Prior works in this category are complementary to SuperServe. For instance, SuperServe's workers can be made more predictable by consolidating choices like Clockwork. Inferline's autoscaling policy can be used on top of SuperServe Triton's model optimizations for GPU serving can be done to the SuperNet itself.

In contrast, *automated serving systems* [260, 342] automate the navigation of the accuracy-latency trade-off space with a policy, resulting in automatic DNN selection at runtime. However, both [260] and [342] use state-of-the-art DNNs (e.g., ResNets, MobileNets) and rely on model loading mechanisms instead of SuperNets, which offers better pareto-optimality and orders of magnitude faster model switching enabled via proposed SubNetAct. More importantly, these mechanisms implicitly bias their policies to avoid model switching, which limits their ability to respond to bursty request rates in an agile fashion. Specifically, InFaaS's DNN switching policy is biased towards selecting the least accurate DNNs that satisfy accuracy constraints, as the goal of the stated goal of the system is to satisfy constraints instead of treating accuracy as an optimization objective. SuperServe supports model serving via SubNet activation, thus addressing the model switching overhead through its proposed SubNetAct and SlackFit.

5.8 Conclusion

We describe a novel mechanism SubNetAct that carefully inserts specialized control flow operators into SuperNets to enable a resource-efficient, fine-grained navigation of the latency-accuracy tradeoff space. SubNetAct unlocks the design space of reactive scheduling policies. We design a simple, yet effective greedy heuristic-based scheduling policy SlackFit. SuperServe, which uses SubNetAct and SlackFit, achieves 4.67% better accuracy at the same level of SLO attainment or 2.85x better SLO attainment at the same level of accuracy compared to state-of-the-art inference serving systems.

Chapter 6

DAGSched: Deadline and DAG-Aware Declarative Scheduling

An AV's sensor suite generates data at a higher frequency (see Table 1.1) than the processing rate of the computational pipeline shown in Fig. 1.2. As a result, an execution system like ERDOS must efficiently multiplex the available compute resources at the hardware layer amongst the various invocations of an AV's computational pipeline executing concurrently. The AV pipelines developed using D3 present three key scheduling requirements: (*i*) *precedence constraints*, due to the DAG-based structure of D3, (*ii*) *placement preferences*, to efficiently exploit the heterogeneous resources available in the AV, and (*iii*) *timing constraints*, that require the computational pipeline to finish within a deadline.

State-of-the-art schedulers provide partial support for these requirements through either suboptimal heuristics or brittle, hand-crafted mathematical models. These approaches negatively impact the efficiency of resource utilization, the ability to meet deadlines and are tedious and error-prone to implement and maintain. In this chapter, we introduce a framework for simplifying the development of efficient solver-based schedulers through a declarative specification of requirements. Schedulers built using DAGSched simply specify the tasks of the job along with their requirements using the intuitive abstractions of its specification language, STRL++. The language automatically constructs a rich context surrounding the specified scheduling problem, and uses it to guide complex modelling decisions to generate tailored, efficient mathematical models for each invocation, without any human input.

The remainder of this chapter explores the design of the specification language, STRL++, and the optimization framework of DAGSched. §6.6 evaluates the efficacy of our approach with a goal of ensuring higher resource efficiency and attainment of deadlines.

6.1 Introduction

Modern jobs (e.g., data processing and ML training) are structured as Directed Acyclic Graphs (DAGs) of tasks. This programming model has seen widespread success in big

	Schedulers										
	Heuristics							Solver-Based			
	FIFO	EDF	DRF	Tetris	D Rayon	Graphene	Decima	Firmament	D DCM	D TetriSched	D DAGSched
R1	✗	✗	✗	✗	✓	✓	✓	✗	✗	✗	✓
R2	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓
R3	✗	✓	✗	✗	✓	✗	✗	✗	✗	✓	✓

Table 6.1: Current schedulers fail to support R1-R3. Both heuristic and solver-based approaches do not support R1-R3. Non-declarative schedulers (not marked with **D**) rely on experts to develop complex heuristics or models, significantly hindering their development.

data [88, 336, 338, 153, 63, 10, 204, 121], ML training [4, 73, 235] and workflow execution frameworks [27, 83, 247, 117], all of which collectively account for a significant fraction of the resource utilization in modern clusters [237]. As a result, the schedulers for these frameworks have the arduous task of making efficient use of these expensive resources [174], while also meeting three key requirements of the jobs that they execute:

R1: Precedence Constraints arising from data dependencies amongst vertices of the computational DAG of the job [302, 19].

R2: Placement Preferences arising out of task affinity (e.g., preference of heterogeneous hardware), anti-affinity (to prevent interference), and gang scheduling [49, 137, 292, 304].

R3: Timing Constraints arising out of production jobs requiring completion within stringent deadlines [53, 166, 185, 82, 302].

Table 6.1 shows that state-of-the-art schedulers fall short of collectively supporting R1-R3. Indeed, heuristic-based schedulers [115, 126, 82, 125, 198] trade-off task placement decision quality for efficiency, and cannot guarantee job-level objectives such as placement preferences amongst tasks (R2) or deadlines (R3). On the other hand, solver-based schedulers promise high-quality placement decisions, but either: (*i*) rely on domain experts to hand-craft mathematical models [118, 154, 206], or (*ii*) automatically generate models from a declarative specification of scheduling requirements in higher-level languages [82, 310, 287]. While hand-crafted models can be efficient, developing them for the complex application requirements (R1-R3) is a formidable challenge that severely limits scheduler evolvability [287]. On the other hand, declarative approaches to building schedulers vastly simplify the process of developing solver-based schedulers. However, limitations in their specification languages mean that these schedulers cannot collectively support R1-R3 and cannot generate models as efficient as hand-crafted ones.

This paper introduces DAGSched—a declarative framework that resolves this tension in

developing solver-based schedulers through two novel contributions: (*a*) a high-level requirement specification language that can comprehensively capture R1-R3, and (*b*) an extensible optimization framework that leverages the expressivity of this language to automatically generate efficient models. DAGSched's key innovation is its careful language design that automatically constructs a rich context surrounding the scheduling problem from the individual placement choices of tasks. DAGSched then exploits this contextual information to automatically guide modelling decisions that typically require domain experts. Together, DAGSched maintains the simplicity and ease-of-development afforded by declarative schedulers while generating bespoke, efficient models for each scheduler invocation that scale amidst the extended combinatorial search space required to support R1-R3.

The design of a language that declaratively captures R1-R3 necessitates: (*i*) efficient abstractions over resource allocations across space and time to enable placement and timing preferences (R2-R3) of tasks, and (*ii*) expressive primitives to reason about placement and completion times of these tasks, and specify dependencies between them (R1). In §6.4, we introduce STRL++ that provides novel extensions to the robustly validated abstractions of STRL [310] to address (*i*). In addition, STRL++ introduces novel abstractions that enable it to declaratively reason about the start and end time of the placement choices for tasks, allowing it to address (*ii*) efficiently.

To meet R3 while respecting R1, DAGSched must schedule all the individual tasks of an application collectively. However, a direct translation of STRL++ for such increased problem sizes generates intractable models. To address this challenge, DAGSched exploits STRL++'s expressivity to construct additional context surrounding the placement decisions requested by an application. §6.5 elaborates on the extensible optimization framework enabled by this insight that allows DAGSched to scale to real-world problem sizes. Specifically, we provide representative optimizations that exploit this context for: (*i*) *fidelity-preserving transformations* that efficiently prune the model of infeasible placement choices for tasks, and (*ii*) *fidelity-altering transformations* that automatically make modelling decisions which vastly reduce complexity at the expense of decision quality (e.g., the granularity of placement decisions). Prior schedulers rely on solvers for (*i*), which must reconstruct the context from a mathematical representation, significantly hampering scalability. More importantly, they leave the critical modelling decisions for (*ii*) to their users. As a result, the users are forced to make these decisions statically, which significantly degrades the placement decision quality.

We instantiate Spark-DAGSched and evaluate the efficacy of our system DAGSched in scheduling real-world DAGs derived from the Alibaba cluster trace [19] and on a Spark cluster running TPC-H [249] jobs. We achieve a 43.75% improvement in job arrival rate for 99% deadline attainment, and up to 3.7× increase in deadline attainment under high load.

The remainder of this chapter describes the contributions that enable STRL++ and DAGSched, and is organized as follows:

1. §6.2 motivates an R1-R3-aware scheduler.

2. §6.3 presents an overview of DAGSched and the critical challenges it must solve to enable efficient, declarative, R1-R3-aware schedulers.

3. §6.4 details STRL++, the first declarative scheduling language that supports R1-R3. STRL++ is expressive and lends itself to the development of efficient optimizations.

4. §6.5 details DAGSched's optimization framework that exploits STRL++'s expressivity to automatically guide modelling decisions and generate efficient solver encodings.

5. §6.6 evaluates the efficacy of DAGSched across several real-world workloads, with a goal of quantifying its higher resource efficiency and ability to meet end-to-end deadlines.

6.2 Motivation

We now underscore the key concerns that led to the development of DAGSched: (§6.2.1) the increasing prevalence of R1-R3 in modern applications, (§6.2.2) the need for schedulers to consider R1-R3 collectively, and (§6.2.3) the significant complexity of developing mathematical solver-based schedulers.

6.2.1 Scheduling Requirements

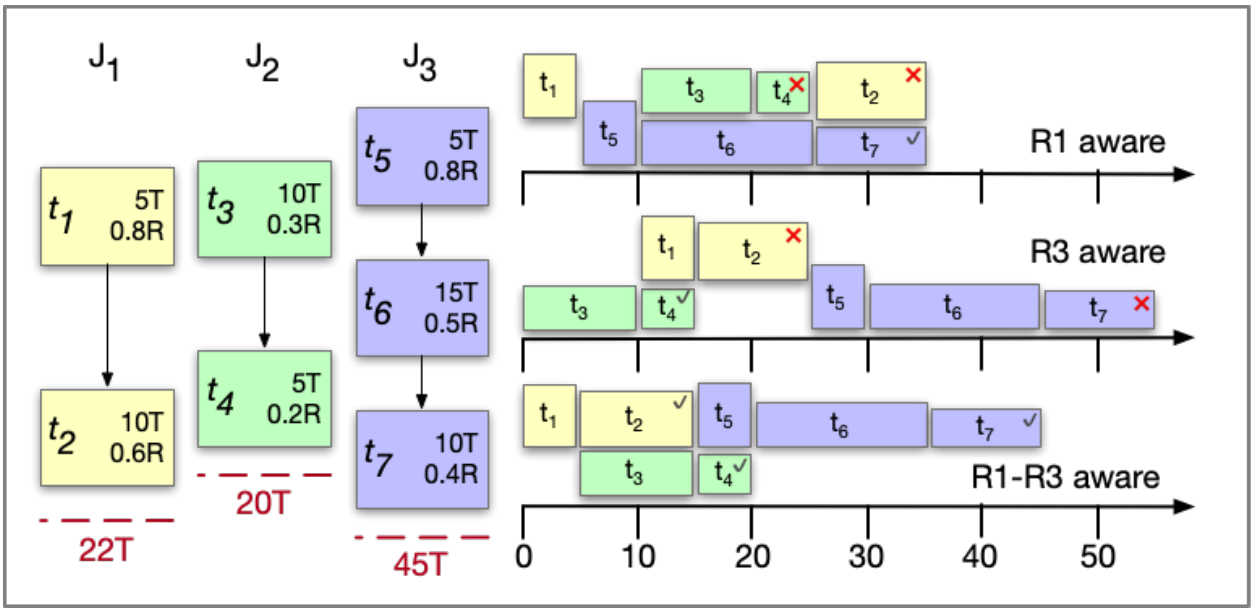
The complexity and importance to decision making of modern applications, such as Big Data processing and ML training, have established three key requirements [137, 282]:

R1: Precedence Constraints. Growing processing demands of applications are addressed with ever larger clusters. To effectively utilize these expensive resources [174], complex jobs are increasingly being broken down into DAGs of inter-dependent tasks to maximize parallel execution. For example, Microsoft's data processing platform has seen a 12× increase in the available resources and a 108× increase in DAG-based jobs in the last decade [245]. Similarly, an analysis of multiple production traces revealed an increase in DAG-based jobs from 2% to 50% between 2011-18, with such jobs now accounting for ≈ 80% of cluster utilization [302].

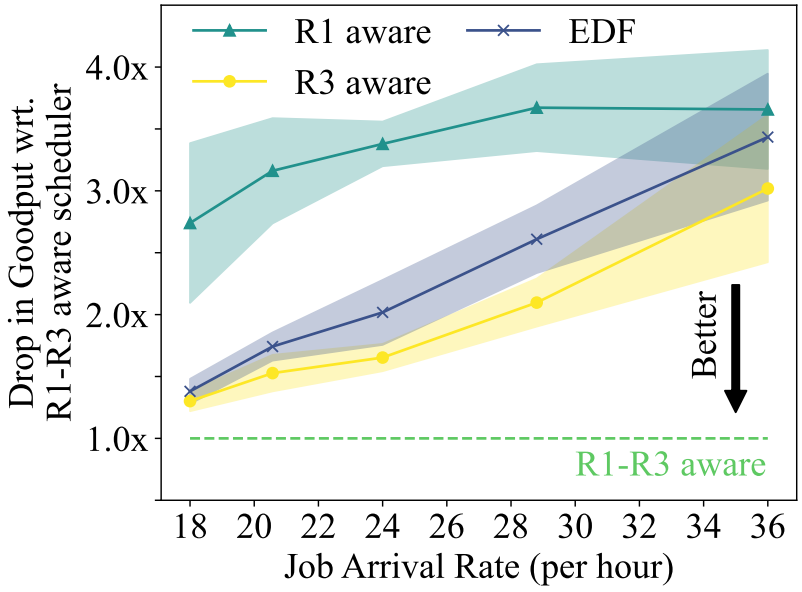
R2: Placement Preferences. Efficient execution requires placement preferences in terms of resource type and data locality. An analysis of the Google cluster in [275] shows that these preferences arise due to: (*i*) *resource heterogeneity*, e.g., GPUs for ML [137, 206], (*ii*) *application optimization*, e.g., preferring tasks to be closely located to required data [88], and (*iii*) *problem avoidance*, which requires tasks to be placed on exclusive resources to prevent interference [115, 126].

R3: Timing Constraints. As these applications become indispensable [137, 282], their output is regularly consumed by analysts, business processes and customers, that demand timely results [53, 166, 302]. A missed deadline can adversely impact quality of service, engineer efficiency, and lead to significant economic losses [53, 137, 185, 166]. This requirement's criticality is evidenced by the fact that it accounts for 25% of scheduler-related escalations in Microsoft's big data cluster [166].

These requirements lead to the following concrete problem:



(a) Illustrative workload where an R1-R3 aware scheduler meets 3× deadlines.



(b) Efficacy of R1-R3 awareness on an industrial trace.

Figure 6.1: Schedulers must support R1-R3. We underscore the need for schedulers to be R1-R3 aware by showing: (*a*) an example workload with jobs J_1-J_3, where collectively considering R1-R3 meets 3× deadlines compared to considering R1 or R3 alone, and (*b*) an up to 4.29× increase in goodput when considering R1-R3 collectively (compared to selectively) on a replay of the Alibaba industrial trace [19].

Problem Statement. Jobs J arrive for execution at a cluster with heterogeneous resources. Each job $J_i \in J$ requests execution of tasks T_i to complete within a deadline D_i (R3). The execution of tasks in T_i is constrained by R1: a precedence relation ($\rightarrow$) that defines a partial order such that for $(t_a, t_b) \in T_i$, $t_a \rightarrow t_b$ implies that t_a must complete before t_b starts. Each task $t \in T_i$ specifies its requirements for resources for a fixed duration, and may (optionally) present preferences for specific hardware, co-location with data, anti-affinity with other tasks etc. (R2). The goal is to map T_i to the available resources under these constraints, and maximize *goodput*, i.e., the number of jobs that complete by their deadlines.

6.2.2 Scheduling Applications

We now establish that maximizing goodput necessitates a collective consideration of R1-R3. Our discussion focuses on R1 and R3, as prior work [90, 206, 253, 323, 310, 49, 287] has thoroughly underscored the importance of R2 awareness.

An Illustrative Example. Fig. 6.1a shows a workload with jobs J_1-J_3 to be scheduled on one node with one unit of a resource. Each job contains a chain of tasks (denoted by t_i) and has a deadline as indicated. The duration and resource requirements of these jobs is labeled in the boxes on the left and indicated by the size of the boxes on the schedules on the right. For example, t_1 requires 0.8 resources for 5 time units.

A typical class of schedulers focuses on minimizing makespan under task precedence (R1) irrespective of deadlines (R3). That is, schedulers like Graphene [125] or Decima [198] maximize resource utilization by analyzing permissible task execution orders across all jobs in the system. In this example, they find that the resource requirements of t_3 and t_6 allow perfect utilization of 1R, likewise t_4 and t_6 as well as t_2 and t_7. But this perfect utilization requires t_1 and t_5 to execute first. The result is the schedule shown in Fig. 6.1a (R1), which violates the deadlines of t_2 and t_4 and shows that optimizing makespan/utilization is not aligned with optimizing goodput.

The other popular class of task-based schedulers instead greedily prioritize the earliest deadline (R3) irrespective of task precedence (R1). That is, EDF simply dispatch ready tasks in order of the deadline of their jobs. As illustrated in Fig. 6.1a (R3 aware), these schedulers prioritize executing the resource intensive task t_3 of job J_2 with deadline 20T. This leaves insufficient resources for ready tasks t_1 and t_5. It delays t_1 until after completion of t_3 and in turn prioritization of the resource intensive task t_2 further delays t_5 and its dependent tasks. The resulting schedule misses all but the first deadline. This shows greedy scheduling of individual tasks irrespective of resource requirements and precedence can cascade into unavoidable delays exacerbated by cascading dependencies.

The optimal R1-R3-aware scheduler carefully plans the progress of all jobs in the system to maximize goodput. It can find the opportune interleaving of the jobs J_1 and J_2 with tight deadlines (R1-R3 aware in Fig. 6.1a). Collectively considering the resource, precedence, and deadline requirements reveals to the optimal scheduler that resource demands of t_2 and t_3 are compatible and can execute concurrently allowing the dependent task t_4 to finish on time. This allows the scheduler to complete more jobs within their specified deadlines (3×

in this case). The questions now are how this improvement opportunity extends to realistic workloads and how to build such optimal scheduler that has tractable overhead.

An Industrial Workload. We now seek to quantify the effects of considering R1-R3 collectively on a real-world industrial trace. We measure the goodput of jobs from the Alibaba big data execution platform under increasing cluster load [19]. To achieve this, we simulate the execution of 300 randomly sampled jobs on a cluster with 30 units of resources and random deadlines $1.1 - 2\times$ the critical path of jobs. We model increasing cluster load through increasing job arrival rate in a Poisson process but avoid cluster overload. We report the mean and standard deviation of the number of jobs completed within their deadline from 5 runs with distinct random seeds.

Fig. 6.1b compares the drop in goodput from a solver-based R1-R3 aware scheduler to solver-based R1 aware, R3 aware schedulers along with a heuristic EDF scheduler. We find that an R1-R3 aware scheduler meets up to $4.29\times$ more deadlines than an R1 aware scheduler (Fig. 6.1b). This is because the latter maximizes task packing, but does not prioritize jobs based on their deadlines. Thus, it achieves a higher cluster utilization (up to 85%), but does not allocate resources to jobs with tighter deadlines earlier. This negative impact of deadline unwareness is more pronounced under increased arrival rates, as the increased resource contention makes the prioritization of jobs with earlier deadlines more pertinent.

Conversely, both EDF and an R3 aware scheduler prioritize jobs according to their deadlines and achieve higher goodput than an R1 aware scheduler. However, these schedulers fail to effectively pack task executions since they either cannot pack (EDF) or do not collectively consider all the tasks of the job (R3 aware). Due to these shortcomings, an R1-R3 aware scheduler meets up to $1.5\times$ more deadlines compared to these baselines even under lower resource utilization regimes. As packing of task executions becomes more important with increased arrival rates, the gap between these baselines and an R1-R3 aware scheduler increases significantly (up to $3.8\times$). This creates a sizeable opportunity gap.

6.2.3 Scheduler Design: Why Declarative?

The challenge for schedulers of moderns applications are jobs with arbitrarily complex requirements. Applications can submit jobs with arbitrary DAGs of precedence constraints, arbitrary placement preferences for any (subset of) tasks, and with or sans deadlines. Ignoring these requirements can severely degrade goodput (§6.2.2). This tasks scheduler developers to consider combinatorially many possible requirement scenarios across heterogeneous jobs in the system. This renders conventional schedulers and heuristics brittle and incapable to evolve, as motivated above and emphasized by [159, 44, 125].

Developing solver-based schedulers is a promising avenue to address complex scheduling problems [310, 207, 287]. A declarative design of solver-based schedulers can unlock an effortless, sound and efficient scheduling. It promises the separation of concerns between *expression* of the scheduling problem and *formulation* of the solver model. A high-level declarative language allows the developer to express their scheduling problem without mathematical modelling expertise by relying instead on a compiler to lower the specification into

a model for the solver to consume automatically. This separation of concerns promises an advantage of enabling independent development of expressing scheduling requirements and rendering scheduling decisions efficiently. Thus, combining an expressive language with an efficient compiler can ensure models that are both more accurate and more performant—the two desired properties for high quality scheduling decisions.

6.3 DAGSched

DAGSched aims to simplify the development of solver-based schedulers by enabling them to specify the jobs (with R1-R3) and the resources in the cluster using its proposed specification language, STRL++. DAGSched determines *which* tasks to dispatch *where* (what resources) and *when* (what start time).

6.3.1 Design Challenges

To enable this, DAGSched must address two core challenges:

(**1**) **Design a concise yet expressive language.** For an effortless specification of R1-R3, the language must be *concise* to elide enumeration of an exponentially large set of a job's placement options. Indeed, this enumeration quickly becomes intractable due to the combinatorial nature of R1-R2, as developers must consider the cumulative cascading effects of a task's placement choices on other tasks. For example, to specify precedence (R1) between M choices of t_1 and N choices of t_2 from Fig. 6.2, the developer must enumerate $\mathcal{O}(MN)$ choices. To achieve *conciseness*, the language must provide a minimum set of expressions whose composition captures the entire space of placement choices. However, this conciseness should not come at the cost of *expressivity*, i.e., the language should never capture a reduced set of placement choices for a job and affect placement quality. For example, for R1, the language must ensure that it accurately captures the complete set of topological orderings for task executions and does not introduce any false dependencies (false positives) nor miss any true "happens before" relations (false negatives) while striving for a concise representation.

(**2**) **Scale the model generation and solving.** A balance between conciseness and expressivity of the language exposes two challenges that affect scheduler scalability:

- *Expanded search space:* An expressive language that captures R1-R3 vastly increases the search space of placement options which an underlying solver must explore for optimal placement. Further, this search space exponentially increases with the number of jobs, impacting solver time and limiting scheduler scalability.
- *Efficient mathematical modelling:* High-level expressions of the language are necessary for a concise representation of a job's placement options. However, expressions can be modelled in a variety of ways (e.g., different types of variables, indicator or integer), that may produce equivalent mathematical models but with vastly different solver runtimes [287]. Crucially, optimal modelling cannot be static as it depends on the workload

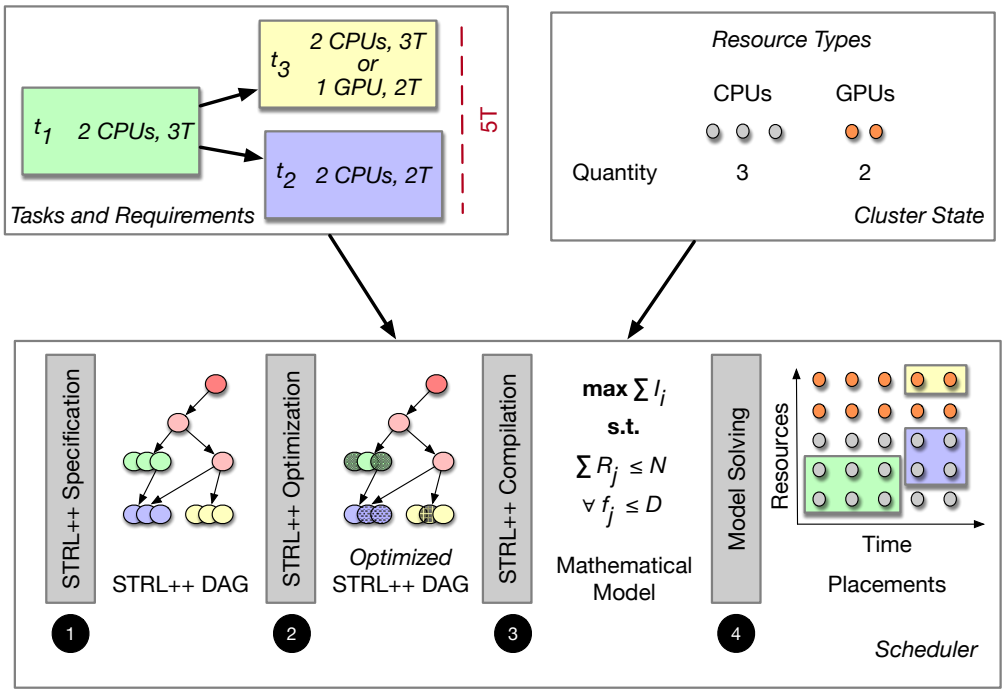


Figure 6.2: Workflow of a scheduler built using DAGSched. The scheduler specifies the tasks and their requirements by constructing a DAG of STRL++ expressions. The DAG is optimized and compiled into a model and passed to an off-the-shelf solver. Finally, the scheduler retrieves the placements for all tasks from DAGSched and dispatches the tasks to the assigned resources at the specified time.

and cluster state, varying widely with each invocation. For instance, placement choices for workloads may be modelled by different granularity of time discretizations to tame solver runtimes with minimal impact on decision quality (§6.5).

6.3.2 DAGSched's Workflow

DAGSched's workflow seeks to overcome these challenges through 4 distinct phases starting from capturing resource requirements to making spatiotemporal placement decisions. Fig. 6.2 illustrates a job following the steps below to determine its start time and resource allocation on 3 CPUs and 2 GPUs:

Step ❶: Specifying R1-R3 in STRL++. The scheduler specifies resource requirements and expected runtimes for t_{1-3}, and identifies the available resources in the cluster by constructing a DAG of STRL++ expressions. An important aspect of this construction is that the scheduler only enumerates the placement choices for each task independently, without considering their impact on choices for other tasks. These expressions (defined in §6.4.2) capture: R1, the ordering between $t_1 \rightarrow \{t_2, t_3\}$; R2, the preference of t_3 for a GPU; and R3, the deadline of $5T$ by which all of the job's tasks must complete. Unlike the job's task

graph, a STRL++ DAG concisely captures the space of placement options for all tasks and how they're constrained by the placement choices of other tasks.

Step ❷: Optimizing the STRL++ DAG. In the face of the expanded search space required to support R1-R3, STRL++ designs all of its expressions to be *time-aware* by automatically adding declarative start and end time annotations to each expression. This unlocks a rich optimization framework atop STRL++ where optimizations reason about the time bounds of the feasible placement choices captured by each expression, and use them to automatically prune the expanded search space. Schedulers can benefit from optimizations provided by the DAGSched framework (§6.5) to their STRL++ DAG.

Step ❸: Compiling the STRL++ DAG. DAGSched automatically compiles the now optimized STRL++ DAG into a mathematical model, e.g., Integer Linear Program (ILP). At this stage, DAGSched automatically uses integer decision variables to efficiently encode R1, leveraging time-aware expressions generated in Step ❶. This key insight allows DAGSched to express the $t_1 \rightarrow t_2$ relation, with M t_1 and N t_2 placement options using ***only*** $\mathcal{O}(1)$ constraints! In sharp contrast, prior work [82] modelled time using indicator variables, requiring $\mathcal{O}(MN)$ constraints to capture $t_1 \rightarrow t_2$. This difference in modelling was a "key limiting factor for practical use", and led [82] authors to abandon solver-based scheduling resolving to best-effort heuristics.

Step ❹: Solving the model. Finally, DAGSched passes the model to an off-the-shelf ILP solver while interposing on the solving process to ensure scalability:

- *Warm Starts:* DAGSched reuses prior placement decisions to provide an initial feasible solution to the current model, speeding up the solver's search for the optimal solution.
- *Interrupts:* DAGSched computes an upper bound on the ILP model's objective and issues a solver interrupt as soon as an objective satisfying solution is found.

6.4 STRL++

This section details the key challenges for a concise yet expressive specification of job requirements R1-R3 (§6.4.1). We then discuss how STRL++'s expressions overcome them to capture a combinatorial space of placement options and enable an effortless declarative specification of the scheduler (§6.4.2). Finally, we elaborate on how DAGSched efficiently compiles the STRL++ expressions to an ILP model (§6.4.3).

6.4.1 Language Requirements

A specification language must provide an abstraction of the available resources. Intuitively, this is represented by a discrete **resource-time space** where each unit in the space represents the availability of a given resource at a particular time. Schedulers then make control decisions w.r.t. allocation of this resource-time space. In this language, the unit of

specification is a **placement choice** that represents a rectangle within the resource-time space. The height of this rectangle refers to the quantity of resources required by a task, and the width—the task's duration. The position of the rectangle in the space indicates the type of resources allocated, along with the task's start time.

Thus, language primitives should then enable the specification of the shape of this resource-time rectangle, as well as the segment of the resource-time space within which it must lie, with the latter—modifiable to support R1-R3 as follows:

- **R3:** Schedulers should be able to enumerate only the placement choices that complete within the task's deadline. To do so, the language should enable the resource-time space to be bounded with respect to time. For example, Fig. 6.3a visualizes a constrained space for t_1 (in red) that restricts rectangles to complete within 5 time units to meet t_1's deadline.

- **R2:** To specify preferences for a task's execution at a certain resource or time, the language must enable schedulers to define partial orders over the discrete units of the resource-time space. For example, for t_3's preference of a GPU, the scheduler orders the two resources CPU $\prec$ GPU, signifying that all time units of resource-time space from the GPU are strictly ordered (i.e., preferred) relative to the CPU, but follow a partial order amongst themselves (Fig. 6.3b).

- **R1:** The language must provide: (a) *Transitive* abstractions that restrict a task's resource-time space based on its dependencies, ensuring that schedulers only need to specify immediate dependencies without considering the broader effects on placement choices of all ancestors and successors. (b) *Commutative* abstractions that merge the resource-time space of tasks without R1, allowing them to execute arbitrarily in parallel.We highlight that a composition of transitive and commutative abstractions enables a concise specification that captures all topological orderings of a job. For example, commutativity captures a parallel execution of t_2 and t_3 from Fig. 6.2, while transitivity constrains the resource time of space of choices due to $t_1 \rightarrow \{t_2, t_3\}$ and requires t_1's placement to finish before t_2 or t_3's placements start (Fig. 6.3c).

6.4.2 STRL++ Expressions

We now discuss the different types of expressions provided by STRL++ using the workload from Fig. 6.2. Schedulers compose these expressions into a STRL++ DAG to concisely capture the placement choices for t_{1-3} in the presence of R1-R3:

❶ `Choose` forms the leaves of the STRL++ DAG. It captures a segment of the resource-time space that contains a combinatorial number of placement options for a task t, all of which start at T_s and consume any k resource units (out of n available). A `Choose` expression is satisfied if one of the $\binom{n}{k}$ placement choices is selected. Fig. 6.4a visualizes the segment of

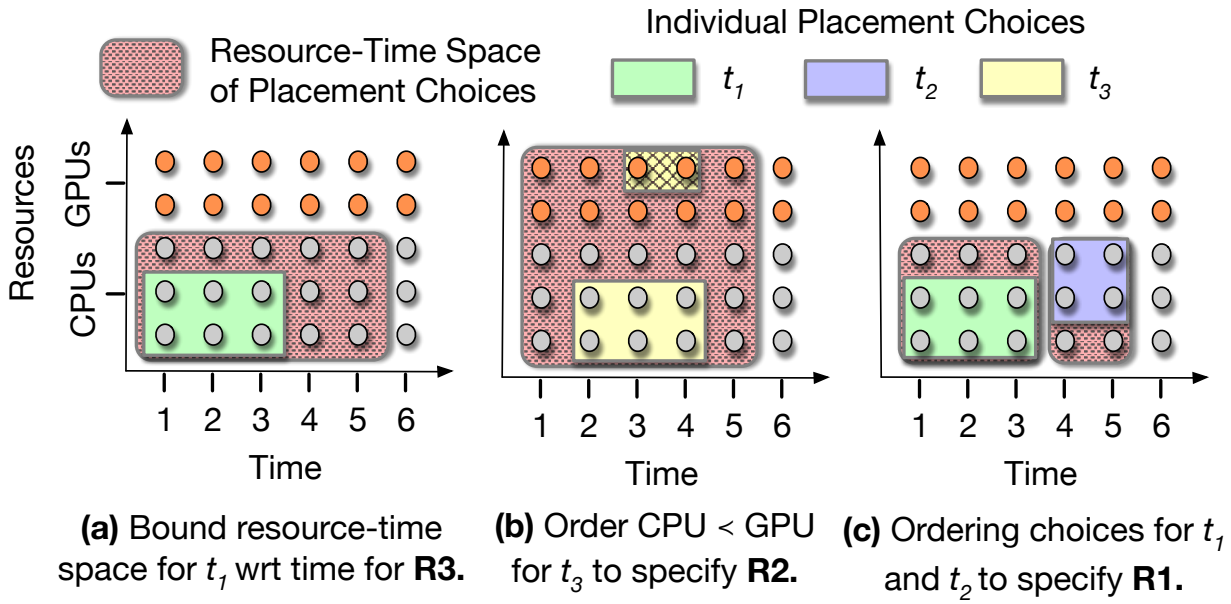


Figure 6.3: A Declarative Language for R1-R3 must provide abstractions for developers to specify the resource-time space of placement choices, and constrain the choices with respect to time to meet R3 and R1. Additionally, developers must be able to partially order the individual units of the space to enable specification of R2.

the space constructed for t_3's placement on a CPU starting at T_s=1, along with a specific choice within the space.

Specifically, `Choose`(t, R, k, T_s, d, w) captures the shape of a rectangle for task t, whose height denotes the usage of k resource units and whose width d denotes the task's duration. In addition, it specifies a segment of the resource-time space where the rectangle is to be positioned by defining the resource type R and the start time T_s. To further reflect preferences (R2), schedulers assign a weight $w \in \mathbb{R}$ indicative of the task's desire to be placed at R starting at T_s. This weight w across all placement choices of a task enables the specification of a partial order, which we discuss in `WeightedAny`.

❷ `WindowedChoose` is a leaf expression that expands the space of a `Choose`'s placement choices across time. Fig. 6.4b shows the extension of the resource-time space of choices from Fig. 6.4a for t_3 until its deadline at $5T$. Additionally, it shows two placement choices for t_3 within this space with $T_s = 1$ or $T_s = 2$, both of which complete within the deadline of $5T$.

In addition to `Choose`'s inputs, it takes a parameter, T_f, that specifies the time by which the choices must complete. Thus, it enables schedulers to bound the resource-time space to include only placement choices that satisfy a task's deadline (R3). It efficiently enumerates $\mathcal{O}(\binom{n}{k} |T_f - d - T_s|)$ placement choices within the time interval $[T_s, T_f - d]$. Like `Choose`, it is satisfied if one of the choices in this space is selected.

❸ `WeightedAny` is a non-leaf expression that enforces the selection of any one placement

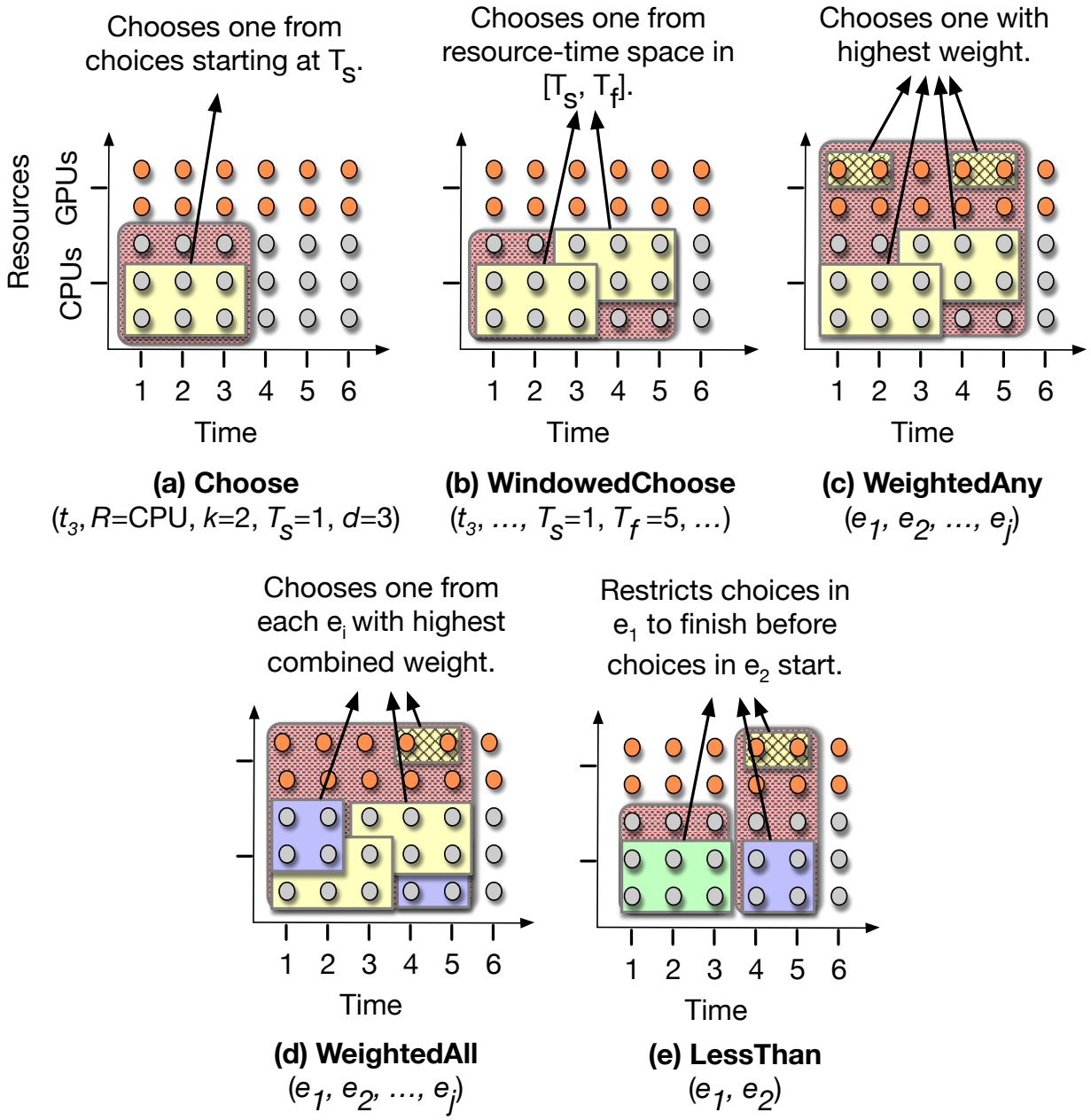


Figure 6.4: Primary expressions in STRL++ that enable schedulers to declaratively specify the tasks and their requirements. For the job in Fig. 6.2, schedulers use `WindowedChoose` to define the resource-time space of placement choices for t_{1-3} that meet their deadline $5T$ (R3). The preference of t_3 for a GPU (R2) is specified by annotating the space with weights and using a `WeightedAny` to select a choice with the highest weight. To specify parallel execution, choices for t_{2-3} are combined with a `WeightedAll` and ordered with t_1 using a `LessThan` to capture R1.

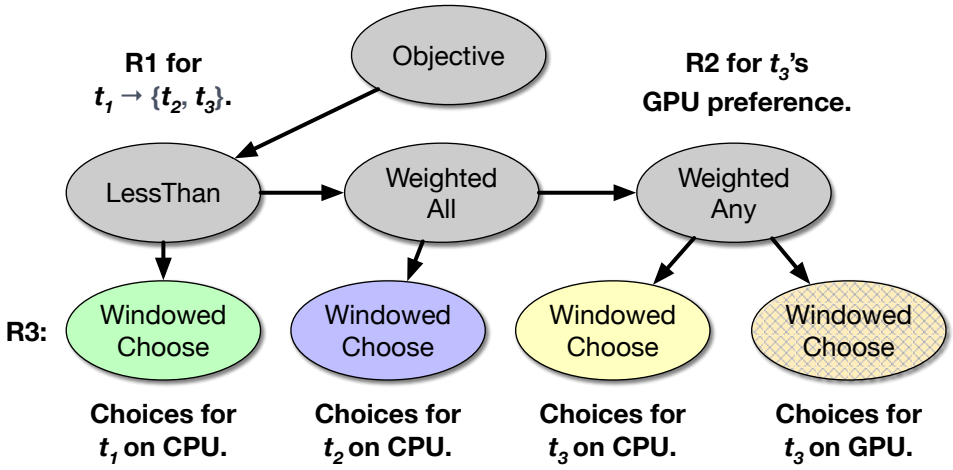


Figure 6.5: STRL++ DAG constructed by a scheduler invocation to specify the tasks shown in Fig. 6.2 and their requirements R1-R3.

choice from its child expressions: e_1, ..., e_j. To make R2-aware decisions, it prioritizes the selection of choices with higher weights reflecting their priority in the partial ordering of the units of the resource-time space.

For instance, to indicate the preference of placing t_3 on a GPU, a scheduler applies `WeightedAny` to two expressions: a `WindowedChoose` with weight w_{cpu} capturing the placement choices for t_3 on the CPU between times 1 and 5, and a `WindowedChoose` with weight w_{gpu} capturing t_3's choices on the GPU. Here, t_3's placement is prioritized on the GPU (if $w_{gpu} > w_{cpu}$) illustrating how a `WeightedAny` satisfies R2.

❹ `WeightedAll` is a non-leaf expression that enforces the selection of a placement choice from each of its child expressions: e_1, ..., e_j. It is a commutative expression such that `WeightedAll`(e_1, ..., e_j) is equivalent to `WeightedAll`(e_j, ..., e_1), enabling schedulers to specify all orderings of the placement choices of tasks that can execute in parallel.

For example, in Fig. 6.2, t_2 and t_3 may execute in parallel, but must begin after t_1 completes. To specify the parallel execution of t_2 and t_3, a scheduler constructs a `WeightedAll` with two children: (a) a `WindowedChoose` enumerating the placement choices of t_2 on the CPU, and (b) a `WeightedAny` containing the two `WindowedChoose` for placing t_3 on the CPU and GPU respectively. As shown in Fig. 6.4d, the `WeightedAll` is satisfied by choosing a placement choice for both t_2 and t_3, while prioritizing the placement of t_3 on the GPU.

❺ `LessThan` is a non-leaf expression that orders the execution of its two children e_1 and e_2 such that the feasible choices for e_1 end before the feasible choices for e_2 start. Crucially, it is transitive such that `LessThan`(e_1, e_2) and `LessThan`(e_2, e_3) imply `LessThan`(e_1, e_3), thus enabling schedulers to only specify each task's immediate dependencies and concisely representing the effects on its placement choices due to R1.

In our example, to specify $t_1 \rightarrow \{t_2, t_3\}$, the scheduler generates a `LessThan` whose left child is a `WindowedChoose` enumerating the placement choices for t_1 on the CPU and whose

right child is a `WeightedAll` capturing the choices of executing t_2 and t_3 in parallel (as discussed above). Fig. 6.4d visualizes the space of choices captured by this `LessThan`.

❻ `Objective` is the root expression of a STRL++ DAG that takes the specification of placement choices for a set of jobs and constructs a joint optimization problem for the solver. Schedulers use an `Objective` expression to collate the individual STRL++ DAGs of requirements for the various independent jobs to be scheduled in a particular invocation. Fig. 6.5 visualizes the DAG that the schedulers must construct to specify R1-R3 for the three tasks t_{1-3} shown in Fig. 6.2.

The above expressions collectively form the set of primary expressions in STRL++ and enable it to concisely capture R1-R3 for any job. However, for practical considerations that arise when dealing with a continuous arrival of jobs with varying priorities, STRL++ provides auxiliary expressions that were useful in the development of our prototype scheduler:

❼ `Scale` is a unary, non-leaf expression that amplifies the weight of its child e. We found it to be useful for two specific cases: (i) prioritizing certain jobs among those available to the scheduler, and (ii) normalizing weights of placing a specific job among other schedulable jobs in an invocation.

During the development of our prototype scheduler, we observed that simply accumulating weights from the children expressions in `WeightedAll` or `LessThan` biased DAGSched towards placement of jobs with more tasks. To counteract this, DAGSched automatically adjusts the weights by integrating `Scale` expressions into the STRL++ DAG to normalize the weights for each job relative to all schedulable jobs.

We emphasize that schedulers only need to generate `Scale` to specify (i), while DAGSched handles (ii) automatically. For example, Fig. 6.6a shows how a scheduler can prioritize the job from Fig 2. by amplifying the weight of the `LessThan` expression that captures the space of feasible choices for t_{1-3}.

❽ `Allocation` is a leaf expression that specifies a fixed placement of a task. Specifically, `Allocation`(t_1, R=CPU, k=2, T_s=1, d=3), shown in Fig. 6.6b, is used to specify a fixed placement of t_1 starting at T_s=1 for $3T$ and taking 2 units of CPU.

We found the `Allocation` expression to be useful in specifying the execution of running tasks that cannot be preempted. Schedulers emit an `Allocation` to inform DAGSched of the placement of such running tasks to: (i) ensure that resources are not oversubscribed, and (ii) R1 can be correctly specified between running and schedulable tasks (i.e., $t_1 \rightarrow \{t_2, t_3\}$ when t_1 is running). This allows DAGSched to constantly re-plan the placements for tasks of in-progress jobs to better meet the scheduler objective during continuous arrivals of jobs.

❾ `MalleableChoose` is a leaf expression that specifies the choice of a fixed number of rectangles of the same size in the resource-time space between T_s and T_f. Specifically, it takes an extra parameter z (in addition to the parameters of a `WindowedChoose`) and is satisfied if z rectangles are allocated, each of whose size is defined by the `WindowedChoose`.

Execution frameworks (notably big data systems such as Apache Spark [338]) parallelize computation through a large number of small tasks that perform the same computation

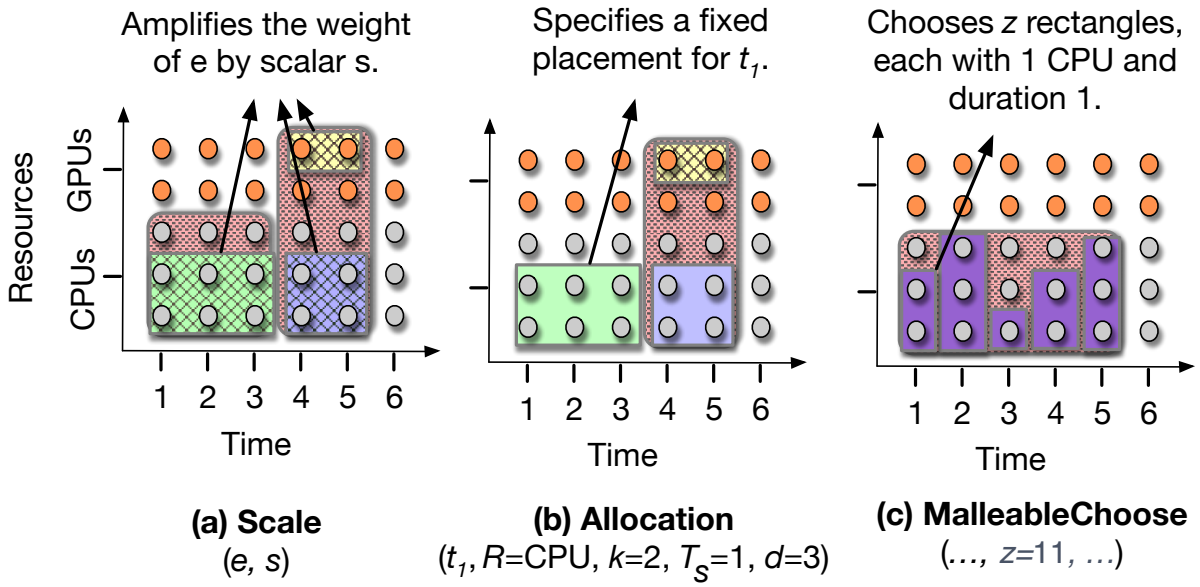


Figure 6.6: Secondary expressions in STRL++ that address practical concerns of specifying: (a) job priorities, (b) non-preemptible tasks, and (c) independent tasks that perform similar computation.

on partitions of the data [336]. As a result, such tasks present equivalent resource and runtime requirements, and can be executed in parallel (i.e., no R1). The specification of such tasks using a `WindowedChoose` and `WeightedAll` is cumbersome and leads to inefficient mathematical models. Thus, `MalleableChoose` enables schedulers to efficiently specify this requirement in STRL++ which collectively places these tasks and constructs a skyline of rectangles as shown in Fig. 6.6.

Decision Variable	Description
T_s	Integer variable for start time of the placement.
T_f	Integer variable for end time of the placement.
I	Indicator for satisfaction of the placement choice.
W	Continuous variable for weight of the placement.

Table 6.2: Decision variables in a `CompiledExpression` used to compile the STRL++ DAG to an ILP model. An expression uses the variables from its children to emit constraints for the ILP, and creates new variables for its parent(s). ERDOS's key insight is to model T_s and T_f using integer decision variables, which reduces the number of constraints required to encode R1 from $\mathcal{O}(MN)$ to $\mathcal{O}(1)$.

6.4.3 Compilation of STRL++

This expressive specification presents an expanded search space that must be carefully and accurately modelled to limit solver runtime without affecting placement decision quality. DAGSched addresses this challenge through two contributions:

Idempotent Compilation. The recursive compilation of a STRL++ DAG begins at the `Objective` expression, which instantiates a shared `ResourceUsage` map to track the usage of resource R at time T. Each expression's compilation generates a `CompiledExpression`, which encapsulates the four decision variables shown in Table 6.2. These variables are used by the expression's parents to emit constraints for an ILP model according to their semantics. Finally, `Objective` emits an optimization objective for the ILP. We concretely specify DAGSched's compilation strategy in Table 6.3.

A critical feature of this compilation is its *idempotency*, i.e., the compilation of a sub-expression generates a unique `CompiledExpression` regardless of the parent invoking it. This ensures an accurate compilation of the STRL++ DAG with arbitrary precedence constraints (R1). For instance, consider the constraints $t_a \rightarrow t_b$ and $t_c \rightarrow \{t_b, t_d\}$. To specify this, a scheduler constructs a single `WindowedChoose` to capture the space of placement choices for t_b and connects it to both t_a and t_c under a `LessThan` expression. Idempotency ensures that the placement variables for t_b are generated only once and used consistently across both constraints. In contrast, the lack of idempotency in prior work [82] results in either: (a) generation of the placement choices for t_b twice, leading to incorrect accounting of resource utilization, or (b) introduction of a false dependency between t_a and t_d, both of which degrade the quality of scheduling decisions by the language.

Integer Domain Modelling of Time. A novel contribution of DAGSched's ILP modelling is to attach integer decision variables for start (T_s) and end (T_f) time with each expression. These variables represent the time at which an expression's selected placement choice(s) begin and end respectively. While leaf expressions (i.e., `Choose`, `WindowedChoose`) provide static values for T_s and T_f, DAGSched automatically emits constraints to compute them for non-leaf expressions, i.e., `WeightedAny`, `WeightedAll` and `LessThan` (Table 6.3).

This crucial insight of attaching T_s and T_f, modelled using integer variables, with each expression enables DAGSched to scale. Specifically, it allows DAGSched to efficiently encode $t_1 \rightarrow t_2$ for M placement choices of t_1 and N choices of t_2 using a single constraint $t_1.T_f < t_2.T_s$ (see Table 6.3). In contrast, prior work [82] models the start and end time of placement choices using an array of indicator variables, with each indicator variable specifying if the placement choice for a particular time was selected. This requires $\mathcal{O}(MN)$ constraints to specify $t_1 \rightarrow t_2$, and was cited as a "key limiting factor for practical use" that led them to abandon solver-based scheduling.

Expression	Compilation	Description
`Objective` $(\{(R_1, n_1), \ldots (R_k, n_k)\})$	$\forall e_c \in e_{child} : e_c$.`compile`();	Recursively compile all children.
	$\forall (R, n)\ \forall T$: `ResourceUsage`$[R][T] \leq n_i$	Ensures that usage of R_i at T never exceeds capacity n_i.
	maximize $\sum e.W$	Maximizes the weighted sum.
`Choose` (t, R, k, T_s, d, w)	$e.T_s = T_s$ $\quad e.T_f = T_s + d$	Start and finish time of choices.
	$e.I = I$ $\quad e.W = w * I$	Weight is w if the expression is satisfied.
	$\forall T \in [e.T_s, e.T_f]$: `ResourceUsage`$[R][T]$ += $k * I$	Allocate k units of R between $[T_s, T_f]$.
`WeightedAny`	$e.I = \sum_{e_c \in e_{child}} e_c.I$	Satisfy up to one child expression.
	$e.T_s = \sum_{e_c \in e_{child}} e_c.I * e_c.T_s$	Start time of the satisfied child.
	$e.T_f = \sum_{e_c \in e_{child}} e_c.I * e_c.T_f$	Finish time of the satisfied child.
	$e.W = \sum_{e_c \in e_{child}} e_c.I * e_c.W$	Weight of the satisfied child.
`WindowedChoose` $(t, R, k, T_s, T_f, d, w)$	-	Automatically decomposed into `WeightedAny`$(\forall T \in [T_s, T_f - d]$: `Choose`$(t, R, k, T, d, w))$
`WeightedAll`	$e.I = I$ s.t. $\sum_{e_c \in e_{child}} e_c.I = \lvert e_{child} \rvert * I$	Enforces the satisfaction if all the children are satisfied.
	$e.T_s = T_s$ s.t. $\forall e_c \in e_{child} : T_s \leq e_c.T_s$	Start time is the earliest start time across all children.
	$e.T_f = T_f$ s.t. $\forall e_c \in e_{child} : T_f \geq e_c.T_f$	Finish time is the highest finish time across all children.
	$e.W = \sum_{e_c \in e_{child}} e_c.W$	Weight is the sum of weights of all the children.
`LessThan`	$e.I = I$ s.t. $e_{left}.I + e_{right}.I = 2 * I$	Enforces the satisfaction if both the children are satisfied.
	$e.T_s \leq e_{left}.T_s$ $\quad e.T_f \geq e_{right}.T_f$	Start and finish time of children.
	$e_{left}.T_f < e_{right}.T_s$	Left finishes earlier than right's start.
	$e.W = e_{left}.W + e_{right}.W$	Sum of weights of the two children.
`Scale` (s)	$e.W = s \times e_{child}.W$	Modifies the weight.
`Allocation` (t, R, k, T_s, d, w)	$e.T_s = T_s$ $\quad e.T_f = T_s + d$	Start and finish time of placement.
	$e = 1$ $\quad e.W = w$	Weight is 1 as expression is satisfied.
	$\forall T \in [e.T_s, e.T_f]$: `ResourceUsage`$[R][T]$ += k	Allocate k units of R between $[T_s, T_f]$.

Table 6.3: DAGSched's strategy for compiling STRL++ expressions into an ILP model. A parent expression recursively compiles its children and emits the specified constraints using the four decision variables from Table 6.2. Finally, the `Objective` expression ensures that no resource is oversubscribed at any time, collates the weights from its children and constructs an optimization objective for an off-the-shelf ILP solver.

6.5 Optimization of STRL++ DAGs

Our decision to design STRL++ as a task-focused specification language raises a critical challenge for scheduler scalability. While the task-focused specification simplifies scheduler development, it captures a combinatorial search space of placement choices in the presence of R1-R2. A literal compilation of these choices generates intractable models (even with efficient encodings, like the one discussed in §6.4.3).

This section discusses STRL++'s approach to addressing this challenge: *time-aware expressions*. §6.5.1 discusses how STRL++ annotates each expression with information about its captured resource-time space of choices, facilitating an effortless development of a rich class of optimizations that can efficiently prune the search space. §6.5.2 provides key optimizations implemented by DAGSched atop time-aware expressions, that have been essential in ensuring its scalability.

6.5.1 Interface: *Time-Aware Expressions*

A composition of STRL++ expressions constrains the space of feasible task placement choices to reflect R1-R3. However, a naive compilation of this specification passes the combinatorial complexity to the solver by generating mathematical encodings for the entire set of placement choices for each task, along with the R1-R3 constraints. . This raises two challenges: (*i*) it generates models with a large number of variables and constraints leading to significant compilation bottlenecks, and (*ii*) the solver must sift through an expanded combinatorial search space, which negatively affects solver runtime and significantly hinders scheduler scalability.

The goal of STRL++ is to programmatically expose the effects of these constraints on the set of feasible choices captured by an expression. This seeks to enable the development of optimizations that can make efficient modelling decisions and prune the search space by reasoning about the effects of the constraints before invoking the underlying solver.

STRL++ achieves this by making expressions *time-aware*, i.e., it exposes the method `getTimeBounds`, which outputs a tuple (`start`, `end`) representing the feasible time ranges for the start and end of the choices captured by an expression. Specifically, the range is captured through a lower bound (`lb`) representing the earliest time that the choices can start (or end) and upper bound (`ub`) representing the latest time that the choices can start (or end). In addition, `updateTimeBounds` takes the (`start`, `end`) bounds and removes the infeasible choices that do not fall within the updated bounds.

For example, for a `WindowedChoose` that captures the choices within $[T_s, T_f]$, `start` provides the time bounds on the start time of choices, i.e., $[T_s, T_f - d]$, where d is the task's duration. This conveys that no placement choice can start before `lb`$= T_s$ or after `ub`$= T_f - d$. Similarly, `end` can vary between $[T_s + d, T_f]$, signifying that the earliest placement choice finishes at `lb`$= T_s + d$, while the last finishes at `ub`$= T_f$.

```
   Input: STRL++ DAG
1  for e ∈ DFS_PostOrder(DAG) do
2      if e.type=LessThan then
3          l_bounds ← e_left.getTimeBounds();
4          r_bounds ← e_right.getTimeBounds();
5          if r_bounds.start.lb < l_bounds.end.lb then
6              // e_right must start after the earliest e_left ends
7              r_bounds.start.lb = l_bounds.end.lb;
8              e_right.updateTimeBounds(r_bounds);
9          end
10         if l_bounds.end.ub > r_bounds.start.ub then
11             // e_left must end before latest e_right starts
12             l_bounds.end.ub = r_bounds.start.ub;
13             e_left.updateTimeBounds(l_bounds);
14         end
15     end
16 end
```

Algorithm 2: Critical Path optimization purges placement choices that are rendered infeasible due to precedence ($\rightarrow$) using STRL++'s time-aware expression `LessThan`.

6.5.2 DAGSched's Optimizations

We find STRL++'s time-aware expressions to be a powerful abstraction to enable two key classes of optimization passes: *fidelity-preserving* and *fidelity-altering*. *Fidelity-preserving* optimizations provide a sound analysis of the STRL++ DAG, and purge placement choices that are deemed infeasible. Conversely, *fidelity-altering* optimizations make modelling decisions that improve solver runtime with minimal impact to placement decision quality. This section details three optimizations provided by DAGSched spanning these two classes.

Critical Path Optimization

The Critical Path optimization is a fidelity-preserving optimization that removes placement choices rendered infeasible by the cumulative effects of a task's dependencies. For example, the start time of 2 in Fig. 6.2 is infeasible for t_1 due to $t_1 \rightarrow t_2$ and can be purged.

STRL++'s time-aware expressions significantly ease the implementation of this optimization (Algorithm 2). The optimization does a post-order depth-first traversal of the STRL++ DAG, and purges placement choices from the children e_{left} and e_{right} of a `LessThan` using two conditions: (i) Line 5 ensures that the earliest placement choice for e_{right} should only start after the earliest placement for e_{left} ends, and (ii) Line 10 ensures that the last choice for e_{left} should end before the last placement for e_{right} starts. The optimization computes the updated bounds for e_{left} and e_{right}, and uses STRL++'s `updateTimeBounds` to recursively purge the placement choices due to the new bounds in both children.

```
   Data: STRL++ DAG
 1 Procedure NonOverlappingExpressions
 2 |  NonOverlapExprs ← {};
 3 |  for e ∈ DFS_PostOrder(DAG) do
 4 |  |  if e.isLeaf() then
 5 |  |  |  // A choice e cannot overlap with itself.
 6 |  |  |  NonOverlapExprs [e] ← {e}
 7 |  |  end
 8 |  |  else if e.type=WeightedAny or e.type=LessThan then
 9 |  |  |  //  Mutually exclusive so no children can overlap.
10 |  |  |  children ← e.getChildren();
11 |  |  |  for e_child ∈ children do
12 |  |  |  |  for e_other ∈ children − e_child do
13 |  |  |  |  |  NonOverlapExprs [e_child] ∪ NonOverlapExprs [e_other];
14 |  |  |  |  end
15 |  |  |  end
16 |  |  end
17 |  end
   Input: STRL++ DAG
18 MaxUsage ← {};
19 for e ∈ Leaves(DAG) do
20 |  bounds ← e.GetTimeBounds();
21 |  // i is a unique identifier for E
22 |  (i, E) ← NonOverlappingExpressions(e);
23 |  for t ← [bounds.start.lb, bounds.end.ub] do
24 |  |  // Get max usage of non-overlapping expressions.
25 |  |  u_i ← 0;
26 |  |  for e_j ∈ E do
27 |  |  |  u_i ← max(u_i, e_j.GetResQty());
28 |  |  end
29 |  |  MaxUsage [(i, t)] ← u_i;
30 |  end
31 end
32 for t ∈ ResourceUsage do
33 |  maxUsageAtTime ← ∑_i MaxUsage [(i, t)];
34 |  if maxUsageAtTime ≤ n then
35 |  |  remove ResourceUsage[t];
36 |  end
37 end
```

Algorithm 3: Resource Constraint Purging tracks the usage of each resource at t by computing the maximum quantity of resources required by each set of expressions with non-overlapping placement choices E. If the resource cannot be over-subscribed at t, DAGSched does not emit capacity constraints for t.

Resource Constraint Purging

DAGSched must ensure that the placements of the tasks do not violate resource quantities, i.e., for a resource R_i with a total quantity n_i, the usage of a set of tasks running at each time on R_i should not exceed n_i. DAGSched achieves this by tracking the usage of a resource at each time through a `ResourceUsage` map. Leaves of a STRL++ DAG that efficiently enumerate the placement choices (i.e., `Choose` and `WindowedChoose`) register their usage of a resource R at a particular time T by adding $k \times I$, where I indicates the selection of the choice, to `ResourceUsage`$[R][T]$ (as shown in Table 6.3).

To ensure that the resource quantities are not violated, a literal translation of STRL++ needs to emit $\mathcal{O}(|R||T|)$ constraints. However, the specification of R1-R2 ensures that some of these constraints will never be violated. For example, consider t_1 and t_2 from Fig. 6.2 that each require 2 CPUs and must must be placed atop a cluster with 3 CPUs. The constraint $t_1 \rightarrow t_2$ ensures that selected placement choices for t_1 and t_2 can never overlap. This ensures that for any given time, only one of t_1 and t_2 can run, and an individual usage from these tasks cannot violate the resource quantity of 3 CPUs.

Algorithm 3 depicts DAGSched's fidelity-preserving optimization to remove such superfluous constraints. It computes the sets of expressions that present non-overlapping placement choices with each expression e, and tracks the maximum usage of a specific resource across all these choices. Finally, it ensures that the constraint to prevent resource oversubscription at time t is only emitted to the solver if the sum of the maximum usage across all sets of non-overlapping choices is higher than the quantity n.

Dynamic Discretization

The Dynamic Discretization optimization is a fidelity-altering optimization that aims to remove the finer-grained placement choices based on the workload dynamics. The key insight in this optimization pass is to remove such placement choices from the resource-time space where the resources are less contended. Resources remain less contended when the arrival rate in the workload becomes low. Under such conditions, making coarse-grained decisions can vastly reduce solver time with little or no effect on decision quality.

To remove fine-grained placement choices, this pass varies the discretization of the resource-time space. A lower discretization renders fine-grained placement choices, whereas, higher discretization leads to coarse-grained choices. It leverages STRL++'s time-aware expressions to calculate the resource contentions over the entire space. Specifically, this pass iterates over leaf STRL++'s expressions and adds its requested resource to the resource contention map for the time given by the `getTimeBounds` method. Then, this pass uses a simple linear decay method that predicts a higher discretization for low resource contention and vice-versa. Using this function, it predicts a discretization for each contended resource. Based on the predicted discretization, it removes all fine-grained child expressions from `WeightedAny` expressions that lie within the discretization window. The algorithm is detailed in Algorithm 4.

Input: STRL++ DAG
1 $ResContention\,[t] = 0\ ,\ \forall\ t \in [0, T]$
2 // calculate resource contention map
3 **for** $e \in LEAVES(DAG)$ **do**
4 $\quad bounds \leftarrow e.$GetTimeBounds()
5 $\quad$ **for** $t \leftarrow [bounds.start.lb, bounds.end.ub]$ **do**
6 $\quad\quad ResContention\,[t]$ += $e.$GetResQty()
7 $\quad$ **end**
8 **end**
9 // decay fn that outputs a discretization for res contention (r)
10 $F(r) = \text{Max}(MaxDisc - k * r, MinDisc)$ // k is decay rate
11 $PredGranularity = [\ ]$
12 **for** $t \in ResContention$ **do**
13 $\quad predDisc = F(ResContention\,[t])$
14 $\quad$ **while** $predDisc \neq t + 1$ **do**
15 $\quad\quad AvgContention = \frac{\sum_{t}^{t+predDisc} ResContention[t]}{predDisc}$
16 $\quad\quad$ **if** $predDisc == F(AvgContention)$ **then**
17 $\quad\quad\quad PredGranularity.$append$((t, t + predDisc))$
18 $\quad\quad\quad$ break
19 $\quad\quad$ **end**
20 $\quad\quad predDisc$ -=1
21 $\quad$ **end**
22 **end**
23 // remove fine-grain exprs
24 **for** $t_s, t_f \in PredGranularity$ **do**
25 $\quad$ **for** $e_{max} \in MaxOverLEAVES(DAG)$ **do**
26 $\quad\quad cnt \leftarrow 0$
27 $\quad\quad$ **for** $e \in e_{max}.ChildWithTimeBound(t_s, t_f)$ **do**
28 $\quad\quad\quad$ **if** $cnt \neq 0$ **then**
29 $\quad\quad\quad\quad e_{max}.$RemoveChild$(e)$
30 $\quad\quad\quad$ **end**
31 $\quad\quad\quad cnt$ +=1
32 $\quad\quad$ **end**
33 $\quad$ **end**
34 **end**

Algorithm 4: Dynamic Discretization optimization automatically decides the granularity of placement choices based on resource contention and prunes finer-grained `Choose` expressions that do not conform to the chosen granularity.

6.5.3 Implementation

DAGSched is an open-source framework implemented in $7k$ lines of C++. We instantiate an open-source scheduler for Apache Spark [338] using DAGSched, and refer to it as Spark-DAGSched in the remainder of the paper. Below, we discuss our implementation of DAGSched (§6.5.3) and Spark-DAGSched (§6.5.3).

DAGSched

DAGSched is provided as a dynamic library that schedulers attach into their process space. The library provides both C++ and Python interfaces for constructing STRL++ DAGs, invoking optimizations and retrieving `Placements` (see Fig. 6.2).

We highlight the key features of our implementation below:

Parallel Compilation. DAGSched exploits the natural decomposition of a STRL++ DAG to automatically parallelize the compilation process – a significant bottleneck in earlier implementations of DAGSched, sometimes taking upwards of 1s.

Selective Rescheduling. We explore ideas from [207] to allow schedulers to specify η, such that DAGSched randomly drops the replanning of η% of the jobs in each scheduling cycle by converting their `WindowedChoose` expressions into `Allocation` expressions. §6.6 explores its effects on the placement decision quality and the runtime of DAGSched.

Interaction with Multiple Solvers. DAGSched implements a shim layer mimicking various off-the-shelf solvers, and automatically interfaces with either of CPLEX [148], Gurobi [135] or Google OR-Tools [239] based on availability.

Warm Starts. Since solvers may spend a significant amount of time finding initial solutions [134], DAGSched provides initial values for the variables retrieved from prior invocations. This enables the solver to begin its search from a feasible, but suboptimal solution, thus significantly reducing solver runtime.

Objective-Based Interrupts. Solvers repair the initial feasible solution provided by DAGSched, and are able to heuristically construct an optimal solution to the ILP model for a significant fraction of our scheduler invocations. However, to terminate, the solver must prove the solution's optimality – an expensive process. DAGSched uses STRL++'s `Objective` to automatically construct an upper bound on the achievable objective value, and interrupts the solver as soon as the objective is reached.

Spark-DAGSched

We develop a scheduler for Apache Spark [338], Spark-DAGSched, atop DAGSched in 2.6k lines of Python. Our implementation executes in Spark's standalone cluster mode, and allocates the available executors (i.e., the smallest resource unit) across the tasks. To achieve this, Spark-DAGSched interacts with Spark via gRPC [129], and executes the following steps:

1. Applications arrive at the Spark Master with a deadline and are compiled into a DAG. It then communicates the DAG along with the deadline to Spark-DAGSched via RPC.

2. Spark-DAGSched retrieves the job's execution profiles and constructs a STRL++ DAG. It then invokes DAGSched and retrieves the `Placements` mapping the tasks of the DAG to the number of executors along with a start time.

3. The typical workflow in Spark standalone involves the Master allocating executors to an application's DAGScheduler, which allocates them to its tasks (i.e., stages). We modify this workflow such that the DAGScheduler regularly polls Spark-DAGSched, which informs it of the number of executors to allocate to a stage at the start time computed by DAGSched.

4. The DAGScheduler in Spark requests the specific number of executors from Master, and executes the specified tasks.

5. Finally, upon task completion, the DAGScheduler returns the executors, and notifies Spark-DAGSched of the completion.

6.6 Evaluation

We evaluate the benefits of DAGSched through Spark-DAGSched, an open-source scheduler for Apache Spark [338] built using DAGSched, on a real-world Spark cluster and in simulation. We refer the interested reader to the implementation details of Spark-DAGSched's integration with Spark to §6.5.3. Our key metric of success is *goodput* maximization, i.e., the number of jobs that meet their deadline, or minimization of *deadline misses*. Our evaluation answers three key questions:

1. Can Spark-DAGSched scale to realistic workloads while providing better placement decisions than prior works? (§6.6.1)

2. Does the ability of DAGSched to make R1-R3-aware placement decisions lead to quantitative benefits? (§6.6.2)

3. How well does Spark-DAGSched perform under periods of stress – constrained clusters and tighter deadlines? (§6.6.3)

4. Do DAGSched's automatic optimization and compilation strategies help reduce scheduler runtime? (§6.6.4)

Experimental Setup. We instantiate a 20-node CloudLab [98] cluster using c6420 machines as worker nodes for Spark, and run Spark-DAGSched on a GCP c2d-highcpu-32 VM running Ubuntu 22.04 and Gurobi 11.0. We consider the following scheduler baselines:

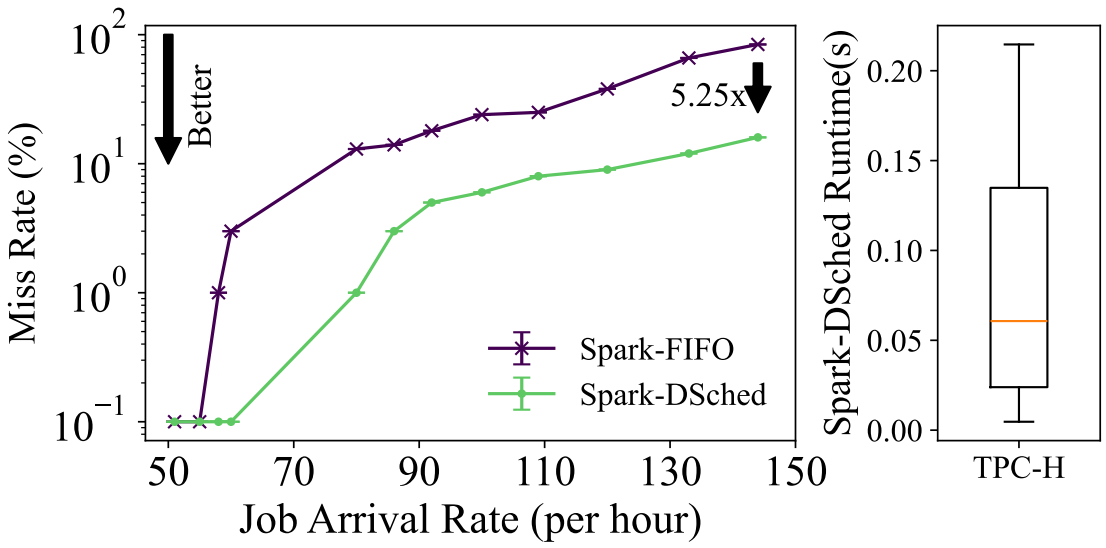


Figure 6.7: Spark-DAGSched scales to real workloads and achieves up to 5.25× higher goodput on a workload of 100 TPC-H jobs. DAGSched's optimizations help keep the latency under 225ms.

1. Spark's FIFO scheduler that allocates resources based on a task's arrival time, and is a non R1-R3-aware heuristic.

2. An EDF scheduler that prioritizes tasks based on their deadlines (R3), greedily tries to satisfy R2, and is R1-unaware (i.e., it only schedules among the ready tasks).

3. TetriSched [310], an R2-R3-aware scheduler that packs among the set of ready tasks (i.e., is R1-unaware). We perform a sweep of its hyper-parameter, *plan-ahead*, which decides how long to plan task placements into the future, and report the most favorable results to TetriSched.

4. A faithful re-implementation of Graphene's two-step scheduling [125] where: (i) the offline stage finds the most efficient topological ordering of each job on the cluster using an ILP solver, and (ii) the online stage packs among the set of ready tasks of each job using an ILP solver.

We evaluate these baselines on a faithful discrete time simulator written in 7k lines of Python code using a workload from the real-world Alibaba trace [19]. Additionally, we compare Spark-DAGSched to Spark's default FIFO scheduling on our Spark cluster using the TPC-H [249] workload.

6.6.1 Can DAGSched scale?

We first evaluate Spark-DAGSched's ability to provide quantitative benefits on its metric of success, *goodput*, on real workloads. For the workload, we randomly sample 100 TPC-H

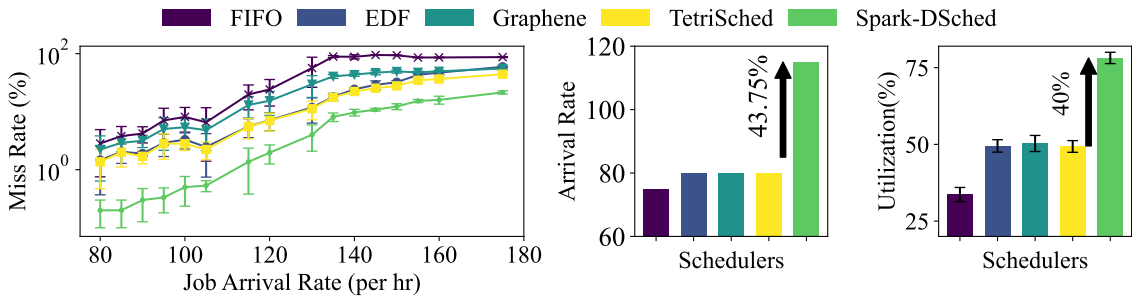


Figure 6.8: Benefits of R1, R3 awareness (a) Spark-DAGSched is compared by varying arrival rates, achieves 2x less deadline misses on higher arrival rates. (b) Spark-DAGSched serves 43.75% more requests per hour, and (c) has 40% more cluster utilization at 99% goodput on jobs (DAGs) derived from the Alibaba Trace. Spark-DAGSched outperforms baselines as DAGSched captures both **R1** and **R3** effectively.

jobs from all 22 queries and an input size of 250GB, resulting in jobs that vary in their execution time from 2 minutes to 10 minutes. These jobs are submitted to our Spark cluster using a Poisson process, with the inter-arrival time varying from 25 to 70s. We emphasize that these conditions are similar to prior works [125, 198], and hence, represent real workloads that Spark-DAGSched may be required to schedule.

Fig. 6.7 (left) compares Spark-DAGSched to Spark's default FIFO scheduling policy, and plots the *deadline miss rate*, i.e., the % of jobs that miss their deadline, as the jobs arrive more frequently (i.e., inter-arrival time between jobs decreases). We highlight two key results pertaining to Spark-DAGSched's ability to meet deadlines: (*i*) Spark-DAGSched can maintain a 38% higher job arrival rate than Spark's FIFO scheduler while meeting 99% of job deadlines. (*ii*) Spark-DAGSched can meet upto 5.25× more deadlines under periods of high cluster load. Thus, Spark-DAGSched provides significant benefits in goodput, under both normal cluster utilization, and under stress.

Fig. 6.7 (right) highlights Spark-DAGSched's efficacy in addressing the second critical concern: the ability to make scheduling decisions quickly. DAGSched's effective optimizations and modelling enable Spark-DAGSched to maintain a p90 latency of under 225ms, leading to a sub 0.2% scheduling latency of the length of the shortest job.

Takeaway: *Spark-DAGSched meets significantly more deadlines in real deployments – managing up to* 5.2× *higher goodput – while maintaining a p*90 *latency of under* 225*ms.*

6.6.2 Does R1-R3 awareness benefit DAGSched?

We now establish that DAGSched's ability to collectively consider R1-R3 in making its placement decisions leads to significant quantitative benefits in meeting deadlines. We carry out two independent experiments to isolate the efficacy of R1-R3 using a common workload derived from the Alibaba trace [19].

Our workload consists of 1000 random jobs sampled from the Alibaba trace being exe-

cuted atop a cluster with 30k executors. We choose these values to remain faithful to the experiment settings of prior works [198]. Further, each job is randomly assigned a deadline between 1.1× to 2.5× its critical path — a value we define as the *deadline factor*.

Increasing Job Arrivals. Our first experiment isolates the efficacy of considering R1 and R3. We compare Spark-DAGSched with our baselines on increasing arrival rates from 80-175 jobs per hour (as a Poisson process). Intuitively, we expect R3 aware schedulers to shine in periods of low cluster utilizations as efficient packing of jobs matters less and execution of jobs must be prioritized by their deadlines. However, as the cluster load increases under increased job arrivals, the consideration of both R1 and R3 is critical as efficient schedules require prioritization of tight deadlines and an efficient utilization of the contended resources through better packing using R1.

Fig. 6.8 shows the results of our experiment. As expected, under lower arrival rates when packing of jobs is not the critical concern, R3-aware schedulers like EDF and TetriSched achieve high goodput, and maintain a lower miss rate than just R1-aware schedulers like Graphene. Importantly, even under low arrival rates, DAGSched achieves a higher goodput than just R3-aware schedulers. However, DAGSched's ability to be R1-aware really shines, as it is able to sustain an up to 43.75% higher job arrival rate while meeting 99% of the job deadlines. The efficacy of R1-aware packing here is evidenced by the fact that DAGSched can sustain 99% goodput with a high cluster utilization of 75%, while R1-unaware baselines require a 40% lower cluster utilization for the same, signifying their inefficient utilization of resources. As the job arrival rate and the load in the cluster increase, DAGSched benefits from R1-R3 awareness by missing 2× less deadlines than TetriSched. Tetrisched misses out on the opportunity to pack tasks efficiently, signifying the benefits of capturing R1.

For the remainder of the experiments, we choose to show results for an arrival rate of 135 jobs per hour, as it achieves a 90% goodput under 86% cluster utilization, and is a good representative of the stable state of our dagsched.

Varying Resource Heterogeneity. We now additionally evaluate the benefits of R2-awareness to DAGSched's metric of success. We modify our cluster and convert 5k executors (from a total of 30k) to provide a 50% reduction in task runtimes, and vary the heterogeneous factor – the percentage of tasks that have preferences for these 5k executors. The deadlines are determined based on the runtime of executing the critical path on their preferred resources. We expect Spark-DAGSched's gains to considerably increase as its R2-R3 awareness can better identify the packing opportunities for tasks.

We skip the comparison with Graphene as its offline step cannot be extended to be R2-aware. At a low heterogeneity factor, the majority of jobs have relaxed deadlines and only a minority of tasks have heterogeneous preferences. Therefore, schedulers can simply place tasks over either of the resources and still meet the deadline. Therefore, we observe that all the schedulers (including R2-unaware schedulers like EDF, FIFO) perform better at low heterogeneous factors. At higher heterogeneity factor, however, job deadlines are tighter, with the majority of tasks having a preference. Tighter deadlines require schedulers to prioritize urgent jobs and an increase in the preference for 5k executors demands optimal

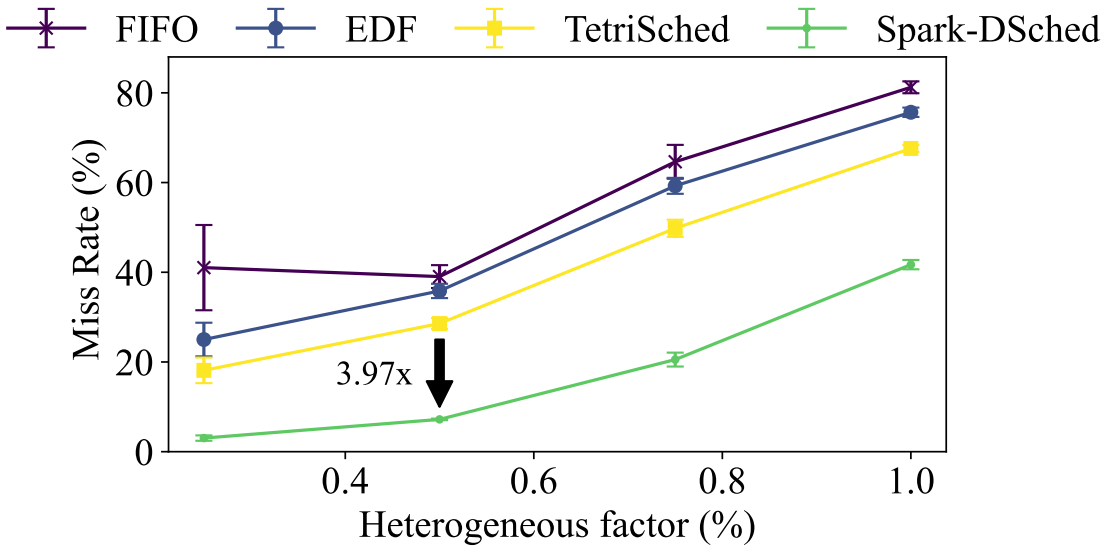


Figure 6.9: Benefits of R2-awareness. Spark-DAGSched achieves up to 3.97× lower deadline misses than the baselines.

packing of tasks. We, therefore, observe that the schedulers that are R2 and/or R3 unaware perform worse with the increase in the heterogeneous factor. TetriSched, though being R2-aware misses out on optimally packing tasks as it is R1-unaware. Spark-DAGSched has the fewest deadline misses at high heterogeneous factor, as it is both R1 and R2 aware.

Takeaway: *Spark-DAGSched benefits from R1-R3 aware decisions, achieving* 2× *reduced deadlines misses and sustaining a* 43.75% *increased job arrival rate to meet* 99% *of job deadlines. R2-consideration leads to higher gains– up to* 3.97× *reduction in deadline misses.*

6.6.3 How well does DAGSched handle stress?

We now stress test DAGSched to understand how its performance degrades.

Constrained Deadlines. We subject Spark-DAGSched and our baselines to reduced deadline factors from 3× to 1.1×. For the entire experiment, we keep both arrival rate and resources constant at 135 jobs per hour and 30k executors respectively. We believe that the tighter deadlines will require better scheduling, and expect Spark-DAGSched to shine there.

Fig. 6.10 compares Spark-DAGSched with the baselines on the varied deadline factor. Even under tighter deadlines (1.1x deadline factor), Spark-DAGSched miss rate remains lower than 10%. This demonstrates that DAGSched succinctly captures R3 preferences. Moreover, Spark-DAGSched consistently outperforms baselines. Under relaxed deadlines, the schedulers simply pack tasks efficiently and still meet deadlines. Therefore, we observe that the R3 unaware scheduler like Graphene tends to perform better under relaxed deadlines in Fig. 6.10. Under tighter deadlines, prioritizing jobs with tighter deadlines matters, and R3 unaware schedulers start performing worse. Moreover, under tighter deadlines, packing tasks based on their dependencies also matters as it provides a better task ordering to satisfy

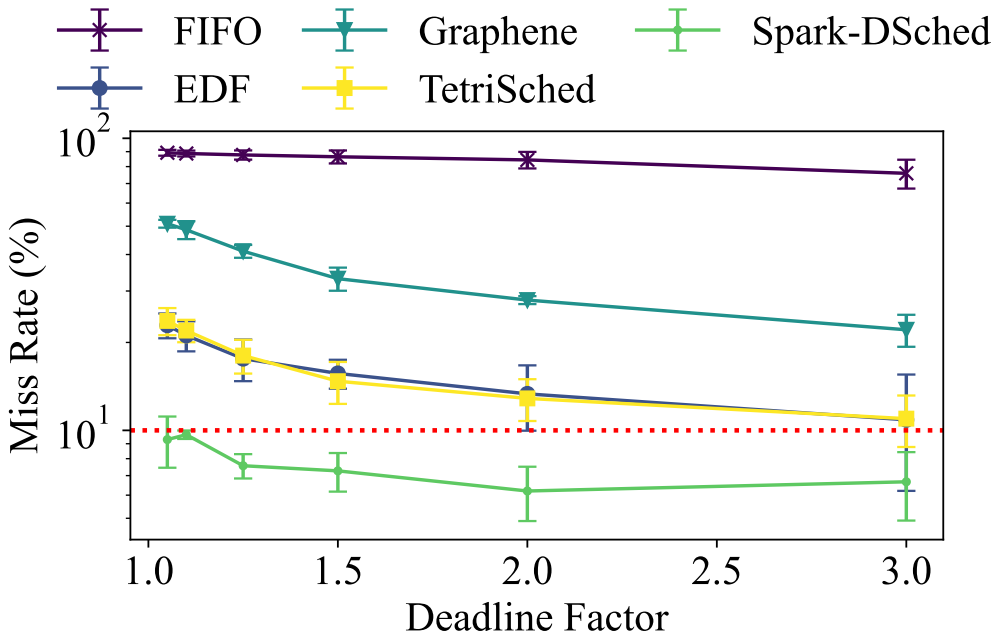


Figure 6.10: Comparison on Varying Deadlines. Spark-DAGSched is compared with the baselines by varying deadlines with the deadline factor. Spark-DAGSched achieves lower deadline miss rate than baselines. It violates a mere 10% of the jobs with extremely tight deadlines (1.1x deadline factor).

jobs' end-to-end deadlines. Therefore, Spark-DAGSched performs better than R1-unaware schedulers TetriSched and EDF under tighter deadlines.

Constrained Cluster. We simulate a constrained cluster by lowering the cluster size from 30k to 5k, with all other settings remaining the same (arrival rate fixed at 135 jobs/hour and job deadlines sampled from 1.1-1.25 deadline factor). Spark-DAGSched still outperforms baselines, achieving the highest goodput. We expect Spark-DAGSched gains to increase further under tighter cluster size conditions, as a more constrained cluster demands efficient packing of tasks from the scheduler.

Fig. 6.11 compares Spark-DAGSched with the baselines on different cluster sizes. Spark-DAGSched's gains in goodput increase with tighter cluster sizes. It achieves upto 6.42x better goodput than the baselines. Spark-DAGSched achieves better performance as DAGSched makes it R1-R3 aware. Specifically, considering all three requirements together provides better packing opportunities to Spark-DAGSched. Schedulers that pack without deadline awareness (e.g., Graphene) achieve less goodput as they don't prioritize urgent jobs. TetriSched is R3-aware but doesn't consider R1, missing opportunities for better packing.

Takeaway: *Spark-DAGSched gracefully degrades under conditions of extreme stress–managing to sustain 90% goodput with tight deadline factor of* 1.1×*, and upto* 6.42× *better goodput under constrained cluster sizes.*

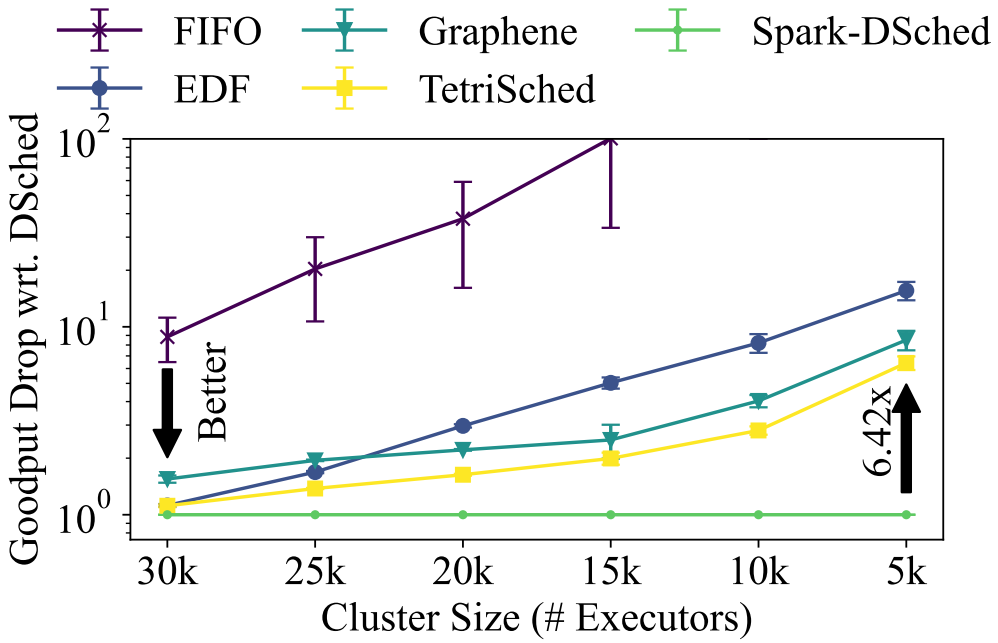


Figure 6.11: Comparison under Resource Constraints. Spark-DAGSched is compared with the baselines under resource constraints by reducing the cluster size from 30k to 5k. DAGSched consistently outperforms the baselines and is upto 6.42x better in goodput.

6.6.4 Do DAGSched's Optimizations Help?

We now assess the efficacy of DAGSched's proposed optimization. We ask whether these optimizations reduce the scheduler runtime. To provide attribution of benefits to each of the optimizations passes, we incrementally add optimization to Spark-DAGSched's STRL++ DAG and compare scheduler runtime and goodput. The comparison is done for the arrival rate of 135 jobs/hr (where DAGSched reaches 85% cluster utilization). To give a fair chance to DAGSched run with no optimizations, the scheduler runtime limit is kept 60s (10% of the jobs critical path). Fig. 6.12 compares DAGSched run with no optimization passes (DAGSched NoOpt) to incrementally added optimization passes that include critical path (+CP) and dynamic discretization (+Dyn) optimization pass. We add one more optimization (+SEL) derived from prior work [207] which allows Spark-DAGSched to specify η, such that DAGSched randomly drops the replanning of η% of the jobs in each scheduling cycle.

Our key finding is that the optimization passes not only reduce the solver runtime but also increase goodput. Specifically, with the dynamic discretization added as the optimization pass DAGSched reaches 96% goodput, increases goodput by 11% and reduces solver runtime's 90th percentile latency by 6x compared to DAGSched run with no optimization. The increase in goodput comes as the search space reduced by the DAGSched's optimization passes reduces the presolve time and allows solvers to spend more time finding better solutions.

Takeaway: *DAGSched's optimizations reduce solver runtime with 60x reduction in p90 latency and increase goodput by 5.29% compared to DAGSched with no optimization passes.*

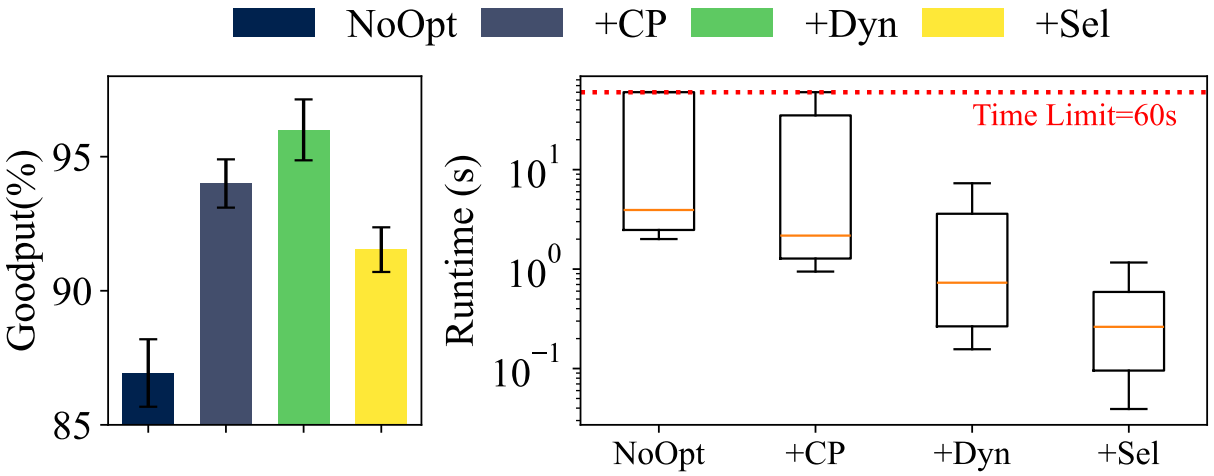


Figure 6.12: Evaluating DAGSched's Optimization Passes. DAGSched's optimization passes (a) increase goodput of the scheduler by upto 1.1x under 60s time limit, and (b) provide tighter/better runtime distributions and exceed no more than a second (for +SEL).

6.7 Related Work

Heuristic-Based Schedulers like Graphene [125], Decima [198], Tetris [126], DRF [115] provide selective support for R1-R3, and often trade-off decision quality for efficiency.

Solver-Based, Non-Declarative Schedulers. Quincy [154] and Firmament [118] model the scheduling problem as a min-cut, max-flow network graph, but crucially cannot support either of R1-R3. Other solver-based workers that model the scheduling problem as ILPs, e.g., Gravel [206] develop hand-crafted mathematical models that do not support R1, R3.

Solver-Based, Declarative Schedulers. Alsched [309] and DCM [287] propose languages with no notion of time, and thus unable to express deadlines (R3) or task orderings (R1). RDL [82] and STRL [310] provide similar abstractions to construct the resource-time space of placement choices. However, STRL does not support ordering constraints (R1), and RDL has no R2 awareness, unable to partially order the choices according to task preferences. Furthermore, specifying R1 for complex jobs in RDL requires introduction of false task dependencies. This prevents RDL from capturing all topological orderings of the job affecting placement decision quality.

6.8 Conclusion

We have proposed a declarative scheduling framework DAGSched, whose specification language STRL++ enables concise yet expressive specification of R1-R3. We instantiate a scheduler for Apache Spark using DAGSched, and evaluate it on a production trace, achieving a 43.75% improvement in job arrival rate for 99% deadline attainment, and up to 6.42× increase in deadline attainment under high load.

Chapter 7

Conclusions

This dissertation explored a clean-slate design for a synergistic development of modular AV pipelines in tandem with the "execution systems" orchestrating their execution on the constrained resources in the AV. We observe and underscore the need for AVs to adhere to *dynamically-varying* deadlines that are dictated by the environment around the AV. To help achieve this, we proposed a new programming model, D3, that centralizes deadline management and exploits the presence of a multitude of solutions for each component in the AV decision-making computational pipeline to meet dynamic deadlines.

We show how to realize the D3 model through novel extensions to concepts from streaming data systems in ERDOS. ERDOS exposes fine-grained execution events and allows the enforcement of dynamic daedlines between any two events. To help achieve these deadlines, ERDOS crucially provides speculative execution, whose efficient and timely execution is enabled by two key scheduling techniques: SuperServe and DAGSched. In SuperServe, we develop a fine-grained inference serving system for ML models that unlocks a resource-efficient serving of the entire range of ML models spanning the latency-accuracy tradeoff space. DAGSched proposes a declarative language, STRL++, and unlocks an efficient multiplexing of the available compute resources amongst the decision making components, while maximizing the resource utilization and attainment of deadlines.

Finally, we address the crucial lack of AV benchmarks by providing the first completely open-source AV pipeline, Pylot, and use it to evaluate the positive effects of D3 and ERDOS on the driving safety of AVs. Over a challenging driving scenario spanning 50kms, D3 achieves a 68% reduction in collisions as compared to prior state-of-the-art programming models.

We open-source all of the contributions of this thesis, and hope that these systems and platforms will empower the broader AV community to conduct impactful research that will lead to a safer autonomous future.

In the remainder of this chapter, we look back towards the work conducted in the dissertation and summarize the lessons learnt during the process (§7.1). In addition, in §7.2, we enumerate several concrete lines of research that remain unexplored, and will prove critical to enabling D3 and ERDOS to be deployed on production vehicles.

7.1 Lessons Learnt

We now summarize the key lessons learnt during the development of the abstractions and systems that support D3 and make it amenable for AV development.

Harnessing Application Knowledge for Efficient Abstractions

We believe that the techniques that we have proposed in this dissertation remain broadly applicable to modern cyber-physical systems that must continuously interact with an environment inhabited by humans, but without a human safety operator in the loop.

Our initial prototypes of ERDOS began by trying to develop abstractions that could act as a common substrate for all such applications. Specifically, we ported common applications involving robot arms, specialized gripping mechanisms etc., but found that these collective applications led to general abstractions that did not highlight unique characteristics. Thus, while our initial prototypes offered various quality-of-life improvements for developers of robotic applications as compared to ROS (e.g., centralized package management, ease of deployment across versions, better operator scheduling leading to efficient communication etc.), they did not lead to the design of fundamentally new abstractions that could offer any tangible safety benefits to cyber-physical systems.

To achieve a better understanding of the challenges, we decided to focus on AVs as a key example of modern cyber-physical systems, and spearheaded an initial prototype of Pylot. Pylot was a multi-year effort whose interfaces amongst components had to be constantly reworked in order to understand the key characteristics that must be supported by execution systems. Our development efforts led us to distill the two unique characteristics: *Environment-Dependent Deadlines* (**C1**) and *Environment-Dependent Computational Complexity* (**C2**), which were crucial in the development of the D3 programming model.

Similarly, the port of Pylot to a real vehicle, provided us with greater insight on Pylot's interfaces amongst components, and led to the design of cleaner abstractions in ERDOS that could help us deploy Pylot to both simulators and real-vehicles seamlessly.

Performance Through Simple Abstractions

A key driving goal in the research proposed in this dissertation was the search for minimalist abstractions. Surprisingly, we found that the design of such simple abstractions often did not come at the cost of the performance, and instead, aided in achieving better performance than prior state-of-the-art abstractions.

A prime example of these simple abstractions is STRL++ (§6.4.2), the declarative language for capturing the three key scheduling requirements: *Precedence Constraints* (R1), *Placement Preferences* (R2) and *Timing Constraints* (R3). We strove to ensure that STRL++ borrowed from prior languages that were robustly validated by the community, while adding the minimal set of new language constructs required to support R1-R3. Together, we found that these new abstractions captured a wealth of information about the placement choices

that could guide an automatic optimization of the generated mathematical models and scale them to bigger problem sizes at lower latencies.

A similar observation was made in the design of the *timestamp* and *frequency* deadlines that were able to efficiently capture the entire set of possible deadlines enumerated in prior works using just the simple boolean functions discussed in §3.4.1. These functions were not only easier to specify for the developers, they were also extremely simple to evaluate and allowed ERDOS to provide extremely fine-grained execution events that dynamically-varying deadlines could be enforced upon.

7.2 Future Work

We now enumerate two concrete lines of future work that present unique challenges, and will prove critical in enabling D3 and ERDOS to be deployed on production vehicles.

7.2.1 STRL++: An Intermediate Representation for Scheduling

Prior work in resource scheduling algorithms either relies on application-specific heuristics [125, 79, 80, 260, 126, 198] or optimization solver-backed mathematical models [287, 310, 234] tailored to specific requirements of an application. However, these approaches typically construct their own intermediate data structures to represent a scheduling problem [125, 198, 287] which fail to generalize across different application requirements. As a result, each subsequent work must undergo the extremely tedious work of constructing an appropriate formulation for their specific requirements, which ultimately hinders the reproducibility, analysis and comparison of different scheduling algorithms.

In this line of work, we draw inspiration from the success of the LLVM IR [177] and logical algebra in DBMS query planning [124, 213] to undertake the development of an expressive intermediate representation that allows applications to specify their placement requirements and scheduling techniques to convert them to space-time allocations on physical resources. Both LLVM and logical algebra for query planning significantly eased the development of efficient, backend-specific program/query transformations to exploit parallelism and application-specific optimizations [213]. Similarly, a scheduling IR will enable subsequent works to eschew the development of application-specific data structures and instead suggest novel transformations to the IR to meet application-specific scheduling requirements.

For example, prior work in optimization solver-backed scheduling algorithms [287, 310, 234, 154, 118] relies on vastly different optimization techniques. Specifically, while DCM [287] uses Constraint Programming (CP; realized using Google OR-Tools [240]) for scheduling its application, Tetrisched [310] relies on Mixed Integer-Linear Programming (MILP; realized using CPLEX [148] and Gurobi [135]) to achieve its objectives. Thus, while they both compute space-time allocation of resources to an application, their underlying mechanisms (CP and MILP) provide vastly different primitives to achieve their objectives that have been shown to have complimentary strength and weaknesses [6, 142, 143, 141]. The goal of an

expressive IR should be to enable application's requirements to be efficiently lowered into either of the aforementioned formulations in order to dynamically switch between them and exploit their complementary strengths and offset their weaknesses.

To realize our vision of the development of a general, yet expressive intermediate representation for space-time scheduling of applications, we propose the following action items:

MILP	CP-SAT
$Pl(T,t) = \begin{cases} R_T & \text{if task } T \text{ started at time } t \\ 0 & \text{otherwise} \end{cases}$	$S(T) = \text{Start time of task } T$ (7.1)

(a) **Start-time formulation.** In MILP, we associate an integer $Pl(T,t)$ for each possible start time t that resolves to R_T if task T starts at t, and 0 otherwise. In CP-SAT, the start time of a task is represented as an integer variable $S(T)$.

$\sum_{t=s}^{D-r} Pl(T,t) \leq R_T \; \forall \; T \in \{T_1, T_2\}$	$S(T) \in [s, D-r)$ (7.2)

(b) **Mutual exclusivity of placements.** In MILP we ensure that only one start time is chosen for each task T from all start times $t \in [s, D-r)$. In CP-SAT, we restrict the domain of the start time $S(T)$ to the same range.

$\forall\, t \in \{s \ldots D\}:$ $\sum_{t_1=t-r}^{t} Pl(T_1+t_1) + \sum_{t_2=t-r}^{t} Pl(T_2+t_2) \leq \lvert M \rvert$	$S(T_1)+r \geq S(T_2)$ $\vee S(T_2)+r \geq S(T_1)$ $\implies R_{T_1} + R_{T_2} \leq \lvert M \rvert$ (7.3)

Respecting resource constraints. In MILP, we ensure that for all discretizations t, the sum of resource-requirements of all task placements for $T \in \{T_1, T_2\}$ that could possibly execute at t is less than the available resources. In CP-SAT, we use logical conditions to ensure that if T_1 and T_2 overlap, they do not oversubscribe the set of available resources.

Figure 7.1: Alternate formulations for MILP and CP-SAT backend for placing two tasks $\{T_1, T_2\}$ both with a start time s, deadline D, runtime r and requirements of R_{T_1} and R_{T_2} resources respectively from a set of M machines.

(1) **Generalizing STRL++ to support alternate backends (CP-SAT, MILP, Flow).** This line of work seeks to focus on the correct return and side-effect semantics of each expression such that they can be efficiently lowered to multiple backends (e.g., CP-SAT [240] and Min-Cost, Max-Flow (MCMF) [154, 118]). We discuss one such lowering in Fig. 7.1,

where we show a hand-written MILP and CP modelling of a deadline-aware scheduling of two tasks: T_1 and T_2. Notably, while MILP formulation represents the start time S_T of a task as a discrete choice amongst integer placement variables $Pl(T,t)$ such that $Pl(T,t) \implies S(T) = t$ (enforced using Eq. (7.1), Eq. (7.2)). On the other hand, CP-SAT encodes the start time as an integer variable whose domain ranges from $[s, D-r)$. As a result of the difference in the way each approach models the start time of tasks, both MILP and CP-SAT require fundamentally different modelling techniques to ensure that capacity constraints of the resource-time space are respected. Eq. (7.3) shows that MILP ensures that all possible start times of each Task T that could use the resource at a time t require $\leq |M|$ machines to execute. On the other hand, CP-SAT uses logical conditions to ensure that if T_1 and T_2 overlap, they do not oversubscribe the set of available resources. Moreover, prior work [154, 118] has shown considerable performance improvements by using network flows to schedule a set of tasks. However, network flows also require a different approach to modeling the scheduling problem, and only work for a particular structure of scheduling problems (namely, they do not support DAG-aware scheduling). This difference in modelling encourages us to investigate the correct semantics for each expression such that STRL++ is amenable to multiple backends, and can harness their performance benefits. Thus, we seek to specify the formal semantics of each STRL++ expression and define their composability rules.

(**2**) **Optimization and Performance Scaling.** The ability to lower STRL++ to multiple solver backends will unlock a systematic and detailed study of the strengths and weaknesses of such backends for scheduling soft real-time ML applications. Prior works [6, 160] have discussed the general strengths and weaknesses of both MILP and CP approaches. We seek to build on this work to understand the nature of specific problem instances (described by both the structure of the DAGs available for scheduling and the state of the resources in the system) that benefit from each modelling approach. For example, conventional wisdom [160] dictates that CP techniques are much more efficient at finding feasible solutions whereas MILP solvers are more efficient at optimization. Using this intuition, we conjecture that the latency of the scheduler would benefit from using the CP formulation when the deadline of the tasks is lax and thus feasible solutions are easier to find. On the other hand, the latency of the MILP solver will be lower when the deadline is tight, feasible solutions are hard to find and an optimal solution is required. Fig. 7.2 tests this hypothesis by comparing the latency of solving hand-written MILP and CP models (Fig. 7.1) as a function of the SLO factor of tasks. We notice that at tighter deadlines, MILP is 2.3× faster than CP-SAT, whereas CP-SAT is 2.04× faster as the SLO factor increases and the deadlines become more lax.

In addition, the definition of the formal semantics of the STRL++ language along with the composability rules for expressions will enable us to define semantic-preserving transformations that can significantly impact the overall solver time by constructing more efficient underlying mathematical models for the given problem. Finally, we seek to explore the expressiveness and efficacy of STRL++ as an IR for development of scheduling heuristics, similar to the ones proposed in literature [125, 126, 132, 79, 260]. Specifically, we will undertake the expression of selected scheduling heuristics as transformations to the STRL++ represen-

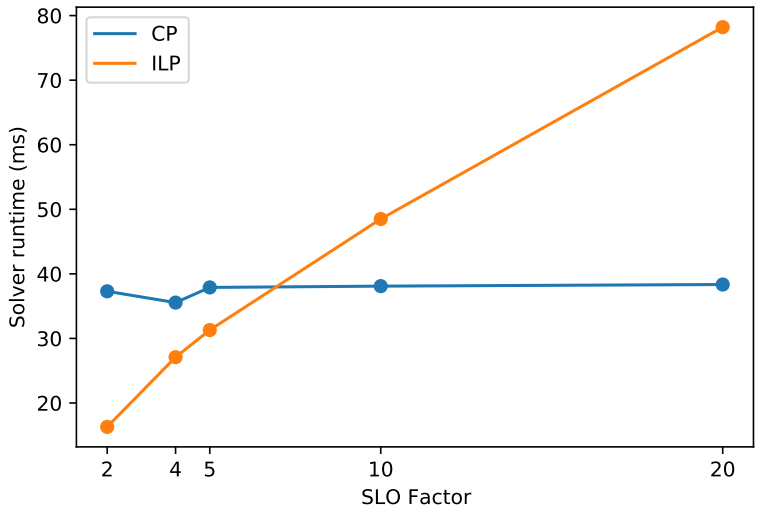


Figure 7.2: Comparing MILP and CP. MILP optimizes placement 2.3× faster under tight SLOs due to a reduced number of placement choices. CP is 2.04× faster at lax deadlines since it is able to find feasible solutions quickly.

tation of an application's requirements. In addition, we also seek to develop novel heuristics atop STRL++ that are not only performant and provide efficient resource-time placements for specific application requirements, but can also be used to come up with initial feasible solutions quickly. These initial feasible solutions can then be amended by MILP solvers to construct optimal resource-time placements for the application quickly.

7.2.2 Scheduling for Conditional DAGs

In Chapter 6, we explored DAGSched, a technique for end-to-end scheduling of job DAGs that conform to the D3 programming model. In DAGSched, we assume knowledge of the job graph along with an estimate of the runtime and resource profiles for each component. These constraints are similar to data-parallel jobs where the job graphs are considered to be static (or are reconfigured by external mechanisms that are known to the scheduler).

However, a key requirement of D3 is to allow components to adapt their computation to meet the dynamically-varying deadlines. This leads to an execution model where job graphs encode all possible execution paths of the computation, and *dynamically actuate* only some paths at runtime corresponding to the data-driven execution choices made by the graph.

This execution model has been recently formalized as the "*conditional DAG model*" [201]. In addition to D3, such an execution model shows up in two major areas of work:

(**1**) **Video Analytics** where recent works [162, 328, 130] have transformed the optimization of the latency-accuracy curve for video analytics into a job graph with dynamically-chosen branches. Each branch represents a choice of the "knobs" that provides a singular point on

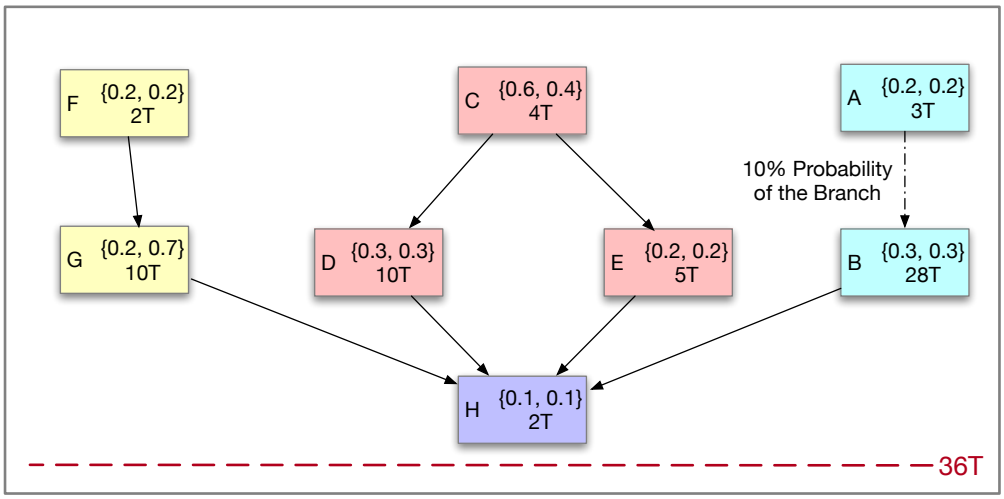


Figure 7.3: A sample job exhibiting the need for conditionality-aware scheduling. Each task in the job requests two resources $\{R_1, R_2\}$ and executes for the annotated T time units. The deadline of the job is set at $36T$, and task A executes task B with a 10% probability.

the pareto-optimal curve of latency-accuracy for the choice of detection + tracking models.

(**2**) **Workflow Orchestration** tools (e.g., Airflow [27], Dagster [83], Prefect [247] etc.) provide constructs for branching inside the DAG, which eases the analysis of the job graph.

A formal analysis has shown that the schedulability of any possible execution variant of conditional-DAGs (preemptive vs. non-preemptive, etc.) is P-SPACE complete [45]. In Fig. 7.3, we show a sample job that contains 8 tasks, each of which request a fixed amount of the resources $\{R_1, R_2\}$ (each with a unit of resource available), and task A only conditionally executes the task G with a 10% probability. Fig. 7.4 shows two equivalent schedules (decided by DAGSched) under the worst-case assumption that all tasks (including task B) execute. However, we observe that the two schedules are not equivalent. The schedule shown in the top box of Fig. 7.4 lends itself to a less efficient resource utilization when task B does not execute, and the schedule is replanned after task A finishes execution. On the other hand, the schedule shown in the bottom box of Fig. 7.4 leads to a much more efficient resource utilization. In this line of work, we seek to incorporate the notion of *robustness* under the worst-case schedule planning, to ensure that the resources are efficiently utilized when mispredictions occur in the conditional DAG execution model.

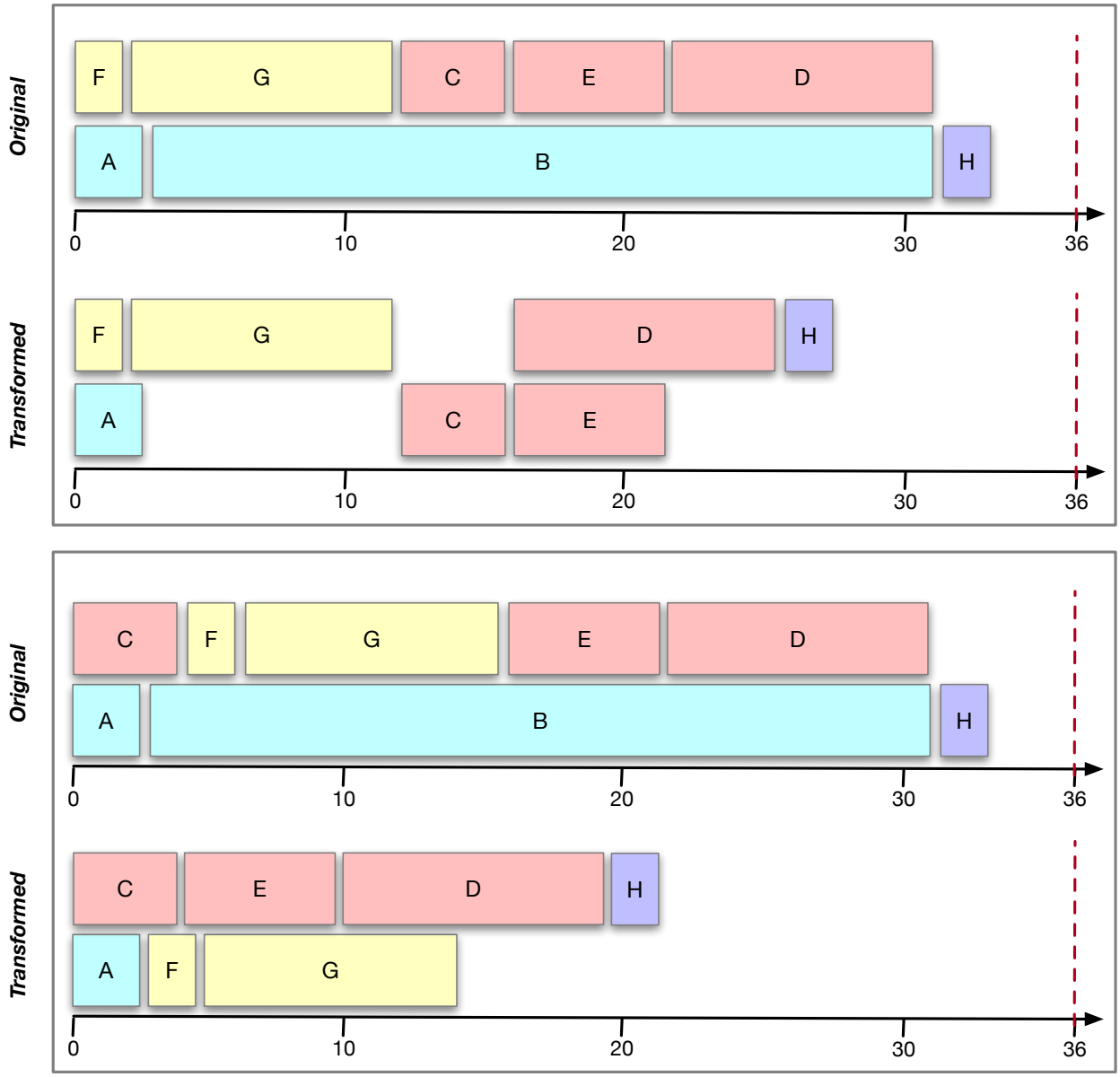


Figure 7.4: A need to consider *robustness* in the scheduling algorithms. Both "Original" schedules are equivalent when task B is executed, but the bottom schedule is much more robust to mispredictions in the branches of the conditional DAG, and leads to more efficient resource utilization under mispredictions.

Bibliography

[1] Federal Aviation Administration (FAA). *Advanced Avionics Handbook, Chapter 4: Automated Flight Control*. https://www.faa.gov/regulations_policies/handbooks_manuals/aviation/advanced_avionics_handbook/media/aah_ch04.pdf.

[2] Daniel J Abadi et al. "The Design of the Borealis Stream Processing Engine". In: *Proceedings of the 2nd Biennial Conference on Innovative Data Systems Research (CIDR)*. 2005, pp. 277–289.

[3] Martín Abadi et al. "TensorFlow: A System for Large-Scale Machine Learning". In: *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Savannah, Georgia, USA, Nov. 2016.

[4] Martín Abadi et al. "TensorFlow: A System for Large-Scale Machine Learning". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 2016, pp. 265–283.

[5] *Accelerate AI development with Google Cloud TPUs*. https://cloud.google.com/tpu.

[6] Tobias Achterberg. "SCIP: solving constraint integer programs". In: *Mathematical Programming Computation* 1 (2009), pp. 1–41.

[7] Michael Aeberhard et al. "Automated Driving with ROS at BMW". In: *ROSCon 2015 Hamburg, Germany* (2015).

[8] Michael Aeberhard et al. *Automated Driving with ROS at BMW*. http://www.ros.org/news/2016/05/michael-aeberhard-bmw-automated-driving-with-ros-at-bmw.html.

[9] Tyler Akidau et al. "MillWheel: Fault-tolerant Stream Processing at Internet Scale". In: *Proceedings of the VLDB* 6.11 (Aug. 2013). ISSN: 2150-8097. DOI: 10.14778/2536222.2536229. URL: http://dx.doi.org/10.14778/2536222.2536229.

[10] Tyler Akidau et al. "Millwheel: Fault-Tolerant Stream Processing at Internet Scale". In: *Proceedings of the VLDB Endowment* 6.11 (2013), pp. 1033–1044.

[11] Tyler Akidau et al. "The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-scale, Unbounded, Out-of-order Data Processing". In: *Proceedings of the VLDB Endowment* 8.12 (2015), pp. 1792–1803.

[12] Tyler Akidau et al. "The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-scale, Unbounded, Out-of-order Data Processing". In: *Proceedings of the VLDB Endowment* 8.12 (Aug. 2015), pp. 1792–1803. ISSN: 2150-8097. DOI: 10.14778/2824032.2824076. URL: http://dx.doi.org/10.14778/2824032.2824076.

[13] Tyler Akidau et al. "Watermarks in Stream Processing Systems: Semantics and Comparative Analysis of Apache Flink and Google Cloud Dataflow". In: *Proceedings of the VLDB Endowment* 14.12 (2021).

[14] Alexandre Alahi et al. "Social LSTM: Human Trajectory Prediction in Crowded Spaces". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, pp. 961–971.

[15] Sergi Alcaide et al. "High-Integrity GPU Designs for Critical Real-Time Automotive Systems". In: *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2019, pp. 824–829.

[16] Sergi Alcaide et al. "Safety-Related Challenges and Opportunities for GPUs in the Automotive Domain". In: *IEEE Micro* 38.6 (2018), pp. 46–55.

[17] Miguel Alcon et al. "Timing of Autonomous Driving Software: Problem Analysis and Prospects for Future Solutions". In: *Proceedings of the 26th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE. 2020, pp. 267–280.

[18] Waleed Ali et al. "Yolo3d: End-to-end real-time 3d oriented object bounding box detection from lidar point cloud". In: *Proceedings of the European Conference on Computer Vision (ECCV) Workshops*. 2018.

[19] *Alibaba Cluster Trace v2018*. https://tinyurl.com/alibaba-2018.

[20] Luis Almeida et al. "A Dynamic Scheduling Approach to Designing Flexible Safety-Critical Systems". In: *Proceedings of the 7th ACM & IEEE International Conference on Embedded Software*. 2007, pp. 67–74.

[21] Gautam Altekar and Ion Stoica. "ODR: Output-Deterministic Replay for Multicore Debugging". In: *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*. 2009, pp. 193–206.

[22] Amara D. Angelica. *Google's self-driving car gathers nearly 1 GB/sec*. http://www.kurzweilai.net/googles-self-driving-car-gathers-nearly-1-gbsec.

[23] Brandon Amos et al. "Differentiable MPC for End-to-End Planning and Control". In: *Proceedings of the 32nd International Conference on Neural Information Processing Systems (NeurIPS)*. 2018, pp. 8299–8310.

[24] Seifemichael B Amsalu et al. "Driver Behavior Modeling near Intersections using Support Vector Machines based on Statistical Feature Extraction". In: *Proceedings of the IEEE Intelligent Vehicles Symposium (IV)*. IEEE. 2015, pp. 1270–1275.

[25] Ganesh Ananthanarayanan et al. "Real-time video analytics: The killer app for edge computing". In: *computer* 50.10 (2017), pp. 58–67.

[26] Dragomir Anguelov. *Taming The Long Tail of Autonomous Driving Challenges*. https://www.youtube.com/watch?v=Q0nGo2-y0xY. 2019.

[27] *Apache Airflow™*. https://airflow.apache.org.

[28] *Apollo Planning Frequency*. https://github.com/ApolloAuto/apollo/blob/master/modules/planning/common/planning_gflags.cc#L23.

[29] *Apollo's Traffic Light Perception*. https://github.com/ApolloAuto/apollo/blob/master/docs/06_Perception/traffic_light.md.

[30] *ApolloAuto's Third-Party Software Dependencies*. https://github.com/ApolloAuto/apollo/tree/master/third_party.

[31] *Are Airplane Autopilot Systems the Same as Self-Driving Car AI?* https://www.aitrends.com/ai-insider/airplane-autopilot-systems-self-driving-car-ai/.

[32] *Argoverse*. https://www.argoverse.org/.

[33] M Sanjeev Arulampalam et al. "A tutorial on particle filters for online nonlinear/non-Gaussian Bayesian tracking". In: *IEEE Transactions on signal processing* 50.2 (2002), pp. 174–188.

[34] *As Automakers Add Technology to Cars, Software Bugs Follow*. https://www.nytimes.com/2022/02/08/business/car-software-lawsuits.html.

[35] Autoware. *Autoware User's Manual - Document Version 1.1*. https://tinyurl.com/2v2jkk9n.

[36] *AWS EC2 instance timeline*. https://instancetyp.es.

[37] *AWS Inferentia*. https://aws.amazon.com/machine-learning/inferentia/.

[38] *AWS Inferentia*. https://aws.amazon.com/machine-learning/inferentia/.

[39] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. "Layer normalization". In: *arXiv preprint arXiv:1607.06450* (2016).

[40] Baidu. *Apollo 3.0 Software Architecture*. https://tinyurl.com/mhd6dfka.

[41] Baidu. *Apollo Cyber RT*. https://github.com/ApolloAuto/apollo/tree/master/cyber.

[42] Baidu. *Apollo Data Open Platform*. http://data.apollo.auto/.

[43] Magdalena Balazinska et al. "Fault-Tolerance in the Borealis Distributed Stream Processing System". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*. Baltimore, Maryland, 2005, pp. 13–24.

[44] Sanjoy Baruah. "An ILP representation of a DAG scheduling problem". In: *Real-Time Systems* 58.1 (2022), pp. 85–102.

[45] Sanjoy Baruah. "Feasibility analysis of conditional dag tasks is co-npnp-hard". In: *Proceedings of the 29th International Conference on Real-Time Networks and Systems*. 2021, pp. 165–172.

[46] Sanjoy Baruah. "Improved Multiprocessor Global Schedulability Analysis of Sporadic DAG Task Systems". In: *Proceedings of the 26th Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE. 2014, pp. 97–105.

[47] Sanjoy Baruah, Vincenzo Bonifaci, and Alberto Marchetti-Spaccamela. "The Global EDF Scheduling of Systems of Conditional Sporadic DAG Tasks". In: *Proceedings of the 27th Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE. 2015, pp. 222–231.

[48] Alex Bewley et al. "Simple Online and Realtime Tracking". In: *Proceedings of the 23th IEEE International Conference on Image Processing (ICIP)*. 2016, pp. 3464–3468. DOI: 10.1109/ICIP.2016.7533003.

[49] Romil Bhardwaj et al. "ESCHER: Expressive Scheduling with Ephemeral Resources". In: *Proceedings of the 13th Symposium on Cloud Computing*. 2022, pp. 47–62.

[50] Peter Biber and Wolfgang Straßer. "The Normal Distributions Transform: A New Approach to Laser Scan Matching". In: *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Vol. 3. IEEE. 2003, pp. 2743–2748.

[51] Aaron Block et al. "An Adaptive Framework for Multiprocessor Real-Time System". In: *Proceedings of the 20th Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE. 2008, pp. 23–33.

[52] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. "YOLOv4: Optimal Speed and Accuracy of Object Detection". In: (2020). eprint: arXiv:2004.10934. URL: https://arxiv.org/abs/2004.10934.

[53] Peter Bodík et al. "Deadline-Aware Scheduling of Big-Data Processing Jobs". In: *Proceedings of the 26th ACM symposium on Parallelism in algorithms and architectures*. 2014, pp. 211–213.

[54] Vincenzo Bonifaci et al. "Feasibility Analysis in the Sporadic DAG Task Model". In: *Proceedings of the 25th Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE. 2013, pp. 225–233.

[55] Sol Boucher et al. "Lightweight Preemptible Functions". In: *Proceedings of the 31st USENIX Annual Technical Conference (ATC)*. 2020, pp. 465–477.

[56] Brad Templeton. *Companies Have Spent Over $16 Billion on Robocars. It's A Drop in the Bucket*. https://www.forbes.com/sites/bradtempleton/2020/02/18/companies-have-spent-over-16b-on-robocars--its-a-drop-in-the-bucket/.

[57] *Building Warehouse-Scale Computers at Google Cloud*. https://www.youtube.com/watch?v=9i7HuU8d3_4.

[58] Giorgio C Buttazzo et al. "Elastic Scheduling for Flexible Workload Management". In: *IEEE Transactions on Computers* 51.3 (2002), pp. 289–302.

[59] Han Cai, Ligeng Zhu, and Song Han. "Proxylessnas: Direct neural architecture search on target task and hardware". In: *arXiv preprint arXiv:1812.00332* (2018).

[60] Han Cai et al. *OFA Pretained Models*. https://drive.google.com/drive/folders/10leLmIiMtaRu4J46KwrBaMydvQt0qFuI?usp=sharing.

[61] Han Cai et al. "Once-for-All: Train One Network and Specialize it for Efficient Deployment". In: *International Conference on Learning Representations*. 2020. URL: https://openreview.net/forum?id=HylxE1HKwS.

[62] Paris Carbone et al. "Apache Flink: Stream and Batch Processing in a Single Engine". In: *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36.4 (2015).

[63] Paris Carbone et al. "Apache Flink™: Stream and Batch Processing in a Single Engine". In: *The Bulletin of the Technical Committee on Data Engineering* 38.4 (2015).

[64] Nicolas Carion et al. "End-to-end object detection with transformers". In: *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part I 16*. Springer. 2020.

[65] Don Carney et al. "Monitoring Streams - A New Class of Data Management Applications". In: *Proceedings of the 28th International Conference on Very Large Databases (VLDB)*. 2002, pp. 215–226.

[66] Daniel Casini et al. "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.

[67] Yuning Chai et al. "MultiPath: Multiple probabilistic anchor trajectory hypotheses for behavior prediction". In: (2019). eprint: arXiv:1910.05449. URL: https://arxiv.org/abs/1910.05449.

[68] Kaiyuan Eric Chen et al. "FogROS: An Adaptive Framework for Automating Fog Robotics Deployment". In: *2021 IEEE 17th International Conference on Automation Science and Engineering (CASE)*. IEEE. 2021, pp. 2035–2042.

[69] Liang-Chieh Chen et al. "Rethinking atrous convolution for semantic image segmentation". In: *arXiv preprint arXiv:1706.05587* (2017).

[70] Minghao Chen et al. "Autoformer: Searching transformers for visual recognition". In: *Proceedings of the IEEE/CVF international conference on computer vision*. 2021, pp. 12270–12280.

[71] Tianyang Chen and Linh Thi Xuan Phan. "SafeMC: A System for the Design and Evaluation of Mode-Change Protocols". In: *Proceedings of the 24th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE. 2018, pp. 105–116.

[72] Xiaozhi Chen et al. "Multi-View 3D Object Detection Network for Autonomous Driving". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2017, pp. 1907–1915.

[73] Aakanksha Chowdhery et al. "PaLM: Scaling Language Modeling with Pathways". In: *arXiv preprint arXiv:2204.02311* (2022).

[74] Hoon Sung Chwa et al. "Physical-State-Aware Dynamic Slack Management for Mixed-Criticality Systems". In: *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE. 2018, pp. 129–139.

[75] Laurene Claussmann et al. "A Review of Motion Planning for Highway Autonomous Driving". In: *IEEE Transactions on Intelligent Transportation Systems* 21.5 (2019), pp. 1826–1848.

[76] Henry Claypool, Amitai Bin-Nun, and Jeffrey Gerlach. "Self-Driving Cars: The Impact on People with Disabilities". In: *Newton, MA: Ruderman Family Foundation* (2017).

[77] Marius Cordts et al. "The Cityscapes Dataset for Semantic Urban Scene Understanding". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016.

[78] R Craig Coulter. *Implementation of the pure pursuit path tracking algorithm*. Tech. rep. Carnegie-Mellon UNIV Pittsburgh PA Robotics INST, 1992.

[79] Daniel Crankshaw et al. "Clipper: A Low-Latency Online Prediction Serving System". In: *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. 2017, pp. 613–627.

[80] Daniel Crankshaw et al. "Janus: Latency-Aware Provisioning and Scaling for Prediction Serving Pipelines". In: *Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC)*. 2020.

[81] Henggang Cui et al. "Deep Kinematic Models for Kinematically Feasible Vehicle Trajectory Predictions". In: *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2020, pp. 10563–10569.

[82] Carlo Curino et al. "Reservation-based Scheduling: If You're Late Don't Blame Us!" In: *Proceedings of the ACM Symposium on Cloud Computing*. 2014, pp. 1–14.

[83] *Dagster*. https://dagster.io.

[84] B Dasarathy. "Timing constraints of real-time systems: Constructs for expressing them, methods of validating them". In: *IEEE Transactions on Software Engineering* 1 (1985), pp. 80–86.

[85] Dakshina Dasari et al. "Identifying the Sources of Unpredictability in COTS-based Multicore Systems". In: *Proceedings of the 8th International Symposium on Industrial Embedded Systems (SIES)*. IEEE. 2013, pp. 39–48.

[86] *Dataspeed: The Industry-Leading Drive-by-Wire Kit*. https://www.dataspeedinc.com/.

[87] Dionisio De Niz, Karthik Lakshmanan, and Ragunathan Rajkumar. "On the Scheduling of Mixed-Criticality Real-Time Task Sets". In: *Proceedings of the 30th IEEE Real-Time Systems Symposium (RTSS)*. IEEE. 2009, pp. 291–300.

[88] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *Communications of the ACM* 51.1 (2008), pp. 107–113.

[89] Johannes Deichmann et al. *Autonomous driving's future: Convenient and connected*. https://www.mckinsey.com/industries/automotive-and-assembly/our-insights/autonomous-drivings-future-convenient-and-connected.

[90] Christina Delimitrou and Christos Kozyrakis. "Paragon: QoS-aware scheduling for heterogeneous datacenters". In: *ACM SIGPLAN Notices* 48.4 (2013), pp. 77–88.

[91] Jia Deng et al. "Imagenet: A large-scale hierarchical image database". In: *2009 IEEE conference on computer vision and pattern recognition*. Ieee. 2009, pp. 248–255.

[92] Patricia Derler et al. *PTIDES: A Programming Model for Distributed Real-Time Embedded Systems*. Tech. rep. University of California, Berkeley, 2008.

[93] *DETR - Hugging Face*. https://huggingface.co/docs/transformers/model_doc/detr.

[94] Dirk Thomas. *Changes between ROS 1 and ROS 2*. http://design.ros2.org/articles/changes.html.

[95] Dmitri Dolgov et al. "Practical Search Techniques in Path Planning for Autonomous Driving". In: *Proceedings of the 1st International Symposium on Search Techniques in Artificial Intelligence and Robotics (STAIR)*. Vol. 1001. Ann Arbor, USA, 2008, pp. 18–80.

[96] Alexey Dosovitskiy et al. "CARLA: An Open Urban Driving Simulator". In: *Proceedings of the 1st Conference on Robot Learning (CoRL)*. 2017, pp. 1–16.

[97] Yangliu Dou, Fengjun Yan, and Daiwei Feng. "Lane Changing Prediction at Highway Lane Drops using Support Vector Machine and Artificial Neural Network Classifiers". In: *Proceedings of the IEEE International Conference on Advanced Intelligent Mechatronics (AIM)*. IEEE. 2016, pp. 901–906.

[98] Dmitry Duplyakin et al. "The Design and Operation of CloudLab". In: *Proceedings of the USENIX Annual Technical Conference (ATC)*. July 2019, pp. 1–14. URL: https://www.flux.utah.edu/paper/duplyakin-atc19.

[99] Glenn A Elliott, Bryan C Ward, and James H Anderson. "GPUSync: A Framework for Real-Time GPU Management". In: *Proceedings of the 34th Real-Time Systems Symposium (RTSS)*. IEEE. 2013, pp. 33–44.

[100] Glenn A. Elliott. "Real-Time Scheduling for GPUs with Applications in Advanced Automotive Systems". PhD thesis. University of Nort Carolina at Chapel Hill, Jan. 2015.

[101] Elmootazbellah Nabil Elnozahy et al. "A Survey of Rollback-Recovery Protocols in Message-Passing Systems". In: *ACM Computing Surveys (CSUR)* 34.3 (2002), pp. 375–408.

[102] *ERDOS: Elastic Robot Data-flow Operating System*. https://github.com/erdos-project/erdos.

[103] Ford. *A Matter of Trust: Ford's Approach to Developing Self-driving Vehicles*. https://media.ford.com/content/dam/fordmedia/pdf/Ford_AV_LLC_FINAL_HR_2.pdf.

[104] Armando Fox et al. "Above the clouds: A berkeley view of cloud computing". In: *Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS* 28 (2009).

[105] Romero Francisco, Qian Li, and Christos Kozyrakis. Personal Communication. Dec. 2022.

[106] Andreas Fregin et al. *Building a Computer Vision Research Vehicle with ROS*. http://www.ros.org/news/2018/07/roscon-2017-building-a-computer-vision-research-vehicle-with-ros----andreas-fregin.html.

[107] Vijay Gadepally, Ashok Krishnamurthy, and Umit Ozguner. "A framework for estimating driver decisions near intersections". In: *IEEE Transactions on Intelligent Transportation Systems* 15.2 (2013), pp. 637–646.

[108] Shen Gao et al. "Abstractive text summarization by incorporating reader comments". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 33. 01. 2019, pp. 6399–6406.

[109] Carlos E Garcia, David M Prett, and Manfred Morari. "Model predictive control: Theory and practice—A survey". In: *Automatica* 25.3 (1989), pp. 335–348.

[110] Andreas Geiger, Philip Lenz, and Raquel Urtasun. "Are we ready for Autonomous Driving? The KITTI Vision Benchmark Suite". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2012.

[111] Andreas Geiger et al. *The KITTI Vision Benchmark Suite*. http://www.cvlibs.net/datasets/kitti/.

[112] Andreas Geiger et al. "Vision Meets Robotics: The KITTI Dataset". en. In: *The International Journal of Robotics Research* 32.11 (Sept. 2013), pp. 1231–1237. ISSN: 0278-3649. DOI: 10.1177/0278364913491297.

[113] General Motors. *2018 Self-driving safety report*. https://www.gm.com/content/dam/company/docs/us/en/gmcom/gmsafetyreport.pdf.

[114] Yilong Geng et al. "Exploiting a Natural Network Effect for Scalable, Fine-Grained Clock Synchronization". In: *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. 2018, pp. 81–94.

[115] Ali Ghodsi et al. "Dominant Resource Fairness: Fair Allocation of Multiple Resource Types". In: *8th USENIX symposium on networked systems design and implementation (NSDI 11)*. 2011.

[116] KAHN Gilles. "The semantics of a simple language for parallel programming". In: *Information processing* 74 (1974), pp. 471–475.

[117] Ionel Gog et al. "D3: a dynamic deadline-driven approach for building autonomous vehicles". In: *Proceedings of the Seventeenth European Conference on Computer Systems*. 2022, pp. 453–471.

[118] Ionel Gog et al. "Firmament: Fast, Centralized Cluster Scheduling at Scale". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 2016, pp. 99–115.

[119] Ionel Gog et al. "Pylot: A Modular Platform for Exploring Latency-Accuracy Tradeoffs in Autonomous Vehicles". In: *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2021, pp. 8806–8813.

[120] Chengyue Gong et al. "Nasvit: Neural architecture search for efficient vision transformers with gradient conflict aware supernet training". In: *International Conference on Learning Representations*. 2021.

[121] *Google Cloud Dataflow*. http://cloud.google.com/dataflow/. Google Inc.

[122] *Google Cloud TPU*. https://cloud.google.com/tpu.

[123] Joël Goossens, Shelby Funk, and Sanjoy Baruah. "Priority-Driven Scheduling of Periodic Task Systems on Multiprocessors". In: *Real-time Systems* 25.2 (2003), pp. 187–205.

[124] Goetz Graefe and William J McKenna. "The volcano optimizer generator: Extensibility and efficient search". In: *Proceedings of IEEE 9th international conference on data engineering*. IEEE. 1993, pp. 209–218.

[125] Robert Grandl et al. "GRAPHENE: Packing and Dependency-aware Scheduling for Data-Parallel Clusters". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 2016, pp. 81–97.

[126] Robert Grandl et al. "Multi-Resource Packing for Cluster Schedulers". In: *SIGCOMM Computer Communication Review* 44.4 (2014), pp. 455–466.

[127] Jim Gray. "Why Do Computers Stop and What Can Be Done About It?" In: *Symposium on Reliability in Distributed Software and Database Systems*. Los Angeles, CA, USA. 1986, pp. 3–12.

[128] Matthew P Grosvenor et al. "Queues {don't} matter when you can {JUMP} them!" In: *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. 2015, pp. 1–14.

[129] *gRPC*. https://grpc.io/.

[130] *GStreamer: open source multimedia framework*. https://gstreamer.freedesktop.org/.

[131] Jacopo Guanetti, Yeojun Kim, and Francesco Borrelli. "Control of Connected and Automated Vehicles: State of the Art and Future Challenges". en. In: *Annual Reviews in Control* 45 (Jan. 2018), pp. 18–40. ISSN: 1367-5788. DOI: 10.1016/j.arcontrol.2018.04.011.

[132] Arpan Gujarati et al. "Serving DNNs like Clockwork: Performance Predictability from the Bottom Up". In: *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Banff, Canada, Nov. 2020.

[133] Arpan Gujarati et al. "Swayam: Distributed Autoscaling to Meet SLAs of Machine Learning Inference Services with Resource Efficiency". In: *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*. Middleware '17. Las Vegas, Nevada: Association for Computing Machinery, 2017, pp. 109–120. ISBN: 9781450347204. DOI: 10.1145/3135974.3135993. URL: https://doi.org/10.1145/3135974.3135993.

[134] *Gurobi - MIP Starts*. https://www.gurobi.com/documentation/current/examples/mip_starts.html.

[135] Gurobi Optimization, LLC. *Gurobi Optimizer Reference Manual*. 2023. URL: https://www.gurobi.com.

[136] *Hardware prerequisites of Baidu's ApolloAuto*. https://github.com/ApolloAuto/apollo#prerequisites.

[137] Kim Hazelwood et al. "Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective". In: *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2018, pp. 620–629.

[138] Kim Hazelwood et al. "Applied machine learning at facebook: A datacenter infrastructure perspective". In: *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2018, pp. 620–629.

[139] Kaiming He et al. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[140] Christopher Ho et al. *ROS2 on Autonomous Vehicles*. https://roscon.ros.org/2018/presentations/ROSCon2018_ROS2onAutonomousDrivingVehicles.pdf.

[141] John N Hooker. "Logic-based Benders decomposition for large-scale optimization". In: *Large Scale Optimization in Supply Chains and Smart Manufacturing: Theory and Applications* (2019), pp. 1–26.

[142] John N Hooker. "Logic, optimization, and constraint programming". In: *INFORMS Journal on Computing* 14.4 (2002), pp. 295–321.

[143] John N Hooker and Greger Ottosson. "Logic-based Benders decomposition". In: *Mathematical Programming* 96.1 (2003), pp. 33–60.

[144] Lu Hou et al. "DynaBERT: Dynamic BERT with Adaptive Width and Depth". In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle et al. Vol. 33. Curran Associates, Inc., 2020, pp. 9782–9793. URL: https://proceedings.neurips.cc/paper/2020/file/6f5216f8d89b086c18298e043bfe48ed-Paper.pdf.

[145] *How Does a Self-Driving Car See?* https://blogs.nvidia.com/blog/2019/04/15/how-does-a-self-driving-car-see/.

[146] *How Uber Self-Driving Cars See The World*. https://www.therobotreport.com/how-uber-self-driving-cars-see-world/.

[147] Jonathan Huang et al. "Speed/Accuracy Trade-Offs for Modern Convolutional Object Detectors". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. July 2017.

[148] *IBM ILOG CPLEX Optimization Studio*. https://tinyurl.com/ilog-cplex.

[149] International Standardization Organization. *Road vehicles - Functional safety*. International Standardization Organization, 2018.

[150] *Introducing the 5th Generation Waymo Driver*. https://blog.waymo.com/2020/03/introducing-5th-generation-waymo-driver.html.

[151] *Investigators found Autopilot was engaged in a Tesla crash in Florida*. https://qz.com/1621235/autopilot-was-engaged-in-a-2019-tesla-model-3-crash-in-florida/.

[152] Sergey Ioffe and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift". In: *International conference on machine learning*. PMLR. 2015, pp. 448–456.

[153] Michael Isard et al. "Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks". In: *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. 2007, pp. 59–72.

[154] Michael Isard et al. "Quincy: Fair Scheduling for Distributed Computing Clusters". In: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 2009, pp. 261–276.

[155] Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. "Serving deep learning models in a serverless platform". In: *CoRR* abs/1710.08460 (2017). arXiv: 1710.08460. URL: http://arxiv.org/abs/1710.08460.

[156] Xabier Iturbe et al. "Addressing functional safety challenges in autonomous vehicles with the arm TCL S architecture". In: *IEEE Design & Test* 35.3 (2018), pp. 7–14.

[157] Ashesh Jain et al. "Recurrent Neural Networks for Driver Activity Anticipation via Sensory-Fusion Architecture". In: *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2016, pp. 3118–3125.

[158] Joel Janai et al. "Computer Vision for Autonomous Vehicles: Problems, Datasets and State of the Art". English. In: *Foundations and Trends in Computer Graphics and Vision* 12.1–3 (July 2020), pp. 1–308. ISSN: 1572-2740, 1572-2759. DOI: 10.1561/0600000079.

[159] Klaus Jansen. "Analysis of scheduling problems with typed task systems". In: *Discrete Applied Mathematics* 52.3 (1994), pp. 223–232.

[160] Jaromił Najman. *What are the differences between Constraint Programming and MIP?* 2023. URL: https://support.gurobi.com/hc/en-us/articles/360048197891-What-are-the-differences-between-Constraint-Programming-and-MIP-.

[161] Haoming Jiang et al. "SMART: Robust and Efficient Fine-Tuning for Pre-trained Natural Language Models through Principled Regularized Optimization". In: *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Online: Association for Computational Linguistics, July 2020, pp. 2177–2190. DOI: 10.18653/v1/2020.acl-main.197. URL: https://aclanthology.org/2020.acl-main.197.

[162] Junchen Jiang et al. "Chameleon: Scalable Adaptation of Video Analytics". In: *Proceedings of the ACM Special Interest Group on Data Communication Conference (SIGCOMM)*. Budapest, Hungary, 2018, pp. 253–266. ISBN: 978-1-4503-5567-4. DOI: 10.1145/3230543.3230574. URL: http://doi.acm.org/10.1145/3230543.3230574.

[163] Jim Foerster. *Weather Creates Challenges For Next Generation Of Vehicles*. https://www.forbes.com/sites/jimfoerster/2019/11/22/weather-creates-challenges-for-next-generation-of-vehicles/?sh=5baaa34a260e.

[164] John Koetsier. *Self-Driving Investment Crash: 58% Drop In Autonomous Vehicle Dollars*. https://www.forbes.com/sites/johnkoetsier/2023/04/19/self-driving-investment-crash-58-drop-in-autonomous-vehicle-dollars/.

[165] Ameet V Joshi and Ameet V Joshi. "Amazon's machine learning toolkit: Sagemaker". In: *Machine learning and artificial intelligence* (2020), pp. 233–243.

[166] Sangeetha Abdu Jyothi et al. "Morpheus: towards automated {SLOs} for enterprise clusters". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 2016, pp. 117–134.

[167] Yutaka Kanayama et al. "A stable Tracking Control Method for an Autonomous Mobile Robot". In: *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 1990, pp. 384–389.

[168] Sean Kane et al. "Toyota sudden unintended acceleration". In: *Safety Research & Strategies* (2010).

[169] Sertac Karaman and Emilio Frazzoli. "Sampling-Based Algorithms for Optimal Motion Planning". In: *The International Journal of Robotics Research* 30.7 (2011), pp. 846–894.

[170] Sertac Karaman et al. "Anytime Motion Planning using the RRT". In: *Proceedings of IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2011, pp. 1478–1483.

[171] Andrej Karpathy. *CVPR '20 - Workshop on Scalability in Autonomous Driving*. https://sites.google.com/view/cvpr20-scalability/archived-talks/keynotes. 2020.

[172] Christos Katrakazas et al. "Real-time Motion Planning Methods for Autonomous On-Road Driving: State-of-the-art and Future Research Directions". In: *Transportation Research Part C: Emerging Technologies* 60 (2015), pp. 416–442.

[173] Ben Kehoe et al. "A survey of research on cloud robotics and automation". In: *IEEE Transactions on automation science and engineering* 12.2 (2015), pp. 398–409.

[174] Christos Kozyrakis. "Resource Efficient Computing for Warehouse-scale Datacenters". In: *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2013, pp. 1351–1356.

[175] Daniel Krajzewicz et al. "Recent development and applications of SUMO-Simulation of Urban MObility". In: *International journal on advances in systems and measurements* 5.3&4 (2012).

[176] T-W Kuo and Aloysius K Mok. "Load Adjustment in Adaptive Real-Time Systems". In: *Proceedings of the 12th Real-Time Systems Symposium (RTSS)*. IEEE. 1991, pp. 160–161.

[177] Chris Lattner and Vikram Adve. "LLVM: A compilation framework for lifelong program analysis & transformation". In: *International symposium on code generation and optimization, 2004. CGO 2004.* IEEE. 2004, pp. 75–86.

[178] Steven M. LaValle. *Planning Algorithms*. USA: Cambridge University Press, 2006. ISBN: 0521862051.

[179] Namhoon Lee et al. "Desire: Distant future prediction in dynamic scenes with interacting agents". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2017, pp. 336–345.

[180] Stéphanie Lefèvre, Dizan Vasquez, and Christian Laugier. "A survey on motion prediction and risk assessment for intelligent vehicles". In: *ROBOMECH journal* 1.1 (2014), pp. 1–14.

[181] Joseph Leung and Hairong Zhao. *Real-time scheduling analysis*. Office of Aviation Research, Federal Aviation Administration, 2005.

[182] Mengtian Li, Yuxiong Wang, and Deva Ramanan. "Towards Streaming Perception". In: *Proceedings of the European Conference on Computer Vision (ECCV)*. Glasgow, Scotland, Aug. 2020.

[183] Zhuohan Li et al. "{AlpaServe}: Statistical Multiplexing with Model Parallelism for Deep Learning Serving". In: *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 2023, pp. 663–679.

[184] Ming Liang et al. "PnPNet: End-to-End Perception and Prediction with Tracking in the Loop". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 2020, pp. 11553–11562.

[185] Richard Liaw et al. "HyperSched: Dynamic Resource Reallocation for Model Development on a Deadline". In: *Proceedings of the ACM Symposium on Cloud Computing*. 2019, pp. 61–73.

[186] Shih-Chieh Lin et al. "The Architectural Implications of Autonomous Driving: Constraints and Acceleration". In: *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Williamsburg, VA, USA, 2018, pp. 751–766. ISBN: 978-1-4503-4911-6. DOI: 10.1145/3173162.3173191. URL: http://doi.acm.org/10.1145/3173162.3173191.

[187] Shih-Chieh Lin et al. "The architectural implications of autonomous driving: Constraints and acceleration". In: *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. 2018, pp. 751–766.

[188] Chang Liu et al. "Path Planning for Autonomous Vehicles using Model Predictive Control". In: *Proceedings of the IEEE Intelligent Vehicles Symposium (IV)*. IEEE. 2017, pp. 174–179.

[189] Chung Laung Liu and James W Layland. "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment". In: *Journal of the ACM (JACM)* 20.1 (1973), pp. 46–61.

[190] Hanxiao Liu, Karen Simonyan, and Yiming Yang. "Darts: Differentiable architecture search". In: *arXiv preprint arXiv:1806.09055* (2018).

[191] Jane W.S. Liu. *Real-Time Systems*. Prentice Hall, 2000.

[192] Wei Liu et al. "SSD: Single Shot Multibox Detector". In: *Proceedings of the European Conference on Computer Vision (ECCV)*. Springer. 2016, pp. 21–37.

[193] Yinhan Liu et al. "Roberta: A robustly optimized bert pretraining approach". In: *arXiv preprint arXiv:1907.11692* (2019).

[194] Ze Liu et al. "Swin transformer: Hierarchical vision transformer using shifted windows". In: *Proceedings of the IEEE/CVF international conference on computer vision*. 2021, pp. 10012–10022.

[195] Zhuang Liu et al. "A convnet for the 2020s". In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2022, pp. 11976–11986.

[196] Marten Lohstroh et al. "Actors Revisited for Time-Critical Systems". In: *Proceedings of the 56th ACM/IEEE Design Automation Conference (DAC)*. IEEE. 2019, pp. 1–4.

[197] Chenyang Lu et al. "Performance Specifications and Metrics for Adaptive Real-Time Systems". In: *Proceedings of the 21st Real-Time Systems Symposium (RTSS)*. IEEE. 2000, pp. 13–23.

[198] Hongzi Mao et al. "Learning Scheduling Algorithms for Data Processing Clusters". In: *Proceedings of the ACM Special Interest Group on Data Communication Conference (SIGCOMM)*. 2019, pp. 270–288.

[199] Yuya Maruyama, Shinpei Kato, and Takuya Azumi. "Exploring the performance of ROS2". In: *Proceedings of the 13th International Conference on Embedded Software*. 2016, pp. 1–10.

[200] Matt Ranney. *Self-Driving Cars As Edge Computing Devices*. https://www.infoq.com/presentations/uber-atg/.

[201] Alessandra Melani et al. "Response-time analysis of conditional DAG tasks in multiprocessor systems". In: *2015 27th Euromicro Conference on Real-Time Systems*. IEEE. 2015, pp. 211–221.

[202] Michele Bertoncello, and Dominik Wee. *Ten Ways Autonomous Driving Could Redefine the Automotive World*. https://www.mckinsey.com/industries/automotive-and-assembly/our-insights/ten-ways-autonomous-driving-could-redefine-the-automotive-world.

[203] Sajjad Mozaffari et al. "Deep Learning-Based Vehicle Behavior Prediction for Autonomous Driving Applications: A Review". In: *IEEE Transactions on Intelligent Transportation Systems* (2020), pp. 1–15. ISSN: 1524-9050, 1558-0016. DOI: 10.1109/TITS.2020.3012034.

[204] Derek G Murray et al. "Naiad: A Timely Dataflow System". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 2013, pp. 439–455.

[205] Derek G. Murray et al. "Naiad: A Timely Dataflow System". In: *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*. Nemacolin Woodlands, Pennsylvania, USA, Nov. 2013, pp. 439–455. ISBN: 978-1-4503-2388-8.

[206] Deepak Narayanan et al. "Heterogeneity-Aware Cluster Scheduling Policies for Deep Learning Workloads". In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 2020, pp. 481–498.

[207] Deepak Narayanan et al. "Solving Large-Scale Granular Resource Allocation Problems Efficiently with POP". In: *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 2021, pp. 521–537.

[208] National Highway Traffic Safety Administration. *Automated Vehicles for Safety*. https://www.nhtsa.gov/technology-innovation/automated-vehicles-safety.

[209] National Highway Traffic Safety Administration. *Critical Reasons for Crashes Investigated in the National Motor Vehicle Crash Causation Survey*. https://crashstats.nhtsa.dot.gov/Api/Public/ViewPublication/812115.

[210] National Highway Traffic Safety Administration. *Pre-Crash Scenario Typology for Crash Avoidance Research*. https://www.nhtsa.gov/sites/nhtsa.dot.gov/files/pre-crash_scenario_typology-final_pdf_version_5-2-07.pdf.

[211] National Highway Traffic Safety Administration. *Traffic Safety Facts (2019 Data)*. https://crashstats.nhtsa.dot.gov/Api/Public/Publication/813060.

[212] National Highway Traffic Safety Administration. *Traffic Safety Facts (2022 Data)*. https://crashstats.nhtsa.dot.gov/Api/Public/ViewPublication/813560.

[213] Thomas Neumann. "Efficiently compiling efficient query plans for modern hardware". In: *Proceedings of the VLDB Endowment* 4.9 (2011), pp. 539–550.

[214] *NHTSA-inspired Pre-crash Scenarios*. https://carlachallenge.org/challenge/nhtsa/.

[215] Jan Nowotsch and Michael Paulitsch. "Leveraging Multi-Core Computing Architectures in Avionics". In: *Proceedings of the 9th European Dependable Computing Conference (EDCC)*. IEEE. 2012, pp. 132–143.

[216] *NTSB's Accident Report on the Uber Self-Driving Vehicle Crash*. https://tinyurl.com/y334xnez.

[217] NVIDIA. *2018 Self-driving safety report*. https://www.nvidia.com/content/dam/en-zz/Solutions/self-driving-cars/safety-report/NVIDIA-Self-Driving-Safety-Report-2018.pdf.

[218] NVIDIA, Péter Vingelmann, and Frank H.P. Fitzek. *CUDA, release: 10.2.89*. 2020. URL: https://developer.nvidia.com/cuda-toolkit.

[219] *NVIDIA A100 Specifications*. https://www.nvidia.com/en-us/data-center/a100/.

[220] *NVIDIA Drive AGX Orin Developer Kit*. https://tinyurl.com/u2rxefpt.

[221] *NVIDIA DRIVE: Hardware for Self-Driving Cars*. https://www.nvidia.com/en-us/self-driving-cars/drive-platform/hardware/.

[222] *NVIDIA Enters Production With DRIVE Orin, Unveils Next-Gen DRIVE Hyperion AV Platform*. https://tinyurl.com/2s38djwr.

[223] *NVIDIA GPU Cloud Computing*. https://www.nvidia.com/en-us/data-center/gpu-cloud-computing/.

[224] *NVIDIA Introduces DRIVE AGX Orin*. https://tinyurl.com/6pjsxzw7.

[225] *NVIDIA Triton Inference System*. https://github.com/triton-inference-server/server/tree/v2.21.0.

[226] *Off road, but not offline: How simulation helps advance our Waymo Driver*. https://blog.waymo.com/2020/04/off-road-but-not-offline--simulation27.html.

[227] Nick Oliver, Kristina Potočnik, and Thomas Calvard. *To Make Self-Driving Cars Safe, We Also Need Better Roads and Infrastructure*. https://hbr.org/2018/08/to-make-self-driving-cars-safe-we-also-need-better-roads-and-infrastructure.

[228] Christopher Olston et al. "Tensorflow-serving: Flexible, high-performance ml serving". In: *arXiv preprint arXiv:1712.06139* (2017).

[229] Kalin Ovtcharov et al. "Accelerating deep convolutional neural networks using specialized hardware". In: *Microsoft Research Whitepaper* 2.11 (2015), pp. 1–4.

[230] Brian Paden et al. "A Survey of Motion Planning and Control Techniques for Self-Driving Urban Vehicles". In: *IEEE Transactions on Intelligent Vehicles* 1.1 (2016), pp. 33–55.

[231] Arthi Padmanabhan et al. "Gemel: Model Merging for {Memory-Efficient},{Real-Time} Video Analytics at the Edge". In: *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 2023, pp. 973–994.

[232] Christos H Papadimitriou and Kenneth Steiglitz. *Combinatorial optimization: algorithms and complexity*. Courier Corporation, 1998.

[233] Jongsoo Park et al. "Deep learning inference in facebook data centers: Characterization, performance optimizations and hardware implications". In: *arXiv preprint arXiv:1811.09886* (2018).

[234] Jun Woo Park et al. "3sigma: distribution-based cluster scheduling for runtime uncertainty". In: *Proceedings of the Thirteenth EuroSys Conference*. 2018, pp. 1–17.

[235] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in neural information processing systems* 32 (2019).

[236] Adam Paszke et al. "Pytorch: An Imperative Style, High-performance Deep Learning Library". In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2019, pp. 8026–8037.

[237] Hiren Patel, Alekh Jindal, and Clemens Szyperski. "Big Data Processing at Microsoft: Hyper Scale, Massive Complexity, and Minimal Cost". In: *Proceedings of the ACM Symposium on Cloud Computing*. 2019, pp. 490–490.

[238] Rodolfo Pellizzoni and Marco Caccamo. "Toward the Predictable Integration of Real-Time COTS Based Systems". In: *Proceedings of the 28th International Real-Time Systems Symposium (RTSS)*. IEEE. 2007, pp. 73–82.

[239] Laurent Perron, Frédéric Didier, and Steven Gay. "The CP-SAT-LP Solver". In: *29th International Conference on Principles and Practice of Constraint Programming (CP 2023)*. Ed. by Roland H. C. Yap. Vol. 280. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023, 3:1–3:2. ISBN: 978-3-95977-300-3. DOI: 10.4230/LIPIcs.CP.2023.3. URL: https://drops.dagstuhl.de/opus/volltexte/2023/19040.

[240] Laurent Perron, Frédéric Didier, and Steven Gay. "The CP-SAT-LP Solver". In: *29th International Conference on Principles and Practice of Constraint Programming (CP 2023)*. Ed. by Roland H. C. Yap. Vol. 280. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023, 3:1–3:2. ISBN: 978-3-95977-300-3. DOI: 10.4230/LIPIcs.CP.2023.3. URL: https://drops.dagstuhl.de/opus/volltexte/2023/19040.

[241] Jonah Philion and Sanja Fidler. "Lift, Splat, Shoot: Encoding Images from Arbitrary Camera Rigs by Implicitly Unprojecting to 3D". In: *Proceedings of the European Conference on Computer Vision (ECCV)*. Springer. 2020, pp. 194–210.

[242] Jonah Philion, Amlan Kar, and Sanja Fidler. "Learning to Evaluate Perception Models using Planner-Centric Metrics". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2020, pp. 14055–14064.

[243] Michael K Pitt and Neil Shephard. "Filtering via simulation: Auxiliary particle filters". In: *Journal of the American statistical association* 94.446 (1999), pp. 590–599.

[244] *Planning Loop Rate in Autoware AV*. https://tinyurl.com/56m3uze2.

[245] Conor Power et al. "The Cosmos Big Data Platform at Microsoft: Over a Decade of Progress and a Decade to Look Forward". In: *Proceedings of the VLDB Endowment* 14.12 (2021), pp. 3148–3161.

[246] *Pre-Crash Scenario Typology for Crash Avoidance Research*. https://www.nhtsa.gov/sites/nhtsa.gov/files/pre-crash_scenario_typology-final_pdf_version_5-2-07.pdf.

[247] *Prefect*. https://www.prefect.io.

[248] Morgan Quigley et al. "ROS: An Open-Source Robot Operating System". In: *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA); Workshop on Open Source Robotics*. Vol. 3. Kobe, Japan, May 2009, p. 5.

[249] Greg Rahn. *GitHub - gregrahn/tpch-kit: TPC-H benchmark kit with some modifications/additions — github.com*. https://github.com/gregrahn/tpch-kit. [Accessed 19-04-2024].

[250] Jorge Real and Alfons Crespo. "Mode Change Protocols for Real-Time Systems: A Survey and a New Proposal". In: *Real-time systems* 26.2 (2004), pp. 161–197.

[251] Rebecca Fannin. *Where the billions spent on autonomous vehicles by U.S. and Chinese giants is heading*. https://www.cnbc.com/2022/05/21/why-the-first-autonomous-vehicles-winners-wont-be-in-your-driveway.html.

[252] John Reif and Hongyan Wang. "The complexity of the two dimensional curvature-constrained shortest-path problem". In: *Third International Workshop on Algorithmic Foundations of Robotics*. 1998, pp. 49–57.

[253] Charles Reiss et al. "Heterogeneity and dynamicity of clouds at scale: Google trace analysis". In: *Proceedings of the third ACM symposium on cloud computing*. 2012, pp. 1–13.

[254] Shaoqing Ren et al. "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks". In: *Proceedings of the International Conferences on Advances in Neural Information Processing Systems (NeurIPS)*. 2015, pp. 91–99.

[255] *Rethinking Cruise's AV Development Loop During COVID-19*. https://medium.com/cruise/cruise-av-development-loop-covid-19-1daef2f0c3d5.

[256] Nicholas Rhinehart, Kris M Kitani, and Paul Vernaza. "R2P2: A Reparameterized Pushforward Policy for Diverse, Precise Generative Path Forecasting". In: *Proceedings of the European Conference on Computer Vision (ECCV)*. 2018, pp. 772–788.

[257] Nicholas Rhinehart et al. "PRECOG: Prediction Conditioned on Goals in Visual Multi-Agent Settings". In: *Proceedings of the IEEE International Conference on Computer Vision (CVPR)*. 2019, pp. 2821–2830.

[258] Alexandre Robicquet et al. "Learning Social Etiquette: Human Trajectory Understanding in Crowded Scenes". In: *Proceedings of the European Conference on Computer Vision (ECCV)*. Springer. 2016, pp. 549–565.

[259] Francisco Romero et al. *InFaaS Policy's Decisions with respect to Accuracy Constraints*. https://tinyurl.com/yfwa36fm.

[260] Francisco Romero et al. "INFaaS: Automated Model-less Inference Serving." In: *USENIX Annual Technical Conference*. 2021, pp. 397–411.

[261] Guodong Rong et al. "LGSVL Simulator: A High Fidelity Simulator for Autonomous Driving". In: *Proceedings of the 23rd International Conference on Intelligent Transportation Systems (ITSC)*. IEEE. 2020, pp. 1–6.

[262] Sven Rönnbäck. *Developement of a INS/GPS navigation loop for an UAV*. 2000.

[263] Daniela Rosu et al. "On Adaptive Resource Allocation for Complex Real-Time Applications". In: *Proceedings of the 18th Real-Time Systems Symposium (RTSS)*. IEEE. 1997, pp. 320–329.

[264] Manas Sahni et al. "CompOFA – Compound Once-For-All Networks for Faster Multi-Platform Deployment". In: *International Conference on Learning Representations*. 2021. URL: https://openreview.net/forum?id=IgIk8RRT-Z.

[265] Claude Samson. "Path Following and Time-Varying Feedback Stabilization of a Wheeled Mobile Robot". In: *Proceedings of the IEEE International Conference on Control, Automation, Robotics and Vision (ICARCV)*. 1992.

[266] Mike Santora. *Simulation Paves Way for Drive.ai Self-Driving Vehicles*. https://www.therobotreport.com/drive-ai-self-driving-simulation/.

[267] *ScenarioRunner for CARLA*. https://github.com/carla-simulator/scenario_runner.

[268] Edward Schmerling, Lucas Janson, and Marco Pavone. "Optimal Sampling-Based Motion Planning under Differential Constraints: The Driftless Case". In: *Proceedings of IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2015, pp. 2368–2375.

[269] Wilko Schwarting, Javier Alonso-Mora, and Daniela Rus. "Planning and Decision-Making for Autonomous Vehicles". en. In: *Annual Review of Control, Robotics, and Autonomous Systems* 1.1 (May 2018), pp. 187–210. ISSN: 2573-5144. DOI: 10.1146/annurev-control-060117-105157.

[270] Gur-Eyal Sela et al. "Context-aware streaming perception in dynamic environments". In: *European Conference on Computer Vision*. Springer. 2022, pp. 621–638.

[271] Mehul A Shah, Joseph M Hellerstein, and Eric Brewer. "Highly Available, Fault-Tolerant, Parallel Dataflows". In: *Proceedings of the 2004 ACM SIGMOD international conference on Management of data (SIGMOD)*. ACM. 2004, pp. 827–838.

[272] Shital Shah et al. "AirSim: High-Fidelity Visual and Physical Simulation for Autonomous Vehicles". In: (2017). eprint: arXiv:1705.05065. URL: https://arxiv.org/abs/1705.05065.

[273] Mohammad Shahrad et al. "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider". In: *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 2020, pp. 205–218.

[274] Marc Shapiro et al. "Conflict-Free Replicated Data Types". In: *Symposium on Self-Stabilizing Systems*. Springer. 2011, pp. 386–400.

[275] Bikash Sharma et al. "Modeling and Synthesizing Task Placement Constraints in Google Compute Clusters". In: *Proceedings of the 2nd ACM Symposium on Cloud Computing*. 2011, pp. 1–14.

[276] Xu Shen, Xiaojing Zhang, and Francesco Borrelli. "Autonomous Parking of Vehicle Fleet in Tight Environments". In: *Proceedings of the American Control Conference (ACC)*. IEEE. 2020, pp. 3035–3040.

[277] *Sight Distance Guidelines*. https://mdotcf.state.mi.us/public/tands/Details_Web/mdot_sight_distance_guidelines.pdf.

[278] Society of Automotive Engineers. *Taxonomy and Definitions for Terms Related to On-Road Motor Vehicle Automated Driving Systems*. SAE International, 2018.

[279] Jonathan Soifer et al. "Deep learning inference service at microsoft". In: *2019 USENIX Conference on Operational Machine Learning (OpML 19)*. 2019, pp. 15–17.

[280] Utkarsh Srivastava and Jennifer Widom. "Flexible time management in data stream systems". In: *Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 2004, pp. 263–274.

[281] State of California Department of Motor Vehicles. *Autonomous Vehicle Disengagement Reports 2018*. https://www.dmv.ca.gov/portal/dmv/detail/vr/autonomous/disengagement_report_2019.

[282] Ion Stoica et al. "A Berkeley View of Systems Challenges for AI". In: *arXiv preprint arXiv:1712.05855* (2017).

[283] Nikolay Stoimenov, Simon Perathoner, and Lothar Thiele. "Reliable Mode Changes in Real-Time Systems with Fixed Priority or EDF Scheduling". In: *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2009, pp. 99–104.

[284] H Streich. "Taskpair-Scheduling: An Approach for Dynamic Real-Time Systems". In: *Second Workshop on Parallel and Distributed Real-Time Systems*. IEEE. 1994, pp. 24–31.

[285] Pei Sun et al. "Scalability in Perception for Autonomous Driving: Waymo Open Dataset". In: (2019). eprint: arXiv:1912.04838. URL: https://arxiv.org/abs/1912.04838.

[286] Martin Sundermeyer, Ralf Schlüter, and Hermann Ney. "LSTM neural networks for language modeling". In: *Thirteenth annual conference of the international speech communication association*. 2012.

[287] Lalith Suresh et al. "Building Scalable and Flexible Cluster Managers Using Declarative Programming". In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 2020, pp. 827–844.

[288] Hamid Tabani et al. "Assessing the Adherence of an Industrial Autonomous Driving Framework to ISO 26262 Software Guidelines". In: *Proceedings of the 56th Annual Design Automation Conference (DAC)*. IEEE. 2019, pp. 1–6.

[289] Mingxing Tan, Ruoming Pang, and Quoc V. Le. "EfficientDet: Scalable and Efficient Object Detection". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2020.

[290] Mingxing Tan et al. "Mnasnet: Platform-aware neural architecture search for mobile". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2019, pp. 2820–2828.

[291] Charlie Tang and Russ R Salakhutdinov. "Multiple Futures Prediction". In: *Proceedings of the International Conference on Advances in Neural Information Processing Systems (NeurIPS)*. 2019, pp. 15398–15408.

[292] Chunqiang Tang et al. "Twine: A unified cluster management system for shared infrastructure". In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 2020, pp. 787–803.

[293] Nesime Tatbul, Ugur Çetintemel, and Stan Zdonik. "Staying Fit: Efficient Load Shedding Techniques for Distributed Stream Processing". In: *Proceedings of the 33rd International Conference on Very large Databases (VLDB)*. 2007, pp. 159–170.

[294] Nesime Tatbul et al. "Load Shedding in a Data Stream Manager". In: *Proceedings of the 29th International Conference on Very Large Databases (VLDB)*. 2003, pp. 309–320.

[295] *Tesla Firmware Upgrade Tracker*. https://ev-fw.com/reports.php.

[296] *Tesla recalls 362,000 U.S. vehicles over Full Self-Driving software*. https://www.reuters.com/business/autos-transportation/tesla-recalls-362000-us-vehicles-over-full-self-driving-software-2023-02-16/.

[297] *Tesla Talks FSD Hardware 4.0, but There Will Not Be Retrofits*. https://www.notateslaapp.com/news/1172/tesla-talks-fsd-hardware-4-0-but-there-will-not-be-retrofits.

[298] *The CARLA Autonomous Driving Challenge*. https://leaderboard.carla.org/.

[299] *The State of the Self-Driving Car Race in 2020*. https://www.bloomberg.com/features/2020-self-driving-car-race/.

[300] *The Trace Event Profiling Tool (about:tracing)*. https://www.chromium.org/developers/how-tos/trace-event-profiling-tool.

[301] Sebastian Thrun et al. "Stanley: The robot that won the DARPA Grand Challenge". In: *Journal of field Robotics* 23.9 (2006), pp. 661–692.

[302] Huangshi Tian, Yunchuan Zheng, and Wei Wang. "Characterizing and Synthesizing Task Dependencies of Data-Parallel Jobs in Alibaba Cloud". In: *Proceedings of the ACM Symposium on Cloud Computing*. 2019, pp. 139–151.

[303] Ken Tindell, Alan Burns, and Andy J Wellings. "Mode Changes In Priority Pre-Emptively Scheduled Systems". In: *Proceedings of the 13th Real-Time Systems Symposium (RTSS)*. Vol. 92. Citeseer. 1992, pp. 100–109.

[304] Muhammad Tirmazi et al. "Borg: The Next Generation". In: *Proceedings of the fifteenth European conference on computer systems*. 2020, pp. 1–14.

[305] Tom Tomazin. *A Data Center on Wheels: NVIDIA Unveils DRIVE Atlan Autonomous Vehicle Platform*. https://blogs.nvidia.com/blog/2021/04/12/nvidia-drive-atlan-autonomous-vehicle-platform/.

[306] *TorchScript*. https://pytorch.org/docs/stable/jit.html.

[307] *Toyota recalls 1.9 million cars for software glitch*. https://www.cnbc.com/2014/02/12/toyota-recalls-19-million-cars-for-software-glitch.html.

[308] Peter A. Tucker et al. "Exploiting Punctuation Semantics in Continuous Data Streams". In: *IEEE Transactions on Knowledge and Data Engineering* 15.3 (2003), pp. 555–568.

[309] Alexey Tumanov et al. "alsched: Algebraic Scheduling of Mixed Workloads in Heterogeneous Clouds". In: *Proceedings of the third ACM Symposium on Cloud Computing*. 2012, pp. 1–7.

[310] Alexey Tumanov et al. "TetriSched: Global Rescheduling with Adaptive Plan-Ahead in Dynamic Heterogeneous Clusters". In: *Proceedings of the Eleventh European Conference on Computer Systems*. 2016, pp. 1–16.

[311] Udacity. *An Open Source Self-Driving Car*. https://www.udacity.com/self-driving-car.

[312] Nicolo Valigi. *Lessons Learned Building a Self-Driving Car on ROS*. https://roscon.ros.org/2018/presentations/ROSCon2018_LessonsLearnedSelfDriving.pdf. 2018.

[313] Ashish Vaswani et al. "Attention is all you need". In: *Advances in neural information processing systems* 30 (2017).

[314] Prahlad Venkatapuram, Zhao Wang, and Chandra Mallipedi. "Custom Silicon at Facebook: A Datacenter Infrastructure Perspective on Video Transcoding and Machine Learning". In: *2020 IEEE International Electron Devices Meeting (IEDM)*. IEEE. 2020, pp. 9–7.

[315] Peng Wang et al. "The Apolloscape Open Dataset for Autonomous Driving and its Application". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2019).

[316] Stephanie Wang et al. "Lineage stash: fault tolerance off the critical path". In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 2019, pp. 338–352.

[317] Yan Wang et al. "Anytime Stereo Image Depth Estimation on Mobile Devices". In: *2019 International Conference on Robotics and Automation (ICRA)*. IEEE. 2019, pp. 5893–5900.

[318] Waymo. *Waymo Safety Report: On the Road to Fully Self-Driving*. https://storage.googleapis.com/sdc-prod/v1/safety-report/SafetyReport2018.pdf.

[319] *Wayve.AI*. https://wayve.ai.

[320] Junqing Wei, John M Dolan, and Bakhtiar Litkouhi. "Autonomous Vehicle Social Behavior for Highway Entrance Ramp Management". In: *Proceedings of the IEEE Intelligent Vehicles Symposium (IV)*. IEEE. 2013, pp. 201–207.

[321] *Welcome, Riders*. https://getcruise.com/news/blog/2022/welcome-riders/. Aug. 2021.

[322] *Welcoming our first riders in San Francisco*. https://blog.waymo.com/2021/08/welcoming-our-first-riders-in-san.html. Feb. 2022.

[323] Qizhen Weng et al. "{MLaaS} in the wild: Workload analysis and scheduling in {Large-Scale} heterogeneous {GPU} clusters". In: *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 2022, pp. 945–960.

[324] Moritz Werling et al. "Optimal Trajectory Generation for Dynamic Street Scenarios in a Frenet Frame". In: *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2010, pp. 987–993.

[325] Will Knight. *Snow and Ice Pose a Vexing Obstacle for Self-Driving Cars*. https://www.wired.com/story/snow-ice-pose-vexing-obstacle-self-driving-cars/.

[326] Nicolai Wojke, Alex Bewley, and Dietrich Paulus. "Simple Online and Realtime Tracking with a Deep Association Metric". In: *Proceedings of the 24th IEEE International Conference on Image Processing (ICIP)*. IEEE. 2017, pp. 3645–3649. DOI: 10.1109/ICIP.2017.8296962.

[327] Carole-Jean Wu et al. "Machine learning at facebook: Understanding inference at the edge". In: *2019 IEEE international symposium on high performance computer architecture (HPCA)*. IEEE. 2019, pp. 331–344.

[328] Ran Xu et al. "SMARTADAPT: Multi-branch Object Detection Framework for Videos on Mobiles". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2022, pp. 2528–2538.

[329] Wenda Xu et al. "A Real-Time Motion Planner with Trajectory Optimization for Autonomous Vehicles". In: *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2012, pp. 2061–2067.

[330] Akihiro Yamaguchi et al. "In-vehicle Distributed Time-critical Data Stream Management System for Advanced Driver Assistance". In: *Journal of Information Processing* 25 (2017), pp. 107–120.

[331] Ming Yang et al. "Avoiding Pitfalls when using NVIDIA GPUs for Real-Time Tasks in Autonomous Systems". In: *Proceedings of the 30th Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE. 2018.

[332] Yanan Yang et al. "INFless: A Native Serverless System for Low-Latency, High-Throughput Inference". In: *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '22. Lausanne, Switzerland: Association for Computing Machinery, 2022, pp. 768–781. ISBN: 9781450392051. DOI: 10.1145/3503222.3507709. URL: https://doi.org/10.1145/3503222.3507709.

[333] *YouTube recommended upload encoding settings*. https://support.google.com/youtube/answer/1722171.

[334] Jiahui Yu et al. "BigNAS: Scaling up Neural Architecture Search with Big Single-Stage Models". In: *Computer Vision – ECCV 2020*. Ed. by Andrea Vedaldi et al. Cham: Springer International Publishing, 2020, pp. 702–717. ISBN: 978-3-030-58571-6.

[335] Sergey Zagoruyko and Nikos Komodakis. "Wide residual networks". In: *arXiv preprint arXiv:1605.07146* (2016).

[336] Matei Zaharia et al. "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing". In: *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. 2012, pp. 15–28.

[337] Matei Zaharia et al. "Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing". In: *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. San Jose, California, USA, Apr. 2012, pp. 15–28. URL: http://dl.acm.org/citation.cfm?id=2228298.2228301.

[338] Matei Zaharia et al. "Spark: Cluster Computing with Working Sets". In: *2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 10)*. 2010.

[339] Wenyuan Zeng et al. "End-to-End Interpretable Neural Motion Planner". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 2019, pp. 8660–8669.

[340] Hao Zhang et al. "Dino: Detr with improved denoising anchor boxes for end-to-end object detection". In: *International Conference on Learning Representations*. 2022.

[341] Hong Zhang et al. "{SHEPHERD}: Serving {DNNs} in the Wild". In: *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 2023, pp. 787–808.

[342] Jeff Zhang et al. "{Model-Switching}: Dealing with Fluctuating Workloads in {Machine-Learning-as-a-Service} Systems". In: *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*. 2020.

[343] Minjia Zhang et al. "Accelerating Large Scale Deep Learning Inference through DeepCPU at Microsoft". In: *2019 USENIX Conference on Operational Machine Learning (OpML 19)*. 2019, pp. 5–7.

[344] Pu Zhang et al. "SR-LSTM: State Refinement for LSTM Towards Pedestrian Trajectory Prediction". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2019, pp. 12085–12094.

[345] Yueming Zhao. "GPS/IMU integrated system for land vehicle navigation based on MEMS". PhD thesis. KTH Royal Institute of Technology, 2011.

[346] Zheng Zhu et al. "Distractor-Aware Siamese Networks for Visual Object Tracking". In: *Proceedings of the European Conference on Computer Vision (ECCV)*. 2018.

[347] Shlomo Zilberstein. "Using Anytime Algorithms in Intelligent Systems". In: *AI magazine* 17.3 (1996), pp. 73–83.

[348] Barret Zoph and Quoc V Le. "Neural architecture search with reinforcement learning". In: *arXiv preprint arXiv:1611.01578* (2016).

[349] Barret Zoph et al. "Learning transferable architectures for scalable image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 8697–8710.