Towards Efficient Autonomous Vehicle Systems: A Multi-Layer Approach

Dong Li

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Engineering

Haibo Zeng, Chair
Thidapat (Tam) Chantem
Mazen Farhood
Ryan M. Gerdes
Michael S. Hsiao

June 23, 2025
Blacksburg, Virginia

Keywords: Autonomous Vehicle Systems, Intersection Management, Real-Time Scheduling, Wait-Free Multicore Data Communication

Towards Efficient Autonomous Vehicle Systems: A Multi-Layer Approach

Dong Li

(ABSTRACT)

This dissertation presents a multi-layer framework to enhance the efficiency, predictability, and scalability of autonomous vehicle (AV) systems by addressing critical challenges across system, application, and communication layers. At the system layer, a novel Extended Conflict Directed Graph (ECDG) model is introduced to enable dynamic lane assignment in unsignalized intersection management, improving flexibility and traffic flow coordination. A scheduling algorithm based on breadth-first search achieves up to 16.3% reduction in intersection evacuation time and 27.7% improvement in traffic efficiency. At the application layer, a 1-opt local search-based scheduling framework is proposed for non-preemptive real-time Directed Acyclic Graph (DAG) tasks. Using convex subspace partitioning and linear programming to minimize timing metrics such as Data Age, Reaction Time, and Time Disparity, the approach yields 20% to 40% reductions in worst-case latency with strong scalability and polynomial-time complexity. At the communication layer, the Partitioned Combined-DBP-TCCP (PCDT) protocol enables wait-free multicore data communication with configurable buffer strategies tailored to task timing requirements. Two optimization strategies, priority assignment with preemption thresholds (PA-MBTT) and wait-free-aware task partitioning (WFAP+), further reduce memory demands, achieving over 50% savings in simulation and 43% in a real-world automotive case study. Together, these contributions form an integrated and theoretically grounded solution for building real-time, safe, and

resource-efficient AV platforms, advancing the state of the art in autonomous systems and real-time embedded computing.

Towards Efficient Autonomous Vehicle Systems: A Multi-Layer Approach

Dong Li

(GENERAL AUDIENCE ABSTRACT)

Autonomous vehicles (AVs), also known as self-driving cars, have the potential to make transportation safer, more efficient, and more convenient. However, making these vehicles truly safe and reliable requires solving many complex problems, especially in how they make decisions, handle real-time tasks, and communicate within their onboard computers. This dissertation explores how to improve the systems that support autonomous driving by focusing on three important challenges. First, it proposes a smarter way for self-driving cars to safely pass through intersections that do not use traffic lights. This method allows cars to negotiate lane choices and timing with each other using connected communication, which helps reduce traffic delays and the risk of accidents. Second, it introduces a new scheduling method that helps a vehicle's computer process important tasks, like combining data from sensors or reacting to road conditions, in the right order and within strict time limits. This approach is especially useful for systems that avoid interrupting tasks once they start, which is common in modern vehicle computers. Third, the research develops a more efficient way for different parts of a vehicle's computing system to share data without delays or memory waste, which is critical in safety-focused environments. Together, these contributions help make the systems inside autonomous vehicles more dependable and efficient, bringing us closer to a future where self-driving technology can be safely used on a large scale.

Acknowledgments

This dissertation would not have been possible without the support, guidance, and encouragement of many people to whom I am deeply grateful. First and foremost, I would like to thank my advisor, Prof. Haibo Zeng, for his invaluable mentorship, insight, and patience throughout my doctoral journey. His high standards for research excellence and thoughtful guidance shaped not only this work but also my growth as a researcher. I am also thankful to my committee members for their constructive feedback and generous investment of time. I gratefully acknowledge the contributions of collaborators and colleagues whose technical discussions, paper reviews, and support in project coordination enriched the quality of this research. I am deeply thankful to my family for their unwavering love, sacrifices, and constant belief in me. In particular, I am profoundly grateful to my wife, Chengyi Lyu, thank you for your endless love, patience, and encouragement throughout every stage of this journey. Your support has been my foundation and greatest strength. Finally, I dedicate this dissertation to everyone who stood beside me with kindness and support. Your presence gave me the strength to complete this work.

Contents

List of Figures xii

List of Tables xvi

1 Introduction 1

1.1 Motivation and Background 1

1.1.1 System-Layer Challenges in Unsignalized Intersections 2

1.1.2 Application-Layer Challenges in DAG Task Scheduling 4

1.1.3 Challenges for Middleware-Layer Communication Mechanisms in Multicore 5

1.2 Overview of Approach and Results 6

1.2.1 System Layer: Dynamic Coordination at Unsignalized Intersections . 6

1.2.2 Application Layer: Scheduling of Non-Preemptive DAG Tasks 7

1.2.3 Middleware Layer: Wait-Free Communication for Multicore Systems 7

1.2.4 Summary and Chapter Guide 8

1.3 Copyright Statements from IEEE 9

2 Safe and Efficient Unsignalized Intersection Management: Dynamic Lane Assignment for Connected and Automated Vehicles 10

2.1 Abstract . . . 10
2.2 Introduction . . . 11
2.3 Problem Statement . . . 13
2.3.1 Assumptions and Notations . . . 14
2.3.2 Extended Conflict Directed Graph (ECDG) . . . 16
2.3.3 Intersection Management Problem . . . 20
2.4 Methodology . . . 22
2.4.1 Schedule with Fixed Lane Assignment . . . 23
2.4.2 Dynamic Lane Assignment . . . 27
2.4.3 Assign Time Window and Scheduled Departure Lane . . . 30
2.4.4 Time Complexity Analysis . . . 30
2.5 Experiments . . . 31
2.6 Conclusion . . . 33

3 Time-Triggered Scheduling for Non-Preemptive Real-Time DAG Tasks Using 1-Opt Local Search 35

3.1 Abstract . . . 35
3.2 Introduction . . . 36
3.3 Related Work . . . 38
3.4 System model and problem description . . . 40
3.4.1 System Model . . . 40

3.4.2 General Schedule Optimization Problem Formulation 41
3.4.3 Example Problem: End-to-end Latency Optimization 42
3.4.4 Example Problem: Time Disparity Optimization 44
3.4.5 Resource Bound Constraint: Interval Overlapping Test 46
3.4.6 Model Assumptions 47
3.4.7 Challenges 48
3.5 Job order and scheduling 48
3.5.1 Job Order 49
3.5.2 Scheduling with Job Order 50
3.6 Two-stage optimization scheduling 52
3.6.1 Optimization Concepts Review 52
3.6.2 Two-stage Optimization Method (TOM) 54
3.6.3 Theorems on 1-opt Conditions 54
3.6.4 Optimization Algorithm Towards 1-opt Schedules 57
3.7 Enhancing TOM: Strategies for Improved Performance and Efficiency 58
3.7.1 Skipping Unschedulable Job Orders 58
3.7.2 More Relaxed Constraints in LP 59
3.7.3 Simple Job Order Scheduler 60
3.8 Implementation details 61
3.8.1 Initial Solution Estimation 61

3.8.2 Faster Implementation within Time Limits 61
3.8.3 When to Assign Processor 62
3.8.4 Worst-Case Complexity Analysis 62
3.9 Extensions and Limitations 63
3.9.1 Alternative Objective Functions 63
3.9.2 Extension For Preemptive Scheduling 64
3.9.3 Finding Feasible Initial Schedules 64
3.9.4 Limitations 66
3.10 Experiment 66
3.10.1 Task Set Generation and Results 67
3.10.2 Result Analysis and Discussion 70
3.10.3 Time Disparity Optimization Result 72
3.11 Conclusions 72

4 A Wait-Free Communication Protocol for Data Consistency in Multicore Embedded Systems 74

4.1 Abstract 74
4.2 Introduction 75
4.3 Related Work 79
4.4 System Model and Background 82
4.4.1 Task Model 82

4.4.2 Data Communication Model 83
4.4.3 Response Time Analysis 83
4.5 Wait-Free Data Communication 84
4.5.1 Overview of Waif-Free Data Communication 84
4.5.2 Multicore Extensions for Existing Wait-Free Protocols 90
4.5.3 Partitioned Combined-DBP-TCCP Protocol 91
4.6 Optimizing Memory Footprint For PCDT 95
4.6.1 Preemption Thresholds with PCDT 95
4.6.2 Priority Assignment with Preemption Threshold 99
4.6.3 Task Partitioning Algorithm with Wait-Free Insights 101
4.6.4 Memory Footprint Optimization via Local Search 105
4.7 Deployment of PCDT 106
4.7.1 Software Application API 107
4.7.2 Operating System Level Support 107
4.8 Evaluation and Results 111
4.8.1 Case Study: Advanced Driver Assistant System (ADAS) Application 112
4.8.2 Simulation Experiments 118
4.9 Conclusion 123

5 Conclusion **124**

5.1 Summary of Contributions 124

5.1.1 System Layer: Intelligent Intersection Management 124

5.1.2 Application Layer: Real-Time DAG Scheduling with Non-Preemptive Execution 125

5.1.3 Middleware Layer: Wait-Free Communication for Multicore Systems 125

5.2 Implications and Broader Impact 126

5.3 Future Directions 127

Bibliography **128**

List of Figures

2.1 Example four-way intersection. This scenario is used as an example to demonstrate the proposed Extended Conflict Directed Graph (ECDG) model and all proposed methods. 15

2.2 Extended conflict directed graph (ECDG) for intersection in Fig. 2.1. Vertex 0 represents the virtual leading vehicle, while other CAVs are indexed based on their order of entry into the control zone. CAVs with diverging conflicts are connected by black unidirectional edges, those with crossing or converging conflicts are connected by red bidirectional edges, and CAVs with competing conflicts are connected by green bidirectional edges. If the efficiency offset is non-zero, it is indicated in red. Vertex estimated arrival time (EAT) weights are depicted in green, while vertex estimated travel time (ETT) weights are shown in blue. 20

2.3 Result spanning tree and schedule details of BFST method of the example intersection of 2.1. The scheduled time windows are highlighted in blue. Candidates present in the candidate list for each loop are displayed in a row and are updated in each loop. Each candidate follows the form $\langle i, d_i, p_i \rangle$, where i denotes the target vertex index, p_i is the possible parent of vertex i, and d_i represents the depth of i if connected via p_i. The chosen candidate within a loop is denoted in bold. 25

2.4 Result spanning tree and schedule details of BFST-DynaLane method of the example intersection of 2.1. The scheduled departure lanes and time windows are highlighted in blue. Candidates present in the candidate list for each loop are displayed in a row and are updated in each loop. For simplicity, the scheduled departure lanes of candidates are shown in the header of the schedule table. Each candidate follows the form $\langle i, d_i, p_i \rangle$, where i denotes the target vertex index, p_i is the possible parent of vertex i, and d_i represents the depth of i if connected via p_i. The chosen candidate within a loop is denoted in bold. 28

2.5 Average evacuation time for different traffic demands. The intersection's geospatial distribution remains consistent with the layout depicted in Fig. 2.1, in which two legs have one approach lane and one departure lane, while the other two legs have two approach lanes and two departure lanes. 32

2.6 Average evacuation time for different traffic demands. All scenarios utilized a four-way intersection, where each leg has three approach lanes and three departure lanes respectively. 33

3.1 Example DAG. 43

3.2 Longest immediate forward and backward job chains for cause-effect chain $\mathcal{C} = \{\tau_0 \rightarrow \tau_2\}$. 44

3.3 TOM intuition. The solution space is divided into multiple "sub-spaces", and the optimal solution within each sub-space can be found efficiently by solving a linear programming (LP) problem. This process is visualized above: Each job order defines a convex sub-space (because all the constraints are linear after specifying a job order) and is informally visualized as a grid in the figure above. The optimal solution within each grid is denoted as a solid circle. The original optimization problem, which needs to explore the whole solution space, is simplified into evaluating only the optimal solutions within each sub-space. 53

3.4 Main optimization framework. We begin with an initial feasible solution $\boldsymbol{s}$ and its job order $\mathcal{O}$. Then in each iteration, we search for a better job order in $\mathcal{O}$'s adjacent job order permutation $\boldsymbol{\mathcal{B}}(\mathcal{O})$ and update the best job order found yet. Eventually, the iteration will terminate at a 1-opt solution. . . . 55

3.5 Performance gap and running time for optimizing end-to-end latency and time disparity on synthetic task sets. 69

4.1 Wait-Free data communication diagram for a single data label: Task 1 generates new data in each job instance, while Task 2 and Task 3 read the most recently updated data from Task 1. As described in Section 4.5.1, data communication can occur at any point during task execution. The multi-buffer mechanism ensures wait-free operations while preserving data consistency. . 86

4.2 Wait-Free data communication with Preemption Threshold: Task 2 is assigned a preemption threshold of 2, making it mutually non-preemptive with Task 1. This configuration reduces unnecessary preemptions, allowing the total buffer requirement for the wait-free protocol to be minimized from 3 to 2 with appropriate preemption threshold settings. 99

4.3 Average memory footprint under increasing system utilization in the case study. The system consists of two cores, with a theoretical maximum utilization of 200%. Some curves terminate when the system utilization reaches 116.5% as the system becomes unschedulable with original task partitioning, where Core B reaches 92.3% utilization. When the proposed Wait-Free-Aware Partitioning (WAFP+) algorithm is applied, the system's feasible utilization range is significantly extended while preserving a stable memory usage trend. 114

4.4 Experimental evaluation of PCDT on the STM32H747I-DISCO prototype platform. (a) Breakdown of measured memory consumption under varying communication loads, showing consistency between practical and theoretical memory usage. (b) Cycle-level profiling of PCDT read/write operations on the Cortex-M7 core, demonstrating moderate write overhead due to buffer scanning. (c) PTCCP profiling for comparison, showing a slightly faster write path using circular buffering, while maintaining similar read timing. 115

4.5 Experimental evaluation on different data communication loads and scheduling policies. 119

4.6 Impact of task partitioning policies on system performance. 121

List of Tables

4.1 Task Distribution and Software Component Count in Dual-Core ADAS Vehicle MCU System 112

Chapter 1

Introduction

1.1 Motivation and Background

Autonomous Vehicle (AV) systems are rapidly emerging as a transformative force in modern transportation. These systems promise to improve road safety, improve traffic efficiency, reduce energy consumption, and introduce new paradigms of personal mobility. AVs are expected to mitigate human error, one of the leading causes of traffic accidents, while also addressing long-standing issues such as traffic congestion and inefficient use of transportation infrastructure. Realizing these promises, however, requires much more than artificial intelligence or perception algorithms. It also demands the development of highly integrated real-time embedded systems capable of performing complex computations with stringent guarantees on timing, safety, and reliability.

The operational requirements of autonomous vehicles are fundamentally different from those of traditional computing systems. AVs must continuously sense the environment, fuse sensor data, make decisions, and execute controls, all in real time. These tasks are executed on embedded platforms that often have limited computational resources and strict power constraints, particularly when deployed on edge devices. As such, AV systems must be designed to deliver high performance with predictability, robustness, and scalability under dynamic and uncertain conditions.

Much of the recent progress in autonomous driving has focused on machine learning and perception, with breakthroughs in object detection, semantic segmentation, and trajectory prediction [8, 24, 79, 83]. While these advances are essential, the timely and safe execution of perception and control algorithms depends critically on the efficiency and predictability of the underlying embedded and real-time systems. These foundational systems handle task scheduling, inter-core communication, memory management, and coordination protocols—all of which must operate reliably under strict deadlines.

Three key architectural layers are especially critical to achieving high-performance and predictable AV operation:

1. **System-layer vehicle coordination**, such as efficient scheduling at intersections without traditional traffic lights.
2. **Application-layer real-time task scheduling**, particularly for complex pipelines modeled as directed acyclic graphs (DAGs).
3. **Middleware-layer communication mechanisms for multicore platforms**, where inter-core data consistency and memory efficiency must be maintained.

1.1.1 System-Layer Challenges in Unsignalized Intersections

Intersections are among the most complex and accident-prone segments of urban road networks. According to data from the Federal Highway Administration, intersections accounted for over 27.4% of traffic fatalities in the United States in 2022 [33], with roughly half of all traffic injuries also occurring at these locations. The need for safer and more efficient intersection management is both urgent and well-documented.

The introduction of Connected and Automated Vehicles (CAVs) presents a transforma-

tive opportunity to re-imagine intersection control. CAVs can leverage vehicle-to-vehicle (V2V) and vehicle-to-infrastructure (V2I) communication technologies [74] to coordinate their movement in a decentralized and intelligent manner, potentially eliminating the need for traditional traffic signals. This paradigm shift holds the promise of reducing delays, minimizing collisions, and increasing intersection throughput. However, achieving this vision requires the development of robust algorithms capable of handling a wide variety of intersection layouts, vehicle trajectories, and traffic conditions, all in real time.

Numerous approaches have been proposed for autonomous intersection management. These include Model Predictive Control (MPC)[4], centralized optimization techniques[73, 82, 110], and Dynamic Programming (DP)-based scheduling [108]. While these methods offer strong theoretical performance, they often rely on highly accurate vehicle models and require substantial computational resources, which can hinder real-time deployment in dynamic traffic environments.

To address these limitations, some researchers have proposed simplifications such as virtual platooning, conflict point reservation, or discretized grid-based methods. Another promising class of approaches uses graph-based abstractions [20, 22, 66, 105], in which vehicle trajectories and conflicts are encoded as graph elements. Among these, Conflict Directed Graphs (CDGs) have shown particular promise for scalable and flexible intersection scheduling.

Despite these advancements, an important limitation of many graph-based approaches is the assumption of fixed departure lanes. Most existing models treat lane changing and intersection scheduling as sequential, decoupled processes. This restricts scheduling flexibility and often leads to suboptimal traffic flow. Prior work on "all-direction turn lanes" [47, 50] and "flexible lane directions" [19] provides valuable insights into adaptable infrastructure, but these methods typically constrain each vehicle to a predetermined lane selection strategy. There also remains a gap in the literature for methods that integrate dynamic lane

assignment directly into intersection scheduling decisions.

1.1.2 Application-Layer Challenges in DAG Task Scheduling

Autonomous vehicle applications increasingly rely on computational pipelines that consist of tightly coupled tasks with complex data dependencies. These pipelines are frequently modeled as directed acyclic graphs (DAGs), where each node represents a task and edges indicate data or control dependencies [1, 32, 41, 42, 55, 70, 95]. Scheduling such DAGs on embedded processors is a nontrivial challenge, especially under nonpreemptive execution constraints commonly used on GPU-like architectures.

The most critical performance metrics in this domain include the Data Age, Reaction Time, and Time Disparity (DARTD) of cause-effect chains. These metrics directly affect the responsiveness and control stability of the AV system. Ensuring low DARTD is essential for real-time control systems that operate on streaming sensor data.

Traditional methods for scheduling DAG tasks include greedy heuristics [92, 100] and metaheuristic algorithms [95, 123, 128], which offer general applicability and modest computational costs. However, these approaches lack performance guarantees and often fail to scale to larger DAGs. Exact optimization techniques, such as Integer Linear Programming, provide strong optimality bounds but suffer from prohibitive computational complexity.

There is a pressing need for scheduling algorithms that combine scalability with strong performance bounds in minimizing DARTD, especially under the non-preemptive constraints common in real-time AV platforms.

1.1.3 Challenges for Middleware-Layer Communication Mechanisms in Multicore

Modern AV platforms increasingly deploy multicore embedded processors to support parallel execution of perception, planning, and control tasks. While multicore architectures offer significant performance benefits, they also introduce complexity in managing communication between tasks, particularly when tasks are mapped to different cores.

Efficient inter-core communication is essential for maintaining data consistency and timing predictability in real-time systems. Traditional lock-based synchronization protocols, such as the Priority Ceiling Protocol [87] and Stack Resource Policy [9], are widely used for protecting shared resources. However, these mechanisms introduce blocking time and priority inversion, which are highly undesirable in systems with strict timing requirements. These effects are unacceptable in hard real-time systems where deadline misses can compromise safety.

As a result, lock-free and wait-free communication protocols have gained attention. Wait-free methods are particularly attractive because they provide guaranteed bounds on execution time, eliminating the risk of blocking entirely. However, most wait-free protocols replicate data across multiple buffers to preserve consistency, leading to high memory consumption. This is especially problematic in memory-constrained embedded systems.

Furthermore, existing wait-free strategies often neglect global system-level factors such as task priorities, preemption behavior, and partitioning policies. There remains a gap in integrating communication protocols with system-aware scheduling and memory optimization strategies that work efficiently in multicore platforms.

1.2 Overview of Approach and Results

This dissertation proposes a multi-layer optimization framework to address key inefficiencies and challenges in autonomous vehicle systems. It focuses on three interconnected layers: (1) system-layer vehicle coordination at unsignalized intersections, (2) application-layer scheduling of non-preemptive real-time DAG tasks, and (3) middleware-layer wait-free communication in multicore embedded systems. Each layer contributes a set of novel methodologies supported by theoretical analysis and experimental evaluation. Together, they form a cohesive architecture for building scalable, safe, and efficient AV systems.

1.2.1 System Layer: Dynamic Coordination at Unsignalized Intersections

To enhance coordination and throughput in unsignalized intersections, this work introduces the Extended Conflict Directed Graph (ECDG) model. Unlike previous graph-based approaches that assume fixed departure lanes, the ECDG model incorporates dynamic lane assignment directly into the scheduling process. This enables vehicles to dynamically select from multiple departure lanes based on current intersection conditions, eliminating unnecessary lane changes and improving scheduling flexibility.

A breadth-first-search-based scheduling algorithm is developed to compute conflict-free time windows for each vehicle, considering both vehicle dynamics and route information. Experimental results show that this dynamic lane assignment strategy reduces intersection evacuation time by up to 16.3% and improves overall traffic efficiency by as much as 27.7% compared to state-of-the-art methods.

1.2.2 Application Layer: Scheduling of Non-Preemptive DAG Tasks

For real-time AV applications modeled as Directed Acyclic Graphs (DAGs), a new scheduling framework is proposed to minimize critical timing metrics such as Data Age, Reaction Time, and Time Disparity (DARTD). These metrics are essential for ensuring timely data processing and stable control in safety-critical systems.

This dissertation presents a 1-opt local search-based scheduling algorithm that overcomes the non-convex and discontinuous nature of DARTD. The solution space is partitioned into convex subspaces, each solved using linear programming. An iterative refinement procedure traverses these subspaces to reach a locally optimal solution under 1-opt criteria.

The algorithm offers polynomial-time complexity and consistently achieves 20% to 40% reductions in worst-case latency compared to existing heuristic and metaheuristic approaches. It also scales effectively to large DAG task sets, making it well-suited for embedded platforms in AVs.

1.2.3 Middleware Layer: Wait-Free Communication for Multicore Systems

To address inefficiencies in inter-core data sharing, this work introduces the Partitioned Combined-DBP-TCCP (PCDT) protocol. This hybrid wait-free communication scheme allows reader tasks to choose between two buffer protocols based on their individual timing constraints: the Dynamic Buffering Protocol (DBP) and the Temporal Concurrency Control Protocol (TCCP). This design judiciously balances the trade-off between timing predictability and memory efficiency.

To further improve system performance, two system optimization techniques are developed:

- PA-MBTT (Priority Assignment with Maximum Blocking Time Tolerance): A strategy that minimizes buffer usage through priority assignment and preemption threshold tuning.

- WFAP and WFAP+ (Wait-Free-Aware Partitioning): Task partitioning heuristics that maximize buffer reuse across multicore processors. WFAP+ includes a local search enhancement that further reduces communication overhead.

These techniques together provide substantial memory savings. Simulation results show an average memory reduction of over 50%, and in an automotive ADAS case study, the integrated system achieved a 43% reduction in memory usage while maintaining real-time schedulability.

1.2.4 Summary and Chapter Guide

Each of the three technical contributions forms a layer in the proposed multi-layer AV system architecture. Their integration enables end-to-end improvements in safety, performance, and efficiency, making this dissertation a step toward scalable and dependable AV system deployment.

The remainder of this dissertation is organized as follows:

- Chapter 2 presents the contribution in the system layer: intelligent intersection management using the Extended Conflict Directed Graph (ECDG) and dynamic lane assignment strategies.

- Chapter 3 addresses the application layer, introducing the 1-opt scheduling framework for non-preemptive DAG tasks under real-time constraints.

- Chapter 4 discusses the middleware layer, including the PCDT protocol, PA-MBTT priority assignment, and WFAP/WFAP+ task partitioning for efficient multicore communication.
- Chapter 5 concludes the dissertation with a summary of contributions and outlines future research directions.

1.3 Copyright Statements from IEEE

The chapters included in this dissertation are based on research papers previously published by the IEEE and are reprinted here with permission.

In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of Virginia Polytechnic Institute and State University's products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to http://www.ieee.org/publications_standards/publications/rights/rights_link.html to learn how to obtain a License from RightsLink. If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.

Chapter 2

Safe and Efficient Unsignalized Intersection Management: Dynamic Lane Assignment for Connected and Automated Vehicles

2.1 Abstract

Intersections present significant challenges in traffic safety and efficiency. The advent of Connected and Automated Vehicles (CAVs) holds promise in revolutionizing transportation systems by enhancing efficiency, safety, and congestion mitigation. However, existing literature often overlooks critical aspects such as lane assignment, which, if managed properly, can greatly improve the efficiency of intersections. This chapter addresses this gap by proposing a novel strategy that enables CAVs to dynamically assign departure lanes when traveling through unsignalized intersections. Unlike existing approaches that restrict departure lane choice, our framework leverages all available departure lanes whenever possible. The Extended Conflict Directed Graph (ECDG) is proposed to model the intersection, integrating vehicle dynamics and intersection geospatial distribution for improved efficiency. Additionally, breadth-first-search-based methods are proposed to solve the problems efficiently. Ex-

tensive experiments demonstrate significant improvements in intersection evacuation time, with up to a 27.7% enhancement in traffic efficiency compared to state-of-the-art methods.

2.2 Introduction

In 2022, intersections were associated with over 27.4% of total traffic fatalities in the United States, with approximately half of all traffic injuries occurring at intersections, as reported by Federal Highway Administration [33]. Connected and Automated Vehicles (CAVs), as a revolutionary technology, promise to enhance transportation efficiency, improve road safety, and alleviate intersection congestion. Moreover, in scenarios where only CAVs are operational, traffic lights may even be eliminated as vehicle-to-vehicle (V2V)and vehicle-to-infrastructure (V2I) communication technologies advance [74]. Realizing these potential improvements necessitates carefully designed algorithms to address the challenges posed by traffic modeling and scheduling problems of complex intersection scenarios.

Previous approaches to intersection management, such as Model Predictive Control[4], Optimization-based techniques [73, 82, 110], and Dynamic Programming algorithms [108], have demonstrated effectiveness in optimizing traffic flow at unsignalized intersection. However, these methods typically rely on accurate models of vehicle dynamics and often require significant computational resources, making them impractical for real-time applications. In addition to optimal scheduling, some studies simplify intersection models by employing virtual platooning [130] or mixed platooning [104] to consolidate vehicles from all directions into a simplified one-dimensional scheduling problem. Lin *et al.*[65] subdivided intersections into multiple grids and proposed a grid reservation method. Furthermore, graph-based approaches offer an alternative model that separates the complexities of vehicle dynamics and control mechanisms from intersection management. Li and Wang [63] introduced the spanning

tree method to address the intersection scheduling problem, while Monte Carlo Tree search methods [106, 107] have been proposed to enhance efficiency. Noteworthy graph-based models, including Conflict Directed Graphs (CDGs) and their variants, have been increasingly utilized for intersection management in recent years [20, 22, 66, 105]. By abstracting intersection geometries and vehicle behaviors into a graph representation, these methods enable efficient scheduling and coordination of CAVs while maintaining safety and reliability.

Despite these advancements, existing graph-based literature often neglects the critical aspect of lane assignment in intersection management. Many approaches consider fixed lane direction and treat lane changing and intersection scheduling as separate, sequential processes [21, 99]. CAVs need to change approach lanes before going in their target directions. The concept of all-direction turn lanes has been proposed to mitigate the need for lane changes before CAVs enter the intersection [47, 50], enabling CAVs to travel in any direction from any approach lane. Similarly, Cai *et al.*[19] introduced the idea of flexible lane direction. However, these methods often assume that CAVs can only transition to a single predetermined departure lane in each direction, which imposes limitations on their performance.

Our research aims to bridge this gap by proposing a novel approach that allows CAVs to dynamically assign any available departure lanes while considering intersection scheduling for CAVs. This approach removes the need for lane changes before entering the intersection, thereby reducing the risk of collisions. It also provides flexible choices for the departure lane, so CAVs can be dynamically adjusted to mitigate conflicts and improve traffic efficiency. Additionally, we proposed the Extended Conflict Directed Graph (ECDG) model to incorporate the dynamic lane assignment framework. To solve the intersection management problem, breadth-first-search-based methods are proposed, in which CAVs are assigned customized time windows to safely pass the intersection according to their routes and dynamic characteristics. The contributions of this chapter are summarized as follows:

1. We introduce the dynamic lane assignment strategy for unsignalized intersection management, allowing CAVs to utilize all available departure lanes in the target leg.

2. We propose the Extended Conflict Directed Graph (ECDG) model of intersections, which incorporates competing conflicts to model the competition introduced by dynamic lane assignment. We also integrate the vehicle dynamics and intersection geospatial distribution into the ECGD model, facilitating further enhancements to traffic efficiency.

3. We propose innovative methods to tackle the dynamic lane assignment problem. Extensive experiments demonstrate significant improvements in intersection evacuation time and show that our approach can enhance traffic efficiency by up to 27.7% compared to the state-of-the-art method.

This chapter is from the author's previous work, Dong Li, Sen Wang, and Haibo Zeng. "Safe and Efficient Unsignalized Intersection Management with Breadth-First Spanning Tree," In *2024 IEEE 27th International Conference on Intelligent Transportation Systems (ITSC)* [61].

The remainder of this chapter is organized as follows: In Section 2.3, we introduce the ECDG model and provide the formal problem definition. Section 2.4 outlines methods proposed to solve the intersection management problem under fixed and dynamic lane assignment, respectively. In Section 2.5, we present the experimental results, and finally, the chapter is concluded in Section 2.6.

2.3 Problem Statement

We consider the unsignalized intersection scenarios where only Connected and Automated Vehicles (CAVs) are present. CAVs transmit real-time status and intended departure leg data

upon entering the control zone to a central controller, potentially located within roadside units. This controller coordinates CAV movements, advising optimal passage times through the intersection. Communication and coordination occur via V2V and V2I communication protocols, and the need for traditional traffic signal infrastructure is eliminated.

We address the intricate challenge of intersection management through a scheduling paradigm. This involves formulating a scheduling problem to allocate departure lanes and distinct time windows to individual vehicles, ensuring conflict-free crossings within the intersection to prevent collisions. We model this scheduling problem using the Extended Conflict Directed Graph, detailed in Section 2.3.2. Additionally, we aim to enhance intersection efficiency while addressing the scheduling problem. A key metric for evaluating our approaches is evacuation time, representing the duration for all vehicles to pass through the intersection conflict area. Thus, our paper has a dual objective: ensuring intersection safety and optimizing evacuation time.

2.3.1 Assumptions and Notations

Here, we introduce assumptions about the unsignalized intersection scenario to focus on fundamental aspects of our research and develop more efficient algorithms.

Assumption 1. All CAVs are equipped with ample computational resources, enabling them to engage in fully autonomous driving.

Assumption 2. All CAVs and the Road Side Units (RSU) are seamlessly interconnected, where packet loss is non-existent, and communication delays are negligible.

We assume right-hand traffic. Intersection legs and lanes are indexed clockwise, where Leg_i represents the i^{th} leg and L_i represents the i^{th} lane. The total number of legs and

total number of lanes is determined by the intersection's geospatial characteristics. CAVs are numbered according to their order of entry into the control zone, and we use CAV_i to represent the i^{th} CAV within the intersection. Upon entering the control zone, CAV_i will use a fixed approach lane, denoted as L_i^a, to approach the intersection. Each CAV_i also has an initial intended departure lane, denoted as $L_i^{d\_init}$, from which it intends to depart based on its routing. The actual scheduled departure lane, denoted as $L_i^{d\_sched}$, may differ from $L_i^{d\_init}$ if a dynamic lane assignment strategy is employed, as detailed in Definition 2.3. The route $R_i : L_i^a \rightarrow L_i^{d\_sched}$ of CAV_i in the intersection is explicitly determined by CAV_i's approach lane and scheduled departure lane. The physical path within the intersection conflict area is solely decided by the route and thus is decoupled with CAV's path planning policy. Two CAVs will follow the same route and path as long as they share the same approach lane and scheduled departure lane.

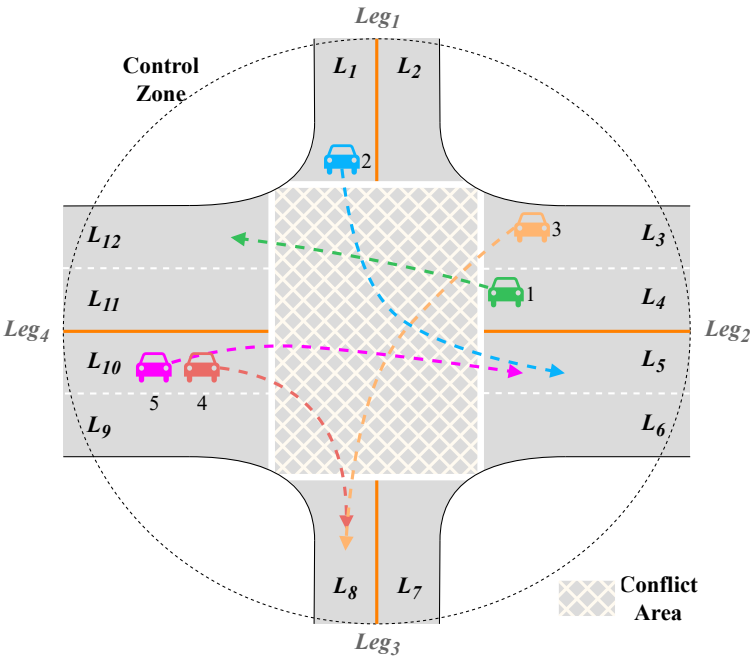


Figure 2.1: Example four-way intersection. This scenario is used as an example to demonstrate the proposed Extended Conflict Directed Graph (ECDG) model and all proposed methods.

2.3.2 Extended Conflict Directed Graph (ECDG)

Fig. 2.1 illustrates a typical intersection scenario where five CAVs are navigating a four-way intersection. Dashed arrows represent the routes of all CAVs, starting from their approach lanes and pointing towards their initial intended departure lanes. The control zone encompasses a predefined area surrounding the intersection's central point. Upon entering this zone, a CAV initiates communication with the intersection manager to receive control instructions. The intersection conflict area refers to the convergence central zone of all lanes, where multiple CAVs may encounter conflicting routes.

Xu *et al.*[105] and Chen *et al.*[22] utilized Conflict Directed Graphs (CDG) to model intersections. However, this model assumed a uniform crossing time for all CAVs traversing the intersection, overlooking vehicle dynamics and arrival time constraints. To enhance the intersection model with more detailed information and devise more efficient schedules, this chapter extends the CDG model by introducing the Extended Conflict Directed Graph (ECDG). Vertex and edge attributes relevant to estimated arrival time, estimated travel time, and efficiency offset are added to the ECDG. Additionally, we introduce a novel competing conflict relationship alongside conventional conflict relationships.

Conflict Relationship: For every pair of distinct CAVs, conflict relationships are determined by their respective routes. The following four conflict relationships are kept from CDG [22].

a) Crossing Conflict: Two CAVs exhibit a crossing conflict when their routes intersect within the conflict area.

b) Diverging Conflict: Two CAVs experience a diverging conflict when they share the same approach lane.

c) Converging Conflict: Two CAVs engage in a converging conflict when they share the same

departure lane.

d) Non-conflicting relationship: Two CAVs are deemed non-conflicting if they do not share any other conflict relationships.

Taking the intersection in Fig. 2.1 as an example, CAV_1 and CAV_2 experience a crossing conflict, CAV_4 and CAV_5 have a diverging conflict, CAV_3 and CAV_4 encounter a converging conflict, while CAV_1 and CAV_5 are non-conflicting.

At a typical intersection, multiple vehicles often converge onto the same departure lane from different approach lanes, leaving other lanes underutilized. In Fig. 2.1, for example, both CAV_2 and CAV_5 intend to depart the intersection using L_5 in Leg_2. If these two CAVs are the only ones present in the control zone, the intersection would require a time greater than the sum of their travel times to clear the conflict area. However, an alternative exists: lane L_6 in Leg_2 is available and underutilized. By dynamically reallocating CAV_5 to depart using L_6, both CAVs can pass through simultaneously, potentially halving the total travel time for this scenario. We introduce a new competing conflict in the ECDG model to model such interactions.

e) Competing Conflict: When two CAVs from different approach lanes intend to depart to the same leg with multiple departure lanes, they are said to be in a competing conflict relationship. In Fig. 2.1, CAV_2 and CAV_5 are competing for Leg_2. Notably, two CAVs don't need to have the same departure lane to be in a competing conflict.

The competing conflicts identify the bottlenecks of the traffic flow and can be resolved by dynamically assigning departure lanes to competing CAVs. Experimental results demonstrate that dynamic lane assignment, as a solution to competing conflicts, can improve intersection evacuation time by up to 16.3%. Details are introduced in Section 2.4.2.

Note that a single pair of CAVs can have multiple types of conflict relationships simultaneously. To simplify our notation throughout this chapter, we assign symbols to all conflict relationships: $\{\mathbf{C}_{cro}, \mathbf{C}_{div}, \mathbf{C}_{conv}, \mathbf{C}_{no}, \mathbf{C}_{comp}\}$, representing crossing, diverging, converging, non-conflicting and competing conflicts, respectively. We denote the conflict relationships between CAV_i and CAV_j as a set $CR(i,j)$, which may only include the provided symbols. For instance, in Fig. 2.1, CAV_2 and CAV_5 experience both converging and competing conflicts, then we have $CR(2,5) = \{\mathbf{C}_{conv}, \mathbf{C}_{comp}\}$.

Vertex Attributes: Two key vertex (vehicle) attributes are considered in the scheduling problem to account for vehicle dynamics and intersection geospatial distribution: 1) Estimated Arrival Time (EAT): This represents the earliest time that a CAV can arrive at the intersection stop line, accounting for traffic conditions and vehicle dynamic constraints. A general estimate typically considers the free-flow arrival time of the vehicle. 2) Estimated Travel Time (ETT): This denotes the actual time a CAV requires to pass through the conflict area, determined by the CAV's route and the geospatial information of the intersection.

Edge Attributes: Efficiency offsets, treated as edge attributes in an ECDG, are applied to diverging vehicles to accommodate the reduced time when scheduled together rather than separately. This is because when two diverging vehicles are scheduled consecutively, the latter can pass through the intersection more swiftly without stopping at the stop line, thereby saving time. Efficiency offsets typically have negative values and are excluded for edges connected to the virtual leading vehicle (CAV_0) or bidirectional edges, where offsets are consistently set as 0 for clarity and consistency.

Definition 2.1 (Extended Conflict Directed Graph)**.** The Extended Conflict Directed Graph (ECDG) serves as the framework for modeling the traffic intersections. An ECDG is denoted as $\mathcal{G} = (\mathcal{V}, \mathcal{E}, w_{\mathcal{V}}^{EAT}, w_{\mathcal{V}}^{ETT}, w_{\mathcal{E}})$, which contains a vertex set $\mathcal{V}$, an edge set $\mathcal{E}$, vertex EAT weights $w_{\mathcal{V}}^{EAT}$, vertex ETT weights $w_{\mathcal{V}}^{ETT}$ and edge weights $w_{\mathcal{E}}$.

$\mathcal{V} = \{0, 1, 2 \dots N\}$ is the vertex set in which each CAV is represented as a single vertex and vertex 0 symbolizes a virtual leading vehicle. N is the total number of CAVs present in the intersection control zone. For simplicity in notation, a vertex will also be used to signify the vehicle it represents.

$\mathcal{E}$ is the edge set containing directional edges. We use (i, j) to represent a unidirectional edge from vertex i to vertex j in $\mathcal{G}$. All conflict relationships are captured using edges in the ECDG. Diverging conflicts impose strict passing order requirements for two CAVs, hence they are represented using unidirectional edges. If $CR(i, j)$ includes a diverging conflict, there exists a unidirectional edge satisfying Eq. (2.1). The virtual leading vehicle CAV_0 defaults to having a diverging conflict with every other CAV, represented as in Eq. (2.2). Crossing and converging conflicts require that two CAVs cannot simultaneously pass through the intersection for safety, but their passing order remains unrestricted. CAVs experiencing these conflicts are mutually exclusive and are linked by bidirectional edges, as in Eq. (2.3). Competing conflicts arise within crossing or converging conflicts, hence they are also depicted by bidirectional edges. Non-conflicting CAVs have no edges between them.

$$(i, j) \in \mathcal{E} \ \& \ i < j, \quad \text{if} \ \ \mathbf{C}_{div} \in CR(i, j) \tag{2.1}$$

$$(0, i) \in \mathcal{E}, \ \forall i \in \mathcal{V} \ \& \ i \neq 0 \tag{2.2}$$

$$(i, j) \in \mathcal{E} \ \& \ (j, i) \in \mathcal{E}, \quad \text{if } CR(i, j) \cap \{\mathbf{C}_{cro}, \mathbf{C}_{conv}, \mathbf{C}_{comp}\} \neq \emptyset \tag{2.3}$$

$w_{\mathcal{E}}$ represent the edge weights that assign efficiency offsets to all edges. For bidirectional

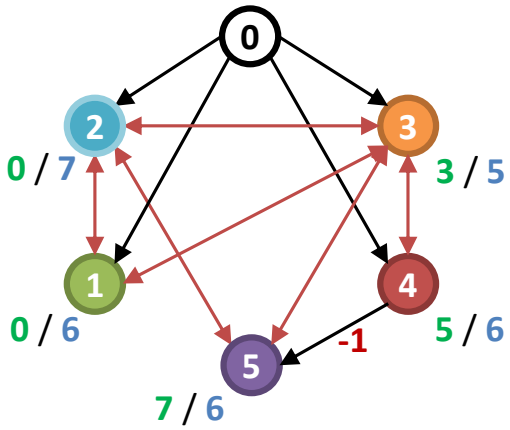


Figure 2.2: Extended conflict directed graph (ECDG) for intersection in Fig. 2.1. Vertex 0 represents the virtual leading vehicle, while other CAVs are indexed based on their order of entry into the control zone. CAVs with diverging conflicts are connected by black unidirectional edges, those with crossing or converging conflicts are connected by red bidirectional edges, and CAVs with competing conflicts are connected by green bidirectional edges. If the efficiency offset is non-zero, it is indicated in red. Vertex estimated arrival time (EAT) weights are depicted in green, while vertex estimated travel time (ETT) weights are shown in blue.

edges, the same weight is applied in both directions, adhering to Eq. (2.4).

$$w_{\mathcal{E}}(i,j) = w_{\mathcal{E}}(j,i), \text{ if } (i,j) \in \mathcal{E} \ \& \ (j,i) \in \mathcal{E} \tag{2.4}$$

$w_{\mathcal{V}}^{EAT}$ and $w_{\mathcal{V}}^{ETT}$ denote the vertex weights for vehicle estimated arrival time and vehicle estimated travel time respectively, with $w_{\mathcal{V}}^{EAT}(0) = 0$ and $w_{\mathcal{V}}^{ETT}(0) = 0$.

Fig 2.2 provides an illustrative example of an ECDG for the intersection scenario depicted in Fig. 2.1.

2.3.3 Intersection Management Problem

The main objective of intersection management is to safely and efficiently guide all CAVs through the intersection in the shortest time possible. Each CAV is assigned a scheduled

departure lane and a designated time window for traversing the intersection conflict area. The considered unsignalized intersection management problem is formally defined in Problem 1.

Definition 2.2 (Time Window)**.** The time window for CAV_i, denoted as $window_i = [t_i^a, t_i^d)$, represents the authorized time interval for occupying the intersection conflict area. It satisfies the conditions specified in (2.5), where t_i^a denotes the earliest time to approach the conflict area, and t_i^d indicates the latest departure time for CAV i from the conflict area.

$$t_i^a \geq w_{\mathcal{V}}^{EAT}(i) \textbf{and} t_i^d - t_i^a \geq w_{\mathcal{V}}^{ETT}(i) \tag{2.5}$$

The virtual leading vehicle has a fixed time window $window_0 = [-1, 0)$, which means the virtual leading vehicle will pass through the intersection before time 0.

Definition 2.3 (Scheduled Departure Lane)**.** The scheduled departure lane $L_i^{d\_sched}$ for CAV_i is the actual lane from which CAV_i departs the intersection.

The $L_i^{d\_sched}$ may differ from the initial intended departure lane $L_i^{d\_init}$ due to dynamic lane assignment. However, both $L_i^{d\_sched}$ and $L_i^{d\_init}$ must belong to the same leg to ensure path consistency.

Definition 2.4 (Intersection Schedule)**.** An intersection schedule includes time windows and scheduled departure lanes for all CAVs within the intersection control zone.

Theorem 2.5 (Safe Schedule)**.** *An intersection schedule is deemed safe if it satisfies Eq. (2.6), where conflict relationships are determined by the scheduled departure lanes.*

$$\forall i, j \ window_i \cap window_j = \varnothing, \quad if\ CR(i,j) \cap \{\mathbf{C}_{cro}, \mathbf{C}_{div}, \mathbf{C}_{conv}\} \neq \emptyset \tag{2.6}$$

Proof. Assuming that the intersection schedule is unsafe, there must exist two CAVs, denoted as CAV_i and CAV_j, colliding within the intersection conflict area. Consequently, CAV_i and CAV_j must hold at least one crossing, diverging, or converging conflict. Since both CAVs are present in the conflict area simultaneously at some point in time, this violates Eq. (2.6). □

Definition 2.6 (Intersection Evacuation Time)**.** The evacuation time T of the intersection is defined as the time when the last CAV departs the intersection conflict area.

$$T = \max_{1 \le i \le N} t_i^d \tag{2.7}$$

Problem 1 (Intersection Management)**.** Given an unsignalized intersection and its ECDG $\mathcal{G}$, the intersection management problem aims to find a safe schedule with the minimum evacuation time T.

2.4 Methodology

This section presents two approaches to solve Problem 1 under fixed and dynamic lane assignment scenarios, respectively. The fundamental idea is to extract a minimum-depth spanning tree from the ECDG model $\mathcal{G}$ and generate an intersection schedule based on the precedence constraints specified by the spanning tree.

We propose utilizing a breadth-first search to find the spanning tree with the smallest possible depth. This approach differs from the depth-first spanning tree method utilized by Chen *et al.*[22], which relies on depth-first searching. Importantly, this isn't merely a switch of searching methods; rather, specifically designed algorithms are tailored for the ECDG model. Starting from the virtual leading vehicle as the root, each iteration aims to identify a conflict-free vertex that minimally increases the modified tree depth, as defined in Definition 2.8,

subsequently adding it to the tree. Throughout this process, conflict relationships, vertex attributes, and edge attributes are carefully considered and updated, particularly when CAVs are dynamically allocated new departure lanes. Ultimately, the spanning tree is transformed into the intersection schedule.

Definition 2.7 (Spanning Tree)**.** *The Spanning Tree (ST) of a directed graph $\mathcal{G}$ rooted at root is a subgraph $\mathcal{T}$ of $\mathcal{G}$ such that $\mathcal{T}$ forms a directed tree and there exist unique directed paths from the root to every other vertex in $\mathcal{G}$.*

Definition 2.8 (Depth of Vertex and ST)**.** *In a given ST, the depth of the root vertex d_{root} is 0. For any other vertex i, there exists an unique directed path from the root to vertex i: $path_i = \{root, p_1, p_2, ..., p_k, i\}$. The depth of vertex i, denoted as d_i, is calculated as $d_i = \max(d_{p_k}, w_{\mathcal{V}}^{EAT}(i)) + w_{\mathcal{E}}(p_k, i) + w_{\mathcal{V}}^{ETT}(i)$. The depth of an ST is the maximum vertex depth within it, i.e. $d_{ST} = \max_{\forall i \in \mathcal{V}} d_i$.*

The depth of a vertex is carefully designed to account for both the precedence constraints illustrated by the spanning tree and the vehicle dynamic and intersection geospatial constraints. The depth can be safely set to the latest departure time in the vertex's time window. This calculation is performed recursively for each vertex.

2.4.1 Schedule with Fixed Lane Assignment

The Breadth-First Spanning Tree (BFST) method is employed for fixed lane assignment scenarios. BFST aims to expand the tree with the minimum increase in tree depth. A vertex is eligible for inclusion in the tree only if there exists an edge in $\mathcal{G}$ that connects it to a vertex already added to the tree.

Definition 2.9 (Vertex Schedule)**.** The process of adding a vertex into the tree is referred

to as scheduling a vertex. A vertex is deemed scheduled if it has been added to the tree; otherwise, it remains an unscheduled vertex.

To facilitate the selection of vertices and the construction of the tree, a meticulously designed candidate data structure, as defined in Definition 2.10, is employed. This structure generates a candidate tuple for each unscheduled vertex and collects them into a candidate list. The candidate list is dynamically updated in each iteration, and the candidate with the minimum depth is selected for connection to the tree. This iterative process continues until all vertices are scheduled.

Definition 2.10 (Candidate Data Structure)**.** In the context of the BFST method, a candidate data structure is represented by a tuple $\langle i, d_i, p_i \rangle$, where i denotes the target vertex index, p_i is the possible parent of vertex i, and d_i represents the depth of i if connected via p_i.

Algorithm 1 outlines the pseudocode of the BFST heuristic. Following initialization, the algorithm proceeds through a primary while loop to iteratively incorporate vertices into the tree. Within each iteration, the algorithm selects the candidate with the smallest depth from the candidate list, adds it to the tree, updates the candidate list by removing conflicting candidates and introducing new candidates, and sorts the candidate list in preparation for the next iteration.

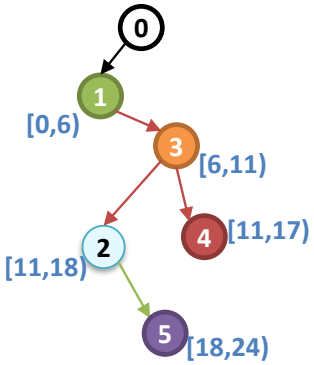
During initialization (lines 1 - 5 of Algorithm 1), the candidate list begins with only the virtual leading vehicle to ensure its selection as the root. Subsequently, the unidirectional parents for all vertices are identified. A vertex cannot be scheduled until all its unidirectional parent vertices have been scheduled.

The candidate list is sorted based on depth in non-decreasing order. In each iteration, the vertex with the smallest depth in the candidate list is incorporated into the tree and removed

from the list (lines 7 - 12). Adding a new vertex into the spanning tree may invalidate other candidates due to conflicts with unscheduled vertices; thus, incompatible candidates are removed from the list (lines 13 - 19). New candidates are subsequently introduced to the list based on the scheduled vertices.

The sets $\mathcal{V}_{bfst}$ and $\mathcal{E}_{bfst}$ are employed to record the vertices and edges chosen for inclusion in the spanning tree, respectively. After the iteration terminates, the scheduled BFST can be constructed using $\mathcal{V}_{bfst}$ and $\mathcal{E}_{bfst}$. The scheduled time window and departure lane can be readily determined following Algorithm 3. Specifically, the time window for each vehicle ends at the depth of the corresponding vertex, while the scheduled departure lane remains consistent with the initially intended departure lane.

Fig. 2.3 illustrates the schedule details of the BFST method, showcasing an example result spanning tree for the intersection depicted in Fig. 2.1. The evacuation time of the intersection is 24 seconds.



Loop	0	1	2	3	4	5
#0	$\langle \mathbf{0,0,-1} \rangle$	NA	NA	NA	NA	NA
#1		$\langle \mathbf{1,6,0} \rangle$	⟨2,7,0⟩	⟨3,8,0⟩	⟨4,11,0⟩	NA
#2			⟨2,13,1⟩	$\langle \mathbf{3,11,1} \rangle$	⟨4,11,0⟩	NA
#3			⟨2,18,3⟩		$\langle \mathbf{4,17,3} \rangle$	NA
#4			$\langle \mathbf{2,18,3} \rangle$			⟨5,22,4⟩
#5						$\langle \mathbf{5,24,2} \rangle$

Figure 2.3: Result spanning tree and schedule details of BFST method of the example intersection of 2.1. The scheduled time windows are highlighted in blue. Candidates present in the candidate list for each loop are displayed in a row and are updated in each loop. Each candidate follows the form $\langle i, d_i, p_i \rangle$, where i denotes the target vertex index, p_i is the possible parent of vertex i, and d_i represents the depth of i if connected via p_i. The chosen candidate within a loop is denoted in bold.

Algorithm 1 Breadth-First Spanning Tree Method.

Input Extended Conflict Directed Graph $\mathcal{G}$
Output: Breadth-First Spanning Tree $\mathcal{T}_{bfst}$

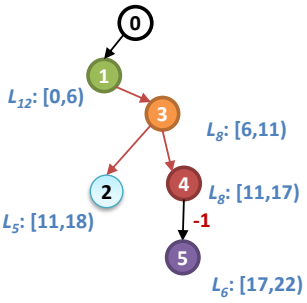
$candidate\_list = \{\langle 0, 0, -1\rangle\}$
$\mathcal{V}_{bfst} = \emptyset$, $\mathcal{E}_{bfst} = \emptyset$
for $j = 0$ to N **do**
    $\mathcal{P}_u^j = \{i \in \mathcal{V} \mid \mathbf{C}_{div} \in CR(i,j)\}$ // *unidirectional parents*
end for
while $candidate\_list \neq \emptyset$ **do**
    $\langle i, d_i, p_i\rangle = candidate\_list$.front()
    $\mathcal{V}_{bfst} = \mathcal{V}_{bfst} \cup \{i\}$
    if $i \neq 0$ **then**
        $\mathcal{E}_{bfst} = \mathcal{E}_{bfst} \cup \{(p_i, i)\}$
    end if
    $candidate\_list$.erase($candidate\_list$.begin())
    for $\langle j, d_j, p_j\rangle$ in $candidate\_list$ **do**
        if $(i,j) \in \mathcal{E}$ **then**
            if $\max(d_i, w_{\mathcal{V}}^{EAT}(j)) + w_{\mathcal{E}}(i,j) + w_{\mathcal{V}}^{ETT}(j) > d_j$ **then**
                $candidate\_list$.remove($\langle j, d_j, p_j\rangle$)
            end if
        end if
    end for // *remove unfeasible candidates*
    for $j = 0$ to N **do**
        if $j \notin \mathcal{V}_{bfst}$ **and** $j \notin candidate\_list$ **and** $\mathcal{P}_u^j \subseteq \mathcal{V}_{bfst}$ **and** $(i,j) \in \mathcal{E}$ **then**
            $candidate\_list$.push_back($\langle j, \max(d_i, w_{\mathcal{V}}^{EAT}(j)) + w_{\mathcal{E}}(i,j) + w_{\mathcal{V}}^{ETT}(j), i\rangle$)
        end if
    end for // *add new candidates*
    sort($candidate\_list$)
end while
return $\mathcal{T}_{bfst} = (\mathcal{V}_{bfst}, \mathcal{E}_{bfst}, w_{\mathcal{E}}, w_{\mathcal{V}}^{ETT})$

2.4.2 Dynamic Lane Assignment

The Breadth-First Spanning Tree with Dynamic Lane Assignment (BFST-DynaLane) method is used to solve problems with dynamic lane assignment. It aims to mitigate competing conflicts among vehicles, thereby further enhancing traffic efficiency. The fundamental idea involves reallocating departure lanes when two or more vehicles share the same departure leg, potentially reducing conflicts. BFST-DynaLane follows the structure of BFST but expands the candidate list to include an additional entry for the target scheduled departure lane, as defined in Definition 2.11. Multiple candidates are created for a single vertex, each corresponding to a specific departure lane. Specific maneuvers to address competing conflicts are also introduced.

Definition 2.11 (Candidate Data Structure For Dynamic Lane Assignment)**.** In the context of BFST-DynaLane, a candidate data structure is represented by a tuple $\langle i, d_i, p_i, L_i^{d\_sched} \rangle$, where i denotes the target vehicle index, p_i is the possible parent of vertex i, d_i presents the depth of i if connected via p_i, and $L_i^{d\_sched}$ denotes the scheduled departure lane for vehicle i.

Algorithm 2 presents the pseudocode for BFST-DynaLane. The set $\mathcal{L}_j^{d\_sched}$ is used to record all available lanes for the scheduled vertex that result in the same vertex depth. Within each iteration, the first candidate in the list is selected to be added to the spanning tree. Line 12 updates $\mathcal{L}_*^{d\_sched}$ for all scheduled vertices. First, departure lanes from all candidates of i sharing the same depth as the selected candidate are added to $\mathcal{L}_i^{d\_sched}$. Subsequently, conflicting lanes in $\mathcal{L}_j^{d\_sched}$ of other scheduled vertices j are removed due to the scheduling of vertex i. Since a vertex can only be scheduled once, all other candidates associated with vertex i are removed in line 14. Lines 15 - 21 remove candidates that may have crossing or converging conflicts with the scheduled vertex i, where $R_j^{L_j^{d\_sched}}$ represents the new route of



Loop	0	1		2		3	4	5	
		L_{11}	L_{12}	L_5	L_6	L_8	L_8	L_5	L_6
#0	$\langle \mathbf{0,0,-1} \rangle$	NA	NA	NA	NA	NA	NA	NA	NA
#1		$\langle \mathbf{1,6,0} \rangle$	$\langle \mathbf{1,6,0} \rangle$	$\langle 2,7,0 \rangle$	$\langle 2,7,0 \rangle$	$\langle 3,8,0 \rangle$	$\langle 4,11,0 \rangle$	NA	NA
#2				$\langle 2,13,1 \rangle$	$\langle 2,13,1 \rangle$	$\langle \mathbf{3,11,1} \rangle$	$\langle 4,11,0 \rangle$	NA	NA
#3				$\langle 2,18,3 \rangle$	$\langle 2,18,3 \rangle$		$\langle \mathbf{4,17,3} \rangle$	NA	NA
#4				$\langle \mathbf{2,18,3} \rangle$	$\langle \mathbf{2,18,3} \rangle$			$\langle 5,22,4 \rangle$	$\langle 5,22,4 \rangle$
#5								$\langle 5,24,2 \rangle$	$\langle \mathbf{5,22,4} \rangle$

Figure 2.4: Result spanning tree and schedule details of BFST-DynaLane method of the example intersection of 2.1. The scheduled departure lanes and time windows are highlighted in blue. Candidates present in the candidate list for each loop are displayed in a row and are updated in each loop. For simplicity, the scheduled departure lanes of candidates are shown in the header of the schedule table. Each candidate follows the form $\langle i, d_i, p_i \rangle$, where i denotes the target vertex index, p_i is the possible parent of vertex i, and d_i represents the depth of i if connected via p_i. The chosen candidate within a loop is denoted in bold.

j determined by applying $L_j^{d\_sched}$ as the departure lane. In the stage to add new candidates, lines 22 - 28 try each possible lane on the departure leg. Since the new candidate can be connected to the tree by any scheduled vertex, the function *candidate_list*.addCandidate(j, $L_j^{d\_sched}$) in line 25 finds the parent with the minimum depth for j when departing in lane $L_j^{d\_sched}$. Finally, the candidate list is sorted again to prepare for the next iteration.

Fig. 2.4 shows an example of the scheduled spanning tree of BFST-DynaLane method for the intersection in Fig. 2.1. The scheduled departure lanes of candidates are displayed in the header of the schedule table. In loop #4, both lanes, L_5 and L_6, are identified as possible departure lanes for vertex 2 because they have the same depth in the spanning tree. Furthermore, it is possible for different departure lanes to have distinct depths; for instance, in loop #5, the updated candidate of vehicle 5 using L_5 differs from the candidate using L_6. Then only the candidate with the smallest depth is selected, this will also disable the choice of L_6 for vertex 2. The evacuation time of the intersection is 22 seconds using BFST-DynaLane method.

Algorithm 2 Breadth-First Spanning Tree with Dynamic Lane Assignment Method.

Input Extended Conflict Directed Graph $\mathcal{G}$
Output: Breadth-First Spanning Tree with Dynamic Lane Assignment $\mathcal{T}_{Dyna}$

$candidate\_list = \{\langle 0, 0, -1, null\rangle\}$
$\mathcal{V}_{Dyna} = \emptyset$, $\mathcal{E}_{Dyna} = \emptyset$
for $j = 0$ to N **do**
    $\mathcal{L}_j^{d\_sched} = \emptyset$ // *possible departure lanes of* j
    $\mathcal{P}_u^j = \{i \in \mathcal{V} \mid \mathbf{C}_{div} \in CR(i,j)\}$ // *unidirectional parents*
end for
while $candidate\_list \neq \emptyset$ **do**
    $\langle i, d_i, p_i, L_i^{d\_sched}\rangle = candidate\_list$.front()
    $\mathcal{V}_{Dyna} = \mathcal{V}_{Dyna} \cup \{i\}$
    if $i \neq 0$ **then**
        $\mathcal{E}_{Dyna} = \mathcal{E}_{Dyna} \cup \{(p_i, i)\}$
        update $\mathcal{L}_j^{d\_sched}$ for all scheduled vertex $j \in \mathcal{V}_{Dyna}$
    end if
    $candidate\_list$.removeCandidatesAssociatedWith(i)
    for $\langle j, d_j, p_j, L_j^{d\_sched}\rangle$ in $candidate\_list$ **do**
        if $(i,j) \in \mathcal{E}$ and $R_j^{L_j^{d\_sched}}$ is crossing or converging conflict with any R_i **then**
            if $\max(d_i, w_{\mathcal{V}}^{EAT}(j)) + w_{\mathcal{E}}(i,j) + w_{\mathcal{V}}^{ETT}(j) > d_j$ **then**
                $candidate\_list$.removeCandidatesAssociated With(j)
            end if
        end if
    end for // *remove candidates that may be unfeasible*
    for $j = 0$ to N **do**
        if $j \notin \mathcal{V}_{Dyna}$ **and** $j \notin candidate\_list$ **and** $\mathcal{P}_u^j \subseteq \mathcal{V}_{Dyna}$ **then**
            for $L_j^{d\_sched}$ in vehicle j's departure leg **do**
                $candidate\_list$.addCandidate(j, $L_j^{d\_sched}$)
            end for
        end if
    end for // *add new candidates*
    sort($candidate\_list$)
end whilereturn $\mathcal{T}_{Dyna} = (\mathcal{V}_{Dyna}, \mathcal{E}_{Dyna}, w_{\mathcal{E}}, w_{\mathcal{V}}^{ETT})$

2.4.3 Assign Time Window and Scheduled Departure Lane

The scheduled time window and departure lane can be easily obtained from the spanning tree as the depth of a vertex corresponds to the latest departure time of the CAVs. The pseudocode for this process is presented in Algorithm 3.

Algorithm 3 Schedule From Spanning Tree.

Input: Spanning tree $\mathcal{T}$
Output: Scheduled time window and departure lane

for $i = 1$ to N **do**
  get i's parent p_i in $\mathcal{T}$
  calculate d_i and d_{p_i} based on Definition 2.8
  $window_i = [\max(d_{p_i}, w_{\mathcal{V}}^{EAT}(i)), d_i)$
  if using BFST method **then**
    $L_i^{d\_sched} = L_i^{d\_init}$
  else if using BFST-DynaLane method **then**
    $L_i^{d\_sched} = \mathcal{L}_i^{d\_sched}[0]$ from Algorithm 2
  end if
end for

2.4.4 Time Complexity Analysis

Both the BFST and BFST-DynaLane methods exhibit polynomial time complexity. Specifically, given that a single vertex is scheduled in each iteration, the outermost while loop terminates after N iterations, where N represents the total number of vehicles in the intersection control zone. Within each iteration, the most computationally intensive task for both methods involves adding new candidates. For BFST, adding new candidates incurs a time complexity of $O(N)$. For BFST-DynaLane, the time complexity increases to $O(N^2K)$, where K denotes the maximum number of lanes in a leg. Consequently, the overall time complexity is $O(N^2)$ for BFST and $O(N^3K)$ for BFST-DynaLane.

2.5 Experiments

Extended numerical experiments are conducted to evaluate the effectiveness of the proposed methods. In addition to the BFST and BFST-DynaLane methods, two baseline approaches, First-In-First-Out (FIFO) and improved Depth-First Spanning Tree (iDFST), are evaluated. The FIFO method employs a straightforward first-in-first-out scheduling policy, allocating conflict-free time windows for all vehicles to generate a safe intersection schedule. iDFST follows Chen *et al.*[22]'s work, the time window interval of each batch in iDFST is determined as the maximum estimated travel time among all vehicles within the scheduled batch. The code for these methods is available on GitHub [1].

Different traffic demand levels are tested, with the number of vehicles inside the control zone chosen from the set $\{10, 50, 100, 200\}$. The intersection's geospatial distribution follows the settings illustrated in Fig. 2.1. For each traffic demand, 10,000 random scenarios are generated, where the routes of each vehicle are randomly selected. The estimated travel time follows a uniform distribution $\mathbf{U}[5, 7]$ and the arrival time interval between vehicles follows a Poisson distribution with an average of 2 seconds.

The average evacuation time for each traffic demand level is shown in Fig. 2.5. The proposed BFST-DynaLane method achieves the best schedule for all scenarios. It achieves a significant reduction in evacuation time compared to the FIFO method, with savings of 36.31%, and the iDFST method, with savings of 22.19%. Even in cases where dynamic lane assignment is not feasible due to regulations, the BFST method still demonstrates superior performance compared to the state-of-the-art method iDFST, achieving a 15.48% reduction in evacuation time.

Dynamic lane assignment proves to be particularly effective in intersections where each leg

[1] https://github.com/LIDONGgittt/intersection_ECDG/tree/main

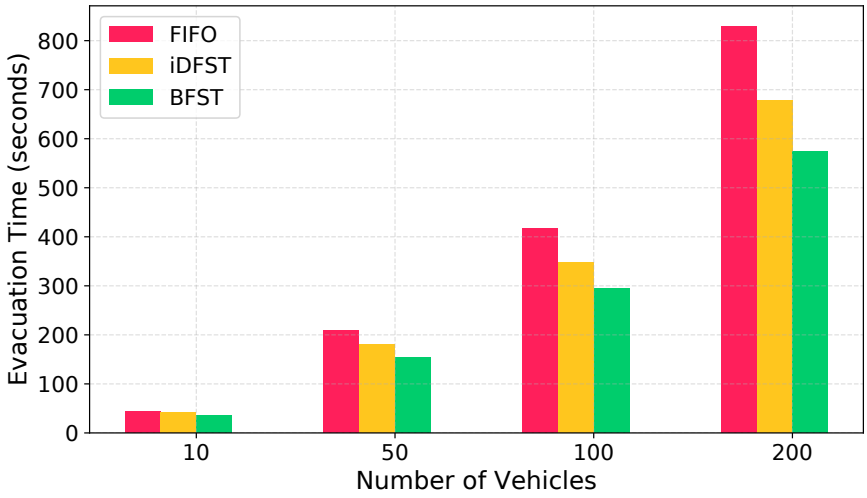


Figure 2.5: Average evacuation time for different traffic demands. The intersection's geospatial distribution remains consistent with the layout depicted in Fig. 2.1, in which two legs have one approach lane and one departure lane, while the other two legs have two approach lanes and two departure lanes.

offers multiple choices for departure lanes. To assess this, another geospatial characteristic is tested, featuring four legs, each with three approach lanes and three departure lanes. Similar to previous experiments, estimated travel time and arrival time interval distributions remain the same. The result is shown in Fig. 2.6. When more departure lanes are available, dynamic lane assignment demonstrates its full potential, achieving substantial improvements in evacuation time. Specifically, dynamic lane assignment saves as much as 44.75%, 27.74%, and 17.44% in evacuation time, when compared to FIFO, iDFST, and BFST, respectively. These findings underscore the effectiveness of dynamic lane assignment strategies, particularly in busy intersections that provide multiple departure lane choices on each leg. By dynamically allocating lanes based on real-time traffic conditions, this approach optimizes traffic flow and reduces congestion, ultimately enhancing overall intersection efficiency.

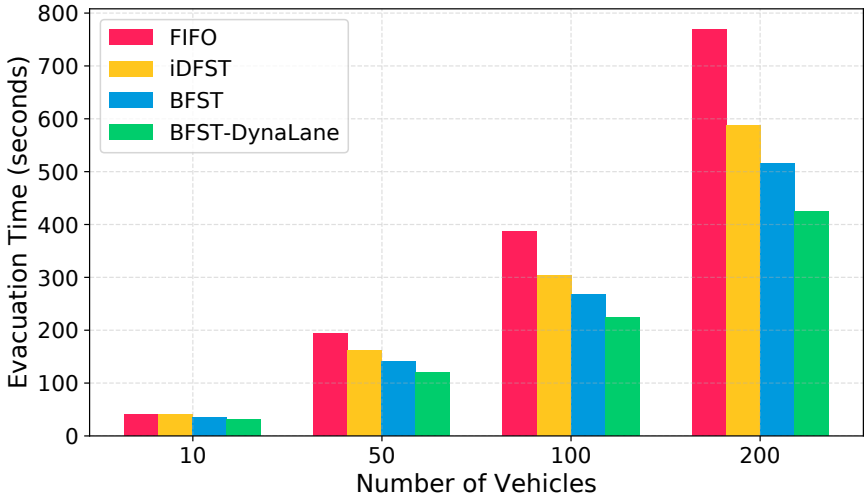


Figure 2.6: Average evacuation time for different traffic demands. All scenarios utilized a four-way intersection, where each leg has three approach lanes and three departure lanes respectively.

2.6 Conclusion

This chapter has proposed an Extended Conflict Directed Graph (ECDG) model to address the unsignalized intersection management problem. We introduced two novel breadth-first-search-based methods for efficient intersection scheduling, along with a pioneering dynamic lane assignment approach that significantly enhances traffic efficiency. Looking ahead, there are several promising avenues for future research. Firstly, exploring tie-breaking policies could refine intersection scheduling algorithms, especially when multiple candidates have a tie depth. Secondly, incorporating considerations of fairness into intersection management algorithms could lead to more equitable traffic distribution, benefiting all road users. Lastly, enhancing the safety operations of CAVs, particularly ensuring the safety of critical scenarios, such as emergency stops, remains a crucial aspect of intersection management systems and warrants continued attention and innovation. By addressing these areas, we can further advance the field of intersection management and contribute to safer, more efficient

transportation systems for communities.

Chapter 3

Time-Triggered Scheduling for Non-Preemptive Real-Time DAG Tasks Using 1-Opt Local Search

3.1 Abstract

Modern real-time systems frequently comprise numerous computational tasks with complex interdependencies. In these systems, information is transmitted along cause-effect chains connecting dependent tasks, making it critical to impose stringent end-to-end latency bounds for maintaining system reliability. This chapter introduces advanced non-preemptive scheduling methods aimed at reducing the worst-case end-to-end latency as well as the time disparity of a set of tasks represented by the directed acyclic graphs (DAGs). Reaching this objective is particularly challenging due to the discontinuous and non-convex properties of the objective functions, which hinder the effectiveness of conventional optimization approaches. Tailored optimization frameworks that strive for exact solutions often encounter scalability bottlenecks, while heuristic methods generally fall short in offering theoretical performance assurances. To overcome these obstacles, the concept of "1-opt" is adopted from optimization theory, where a solution is considered 1-opt if no improvement is possible by modifying a single decision variable. This principle guides the development of our algo-

rithm. We introduce a novel optimization approach that strikes a practical balance between provable performance and computational efficiency. The proposed algorithm delivers 1-opt solutions with guaranteed polynomial-time complexity. Empirical results from extensive large-scale experiments show that it consistently reduces latency-related metrics by 20% to 40% compared to existing approaches.

3.2 Introduction

Ensuring timeliness, short end-to-end latency, and small data communication time disparity is a paramount consideration across various domains, including control engineering, body electronics, and automotive systems [32]. For example, the RTSS2021 Industry Challenge [81] underscores the importance of bounding worst-case end-to-end latency and time disparity in *non-preemptive* autonomous driving systems. Non-preemptive systems are becoming more popular due to the wide adoption of Single-Instruction-Multi-Data (SIMD) computing architectures such as GPU. Since preemption with GPU usually has a much higher overhead than CPU devices, embedded GPU devices often only provide limited, if any, support for preemption [77].

Scheduling and optimizing systems with respect to Data Age, Reaction Time, and Time Disparity (DARTD)[1] pose significant challenges [1, 30, 32, 41, 55, 95] due to their non-convex and non-continuous characteristics. These attributes hinder the application of standard mathematical programming frameworks, such as integer linear programming and convex optimization. However, naively employing highly general optimization frameworks like meta-heuristics often lacks theoretical performance guarantees. Conversely, developing customized

[1]Given a cause-effect chain, data age measures the maximum duration for which a sensor event influences the computational system, while reaction time measures the maximum latency for the system to first react to a sensor event. Additionally, time disparity quantifies the maximum difference in the generation times of multiple source data from which one task reads input.

frameworks targeted at yielding optimal solutions [95] encounters scalability issues, which is particularly important in modern computation systems, where hundreds of computation tasks may exist [56, 100]. To tackle these challenges, we propose a computationally efficient optimization algorithm with some theoretical performance guarantees.

In this chapter, we leverage the *1-opt* concept, drawn from the optimization literature [54, 80] as a foundation in the development of our optimization algorithm. A solution vector $\boldsymbol{x} \in \mathbb{R}^N$ for an optimization problem is called 1-opt if changing any single component $\boldsymbol{x}_i \in \boldsymbol{x}$ does not result in an improvement beyond the current solution $\boldsymbol{x}$. We refer to algorithms that yield 1-opt solutions as 1-opt algorithms. In contrast to heuristic algorithms, 1-opt algorithms provide stronger theoretical performance guarantees. Moreover, they often demonstrate superior scalability when compared to algorithms aimed at finding optimal solutions.

Nevertheless, constructing 1-opt algorithms for optimizing non-convex and non-continuous metrics such as DARTD is very challenging. Naively employing brute-force algorithms can result in exponential complexity in worst-case scenarios. To address this, we propose a novel algorithm that employs a technique to partition the solution space into multiple convex subspaces, allowing for the efficient utilization of linear programming (LP) to minimize DARTD within each subspace. Subsequently, an iterative subroutine efficiently traverses among the subspaces, ensuring that the output is 1-opt. Furthermore, we prove that the solution of each LP is local optimal in non-preemptive single-core systems. In comparison with simple scheduling heuristics such as list scheduling [93], scheduling with LP can explore a much larger solution space, leading to enhanced performance. Moreover, the polynomial run-time complexity of solving LP enhances algorithm scalability compared to optimal algorithms that exhibit exponential run-time complexities in the worst case. Finally, to further improve the efficiency of LP, we propose an algorithm capable of efficiently performing non-preemptive schedulability analysis.

Contributions. The contributions are as follows:

1. We employ the 1-opt concept in the development of schedule optimization algorithms. To the best of our knowledge, this is the first work to utilize the 1-opt concept in real-time system scheduling problems, and it achieves superior performance compared to state-of-the-art methods.

2. We propose a novel optimization framework designed to minimize worst-case DARTD, which is proven to yield 1-opt solutions with only polynomial run-time complexity.

3. To the best of our knowledge, this is the first work that considers optimizing time disparity with time-triggered scheduling.

4. Large-scale experiments demonstrate that 1-opt methods achieve 20% to 40% latency reductions and enhanced scalability compared to state-of-the-art techniques.

This chapter is from the author's previous work, Sen Wang, Dong Li, Shao-Yu Huang, Xuanliang Deng, Ashrarul H Sifat, Jia-Bin Huang, Changhee Jung, Ryan Williams, and Haibo Zeng. "Time-triggered scheduling for nonpreemptive real-time dag tasks using 1-opt local search". In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* [101].

3.3 Related Work

As an important indicator of system safety, end-to-end latency has been thoroughly studied. Numerous analyses have delved into cause-effect chains or task sets structured with directed acyclic graphs (DAG) dependency [1, 32, 41, 42, 55, 70, 95]. These analytical approaches address diverse scenarios, including different scheduling algorithms (e.g., fixed-priority schedul-

ing, earliest deadline first scheduling) and communication protocols (e.g., implicit communication, logical execution time). Moreover, some studies explore temporal variations across various contexts [51, 64]. Beyond the analysis of end-to-end latency, a considerable body of work focuses on scheduling and the schedulability of DAG task sets [11, 34, 62, 71]. These comprehensive analyses build the foundation for the optimization works performed in this chapter.

General optimization techniques in real-time systems can be broadly categorized into two categories: heuristic algorithms with general applicability but lacking solution quality guarantees [29, 92, 100], and optimal algorithms built with sophisticated assumptions and problem modeling [95, 128, 129]. However, the latter may encounter scalability issues when facing large-scale optimization problems and the performance may also degrade seriously. Considering the challenge of finding the "perfect" algorithms (optimal and fast) for many real-world problems, algorithm designers often face a trade-off between solution quality and run-time complexity.

There are many works that optimize the end-to-end latency with different types of variables. Within the logical execution time (LET) protocol, many works consider optimizing the time to read/write data, where both optimal [69, 102] and heuristic [15, 68] algorithms have been proposed. Some other works consider implicit communication protocol, primarily concentrating on optimizing task and message schedules [39, 95, 113]. Besides, there are also works that improve different metrics related to end-to-end latency by performing priority assignments [78, 91].

This chapter differs from existing literature in proposing to use a new concept, 1-opt, to guide the algorithm design process. We also designed a novel optimization algorithm which is proved to find 1-opt solutions and demonstrated to achieve significantly better performance than the state-of-the-art methods.

3.4 System model and problem description

In this chapter, bold fonts are used to represent vectors or sets, while light characters denote scalars or individual elements. The double bars notation $\|\|\|$ denotes norm-2. During iterations, the k^{th} iteration is denoted by a superscript, such as $\boldsymbol{x}^{(k)}$.

3.4.1 System Model

We consider a multi-rate Directed Acyclic Graph (DAG) model $\boldsymbol{\mathcal{G}} = (\boldsymbol{\tau}, \boldsymbol{E})$, in which each task $\tau_i \in \boldsymbol{\tau}$ is represented as a node, and a directed edge $E_k \in \boldsymbol{E}$ from τ_i to τ_j denotes that τ_j reads input from τ_i. The total number of tasks in $\boldsymbol{\tau}$ is denoted as n. Each task releases jobs (i.e., *instances of the task*) periodically with a nominal period. A task τ_i is characterized by a tuple $\{T_i, C_i, D_i\}$, which denotes the period, worst-case execution time (WCET), and the relative deadline, respectively. We assume $D_i \leq T_i$. The k^{th} released job of τ_i is denoted as $J_{i,k}$ and it is released at the time $k \cdot T_i$. The DAG $\boldsymbol{\mathcal{G}}$ is not necessarily fully connected. Without loss of generality, we assume all the tasks are released simultaneously at time 0. However, if there is an offset when all tasks are initially released, our optimization algorithm can also be applied by modifying the schedulability analysis algorithms and optimization constraints accordingly.

The hyper-period (i.e., the least common multiple of periods of all tasks in $\boldsymbol{\mathcal{G}}$) is denoted as H. Within a hyper-period, each job $J_{i,k}$ starts execution at time $s_{i,k}$ *non-preemptively* and finishes at $f_{i,k} = s_{i,k} + C_i$. Such a non-preemptive policy eliminates preemption overhead, which could be large in GPU computation. The total number of jobs within a hyper-period is denoted as N. Potential generalizations into preemptive systems are discussed in Section 3.9.2.

In a DAG $\boldsymbol{\mathcal{G}}$, tasks with chained reading/writing dependency formulate a cause-effect chain $\mathcal{C} = \{\tau_{p_0} \rightarrow \tau_{p_1} \rightarrow ... \rightarrow \tau_{p_k}\}$, which represents a data communication path. The implicit communication protocol [43] is utilized in data communication where each job $J_{i,k}$ reads data at its start time $s_{i,k}$, and writes data at $f_{i,k} = s_{i,k} + C_i$ even if $J_{i,k}$ may finish earlier than its worst-case execution time. Multiple cause-effect chains may share tasks, and the set of cause-effect chains is denoted as $\boldsymbol{\mathcal{C}}$.

In scenarios where a single task reads data from the outputs of multiple tasks, we refer to the tasks providing data as the source tasks, and the task that reads these outputs as the sink task. The source tasks and the sink task collectively formulate a "merge" $\mathcal{M}$ (For example, see Example 1). The set containing all merges to be optimized is denoted as $\boldsymbol{\mathcal{M}}$.

The DAG task set is processed by a multi-processor system. We assume that each job has a known processor assignment before performing the schedule optimization, and we do not consider processor migration during execution. For presentation simplicity, we assume using a homogeneous multi-processor system. However, the heterogeneous computation can be handled easily by modifying the resource-bound constraint correspondingly after obtaining processor assignments. In experiments, the processor is assigned following the First-Come-First-Serve heuristic, same as Verucchi *et al.* [95] for a fair comparison. The proposed optimization framework does not optimize processor assignments.

3.4.2 General Schedule Optimization Problem Formulation

We consider the schedule optimization problem of time-triggered systems, focusing on reducing the worst-case end-to-end latency and/or time disparity. The optimization variables for our scheduling problem are called a schedule:

Definition 3.1 (Schedule)**.** Given a DAG $\boldsymbol{\mathcal{G}} = (\boldsymbol{\tau}, \boldsymbol{E})$, a schedule $\boldsymbol{s} \in \mathbb{R}^N$ is a vector of the

start time of all jobs of all tasks in $\boldsymbol{\tau}$ within a hyper-period H.

A general schedule optimization problem consists of an objective function and a set of schedulability constraints:

$$\underset{\boldsymbol{s}}{\text{Minimize}} \quad \mathcal{F}(\boldsymbol{s}) \tag{3.1}$$

Subject to :

$\forall i \in \{0, ..., n-1\}, \forall k \in \{0, ..., H/T_i - 1\},$

$$k \cdot T_i \leq s_{i,k} \leq k \cdot T_i + D_i - C_i \tag{3.1a}$$

$$\text{ResourceBound}(\boldsymbol{s}) = 0. \tag{3.1b}$$

Constraint (3.1a) guarantees every job starts and finishes within its schedulable range. The resource bound constraint (3.1b) specifies that no computation resources are overloaded (e.g., one CPU core executes more than one job simultaneously). The specific form of Eq. (3.1b) will be introduced later in Section 3.4.5. A schedule $\boldsymbol{s}$ is feasible (or equivalently, schedulable) if it satisfies both (3.1a) and (3.1b). Given a schedule $\boldsymbol{s}$, the finish time $f_{i,k}$ of each job $J_{i,k}$ in non-preemptive systems is implicitly decided: $f_{i,k} = s_{i,k} + C_i$.

3.4.3 Example Problem: End-to-end Latency Optimization

Each cause-effect chain $\mathcal{C}$ could trigger multiple job chains within a hyper-period. The worst-case data age (reaction time) of a cause-effect chain $\mathcal{C}$ is the length of its longest immediate backward (forward) job chain [30, 41]. These definitions are briefly reviewed below:

Definition 3.2 (Job chain [30, 41])**.** Given a cause-effect chain $\mathcal{C} = \{\tau_{p_0} \rightarrow \tau_{p_1} \rightarrow ... \rightarrow \tau_{p_k}\}$, a job chain $\mathcal{C}^J$ is a sequence of jobs $\{J_{p_0,q_0} \rightarrow J_{p_1,q_1} \rightarrow ... \rightarrow J_{p_k,q_k}\}$, where J_{p_i,q_i} is the q_i^{th} job of τ_{p_i}, and the data produced by J_{p_i,q_i} is read by $J_{p_{i+1},q_{i+1}}$.

Definition 3.3 (Length of a job chain)**.** The length of a job chain $\mathcal{C}^J = \{J_{p_0,q_0} \rightarrow J_{p_1,q_1} \rightarrow ... \rightarrow J_{p_k,q_k}\}$ is the time interval from the start time of J_{p_0,q_0} till the finish time of J_{p_k,q_k}. It is denoted as $L(\mathcal{C}^J) = f_{p_k,q_k} - s_{p_0,q_0}$.

Definition 3.4 (Immediate backward (forward) job chain [30, 41])**.** A job chain $\mathcal{C}^J = \{J_{p_0,q_0} \rightarrow J_{p_1,q_1} \rightarrow ... \rightarrow J_{p_k,q_k}\}$ is the immediate backward (forward) chain under schedule $\boldsymbol{s}$ if Eq. (3.2) (Eq. (3.3)) is satisfied.

$$\forall i \in \{1, ..., k\}, \quad f_{p_{i-1},q_{i-1}} \leq s_{p_i,q_i} < f_{p_{i-1},(q_{i-1}+1)} \tag{3.2}$$

$$\forall i \in \{0, ..., k-1\}, \quad s_{p_{i+1},(q_{i+1}-1)} < f_{p_i,q_i} \leq s_{p_{i+1},q_{i+1}} \tag{3.3}$$

Example 1. Fig. 3.1 shows a simple DAG with three tasks: $\boldsymbol{\tau} = \{\tau_0, \tau_1, \tau_2\}$ and two edges: $\boldsymbol{E} = \{\tau_0 \rightarrow \tau_2, \tau_1 \rightarrow \tau_2\}$. The WCET, period, and relative deadline of each task is: $\{C_0 = 1, T_0 = 10, D_0 = 10\}$, $\{C_1 = 2, T_1 = 20, D_1 = 20\}$, $\{C_2 = 3, T_2 = 20, D_2 = 20\}$. The task set is executed on 2 identical processors unless otherwise stated. The hyper-period is 20. The schedule variable contains the start time of $N = 4$ jobs: $\boldsymbol{s} = [s_{0,0}, s_{0,1}, s_{1,0}, s_{2,0}]$.

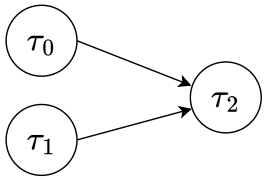


Figure 3.1: Example DAG.

Suppose we have a schedule $\boldsymbol{s} = [0, 10, 1, 3]$. For the cause-effect chain $\mathcal{C} = \{\tau_0 \rightarrow \tau_2\}$, the job chain $\mathcal{C}_0^J = \{J_{0,0} \rightarrow J_{2,0}\}$ is both an immediate backward job chain and immediate forward job chain with length $L(\mathcal{C}_0^J) = 6$. $\mathcal{C}_1^J = \{J_{0,1} \rightarrow J_{2,1}\}$ is another immediate forward job chain with length $L(\mathcal{C}_1^J) = 16$. Thus, $\max \boldsymbol{DA}_{\mathcal{C}}(\boldsymbol{s}) = 6$, $\max \boldsymbol{RT}_{\mathcal{C}}(\boldsymbol{s}) = 16$. The longest job chains for this scenario are shown in Fig. 3.2.

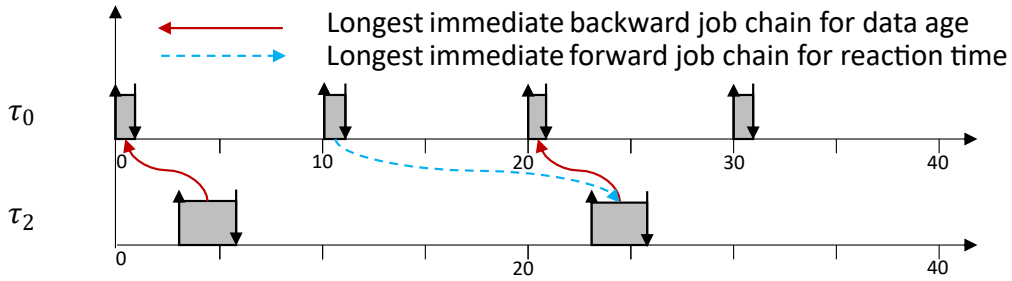


Figure 3.2: Longest immediate forward and backward job chains for cause-effect chain $\mathcal{C} = \{\tau_0 \rightarrow \tau_2\}$.

Given a schedule $\boldsymbol{s}$, we use $\boldsymbol{DA}_{\mathcal{C}}(\boldsymbol{s})$ ($\boldsymbol{RT}_{\mathcal{C}}(\boldsymbol{s})$) to denote the vector of data age (reaction time) for all job chains of a cause-effect chain $\mathcal{C}$ within a hyper-period.

To summarize, when optimizing the worst-case data age or reaction time, the objective function in (3.1) becomes:

$$\mathcal{F}(\boldsymbol{s}) = \sum_{\mathcal{C} \in \boldsymbol{\mathcal{C}}} \max \boldsymbol{DA}_{\mathcal{C}}(\boldsymbol{s}) \tag{3.4}$$

or

$$\mathcal{F}(\boldsymbol{s}) = \sum_{\mathcal{C} \in \boldsymbol{\mathcal{C}}} \max \boldsymbol{RT}_{\mathcal{C}}(\boldsymbol{s}) \tag{3.5}$$

3.4.4 Example Problem: Time Disparity Optimization

Similar to a cause-effect chain, a merge $\mathcal{M}$ may have multiple job-level merges:

Definition 3.5 (Job merge)**.** A job merge $\mathcal{M}^J$ contains a sink job $J_{j,l}$ and a set of source jobs $\boldsymbol{J}_{j,l}^{Src}$, from which $J_{j,l}$ directly reads data:

$$\forall J_{i,k} \in \boldsymbol{J}_{j,l}^{Src}, \quad f_{i,k} \leq s_{j,l} < f_{i,k+1} \tag{3.6}$$

Definition 3.6 (Time disparity [81, 102])**.** The time disparity of a job merge $\mathcal{M}^J$, denoted as $TD(\mathcal{M}^J)$, is defined as the difference between the earliest and latest finish times of all source jobs in $\mathcal{M}^J$.

$$TD(\mathcal{M}^J) = \max_{J \in \boldsymbol{J}_{j,l}^{Src}} f_J - \min_{J \in \boldsymbol{J}_{j,l}^{Src}} f_J \tag{3.7}$$

where f_J represents the finish time of a job J.

Given a schedule $\boldsymbol{s}$, we use $\boldsymbol{TD}_{\mathcal{M}}(\boldsymbol{s})$ to denote the vector of time disparities for all job merges of $\mathcal{M}$ within a hyper-period. When optimizing the worst-case time disparity metric, the objective function in (3.1) is formulated as follows:

$$\mathcal{F}(\boldsymbol{s}) = \sum_{\mathcal{M} \in \boldsymbol{\mathcal{M}}} \max \boldsymbol{TD}_{\mathcal{M}}(\boldsymbol{s}) \tag{3.8}$$

Other forms of the objective functions are discussed in Section 3.9.1.

Example 2. In Example 1, there is only one merge $\mathcal{M}$ in the DAG with τ_2 as the sink task. The corresponding job merge has $J_{2,0}$ as the sink job and $\{J_{0,0}, J_{1,0}\}$ as the source jobs. The maximum time disparity is $\max \boldsymbol{TD}_{\mathcal{M}}(\boldsymbol{s}) = 3 - 1 = 2$.

Theorem 3.7. *The objective functions* (3.4), (3.5), *and* (3.8) *are all non-convex.*

Proof. We prove it by providing counter-examples. Remember that a function $f : \mathbb{R}^n \to \mathbb{R}$ is convex if for all $\boldsymbol{s}^{(1)}$ and $\boldsymbol{s}^{(2)}$ in its domain and $\forall t \in [0, 1]$, we have $f(t\boldsymbol{s}^{(1)} + (1 - t)\boldsymbol{s}^{(2)}) \leq tf(\boldsymbol{s}^{(1)}) + (1 - t)f(\boldsymbol{s}^{(2)})$. We now give a counterexample for reaction time, and the counterexamples for data age and time disparity are similar. In Example 1, consider $\boldsymbol{s}^{(1)} = [0, 10, 1, 3]$ with a reaction time 16, $\boldsymbol{s}^{(2)} = [0, 10, 1, 11]$ whose reaction time is 14. If we define $t = 0.5$, then $\boldsymbol{s}^{(t)} = t\boldsymbol{s}^{(1)} + (1 - t)\boldsymbol{s}^{(2)} = [0, 10, 1, 7]$, but the reaction time of $\boldsymbol{s}^{(t)}$ is 20, which violates the property required by convex functions. □

3.4.5 Resource Bound Constraint: Interval Overlapping Test

In a non-preemptive system, the Interval Overlapping Test (IO Test) analyzes whether processors are overloaded (one processor executes multiple jobs in parallel) for a given schedule $\boldsymbol{s}$. In this case, each job $J_{i,k}$ can be modeled as an interval $[s_{i,k}, f_{i,k}]$ that starts execution at $s_{i,k}$ and finishes at $f_{i,k} = s_{i,k} + C_i$. Inspired by the demand bound function [10], we propose an efficient non-preemptive schedulability analysis for optimization. Intuitively speaking, there are no overloaded processors if any two job intervals mapped to the same processor do not overlap.

Theorem 3.8 (IO test)**.** *In non-preemptive systems, there are no overloaded processors if the following inequality holds for any two jobs $J_{i,k}$ and $J_{j,l}$ assigned to the same processor:*

$$\textit{if } f_{j,l} \geq s_{i,k}, \textit{ then } f_{j,l} - s_{i,k} \geq C_i + C_j \tag{3.9}$$

Proof. Prove by contradiction. If there are overloaded processors, by definition, there must be two job execution intervals overlapping with each other. Let's denote the job with a larger finish time as $J_{j,l}$, the other job as $J_{i,k}$, then we have:

$$f_{j,l} - s_{i,k} < C_i + C_j \tag{3.10}$$

This contradicts the IO test assumption above. □

Theorem 3.9. *Given a set of job intervals $\boldsymbol{I} = \{[s_{i,k}, f_{j,l}]\}$ sorted based on its start time $s_{i,k}$ in increasing order, no intervals overlap with each other if any two adjacent job intervals do not overlap with each other.*

Proof. Skipped. It can be proved easily by contradiction. □

Since a schedule will repeat in every hyper-period, the IO test only needs to consider all jobs within a hyper-period. Within partitioned scheduling, each processor has to be tested separately. The time complexity of the IO test is $O(Nlog(N))$.

Example 3. Let us continue with the task set in Example 1. Suppose we only have one processor and have a schedule $\boldsymbol{s} = [0, 10, 1, 3]$. If without sorting, the IO test requires verifying whether the following six pairs of intervals overlap:

$$\{[s_{0,0}, f_{0,0}], [s_{0,1}, f_{0,1}]\} \qquad \{[s_{0,0}, f_{0,0}], [s_{1,0}, f_{1,0}]\}$$
$$\{[s_{0,0}, f_{0,0}], [s_{2,0}, f_{2,0}]\} \qquad \{[s_{0,1}, f_{0,1}], [s_{1,0}, f_{1,0}]\}$$
$$\{[s_{0,1}, f_{0,1}], [s_{2,0}, f_{2,0}]\} \qquad \{[s_{1,0}, f_{1,0}], [s_{2,0}, f_{2,0}]\}$$

With sorting, only the following 3 pairs require verification:

$$\{[s_{0,0}, f_{0,0}], [s_{1,0}, f_{1,0}]\} \qquad \{[s_{1,0}, f_{1,0}], [s_{2,0}, f_{2,0}]\}$$
$$\{[s_{2,0}, f_{2,0}], [s_{0,1}, f_{0,1}]\}$$

If there is no overlap, then the IO test states that the processor is not overloaded.

Now, we can give the complete form of the resource bound constraint (3.1b) in non-preemptive systems:

$$\text{ResourceBound}(\boldsymbol{s}) = \begin{cases} 0, & \text{if } \boldsymbol{s} \text{ passes IO test} \\ 1, & \text{otherwise} \end{cases} \tag{3.11}$$

3.4.6 Model Assumptions

Assumption 3. The start time of each job could take continuous value.

Although the computer time is integer multiples of CPU cycles, the very high CPU run-time frequency (MHz or GHz) means that rounding a float-point number into its adjacent integers only incurs a small precision loss in timing metrics, if the jobs' relative reading/writing time order remains the same.

Assumption 4. A feasible schedule (a solution that satisfies constraints (3.1a) and (3.1b)) is available to start the iterative algorithms introduced next.

Normally, Assumption 4 can be easily satisfied with simple list schedulers [95]. This chapter focuses on optimizing the timing metrics rather than finding a schedulable schedule, although such an extension is possible (see Section 3.9.3).

3.4.7 Challenges

Solving the optimization problem (3.1) for DARTD is difficult because the objective function follows a *nonlinear, non-monotonic, non-convex, and non-continuous* relationship with the variables (see Theorem 3.7 and its proof). Therefore, most popular optimization frameworks cannot be directly utilized except integer linear programming (ILP). However, ILP requires introducing many extra binary variables and could suffer from bad algorithm scalability.

3.5 Job order and scheduling

The proposed optimization framework that solves the problem (3.1) is built upon the concept of the job order, which specifies the jobs' reading/writing relationships and simplifies the problem into a set of linear programming problems.

3.5.1 Job Order

Definition 3.10 (Job scheduling time)**.** The job scheduling time of a job $J_{i,k}$ is denoted as $\mathcal{T}_{i,k}$, which could be either the start time (denoted as $\mathcal{T}^s_{i,k}$, called *scheduling start time*) or the finish time (denoted as $\mathcal{T}^f_{i,k}$, called *scheduling finish time*) of $J_{i,k}$.

Since we adopt the implicit communication protocol and non-preemptive scheduling, a job $J_{i,k}$'s reading time is its start time, and its writing time is its finish time.

Example 4. In Example 1, consider a schedule $\boldsymbol{s} = [0, 10, 1, 3]$. The job $J_{0,0}$ has two scheduling times: scheduling start time $\mathcal{T}^s_{0,0} = 0$, and scheduling finish time $\mathcal{T}^f_{0,0} = 1$.

Definition 3.11 (Job order)**.** Given a set of jobs $\boldsymbol{J}$, a job order $\mathcal{O}$ of $\boldsymbol{J}$ is an ordered list containing all job scheduling times (both start and finish) of all the jobs in $\boldsymbol{J}$. The job scheduling times are ordered in non-decreasing order.

For notation convenience, we use $\mathcal{O}(i)$ to denote the i^{th} job scheduling time in the job order $\mathcal{O}$. For any two job scheduling times $\mathcal{T}_{i,k}, \mathcal{T}_{j,l} \in \mathcal{O}$, if $\mathcal{T}_{i,k}$ has a smaller index than $\mathcal{T}_{j,l}$ in $\mathcal{O}$, denoted as $\mathcal{T}_{i,k} \prec \mathcal{T}_{j,l}$, then that means $\mathcal{T}_{i,k}$ happens earlier than or at the same time as $\mathcal{T}_{j,l}$.

Example 5. Consider the task set in Example 1. There are four jobs within a hyper-period. For a schedule $\boldsymbol{s} = [s_{0,0}, s_{0,1}, s_{1,0}, s_{2,0}] = [0, 10, 1, 3]$, its job order is $\mathcal{O} = \{\mathcal{T}^s_{0,0}, \mathcal{T}^f_{0,0}, \mathcal{T}^s_{1,0}, \mathcal{T}^f_{1,0}, \mathcal{T}^s_{2,0}, \mathcal{T}^f_{2,0}, \mathcal{T}^s_{0,1}, \mathcal{T}^f_{0,1}\}$. We also give two examples for indexing: $\mathcal{O}(0) = \mathcal{T}^s_{0,0}$, $\mathcal{O}(3) = \mathcal{T}^f_{1,0}$.

A job order $\mathcal{O}$ implies a set of linear constraints on the schedule $\boldsymbol{s}$ of the optimization problem (3.1):

$$\forall\ i < j,\ \ Time(\mathcal{O}(i)) \leq Time(\mathcal{O}(j)) \tag{3.12}$$

where $Time(\mathcal{T}_{i,k})$ denotes the time that $\mathcal{T}_{i,k}$ happens. If $\mathcal{T}_{i,k}$ is a scheduling start time, $Time(\mathcal{T}_{i,k}) = s_{i,k}$, otherwise, $Time(\mathcal{T}_{i,k}) = s_{i,k} + C_i$.

3.5.2 Scheduling with Job Order

Finding a schedule that satisfies a given job order $\mathcal{O}$ is equivalent to solving the problem (3.1) with extra linear constraints given by Equation (3.12). Here we provide the job order scheduling problem for $\mathcal{O}$:

$$\underset{\boldsymbol{s}}{\text{Minimize}}\ \mathcal{F}(\boldsymbol{s}) \tag{3.13}$$

Subject to :

$\forall i \in \{0, ..., n-1\}, \forall k \in \{0, ..., H/T_i - 1\},$

$$k \cdot T_i \leq s_{i,k} \leq k \cdot T_i + D_i - C_i \tag{3.13a}$$

$$\text{ResourceBound}(\boldsymbol{s}) = 0 \tag{3.13b}$$

$$\forall i \in \{0, ..., 2N-2\},\ \ \mathit{Time}(\mathcal{O}(i)) \leq \mathit{Time}(\mathcal{O}(i+1)). \tag{3.13c}$$

where the objective function $\mathcal{F}(\boldsymbol{s})$ could be, for example, data age (3.4), reaction time (3.5), or time disparity (3.8).

Theorem 3.12. *The constraints from a job order $\mathcal{O}$ simplify the problem (3.13) into a convex problem, specifically, a linear programming problem, when the optimization objective is DARTD.*

Proof. Given a job order $\mathcal{O}$, the relative start/finish relationship of any two jobs is known, therefore all the job chains and job merges are decided. Then $\boldsymbol{DA}(\boldsymbol{s})$ and $\boldsymbol{RT}(\boldsymbol{s})$ become linear functions (lengths of all job chains in Definition 3.3). The $\boldsymbol{TD}(\boldsymbol{s})$ can also be similarly transformed into linear functions following [102]. Constraints (3.13a) and (3.13c) are evidently linear functions. As for the computational resource bounds (3.13b) from the IO test (3.9), since the given job order $\mathcal{O}$ already specifies the relative order of all the job scheduling times, the constraint (3.9) becomes linear inequalities. Therefore, problem (3.13) is a linear

programming problem. □

Next, we use $\pi^*(\mathcal{O})$ to denote the optimal schedule for the problem (3.13). Note that the $\pi^*(\mathcal{O})$ depends on the specific forms of objective functions and constraints.

Definition 3.13 (Optimal job order schedule)**.** The optimal job order schedule, $\boldsymbol{s}^* = \pi^*(\mathcal{O}) = \text{argmin}_{\boldsymbol{s}}\, \mathcal{F}(\boldsymbol{s})$, is the optimal solution of the optimization problem (3.13).

Example 6. In Example (1), consider a job order: $\mathcal{O} = \{\mathcal{T}^s_{0,0}, \mathcal{T}^f_{0,0}, \mathcal{T}^s_{1,0}, \mathcal{T}^f_{1,0}, \mathcal{T}^s_{2,0}, \mathcal{T}^f_{2,0}, \mathcal{T}^s_{0,1}, \mathcal{T}^f_{0,1}\}$, where we assume $J_{0,0}$ and $J_{1,0}$ are assigned to one processor $\mathcal{P}_0$, while $J_{2,0}$ and $J_{0,1}$ are assigned to another processor $\mathcal{P}_1$. Next, consider optimizing the reaction time of a cause-effect chain $\mathcal{C} = \{\tau_0 \to \tau_2\}$. The problem (3.13) can be transformed into a linear programming problem as follows:

$$\underset{\boldsymbol{s}}{\text{Minimize}}\ \max\{f_{2,0} - s_{0,0}, f_{2,1} - s_{0,1}\} \tag{3.14}$$

Subject to :

$$f_{0,0} = s_{0,0} + C_0,\ f_{0,1} = s_{0,1} + C_0 \tag{3.14a}$$

$$f_{1,0} = s_{1,0} + C_1,\ f_{2,0} = s_{2,0} + C_2 \tag{3.14b}$$

$$f_{2,1} = s_{2,0} + H + C_2 \tag{3.14c}$$

$$0 \le s_{0,0} \le D_0 - C_0,\ T_0 \le s_{0,1} \le T_0 + D_0 - C_0 \tag{3.14d}$$

$$0 \le s_{1,0} \le D_1 - C_1,\ 0 \le s_{2,0} \le D_2 - C_2 \tag{3.14e}$$

$$f_{1,0} - s_{0,0} \ge C_0 + C_1,\ f_{0,1} - s_{2,0} \ge C_0 + C_2 \tag{3.14f}$$

$$s_{0,0} \le s_{0,0} + C_0 \le s_{1,0} \le s_{1,0} + C_1 \le s_{2,0} \tag{3.14g}$$

$$s_{2,0} \le s_{2,0} + C_2 \le s_{0,1} \le s_{0,1} + C_0. \tag{3.14h}$$

The objective function (3.14) considers the length of two job chains initiated by $J_{0,0}$ and

$J_{0,1}$ within a hyper-period. The constraints (3.14a), (3.14b) and (3.14c) are due to the non-preemptive scheduling. Constraints (3.14d) and (3.14e) are schedulability constraints. Inequalities (3.14f) are the resource bound constraint (3.13b). There are only two IO-test constraints because jobs assigned to different processors can overlap. Constraints (3.14g) and (3.14h) posed by the given job order.

Definition 3.14 (Schedulable job order)**.** A job order $\mathcal{O}$ is schedulable if there exists a schedulable schedule $\boldsymbol{s}$ that also satisfies the job order constraints (3.13c).

3.6 Two-stage optimization scheduling

Although finding the optimal schedule given a job order is simple and efficient, enumerating all the possible job orders naively requires high computation costs. Therefore, we propose an iterative algorithm, Two-stage Optimization Scheduling (TOM), to search for better job orders. TOM is proven to find 1-opt solutions.

3.6.1 Optimization Concepts Review

Definition 3.15 (Global optimality)**.** A solution $\boldsymbol{s}^*$ for the problem (3.1) is global optimal if there is no other feasible solutions $\boldsymbol{s}$ such that $\mathcal{F}(\boldsymbol{s}) < \mathcal{F}(\boldsymbol{s}^*)$.

Definition 3.16 (Local optimality)**.** A solution $\boldsymbol{s}^*$ for the problem (3.1) is local optimal if there exists a small number $\delta > 0$, such that there is no other feasible solutions $\boldsymbol{s} \in \boldsymbol{\mathcal{B}}(\boldsymbol{s}^*)$ where $\mathcal{F}(\boldsymbol{s}) < \mathcal{F}(\boldsymbol{s}^*)$, $\boldsymbol{\mathcal{B}}(\boldsymbol{s}^*) = \{\boldsymbol{s} \mid \|\boldsymbol{s} - \boldsymbol{s}^*\| \leq \delta\}$.

Definition 3.17 (1-opt, [54, 80])**.** A solution $\boldsymbol{s}^{1*}$ for the problem (3.1) is *1-opt* if "the objective value at $\boldsymbol{s}^{1*}$ does not improve by changing a single coordinate", i.e., $\mathcal{F}(\boldsymbol{s}^{1*}) \leq \mathcal{F}(\boldsymbol{s}^{1*} + \mathbf{e}_i c)$ for arbitrary unit vector $\mathbf{e}_i = \{0, ..., 1, ..., 0\}$ and $c \neq 0$.

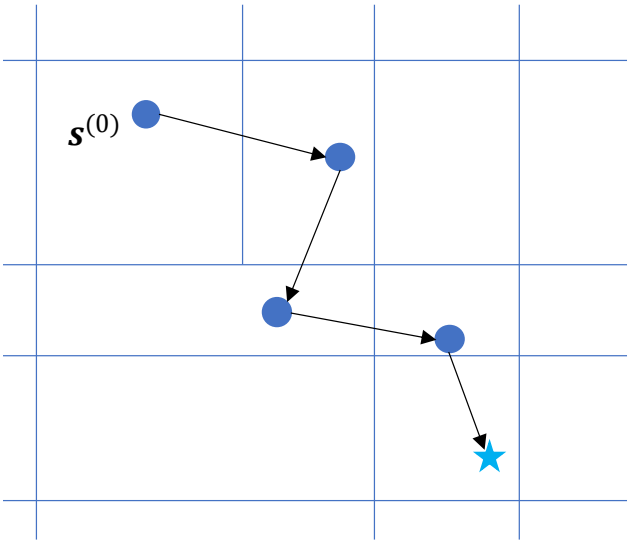


Figure 3.3: TOM intuition. The solution space is divided into multiple "sub-spaces", and the optimal solution within each sub-space can be found efficiently by solving a linear programming (LP) problem. This process is visualized above: Each job order defines a convex sub-space (because all the constraints are linear after specifying a job order) and is informally visualized as a grid in the figure above. The optimal solution within each grid is denoted as a solid circle. The original optimization problem, which needs to explore the whole solution space, is simplified into evaluating only the optimal solutions within each sub-space.

Although a global optimal solution is also local optimal and 1-opt, local optimal and 1-opt solutions are not inclusive of each other. In many real-time system problems, achieving global optimal or even local optimal solutions within reasonable time limits is difficult. In these cases, 1-opt provides a better trade-off between optimality and run-time complexity.

3.6.2 Two-stage Optimization Method (TOM)

Due to the non-convex and non-continuous nature of problem (3.1), straightforward optimization algorithms necessitate an infinite number of objective function evaluations to verify whether a solution is 1-opt. However, the concept of job order significantly simplifies the problem (3.1) and allows us to verify whether a solution is 1-opt with only polynomial time complexity. Therefore, we propose a two-stage optimization method (TOM). Fig. 3.4 shows an overview of TOM. Starting from an initial feasible schedule, the first stage searches for better job orders based on an iterative algorithm, while the second stage finds the optimal schedule by solving problem (3.13) for each job order to evaluate.

3.6.3 Theorems on 1-opt Conditions

Definition 3.18 (Adjacent schedule permutation)**.** The adjacent schedule permutation $\boldsymbol{\mathcal{B}}(\boldsymbol{s})$ of a schedule $\boldsymbol{s}$ is a set of schedules, where each schedule $\boldsymbol{\mathcal{B}}(\boldsymbol{s})_l$ differs from $\boldsymbol{s}$ by only one job's start time.

Definition 3.19 (Adjacent job order permutation)**.** Adjacent job order permutation $\boldsymbol{\mathcal{B}}(\mathcal{O})$ of a job order $\mathcal{O}$ is a finite set of distinct job orders. For each job order $\boldsymbol{\mathcal{B}}(\mathcal{O})_l$, there is one and only one job $J_{i,k}$ that the position of its scheduling start time $\mathcal{T}^s_{i,k}$, or its scheduling finish time $\mathcal{T}^f_{i,k}$, or both, are different from those in $\mathcal{O}$. The relative order of all the other jobs' scheduling time in $\mathcal{O}$ and $\boldsymbol{\mathcal{B}}(\mathcal{O})_l$ remain the same.

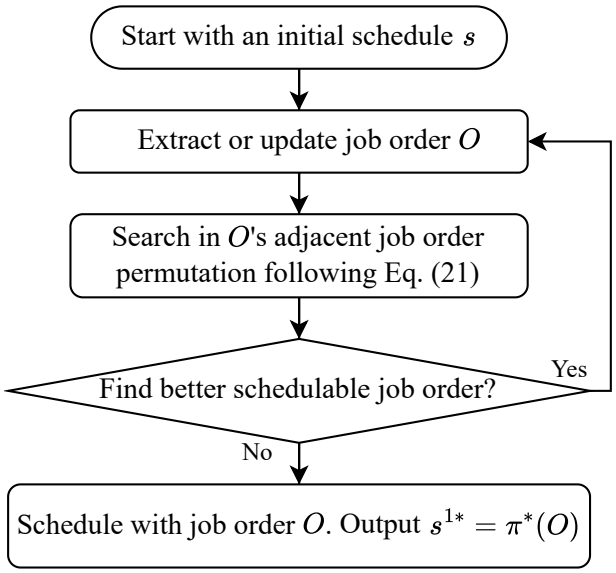


Figure 3.4: Main optimization framework. We begin with an initial feasible solution $\boldsymbol{s}$ and its job order $\mathcal{O}$. Then in each iteration, we search for a better job order in $\mathcal{O}$'s adjacent job order permutation $\boldsymbol{\mathcal{B}}(\mathcal{O})$ and update the best job order found yet. Eventually, the iteration will terminate at a 1-opt solution.

Example 7. Following the Example 1, let's consider a job order $\mathcal{O} = \{\mathcal{T}^s_{0,0}, \mathcal{T}^f_{0,0}, \mathcal{T}^s_{1,0}, \mathcal{T}^f_{1,0}, \mathcal{T}^s_{2,0}, \mathcal{T}^f_{2,0}, \mathcal{T}^s_{0,1}, \mathcal{T}^f_{0,1}\}$. As an example, $\boldsymbol{\mathcal{B}}(\mathcal{O})$ could include an job order such as $\{\mathcal{T}^s_{0,0}, \mathcal{T}^f_{0,0}, \mathcal{T}^s_{2,0}, \mathcal{T}^f_{2,0}, \mathcal{T}^s_{1,0}, \mathcal{T}^f_{1,0}, \mathcal{T}^s_{0,1}, \mathcal{T}^f_{0,1}\}$ by moving $J_{1,0}$ to the end of $J_{2,0}$. An alternative adjacent job order could be $\{\mathcal{T}^s_{0,0}, \mathcal{T}^s_{1,0}, \mathcal{T}^f_{0,0}, \mathcal{T}^f_{1,0}, \mathcal{T}^s_{2,0}, \mathcal{T}^f_{2,0}, \mathcal{T}^s_{0,1}, \mathcal{T}^f_{0,1}\}$ where $\mathcal{T}^f_{0,0}$ is moved to the back of $\mathcal{T}^s_{1,0}$, which means $J_{1,0}$ will start execution before $J_{0,0}$ finishes. It is schedulable if there is more than 1 processor.

Theorem 3.20. *Consider a schedule $\boldsymbol{s}^{1*}$ and its job order $\mathcal{O}^{1*}$. $\boldsymbol{s}^{1*}$ is a 1-opt solution for the optimization problem (3.1) if it satisfies the following conditions:*

$$\mathcal{O}^{1*} = \underset{\mathcal{O} \in \boldsymbol{\mathcal{B}}(\mathcal{O}^{1*}) \cap \Omega}{\operatorname{argmin}} \mathcal{F}(\pi^*(\mathcal{O})) \tag{3.15}$$

$$\boldsymbol{s}^{1*} = \pi^*(\mathcal{O}^{1*}) \tag{3.16}$$

where $\pi^*(\mathcal{O})$ denotes the optimal schedule obtained by solving the problem (3.13) for $\mathcal{O}$, Ω

denotes the set of schedulable job orders following Definition 3.14.

Proof. Consider an arbitrary solution $\hat{\boldsymbol{s}}$ which differs from $\boldsymbol{s}^{1*}$ by only one job's start time, and denote the job order of $\hat{\boldsymbol{s}}$ as $\hat{\mathcal{O}}$. In the case, we can introduce a function $\pi(\cdot)$ which obtains the schedule $\hat{\boldsymbol{s}} = \pi(\hat{\mathcal{O}})$. $\pi(\cdot)$ is possibly different from $\pi^*(\cdot)$ in Definition 3.13. Following Definition 3.19, we know $\hat{\mathcal{O}} \in \boldsymbol{\mathcal{B}}(\mathcal{O}^{1*})$, and therefore

$$\mathcal{F}(\hat{\boldsymbol{s}}) = \mathcal{F}(\pi(\hat{\mathcal{O}})) \geq \mathcal{F}(\pi^*(\mathcal{O}^{1*})) = \mathcal{F}(\boldsymbol{s}^{1*}) \tag{3.17}$$

Therefore, $\boldsymbol{s}^{1*}$ is 1-opt. □

Example 8. Let us continue with Example 1 and consider the reaction time optimization problem of a chain $\mathcal{C} = \{\tau_0 \rightarrow \tau_2\}$. A 1-opt schedule could be $\boldsymbol{s}^{1*} = [s_{0,0}, s_{0,1}, s_{1,0}, s_{2,0}] = [9, 10, 18, 11]$. This solution is 1-opt because there is no better feasible solution if only changing one job's start time while leaving the other 3 jobs' start times unchanged.

Lemma 3.21. *If there are six variables which satisfy* $a_1 + c_1 \leq b_1$, $b_2 + c_2 \leq a_2$, *then* $max(|a_1 - a_2|, |b_1 - b_2|) \geq min(c_1, c_2)$.

Proof. Prove by contradiction. Assume $max(|a_1 - a_2|, |b_1 - b_2|) < min(c_1, c_2)$, then we have

$$a_2 - a_1 < c_1,\ b_1 - b_2 < c_2 \tag{3.18}$$

Combine with the theorem assumptions, we can derive

$$a_2 < a_1 + c_1 \leq b_1,\ b_1 < b_2 + c_2 \leq a_2 \tag{3.19}$$

The two inequalities above conflict with each other, therefore the lemma is proven. □

Theorem 3.22. *Assume each job has a non-zero execution time and is executed in single-core systems non-preemptively. Any schedule $\boldsymbol{s}$ obtained by solving the linear programming problem (3.13) is local optimal.*

Proof. Prove by contradiction. Assume $\boldsymbol{s}$ is not a local optimal solution. This implies the existence of another feasible solution $\boldsymbol{s}^*$ such that $\mathcal{F}(\boldsymbol{s}^*) < \mathcal{F}(\boldsymbol{s})$, where $\|\boldsymbol{s} - \boldsymbol{s}^*\| < \delta$, and $\delta > 0$ is a very small number. Denote the job order of $\boldsymbol{s}$ and $\boldsymbol{s}^*$ as $\mathcal{O}$ and $\mathcal{O}^*$, respectively. Then we must have $\mathcal{O}^* \neq \mathcal{O}$ because $\boldsymbol{s}$ is optimal for the problem (3.13) given the job order $\mathcal{O}$.

Since we are considering a non-preemptive single-core platform, no jobs can run in parallel. Furthermore, since the job orders are different, there must exist at least two jobs $J_{i,k}$ and $J_{j,l}$, whose relative execution order is different. Without loss of generality, assume $J_{i,k}$ runs earlier than $J_{j,l}$ in $\mathcal{O}$, and $J_{j,l}$ runs earlier in $\mathcal{O}^*$. Mathematically speaking, that means:

$$s_{i,k} + C_i \leq s_{j,l}, \;\; s^*_{j,l} + C_j \leq s^*_{i,k} \tag{3.20}$$

Based on Lemma 3.21, we have $max(|s_{i,k} - s^*_{i,k}|, |s_{j,l} - s^*_{j,l}|) \geq min(C_1, C_2)$. Therefore, $\|\boldsymbol{s} - \boldsymbol{s}^*\| \geq max(|s_{i,k} - s^*_{i,k}|, |s_{j,l} - s^*_{j,l}|) \geq min(C_1, C_2) > \delta$, which causes a contradiction. Therefore, the theorem is proved. □

Thus, the 1-opt schedule $\boldsymbol{s}^{1*}$ from Theorem 3.20 for a non-preemptive single-core system is also local optimal.

3.6.4 Optimization Algorithm Towards 1-opt Schedules

Following Theorem 3.20, we can design a simple algorithm to search for better job orders iteratively. The algorithm will update the job order following Eq. (3.21) and terminate when

the iterations converges, i.e. $\mathcal{O}^{(k+1)} = \mathcal{O}^{(k)}$.

$$\mathcal{O}^{(k+1)} = \underset{\mathcal{O} \in \boldsymbol{\mathcal{B}}(\mathcal{O}^{(k)}) \cap \Omega}{\operatorname{argmin}} \mathcal{F}(\pi^*(\mathcal{O})) \tag{3.21}$$

where $\pi^*(\mathcal{O})$ is the optimal job order schedule of $\mathcal{O}$, Ω denotes the set of schedulable job orders following Definition 3.14.

Theorem 3.23. *An iterative algorithm that updates the job order variables following Eq. (3.21) will terminate after a finite number of iterations, and the solution found is 1-opt.*

Proof. The iterative algorithm will terminate after a finite number of iterations because a new iteration is initiated only after finding a feasible, better solution in previous iterations. Considering that the optimal objective function value is positive, the algorithm is guaranteed to terminate after a finite number of iterations. When the algorithm terminates, the two conditions in Theorem 3.20 are both satisfied and therefore the solution is 1-opt. □

3.7 Enhancing TOM: Strategies for Improved Performance and Efficiency

3.7.1 Skipping Unschedulable Job Orders

Although the feasibility of a job order can be analyzed by solving the linear programming problem in problem (3.13), the average run-time complexity is $O(N^{2.5})$ [25]. Therefore, we propose the following lightweight lemma to quickly examine whether a job order is schedulable with $O(N)$ complexity. These lemmas are necessary, but not sufficient, conditions of schedulability:

Lemma 3.24. *Given a job order $\mathcal{O}$, if there exists one job $J_{i,k}$ whose scheduling finish time $\mathcal{T}^f_{i,k}$ precedes its scheduling start time $\mathcal{T}^s_{i,k}$, then $\mathcal{O}$ is not schedulable.*

Lemma 3.25. *Given a job order $\mathcal{O}$, if the maximum number of concurrent jobs exceeds the total number of processors, then $\mathcal{O}$ is not schedulable.*

Proofs of these lemmas are straightforward as they breach either constraints (3.13a) or (3.13b).

3.7.2 More Relaxed Constraints in LP

The solution quality of an optimization problem could become better if its constraints are relaxed. In problem (3.13), although we cannot relax the constraints (3.13a) and (3.13b) (hard schedulability constraints), we can relax the job order constraint (3.13c) because it is only necessary to maintain the relative order of jobs that influence the objective functions (because not all the tasks contribute to the cause-effect chains or merges) to guarantee that the objective functions can be equivalently transformed into linear functions.

Example 9. Continue with Example 1, given a job order $\mathcal{O} = \{\mathcal{T}^s_{0,0}, \mathcal{T}^f_{0,0}, \mathcal{T}^s_{1,0}, \mathcal{T}^f_{1,0}, \mathcal{T}^s_{0,1}, \mathcal{T}^f_{0,1}, \mathcal{T}^s_{2,0}, \mathcal{T}^f_{2,0}\}$, suppose we only have one processor and want to optimize the reaction time of the cause-effect chain $\mathcal{C} = \{\tau_0 \to \tau_2\}$. In this case, the optimal schedule $\pi^*(\mathcal{O}) = [s_{0,0}, s_{0,1}, s_{1,0}, s_{2,0}] = [7, 10, 8, 11]$, the worst-case reaction time is 7 from the job chain $\{J_{0,0} \to J_{2,0}\}$. Since $J_{1,0}$ does not influence the length of the cause-effect chain $\mathcal{C} = \{\tau_0 \to \tau_2\}$, only enforcing the relative job order among $\{J_{0,0}, J_{0,1}, J_{2,0}\}$ is enough to transform the objective function (3.5) into linear functions. Then the optimal schedule with relaxed constraints become $\boldsymbol{s}^{relaxed} = [s_{0,0}, s_{0,1}, s_{1,0}, s_{2,0}] = [9, 10, 0, 11]$. The worst-case reaction time is reduced to 5.

3.7.3 Simple Job Order Scheduler

In cases when the run-time complexity becomes a major performance bottle-neck, we can use a heuristic scheduling algorithm with $O(N)$ complexity to replace solving the linear programming problem (3.13) that usually requires $O(N^{2.5})$ time complexity [25]. The simple job order scheduler adopts a First-In-First-Out scheduling policy. A job becomes ready for execution after satisfying two conditions: (i) its release time has passed; (ii) its previous job scheduling time has happened. Algorithm 4 shows the pseudocode of the simple job order scheduler in a simulation environment.

Algorithm 4 Simple Job Order Scheduler

Input: Job order $\mathcal{O}$ **Output:** Schedule $\boldsymbol{s}$

```
1: t = 0                                                    ▷ Record current time
2: for each T_i in O do
3:     J_i = GetJob(T_i)
4:     if T_i is job scheduling start time then
5:         t = max(t, J_i.release_time, NextProcessorAvailableTime() )
6:         s_i = t
7:     else
8:         if s_i + C_i ≤ t then
9:             t = s_i + C_i, f_i = s_i + C_i
10:        else
11:            return 0                                     ▷ O is unschedulable
12:        end if
13:    end if
14: end for
15: return s
```

Example 10. Continue with Example 9, consider the same job order $\mathcal{O} = \{\mathcal{T}^s_{0,0}, \mathcal{T}^f_{0,0}, \mathcal{T}^s_{1,0}, \mathcal{T}^f_{1,0}, \mathcal{T}^s_{0,1}, \mathcal{T}^f_{0,1}, \mathcal{T}^s_{2,0}, \mathcal{T}^f_{2,0}\}$. If there is only one computation core, the schedule obtained from the simple order scheduler is $[s_{0,0}, s_{0,1}, s_{1,0}, s_{2,0}] = [0, 10, 1, 11]$. In case of two cores, the schedule is $[s_{0,0}, s_{0,1}, s_{1,0}, s_{2,0}] = [0, 10, 0, 10]$.

Despite its fast speed, the simple job order scheduler suffers from two major disadvantages: non-exact schedulability analysis and non-optimal schedule without any theoretical guarantee. It is only encouraged to use if solving the problem (3.13) iteratively suffers from a big time-out issue.

3.8 Implementation details

3.8.1 Initial Solution Estimation

In the experiments, we use a simple list-scheduling method [93] to obtain an initial schedule. If multiple jobs become ready, jobs with the least finish time will be dispatched first. The processor assignments are decided based on a simple First-Come-First-Serve strategy. In practice, other methods can also be used to obtain a feasible initial schedule.

3.8.2 Faster Implementation within Time Limits

TOM is implemented slightly differently from (3.21) for faster run-time efficiency. When searching for an optimal job order $\mathcal{O}^{(k)*}$ within $\boldsymbol{\mathcal{B}}(\mathcal{O}^{(k)})$, we immediately accept a new job order $\mathcal{O}$ if it improves $\mathcal{O}^{(k)}$. Algorithm 5 shows the pseudocode of one single iteration. In line 3, $\boldsymbol{\mathcal{B}}^{J_i}(\mathcal{O}^{tmp})$ denotes the adjacent job order permutation of $\mathcal{O}^{tmp}$ by only changing the index of J_i's job scheduling time. $\mathcal{O}^{tmp}$ will be updated if a better job order is found. Following Theorem 3.23, algorithm 5 also finds 1-opt solutions after algorithm termination.

Algorithm 5 Single Iteration of TOM

Input: Job order $\mathcal{O}^{(k)}$, job set $\boldsymbol{J}$ containing all jobs in a hyper-period
Output: $\mathcal{O}^{(k+1)}$

$\mathcal{O}^{tmp} = \mathcal{O}^{(k)}$
for each job J_i in $\boldsymbol{J}$ **do**
  for each job order $\mathcal{O}$ in $\boldsymbol{\mathcal{B}}^{J_i}(\mathcal{O}^{tmp})$ **do**
    if $\mathcal{F}(\pi^*(\mathcal{O})) < \mathcal{F}(\pi^*(\mathcal{O}^{tmp}))$ **then**
      $\mathcal{O}^{tmp} = \mathcal{O}$
    end if
  end for
end for
$\mathcal{O}^{(k+1)} = \mathcal{O}^{tmp}$
return $\mathcal{O}^{(k+1)}$

3.8.3 When to Assign Processor

A simple First-Come-First-Serve (FCFS) policy is used for processor assignment for each job. In experiments, we utilize the simple job order scheduler (Section 3.7.3) to generate the processor assignment before evaluating a job order (i.e., solving problem (3.13)). After obtaining the processor assignments, we formulate the resource-bound constraints for problem (3.13).

3.8.4 Worst-Case Complexity Analysis

The overall algorithm's complexity depends on the complexity of each iteration and the total number of iterations. In the experiments, TOM usually terminates in less than 10 iterations. Following (3.21), the cost of each iteration depends on the number of job orders to search and the cost to evaluate a single job order (problem (3.13)). In the worst case, the total number of adjacent job order permutations could be $O(N^3)$. However, techniques from Section 3.7.1 can greatly reduce the possible permutations. Evaluating a single job order has two steps:

obtaining a schedule and then evaluating the objective function. The former could be as fast as $O(N)$ if a simple job order scheduler is used. In terms of solving the linear program, the complexity could increase to $O(N^{2.5})$ in average case [25] (In reality, since problem (3.13) is very sparse, the real run-time speed should be much faster than $O(N^{2.5})$). Finally, evaluating the objective function given a schedule requires $O(N^2)$ complexity in worst cases.

Overall, the worst-case complexity in one iteration is $O(N^3 \cdot (N^{2.5} + N^2))$ if an optimal job order scheduler (solving problem (3.13)) is used. However, most experiments finish optimizing task sets of thousands of jobs within 1000 seconds, which suggests the average time complexity to be $O(N^4)$.

3.9 Extensions and Limitations

This section briefly discusses several possible extensions and leaves the experiment verification to future works.

3.9.1 Alternative Objective Functions

Apart from the objective functions shown in Section 3.4.3 and 3.4.4, TOM also supports other forms of objective functions: such as linear combination of data age, reaction time, and time disparity. Besides, TOM can also optimize nonlinear functions of different timing metrics (such as jitters of end-to-end latency) and solve them with nonlinear programming methods [76, 100], though without the 1-opt or local-optimal guarantee anymore.

3.9.2 Extension For Preemptive Scheduling

While the TOM framework is designed for non-preemptive time-triggered scheduling systems, it can be extended to work with preemptive systems. Firstly, similar to the start time variables, an extra set of finish time variables has to be incorporated into the optimization problem formulation. The schedulability analysis constraints (Section 3.4.5) have to be replaced with the demand bound function used in [10]. The concept of job order remains the same because it already incorporates the finish time.

3.9.3 Finding Feasible Initial Schedules

The TOM optimization framework can also be utilized to find feasible schedules. This subsection briefly discusses the theoretical foundations. Since feasibility is a binary metric that is not friendly for optimization, we utilize "tardiness' as the optimization objective function (similar to [100]). The feasibility optimization problem is formulated as follows:

$$\underset{\boldsymbol{s}}{\text{Minimize}} \sum_{i=0}^{n-1} \sum_{k=0}^{H/T_i-1} \text{Barrier}(kT_i + D_i - C_i - s_{i,k}) \tag{3.22}$$

$$\text{Barrier}(x) = \begin{cases} 0 & x \geq 0 \\ -x & x < 0 \end{cases} \tag{3.22a}$$

Subject to :

$$\forall i \in \{0, ..., n-1\}, \forall k \in \{0, ..., H/T_i - 1\}, kT_i \leq s_{i,k} \tag{3.22b}$$

$$\text{ResourceBound}(\boldsymbol{s}) = 0 \tag{3.22c}$$

Theorem 3.26. *If a solution $\boldsymbol{s}$ can reduce the objective function in problem (3.22) into 0 while also being feasible for problem (3.22), then $\boldsymbol{s}$ is a schedulable schedule.*

Proof. If the objective function is reduced to 0, no jobs violate the deadline constraints. Combined with the job release constraint (3.22b) and processor overloading constraint (3.22c), the schedule $\boldsymbol{s}$ is schedulable by definition. □

Theorem 3.27. *List scheduling can always provide a feasible initial solution to problem* (3.22).

Proof. The schedule found by list scheduling is always feasible for problem (3.22) because a job is dispatched for execution whenever there is an idle processor (satisfying constraint (3.22c)) after the job is released (constraints (3.22b)). □

Theorem 3.28. *The problem* (3.22) *can be equivalently transformed into a linear programming problem after adding an extra set of job order constraints (the inequality constraint* (3.13c)*).*

Proof. Following Theorem 3.12, we only need to prove that the objective function (3.22) can be transformed into linear functions. This can be easily done by introducing an artificial variable $z_{i,k}$ for each term following [102]. After that, the objective function becomes:

$$\underset{\boldsymbol{s}}{\text{Minimize}} \sum_{i=0}^{n-1} \sum_{k=0}^{H/T_i-1} z_{i,k} \tag{3.23}$$

with extra linear constraints:

$$\begin{aligned} &\forall i \in \{0, ..., n-1\}, \forall k \in \{0, ..., H/T_i - 1\}, \\ &z_{i,k} \geq 0 \ \ \& \ \ z_{i,k} \geq -1 \cdot (kT_i + D_i - C_i - s_{i,k}) \end{aligned} \tag{3.24}$$

Since both the objective functions and the constraints are linear functions after transformation, the theorem is proved. □

The theorems above show that TOM can also solve the feasibility problem (3.22). It is also guaranteed to perform better than simple scheduling heuristics such as list scheduling because TOM utilizes them as initial solutions.

3.9.4 Limitations

Compared with global optimality, 1-opt provides a weaker form of theoretical guarantee. However, in general cases, obtaining global optimal solutions requires significantly higher computation costs. Therefore, given the same computation costs, 1-opt could potentially achieve better performance, as shown in our experiments.

TOM's computation cost depends on the number of jobs within a hyper-period. Therefore, there could be a higher computation cost in non-harmonic task sets. However, in realistic time-triggered scheduling (TTS) systems [72], there cannot be too many jobs within a hyper-period because that would incur a high overhead in task management and scheduling. Therefore, it is expected that the computation cost associated with TOM should be reasonably low in real-world systems.

3.10 Experiment

The proposed framework was implemented in C++ and tested on a computing cluster (AMD EPYC 7702 CPU). We consider the following methods in experiments:

- List Scheduling [93]. Whenever there are available processors, it dispatches the ready job with the least finish time for execution.

- Simulated Annealing [94]. A general heuristic method for optimization problems. The

initial temperature is $1e8$, and the cooling rate is 0.99, which encourages the algorithm to explore the solution space. The initial schedule is obtained from the list scheduling, the same as TOM.

- Verucchi20 [95]. It was proposed to minimize the worst-case data age and reaction time in multi-rate DAG. The code implementation is adopted from their official release repository. If it does not run out of time, its solution quality is close to optimal solutions. To the best of our knowledge, it is also the most recent state-of-the-art work that considers a similar problem setting.

- TOM. The optimization framework proposed in this chapter. When solving problem (3.13), CPLEX [26] is used to find optimal solutions.

- TOM_SimpleScheduler. Similar to TOM, except that the simple job order scheduler (Section 3.7.3) instead of LP is used when obtaining a schedule from a job order.

- TOM_Extended. Similar to TOM, except that we also enabled the relaxations on the linear programming problem's constraints, which is introduced in Section 3.7.2.

If one method runs time-out without a feasible solution, we use the results of list scheduling during the result analysis.

3.10.1 Task Set Generation and Results

The simulated DAG task sets are generated following a real-world automotive benchmark [56], all the tasks' periods are randomly generated from a limited set {1, 2, 5, 10, 20, 50, 100, 200, 1000}, with relative probability distribution: {3, 2, 2, 25, 25, 3, 20, 1, 4}. The overall task set's utilization is set to $0.9m$, where m is the number of cores available, 4 in our experiments. Each task's worst-case execution time is generated by UUnifast [13] while

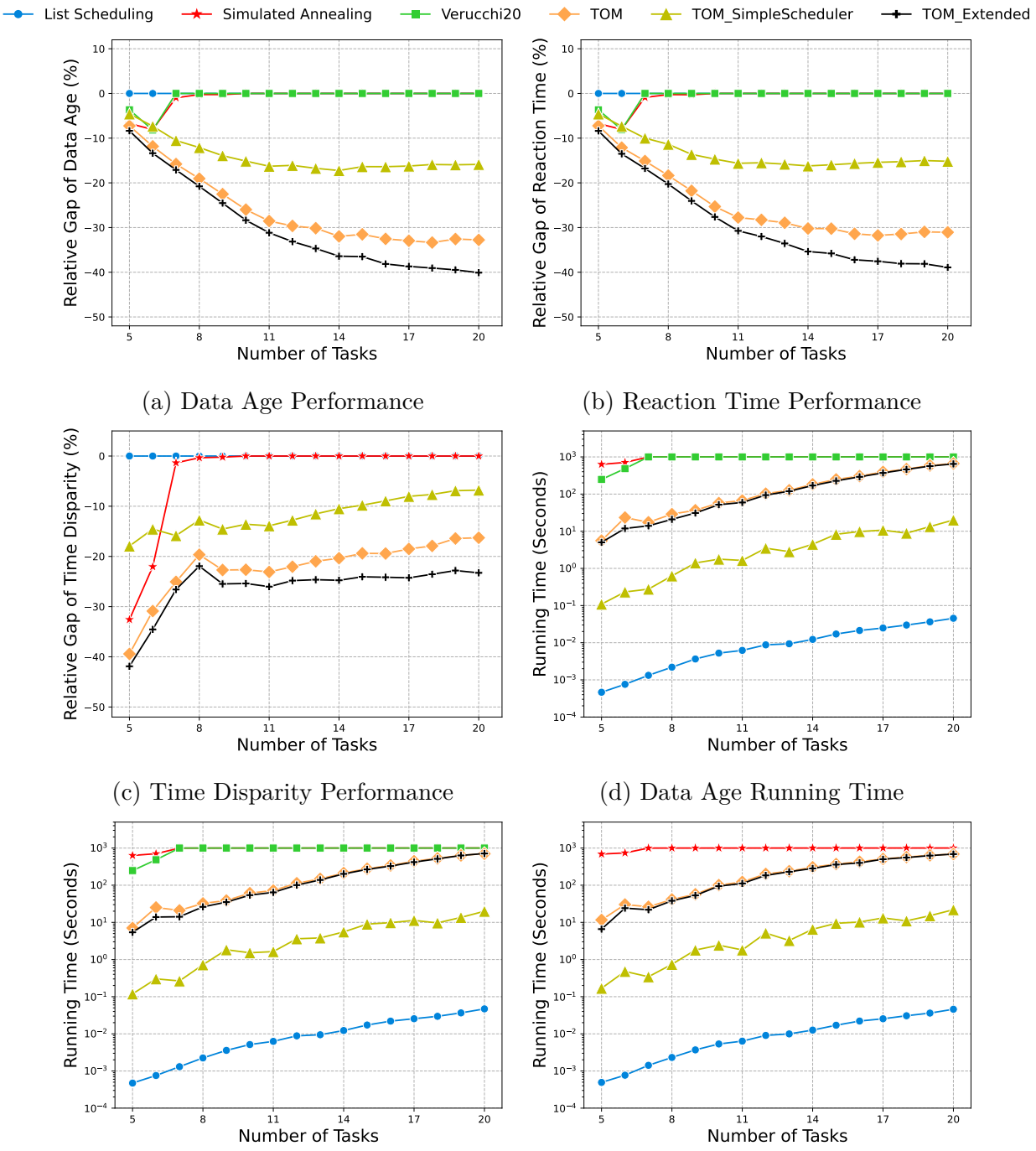
following the multi-core adaptation implementation in [100]. Each task's relative deadline is the same as its period. Task sets generated in this way usually have hundreds or *thousands* of jobs to schedule.

Task dependencies are generated randomly following He *et al.*[46]. After generating individual tasks, we go through each pair of tasks and randomly add an edge from one task to another with a given probability, 0.9 in our experiments (smaller probabilities are usually insufficient to generate many cause-effect chains in the DAG). The number of tasks in a task set ranges from 5 to 20. Cause-effect chains are generated as the paths between random pairs of tasks using the shortest path algorithm in Boost Graph Library [88]. Task merges are generated by randomly selecting a sink task and then collecting all source tasks on which the sink task directly depends.

For a task set with n tasks, there are n to $2n$ random cause-effect chains and $\lfloor 0.25n \rfloor$ to n random task merges. The maximum number of source tasks in a merge varies from 2 to 9 following ROS [64]. The lengths and activation patterns of the cause-effect chains adhere to distributions outlined in Table VI and Table VII of the automotive benchmark [56]. To meet distribution criteria, we initially generate plenty of task sets, evaluate the likelihood for each task set, and then sample 1000 random task sets weighted by the likelihood for each given number of tasks. All task sets are schedulable under the list scheduling method. The run-time limit for scheduling one task set is 1000 seconds per method.

We tested the performance of each method in optimizing data age, reaction time, and time disparity separately. The experiment results are reported in Fig. 3.5. All performance gaps are compared against the list scheduling method:

$$\frac{\mathcal{F}_{\text{method}} - \mathcal{F}_{\text{List\_Scheduling}}}{\mathcal{F}_{\text{List\_Scheduling}}} \times 100\% \tag{3.25}$$



(a) Data Age Performance

(b) Reaction Time Performance

(c) Time Disparity Performance

(d) Data Age Running Time

(e) Reaction Time Running Time

(f) Time Disparity Running Time

Figure 3.5: Performance gap and running time for optimizing end-to-end latency and time disparity on synthetic task sets.

3.10.2 Result Analysis and Discussion

Overall, TOM and its extensions significantly outperform other methods in various experiments. Next, we provide a more detailed analysis of different aspects.

Comparison with baseline methods

Compared with other baseline methods, the performance improvements of TOM and TOM_Extended are not obvious when the number of tasks is small ($n = 5$). This is because the solution space is very small and most methods can find good solutions. However, as the number of tasks increases, Verucchi20 quickly reaches time limits and can barely find schedulable schedules or schedules with low end-to-end latency. Simulated annealing always starts its iteration with a feasible schedule. However, due to its inefficient solution space exploration techniques, it usually requires a long time to find a good solution, which often exceeds the given time limit and therefore cannot show much performance improvement. In contrast, guided by 1-opt, TOM and TOM_Extended are able to explore the solution space efficiently while still maintaining good solution quality. These experiment results show the benefits of both 1-opt optimality and the proposed TOM optimization algorithms.

TOM vs TOM_SimpleScheduler

The performance improvements of TOM against TOM_SimpleScheduler show the benefits of the LP formulation. Compared with simple heuristics such as list scheduling, LP explores a larger solution space, can find non-work-conserving schedules, and thus achieves better solution quality. The disadvantage of the LP approach is the higher computation cost. To compensate for the extra computation costs, many heuristics are proposed in this chapter without sacrificing the theoretical guarantee, such as using fast necessary conditions

to filter un-schedulable job orders (Section 3.7.1), exploring the sparse structure in implementation (the resource bound constraints are sparse linear constraints). However, TOM _SimpleScheduler could still be an option in situations with many tasks/jobs.

TOM vs TOM_Extended

The performance improvements of TOM_Extended against TOM show the effectiveness of the heuristics (Section 3.7.2) to further improve upon 1-opt while maintaining a similar runtime speed. Since the results obtained from both TOM_Extended and TOM are 1-opt (if not running time-out), it implies that there are potentially many 1-opt solution candidates with varying solution qualities in the whole solution space. If applicable, utilizing heuristics to further improve upon 1-opt solutions is beneficial.

Time-out issue

It is possible that TOM does not finish iterations before running time out. In these cases, TOM degrades into heuristic algorithms without a theoretical guarantee. However, the trend in Fig. 3.5 shows that running time-out does not seriously degrade the solution quality even though more than 30% cases running time out when $n = 20$ (around 4000 jobs per task set). We expect TOM to work reasonably well for task sets with less than 10^4 jobs if the time limit is 1000 seconds. Optimizing larger task sets, such as those with 10^5 jobs, would require a much longer time limit.

Data Age vs Reaction Time

Experiments show that data age and reaction time optimization have similar results. Furthermore, reducing one metric usually reduces the other, which is broadly consistent with

the findings in [42]. This observation may improve the algorithm efficiency in cases where both data age and reaction time need to be optimized: we may just consider only one metric in the objective function and leave the other out.

3.10.3 Time Disparity Optimization Result

Although the overall results on time-disparity optimization are good, Fig. 3.5c shows that the performance seems to become worse when the number of tasks increases from 5 to 8. This is mainly due to the nature of the problem itself, rather than the limitations of the optimizers. For example, consider two merges where one merge has 2 source tasks and 1 sink task, and another merge has the same sink task, the same 2 source tasks, and 2 more extra source tasks. In this case, the maximum source time disparity of the second merge could never become smaller than the first merge. In practice, adding more source tasks does not necessarily make the list scheduling perform worse after reaching certain limits, but it does make the optimization more difficult, and limits the performance improvements even for global optimal solutions.

3.11 Conclusions

In this chapter, we investigate a multi-rate DAG scheduling problem to reduce the worst-case end-to-end latency and/or time disparity metrics. Given the potentially vast number of variables within the solution space, we advocate for guiding the scheduling design with 1-opt. Our optimization algorithm introduces a novel technique called *job order* to partition the solution space into multiple convex sub-spaces. This partitioning strategy allows utilizing linear programming to minimize DARTD within each subspace. Building upon

this partition, our algorithm iteratively traverses among the sub-spaces, ensuring that the output is 1-opt. In contrast to alternative optimization algorithms, such as meta-heuristics algorithms lacking any theoretical performance guarantees, or optimal algorithms that may require exponential run-time complexity, the 1-opt algorithm balances the trade-off between theoretical performance guarantee and run-time complexity. We rigorously prove that our optimization algorithm achieves 1-opt solutions while maintaining polynomial run-time complexity. Further optimization heuristics are also proposed to improve the algorithm's performance and efficiency without compromising the 1-opt solution guarantee. Experimental results indicate significant improvements over state-of-the-art methods in both performance and computational efficiency.

Chapter 4

A Wait-Free Communication Protocol for Data Consistency in Multicore Embedded Systems

4.1 Abstract

Ensuring data consistency in multicore embedded systems with concurrent data read-write operations presents significant challenges, especially under real-time constraints. Instead of relying on traditional lock-based protocols that worsen real-time schedulability due to their blocking times, this chapter adopts wait-free protocols that duplicate data communication buffers to avoid the blocking times. Specifically, we first extend both the Dynamic Buffering Protocol (DBP) and the Temporal Concurrency Control Protocol (TCCP) to multicore settings under fixed-priority partitioned scheduling. We further propose a novel wait-free protocol, Partitioned Combined-DBP-TCCP Protocol (PCDT), for data communication through shared memory. This method enables different data consumers to selectively adopt either DBP or TCCP based on their specific timing requirements to optimize resource allocation.

Beyond this novel protocol design, we introduce memory-aware system-level optimization techniques to further reduce the memory footprint of PCDT. Specifically, we adopt a priority

assignment strategy, PA-MBTT [112, 115], and show that when combined with preemption threshold scheduling, it significantly improves memory efficiency over standard Deadline Monotonic scheduling. We also design a Wait-Free-Aware Partitioning (WFAP) heuristic and its local search variant (WFAP+) to optimize task partition with respect to the memory overhead of wait-free communication. These methods enable scalable, memory-efficient deployment across various system configurations.

Extensive numerical experiments demonstrate that PCDT achieves more than 36.01% reduction in memory footprint compared to PDBP or PTCCP while ensuring wait-free, non-blocking data access and timely updates for each data reader. Combined with PA-MBTT and WFAP+, the total memory usage can be reduced by an average of 53.67%. A case study in an automotive advanced driver assistant system further underscores the practicality and effectiveness of the proposed approach, achieving more than 43.43% memory savings in the real-world setting.

4.2 Introduction

Ensuring efficient data communication and consistency in multicore embedded systems is critical in several application domains such as automotive, aerospace, and industrial automation. Multicore architectures offer enhanced computational power, however, they also introduce complexities in coordinating concurrent data access across tasks with strict timing requirements, a challenge that is especially vital in hard real-time systems. Traditionally, lock-based protocols, such as Priority Ceiling Protocol (PCP) [87], Stack Resource Policy (SRP) [9], along with their multicore extensions, MPCP [84] and MSRP [37], have been widely used to coordinate shared data access. Although effective in maintaining data consistency, lock-based approaches inherently introduce blocking time as tasks must wait for locks

to be released, which can lead to priority inversion and unpredictable latency. Such delays are particularly undesirable in real-time systems, where timing predictability is important for safe and reliable operation.

To overcome the long blocking time of lock-based methods, lock-free and wait-free communication protocols have gained significant attention in the field of real-time embedded systems. Lock-free protocols allow tasks to access shared data without blocking, enabling progress even when multiple tasks access the same data simultaneously [5, 35]. In a lock-free approach, a reader task reads data optimistically, without waiting for exclusive access. At the end of each operation, the reader performs a consistency check; if a concurrent write occurred during its read, the task repeats the operation to ensure data consistency. While lock-free protocols reduce blocking and improve system responsiveness, they may still introduce delays due to these retries, making them less predictable for real-time constraints.

In contrast, wait-free protocols guarantee that each operation is completed within a finite number of steps, regardless of contention from other tasks, thereby eliminating blocking entirely. These protocols typically rely on using multiple buffers to replicate communication data, enabling concurrent access while maintaining consistency. This property makes wait-free synchronization protocols particularly well-suited for real-time systems with stringent timing requirements. For instance, the Temporal Concurrency Control Protocol (TCCP) [23] leverages task lifetime, allocating resources based on task worst-case response time to optimize data consistency and memory efficiency. Similarly, the Dynamic Buffering Protocol (DBP) [90] allows concurrent readers to access dynamically allocated buffers without locking, maintaining data consistency across tasks. However, these approaches can lead to notable memory overhead due to the replication of communication data across multiple buffers, which may strain resources in memory-constrained environments. Consequently, optimizing memory usage while preserving real-time guarantees remains a critical challenge, especially

in multicore real-time embedded systems where resource contention and scalability further exacerbate the issue.

In recent years, researchers have proposed optimization techniques to mitigate memory overhead in wait-free protocols. These approaches enable the selective use of lock-based or wait-free synchronization protocols based on task requirements and system constraints, often employing Integer Linear Programming (ILP) to achieve optimal core allocation. However, ILP-based methods are not always scalable for larger systems, necessitating the use of heuristics to provide efficient solutions [2, 3].

Existing solutions largely focus on lock-based protocols, and the interplay between wait-free protocols and system-level scheduling strategies like priority assignment and task partitioning remains underexplored. This motivates the development of wait-free protocols that are not only adaptable to multicore systems but also resource-efficient. In this chapter, we consider a partitioned scheduling real-time system in which tasks are statistically assigned to specific cores and do not migrate during execution. We propose the Partitioned Combined-DBP-TCCP Protocol (PCDT), a novel wait-free protocol designed specifically for such multicore architectures. PCDT leverages both DBP and TCCP, allowing data readers to adopt either protocol based on individual timing requirements to optimize resource allocation. This partitioned approach enables systems to benefit from both protocols, achieving a reduced memory footprint while ensuring data consistency and timing predictability.

To further enhance memory efficiency and system schedulability, we incorporate two key system-level optimization techniques: A priority assignment strategy based on the Maximum Blocking Time Tolerance (MBTT) that minimizes buffer demand under wait-free communication while leveraging preemption thresholds to reduce data buffers. A family of task partitioning heuristics, Wait-Free-Aware Partitioning (WFAP), and its local search-enhanced variant WFAP+, which are tailored to reduce communication-induced memory costs across

multicore partitions.

Our results demonstrate the complementary benefits of protocol design and system-level enhancements. WFAP provides an average of 30.72% memory savings compared to baseline partitioning, while WFAP+ can achieve an average of 53.67% savings in offline settings. PA-MBTT with preemption thresholds consistently outperforms traditional fixed-priority strategies in memory efficiency. Together with PCDT, the integrated system can handle high data-sharing demands while significantly reducing the required memory footprint.

The contributions of this chapter are summarized as follows:

- We extend existing wait-free communication protocols to support multicore, partitioned scheduling in real-time embedded systems.
- We propose the PCDT protocol, a hybrid wait-free data communication method designed specifically for multicore real-time systems, which further optimizes memory usage.
- We leverage a priority assignment strategy (PA-MBTT) that exploits preemption threshold scheduling to reduce memory usage in wait-free settings.
- We propose two wait-free-aware task partitioning techniques (WFAP and WFAP+), designed to minimize inter-core communication costs and improve schedulability.
- We present the algorithm design for the PCDT protocol, providing a detailed explanation of its structure and how it can be practically applied.
- We evaluate the proposed solutions through extensive simulation and a real-world case study. Results show that PCDT combined with PA-MBTT and WFAP+ achieves an average of 53.67% memory reduction and maintains high schedulability even under

heavy communication loads. In an automotive Advanced Driver-Assistance System (ADAS) case study, we observe 43.43% memory savings using our integrated approach.

4.3 Related Work

Data Consistency Mechanisms. Efficient data communication in real-time embedded systems has long been a research focus. We first review the related work on mechanisms for guaranteeing data consistency, focusing on the wait-free methods. Early work by Kopetz and Reisinger [6] formally defined the Non-Blocking Write (NBW) protocol, establishing theoretical guarantees for wait-free protocol. Later, Chen and Burns [23] introduced the Temporal Concurrency Control Protocol (TCCP), which leverages task worst-case response times to manage buffer allocation, significantly improving memory efficiency. The Dynamic Buffering Protocol (DBP) [90] enables wait-free communication by associating multiple buffers with each shared variable to eliminate blocking. Huang et al. [48] adapted these ideas to optimize interprocess communication by reducing overhead in constrained environments. Building on these efforts, Wang et al. [98] proposed the Combined DBP-TCCP (CDT) protocol, which integrates the advantages of DBP and TCCP in a single-core context. CDT highlights the importance of both task response times and priority assignments in determining the memory footprint of wait-free protocols.

On multicore architectures, Zeng et al. [111] summarized the wait-free methods and lock-based approaches in terms of their tradeoffs on timing and memory. It also extended the wait-free methods for the purpose of flow preservation of synchronous reactive models (a critical requirement in the model-based design of control-centric applications). Han et al. [44] implemented the lock-based and wait-free methods on two open source RTOSes. It also presented heuristic algorithms to select these mechanisms to minimize the memory overhead

of wait-free methods while satisfying the schedulability constraints.

Optimization for Systems with Partitioned Fixed Priority and Preemption Threshold Scheduling. Task partitioning has been extensively studied for multicore real-time systems with partitioned scheduling. A broader review of task partitioning techniques can be found in the survey [89]. In particular, simple heuristics such as Best-Fit Decreasing (BFD) [52] offer efficient but suboptimal allocations. Baruah and Fisher [12] highlighted the limitations of bin-packing heuristics for sporadic tasks, particularly when synchronization costs are overlooked. Lakshmanan et al. [59] proposed partitioned scheduling approaches that account for blocking times from lock-based approaches for accessing shared resources. The Synchronization-Aware Partitioning Algorithm (SPA) [58] and Blocking-Aware Partitioning Algorithm (BPA) [75] further improved partitioning by considering inter-task attractions induced by shared resource usage. However, these methods primarily target lock-based synchronization and assume a small number of shared resources, limiting their scalability in systems with large-scale label-based communication typical of modern wait-free protocols. Resource-Oriented Partitioning (ROP) [49, 109] introduced distributed sharing policies to reduce contention, although its dependence on specific scheduling policies reduces its general applicability. Han et al. [45] studied the blocking aware partitioning on heterogeneous multicore architectures. Al-bayati et al. [2, 3] considered wait-free methods as an alternative for data consistency, and presented algorithms for partitioning and selection between wait-free and lock-based approaches. Compared to these studies, our work is unique in its focus on wait-free methods as the only mechanism for ensuring data consistency.

Since in partitioned scheduling, each core is scheduled separately, we also review the related work on the priority and preemption threshold assignment. For priority assignment, common schemes include Rate Monotonic (RM) [67], Deadline Monotonic (DM) [60], and Audsley's

Optimal Priority Assignment (OPA) [7]. More priority assignment strategies can be found in these survey works [27, 28]. More recently, Zhao et al. [122, 123, 124, 125, 126, 127] developed a series of optimization algorithms for priority assignment that are efficient and optimal. However, they all rely on that the associated schedulability analysis satisfy the compatibility conditions of OPA, except [124]. Still, the framework in [124] imposes optimality conditions that are violated in preemption threshold scheduling.

Preemption Threshold (PT) scheduling offers a balanced approach between fully preemptive and non-preemptive scheduling. Initially proposed by Wang and Saksena [103], PT assigns each task a base priority and a preemption threshold. A task can only be preempted by another task with a priority higher than its threshold. This significantly reduces unnecessary preemptions, improves cache locality, decreases context-switch overhead, and enables memory optimizations such as stack sharing. Saksena and Wang [86] demonstrated the scalability of PT for large systems. Ghattas and Dean [38] analyzed stack usage under PT, showing its ability to achieve stack-optimal configurations. Bril et al. [16] incorporated cache-related preemption delay analysis into PT, enhancing its practical applicability. Preemption threshold scheduling has been extended to various settings to save memory and stack usage, including mixed-criticality systems [116, 117, 118, 120, 121], global fixed-priority scheduling on multicore [97], cache partitioning [17, 40, 96], and is supported by the automotive standard AUTOSAR [119], further demonstrating the versatility of this approach in modern embedded applications. However, none of the above considered the memory minimization for wait-free methods.

4.4 System Model and Background

In this section, we describe the system model and provide relevant background to establish a foundation for analyzing wait-free synchronization in multicore real-time systems. We consider a partitioned, fixed-priority preemptive scheduling scenario on a multicore platform, where tasks are statically assigned to specific cores, each with defined temporal and communication attributes.

4.4.1 Task Model

We model system as a set of tasks $\mathcal{T} = \{\tau_1, \tau_2, \ldots, \tau_n\}$, where each task τ_i is characterized by the following parameters $(T_i, C_i, D_i, \pi_i, \theta_i, core_i)$. Each task is assigned to a specific core $core_i \in \{1, 2, \ldots, m\}$, and task migration across cores is not allowed during execution. The period T_i represents the interval at which τ_i is released. The worst-case execution time (WCET) C_i denotes the maximum time task τ_i takes to execute on its assigned core $core_i$ under the worst-case conditions, including the time required for all reading and writing operations in data communication. The relative deadline D_i indicates the maximum allowable time from task release to its completion, with the assumption of constrained deadlines, where $D_i \leq T_i$. The nominal priority π_i of each task defines its execution order relative to other tasks on the same core, with higher-priority tasks preempting lower-priority ones. A higher value of π_i corresponds to a higher priority level. Task priorities remain fixed throughout execution. When preemption threshold scheduling is enabled, each task is assigned a preemption threshold θ_i, which specifies that only tasks with a priority higher than θ_i can preempt τ_i.

4.4.2 Data Communication Model

Tasks communicate by reading from and writing to shared memory, enabling efficient data exchange across the system. We adopt the widely used single-writer, multiple-reader data communication model in real-time embedded systems, where a single task writes data while multiple tasks may read it concurrently [23, 90]. In this model, each data item communicated between tasks is referred to as a *label*. Each label L_i has a specific label size S_{L_i}, typically a few bytes, representing a small unit of shared information. For each label, there is exactly one writer task responsible for updating its value, while multiple reader tasks can access it to retrieve the most recent data.

The classification of a task as a writer or reader is label-specific; a task may serve as a writer for one label while acting as a reader for another label. For clarity, we use *writer* and *reader* to denote tasks that write to and read from a given label, respectively. The proposed data communication model supports the large-scale exchange of labels in real-time systems, often involving tens of thousands of labels. In such scenarios, a directed acyclic graph (DAG) [101] is unsuitable when data communication dependencies form cycles. Given the high volume of data exchange, ensuring efficient and timely data consistency is crucial to preventing delays that could violate the timing constraints of dependent tasks.

4.4.3 Response Time Analysis

In real-time systems, the response time of a task is defined as the time interval between the task's release (or arrival) and its completion [6, 18]. More specifically, the response time includes the time taken for a task to execute, along with any delays caused by other tasks, such as preemption or blocking, during its execution. A key objective in real-time systems is to ensure that tasks complete their execution within their respective deadlines, which

requires an accurate analysis of the worst-case response time for each task in the system.

In a partitioned system, each core schedules its assigned tasks independently based on task priority levels. The response time analysis is performed for each core separately, considering the tasks allocated to that core and the interference between them. In this setup, we apply the traditional response time analysis (RTA) technique used for fixed-priority preemptive scheduling. The response time R_i for task τ_i can be calculated using the following equation (adapted from [6]):

$$R_i = C_i + \sum_{\forall \tau_j \in hp(i, core_i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{4.1}$$

where $hp(i, core_i)$ represents the set of higher-priority tasks allocated to the same core $core_i$ as τ_i, i.e., task τ_j with $\pi_j > \pi_i$ & $core_j = core_i$. The response time is iteratively calculated until it converges to a fixed value, where the task's response time does not change in subsequent iterations. The first term C_i represents the execution time of the task itself, and the second term accounts for the interference caused by the higher-priority tasks within the same core.

4.5 Wait-Free Data Communication

4.5.1 Overview of Waif-Free Data Communication

In real-time systems, wait-free protocols ensure that tasks can access shared data without blocking by allowing each task, whether a reader or writer, to access data independently. Wait-free mechanisms achieve this non-blocking access by using multiple buffers for each data label, where each buffer is typically sized to match the data label size. This ensures that readers can always access the most recent data without waiting for other tasks, especially

the writer task, to complete their operations, thus preventing delays due to lock contention.

In wait-free data communication, multiple buffers are allocated to a single data label. The writer task updates data by writing to one of these buffers while readers access the latest available buffer without interfering with the writer's operation. Fig. 4.1 illustrates this concept, showing the timelines of three tasks running on two cores. Task 1 generates new data in each job, while Task 2 and Task 3 read the latest updated data from Task 1. Task 1 has a higher priority than Task 2, and Task 3 runs on a separate core so that it does not interfere with the other two tasks under partitioned scheduling. For simplicity and clarity, the example only considers a single data label being communicated among tasks, with multiple buffers allocated for its data exchange. The writer (Task 1) always writes to an available buffer that is not currently used by any reader. A reader (Task 2 or Task 3) accesses the latest updated buffer in a non-blocking manner. This multi-buffer mechanism ensures wait-free operations while preserving data consistency. As illustrated in Fig. 4.1, Task 2 begins execution at time t_6 but is preempted by Task 1 at time t_7. Even though Task 2 is still reading Buffer 2, Task 1 can safely update new data in a separate buffer. This wait-free mechanism also extends to multicore scenarios. In the example, Task 3 reads the latest updated buffer at the start of execution and can safely continue referencing it until completion. Despite the absence of locking mechanisms, data consistency is maintained as long as there are sufficient buffers for the writer to store new data. A well-designed wait-free protocol aims to minimize buffer usage while ensuring that all tasks can access data without blocking.

Assumptions

Certain assumptions about system operation and communication requirements are typically made in designing wait-free data communication protocols. These assumptions are necessary

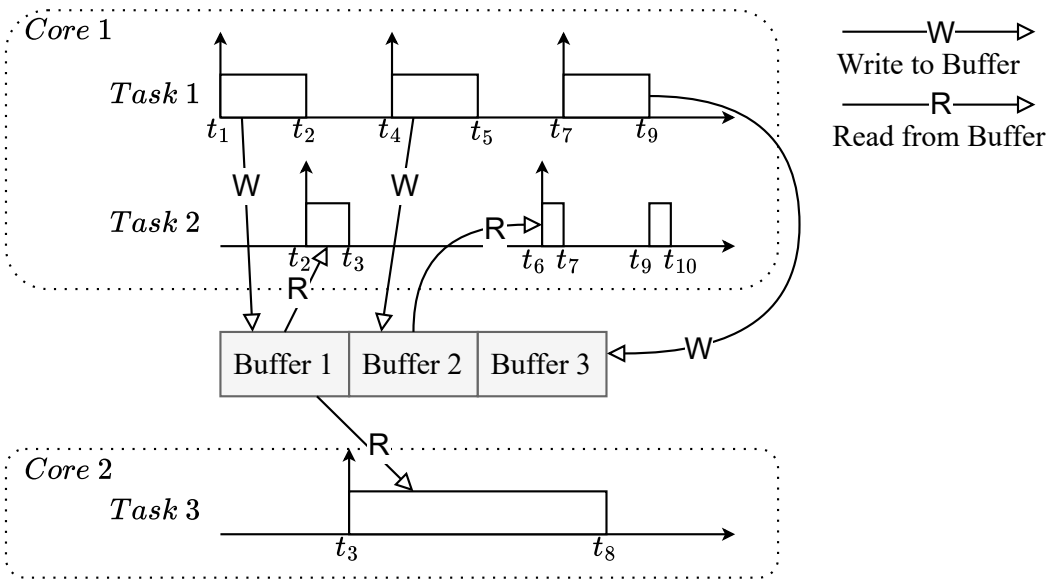


Figure 4.1: Wait-Free data communication diagram for a single data label: Task 1 generates new data in each job instance, while Task 2 and Task 3 read the most recently updated data from Task 1. As described in Section 4.5.1, data communication can occur at any point during task execution. The multi-buffer mechanism ensures wait-free operations while preserving data consistency.

to ensure that the protocols guarantee non-blocking access to data without compromising data consistency or real-time requirements:

- Data communications (reads and writes) may occur at any point during task execution. This flexibility ensures that reader and writer tasks are free to operate on shared data when needed, without being restricted to specific time slots or critical sections.

- The system often supports a large number of data communication labels, frequently exceeding 2,000 labels [57]. These labels can be managed and processed independently. In cases where consecutive communication of multiple labels is required, they can be integrated into a single label with a larger buffer size, thereby simplifying the protocol's operation.

- Wait-free protocols do not require explicit designation of critical sections, meaning

that tasks do not have to request locks or wait to operate on shared data. Instead, the protocol itself ensures data consistency, relieving software developers from having to manage data access manually within the task code.

Evaluation for Wait-Free Protocols

The efficiency of wait-free protocols is evaluated by measuring the total memory footprint consumed by data communication buffers across the system. In a wait-free protocol, each data label is managed independently, with its own dedicated implementation of buffers tailored to its specific label characteristics. For each label, the protocol aims to minimize the number of required buffers while guaranteeing wait-free, consistent data access for all readers. The memory cost for a single label is computed as the product of the number of allocated buffers and the label's data size. The overall system memory footprint is then obtained by summing the memory requirements across all data labels (Eq. 4.2, where NB_{L_i} represents the number of buffers needed for label L_i). Thus, although system-level evaluation focuses on total memory usage, protocol design and optimization typically concentrate on minimizing the buffer count for each individual label to achieve an efficient and scalable solution.

$$Memory \quad Overhead = \sum_{\forall L_i} (NB_{L_i} * size\_of(L_i)) \tag{4.2}$$

Since the memory overhead can be analyzed independently for each label, we use p^w to denote the index of the writer task in the corresponding context, without additional explanation unless otherwise noted. Accordingly, the parameters of the writer task τ_{p^w} are denoted as $(T_{p^w}, C_{p^w}, D_{p^w}, \pi_{p^w}, \theta_{p^w}, core_{p^w})$, representing its period, worst-case execution time, relative deadline, priority, preemption threshold, and assigned core, respectively.

Single-Core Wait-Free Protocols

Different wait-free protocols use various strategies to manage these buffers, aiming to prevent data inconsistency, where a writer overwrites a buffer that is still being read. These buffer management strategies lead to differences in memory efficiency.

The Temporal Concurrent Control Protocol (TCCP) [23] uses a circular buffer mechanism to manage sequential data updates. The total buffer size required by TCCP depends on the maximum data lifetime for each label, where the data lifetime represents the period during which a buffer remains valid for any reader task. The lifetime l_{L_i} of a label L_i is defined as:

$$l_{L_i} = reading\_offset + \max_{\tau_j \text{ reads } L_i} R_j \tag{4.3}$$

where *reading_offset* is the interval between the completion of the writer task and the start of the reader task's execution. In the worst-case scenario, this offset cannot exceed the writer task's period T_{p^w}. The response time R_j of each task τ_j that reads label L_i is analyzed following the discussion in Section 4.4.3.

The required number of buffers for a label L_i under TCCP is determined by:

$$NB_{L_i}^{\text{TCCP}} = \left\lceil \frac{l_{L_i}}{T_{p^w}} \right\rceil = 1 + \max_{\tau_j \text{ reads } L_i} \left\lceil \frac{R_j}{T_{p^w}} \right\rceil, \tag{4.4}$$

This approach ensures that even the longest-lived data remains valid for all reader tasks.

The Dynamic Buffering Protocol (DBP) [90] uses the worst-case number of concurrent reader tasks that can access the same data within one writer task execution to decide the total number of required buffers. For a given label L_i, let τ^w represent the writer task. DBP distinguishes between high-priority and low-priority reader tasks by comparing the priority

with τ_{p^w}. The total number of buffers required for DBP is given by:

$$NB_{L_i}^{\text{DBP}} = 1 + 1 + |\{\tau_j | \pi_j < \pi_{p^w} \ \& \ \tau_j \text{ reads } L_i\}| \tag{4.5}$$

Here, the first term '1' represents the buffer used by the writer task. The second term '1' accounts for the buffer for any high-priority reader task that might preempt the writer task. The last term counts the number of low-priority reader tasks, where $|\cdot|$ is the size of a set. This ensures that each low-priority task has its own buffer to prevent contention with the writer task. It is important to note that a single task can be classified as a low-priority reader task for one label and a high-priority reader task for another label, depending on the priority level of the writer task τ_{p^w}.

Combined DBP and TCCP (CDT): Wang et al. [98] proposed an enhanced wait-free protocol, CDT, that combines the strengths of both DBP and TCCP in a single-core setting. The rationale behind this approach lies in the fact that the memory footprint of wait-free protocols depends on both the response time and the priority assignment of tasks within the system. In TCCP, reader tasks can share buffers with others, and fewer buffers are required when reader tasks have shorter response times. However, prolonged data lifetimes can lead to unbounded memory usage. In contrast, DBP capitalizes on the characteristic that each reader task requires at most one buffer region at a time, assuming constrained deadline scenarios. This constraint limits the worst-case buffer requirement based on the number of reader tasks associated with a label, ensuring a more predictable memory footprint. CDT employs an exhaustive search to determine an optimal splitting pattern, allowing the combined protocol to take advantage of both DBP and TCCP.

4.5.2 Multicore Extensions for Existing Wait-Free Protocols

Partitioned Temporal Concurrent Control Protocol (PTCCP)

While TCCP's straightforward design is advantageous, it demands substantial memory, as it requires buffers to be allocated based on the longest data lifetime among tasks. Though originally developed for single-core systems, TCCP can be directly applied to multicore environments without modifications to its protocol or analysis, provided accurate worst-case lifetime assessments for each label are available.

Partitioned Dynamic Buffering Protocol (PDBP)

DBP achieves memory efficiency by limiting buffer usage according to task priority. However, it is inherently limited to single-core systems due to its reliance on priority comparison. For the extension to multicore systems, a worst-case scenario must be considered: all reader tasks allocated to a different core from the writer task τ_{p^w} may need to read the label L_i during a single writer task execution. Therefore, the total number of buffers required for multicore PDBP is:

$$NB_{L_i}^{\text{PDBP}} = 1 + 1 + |\{\tau_j | core_j = core_{p^w} \ \& \ \pi_j < \pi_{p^w} \ \& \tau_j \text{ reads } L_i\}| + |\{\tau_j | core_j != core_{p^w}\}| \tag{4.6}$$

In essence, PDBP distinguishes between the two subsets of reader tasks. Specifically, for all remote reader tasks, individual buffers are allocated to each task to avoid potential contention, requiring a number of buffers equal to the count of remote reader tasks. On the other hand, for local reader tasks, PDBP follows the same approach as DBP. This entails having one buffer designated for ongoing writing, another buffer for all high-priority local reader tasks, and a separate buffer for each low-priority local reader task.

4.5.3 Partitioned Combined-DBP-TCCP Protocol

Following the rationale of leveraging both the task response time and the priority assignment in the design of wait-free protocols under CDT, we introduce the Partitioned Combined-DBP-TCCP Protocol (PCDT) for multicore wait-free data communication. PCDT strategically integrates the advantages of PDBP and PTCCP by partitioning reader tasks into two distinct sets, with each set utilizing a specific protocol based on its execution characteristics. This method ensures that the memory footprint does not exceed that when either a single PTCCP or a single PDBP protocol is applied.

PCDT introduces several key optimizations that enhance its practicality in real-world embedded systems, beyond merely extending the approach from Wang et al. [98] to a multicore setting. Notably, under limited preemption scheduling, where tasks are only preempted at specific points to reduce context-switch overhead, PCDT can intrinsically achieve greater memory savings by leveraging the reduced variability in task execution. This characteristic enables more efficient buffer usage, making PCDT particularly beneficial for memory-constrained real-time systems. A more detailed discussion of these optimizations is provided in Section 4.6.1.

Once the fixed priorities of all tasks are determined, the PCDT protocol begins by analyzing the response time of all reader tasks and reordering them in a non-decreasing worst-case response time sequence. Tasks with shorter response times can share buffers, as allowed by TCCP, thereby reducing buffer usage. For tasks with longer response times, the number of buffers is limited according to the number of such reader tasks. PCDT then iterates through the ordered reader tasks to identify the optimal splitting point that maximizes buffer efficiency.

For a single label L_i and the corresponding writer task τ_{p^w}, suppose all the reader tasks are

indexed according to the non-decreasing response time order and the set of ordered tasks is $\{\tau_{p_0^r}, \tau_{p_1^r} ... \tau_{p_{n-1}^r}\}$. In this case, the number of readers of label L_i is denoted $NR_{L_i} = n$. The response time of a task $\tau_{p_i^r}$ is denoted as $R_{p_i^r}$, so we have $R_{p_0^r} \leq R_{p_1^r} \leq ... \leq R_{p_{n-1}^r}$. The total number of buffers needed for L_i in PCDT can be found by minimizing the Eq. (4.7).

$$NB_{L_i}^{\text{PCDT}} = \min\left(1 + NR_{L_i}, \min_{0 \leq j < NR_{L_i}} \left(\left(1 + \left\lceil \frac{R_{p_j^r}}{T_{p^w}} \right\rceil\right) + NB_{L_i}^{\text{PDBP}}(\{\tau_{p_{j+1}^r} ... \tau_{p_{n-1}^r}\})\right)\right) \quad (4.7)$$

As discussed earlier, the maximum number of buffers required in a constrained deadline system is bounded by the number of reader tasks, with an additional buffer allocated for ongoing writing. In the second minimization operator of Eq. (4.7), the first component $\left(1 + \left\lceil R_{p_j^r}/T_{p^w} \right\rceil\right)$ represents the number of buffers needed for shared buffer. Given that tasks are ordered in non-decreasing response time, the buffers required by $\tau_{p_j^r}$ can be safely shared among $\{\tau_{p_0^r}, \tau_{p_1^r}, ..., \tau_{p_j^r}\}$ to maintain data consistency. Consequently, the term $NB_{L_i}^{\text{PDBP}}(\{\tau_{p_{j+1}^r} ... \tau_{p_{n-1}^r}\})$ accounts for the buffer analysis of the remaining $(NR_{L_i} - 1 - j)$ tasks, following the PDBP analysis in Eq. (4.6), where the worst-case number of buffers is bounded by $(NR_{L_i} - 1 - j)$.

PCDT guarantees data consistency through a combination of rigorous buffer allocation strategies and optimal reader partitioning. By ensuring that each reader task accesses valid, up-to-date data during its execution, the protocol preserves data integrity while maintaining an efficient memory footprint. These qualities make PCDT exceptionally well-suited for multicore real-time embedded systems, where predictable memory usage and data consistency are essential requirements.

Theorem 4.1. *The data consistency for single-writer, multiple-reader communication in constrained-deadline multicore real-time systems is guaranteed under the Partitioned Combined-*

DBP-TCCP Protocol.

Proof. We prove this by contradiction. Assume, for the sake of contradiction, that the data consistency is violated under the proposed PCDT protocol. Consider a single-writer, multiple-reader communication model in constrained-deadline systems, where the relative deadline is no larger than the period. In this setup, it is not possible for a single label to be written by two instances of the writer task simultaneously, as the single-writer constraint ensures exclusivity. Data consistency cannot be violated if contention occurs only among reader tasks, as they perform read-only operations and do not modify the shared data.

For data consistency to be violated, there must exist a time instance where a buffer unit is simultaneously accessed by the writer task and at least one reader task. Without loss of generality, let τ_{p^w} denote the writer task and τ_{p^r} represent one of the reader tasks involved in this contention.

The PCDT classifies each reader task into one of two protocols, PTCCP or PDBP, based on the optimal solution of Eq. (4.7), which determines the appropriate protocol assignment considering task response time and access constraints. We analyze both cases separately:

Case 1: Reader Task τ_{p^r} Uses PTCCP

Under PTCCP, tasks are not distinguished by their assigned cores, and the number of buffer units is assigned solely based on the lifetime of the task possessing the maximum value. If data inconsistency arises under PTCCP, τ_{p^r} must read partially updated data during an ongoing write operation, which means there is no buffer unit available (a unit is available when no reader task is accessing it) for the writer task to store the new data. However, the maximum number of new data that can be generated by τ_{p^w} during the lifetime of one instance of τ_{p^r} (from the release time to the completion time) is bounded by $1 + \lceil R_{p^r}/T_{p^w} \rceil$, where R_{p^r} is the response time of τ_{p^r} and T_{p^w} is the period of τ_{p^w}. Thus, no inconsistent state

can occur as a sufficient number of buffer units are allocated in this scenario, contradicting the assumption.

Case 2: Reader Task τ_{p^r} Uses PDBP

Under PDBP, the buffer unit is allocated based on whether the assigned core of the reader task ($core_{p^r}$) is the same as the writer task's assigned core ($core_{p^w}$). Specifically:

- When $core_{p^w} = core_{p^r}$, we consider the priorities of the tasks. If the reader task has a higher priority π_{p^r} than the writer task's priority π_{p^w} (note that we assume distinct priority levels across all tasks), it shares a specifically allocated buffer unit with all high-priority (with respect to π_{p^w}) reader tasks. It is impossible for τ_{p^w} to resume execution before all high-priority reader tasks finish accessing the buffer, thus, data inconsistency cannot occur. Conversely, if τ_{p^r} has a lower priority than τ_{p^w}, it is assigned a dedicated buffer unit exclusively to itself. Since an additional buffer unit is always reserved for write operations under PDBP, data inconsistency cannot occur in this scenario.

- When τ_{p^r} and τ_{p^w} are assigned to different cores, where $core_{p^w} \neq core_{p^r}$, PDBP will allocate a specific buffer unit exclusively to τ_{p^r}. Since the writer task τ_{p^w} also has a designated buffer unit, data inconsistency will not occur as there is sufficient buffer, and no contention arises.

Conclusion

In both cases, the sub-protocols (PTCCP and PDBP) inherently prevent overlapping between write operations and reader accesses that could result in data inconsistency, the initial assumption that the proposed PCDT protocol violates data consistency is invalid. Therefore, the PCDT protocol guarantees data consistency for single-writer, multiple-reader communi-

cation in constrained-deadline multicore real-time systems.

□

4.6 Optimizing Memory Footprint For PCDT

The memory footprint of the proposed Partitioned Combined-DBP-TCCP Protocol (PCDT) can be further optimized through improved task priority assignment and partitioning strategies. While PCDT already integrates buffer-sharing mechanisms from PDBP and PTCCP, adopting suitable scheduling policies can further reduce the memory footprint by minimizing the number of required buffers while maintaining data consistency.

This section explores two key techniques to optimize memory usage in multicore wait-free data communication systems: a) Priority assignment with preemption thresholds – limiting preemption behavior to reduce buffer consumption; b) Task partition with wait-free protocol insights – partitioning tasks across cores to improve buffer usage efficiency.

4.6.1 Preemption Thresholds with PCDT

In traditional fixed-priority preemptive scheduling, high-priority tasks can preempt lower-priority tasks at any time. This leads to increased buffer usage, as multiple tasks may simultaneously hold references to different buffers. Moreover, lower-priority tasks can experience longer worst-case response times (WCRT), which in turn extends the lifetime of a label being occupied, increasing the overall memory footprint. To mitigate this, we adopt priority assignment with Preemption Threshold (PT), a technique introduced by Wang et al. [103]. This approach allows tasks to delay preemption by higher-priority tasks up to a specified priority threshold, reducing excessive interruptions while preserving system responsiveness.

Preemption threshold scheduling is a hybrid approach where a task τ_i with priority π_i can only be preempted by tasks with priorities greater than its preemption threshold θ_i, where $\theta_i \geq \pi_i$. Formally, a task τ_j can preempt τ_i only if $\pi_j > \theta_i$. By limiting preemptions, PT reduces contention in shared-memory systems, improves predictability, and optimizes WCRT. In the context of wait-free data communication, these properties allow tasks to complete execution while holding fewer buffers, thereby reducing buffer allocation.

Integration with PCDT

In PCDT, the required buffer size NB_{L_i} for a label L_i is determined by the worst-case response times of consumer tasks (Eq.4.7). By incorporating PT, we can reduce task WCRT, leading to a smaller memory footprint.

PT enables the formation of mutually non-preemptive task pairs and non-preemptive groups, reducing buffer requirements:

Definition 4.2 (Mutually Non-Preemptive)**.** Two tasks τ_i, τ_j are mutually non-preemptive if $core_i = core_j$ & $\pi_i \leq \theta_j$ & $\pi_j \leq \theta_i$.

Since mutually non-preemptive tasks cannot preempt each other, their executions are mutually exclusive, preventing race conditions in data communication.

Definition 4.3 (Non-Preemptive Group)**.** A non-preemptive group consists of a set of tasks $\{\tau_1, \tau_2, ...\tau_k\}$, in which every pair of tasks is mutually non-preemptive.

With preemption threshold scheduling, the buffer demand of PDBP can be further optimized. For readers on the same core as the writer τ_w, no extra buffers are needed for low-priority readers that are mutually non-preemptive with the writer. For readers on different cores, buffer requirements are reduced to the number of non-preemptive groups among

these readers. The buffer requirement for PDBP with PT is given by:

$$NB_{L_i}^{\text{PDBP\_PT}} = 1 + 1 + |\{\tau_j | core_j = core_{p^w} \ \& \ \theta_j < \pi_{p^w} \ \& \ \tau_j \text{ reads } L_i\}| + num\_np\_group(\{\tau_j | core_j ! = core_{p^w}\}) \quad (4.8)$$

where the $num\_np\_group()$ denotes the number of non-preemptive groups in a given set of tasks. A single task can also form its own non-preemptive group. The function's output is upper-bounded by the total number of tasks in the set.

Since finding the minimal number of non-preemptive groups is NP-hard (because it can be reduced from the minimum clique covering problem [53]), we employ a greedy approximation algorithm. Firstly, a random task is selected as the seed for a non-preemptive group. Then, expand the group by adding mutually non-preemptive tasks until no further additions are possible. Repeat until all tasks are assigned to a group. Other heuristic methods can also be explored to refine group formation.

By organizing tasks into non-preemptive groups and allowing them to execute without unnecessary interruptions, we further reduce the worst-case memory demand of PCDT. The buffer bound for PCDT is revised to incorporate the preemption threshold, ensuring that the worst-case buffer allocation remains minimized while preserving a wait-free execution guarantee:

$$NB_{L_i}^{\text{PCDT\_PT}} = \min\left(1 + NR_{L_i}, \min_{0 \le j < NR_{L_i}} \left(\left(1 + \left\lceil \frac{R_{p_j^r}}{T_{p^w}} \right\rceil\right) + NB_{L_i}^{\text{PDBP\_PT}}(\{\tau_{p_{j+1}^r} ... \tau_{p_{n-1}^r}\})\right)\right) \quad (4.9)$$

Response Time Analysis With Preemption Threshold

The response time analysis under PT differs from the original formulation in Eq. 4.1. Regehr [85] extended the response time analysis for PT. For partitioned scheduling, we only consider tasks on the same core. A task τ_i may experience one blocking time B_i from low-priority tasks:

$$B_i = \max_j \{C_j | \theta_j \geq \pi_i > \pi_j\} \tag{4.10}$$

Then the response time analysis is carried out within the longest Level-i active period L_i, which is the fix-point solution of this:

$$L_i = B_i + \sum_{j:\pi_j \geq \pi_i} \left\lceil \frac{L_i}{T_j} \right\rceil C_j \tag{4.11}$$

Within this Level-i busy period, the number of job instances K_i of task τ_i is:

$$K_i = \left\lceil \frac{L_i}{T_i} \right\rceil \tag{4.12}$$

The start time and finish time of the k^{th} job instance of task τ_i are calculated iteratively:

$$s_{i,k} = B_i + (k-1)C_i + \sum_{j:\pi_j > \pi_i} \left(\left\lfloor \frac{s_{i,k}}{T_j} \right\rfloor + 1 \right) C_j \tag{4.13}$$

$$f_{i,k} = s_{i,k} + C_i + \sum_{j:\pi_j > \theta_i} \left(\left\lceil \frac{f_{i,k}}{T_j} \right\rceil - \left(\left\lfloor \frac{s_{i,k}}{T_j} \right\rfloor + 1 \right) \right) C_j \tag{4.14}$$

Thus, the final response time is the maximum among the K_i job instances:

$$R_i = \max_{k \in [1, K_i]} (f_{i,k} - (k-1)T_i) \tag{4.15}$$

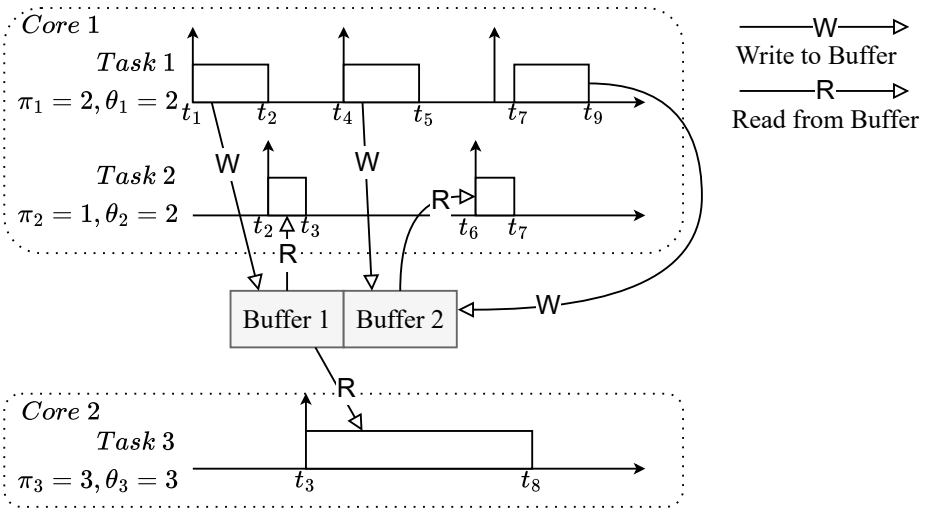


Figure 4.2: Wait-Free data communication with Preemption Threshold: Task 2 is assigned a preemption threshold of 2, making it mutually non-preemptive with Task 1. This configuration reduces unnecessary preemptions, allowing the total buffer requirement for the wait-free protocol to be minimized from 3 to 2 with appropriate preemption threshold settings.

By leveraging preemption thresholds, PCDT achieves significant memory savings while maintaining efficient wait-free execution. The introduction of non-preemptive groups further tightens buffer requirements, making PT a powerful enhancement for memory optimization in real-time multicore systems.

4.6.2 Priority Assignment with Preemption Threshold

Optimizing the preemption threshold configuration requires a careful balance between task priority levels and preemption threshold settings to minimize unnecessary preemptions while promoting efficient buffer sharing. An effective priority assignment can ensure that tasks benefiting the most from preemption thresholds are appropriately prioritized, resulting in improved memory efficiency for wait-free data communication.

This subsection first introduces the maximum preemption threshold assignment algorithm,

which assigns the highest feasible preemption thresholds to tasks under a given priority order while preserving schedulability. To further enhance system performance, we introduce a priority assignment algorithm that incorporates maximum blocking time tolerance, aiming to determine a priority order that enhances the effectiveness of preemption thresholds by considering the ability of each task to tolerate blocking.

Maximum Preemption Threshold Assignment

One key strategy to enhance preemption threshold utilization is to assign the highest possible preemption threshold to tasks that frequently read shared data. Since data communication is highly intensive in real-time embedded systems, where a typical system may have thousands of labels compared to only a few dozen tasks, assigning the maximum preemption thresholds to the task set can prevent excessive buffer usage while maintaining system schedulability.

To achieve this, we employ the maximum preemption threshold assignment algorithm from [103], which iteratively assigns the highest permissible preemption threshold to each task while ensuring system schedulability. Starting from the highest-priority task, each task is assigned the maximum preemption threshold that does not violate schedulability constraints. Given the significantly higher number of labels compared to tasks, this strategy effectively minimizes the worst-case memory footprint while preserving real-time guarantees.

Priority Assignment with Maximum Blocking Time Tolerance (PA-MBTT)

The effectiveness of preemption threshold assignment is closely linked to task priority assignment. A poorly chosen priority order can lead to overly conservative preemption threshold assignments, diminishing the potential benefits of this approach. To overcome this limitation, we adopt the PA-MBTT priority assignment algorithm proposed by Zeng et al. [114, 115],

which prioritizes tasks based on their ability to tolerate blocking.

In PA-MBTT, task priorities are assigned incrementally from the lowest level upward, selecting at each step the task with the highest blocking time limit. The blocking time limit of a task τ_i, denoted as h_i, represents the maximum blocking duration τ_i can endure while still meeting its deadline. Since the exact priority order of higher-priority tasks is unknown during the assignment, a deadline monotonic (DM) order is used as an initial approximation. The algorithm from Wang and Saksena [103] is then used to determine the maximum allowable preemption threshold for each task.

The complete procedure of PA-MBTT is summarized in Algorithm 6, where the task blocking time limit is computed through a binary search until the desired precision is achieved. By prioritizing tasks based on their blocking time tolerance, PA-MBTT enables higher preemption threshold assignments while preserving schedulability. This approach further reduces unnecessary preemption and buffer requirements, enhancing the efficiency of wait-free data communication in real-time embedded systems.

4.6.3 Task Partitioning Algorithm with Wait-Free Insights

Task partitioning plays a critical role in multicore real-time systems, directly influencing schedulability and resource efficiency. Traditionally, task partitioning algorithms prioritize optimizing CPU utilization across cores, often ignoring the underlying communication patterns between tasks. One of the most widely used baseline approaches is Best Fit Decreasing (BFD), which sorts tasks in descending order of utilization and assigns each task to the core with the least remaining capacity that can still accommodate it. While BFD is simple and effective for load balancing, it does not consider resource synchronization and data communication memory overhead, making it suboptimal in systems with intensive inter-task

Algorithm 6 Priority Assignment with Maximum Blocking Time Tolerance (PA-MBTT)

Input: Task set $\mathcal{T}$ **Output:** Priority Assignment.

```
1: unassigned_task ← T
2: for priority_level p = 1 to |T| do
3:     for task τ_i in unassigned_task do
4:         assume p_i = p
5:         assume DM priority for unassigned_task\{τ_i}
6:         get blocking time limit h_i for τ_i
7:         if R_i ≤ D_i then
8:             a_i = j_i
9:         else
10:            a_i = D_i − R_i
11:        end if
12:    end for
13:    select τ_i with the largest a_i and assign p_i = p
14:    unassigned_task = unassigned_task\{τ_i}
15: end for
```

communication.

Other existing methods, SPA [58], BPA [75] and ROP [109], fall short when applied to systems with large-scale, label-based data communication, a fundamental characteristic of wait-free protocols, because they are primarily designed to manage a small number of critical resources. In contrast, modern embedded systems feature a large number of independent data labels, each of which must be individually protected by synchronization protocols. To address this limitation, we propose a novel task partitioning approach called the Wait-Free-Aware Partitioning (WFAP) algorithm. WFAP is specifically designed to handle the unique properties of wait-free data sharing, enabling efficient memory usage while maintaining system schedulability.

WFAP builds upon the BFD framework for its core utilization efficiency but introduces a task-affinity heuristic derived from PDBP insights. The key idea is to estimate the potential memory savings when two tasks are allocated to the same core, which reduces the number

of required communication buffers. While estimating memory savings under the full PCDT protocol is challenging due to the complex interactions among task priorities and preemption thresholds, the PDBP component allows a more tractable analysis. By examining whether communicating tasks share the same core, we can approximate the memory cost. This memory-aware affinity guides the partitioning decisions, favoring task-core assignments that maximize buffer reuse and improve overall memory efficiency.

Best Fit Decreasing (BFD) Task Partitioning

In the BFD approach, tasks are sorted by decreasing utilization. Each task is placed on the core where it "fits best", that is the core with the least remaining capacity that can still schedule the task without violating the utilization bound. If no core has enough remaining capacity, it will request a new core for task partitioning. BFD ensures efficient load distribution but lacks awareness of inter-task communication.

Wait-Free-Aware Partitioning (WFAP)

WFAP augments BFD by introducing a task affinity metric that quantifies the communication-related memory benefit of assigning two tasks to the same core. Specifically, for the PDBP, high-priority readers can share the same buffer for a label, which significantly reduces buffer usage. To leverage this, WFAP computes the task affinity $TA(\tau_i, \tau_j)$ between tasks τ_i and τ_j based on the potential buffer reduction if both tasks are allocated to the same core. The affinity metric assumes the use of the PDBP protocol and a Deadline Monotonic (DM) priority assignment. Under DM, if $D_i < D_j$, then τ_i has a higher priority than τ_j, (i.e. $\pi_i > \pi_j$). In this case, $TA(\tau_i, \tau_j)$ is defined as the number of labels written by τ_j and read by τ_i. Conversely, if $D_i > D_j$, the $TA(\tau_i, \tau_j)$ is defined as the number of labels written by τ_i and read

by τ_j.

This task affinity is used in task partitioning: among all feasible cores, WFAP selects the one with the highest affinity score relative to the task being allocated. The detailed procedure is presented in Algorithm 7. The task set is first sorted in a non-increasing order of task density, where the density of τ_i is defined as C_i/D_i. For each task in that order, the Core Affinity (CA) is computed. If a core already has tasks assigned, the CA is the average task affinity between τ_i and all tasks on that core (line 5 in Algorithm 7). If a core is currently empty, its CA is set to an infinitely large value (line 7 in Algorithm 7) to promote establishing strong anchor tasks early in the partitioning. This design is particularly effective in scenarios with a large number of labels, helping to guide more communication-efficient task clustering. Finally, τ_i is assigned to the core with the highest Core Affinity.

Algorithm 7 Wait-Free-Aware Partitioning (WFAP)

Input: Task set $\mathcal{T}$, total number of cores.
Output: Task core assignment.

1: sort $\mathcal{T}$ in non-increasing task density order
2: **for** task τ_i in $\mathcal{T}$ **do**
3: **for** $c = 1$ to total number of cores **do**
4: **if** exist tasks assigned to $core_c$ **then**
5: $CA_c = \text{Average}([TA(\tau_i, \tau_j) | core_j = core_c])$
6: **else**
7: $CA_c = \inf$
8: **end if**
9: **end for**
10: $core_i \leftarrow \arg\max_c(CA_c)$
11: **end for**

WFAP has a polynomial time complexity of $O(n^2m)$, where n is the total number of tasks and m is the number of processor cores. By incorporating communication-aware heuristics into the partitioning process, WFAP significantly improves memory efficiency while maintaining real-time guarantees. This makes it particularly well-suited for systems with large-scale

wait-free data communication.

4.6.4 Memory Footprint Optimization via Local Search

While the Wait-Free-Aware Partitioning (WFAP) algorithm offers an effective and scalable method for task allocation in systems with large-scale wait-free communication, it does not always guarantee schedulability, particularly under tight timing constraints. In some cases, the partitioning produced by WFAP may lead to task sets that fail the schedulability analysis due to uneven load distribution or unfavorable preemption interactions.

To further enhance memory efficiency and schedulability, we introduce a local search-based optimization technique that refines the initial WFAP solution through iterative improvements, which is referred to as WFAP+. The key idea is to explore neighboring task allocations by making small changes, either reassigning a single task to a different core (referred to as a 1-move) or swapping the cores of two tasks (2-move). These moves aim to discover alternative partitioning that reduces overall buffer usage while satisfying task set schedulability.

The local search procedure operates as follows:

- Initialization: Start with the task partitioning generated by WFAP.
- Evaluation: Compute the total buffer requirement for the current allocation.
- Neighbor Exploration: Perform a 1-move by reassigning a task to a different core and check whether it improves the memory footprint without violating schedulability. Perform a 2-move by swapping two tasks between their respective cores and re-evaluating the system.

- Acceptance: If a move results in a lower buffer requirement and maintains task schedulability, accept the new allocation as the current best and repeat the local search.

- Termination: The process terminates when no further improvement is found or when a predefined time budget or iteration limit is reached.

This approach enables flexible refinement of task partitioning and adapts well to systems where a large time budget is available for offline optimization. Since the memory usage in wait-free systems is highly sensitive to task placement due to buffer-sharing opportunities, even minor changes can lead to significant improvements in overall resource utilization. Experimental results demonstrate that local search can substantially enhance the performance of WFAP, especially in dense systems with a large number of tasks and shared labels.

4.7 Deployment of PCDT

The design and deployment of wait-free protocols involve two primary components: integrating the software application code with specific APIs and adjusting the operating system (OS) to provide the necessary support. Modifications to the software typically occur within the application code, while OS changes may take place within the middleware runtime (e.g., AUTOSAR [36]) or the kernel/module of a Real-Time Operating System (RTOS), such as ThreadX [31].

The following assumptions are made about the system before it starts: 1) The total number of data communication labels and their unique indices are known. 2) For each label, the data size, writer task, and reader tasks are predetermined.

4.7.1 Software Application API

The modifications to software applications are minimal. Instead of directly accessing the shared data (memory), wrapper functions will be used to read and write to different labels. The applications do not need to differentiate between specific buffers for reading or writing. The typical read and write operations are outlined in Algorithm 8.

Algorithm 8 Software Application Routines

```
...
// Reading operations
reading_data_pointer ← readFromLabel(label_id, task_id);
Read data from *reading_data_pointer
...
Non-communication-related calculations
...
// Writing operations
Prepare and populate writing data into *writing_data_pointer
writeToLabel(label_id, task_id, writing_data_pointer);
...
// Cleaning
readingComplete(label_id, task_id);      ▷ Could be invoked early if no more reading is needed
free writing_data_pointer;
```

4.7.2 Operating System Level Support

The Operating System (OS) is responsible for initializing, allocating, and managing multiple valid data buffers associated with a single data communication label. All the buffers needed for the PCDT wait-free protocols should be statically allocated. Upon receiving a new reading or writing request, the OS selects the appropriate buffer block to be utilized for the current operation and directs the tasks to proceed accordingly. Typically, each communication label is processed independently and occupies a preassigned memory region specific to the current label.

The following discussion will be presented from the perspective of a single label. Applying these implementations to the entire set of labels will suffice for the system as a whole. Different labels may request varying data sizes in memory, we use "buffer" to refer to the exact memory size required for one copy of the data of the communication label.

Atomic Get and Set

Although the lock mechanisms are avoided in the wait-free protocols, the atomic instructions remain essential for synchronization across multiple threads and cores. To facilitate this, we introduce two utility functions, Algorithm 9 and 10, for atomic get and set (update) operations using the Compare and Swap (CAS) instruction. CAS, also known as Compare and Set (CAS), is widely supported by hardware instruction sets. It will succeed if the first two input arguments are equivalent, in which case it updates the first argument with the value of the third argument.

Algorithm 9 atomicGet(data_type **p_target*)

Input: *p_target* the pointer to the target data **Output:** The actual data

```
1: while True do
2:     data ← *p_target
3:     if CAS(data, *p_target, *p_target) == success then
4:         break
5:     end if
6: end while
7: return data
```

PCDT Related APIs

Even though the theoretical analysis is complicated, the implementation of PCDT could be simplified. Equation (4.9) provides a tight yet still safe bound on the number of buffers

Algorithm 10 atomicSet(data_type **p_target*, data_type **p_data*)

Input: *p_target* the pointer to the target, *p_data* the pointer to the desired updated value **Output:** Status flag

```
1: while True do
2:     current ← *p_target
3:     if CAS(*p_target, current, *p_data) == success then
4:         break
5:     end if
6: end while
7: return 0
```

needed by analyzing the worst-case scenario. In practice, the allocated buffers are not necessarily to be explicitly categorized into shared buffers or specific buffers designated for a reader task. Instead, the buffer region will be shared among all reader tasks. The pseudocodes of all APIs are provided in Algorithm 11 to Algorithm 14.

- Algorithm 11, initBuffer(): For each label L_i, calculate the maximum number of required buffers $NB_{L_i}^{\text{PCDT}}$. Then allocate a buffer array with a size $NB_{L_i}^{\text{PCDT}}$ for label L_i, where each entry will have a size equal to the data size of label L_i.
- Algorithm 12, writeToLabel(): Update the data of a specific label.
- Algorithm 13, readFromLabel(): Read the latest data value of a specific label.
- Algorithm 14 readingComplete(): Release the occupied buffer by a reader task.

The overhead of the wrapper functions is minimal, typically less than $O(NB_{L_i}^{\text{PCDT\_PT\_PT}})$. It is possible to further optimize by explicitly distinguishing between shared-buffer and specific-buffer tasks, which could reduce the overhead for processing shared-buffer tasks. However, this approach complicates the code structure, as the writer may need to write to two data

Algorithm 11 PCDT initBuffer()

Output: Status flag

1: **for** each label L_i **do**
2: Calculate $NB_{L_i}^{\text{PCDT\_PT}}$ following Eq. (4.9)
3: Get the number of readers NR_{L_i}
4: Declare $data[NB_{L_i}^{PCDT\_PT}]$, $latest\_idx$ and $readers[NR_{L_i}]$
5: Allocate memory and initialize $data[NB_{L_i}^{PCDT\_PT}]$ with initial value
6: $latest\_idx \leftarrow 0$
7: $readers[*] \leftarrow NB_{L_i}^{\text{PCDT\_PT}}$
8: **end for**
9: **return** 0

Algorithm 12 PCDT writeToLabel

Input: *label_id*, *task_id*, *writing_data_pointer*
Output: Status flag

1: Get $latest\_idx$, $NB_{L_i}^{\text{PCDT\_PT}}$ and NR_{L_i} for the current label *label_id*
2: Declare $in\_use[NB_{L_i}^{\text{PCDT\_PT}} + 1]$ and initialize all entries with False
3: $in\_use[latest\_idx] \leftarrow$ True
4: **for** i in $[0, NR_{L_i} - 1]$ **do**
5: $in\_use[atomicGet(\&readers[i])] =$ True
6: **end for**
7: $current\_idx \leftarrow -1$
8: **for** i in $[0, NB_{L_i}^{\text{PCDT\_PT}} - 1]$ **do**
9: **if** $in\_use[i]$ is False **then**
10: $current\_idx \leftarrow i$
11: **break**
12: **end if**
13: **end for**
14: **if** $current\_idx < 0$ **or** $current\_idx \geq NB_{L_i}^{\text{PCDT\_PT}}$ **then**
15: **return** -1 ▷ Protocol fails. Need to check the timing analysis
16: **end if**
17: $data[current\_idx] \leftarrow$ *$writing\_data\_pointer$ ▷ Writing data
18: atomicSet(*&latest_idx*, *&current_idx*);
19: **return** 0

Algorithm 13 PCDT readFromLabel

Input: *label_id*, *task_id* **Output:** Data pointer

1: *local_task_id* ← mapping(*task_id*) ▷ Map the *task_id* into the local index of *readers[]*
2: *reading_idx* = atomicGet(*&latest_idx*)
3: atomicSet(&*readers*[*local_task_id*], *&reading_idx*)
4: **return** *&data[reading_idx]*

Algorithm 14 PCDT readingComplete

Input: *label_id*, *task_id* **Output:** Status flag

1: *local_task_id* ← mapping(*task_id*) ▷ Map the *task_id* into the local index of *readers[]*
2: **return** atomicSet(&*readers*[*local_task_id*], $\&NB_{L_i}^{\text{PCDT\_PT}}$)

buffers for each update (one for shared-buffer tasks and another for specific-buffer tasks), which could increase overall system overhead.

PCDT also requires additional memory to store the indexes of readers' buffers. Nonetheless, this extra memory is minimal compared to the data memory, as basic types suffice for indexing. Additionally, techniques such as using bit-sized storage for necessary data can further reduce memory usage.

4.8 Evaluation and Results

To evaluate the effectiveness, scalability, and practical applicability of the proposed PCDT protocol and associated optimization techniques, we present both a real-world case study and a comprehensive set of simulation experiments. The case study demonstrates the protocol's deployment in a production Advanced Driver-Assistance System (ADAS), highlighting its memory efficiency in a realistic industrial setting. The simulation experiments provide a broader evaluation across synthetic system configurations, examining the comparative per-

formance of different wait-free communication protocols and optimization strategies under varying task and communication loads.

4.8.1 Case Study: Advanced Driver Assistant System (ADAS) Application

We analyzed a data communication pattern from a production Advanced Driver-Assistance System (ADAS) from General Motors. Proprietary names were removed, but the number of software components (SWCs), their periods, and the full label–access matrix were retained. The application executes on two cores (Core A and Core B) and comprises sixteen periodic tasks, eight per core after partitioning. Tasks contain between 2 and 88 SWCs. Communication is highly intensive: 2,130 distinct one-byte labels are exchanged, of which 2,084 are strictly intra-core, while only 46 are cross-core labels, reflecting the industrial preference to avoid inter-core communication. Each label has between 1 and 14 readers. All tasks follow fixed-priority preemptive scheduling with preemption thresholds, and implicit deadlines are assumed. The period and SWC count for each task are summarized in Table 4.1.

Table 4.1: Task Distribution and Software Component Count in Dual-Core ADAS Vehicle MCU System

Core A	**Period (ms)**	**SWC Count**	**Core B**	**Period (ms)**	**SWC Count**
τ_0	5	2	τ_8	5	14
τ_1	10	4	τ_9	10	11
τ_2	25	6	τ_{10}	25	13
τ_3	50	12	τ_{11}	50	34
τ_4	100	20	τ_{12}	100	58
τ_5	250	21	τ_{13}	250	88
τ_6	500	18	τ_{14}	500	48
τ_7	1000	11	τ_{15}	1000	28

Experimental Evaluation

To assess the memory behavior of the wait-free protocols under different computational loads, we scale a *unit WCET*: the execution time of each task is obtained by multiplying this unit by the number of SWCs it contains, so increasing the unit value raises global utilization while keeping task periods unchanged. We compare the three wait-free protocols available for partitioned systems: Partitioned Dynamic Buffering Protocol (PDBP), Partitioned Temporal Concurrent Control Protocol (PTCCP), and Partitioned Combined DBP and TCCP Protocol (PCDT). In addition, we evaluate the proposed Priority Assignment strategy with Maximum Blocking Time Tolerance (PA-MBTT) and the task partitioning algorithm WAFP+ in conjunction with the PCDT protocol. For each utilization level, we carry out response-time analysis (RTA) on each core and count the buffers required by all 2,130 labels, because each label is one byte, this number is directly proportional to SRAM usage.

Figure 4.3 illustrates the average memory footprint against the total system utilization. As expected, all three wait-free protocols allocate more buffers as utilization increases, yet their trends differ markedly. PTCCP shows the steepest growth, followed by PDBP. In contrast, PCDT remains lowest across the schedulable range and, at 90% utilization, requires about 12.5% less memory than PDBP. Even when utilization approaches the feasibility limit (116.5%), PCDT keeps memory usage well controlled, demonstrating that the PCDT protocol scales gracefully with load and is therefore well suited to resource-constrained real-time embedded platforms. When the Priority Assignment with Maximum Blocking Time Tolerance (PA-MBTT) strategy is applied along with the preemption threshold policy, memory usage is further reduced, achieving more than 43.43% savings compared to the PDBP protocol. Moreover, the Wait-Free-Aware Partitioning (WAFP+) not only improves memory efficiency but also extends the feasible utilization range. This indicates its potential to better

leverage limited computing resources and accommodate a larger number of tasks.

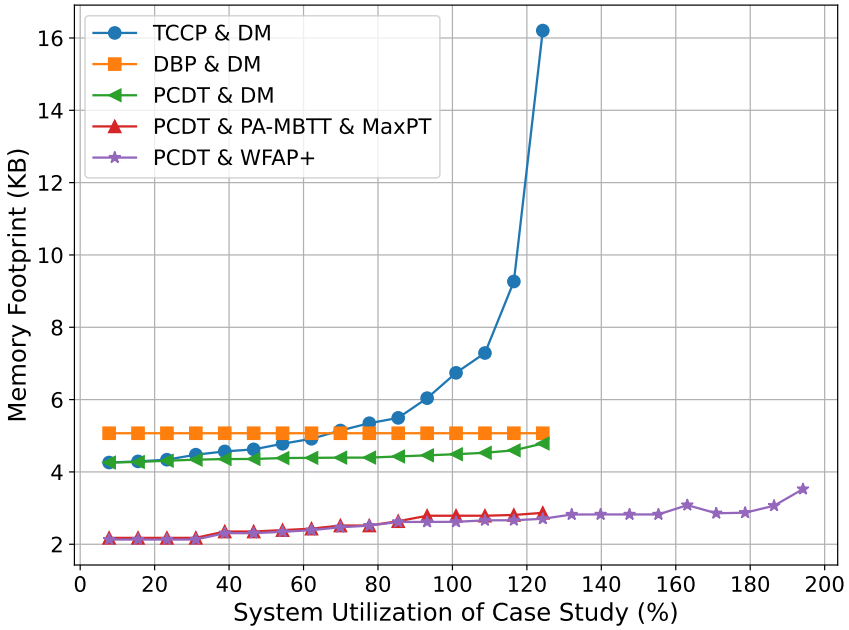


Figure 4.3: Average memory footprint under increasing system utilization in the case study. The system consists of two cores, with a theoretical maximum utilization of 200%. Some curves terminate when the system utilization reaches 116.5% as the system becomes unschedulable with original task partitioning, where Core B reaches 92.3% utilization. When the proposed Wait-Free-Aware Partitioning (WAFP+) algorithm is applied, the system's feasible utilization range is significantly extended while preserving a stable memory usage trend.

Prototype Evaluation on STM32H747I-DISCO

To verify that PCDT's benefits transfer from analysis to real hardware, we implemented the protocol on an STM32H747I-DISCO board running ThreadX [31]. The device features a Cortex-M7 and a Cortex-M4 in an asynchronous multi-processing configuration, each core executes its own fixed-priority schedule, and ThreadX natively provides preemption-threshold support. All inter-core communication is performed through PCDT wait-free buffers.

Memory Footprint: The PCDT runtime support was derived from the algorithm presented in

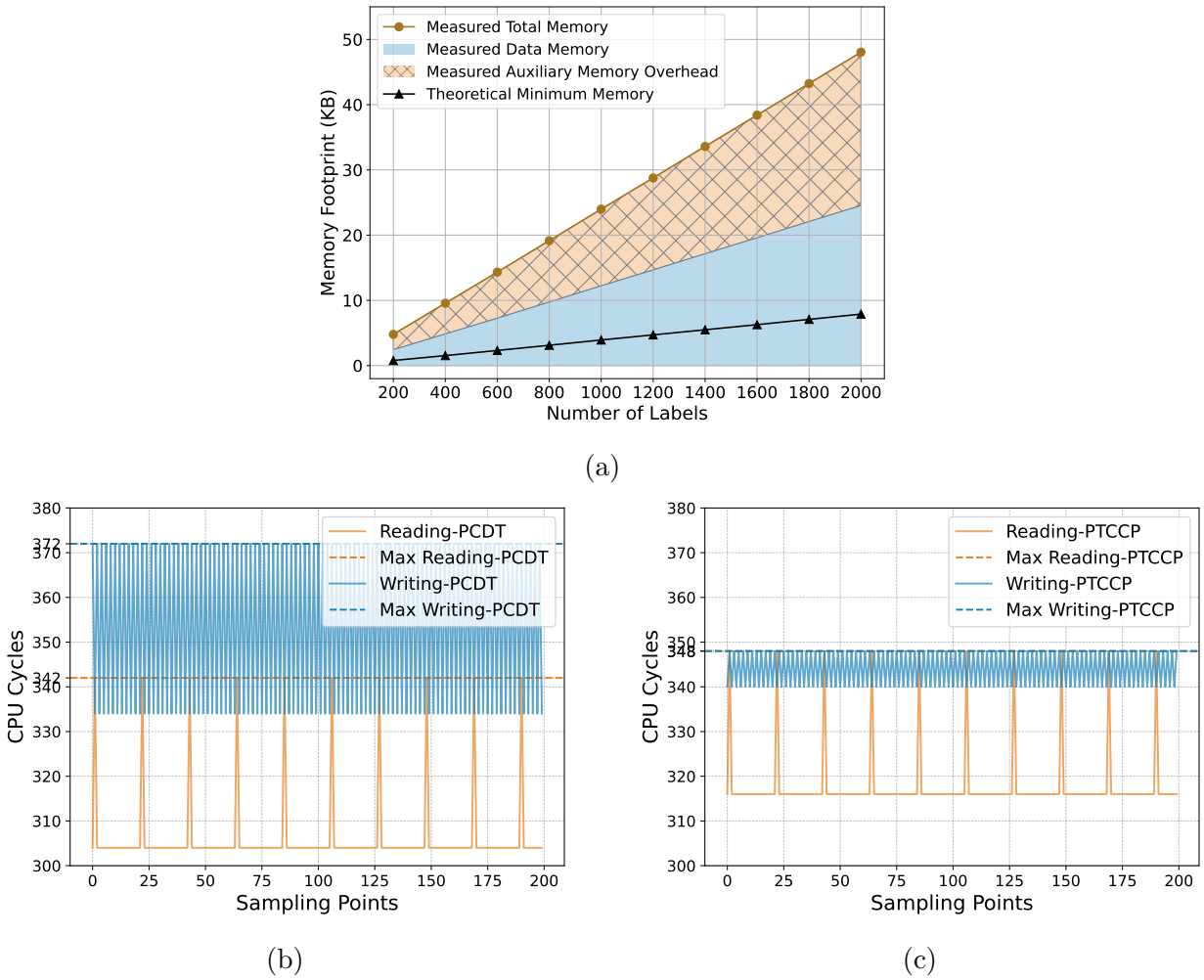


Figure 4.4: Experimental evaluation of PCDT on the STM32H747I-DISCO prototype platform. (a) Breakdown of measured memory consumption under varying communication loads, showing consistency between practical and theoretical memory usage. (b) Cycle-level profiling of PCDT read/write operations on the Cortex-M7 core, demonstrating moderate write overhead due to buffer scanning. (c) PTCCP profiling for comparison, showing a slightly faster write path using circular buffering, while maintaining similar read timing.

Section 4.7, requiring only minimal modifications to the application-level tasks. To evaluate the protocol's memory behavior, we varied the data communication load by adjusting the number of active data labels, rather than scaling task utilization. The resulting memory usage was profiled on the STM32H747I-DISCO board and is illustrated in Fig. 4.4a, which reports four categories of memory overhead.

The Measured Total Memory reflects the overall memory consumed by PCDT's wait-free communication, including all metadata and buffers. The Measured Data Memory isolates the portion specifically allocated to store communication payloads. The Measured Auxiliary Memory Overhead accounts for metadata such as per-reader buffer indices and protocol-related control variables. Finally, the Theoretical Minimum Memory represents the ideal memory requirement derived from the analytical model, assuming perfect packing without alignment constraints.

As expected, due to word alignment and padding requirements imposed by the underlying memory management, the measured memory footprint exceeds the theoretical minimum. However, all measured metrics scale linearly with the theoretical values, showing a tight proportional relationship. This confirms the practicality of applying PCDT in real embedded systems: the memory savings demonstrated analytically are preserved on actual hardware, enabling predictable and efficient deployment in memory-constrained platforms.

Runtime Timing Overhead: We also evaluated the runtime timing overhead introduced specifically by the wait-free communication protocol. In such protocols, multi-buffering mechanisms require each read and write operation to locate the appropriate buffer instance, introducing additional computational overhead as reflected in Algorithm 12 and Algorithm 13.

The execution times of read and write operations were measured on the STM32H747I-DISCO

board using the trace tool TraceX that comes with the ThreadX family. For each protocol, 200 sampling points were collected to profile both data reading and writing. The results are presented in Fig.4.4b for PCDT and in Fig.4.4c for PTCCP. The performance of PDBP is skipped, as it shares a similar runtime profile with PCDT.

As expected, data read operations exhibit consistent timing across all wait-free protocols. These operations primarily involve referencing the most recently updated buffer, which incurs minimal cost. On the 400 MHz Cortex-M7 core, the data read latency ranged from 308 to 348 CPU cycles, corresponding to a worst-case execution time of less than 0.87 μs, a negligible portion of the shortest task WCET in our benchmarks.

In contrast, data write operations vary across protocols. PCDT (and similarly PDBP) must search through the buffer pool to identify an available, unoccupied buffer, while PTCCP employs a circular buffer, enabling faster identification of the next write slot. As a result, PCDT incurs approximately 6.9% more CPU cycles during writes compared to PTCCP. However, as shown in Fig. 4.3, this modest increase in timing overhead is justified by substantial memory savings.

Overall, this prototype demonstrates that PCDT can be deployed on a commercially available platform with minimal integration effort. It introduces only minor, predictable timing overhead and preserves the memory efficiency observed in simulation. PCDT is also compatible with synchronous multi-processing systems, making it broadly applicable. These results confirm that the benefits predicted by analytical models and simulation carry over reliably to real-world embedded platforms.

4.8.2 Simulation Experiments

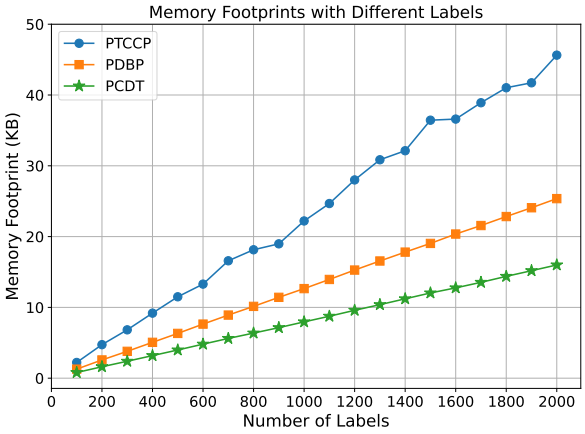
To further validate the memory efficiency and scalability of the proposed PCDT protocol, we conduct a comprehensive set of simulation experiments. These experiments are divided into two parts: The first part compares memory footprints across three wait-free communication protocols, PDBP, PTCCP, and PCDT, under varying system loads. The second part focuses on memory optimization within the PCDT framework by evaluating the impact of different priority assignment and task partitioning strategies.

Cross-Protocol Comparison under Varying Loads

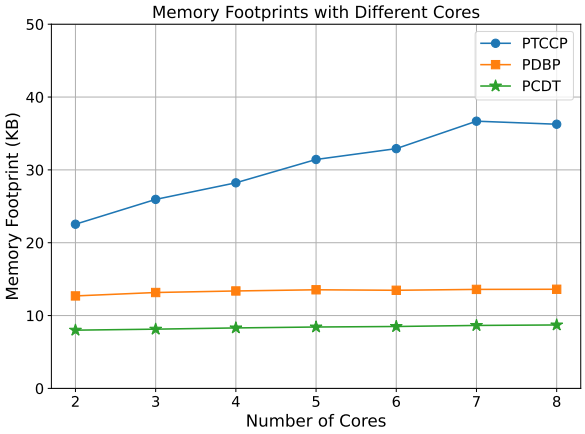
Experimental Setup: We simulate a multicore real-time system based on an industry-standard benchmark [57]. In each experiment, a set of tasks communicates via shared data labels across cores, using one of the three wait-free protocols. All protocols are evaluated under fixed Deadline Monotonic (DM) scheduling. A task set is considered schedulable if the calculated response-time meets the deadline for all tasks; otherwise, the system falls back to the protocol's upper bound memory cost, which is $n + 1$ buffers per label, where n is the number of readers.

For each configuration, we generate 1,000 random task sets, each containing between 5 and 50 tasks. The total system utilization is uniformly sampled from the range $[50\% * m, 90\% * m]$ where m is the number of cores. Individual task utilizations are generated using the UUniFast algorithm [14], and tasks are then randomly partitioned across the cores. The number of communication labels varies from 100 to 2,000, and the number of cores ranges from 2 to 8. Each label is randomly assigned between 2 and 5 reader tasks. The size of each label is randomly selected from {1, 2, 4, 8, 16, 32, 64, 128} bytes, based on the benchmark

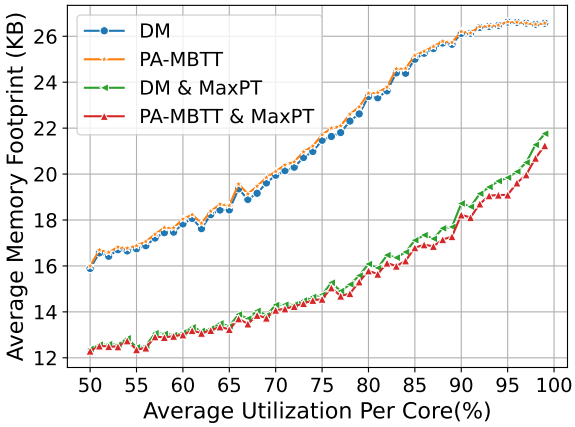
distribution, with corresponding probabilities {34%, 48%, 13%, 1%, 1%, 1%, 1%, 1%} [57]. This experimental design ensures a comprehensive evaluation of protocol memory efficiency across a wide range of computational and communication loads.



(a) Average memory footprint with increasing number of labels.



(b) Average memory footprint with increasing number of cores.



(c) Average memory footprint across different scheduling policies.

Figure 4.5: Experimental evaluation on different data communication loads and scheduling policies.

Results: Fig. 4.5a shows the average memory footprint as the number of labels increases in a 2-core system. As expected, more labels induce more inter-task communication, increasing buffer requirements. Among all tested protocols, PCDT consistently exhibits the lowest

memory footprint. In high-label scenarios, PCDT reduces memory usage by 36.76% to 37.57% compared to PDBP.

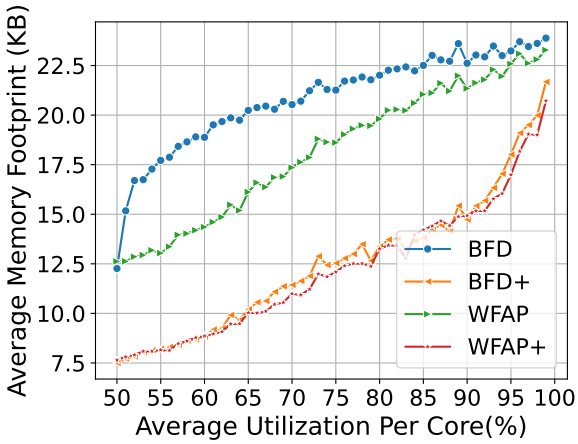
Similarly, Fig. 4.5b evaluates the impact of the number of cores on memory consumption with 1,000 communication labels. As the number of cores increases, the frequency of cross-core communication rises, which amplifies memory usage for all protocols. However, PCDT again outperforms the others, showing 36.01% to 38.22% lower memory usage than its counterparts. These results confirm PCDT's superior scalability for large-scale, high-load systems.

Memory Optimization with PCDT under Priority Assignment and Partitioning Policies

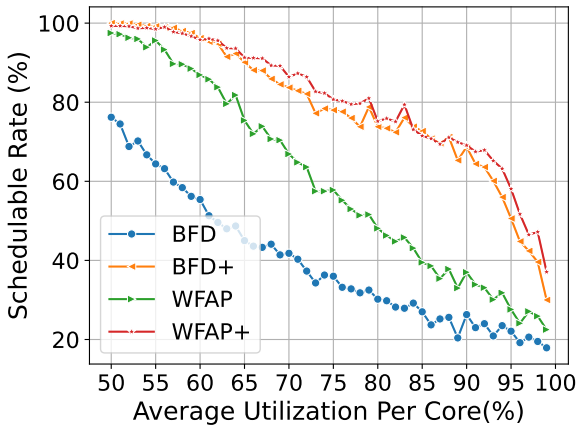
Experimental Focus: In this part, we focus exclusively on PCDT and investigate how scheduling and partitioning strategies affect memory usage. The goal is to explore memory optimization opportunities when more intelligent scheduling policies and task allocation algorithms are applied.

Impact of Scheduling Policies: We evaluate PCDT memory consumption under different priority assignment strategies: the baseline Deadline Monotonic (DM), our proposed PA-MBTT (Priority Assignment with Maximum Blocking Time Tolerance), and their respective variants with preemption threshold (PT) support.

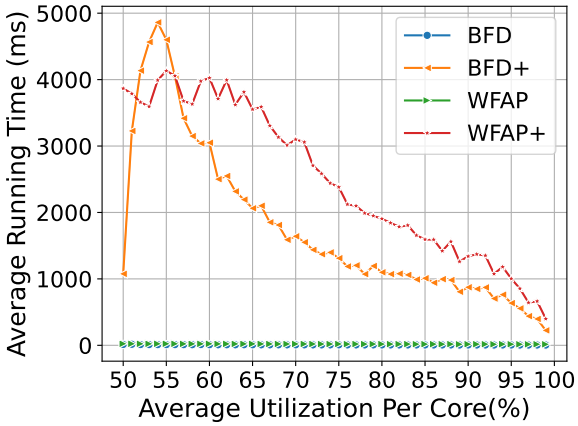
Fig. 4.5c compares these policies while sweeping per-core utilization. While PA-MBTT alone does not always outperform DM in isolation, both PA-MBTT and DM benefit significantly when combined with the assignment of maximum preemption thresholds. In particular, PA-MBTT with Maximum PT consistently achieves the best memory efficiency, reducing the number of required buffers without sacrificing schedulability.



(a) Average memory footprint across all evaluated task sets.



(b) Schedulability ratio achieved by each partitioning policy.



(c) Runtime cost of computing the partitioning results.

Figure 4.6: Impact of task partitioning policies on system performance.

Impact of Task Partitioning Policies: Next, we fix the scheduling policy to PA-MBTT with maximum preemption thresholds and explore the impact of different task partitioning methods. We compare the baseline Best-Fit Decreasing (BFD) approach to our proposed Wait-Free-Aware Partitioning (WFAP), along with their respective local search enhancements (BFD+ and WFAP+).

As shown in Fig. 4.6a, WFAP consistently provides better memory efficiency than BFD, especially under constrained timing budgets such as those in online or dynamic load scenarios. Specifically, WFAP achieves memory savings in the range of 0.0% to 52.52% compared to BFD, with an average saving of 30.72%. When additional computation time is available, WFAP+ further improves the quality of task partitioning. It reduces memory footprint more substantially and increases the schedulability ratio of the system, achieving memory savings of from 22.21% to 70.62% compared to BFD, with an average saving of 53.67%.

Fig. 4.6 quantifies the trade-off between memory efficiency and runtime. These results demonstrate that WFAP is highly effective for real-time adaptation due to its balance between performance and computational cost, while WFAP+ is more appropriate for offline or static system planning where time constraints are relaxed.

This set of experiments confirms that PCDT, when paired with effective scheduling and partitioning heuristics, can further reduce memory demands and improve system schedulability. The combined use of PA-MBTT, preemption thresholds, and WAFP+ offers a highly scalable and memory-efficient framework for real-time embedded systems.

4.9 Conclusion

This chapter introduced the Partitioned Combined-DBP-TCCP (PCDT) protocol, a novel wait-free communication strategy designed to ensure data consistency and eliminate communication blocking in multicore real-time embedded systems. Tailored for high-integrity applications such as ADAS, PCDT enables selective adoption of PDBP or PTCCP semantics for each data consumer, providing flexibility to optimize memory usage and timing guarantees based on individual task requirements.

Beyond protocol design, we proposed a set of system-level memory optimization techniques that significantly reduce the overall communication memory footprint. Our priority assignment scheme, PA-MBTT, in combination with preemption threshold scheduling, improves memory efficiency over traditional scheduling approaches. Additionally, our Wait-Free-Aware Partitioning (WFAP) and its local search variant (WFAP+) offer scalable and memory-conscious task-to-core mapping solutions, demonstrating significant savings even under high communication loads. Extensive simulation and case study evaluations confirmed the advantages of PCDT and the proposed optimizations, achieving an average of 53.67% memory reduction compared to existing solutions.

Future work will explore hybrid memory management strategies that combine lock-based and wait-free mechanisms and develop adaptive scheduling and partitioning frameworks that respond to dynamic workloads in real-time embedded environments.

Chapter 5

Conclusion

This dissertation presents a comprehensive multi-layer framework aimed at improving the efficiency, safety, and predictability of autonomous vehicle (AV) systems. By addressing the challenges at the system, application, and low-level communication layers, this work contributes to the design of real-time embedded solutions that are both computationally efficient and practically deployable.

5.1 Summary of Contributions

5.1.1 System Layer: Intelligent Intersection Management

The system-level component introduces a dynamic scheduling framework for unsignalized intersections, enabling safe and efficient vehicle coordination without traffic signals. Central to this framework is the Extended Conflict Directed Graph (ECDG) model, which captures not only temporal and spatial conflicts among Connected and Automated Vehicles (CAVs) but also incorporates flexible departure lane assignments. This novel representation enables a dynamic scheduling strategy that assigns departure lanes and time windows jointly, reducing the need for prior lane changes and thus minimizing collision risks and route constraints.

The breadth-first search-based scheduling algorithm improves intersection evacuation time by up to 16.3% and boosts overall traffic efficiency by 27.7% compared to prior state-of-the-

art approaches. These results validate the model's scalability and potential for real-world deployment in urban traffic environments.

5.1.2 Application Layer: Real-Time DAG Scheduling with Non-Preemptive Execution

At the application level, the focus shifts to the computational scheduling of DAG-based real-time tasks. The non-preemptive execution model is highly desirable for AV safety-critical systems but introduces complexity due to non-convex timing constraints. This work introduces a novel 1-opt-based optimization approach that partitions the solution space into convex regions and iteratively solves subproblems using linear programming.

The proposed scheduler guarantees convergence to a 1-opt solution, balancing optimization performance and computational tractability. Experiments demonstrate latency reductions ranging from 20% to 40% compared to traditional heuristic and metaheuristic approaches, with superior scalability for large task sets. This result is especially relevant for emerging AV workloads in perception, sensor fusion, and decision-making.

5.1.3 Middleware Layer: Wait-Free Communication for Multicore Systems

At the middleware layer, this work addresses efficient and predictable inter-core communication on multicore embedded processors. Existing synchronization protocols based on locks introduce blocking and priority inversion, both detrimental to real-time guarantees. This dissertation introduces the Partitioned Combined-DBP-TCCP (PCDT) protocol, a hybrid wait-free strategy that adapts to each reader's timing requirements by selectively combining

the strengths of the Dynamic Buffering Protocol (DBP) and Temporal Concurrency Control Protocol (TCCP).

To reduce memory usage, the protocol is supported by a set of system-level techniques: (1) PA-MBTT for priority assignment with preemption thresholds, and (2) WFAP+ for wait-free-aware task-to-core partitioning. Together, these approaches lead to an average of 53.7% memory savings over baseline approaches in simulation, and a 43% reduction in a real-world automotive case study, without compromising schedulability.

5.2 Implications and Broader Impact

Collectively, the three layers demonstrate a unified approach to building dependable, resource-aware, and timing-predictable AV systems. From high-level traffic coordination to low-level inter-core communication, each layer tackles a critical bottleneck in current AV architecture design.

- The system-level contributions enable AVs to safely operate in fully autonomous intersection environments, contributing to the vision of signal-free smart cities.
- The application-level scheduler provides a systematic method for balancing latency and safety in increasingly complex AV software pipelines.
- The middleware-layer protocol and memory optimization strategies ensure these systems can be realized on realistic embedded platforms without excessive resource requirements.

By enabling these innovations, the proposed framework makes autonomous systems more robust, responsive, and scalable.

5.3 Future Directions

This work opens several promising avenues for future research that can extend and enhance the multi-layer framework proposed in this dissertation. At the system level, future efforts may focus on adaptive intersection management by incorporating pedestrian behavior, heterogeneous traffic involving both autonomous and human-driven vehicles, and real-time traffic learning techniques to further improve safety and efficiency in complex urban scenarios.

At the application and communication layers, there are opportunities to extend the 1-opt scheduling framework to support preemptive execution models and optimize across multiple conflicting objectives such as latency, energy, and throughput. Additionally, hybrid communication schemes that integrate wait-free and lock-based mechanisms could be explored to dynamically balance memory usage and timing predictability under variable workloads. Advancing these directions will contribute to the development of fully autonomous, real-time, and resilient cyber-physical mobility systems suitable for deployment at scale.

Bibliography

[1] Jakaria Abdullah, Gaoyang Dai, and Wang Yi. Worst-case cause-effect reaction latency in systems with non-blocking communication. In *Design, Automation & Test in Europe*, pages 1625–1630. IEEE, 2019.

[2] Zaid Al-Bayati, Youcheng Sun, Haibo Zeng, Marco Di Natale, Qi Zhu, and Brett Meyer. Task placement and selection of data consistency mechanisms for real-time multicore applications. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 172–181. IEEE, 2015.

[3] Zaid Al-Bayati, Youcheng Sun, Haibo Zeng, Marco Di Natale, Qi Zhu, and Brett H Meyer. Partitioning and selection of data consistency mechanisms for multicore real-time systems. *ACM Transactions on Embedded Computing Systems*, 18(4):1–28, 2019.

[4] Mahdi Amouzadi, Mobolaji Olawumi Orisatoki, and Arash M Dizqah. Optimal lane-free crossing of cavs through intersections. *IEEE Transactions on Vehicular Technology*, 72(2):1488–1500, 2022.

[5] James H Anderson, Srikanth Ramamurthy, and Kevin Jeffay. Real-time computing with lock-free shared objects. *ACM Transactions on Computer Systems*, 15(2):134–165, 1997.

[6] Neil Audsley, Alan Burns, Mike Richardson, Ken Tindell, and Andy J Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software engineering journal*, 8(5):284–292, 1993.

[7] Neil C Audsley. On priority assignment in fixed priority scheduling. *Information Processing Letters*, 79(1):39–44, 2001.

[8] Yuhao Bai, Baohua Zhang, Naimin Xu, Jun Zhou, Jiayou Shi, and Zhihua Diao. Vision-based navigation and guidance for agricultural autonomous vehicles and robots: A review. *Computers and Electronics in Agriculture*, 205:107584, 2023.

[9] Theodore P. Baker. Stack-based scheduling of realtime processes. *Real-Time Systems*, 3(1):67–99, 1991.

[10] Sanjoy Baruah. Scheduling dags when processor assignments are specified. In *International Conference on Real-Time Networks and Systems*, pages 111–116, 2020.

[11] Sanjoy Baruah, Marko Bertogna, Giorgio Buttazzo, Sanjoy Baruah, Marko Bertogna, and Giorgio Buttazzo. The sporadic dag tasks model. *Multiprocessor Scheduling for Real-Time Systems*, pages 191–204, 2015.

[12] Sanjoy K Baruah and Nathan Wayne Fisher. The partitioned dynamic-priority scheduling of sporadic task systems. *Real-Time Systems*, 36:199–226, 2007.

[13] E. Bini and G. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30:129–154, 2005.

[14] Enrico Bini and Giorgio C Buttazzo. Measuring the performance of schedulability tests. *Real-time systems*, 30(1):129–154, 2005.

[15] Christian Bradatsch, Florian Kluge, and Theo Ungerer. Data age diminution in the logical execution time model. In *Architecture of Computing Systems*, pages 173–184. Springer, 2016.

[16] Reinder J Bril, Sebastian Altmeyer, Martijn MHP van den Heuvel, Robert I Davis, and Moris Behnam. Integrating cache-related pre-emption delays into analysis of fixed

priority scheduling with pre-emption thresholds. In *2014 IEEE Real-Time Systems Symposium*, pages 161–172. IEEE, 2014.

[17] Reinder J Bril, Sebastian Altmeyer, Martijn MHP van den Heuvel, Robert I Davis, and Moris Behnam. Fixed priority scheduling with pre-emption thresholds and cache-related pre-emption delays: integrated analysis and evaluation. *Real-Time Systems*, 53(4):403–466, 2017.

[18] Giorgio C Buttazzo. *Hard real-time computing systems: predictable scheduling algorithms and applications*, volume 24. Springer Science & Business Media, 2011.

[19] Mengchi Cai, Qing Xu, Chaoyi Chen, Jiawei Wang, Keqiang Li, Jianqiang Wang, and Xiangbin Wu. Multi-lane unsignalized intersection cooperation with flexible lane direction based on multi-vehicle formation control. *IEEE Transactions on Vehicular Technology*, 71(6):5787–5798, 2022.

[20] Chaoyi Chen, Qing Xu, Mengchi Cai, Jiawei Wang, Biao Xu, Xiangbin Wu, Jianqiang Wang, Keqiang Li, and Chunyu Qi. A graph-based conflict-free cooperation method for intelligent electric vehicles at unsignalized intersections. In *2021 IEEE International Intelligent Transportation Systems Conference*, pages 52–57. IEEE, 2021.

[21] Chaoyi Chen, Mengchi Cai, Jiawei Wang, Kai Li, Qing Xu, Jianqiang Wang, and Keqiang Li. Cooperation method of connected and automated vehicles at unsignalized intersections: Lane changing and arrival scheduling. *IEEE Transactions on Vehicular Technology*, 71(11):11351–11366, 2022.

[22] Chaoyi Chen, Qing Xu, Mengchi Cai, Jiawei Wang, Jianqiang Wang, and Keqiang Li. Conflict-free cooperation method for connected and automated vehicles at unsignalized intersections: Graph-based modeling and optimality analysis. *IEEE Transactions on Intelligent Transportation Systems*, 2022.

[23] Jing Chen and Alan Burns. Loop-free asynchronous data sharing in multiprocessor real-time systems based on timing properties. In *Proceedings Sixth International Conference on Real-Time Computing Systems and Applications.*, pages 236–246. IEEE, 1999.

[24] Pranav Singh Chib and Pravendra Singh. Recent advancements in end-to-end autonomous driving using deep learning: A survey. *IEEE Transactions on Intelligent Vehicles*, 9(1):103–118, 2023.

[25] Michael B Cohen, Yin Tat Lee, and Zhao Song. Solving linear programs in the current matrix multiplication time. *Journal of the ACM*, 68(1):1–39, 2021.

[26] IBM ILOG Cplex. V12. 1: User's manual for cplex. *International Business Machines Corporation*, 46(53):157, 2009.

[27] Robert I Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM computing surveys (CSUR)*, 43(4):1–44, 2011.

[28] Robert I Davis, Liliana Cucu-Grosjean, Marko Bertogna, and Alan Burns. A review of priority assignment in real-time systems. *Journal of systems architecture*, 65:64–82, 2016.

[29] Xuanliang Deng, Ashrarul H Sifat, Shao-Yu Huang, Sen Wang, Jia-Bin Huang, Changhee Jung, Ryan Williams, and Haibo Zeng. Partitioned scheduling with safety-performance trade-offs in stochastic conditional dag models. *Journal of Systems Architecture*, 153:103189, 2024.

[30] Marco Dürr, Georg von der Brüggen, Kuan-Hsun Chen, and Jian-Jia Chen. End-to-end timing analysis of sporadic cause-effect chains in distributed systems. *ACM Transactions on Embedded Computing Systems*, 18:1 – 24, 2019.

[31] Andrew Eliasz. A review of rtos fundamentals. *Zephyr RTOS Embedded C Programming: Using Embedded RTOS POSIX API*, pages 19–67, 2024.

[32] Nico Feiertag, Kai Richter, Johan Nordlander, and Jan Jonsson. A compositional framework for end-to-end path delay calculation of automotive systems under different path semantics. In *IEEE Real-Time Systems Symposium*, 2009.

[33] FHWA. Federal highway administration intersection safety, 2024. URL https://highways.dot.gov/safety/intersection-safety/about.

[34] José Fonseca, Geoffrey Nelissen, and Vincent Nélis. Improved response time analysis of sporadic dag tasks for global fp scheduling. In *International Conference on Real-Time Networks and Systems*, pages 28–37, 2017.

[35] Keir Fraser. Practical lock-freedom. Technical report, University of Cambridge, Computer Laboratory, 2004.

[36] Simon Fürst, Jürgen Mössinger, Stefan Bunzel, Thomas Weber, Frank Kirschke-Biller, Peter Heitkämper, Gerulf Kinkelin, Kenji Nishikawa, and Klaus Lange. Autosar–a worldwide standard is on the road. In *14th International VDI Congress Electronic Systems for Vehicles, Baden-Baden*, volume 62. Citeseer, 2009.

[37] Paolo Gai, Giuseppe Lipari, and Marco Di Natale. Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. In *Proceedings 22nd IEEE Real-Time Systems Symposium*, pages 73–83. IEEE, 2001.

[38] Rony Ghattas and Alexander G Dean. Preemption threshold scheduling: Stack optimality, enhancements and analysis. In *13th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS'07)*, pages 147–157. IEEE, 2007.

[39] Zonghua Gu, Gang Han, Haibo Zeng, and Qingling Zhao. Security-aware mapping and scheduling with hardware co-processors for flexray-based distributed embedded systems. *IEEE Transactions on parallel and distributed systems*, 27(10):3044–3057, 2016.

[40] Zonghua Gu, Chao Wang, and Haibo Zeng. Cache-partitioned preemption threshold scheduling, 2016.

[41] Mario Günzel, Kuan-Hsun Chen, Niklas Ueter, Georg von der Brüggen, Marco Dürr, and Jian-Jia Chen. Timing analysis of asynchronized distributed cause-effect chains. In *IEEE 27th Real-Time and Embedded Technology and Applications Symposium*, pages 40–52, 2021.

[42] Mario Günzel, Harun Teper, Kuan-Hsun Chen, Georg von der Brüggen, and Jian-Jia Chen. On the equivalence of maximum reaction time and maximum data age for cause-effect chains. In *Euromicro Conference on Real-Time Systems*, 2023. URL https://api.semanticscholar.org/CorpusID:259318171.

[43] Arne Hamann, Dakshina Dasari, Simon Kramer, Michael Pressler, and Falk Wurst. Communication centric design in complex automotive embedded systems. In *Euromicro Conference on Real-Time Systems*, 2017.

[44] Gang Han, Haibo Zeng, Marco Di Natale, Xue Liu, and Wenhua Dou. Experimental evaluation and selection of data consistency mechanisms for hard real-time applications on multicore platforms. *IEEE Transactions on Industrial Informatics*, 10(2):903–918, 2013.

[45] Jian-Jun Han, Sunlu Gong, Zhenjiang Wang, Wen Cai, Dakai Zhu, and Laurence T Yang. Blocking-aware partitioned real-time scheduling for uniform heterogeneous mul-

ticore platforms. *ACM Transactions on Embedded Computing Systems (TECS)*, 19(1): 1–25, 2020.

[46] Qingqiang He, Mingsong Lv, and Nan Guan. Response time bounds for dag tasks with arbitrary intra-task priority assignment. In *Euromicro Conference on Real-Time Systems*, 2021.

[47] Zhengbing He, Liang Zheng, Lili Lu, and Wei Guan. Erasing lane changes from roads: A design of future road intersections. *IEEE Transactions on Intelligent Vehicles*, 3(2): 173–184, 2018.

[48] Hai Huang, Padmanabhan Pillai, and Kang G Shin. Improving wait-free algorithms for interprocess communication in embedded real-time systems. In *USENIX Annual Technical Conference, General Track*, pages 303–316, 2002.

[49] Wen-Hung Huang, Maolin Yang, and Jian-Jia Chen. Resource-oriented partitioned scheduling in multiprocessor systems: How to partition and how to share? In *2016 IEEE Real-Time Systems Symposium (RTSS)*, pages 111–122. IEEE, 2016.

[50] Haoran Jiang, Zhihong Yao, Yangsheng Jiang, and Zhengbing He. Is all-direction turn lane a good choice for autonomous intersections? a study of method development and comparisons. *IEEE Transactions on Vehicular Technology*, 2023.

[51] Xu Jiang, Xiantong Luo, Nan Guan, Zheng Dong, Shaoshan Liu, and Wang Yi. Analysis and optimization of worst-case time disparity in cause-effect chains. In *Design, Automation & Test in Europe*, pages 1–6. IEEE, 2023.

[52] David S Johnson. Fast algorithms for bin packing. *Journal of Computer and System Sciences*, 8(3):272–314, 1974.

[53] Richard M Karp. Reducibility among combinatorial problems. In *50 Years of Integer Programming 1958-2008: from the Early Years to the State-of-the-Art*, pages 219–241. Springer, 2009.

[54] Kiran Khurshid, Safwat Irteza, Adnan Ahmed Khan, and Sumaira Shah. Application of heuristic (1-opt local search) and metaheuristic (ant colony optimization) algorithms for symbol detection in mimo systems. *Communications and Network*, 03:200–209, 2011. URL https://api.semanticscholar.org/CorpusID:44194350.

[55] Tobias Klaus, Matthias Becker, Wolfgang Schröder-Preikschat, and Peter Ulbrich. Constrained data-age with job-level dependencies: How to reconcile tight bounds and overheads. In *IEEE 27th Real-Time and Embedded Technology and Applications Symposium*, pages 66–79, 2021.

[56] Simon Kramer, Dirk Ziegenbein, and Arne Hamann. Real world automotive benchmarks for free. In *International Workshop on Analysis Tools and Methodologies for Embedded and Real-Time Systems*, volume 130, 2015.

[57] Simon Kramer, Dirk Ziegenbein, and Arne Hamann. Real world automotive benchmarks for free. In *6th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, volume 130, 2015.

[58] Karthik Lakshmanan, Dionisio de Niz, and Ragunathan Rajkumar. Coordinated task scheduling, allocation and synchronization on multiprocessors. In *2009 30th IEEE Real-Time Systems Symposium*, pages 469–478. IEEE, 2009.

[59] Karthik Lakshmanan, Shinpei Kato, and Ragunathan Rajkumar. Scheduling parallel real-time tasks on multi-core processors. In *2010 31st IEEE Real-Time Systems Symposium*, pages 259–268. IEEE, 2010.

[60] John P Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *[1990] Proceedings 11th Real-Time Systems Symposium*, pages 201–209. IEEE, 1990.

[61] Dong Li, Sen Wang, and Haibo Zeng. Safe and efficient unsignalized intersection management with breadth-first spanning tree. In *2024 IEEE 27th International Conference on Intelligent Transportation Systems (ITSC)*, pages 3297–3302, 2024. doi: 10.1109/ITSC58415.2024.10920199.

[62] Jing Li, Kunal Agrawal, Chenyang Lu, and Christopher D. Gill. Analysis of global edf for parallel tasks. In *Euromicro Conference on Real-Time Systems*, pages 3–13, 2013.

[63] Li Li and Fei-Yue Wang. Cooperative driving at blind crossings using intervehicle communication. *IEEE Transactions on Vehicular technology*, 55(6):1712–1724, 2006.

[64] Ruoxiang Li, Nan Guan, Xu Jiang, Zhishan Guo, Zheng Dong, and Mingsong Lv. Worst-case time disparity analysis of message synchronization in ros. In *IEEE Real-Time Systems Symposium*, pages 40–52, 2022.

[65] Yi-Ting Lin, Hsiang Hsu, Shang-Chien Lin, Chung-Wei Lin, Iris Hui-Ru Jiang, and Changliu Liu. Graph-based modeling, scheduling, and verification for intersection management of intelligent vehicles. *ACM Transactions on Embedded Computing Systems*, 18(5s):1–21, 2019.

[66] Changliu Liu, Chung-Wei Lin, Shinichi Shiraishi, and Masayoshi Tomizuka. Distributed conflict resolution for connected autonomous vehicles. *IEEE Transactions on Intelligent Vehicles*, 3(1):18–29, 2017.

[67] Chung Laung Liu and James W Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973.

[68] Luiz Maia and Gerhard Fohler. Reducing end-to-end latencies of multi-rate cause-effect chains in safety critical embedded systems. In *12th European Congress on Embedded Real Time Software and Systems (ERTS 2024)*, 2024.

[69] Jorge Martinez, Ignacio Sañudo, and Marko Bertogna. Analytical characterization of end-to-end communication delays with logical execution time. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37:2244–2254, 2018.

[70] Jorge Martinez, Ignacio Sañudo, and Marko Bertogna. End-to-end latency characterization of task communication models for automotive systems. *Real-Time Systems*, 56: 315 – 347, 2020. URL https://api.semanticscholar.org/CorpusID:219935505.

[71] Alessandra Melani, Marko Bertogna, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, and Giorgio C. Buttazzo. Response-time analysis of conditional dag tasks in multiprocessor systems. In *Euromicro Conference on Real-Time Systems*, pages 211–221, 2015.

[72] Anna Minaeva and Zdeněk Hanzálek. Survey on periodic scheduling for time-triggered hard real-time systems. *ACM Computing Surveys*, 54:1 – 32, 2021. URL https://api.semanticscholar.org/CorpusID:233354222.

[73] Amir Mirheli, Mehrdad Tajalli, Leila Hajibabai, and Ali Hajbabaie. A consensus-based distributed trajectory control in a signal-free intersection. *Transportation research part C: emerging technologies*, 100:161–176, 2019.

[74] Elnaz Namazi, Jingyue Li, and Chaoru Lu. Intelligent intersection management systems considering autonomous vehicles: A systematic literature review. *IEEE Access*, 7:91946–91965, 2019.

[75] Farhang Nemati, Thomas Nolte, and Moris Behnam. Partitioning real-time systems

on multiprocessors with shared resources. In *International Conference On Principles Of Distributed Systems*, pages 253–269. Springer, 2010.

[76] Jorge Nocedal and Stephen J Wright. Nonlinear equations. *Numerical Optimization*, pages 270–302, 2006.

[77] Ignacio Sanudo Olmedo, Nicola Capodieci, Jorge Luis Martinez, Andrea Marongiu, and Marko Bertogna. Dissecting the cuda scheduling hierarchy: a performance and predictability perspective. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 213–225, 2020.

[78] Francesco Paladino, Alessandro Biondi, Enrico Bini, and Paolo Pazzaglia. Optimizing Per-Core Priorities to Minimize End-To-End Latencies. In Rodolfo Pellizzoni, editor, *36th Euromicro Conference on Real-Time Systems (ECRTS 2024)*, volume 298 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 6:1–6:25, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-324-9. doi: 10.4230/LIPIcs.ECRTS.2024.6. URL https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECRTS.2024.6.

[79] Darsh Parekh, Nishi Poddar, Aakash Rajpurkar, Manisha Chahal, Neeraj Kumar, Gyanendra Prasad Joshi, and Woong Cho. A review on autonomous vehicles: Progress, methods and challenges. *Electronics*, 11(14):2162, 2022.

[80] Jaehyun Park and Stephen Boyd. A semidefinite programming method for integer convex quadratic minimization. *Optimization Letters*, 12:499–518, 2018.

[81] PerceptIn. 2021 rtss industry challenge. http://2021.rtss.org/industry-session/, 2021.

[82] Bo Qian, Haibo Zhou, Feng Lyu, Jinglin Li, Ting Ma, and Fen Hou. Toward collision-free and efficient coordination for automated vehicles at unsignalized intersection. *IEEE Internet of Things Journal*, 6(6):10408–10420, 2019.

[83] Md Mokhlesur Rahman and Jean-Claude Thill. Impacts of connected and autonomous vehicles on urban transportation and environment: A comprehensive review. *Sustainable Cities and Society*, 96:104649, 2023.

[84] Ragunathan Rajkumar. Real-time synchronization protocols for shared memory multiprocessors. In *Proceedings., 10th International Conference on Distributed Computing Systems*, pages 116–117. IEEE Computer Society, 1990.

[85] John Regehr. Scheduling tasks with mixed preemption relations for robustness to timing faults. In *23rd IEEE Real-Time Systems Symposium, 2002. RTSS 2002.*, pages 315–326. IEEE, 2002.

[86] Manas Saksena and Yun Wang. Scalable real-time system design using preemption thresholds. In *Proceedings 21st IEEE Real-Time Systems Symposium*, pages 25–34. IEEE, 2000.

[87] Lui Sha, Ragunathan Rajkumar, and John P Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9): 1175–1185, 1990.

[88] Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine. The boost graph library - user guide and reference manual. In *C++ in-depth series*, 2001.

[89] Amit Kumar Singh, Piotr Dziurzanski, Hashan Roshantha Mendis, and Leandro Soares Indrusiak. A survey and comparative study of hard and soft real-time dynamic resource

allocation strategies for multi-/many-core systems. *ACM Computing Surveys (CSUR)*, 50(2):1–40, 2017.

[90] Christos Sofronis, Stavros Tripakis, and Paul Caspi. A memory-optimal buffering protocol for preservation of synchronous semantics under preemptive scheduling. In *Proceedings of the 6th ACM & IEEE International conference on Embedded software*, pages 21–33, 2006.

[91] Yue Tang, Xu Jiang, Nan Guan, Dong Ji, Xiantong Luo, and Wang Yi. Comparing communication paradigms in cause-effect chains. *IEEE Transactions on Computers*, 72:82–96, 2023.

[92] K. Tindell, A. Burns, and A. Wellings. Allocating hard real-time tasks: An np-hard problem made easy. *Real-Time Systems*, 4:145–165, 2004.

[93] Haluk Rahmi Topcuoglu, Salim Hariri, and Minyou Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Trans. Parallel Distributed Syst.*, 13:260–274, 2002.

[94] Peter JM Van Laarhoven, Emile HL Aarts, Peter JM van Laarhoven, and Emile HL Aarts. *Simulated annealing*. Springer, 1987.

[95] Micaela Verucchi, Mirco Theile, Marco Caccamo, and Marko Bertogna. Latency-aware generation of single-rate dags from multi-rate task sets. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 226–238, 2020.

[96] Chao Wang, Zonghua Gu, and Haibo Zeng. Integration of cache partitioning and preemption threshold scheduling to improve schedulability of hard real-time systems. In *2015 27th Euromicro Conference on Real-Time Systems*, pages 69–79. IEEE, 2015.

[97] Chao Wang, Zonghua Gu, and Haibo Zeng. Global fixed priority scheduling with preemption threshold: Schedulability analysis and stack size minimization. *IEEE Transactions on Parallel and Distributed Systems*, 27(11):3242–3255, 2016.

[98] Guoqiang Wang, Marco Di Natale, and Alberto Sangiovanni-Vincentelli. Improving the size of communication buffers in synchronous models with time constraints. *IEEE Transactions on Industrial Informatics*, 5(3):229–240, 2009.

[99] Michael I-C Wang, Jiacheng Wang, Charles H-P Wen, and H Jonathan Chao. Roadrunner: Autonomous intersection management with dynamic lane assignment. In *2020 IEEE 23rd International Conference on Intelligent Transportation Systems*, pages 1–7. IEEE, 2020.

[100] Sen Wang, Ryan K Williams, and Haibo Zeng. A general and scalable method for optimizing real-time systems with continuous variables. In *IEEE 29th Real-Time and Embedded Technology and Applications Symposium*, pages 119–132, 2023.

[101] Sen Wang, Dong Li, Shao-Yu Huang, Xuanliang Deng, Ashrarul H Sifat, Jia-Bin Huang, Changhee Jung, Ryan Williams, and Haibo Zeng. Time-triggered scheduling for nonpreemptive real-time dag tasks using 1-opt local search. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 43(11):3650–3661, 2024.

[102] Sen Wang, Dong Li, Ashrarul H. Sifat, Shao-Yu Huang, Xuanliang Deng, Changhee Jung, Ryan Williams, and Haibo Zeng. Optimizing logical execution time model for both determinism and low latency. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 135–148, 2024.

[103] Yun Wang and Manas Saksena. Scheduling fixed-priority tasks with preemption threshold. In *Proceedings Sixth International Conference on Real-Time Computing Systems and Applications*, pages 328–335. IEEE, 1999.

[104] Ruiyi Wu, Hongfei Jia, Qiuyang Huang, Jingjing Tian, Heyao Gao, and Guanfeng Wang. Multi-lane unsignalized intersection cooperation strategy considering platoons formation in a mixed connected automated vehicles and connected human-driven vehicles environment. *IEEE Transactions on Intelligent Transportation Systems*, 2023.

[105] Biao Xu, Shengbo Eben Li, Yougang Bian, Shen Li, Xuegang Jeff Ban, Jianqiang Wang, and Keqiang Li. Distributed conflict-free cooperation for multiple connected vehicles at unsignalized intersections. *Transportation Research Part C: Emerging Technologies*, 93:322–334, 2018.

[106] Huile Xu, Yi Zhang, Li Li, and Weixia Li. Cooperative driving at unsignalized intersections using tree search. *IEEE Transactions on Intelligent Transportation Systems*, 21(11):4563–4571, 2019.

[107] Huile Xu, Christos G Cassandras, Li Li, and Yi Zhang. Comparison of cooperative driving strategies for cavs at signal-free intersections. *IEEE Transactions on Intelligent Transportation Systems*, 23(7):7614–7627, 2021.

[108] Fei Yan, Mahjoub Dridi, and Abdellah El Moudni. Autonomous vehicle sequencing algorithm at isolated intersections. In *2009 12th International IEEE conference on intelligent transportation systems*, pages 1–6. IEEE, 2009.

[109] Maolin Yang, Wen-Hung Huang, and Jian-Jia Chen. Resource-oriented partitioning for multiprocessor systems with shared resources. *IEEE Transactions on Computers*, 68(6):882–898, 2018.

[110] Zhihong Yao, Haoran Jiang, Yangsheng Jiang, and Bin Ran. A two-stage optimization method for schedule and trajectory of cavs at an isolated autonomous intersection. *IEEE Transactions on Intelligent Transportation Systems*, 24(3):3263–3281, 2023.

[111] Haibo Zeng and Marco Di Natale. Mechanisms for guaranteeing data consistency and flow preservation in autosar software on multi-core platforms. In *2011 6th IEEE International Symposium on Industrial and Embedded Systems*, pages 140–149. IEEE, 2011.

[112] Haibo Zeng and Marco Di Natale. Efficient implementation of autosar components with minimal memory usage. In *IEEE International Symposium on Industrial Embedded Systems*, pages 130–137. IEEE, 2012.

[113] Haibo Zeng, Marco Di Natale, Arkadeb Ghosal, and Alberto Sangiovanni-Vincentelli. Schedule optimization of time-triggered systems communicating over the flexray static segment. *IEEE Transactions on Industrial Informatics*, 7(1):1–17, 2010.

[114] Haibo Zeng, Marco Di Natale, and Qi Zhu. Optimizing stack memory requirements for real-time embedded applications. In *Proceedings of 2012 IEEE 17th International Conference on Emerging Technologies & Factory Automation (ETFA 2012)*, pages 1–8. IEEE, 2012.

[115] Haibo Zeng, Marco Di Natale, and Qi Zhu. Minimizing stack and communication memory usage in real-time embedded applications. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(5s):1–25, 2014.

[116] Qingling Zhao, Zonghua Gu, and Haibo Zeng. Integration of resource synchronization and preemption-thresholds into edf-based mixed-criticality scheduling algorithm. In *2013 IEEE 19th International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 227–236. IEEE, 2013.

[117] Qingling Zhao, Zonghua Gu, and Haibo Zeng. Pt-amc: Integrating preemption thresholds into mixed-criticality scheduling. In *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 141–146. IEEE, 2013.

[118] Qingling Zhao, Zonghua Gu, and Haibo Zeng. Resource synchronization and preemption thresholds within mixed-criticality scheduling. *ACM Transactions on Embedded Computing Systems (TECS)*, 14(4):1–25, 2015.

[119] Qingling Zhao, Zonghua Gu, and Haibo Zeng. Design optimization for autosar models with preemption thresholds and mixed-criticality scheduling. *Journal of Systems Architecture*, 72:61–68, 2017.

[120] Qingling Zhao, Zonghua Gu, Haibo Zeng, and Nenggan Zheng. Schedulability analysis and stack size minimization with preemption thresholds and mixed-criticality scheduling. *Journal of Systems Architecture*, 83:57–74, 2018.

[121] Qingling Zhao, Mengfei Qu, Bo Huang, Zhe Jiang, and Haibo Zeng. Schedulability analysis and stack size minimization for adaptive mixed criticality scheduling with semi-clairvoyance and preemption thresholds. *Journal of Systems Architecture*, 124: 102383, 2022.

[122] Yecheng Zhao and Haibo Zeng. The concept of unschedulability core for optimizing priority assignment in real-time systems. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pages 232–237. IEEE, 2017.

[123] Yecheng Zhao and Haibo Zeng. The virtual deadline based optimization algorithm for priority assignment in fixed-priority scheduling. In *IEEE Real-Time Systems Symposium*, pages 116–127, 2017.

[124] Yecheng Zhao and Haibo Zeng. The concept of response time estimation range for optimizing systems scheduled with fixed priority. In *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 283–294. IEEE, 2018.

[125] Yecheng Zhao and Haibo Zeng. The concept of unschedulability core for optimizing

real-time systems with fixed-priority scheduling. *IEEE Transactions on Computers*, 68(6):926–938, 2018.

[126] Yecheng Zhao and Haibo Zeng. The concept of maximal unschedulable deadline assignment for optimization in fixed-priority scheduled real-time systems. *Real-Time Systems*, 55(3):667–707, 2019.

[127] Yecheng Zhao, Vinit Gala, and Haibo Zeng. A unified framework for period and priority optimization in distributed hard real-time systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2188–2199, 2018.

[128] Yecheng Zhao, Runzhi Zhou, and Haibo Zeng. An optimization framework for real-time systems with sustainable schedulability analysis. In *IEEE Real-Time Systems Symposium*, pages 333–344. IEEE, 2020.

[129] Yecheng Zhao, Runzhi Zhou, and Haibo Zeng. Design optimization for real-time systems with sustainable schedulability analysis. *Real-Time Systems*, 58(3):275–312, 2022.

[130] Anye Zhou, Srinivas Peeta, Menglin Yang, and Jian Wang. Cooperative signal-free intersection control using virtual platooning and traffic flow regulation. *Transportation research part C: emerging technologies*, 138:103610, 2022.