BOSTON UNIVERSITY

GRADUATE SCHOOL OF ARTS AND SCIENCES

Dissertation

TOWARDS A CENTRALIZED MULTICORE AUTOMOTIVE SYSTEM

by

SOHAM SINHA

B.E., Bengal Engineering and Science University, Shibpur, 2012
M.Sc., University of Alberta, 2016

Submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

2022

Approved by

First Reader

Richard West, PhD
Professor of Computer Science

Second Reader

Renato Mancuso, PhD
Assistant Professor of Computer Science

Third Reader

Ragunathan “Raj” Rajkumar, PhD
Professor of Electrical and Computer Engineering
Carnegie Mellon University

Fourth Reader

Vasiliki Kalavri, PhD
Assistant Professor of Computer Science

Science talks about very simple things, and asks hard questions about them. As soon as things become too complex, science can't deal with them. . . And it rarely reaches human affairs. Human affairs are way too complicated. . . So the actual sciences tell us virtually nothing about human affairs.

Noam Chomsky

Acknowledgments

I would like to thank my advisor, Richard West for his guidance and support throughout the years of my PhD. He has encouraged me to keep pursuing my research through the ups and downs. Our engaging discussions and his insights have helped me form my own ideas about systems research. I would also like to thank Renato Mancuso, Raj Rajkumar, and Vasiliki Kalavri for providing their invaluable feedback on my research.

To my mother, Kakali Sinha, thank you for being the source of strength and determination throughout all these years. To my wonderful wife, Sritama Basu, thank you for being my best friend, the greatest support and source of happiness. To the rest of my family, thank you for understanding my journey in this uncharted path and having faith in me.

I would like to dedicate this thesis to my father and my maternal uncle. My sincerity and perseverance are entirely my father's credit. I am sure that he would be more proud of my PhD than me, had he been alive today. Finally, it is my maternal uncle's passion and curiosity in engineering and computers that got me interested in it at an early stage.

I want to thank my friends and lab-mates, Ahmad Golchin, Craig Einstein, Anam Farrukh, Tom Cheng, Katheine Missimer, Zhiyuan Ruan and Anton Njavro for their help and time. I would also like to thank my friend and longest flatmate, Aditya Narayan for always being there and hearing me out whenever needed. Thanks to Pratik Sarkar and Kinan Bab for being such great friends and colleagues in this foreign land. My PhD experience has been more memorable and fun because of every one of you.

This research has been generously funded by NSF Grant #1527050, #2007707, Intel and Drako Motors. Thanks to all of them. Special thanks to my colleagues at Drako Motors and Celenum, especially Shiv Sikand, Andy Cook and Nguyen Thi Le Chau.

TOWARDS A CENTRALIZED MULTICORE AUTOMOTIVE SYSTEM

SOHAM SINHA

Boston University, Graduate School of Arts and Sciences, 2022

Major Professor: Richard West, Professor of Computer Science

ABSTRACT

Today's automotive systems are inundated with embedded electronics to host chassis, powertrain, infotainment, advanced driver assistance systems, and other modern vehicle functions. As many as 100 embedded microcontrollers execute hundreds of millions of lines of code in a single vehicle. To control the increasing complexity in vehicle electronics and services, automakers are planning to consolidate different on-board automotive functions as software tasks on centralized multicore hardware platforms. However, these vehicle software services have different and contrasting timing, safety, and security requirements. Existing vehicle operating systems are ill-equipped to provide all the required service guarantees on a single machine. A centralized automotive system aims to tackle this by assigning software tasks to multiple criticality domains or levels according to their consequences of failures, or international safety standards like ISO 26262. This research investigates several emerging challenges in time-critical systems for a centralized multicore automotive platform and proposes a novel vehicle operating system framework to address them.

This thesis first introduces an integrated vehicle management system (VMS), called DriveOS^{TM}, for a PC-class multicore hardware platform. Its separation kernel design enables temporal and spatial isolation among critical and non-critical vehicle services in

different domains on the same machine. Time- and safety-critical vehicle functions are implemented in a sandboxed Real-time Operating System (OS) domain, and non-critical software is developed in a sandboxed general-purpose OS (e.g., Linux, Android) domain. To leverage the advantages of model-driven vehicle function development, DriveOS provides a multi-domain application framework in Simulink. This thesis also presents a real-time task pipeline scheduling algorithm in multiprocessors for communication between connected vehicle services with end-to-end guarantees. The benefits and performance of the overall automotive system framework are demonstrated with hardware-in-the-loop testing using real-world applications, car datasets and simulated benchmarks, and with an early-stage deployment in a production-grade luxury electric vehicle.

Contents

List of Tables

List of Figures

List of Abbreviations

ADAS	.....	Advanced Driver Assistance System
E2E	.....	End-to-end
ECU	.....	Electronic Control Unit
EPT	.....	Extended Page Table
GPOS	.....	General Purpose Operating System
HIL	.....	Hardware-in-the-loop
HVAC	.....	Heating, Ventilation and Air Conditioning
IC	.....	Instrument Cluster
IVI	.....	In-vehicle Infotainment
LBG	.....	Latency Budget Gap
MCS	.....	Mixed-criticality System
MIL	.....	Model-in-the-loop
NLBG	.....	Normalized Latency Budget Gap
OS	.....	Operating System
RTOS	.....	Real-time Operating System
VM	.....	Virtual Machine
VMM	.....	Virtual Machine Monitor
VMS	.....	Vehicle Management System

Chapter 1

Introduction

The role of electronics and software in modern cars has rapidly grown over the last couple of decades. Electronic components now contribute more than 40% of a vehicle's total cost, almost doubled since 2007 [Del19]. Vehicle control functions like chassis, body, powertrain are collectively controlled by nearly 100 embedded microcontrollers [Fle01, MV13, Win19], also known as Electronic Control Units (ECUs), with hundreds of millions of lines of code [BDDK18] in a single vehicle. Each ECU has its own control logic that is costly and inflexible to upgrade, cumbersome to maintain and limited to feature extensions [OTNK18]. Short of reflashing or adding new ECUs, it is difficult, if not impossible, to fix or update the capabilities of a vehicle already in use, resulting in an ever-increasing number of car recalls [Ahs13, MPM+19, Sto20].

The number of ECUs in a single vehicle is expected to rise as electronics play an increasing role in support of advanced features such as automated driving, cloud connectivity, electrification and "virtual cockpit" [CKK+18, HBGR17]. In the current automotive systems, the ECUs are physically interconnected in a vehicle network such as a Controller Area Network (CAN) bus. As emerging vehicle functions add more ECUs, physical network connections and software code, automakers are facing a challenge of unprecedented scale and complexity in the automotive systems [Cha21]. The hardware costs, wiring, packaging and maintenance have become increasingly prohibitive [BDS19]. The

existing vehicle networks are also not suitable to handle the large scale data communication required in new functions such as autonomous driving [Flo21, MSS+18]. Moreover, simple ECU hardware lacks modern-day computing security guarantees [KCR+10, CS16, ERSS+20], and functional errors are hard to detect and fix in a complicated vehicle network [EJ09, EKI17].

An alternative approach to using large numbers of separate ECUs is to develop a centralized vehicle management system (VMS) [SCB+13, BDDK18, Del20, BSPL21] where ECU functions are consolidated as software tasks on a single multicore machine [DNSV10]. Software is more easily upgraded, replaced and extended, without the cost and complexity of added electronics, on a centralized computing platform [Int20]. Instead of long and intertwined vehicular network, simpler networks connect sensors and actuators with simple transreceivers to the central computer, which hosts an appropriate I/O bus interface. Commodity-class multicore hardware are also cheaper than specialized microcontrollers to curb the growing cost of electronics. However, software tasks must now implement ECU functions with different timing, safety and security requirements on a centralized hardware. For example, a powertrain control task may have a stricter timing requirement than an in-vehicle infotainment software. Timing and functional verification of the ECU software also needs to be revised for multicore machines [UOO15, BMC+17]. Furthermore, an ECU task should be capable of communicating with another ECU task within predictable time bounds like in a physical connection. These software guarantees are especially difficult to accomplish in commodity-class multicore platforms [DAN+13, KKR13, WHKA13, YMWP14, YWZC16, MRR+19].

1.1 Problem Statement

Replacing ECUs with multiple software-defined functions requires first and foremost

a suitable operating system. While Linux is used by automotive companies such as Tesla [Tes22], BMW [Lin19], and Toyota [Den17], for its infotainment services, it lacks the necessary timing and safety requirements for correct vehicle operations in all conditions. Real-time components must be executed according to strict timing guarantees. Safety-critical software components must be isolated from less-critical services, according to different safety integrity standards such as ISO 26262-3 [ISO11]. General purpose operating systems (GPOSes) such as Linux and Android are insufficient on their own to provide the necessary spatial separation of highly critical services, with high consequences of failure, from those of low-criticality ones [Ves07]. Similarly, these OSes lack the real-time capabilities, including time-critical sensor and actuator I/O operations, necessary to meet predictable service guarantees [RMF19]. Without significant modifications and subsequent formal method verification, Linux's use in automotive systems is limited to non-critical functions.

Conversely, real-time operating systems (RTOSes) [DLW11, Fre22a, Win22a, eCo22] are capable of providing timing guarantees to the applications, but have limited support of pre-existing libraries, device drivers, services and applications. New vehicle functions such as advanced driver assistance system (ADAS) and In-vehicle Infotainment (IVI), require cutting-edge machine learning (ML), graphics and other libraries which are only available in GPOSes like Linux, Windows and Mac OS. Hence, an RTOS in and itself is not capable of providing all the functions of a VMS. Thus, an integrated vehicle operating system must have both the benefits of a GPOS and an RTOS to support time-critical and non-critical automotive services on a centralized platform.

A significant challenge in integrating vehicle ECU functions in a centralized VMS is how to develop, deploy and support communication between *mixed-criticality* tasks, spanning different operating systems, or domains. To date, most vehicle functions such

as heating, ventilation and air conditioning (HVAC) or powertrain control are developed for simple, single-core ECUs in model-based design languages such as Simulink and LabView [Fri06]. These ECUs host a single, simple RTOS or firmware. Engineers accustomed to model-based languages develop functions for these ECUs without awareness of control flow (e.g., threads), data structures, and low-level communication primitives. Model-based design languages have thus far lacked support for multi-OS domain systems, leaving the burden on expert programmers to port ECU functions.

Moreover, ECUs in a vehicle are often interconnected in a serial pipeline or chain to coordinate over a physical control network [DZDN+07, MN15, BDM+16b]. Similarly, in a centralized VMS, software tasks are connected by data buffers in task pipelines or task chains or "cause-effect" chains [SE16, HDK+17, KBS18, KBS20, CKK20, GCU+21]. The end-to-end properties of a task pipeline like the maximum reaction time of a task chain [DBCC19], are important to establish the worst-case bounds for safety requirements or Quality-of-service guarantees of an automotive system. For example, Rimac Automobili's torque vectoring control has a 10ms worst-case end-to-end latency requirement [RIM16]. However, finding schedulable task runtimes and periods to satisfy the end-to-end properties of a task pipeline is an integer non-linear constraint programming problem that is NP-hard [DZDN+07]. Traditional mixed-integer non-linear programming (MINLP) solvers [BHMH18, BHH+21] are too slow to derive the timing properties of the pipelined tasks and not a feasible OS-level solution. Furthermore, scheduling a set of pipelined tasks under the constraints is not yet addressed in a multiprocessor system.

1.2 Thesis Claims

This thesis makes the following claims:

1. A centralized multicore automotive system based on a separation kernel enables the

legacy and library-dependent vehicle software to gain real-time capabilities, and also empowers the domain-specific, real-time applications to leverage the advantages of pre-exisiting libraries, device drivers and services.

2. The timing requirements of co-running critical and non-critical vehicle services are satisfied, when they are temporally and spatially isolated in multiple domains of a centralized multicore automotive system which communicates with a vehicle CAN bus network.

3. If a multi-domain Simulink model of a vehicle ECU function is ported as a multi-threaded software application in a timing-predictable multicore automotive system, the end-to-end delay of the function model is guaranteed to be under the theoretical upper bounds without compromising on any functional specifications.

4. A schedulable set of task runtime budgets and periods which meets the end-to-end constraints of a connected pipeline of ECU software threads, is derived by a heuristic constraint solver faster than the traditional MINLP solvers.

Chapter 3 presents experimental evidences for Claim 1 and 2. Claim 3 is proved by empirical results in Chapter 4. Chapter 5 corroborates Claim 4.

1.2.1 Thesis Statement

This thesis empirically proves that connected vehicle ECU functions, integrated as multiple software threads into a centralized multicore automotive operating system, achieve predictable end-to-end properties when they communicate with a vehicle CAN bus network using real-world and simulated datasets.

1.3 Research Contributions

This thesis introduces the DriveOS integrated vehicle management system (VMS) that securely and predictably consolidates software-defined ECU functions on a centralized platform. It supports the co-existence of an RTOS and a legacy OS such as Linux or Android on a DX1100 multicore Workstation [Cin22]. DriveOS is based on a real-time separation kernel [Rus81, WLMD16] which maps guest OSes to secure sandbox domains that have direct access to partitioned machine physical resources. Timing-sensitive services are deployed as real-time tasks in a sandboxed RTOS domain running our in-house Quest RTOS [DLW11, Wes22]. Non-critical, legacy and library-dependent software such as Instrument Cluster (IC), IVI and machine learning-based ADAS services are deployed in Linux and Android sandboxes. We have designed and implemented a low-overhead and secure inter-sandbox communication mechanism, named `shmcomm`, which allows both synchronous and asynchronous message-passing between sandbox domains in DriveOS. DriveOS is currently deployed in a production-grade luxury electric vehicle with various applications integrated using our design and interfaces. Experimental evidences with real-world applications and datasets show that DriveOS has predictable end-to-end latency for integrated real-time and legacy software.

We present a multi-domain application development framework in Simulink, named ModelMap, for model-based automotive functions in DriveOS. For engineers familiar with the model-driven vehicle function design paradigm, ModelMap supports binding a real-time periodic thread to a Simulink control task for timing-predictable execution. It provides synchronous and asynchronous inter-task communication primitives, and real-time I/O for commonly used protocols such as CAN bus. Vehicle functions that span OS domains are encapsulated as *nested binaries*, which support the deployment of executable code for multiple application binary interfaces. To the best of our knowledge, ModelMap

is the first open model-based multi-domain VMS application framework. As a case study, an HVAC control Simulink model is integrated in DriveOS as a real-time software thread using ModelMap and shown to have the same functional outputs as its MIL execution.

This thesis also proposes a heuristic constraint solver algorithm for ECU software task pipelines, so that the end-to-end guarantees of task chains are satisfied. Our heuristic algorithm, CoPi derives timing properties of a pipeline of periodic tasks against the maximum end-to-end delay and loss-rate constraints under a fixed-priority task scheduling algorithm. CoPi is demonstrated in uniprocessor and partitioned multiprocessor scheduling with simulated tasksets.

In summary, this thesis presents the design, policies and mechanism of a vehicle operating system framework which addresses some emerging challenges of a centralized automotive platform and demonstrates their advantages with real-world applications and experiments.

1.4 Thesis Organization

The thesis is organized as follows. In the next chapter, we discuss the related work to different pieces of this research and the background of our work. Chapter 3 describes the design and implementation of DriveOS. Chapter 4 presents the ModelMap application framework for model-based vehicle functions in DriveOS. Chapter 5 covers the CoPi heuristic constraint solver algorithm for end-to-end scheduling of real-time task pipelines and demonstrates its benefits. Finally, the thesis conclusions and potential future work are discussed in Chapter 6.

Chapter 2

Background and Related Work

This chapter provides an overview of the background and related work on centralized automotive systems. First, we present an overview of the automotive systems software design and implementation in Section 2.1. Then, we discuss the previous research on real-time task pipeline scheduling Section 2.2.

2.1 Automotive System Software

A number of automakers are now developing OSes for their vehicles. Tesla uses its own version of Linux [Tes22] for its display devices. Toyota's Entune [Den17] for multimedia and telematics is based on Automotive Grade Linux (AGL) [The21]. BMW's driving and infotainment system OS7 is built on Yocto Linux [Lin19]. Volvo's Polestar has adopted a bare-metal Android Automotive OS [Goo21a]. COVESA (formerly, GENIVI) [COV22] and other alliances between automotive companies are also developing AUTOSAR- [AUT22] and AGL-compliant OSes for modern vehicles. There have been many other attempts to use Linux in time- and safety-critical scenarios [Y+99, Win22b, DM03]. While Linux and Android provide a rich set of features, they lack real-time capabilities needed for critical tasks in modern vehicle management systems. DriveOS uses a `PREEMPT_RT`-patched Linux [RMF19] and executes time-critical tasks in an RTOS domain, while ensuring complementary Linux services are sufficiently predictable for use in automotive

applications such as IC, IVI and ADAS.

QNX is a real-time microkernel [Hil92, vdVD04] used by Ford's SYNC infotainment system [For21] and NVIDIA DRIVE OS [Nvi22a]. It runs in millions of vehicles [Bla21]. QNX hypervisor supports real-time tasks in its QNX Neutrino RTOS, and other tasks in a Linux virtualized guest OS. The microkernel handles inter-process communication (IPC), process scheduling and interrupts. In comparison, DriveOS delegates process scheduling and interrupt handling to individual sandboxes. Only the shmcomm module in DriveOS hypervisor manages the IPC between applications in different sandboxes. In contrast to QNX's proprietary architecture, the design and implementation of DriveOS are openly available [SW21].

Large automotive companies like General Motors and Mercedes, are developing their own next-generation vehicle management systems that will control critical and non-critical vehicle functions in a centralized platform [Gen22, Mer22, Ape22, Ope22]. They are planning to utilize some features of Linux, but the system designs are not publicly available. Their planned deployments of these new systems are still a few years away [Mer22]. On the other hand, Quest-V has demonstrated the timing-predictable usage of different Linux distributions like Ubuntu and Yocto Linux in time-critical contexts over the last few years [WLMD16, YCSW18], including in an interactive Android-based automotive system [SGEW20a, SGEW20b]. DriveOS hypervisor is based on the Quest-V [WLMD16] separation kernel [Rus81] that acts like a partitioning hypervisor. Next, we discuss the related works on partitioning hypervisors and the advantages of Quest-V over others.

2.1.1 Partitioning Hypervisors

Quest-V [WLMD16] is a separation kernel that statically partitions hardware resources among multiple guest sandboxes. Each sandbox is responsible for task scheduling and

device handling without the involvement of the Quest-V hypervisor. The Quest RTOS [DLW11, Wes22] bootstraps Quest-V and initializes other sandboxes. Other partitioning hypervisors like Jailhouse [RKLM17] and ACRN [LXRD19] rely on Linux to bootstrap the system. Bao adopts a clean-slate partitioning hypervisor implementation for ARM and RISC-V architectures, without relying on Linux [MTS+20]. Using Linux to bootstrap guests in Jailhouse and ACRN increases the security attack surface of the partitioning hypervisor. In addition, ACRN's Linux-based service OS manages the hardware resources for other safety-critical sandboxes, unlike Quest-V's policy of directly assigning devices to guest sandboxes. PikeOS [KW07] and Muen [BR13] separation kernels also do not support independent interrupt handling by the guest sandboxes.

Inspired by Quest-V, DriveOS is bootstrapped by the relatively small Quest system, with less than 4KB of its codebase remaining within the hypervisor ("ring -1" in x86 terminology) privilege level. This eases the path to verification and certification by regulatory authorities. As the hypervisor occupies the most privileged protection domain, and it is not required for runtime resource management decisions by its guests, it is removed for regular control-flow operations. This heightens the security of the system.

Although Quest-V supports communication between sandboxes [LWCM14], it uses Inter-processor Interrupts for such communications. This reduces the available CPU utilization of the guest sandbox as an interrupt handling thread needs to be dedicated to communication requests. In contrast, DriveOS entirely relies on EPT hardware virtualization in x86 for predictable inter-sandbox communication and provides a POSIX file I/O-like API in Quest and Linux. Unlike Jailhouse and ACRN, DriveOS shows the utility of a partitioning hypervisor in the context of a vehicle management system, where most carmakers are using flavors of Linux. DriveOS also demonstrates the benefits of integrating a real-time virtual CAN interface in Linux for automotive systems.

2.1.2 Autonomous Driving Infrastructure

Recent years have seen numerous efforts to support autonomous driving. Autoware is a self-driving infrastructure for NVIDIA DRIVE PX2 [Nvi22b], which provides machine learning frameworks for object detection, and path planning [KTM+18]. Apollo is another such infrastructure project by Baidu [Apo21]. OpenPilot is an open-source adaptive cruise controller [com20], challenging Tesla's Autopilot and FSD (Full Self-Driving) [Tes21]. There are plenty of other self-driving start-up companies working on various automotive platforms [Way22, Zoo22, Gho22, Nur22, Arg22, Cru22, Gee22]. Most of these projects heavily rely on flavors of Linux and are not focusing on building a full-scale automotive software system. Their approaches are complementary to DriveOS. DriveOS is capable of incorporating third-party applications; it already integrates the IC and IVI applications of our partner software company, as well as OpenPilot.

2.1.3 Model-based Software Development

MATLAB/Simulink is the de-facto design tool for model-based vehicle control software [Fri06]. Proprietary functional blocks are commonly provided by vendors for use on their own ECUs, which traditionally feature single-core microcontrollers. However, the growing popularity of embedded multicore processors has led to the development of task-parallel Simulink models. Pagetti *et al.* have done extensive work on multi-periodic Simulink models for multicore platforms based on the ROSACE architecture for avionics [PSG+14, PFF+18, BGL+20]. They employ formal verification techniques from design to code generation, to meet avionics standards. However, the multicore CPU model and verified code generation are hard to obtain, and they rely on *faithful* code translation from other high-level formally defined design languages like Lustre. This research focuses on multi-domain code generation using Simulink tools for DriveOS.

Formal verification has been applied on Simulink to C code generation [BKÁ+18], and Simulink models are being used in production by automotive companies [Fri06]. Despite design-level verification of Simulink models [dBR06, CCM+03, MBR06, RG14, FPZ15, FPZ15], little has been done on multi-domain code generation, deployment and end-to-end testing, especially on a distributed system-on-chip [FUVC13] platform such as DriveOS.

Emerging multi-domain vehicle management systems [Mer22, Ape22, Ope22, BSC+21] require redesigned control function development tools [SST07]. Although new approaches are being considered [CRZC16, RBH+18], they mostly target ECU-based systems [CWAF14] with a single RTOS [eCo22, The22, Fre22a, Win22a]. In recent years, MathWorks has presented Simulink Desktop Real-time [Mat22f] for real-time simulation of models, but it is not a VMS solution. Simulink now also supports Linux and VxWorks [Win22a] tasks, but without any periodic control mechanism [Mat22g]. Our work on ModelMap presents a model-based application framework for multiple domains in DriveOS.

2.2 Real-time Task Pipeline Scheduling

Real-time embedded and cyber-physical systems are amassed with examples of task pipelines where a series of tasks are connected by data-buffers. In automotive domains, a sensory input is passed on to a pipeline of processing and control tasks that activate an actuation output [HDK+17]. Scheduling algorithms for such communicating tasks against end-to-end constraints have long been an active area of research [GSS95, DZDN+07, XLXC14, ZDB+20].

Gerber et al. presented a generic framework that shows how constraint programming helps in guaranteeing end-to-end constraints in a task graph [GSS95]. Davare *et al.* presented an end-to-end timing analysis of task pipelines and provided an upper bound on

the end-to-end latency [DZDN+07]. This work is closely related to our work on CoPi for end-to-end task pipeline scheduling, as it also proposes the problem of finding task periods as an optimization problem under constraints, that is solved by Geometric Programming. However, Davare *et al.* do not consider loss-rate as a constraint and focuses on latency. In our work, we introduce loss-rate as one of the constraints and refine the optimization problem for finding suitable task periods. In addition, we use an improved end-to-end latency analysis by Dürr *et al.* [DBCC19]. Moreover, we have used latest open-source MINLP solvers [BHMH18, BHH+21] for comparison, which were unavailable at the time of Davare *et al.*'s work.

There are other research studies on task chains that explore the end-to-end timing analysis of task chains [BDM+17, KBS18, KBS20, GCU+21] in practical scenarios such as in drones [CWE18], in ROS [CBLB19, TFG+20]. Proposed scheduling algorithms based on these approaches rely on job release times [BDM+16a, CKK20].

For multiprocessors, Liu and Anderson have analyzed a global scheduling algorithm for pipelined periodic tasks [LA09a, LA09b]. Mixed-criticality processing pipelines are also being investigated [dNAK+17]. Nevertheless, they do not consider end-to-end constraints. Finally, period selection is a widely studied problem in real-time systems [BC08, NF15, CJK16, XCÅ16], even though they are not targeted to real-time task pipelines.

Chapter 3

DriveOS Vehicle Management System

Modern vehicles support 10s to 100s of millions of lines of code [BDDK18]. To counter the growing complexity of software and electronics, chipmakers such as Intel, and analytics firms like McKinsey have called for a centralized vehicle management system to reimagine modern cars from a hardware-driven mechanical machine to a software-driven electronic device [BDS19, Int20]. A few carmakers have recently announced their plans to develop a centralized vehicle operating system platform [Mer22, Ape22, Gen22]. Taking inspiration from AUTOSAR's requirements about future car operating systems [AUT22], we envision a centralized system built on a low-cost, standardized industrial PC [Nie16]. PC-class hardware provides multiple high-performance processing cores, gigabytes of memory, hardware virtualization, potential integration with time-triggered Ethernet or Time-Sensitive Networking (TSN) [Cis17, KAGS05], and support for hardware accelerators for use in machine learning, at a low cost. In comparison, ECUs typically feature small flash memories and megahertz-speed microcontrollers, with limited processing capabilities for next-generation automobiles.

A centralized PC-class vehicle platform needs a suitable operating system. Towards this goal, we introduce DriveOS [SW21], which securely co-hosts an RTOS with a legacy system such as Linux using a partitioning hypervisor. Each guest OS domain manages its own CPU cores, physical memory and I/O devices without runtime involvement of

a virtual machine monitor. DriveOS is bootstrapped by our own in-house real-time OS, Quest [DLW11, **?**, Wes22], which establishes secure communication channels with other guest domains running Linux and/or Android. First-class shared memory channels between co-hosted guest OSes provide real-time, low-latency, high bandwidth, and secure inter-sandbox communication. This enables a mutually beneficial *symbiosis* between the RTOS and each legacy system: legacy systems inherit real-time capabilities without modification, while the RTOS gains access to pre-existing libraries, device drivers, and services.

Unlike GPOSes, DriveOS supports real-time and predictable I/O, similar to what is available in typical microcontrollers. One or more USB-CAN interfaces [Kva22] connect a DriveOS central computer, acting as a CAN concentrator, to a network of sensors and actuators. A real-time USB 3.0 protocol ensures predictable data movement between the DriveOS host and peripheral devices connected to each CAN bus [GCW18].

Figure 3.1 shows a high-level overview of DriveOS on a DX1100 Workstation [Cin22]. This is an industrial PC-class computing platform, being tested within our partner electric car company, Drako Motors for integrated vehicle management. Although DriveOS supports multiple guests such as Ubuntu and Android, we use Yocto Linux in this work. Our Yocto Linux sandbox features: (1) an IC application that displays a graphical speedometer, battery meter and other indicator readings, (2) an IVI application that provides HVAC controls, navigation, and smartphone integration, and (3) ADAS services for adaptive cruise control and autonomous driving. The Quest sandbox implements a real-time CAN gateway service to facilitate sensor data processing, control, and secure communication with separate IC, IVI, and ADAS services. *Real-time service tasks* such as an ADAS longitudinal controller are also executed in Quest.

In this chapter, we: (1) describe the separation kernel that forms the basis of DriveOS, (2) introduce a low-overhead and secure inter-sandbox communication framework, named

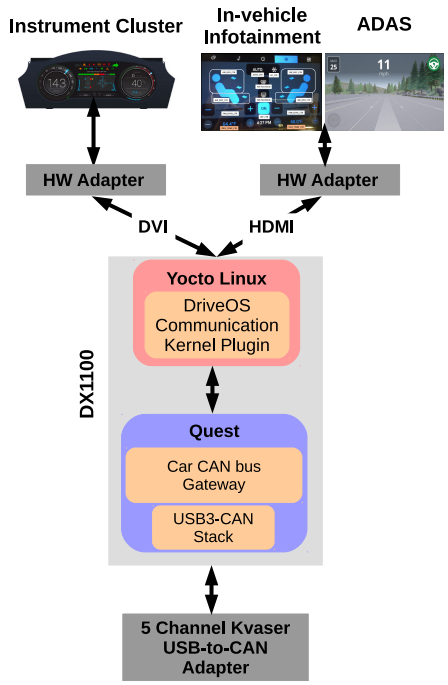


Figure 3.1: Integrated Vehicle Management in DriveOS

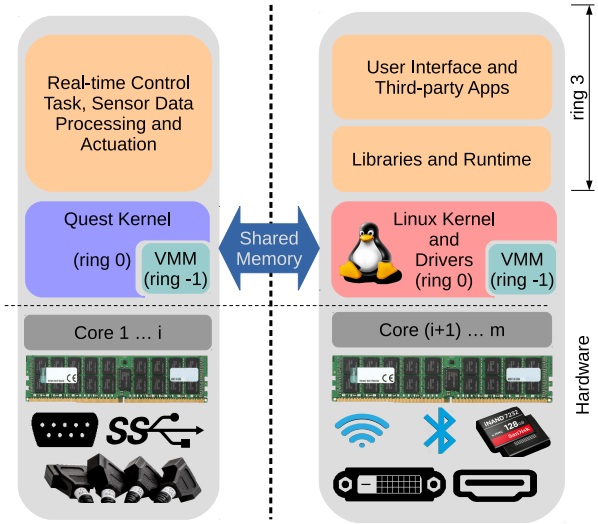


Figure 3.2: Resource Partitioning in DriveOS

`shmcomm`, which allows both synchronous and asynchronous message-passing between guest sandboxes, and (3) integrate IC, IVI and ADAS services in DriveOS to show the feasibility of our approach.

We compare an implementation of DriveOS, which is actively being developed for a production-grade electric car, with an alternative Linux system that is currently used in the automotive industry. Using a hardware-in-the-loop (HIL) CARLA driving simulation [DRC+17], and real car dataset collected from Laguna Seca raceway in California, we show that an optimized Linux supporting real-time tasks is unable to achieve the end-to-end delay and throughput guarantees provided by DriveOS, when tasks process and exchange data with a CAN bus network. At the same time, DriveOS provides the added security and isolation between software components of different criticality levels.

The next section provides a detailed description of the DriveOS design. Section 3.2

outlines the implementation of the DriveOS partitioning hypervisor, used to support secure and predictable separation of different application and kernel components. The inter-sandbox communication framework is then explained in Section 3.3, followed by a description of the DriveOS applications in Section 3.4. The chapter concludes with a system evaluation in Section 3.5.

3.1 DriveOS Design

DriveOS uses Linux as the basis for next-generation IC, IVI applications and ADAS user-interface control, with real-time features handled by our Quest RTOS. For example, an ADAS torque vectoring and traction control service configured for use on wet, dry, or snow-covered roads, must manage updates to wheel torques within specific time bounds to prevent the vehicle skidding out of control. While we want real-time control to be handled by suitably predictable real-time services, the interface to configure ADAS settings will be exposed to Linux. DriveOS hosts both Linux and Quest on a single machine, supporting communication between both guest OSes via secure shared memory channels. Thus, Linux is empowered with real-time capabilities afforded by Quest, and Quest is empowered with improved user-interactivity capabilities (including graphics and touchscreen control) provided by Yocto Linux. We now describe the design of our system in further detail, beginning with the partitioning hypervisor.

3.1.1 DriveOS Partitioning Hypervisor

Figure 3.2 shows a diagram of the DriveOS partitioning hypervisor, configured for a vehicle management system. DriveOS is implemented for the x86 architecture and statically partitions the hardware resources of a physical platform amongst each guest OS. This resource assignment makes use of hardware-assisted virtualization techniques to isolate

guest operating systems in distinct sandboxes.

At boot-time, DriveOS begins by executing Quest as though it were a standalone system. Quest contains the hypervisor logic to partition hardware resources among separate guest sandboxes, according to a boot-time configuration. One instance of Quest is replicated in non-overlapping physical memory for each guest sandbox, and then each guest is launched. Depending on the DriveOS configuration, one instance of Quest will act as a bootloader for Linux or Android, which becomes the active OS in the corresponding sandbox. For a system with at least two sandboxes, it is then possible to have Quest running in a guest domain that is isolated from another running an OS such as Linux. As Quest contains the hypervisor, or virtual machine monitor (VMM) code, to bootstrap a series of guests, each sandbox subsequently contains independent "root mode" (also called "ring -1") hypervisor code. This is a heightened privilege level address space over traditional kernel protection domains that run at "ring 0" on x86 systems. Similarly, each guest actively runs in "non-root mode" but is granted direct access to hardware resources that it can access without invoking the VMM.

Each guest's VMM code is only needed to execute a minimal set of privileged machine instructions that cause *VMExits* (from guest to hypervisor control-flow), and to establish new inter-sandbox shared memory channels between guests. The number of guest sandboxes and the mapping of hardware resources (CPU cores, physical memory ranges, and subsets of I/O devices) to guests is established by static system configuration information. As resources are partitioned rather than shared as in traditional hypervisors such as Xen [BDF+03], there is no duplication of potentially conflicting resource management policies between a guest OS and hypervisor. The lack of runtime resource management functionality in the DriveOS hypervisor means that ring -1 code has a text size of less than 4KB.

Each guest kernel in DriveOS operates in (non-root) ring 0. With the exception of Quest, all other guest OSes are paravirtualized to operate correctly within their virtualized domains. We have paravirtualized Yocto Linux, Ubuntu, and Android for DriveOS with less than 150 lines of code changes in the Linux kernel. These changes are mostly to handle direct memory access (DMA) requests, where guest and machine physical memory addresses differ, and the processor does not provide IO address translation capabilities (e.g., IOMMU support such as VT-d on certain Intel x86 processors). For the purpose of this work, we use only the Yocto Linux distribution for guests that complement Quest.

In this work, DriveOS hosts Quest and Linux on two separate cores in a DX1100 machine [Cin22]. USB, USB-CAN and serial ports are exclusively allocated to Quest, and the remaining I/O devices are allocated to Linux. For Linux to receive information via USB, it must communicate with Quest through an explicit shared memory channel. Details of the inter-sandbox communication mechanism are provided in Section 3.3.

3.1.2 Real-time Task as a Service

DriveOS temporally and spatially isolates Quest and Linux in the same physical machine. This means a guest kernel is unable to interfere with the runtime progress or memory state of another guest. Quest schedules its tasks with its own real-time scheduling policy, independent of the co-existent Linux guest. In DriveOS design, Quest provides real-time task services to other sandboxes via inter-sandbox communication channels. Applications in other sandboxes subscribe to a real-time service via a specific channel. For example, a feed-forward PI controller for a car's adaptive cruise control is implemented in DriveOS as a real-time service in Quest. Linux-based ADAS functions use the real-time PI controller service via a synchronous shared-memory channel.

A real-time service task has the following properties: maximum runtime (`C`), frequency

of execution (or period `T`), and a number of communication channels. Quest schedules the real-time tasks with the rate-monotonic scheduling (RMS) algorithm [LL73]. The service task is given at least `C` time-units in every `T` time-units.

In addition, critical I/O tasks are also implemented in Quest as real-time service tasks. For different classes of I/O devices and I/O-waiting threads, unique `C` and `T` values are computed at runtime to handle interrupts at the correct priority. This is covered further in Section 3.2.1. Other sandboxes use this type of critical I/O device-handling task to implement a *real-time virtual device interface*. For example, DriveOS has a real-time CAN Gateway Service to handle USB-CAN devices in a fast and predictable way. In the future, we plan to consolidate dozens of car ECUs into real-time services in multiple Quest sandboxes and make them available to applications in other sandboxes.

3.1.3 Advantages of the DriveOS Design

The DriveOS architecture brings unique advantages in vehicle management, which are crucial to building a secure, predictable and extensible automotive system.

3.1.3.1 Real-time Task and I/O Services for Linux

In spite of being a non real-time OS, Linux is able to leverage the real-time capabilities of Quest to interface with the timing critical components of a vehicle. I/O data is exchanged with Linux applications without the need to use traditional socket-based interfaces such as Ethernet. Moreover, `SCHED_DEADLINE` scheduling of Linux enables data exchanges with Quest to perform in a sufficiently predictable way. This approach removes interference from device interrupts that are managed in real-time by Quest.

3.1.3.2 Isolated I/O for Timing Sensitive Devices

The USB-CAN interface is timing and safety-critical in automotive systems. The injection of a malicious packet onto the CAN bus has potentially devastating effects [KCR+10, Lev22], dictating the need for secure access to this network. Although malicious packet insertions must be prevented, the vehicle management system must still be able to read from and write to this bus network to receive data and control the components of a vehicle such as the HVAC unit, and engine controller.

The isolated sandboxes in DriveOS prevent unauthorized access to critical I/O devices by guests such as Linux. In our vehicle management system, the USB-CAN device is assigned to Quest and is inaccessible to Linux. CAN data is accessible to Linux only via secure shared memory channels from Quest. Quest additionally filters requests to ensure any malfunction or vulnerability in Linux will be contained within its sandbox.

3.1.3.3 Separation of Criticality Domains

Standards such as ISO 26262-3 [ISO11] define multiple automotive software integrity levels (ASIL-A to D). Our approach enables services of different criticality, or integrity, levels to be assigned to different sandboxed domains. These sandboxes are spatially and temporally isolated as envisioned by partitioning systems for future vehicles [LSOH07]. Because of the relatively small RTOS codebase, DriveOS is amenable to formal verification to ensure functional and timing correctness [GSC+16, LRS+19].

3.1.3.4 Flexibility in Automotive Software Development

If Linux is used to interface directly with an automotive system's critical functionality, it ideally needs to be independently maintained by automotive manufacturers. However, these manufacturers may not have the expertise to develop and maintain a large and com-

plex codebase like Linux.

It is potentially easier for automotive engineers to develop and maintain a simpler codebase such as Quest, which provides timing and safety-critical services to a vehicle. Quest is able to consolidate the real-time functional requirements traditionally managed by separate ECUs within different process address spaces. We have also started the development of a set of toolbox functions for Matlab that is heavily used by the automotive industry, to produce target code for Quest. These functions not only gain the benefits of a real-time OS but are guaranteed isolation from a Linux address space, which is potentially open to third-party, less secure software. Thus, it is beneficial for manufacturers to develop in a separate OS in which they have the flexibility to apply their own safety and security policies. The only Linux development that is needed is the inclusion of a Linux kernel module to facilitate inter-sandbox communication to and from Quest.

Apart from the above advantages, DriveOS is extensible for an implementation of fault-tolerant automotive software system [MWL14]. In addition, the potential single point of failure of a single hardware platform is addressed by introducing backup hardware, albeit with fewer replicas than ECUs found in current vehicles. Memory bit errors are addressed by replicating software functions using techniques such as Triple-Modular Redundancy [LV62], or N-versioning [Avi85]. Hypervisor-based fault tolerance ensures one sandboxed guest is able to recover from failure [BS95]. These techniques serve to validate our approach, but are not the main focus of this work.

3.2 DriveOS Implementation

As stated earlier, DriveOS uses Quest to bootstrap a series of sandboxed guests. Once the first instance of Quest is loaded into memory, it replicates itself for each sandbox that is specified in a static configuration. Hypervisor code extensions to standalone Quest

enable each OS replica to be launched into non-root mode, using a `VMLAUNCH` machine instruction. Depending on the system configuration, a guest may continue to run Quest or may choose to bootstrap another OS such as Linux.

Each sandbox replicates the VMM logic to establish separate guest domains, but as previously noted, this code is only required at runtime to handle instructions that cause VMExits and to establish secure communication channels between guests. Secure communication channels require mappings of guest to machine physical memory using extended page tables (EPTs). Although a VMM's text segment that stores instructions fits within a 4KB page of memory, additional data space is needed for EPTs. For example, assuming a 4GB address space for a single guest requires 24KB memory for the corresponding EPT. DriveOS currently uses Intel VT-x technology to allow a sandbox monitor to create one of more *virtual machine control structures* (VMCSs) per sandbox. One VMCS is created for each CPU core assigned to the sandbox, and stores guest and host state information, virtual machine execution, exit and entry control information, as well as the causes of VMExits. `VMLAUNCH` instructions refer to the active VMCS for the corresponding processor and initialize the corresponding guest state. If the guest is to replace Quest with another OS, configuration parameters required for paravirtualization are sent to the respective kernel at boot time.

Memory Partitioning In the configuration parameters of DriveOS, a tuple containing the base and limit of host physical memory (HPM) must be specified for every sandbox. Each sandbox monitor relocates its guest in HPM according to the specified base address. EPT entries grant guests exclusive access to specific memory regions, while safeguarding the monitor logic. VMMs also manage the guest physical to host physical memory mapping of shared memory regions for the inter-sandbox communication, as described in Section 3.3.

Device Partitioning Device partitioning is accomplished by interposing on ACPI configuration and PCI bus enumeration, thereby ensuring VMExits into a guest's corresponding VMM to check whether the device is blacklisted or not. Each guest's monitor will block their guest's access to a device or IRQ if they are not assigned to that guest. Identity-mapped MMIO regions are used by guest kernels to manage their assigned I/O devices.

The Yocto Linux kernel has been paravirtualized to compensate for the HPM base offset when a physical address is needed for DMA-enabled devices. This avoids implementing VMM drivers to support IOMMU technologies, such as Intel's VT-d, for those devices. As the code size of each VMM is minimized, this helps enforce heightened security and simplifies formal verification.

3.2.1 Real-time I/O in Quest RTOS

In Quest, every real-time task has a budget, C, determined by its worst-case execution time, and a period, T. Quest implements a static priority rate-monotonic scheduling (RMS) algorithm with a sporadic server, to guarantee a task or software thread receives at least C amount of execution time every T. Using the RMS policy, Quest assigns the highest priority level to the task with the smallest period.

Quest ensures that temporal guarantees of real-time tasks are not violated by interrupts from I/O devices. Quest handles an I/O interrupt with a *schedulable thread* at its *proper* priority level [ZW06, DLW11, Wes22]. In general, device interrupts are generated on behalf of tasks issuing I/O requests. Thus, an interrupt must be handled at the same priority level as the waiting task.

Each interrupt handler is divided into two parts: a top- and bottom-half. In Quest, the top-half handler simply acknowledges an interrupt, and determines which task is waiting for the I/O device. Quest then schedules the bottom-half handler as a separate thread at the same priority level as the waiting task. As RMS determines a task's priority by its

period, T, the bottom-half handler (BH) thread is assigned the same period as the task it serves. As with the Quest RTOS [DLW11, Wes22], the budget of the BH thread is derived from the I/O device class and the waiting task's period. Each device class has a utilization percentage value, which is multiplied by the waiting task's period, to derive the budget of the BH thread. For example, the utilization percentage of USB-CAN devices in DriveOS is 10%. If a task with a period of 1 ms waits for a USB-CAN read, then the USB-CAN BH will receive 0.1 ms budget in every 1 ms to read messages from the device. All I/O handling occurs in the context of the BH thread, not in the I/O-waiting task's context. This ensures BH processing is time-budgeted separately from task execution.

3.2.2 Paravirtualized Android Sandbox

An earlier version of DriveOS used Android in a guest sandbox [SGEW20a]. Despite Android's HAL, security and other layers, we patched Android with modifications of only 126 lines of Linux kernel code. The Android sandbox hosted third-party IC and IVI applications along with other default Android applications (apps) such as Google Chrome, Contacts. Android apps accessed the real-time CAN I/O via the shared memory channels with the Java Native Interface functions.

3.3 Inter-Sandbox Communication in DriveOS

A key component of DriveOS is the secure and predictable communication between different guest domains. Address spaces in two different guests use the `shmcomm` inter-sandbox communication mechanism to interact with each other. Inter-sandbox channels are used to create communicating task pipelines [MMO+95, PIBM11]. Figure 3.3 shows the shmcomm control- and data-flow in DriveOS.

A kernel (shmcomm) module mediates requests to map and unmap shmcomm commu-

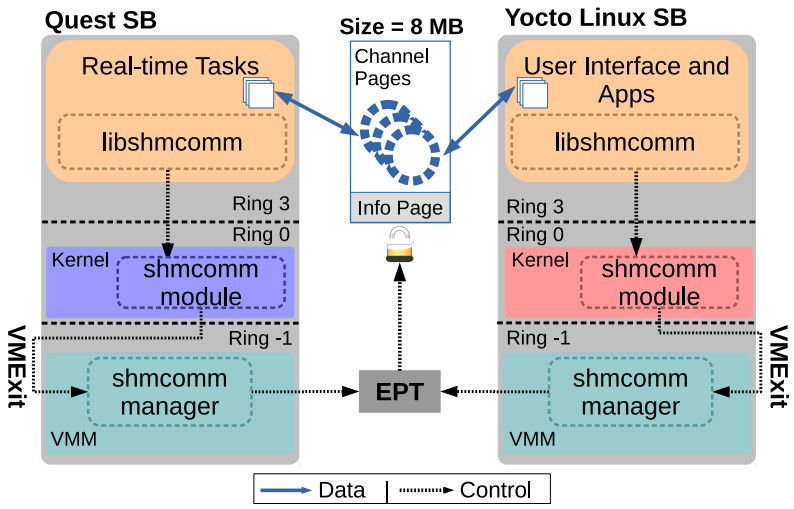


Figure 3.3: `shmcomm` Control and Data Flow in DriveOS

nication channels in an 8MB region shared between the guests. A kernel module within each guest sends requests to the shmcomm manager in its local VMM to configure the channels. The shmcomm manager does not expose the host physical addresses (HPAs) of shmcomm channels to a guest. Instead, it establishes EPT mappings of guest physical addresses (GPAs) to HPAs, for the memory pages used for communication channels. The manager uses a secure Info Page to store all the metadata information of the channels in DriveOS. The Info Page is not mapped to any guest kernels but is instead accessed via a lock held by the VMMs of each sandbox.

Each VMM shmcomm manager resides in ring -1 in DriveOS, and requires less than 500 lines of code, thereby keeping the trusted codebase of the most privileged protection domain small. In addition, the shmcomm manager handles four specific hypercalls to service requests from guest kernels: (1) creating a channel, (2) connecting a channel, (3) getting channel metadata, and (4) destroying a channel. User-level (ring 3) address spaces cannot directly interact with the VMM, unless granted permission by guest kernels.

Once a channel is created by the shmcomm manager, applications in different sandboxes communicate without invoking system calls or VMExits. Applications use a POSIX

I/O-like API provided by `libshmcomm`, to read from and write to these channels for asynchronous and synchronous communication. The channel communication protocols are entirely implemented in userspace. It is possible to extend the library with new communication protocols without modifying the kernel modules or the VMM code. Code Block 3.1 shows the APIs provided by `libshmcomm`.

```
int shmcomm_open_send (unsigned int vshm_key, unsigned int flags,
  marshall_func_t marshall_function, size_t packet_size, int buffer_len);
int shmcomm_open_receive (unsigned int vshm_key, unsigned int flags,
  unmarshall_func_t unmarshall_function, size_t packet_size, int
  buffer_len);
int shmcomm_write (int fd, void* buf, size_t nbytes);
int shmcomm_read (int fd, void* buf, size_t nbytes);
unsigned int get_vshm_key (int fd);
channel_transfer_type_t get_channel_transfer_type (int fd);
int shmcomm_destroy (int fd);
int shmcomm_close (int fd);
```

Code Block 3.1: The `libshmcomm` Library APIs

3.3.1 `shmcomm` Operations

3.3.1.1 Creating and Connecting a Channel

Channels are created and connected for sending and receiving messages using `shmcomm_open_send` and `shmcomm_open_receive` functions, respectively. A unique channel key, `vshm_key`, is used to create a channel between separate guest address spaces. The `flags` argument supports the creation of a new channel (`SHMCOMM_CREATE_CH`) or connection to an existing channel (`SHMCOMM_CONNECT_CH`). The channel type, which is either synchronous (`SHMCOMM_SYNC_CH`) or asynchronous (`SHMCOMM_ASYNC_CH`), is also specified in `flags`.

The shmcomm protocol supports marshalling and unmarshalling before sending and after receiving messages, using specific callback functions. Data marshalling is provided as a convenience because CAN messages often need to be encoded and decoded (e.g., using DBC files). Finally, the size of each message (or packet) and the length of the shared

memory buffer are needed to create a channel. All this information, except the marshalling callbacks, are sent to the shmcomm manager via a system call to the guest kernel and a hypercall to the VMM. The shmcomm manager stores the channel information in the secure Info Page. Connecting to an existing channel does not need any specific information such as `packet_size` or `buffer_len`, as shmcomm supplies this information to the applications from its Info Page. The shmcomm kernel module in a guest sandbox sends a GPA to the shmcomm manager, which is mapped to a private channel HPA. Both `shmcomm_open_send` and `shmcomm_open_receive` return an integer file descriptor that is used to further interact with the channel.

3.3.1.2 Closing and Destroying a Channel

Closing a channel with `shmcomm_close` frees the userspace data-structures for the channel. `shmcomm_destroy` frees the channel memory from the shared memory region and also removes the channel entry from the Info Page. `shmcomm_destroy` implicitly closes a channel.

3.3.1.3 Querying Channel Metadata

A channel's `vshm_key` and type of data-transfer (synchronous or asynchronous) are queried using `get_vshm_ key` and `get_channel_transfer_type`, respectively. An address space could query channel metadata before exchanging messages with a channel.

3.3.1.4 Reading from and Writing to a Channel

Reading from and writing to a channel occur entirely in guest userspace without any system call or VMExit overheads. The channel memory is mapped to userspace in the local (guest process) page table by the shmcomm kernel modules in Linux and Quest. This

reduces communication overheads in DriveOS once channels are established. The shmcomm interface uses a FIFO ring buffer for synchronous communication and Simpson's four-slot algorithm [Sim90] for wait-free asynchronous message passing. The latter is useful when loss tolerant data transfers between guests are acceptable, as long as the most recent data is exchanged (e.g., for sensor data).

3.3.1.5 Example

Code Block 3.2 shows the C code of a sender in Quest that creates a synchronous shmcomm channel to send messages of `struct packet` type. The channel's ID is `101` and ring buffer length is `10`. Code Block 3.3 shows the receiver-side code in Linux, which connects to channel ID=`101` and reads from the channel into a local variable `p_rec`.

```
struct packet {int X; int Y};
int write_fd = shmcomm_open_send(101,
        SHMCOMM_CREATE_CH |
        SHMCOMM_SYNC_CH, NULL,
        sizeof(struct packet), 10);
struct packet p_snd = {.X = 10,
        .Y = 20};
shmcomm_write(write_fd, &p_snd,
sizeof(struct packet));
shmcomm_close(write_fd);
```

Code Block 3.2: Sender in Quest

```
struct packet {int X; int Y};
int read_fd = shmcomm_open_receive(101,
        SHMCOMM_CONNECT_CH, NULL,
        sizeof(struct packet), 10);
struct packet p_rec;
shmcomm_read(read_fd, &p_rec,
sizeof(struct packet));
shmcomm_close(read_fd);
```

Code Block 3.3: Receiver in Linux

3.3.2 Real-time Virtual Device I/O for Linux

Communication pipelines created with our `libshmcomm` library extend real-time I/O in Quest to address spaces in Linux. This enables Linux tasks to perform time-bounded functions such as obstacle detection and avoidance (useful for ADAS), using real-time sensor data processing and actuation tasks in Quest. Our communication APIs have bindings for C, C++, Java and Python in Linux and Android. DriveOS grants permission for IC and IVI tasks in C++, and OpenPilot ADAS tasks in C++ and Python, to interact with USB-CAN I/O services in Quest.

3.4 DriveOS Applications

In this section, we describe the integration of three applications in DriveOS: Instrument Cluster, In-vehicle Infotainment and Advanced Driver Assistance System. We also explain how these applications utilize the *Real-time Task as a Service* model in DriveOS, to guarantee end-to-end throughput and delay requirements for data processing. Since these applications are tested in our hardware-in-the-loop (HIL) simulation infrastructure, we first explain the HIL simulation setup to help describe the integration procedure. The HIL setup is also used in Section 3.5 to evaluate DriveOS against standalone Linux.

3.4.1 Hardware-in-the-loop Experimental Infrastructure

DriveOS is prototyped and tested on a Cincoze DX1100 machine [Cin22], featuring an Intel Coffee Lake i7-8700T processor. This is a low-power industrial PC-class machine that is being used in an electric vehicle under development by our partner company. It has ample processing capacity to support many traditional ECU functions as software threads. Multiple I/O interfaces are capable of interfacing with different USB-CAN networks, and three display ports serve different user interfaces. Table 3.1 lists the machine features.

Table 3.1: DX1100 Specifications

Processor	Intel Coffee Lake i7-8700T ($\leq 2.4GHz$)
RAM	32 GB
eMMC Storage	64 GB
Display and UI	HDMI, DVI and DisplayPort
CAN Connector	8 USB3.x ports
Serial I/O	4 RS-232 Serial ports
Network	2 GbE Ethernet ports (x2) and mPCIe-USB Bluetooth
Power	24V, 5A
Dimension	242 mm $\times$ 174 mm $\times$ 77 mm

Figure 3.4 shows the HIL setup and data-flow via our car's computing hardware. A Kvaser USBcan Pro 5x HS industrial USB-CAN adapter [Kva22] is attached to the

DX1100. The CAN-Hi and CAN-Lo signals from the adapter are suitably terminated with 120Ω resistive loads. These signals feed into a USBcan Light 2x HS USB-CAN adapter attached to an Ubuntu 16.04 Linux machine, which runs the CARLA [DRC+17] driving simulator. The simulator feeds a CAN bus data trace of our partner company's electric car to test onboard IC and IVI application services. To test the ADAS services, the CARLA simulator is updated via OpenPilot running in DriveOS. Before we deploy our system fully on the road, it is necessary to rigorously test and study time-critical metrics using a HIL simulation.

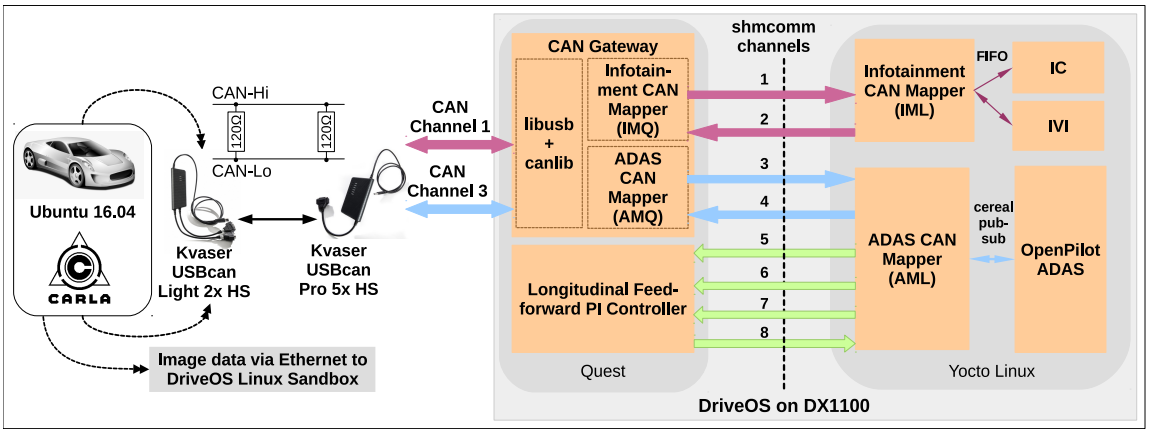


Figure 3.4: Hardware-in-the-loop Simulation Infrastructure for DriveOS

3.4.2 Instrument Cluster (IC) and In-vehicle Infotainment (IVI)

The IC and IVI are third-party Qt C++ applications being developed by a partner company, with sample screenshots shown in Figure 3.1. The IC and IVI rely on sophisticated UI libraries such as Qt, that are only supported for a selected few OSes. Therefore, it is not feasible to implement these applications in an RTOS like Quest, even though they have some critical timing requirements. DriveOS refactors these applications, so that timing-critical components are ported to Quest and the remaining parts run in Linux.

For IC and IVI, sending and receiving CAN messages needs to be fast and predictable. Slow and unpredictable CAN messaging would mean inaccurate and discrepant data in IC and IVI. DriveOS therefore features a CAN Gateway real-time service in Quest that delivers CAN packets to the IC and IVI applications via shmcomm channels. An Infotainment CAN Mapper is split between Quest (IMQ) and Linux (IML) to exchange data (see Figure 3.4). In Linux, FIFO pipes are used to deliver messages between IC, IVI and Infotainment CAN Mapper threads. More details about the CAN Gateway real-time service task are explained later in this section.

3.4.3 OpenPilot Advanced Driver Assistance Systems (ADAS)

DriveOS also incorporates an open-source ADAS, OpenPilot [com20], which is used daily in thousands of cars on the road. OpenPilot is written in C++ and Python and originally developed for Ubuntu and Android. It supports Adaptive Cruise Control, Automated Lane Centering, Forward Collision and Lane Departure Warning. OpenPilot receives radar, gyro and other sensor data via CAN. Live images are collected via local cameras, while simulated CARLA images are received over an Ethernet link managed by Linux. OpenPilot then generates throttle, brake and steering control adjustments based on a Longitudinal PI Controller, and machine-learning (ML)-based Object Detection and Path Planning algorithms. It uses Tensorflow for ML algorithms, and Qt for a UI display.

After OpenPilot decides an intended path by processing an image stream with the Object Detection and Path Planning algorithms, its Longitudinal Feed-forward PI Controller is responsible for generating the final throttle and brake control values. Such a longitudinal controller is central to an automotive system's safety, and is susceptible to timing violations. In general, the automotive industry expects an end-to-end (sensing-processing-actuation) delay in the order of `10ms` for such controllers [RIM16].

OpenPilot currently runs the longitudinal controller as a `SCHED_FIFO` task and main-

tains a 10ms rate via the Linux `clock_gettime` API. Current OpenPilot implementations run on a dedicated Linux machine where no other applications are allowed to run, and all tasks are hand-tuned to meet their timing requirements. As the automotive industry moves towards an *integrated and extensible* vehicle management solution, arbitrary third-party applications in Linux have the potential to interfere with the timing requirements of a controller [AGK+02] such as the one in OpenPilot. Our experiments in Section 3.5 reveal this issue.

In an effort to port OpenPilot to DriveOS, the Longitudinal Controller is implemented in a Quest sandbox as a real-time service task. In addition, the CAN Gateway real-time service task is also utilized by OpenPilot for radar data inputs and brake, throttle and steering outputs. The rest of the OpenPilot module relies on Tensorflow, which is only available on systems such as Linux, Windows, and Mac OS, and on Qt which is also limited to a few OSes. As porting this part of OpenPilot to an RTOS would require a significant effort, it is instead deployed unchanged in the paravirtualized Yocto Linux sandbox of DriveOS. These modules receive a stream of simulated CARLA images over Ethernet directly in Linux. Interactions between the Longitudinal Controller and the rest of the ADAS software are facilitated by inter-sandbox shmcomm channels. Thus, the DriveOS design enables a cross-sandbox implementation of OpenPilot with real-time components running in an RTOS and legacy, library-dependent components running in Linux.

3.4.4 Real-time Service Tasks

The above three DriveOS applications use two real-time service tasks, explained below. These tasks are based on the Real-time Task as a Service model explained in Section 3.1.2. Table 3.2 shows all the shmcomm channels between the real-time service tasks in Quest and applications in Linux.

Table 3.2: Shared Memory Channels in DriveOS

ID	Description	Data-flow (tasks)	Data-flow (sandboxes)	Type	Buffer size
1	IC, IVI Sensor Reading	IMQ → IML	Quest → Linux	Sync	10
2	HVAC (IVI) Control Actuation	IML → IMQ	Linux → Quest	Sync	10
3	CARLA Sensor Reading	AMQ → AML	Quest → Linux	Async	-
4	CARLA Vehicle Control Actuation	AML → AMQ	Linux → Quest	Sync	10
5	LongController Control Command	AML → LongControl.	Linux → Quest	Sync	10
6	LongController Control Data	AML → LongControl.	Linux → Quest	Sync	10
7	LongController Update Input	AML → LongControl.	Linux → Quest	Sync	10
8	LongController Update Output	LongControl. → AML	Quest → Linux	Sync	10

3.4.4.1 CAN Gateway Service

A CAN Gateway Service in Quest mediates real-time CAN messages for Linux applications, enabling a *real-time virtual CAN device interface*. We start describing this service from the right-hand side of Figure 3.4. A Linux application in DriveOS interfaces with a car CAN bus network using a CAN mapper process in Linux. Every CAN mapper process has one thread each for reading and writing CAN messages via shmcomm read and write channels. Each CAN mapper thread in the Linux sandbox interacts with a counterpart in the Quest sandbox via a specific shmcomm channel.

The CAN mapper threads in Quest (IMQ and AMQ in Figure 3.4) are part of the CAN Gateway Service. Program 3.4 shows how the CAN mapper threads (CAN Readers and Writers) are spawned in the Quest sandbox for `NUM_CAN` number of CAN Channels. Acting as a CAN concentrator, the CAN Gateway Service reads CAN messages from different CAN Channels of the Kvaser USBcan Pro 5x HS. We use a real-time USB xHCI (3.0) bus scheduling algorithm [GCW18] in the interrupt bottom-half handler in Quest for fast and predictable CAN I/O. The Gateway then forwards CAN messages to appropriate shmcomm channels. Program 3.5 shows the C code of a CAN reader thread that reads CAN messages from a CAN Channel and forwards to Linux via a shmcomm channel.

```
#define NUM_CAN 2
#define CAN2LINUX_VSHM_KEY0 101
#define LINUX2CAN_VSHM_KEY0 201

int can2linux_channel[NUM_CAN], linux2can_channel[NUM_CAN];

// Structure to pass arguments to the Reader and Writer Threads
typedef struct gate_th_args {
        int can_ch; int can2linux_fd; int linux2can_fd;
        int can_open_flags; int can_freq;
        int can_hnd; int budget_us; int period_us;
} gate_th_args_t;
gate_th_args_t* thr_args = malloc(sizeof(gate_th_args_t) * NUM_CAN);

// Structure to represent a CAN message
typedef struct can_packet {
        uint32_t id; int32_t can_msg_id; unsigned char can_msg[8];
        unsigned int can_dlc;
        unsigned long can_timestamp;
} can_packet_t;

pthread_t can2linux_th[NUM_CAN], linux2can_th[NUM_CAN];

for (i = 0; i < NUM_CAN; i++) {
        // Set the arguments for a CAN Reader Thread
        thr_args[i].can_ch = i;
        thr_args[i].can_open_flags = canOPEN_EXCLUSIVE;
        thr_args[i].can_freq = canBITRATE_500K;
        thr_args[i].can2linux_fd = can2linux_channel[i];
        thr_args[i].budget_us = 100;
        thr_args[i].period_us = 2000;

        // Set up a shmcomm channel from Quest to Linux for a CAN Reader thread
        int shm_key = CAN2LINUX_VSHM_KEY0 + i;
        can2linux_channel[i] = shmcomm_open_send(shm_key,
                SHMCOMM_CREATE_CH | SHMCOMM_SYNC_CH, NULL,
                sizeof(can_packet_t), 10);

        // Spawn a CAN Reader Thread to read data from CAN Channel i
        // and send to Linux via the above shmcomm channel
        pthread_create(&can2linux_th[i], NULL, can2linux_task,
                &thr_args[i]);

        // Set the arguments for a CAN Writer Thread, by only changing
        // the shmcomm channel and keeping everything else same from the Reader
        // thread
        thr_args[i].linux2can_fd = linux2can_channel[i];

        // Set up a shmcomm channel from Linux to Quest for a CAN Writer thread
        shm_key = LINUX2CAN_VSHM_KEY0 + i;
        linux2can_channel[i] = shmcomm_open_receive(shm_key,
                SHMCOMM_CREATE_CH | SHMCOMM_SYNC_CH,
                NULL, sizeof(can_packet_t), 10);

        // Spawn a CAN Writer Thread to write data from Linux to CAN Channel i
        pthread_create(&linux2can_th[i], NULL, linux2can_task,
                &thr_args[i]);
}
```

Code Block 3.4: CAN Reader and Writer in DriveOS CAN Gateway

Spawning CAN Reader and Writer Threads in the CAN Gateway in Quest

Upon receiving messages from a shmcomm channel, the CAN mapper threads in Linux (IML and AML in Figure 3.4) forward them to appropriate applications based on CAN message IDs. Similarly, Linux applications write to CAN channels in the reverse direction. A CAN writer thread in the CAN Gateway is shown in Program 3.6.

We have two CAN mappers in the Gateway service for Infotainment (IC and IVI) and ADAS. Figure 3.4 has color-coded CAN and shmcomm channels to show the CAN data-flow for Infotainment (violet) and ADAS (blue).

```
void* can2linux_task (
        void* thread_args) {
gate_th_args* args = (gate_th_args*)
  thread_args;

// Create a sporadic server in Quest
// corresponding to thread
struct sched_param s_params = {
  .type = MAIN_VCPU,
  .C =args->budget_us,
  .T = args->period_us
};
int new_vcpu = vcpu_create(&s_params);
vcpu_bind_task(new_vcpu);

// Open a CAN Channel
args->can_hnd = canOpenChannel (
  args->can_ch, args->can_open_flags);
canStatus can_stat = canSetBusParams (
  args->can_hnd, args->can_freq, 0,
  0, 0, 0, 0);
can_stat = canBusOn(args->can_hnd);

can_packet_t msg;
unsigned int can_flag, can_dlc;
unsigned long can_time; long can_id;
unsigned char can_msg[CAN_MAX_DLEN];

while(1) {
  // Read from CAN Channel
  can_stat = canRead (args->can_ch,
    &can_id, &can_msg, &can_dlc,
    &can_flag, &can_time);

  //Send to Linux via shmcomm channel
  msg.id = ++cnt;
  msg.can_msg_id = can_id;
  msg.can_dlc = can_dlc;
  msg.can_timestamp = can_time;
  memcpy(msg.can_msg, &can_msg, 8);
  while(shmcomm_write(
    args->can2linux_fd,
    &msg, sizeof(can_packet_t)) <= 0);
}
}
```

Code Block 3.5: CAN Reader Thread in CAN Gateway

```
void* linux2can_task (
  void* thread_args) {
gate_th_args* args = (gate_th_args*)
  thread_args;

struct sched_param s_params = {
  .type = MAIN_VCPU,
  .C =args->budget_us,
  .T = args->period_us
};
int new_vcpu = vcpu_create(&s_params);
vcpu_bind_task(new_vcpu);

args->can_hnd = canOpenChannel (
  args->can_ch, args->can_open_flags);
canStatus can_stat = canSetBusParams (
  args->can_hnd, args->can_freq, 0,
  0, 0, 0, 0);
can_stat = canBusOn(args->can_hnd);

can_packet_t msg;
unsigned int can_flag, can_dlc;
unsigned long can_time; long can_id;
unsigned char can_msg[CAN_MAX_DLEN];

while(1){
  // Read from shmcomm channel
  while(shmcomm_read(args->linux2can_fd,
  &msg, sizeof(can_packet_t)) <=0);

  // Check if this is a valid message
  // and write to the CAN Channel
  canWrite(args->can_ch,
  msg.can_msg_id, msg.can_msg,
  msg.can_dlc, msg.can_timestamp);
}
}
```

Code Block 3.6: CAN Writer Thread in CAN Gateway

The IC and IVI use the synchronous shmcomm channel 1 to read sensor inputs such as speed, engine control type (all-wheel-drive or rear-wheel-drive), temperature inside a car, distance traveled, and so forth. The IVI application sends HVAC control commands via another synchronous shmcomm channel 2 to CAN channel 1. The IC is a read-only application and does not send any CAN messages.

OpenPilot uses the asynchronous shmcomm channel 3 to read the most recent CARLA simulator sensor readings (vehicle speed and angle) and car cruise button status (1 = initialize cruise control, 2 = increase acceleration, 3 = decrease acceleration, 4 = cancel cruise control). OpenPilot computes throttle and brake values by a longitudinal PI controller and applies to CARLA via synchronous shmcomm channel 4 and USB-CAN.

3.4.4.2 ADAS Longitudinal Controller Service

As stated in Section 3.4.3, OpenPilot uses a feed-forward PI longitudinal controller for adaptive cruise control. The refactored implementation of OpenPilot in DriveOS runs the controller in Quest as a synchronous real-time service task. It has a 50 μs budget and 1 ms period, which is the same as in the stock OpenPilot.

For test purposes, CARLA simulator sensor data is delivered to the Longitudinal Controller from the CAN Gateway service via ADAS CAN Mapper threads in Quest and Linux. Section 3.5.2 explains the ADAS Controller pipeline in details. The ADAS Mapper thread in Linux (AML) is responsible for filtering sensor data and feeding it to the Longitudinal Controller in Quest via shmcomm channels (ID 5, 6, 7). AML also receives throttle, brake and other control values from the controller, and sends CAN Control commands to CARLA via shmcomm channel 4. The Linux-side implementation of OpenPilot uses its own cereal publisher-subscriber messaging framework to obtain controller values from AML, which are used for vehicle path and control planning.

Our DriveOS Longitudinal Controller depends on the control commands that it receives via the shmcomm channel ID 5. It supports three control commands: (1) `INIT` for initializing the Longitudinal Controller values such as the proportional, integral, and feed-forward constants, (2) `RESET` to reinitialize the PI loop, and (3) `UPDATE` to compute the controller throttle and brake output by the PI loop. The controller command data for `INIT` and `RESET` is sent via shmcomm channel 6. Both channels 5 and 6 are synchronous channels because missing a control command is forbidden. The `UPDATE` data is separately exchanged via shmcomm channels 7 and 8.

3.5 Evaluation

Linux is commonly used in infotainment systems by popular automotive companies such as BMW [Lin19] and Toyota [Den17], and as the basis of Ubuntu and Android distributions used with the OpenPilot ADAS software [com20]. DriveOS is therefore compared against a standalone `PREEMPT_RT` [RMF19] patched Yocto Linux for use in integrated vehicle management systems.

In the standalone Yocto Linux VMS, software threads and interrupts are not assigned to any specific cores. For comparison, an *optimized version* of the same standalone Yocto Linux is tested. This version pins xHCI interrupts to Core 0, resulting in USB bottom-half processing taking place on the same core, while all other threads execute on Core 1. In summary, experiments test both an unoptimized and domain-optimized standalone Linux against DriveOS.

In our experiments, DriveOS uses two cores although the system is capable of activating more - one is dedicated to Quest and the other is given to a paravirtualized Yocto Linux with the `PREEMPT_RT` patch enabled. DriveOS and the standalone Linux versions are both tested in the HIL simulation infrastructure described in Section 3.4.1, using a real car dataset collected from the Laguna Seca raceway in California (for IC and IVI) and CARLA (for ADAS). For fair comparisons with DriveOS, all standalone Linux systems run on a DX1100 and implement the equivalent CAN Gateway, Infotainment, OpenPilot ADAS, IC and IVI logic as shown for DriveOS in Figure 3.4.

3.5.1 Application Parameters

Table 3.3 shows the real-time task budgets and periods for the Quest real-time USB-CAN interface (USB xHCI Bottom-half handler and `mhydra` USB-CAN driver), CAN Gateway and longitudinal controller real-time service task. Linux-side timing critical tasks are

run with the SCHED_DEADLINE scheduling policy. We also run non-timing-critical background tasks in Linux that log data in the eMMC storage and periodically send data over the network, as is common in modern cars [TE 18]. Next, we describe our experimental results. All experiments were run and averaged over five times.

Table 3.3: Real-time Task Budgets and Periods

ID	Task	Budget (ms)	Period (ms)
Quest			
A	USB Bottom-half Handler (BH)	0.10	1
B	mhydra_rx	0.20	1
C	Infotainment read mapper	0.10	2
D	Infotainment write mapper	0.10	2
E	ADAS read mapper	0.10	2
F	ADAS write mapper	0.10	2
G	Longitudinal Controller	0.05	1
H	mhydra_tx	0.20	1
Linux			
I	Infotainment read mapper	0.10	1
J	Infotainment write mapper	0.10	1
K	ADAS read mapper	0.10	1
L	ADAS write mapper	0.10	1

Table 3.4: Delay Between Consecutive CAN Messages (CAN Channel 1)

System	Average Delay
Raw CAN Frame-based	
Source (Hardware)	2.85 ms
Linux	3.73 ms
Optimized Linux	3.43 ms
DriveOS	2.86 ms
CAN ID-based	
Source (Hardware)	82.5 ms
Linux	98.97 ms
Optimized Linux	94.38 ms
DriveOS	83.25 ms

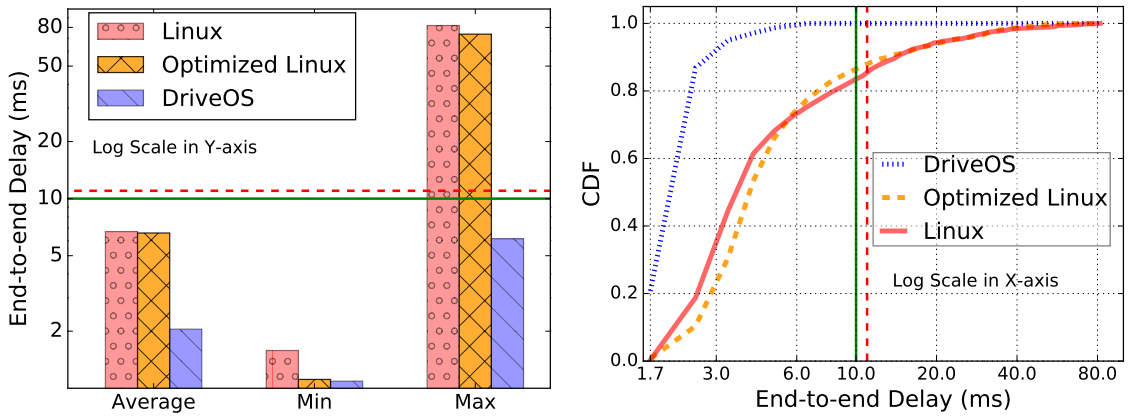
3.5.2 Latency Measurements

We measure two types of latency values: end-to-end delay and delay between consecutive messages. Maximum end-to-end delay gives us an upper bound on the round-trip-time of a sensor input and a corresponding actuator output. Such end-to-end latency is critical for ADAS services, which need to apply throttle and brake changes within a certain time for safety. Hence, we measure latency on the ADAS controller pipeline, while IC and IVI process CAN messages.

The task pipeline in the ADAS controller is as follows (with task IDs from Table 3.3 shown in parentheses): `USB BH` (A) → `mhydra_rx` (B) → `ADAS read mapper (Quest)` (E) → `ADAS read mapper (Linux)` (K) → `OpenPilot Longitudinal Controller` (G) → `ADAS write mapper (Linux)` (L) →

`ADAS write mapper (Quest)` (F) → `mhydra_tx` (H) → `USB BH` (A). The theoretical worst-case end-to-end delay in a pipeline is the summation of the periods of all the tasks [GSW20], assuming input data is available only at the beginning of a period of the pipeline's source task. Therefore, the theoretical end-to-end delay bound for the controller pipeline is: $(T(A)+T(B)+T(E)+T(K)+T(G)+T(L)+T(F)+T(H)+T(A)) = (1+1+2+1+1+1+2+1+1) = 11$ms.

This theoretical end-to-end delay bound is in the ballpark of what is expected in a working automotive system (∼10ms) [RIM16]. Empirical results show that DriveOS performs much better than both the expected and theoretical worst-case delays.



(a) Average, Minimum, Maximum **(b)** Cumulative Distribution Function

Figure 3.5: Controller Pipeline End-to-end Delay

Figure 3.5a (log scale) shows the average, minimum and maximum end-to-end delays of the controller pipeline in standalone Linux, optimized Linux and DriveOS. Figure 3.5b shows the corresponding cumulative distribution function (CDF) of the delays. The experiments are run with 20 non-critical background threads occupying almost 60% CPU utilization in Linux. Such background processes are representative of third-party applications (e.g., Spotify, Maps, and Data Backup). In our experiments, these tasks send data over an Ethernet network via TCP socket connections, and copy backup logs to storage.

The device interrupts generated by these background tasks are intended to reveal potential interference on timing critical tasks.

In Figures 3.5a and 3.5b, the theoretical worst-case (`11ms`) and industry expected (`10ms`) delays are respectively shown with a dashed and solid line. Although Linux performs on average within bounds, the maximum delay is well above what is allowed. In Figure 3.5b, the CDF of delays shows that more than 15% of the end-to-end latencies are greater than 10ms in both standalone Linux versions. This could lead to an unsafe implementation of ADAS services and make the system unfavorable to regulatory authorities. DriveOS performs well within industry standards and theoretical bounds for average, minimum, and maximum end-to-end delays for a safe implementation of ADAS.

The median latencies in every 10 CAN frames in Figure 3.6 further reveals the unpredictable and inconsistent latency in Linux. It also shows that DriveOS has a very low end-to-end delay variation. Even though Linux does not have the additional CAN mapper threads of Quest, it performs badly because it lacks a timing-predictable interrupt handling mechanism. Optimized Linux improves the delay slightly because xHCI interrupts are pinned to Core 0. However, Linux's bottom-half processing of other interrupts on Core 1 is still able to interfere with the execution of more important `SCHED_DEADLINE` threads on that core. Quest correctly matches the scheduling priority of the I/O bottom-half handler with the thread waiting on I/O. Additionally, DriveOS refactors the longitudinal controller logic of OpenPilot to Quest, which provides temporal isolation between tasks and interrupts for time-critical tasks. Consequently, DriveOS achieves $\frac{1}{12}^{th}$ the maximum end-to-end delay observed in Linux.

In another set of experiments, we measure the delay between consecutive CAN frames in CAN Channel 1. Table 3.4 shows two types of average delay over 5000 CAN frames: (1) delay between consecutive arbitrary CAN frames and, (2) delay between consecutive

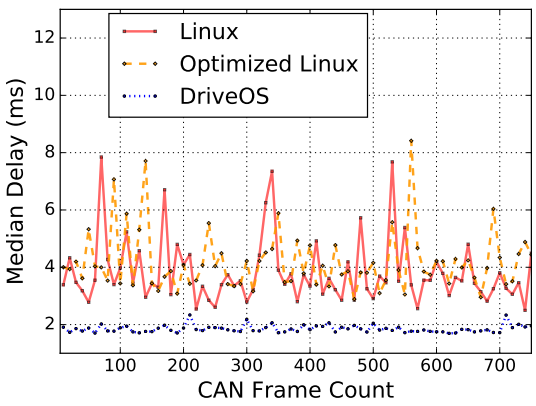


Figure 3.6: Controller Pipeline Median Delay per 10 Frames

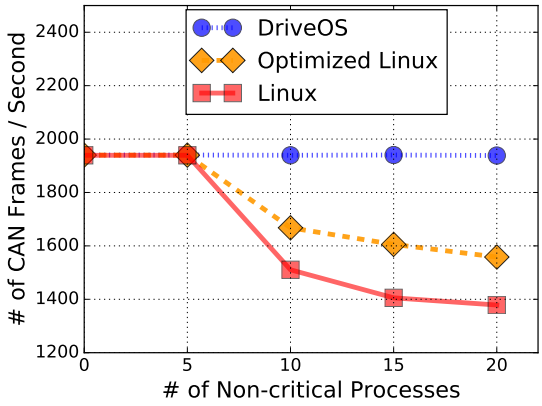


Figure 3.7: Infotainment (IC and IVI) Throughput

frames of the same CAN ID. The source row shows the delay at the CAN message generator on the Ubuntu 16.04 simulator machine. The delay at source is representative of the delay observed at the sensor and actuator hardware. We see that DriveOS receives messages with a similar average delay. However, both versions of standalone Linux receive raw CAN messages delayed by 20-30%. Similar behavior was observed for CAN ID-based delays. This shows that DriveOS introduces negligible latency overhead on top of a CAN hardware source for a real car's CAN dataset, especially in comparison to Linux used in the automotive industry. Therefore, ECU hardware could be safely and predictably replaced by real-time software service tasks in DriveOS, where sensor readings and actuator outputs are communicated via CAN messages.

3.5.3 Throughput Measurements

In this experiment, we test the throughput from CAN Channel 1 to the IC and IVI applications. We measure the throughput at the end of IML in Figure 3.4 before forwarding the data to IC and IVI. Higher throughput means that IC and IVI tasks show more accurate and informative data on the car displays. Figure 3.7 shows average CAN frames per second in a period of 3 minutes, with increasing number of non-timing-critical processes in Linux.

These background processes log CAN frames and make copies of data for safety. In a deployed system multiple third-party applications would actually be running as background threads.

Although Linux performs similar to DriveOS in the absence of any background threads, its performance drops as the number of such threads increases. These non-critical threads increase the number of device interrupts in Linux, and Linux fails for the same reasons stated in the earlier subsection. Optimized Linux performs a little better because xHCI interrupts are delivered to Core 0. However, DriveOS performs consistently better, and independently of the background threads because Quest's USB-CAN I/O handling is not disrupted by the background threads in the Linux sandbox. Furthermore, DriveOS's better performance is especially significant because it has a longer pipeline, traversing through a virtualized Quest sandbox and shmcomm shared-memory channels, that are absent in Linux. This shows the benefits that DriveOS's I/O handling and inter-sandbox communication mechanism provide.

Table 3.5 shows the average throughput and standard deviation of IC and IVI CAN message reading (at IML in Figure 3.4), and OpenPilot CAN message writing (at AML in Figure 3.4). DriveOS achieves higher throughput and better predictability with lower standard deviations. Table 3.5 also shows the throughput data for our version of OpenPilot in Linux, which communicates to CARLA via Ethernet. Although the throughput is worse than DriveOS's performance with CAN, it is similar to standalone Linux's USB-CAN throughput. This shows that a timing-sensitive implementation of Ethernet could be an alternative to a CAN bus network in future automotive systems.

Scaling Time-Critical Processes In the next set of experiments, we test the scalability of critical processes, which are representative of ECU functions implemented as software services. These processes read from and write to the CAN interface ($C = 20\mu s, T = 20ms$).

Increasing the number of such processes should not affect the Infotainment, ADAS and other car services. Figure 3.8 shows the throughput of infotainment services while running 0–15 time-critical processes in the system. The throughput stays the same in DriveOS against increasing number of critical processes, as they are run as real-time services in Quest. In spite of running time-critical processes as `SCHED_DEADLINE` tasks, the drop in infotainment throughput shows that Linux does not scale against time-critical tasks that access the USB-CAN ("CAN I/O", yellow line), and disk and Ethernet devices ("Other I/O", green line). Hence, it is not a favorable choice for future ECU consolidation.

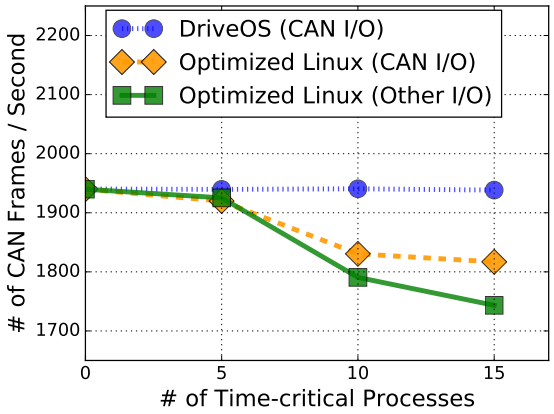


Figure 3.8: Throughput against Time-critical Processes

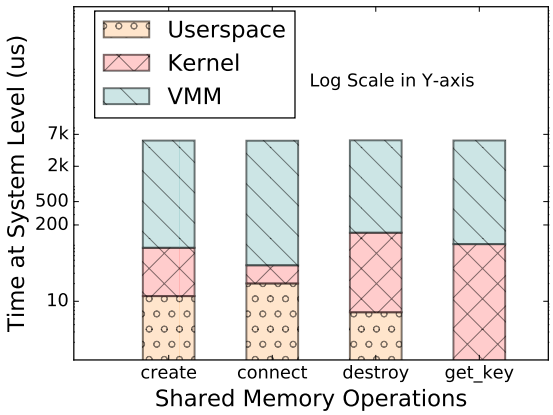


Figure 3.9: Cost of `shmcomm` Operations in Quest

3.5.4 Startup Times

The system and application startup times are important factors for the end-users of a vehicle management system. The next set of experiments investigate whether the paravirtualization of Yocto Linux in DriveOS has any significant effect on either Linux or the applications' startup times.

The average over five cold boots is noted, where the system is initially powered down. The DriveOS paravirtualized Yocto Linux takes 18.89 seconds to boot and start the Linux

Table 3.5: Throughput with 20 Background Processes

System	Average (frame/sec)	Standard Deviation
IC and IVI (CAN Channel 1)		
Linux	1378.6	769.5
Optimized Linux	1558.3	632.5
DriveOS	1938.4	29.36
OpenPilot (CAN Channel 3)		
Optimized Linux (Ethernet)	17.45	0.91
Optimized Linux (CAN)	17.56	1.40
DriveOS	19.94	0.25

Table 3.6: Average Cost of shmcomm Channel Operations

System	Userspace (μs)	Kernel (μs)	VMM (μs)
create			
Quest	12	69	5405
Linux	30	287	
connect			
Quest	20	21	5407
Linux	40	252	
destroy			
Quest	6.5	140	5398
Linux	6.8	391	
close			
Quest	6.5	-	
Linux	6.8		
read			
Quest	0.01	-	
Linux	0.01		
write			
Quest	0.03	-	
Linux	0.03		

shell at the serial port. In comparison, a standalone Linux takes 17.56 seconds to boot. The extra time to boot the paravirtualized Linux is the time Quest takes to boot itself before executing the boot logic of Linux. The IC and IVI application takes 674ms to start in DriveOS, whereas it is 632ms in standalone Linux. The almost negligible additional time in starting the IC and IVI in DriveOS is because of the overhead of establishing the inter-sandbox shmcomm channels. This is studied in the next section. In subsequent work, DriveOS uses ACPI power management techniques to suspend to, and resume from, RAM. A suspended system is shown to consume minimal power but is able to resume critical services in several hundred milliseconds. The details of how this works are out of the scope of this paper.

3.5.5 Inter-sandbox Communication Overhead

For all shmcomm operations, we have measured the cost at three system levels in DriveOS: (1) Guest OS userspace (ring 3 in x86), (2) Guest OS kernel (ring 0) and, (3) VMM or hypervisor (ring -1). Userspace and kernel level measurements are performed separately for Quest and Linux. VMM measurements are common for both Quest and Linux. Table 3.6 shows the average cost of channel operations at different system levels in Quest and in Linux. Creating, connecting and destroying a channel comes with a higher cost because we need to make an expensive hypercall (VMExit) to the VMM for these operations. The major time is thus spent in the VMM. Figure 3.9 shows the cost of channel operations in Quest on a log scale. It reveals how most time is spent in the VMM logic for these operations. In addition, the Linux kernel incurs more overhead in channel operations than Quest. For example, creating a channel takes 287μs in Linux, whereas it is 69μs in the Quest kernel.

Once shared memory channels are established, the costs of reading, writing and closing a channel are negligible, because the channel memory is already mapped to the userspace application. With careful time-budgeting of channel endpoints, the DriveOS inter-sandbox communication mechanism achieves fast and predictable runtime reads and writes.

Chapter 4

Model-based Multi-domain Application Framework

Many vehicle functions such as heating, ventilation and air conditioning (HVAC) or powertrain control are developed for simple, single-core ECUs. These ECUs are managed by a simple RTOS or firmware. Automotive engineers design functions for these ECUs without much knowledge of advanced computer systems constructs like control flow (e.g., threads), data structures, and low-level communication primitives. They prefer model-based design languages like Simulink and LabView [Fri06]. However, these languages lack support for multi-OS domain systems, and advanced programmers are needed to port the ECU functions to a next-generation VMS like DriveOS.

We introduce ModelMap [SFW22], a **model**-based **m**ulti-domain **app**li-cation development framework for automotive functions in DriveOS. ModelMap implements a set of Simulink interfaces that target multiple sandboxes, or OS-level protection domains, in DriveOS. It provides a Simulink interface to bind a task to a real-time periodic thread in DriveOS for timing-predictable execution. It enables `shmcomm` inter-task communication primitives in Simulink. ModelMap also supports real-time I/O for commonly used protocols such as controller area network (CAN) bus. Mixed-criticality vehicle functions in multiple OS domains are encapsulated as *nested binaries* with the support of executable code for multiple application binary interfaces.

We present two Simulink automotive software models and implement them using ModelMap Simulink blocks in DriveOS. These models are: (1) a generic CAN Gateway service, which delivers CAN messages to different software threads in DriveOS, according to end-to-end timing guarantees[1], and (2) a port of a HVAC controller for an electric vehicle being developed with Drako Motors. We demonstrate the HVAC model's functional and timing correctness with a model-in-the-loop (MIL) and hardware-in-the-loop (HIL) execution equivalence against real-world data traces.

The overall contributions of ModelMap are three-fold: (1) we introduce the first model-based multi-domain application development and deployment framework for a VMS; (2) we demonstrate that Simulink models running on a multicore x86 machine in a centralized VMS have predictable end-to-end delays; (3) we illustrate a Simulink model's functional and timing correctness with MIL and HIL equivalence.

In the next section, we describe the ModelMap tools for DriveOS vehicle management and code generation. Section 4.2 explains the nested binary concept and its runtime in DriveOS. Section 4.3 demonstrates the usage of the model-based design tools to implement ECU functionalities in DriveOS, along with an electric vehicle HVAC case study. Then, we present the evaluation results with simulated and real-world datasets.

4.1 Design Tools

Figure 4.1 shows a high-level overview of the ModelMap code generation steps for a DriveOS multi-domain application. A model is first designed with ModelMap and other Simulink blocks. Then, ModelMap block-level and DriveOS system Target Language Compiler (TLC) [Sim22] files are utilized by the Simulink Embedded Coder, to gener-

[1]This gateway is more generic than the one presented in Section 3.4 which works with only two applications.

ate the domain OS-specific C source code. An OS-specific version of `gcc` then cross-compiles C source into Quest and Linux ELF binaries. Finally, a nested binary compiler (see Section 4.2) is used to create a multi-domain binary executable.

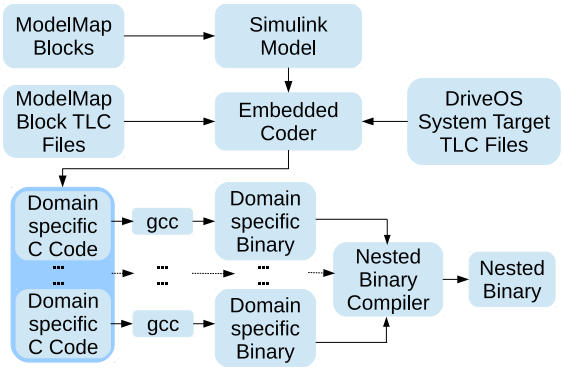


Figure 4.1: The ModelMap Workflow

A modified Embedded Real-Time (ERT) system TLC file [Sim22] specifies the C code generation from Simulink model blocks. The main function C code generation for DriveOS domain-specific OSs is explained in Section 4.1.1.4.

4.1.1 Thread Setup Blocks

A `threadSetup` Simulink block is used to create periodic threads for either Quest or Linux, and aperiodic threads restricted to Linux. The block details are summarized below:

- **Block Type:** C MEX S-function [Mat22c].

- **Block Parameters:** A Simulink block mask [Mat22d] identifies thread-specific parameters. These include the *Thread Name* and *Domain* OS (Quest or Linux). A Quest periodic thread is further parameterized with a *Runtime* and *Period*. A Linux domain periodic thread has a *Runtime*, *Period* and *Deadline*, while a Linux-only aperiodic thread has no further parameters.

- **Block Output:** The output of this block is a function-call trigger that connects to a

function trigger port of a *function-call subsystem* [Mat22h]. This function-call subsystem is executed as a threaded task configured using the above parameters.

4.1.1.1 Quest Periodic Threads

Quest RTOS [DLW11, Wes22] is the high criticality DriveOS domain [Ves07], providing support for periodic threads that perform real-time I/O and secure CAN bus operations. Application threads in this domain leverage a port of the newlib C library API [New22], as well as interfaces specific to Quest.

High criticality control tasks, which require CAN bus access, run as periodic threads in Quest. Example tasks include HVAC and powertrain control, which are designed as Simulink function-call subsystems. The function trigger ports of these subsystems are connected to the output port of a corresponding `threadSetup` block. `threadSetup` is configured with a *Thread Name* parameter, and the *Domain* is set to Quest. A model developer provides the *Runtime* and *Period* parameters that are respectively set as the *budget* (C) and *period* (T) of a Quest periodic task, τ. τ is implemented following a Liu-Layland task model and scheduled using the RMS algorithm [LL73]. This guarantees τ receives its budget, C, every period, T, when runnable.

4.1.1.2 Linux Periodic Threads

Linux is the lower criticality GPOS domain in DriveOS. Although not a hard real-time OS, it provides sufficiently predictable timing guarantees for `SCHED_DEADLINE` [LLFC11] tasks using the `PREEMPT_RT` patch [RMF19,SW21]. Linux provides complementary support for Quest with its libraries, device drivers and services that would take many years of development to implement in a new RTOS. A developer provides the *Runtime*, *Period* and *Deadline* `threadSetup` block parameters that are passed on to the `SCHED_DEADLINE`

policy via the Linux `sched_setattr` system call.

4.1.1.3 Linux Aperiodic Threads

A centralized VMS also runs non-critical operations such as logging, storage and over-the-air updates. `threadSetup` supports these tasks as Linux `pthread`s. No additional `threadSetup` block parameters are needed.

4.1.1.4 Code Generation

The `threadSetup` S-function block's properties are described in a C MEX (`threadSetup.c`) file for simulation, and in a TLC file (`threadSetup.tlc`) for code generation. As the `threadSetup` block is designed for model deployment in a DriveOS system, `threadSetup.c` only saves the block parameters (*Thread Name*, *Domain Name*, *Runtime*, and so forth) for code generation, without any simulation. In the `threadSetup.tlc` file, three key steps are followed to generate its corresponding C code:

1) The `BlockInstanceSetup` TLC function of a S-function block is executed at the very start of code generation [Mat22a]. ModelMap uses this function to retrieve all the block parameters, such as *Thread Name* and *Domain*, from the simulation environment.

2) The `Start` TLC function (Code Block 4.1) includes any initializing code in the final C source code [Mat22a]. ModelMap uses this function to assign the previously retrieved block parameters (*Runtime*, *Period* and, if applicable, *Deadline*) to a DriveOS domain-specific C structure (`linux_` or `Quest_sched_param_t`).

ModelMap uses an array of `*_sched_param_t` structures, called `all_thrd_parms`, to save the parameters of multiple threads in the same domain, each time a `Start` function of a `threadSetup` block is called. The DriveOS system TLC file declares `all_thrd_parms`.

```
%% Simulink TLC Code starts with a %; TLC comments start with a %%
%% Any other code goes to an initializing block of the final C code
%if targetosVal == 0 %% SmaRTOS domain
  s_params->C = %<smartosbudget>; s_params->T = %<smartosperiod>;
%elseif linuxschedpolicyVal == 1 %% Linux domain SCHED_DEADLINE
  s_params->is_sched_deadline = 1; s_params->C = %<linuxruntime>;
  ...
%endif
s_params->threadfuncname = %<threadName>_func;
```

Code Block 4.1: A snippet of the `Start` TLC function

3) Finally, the `Output` TLC function of a block [Mat22a] is used to generate the block's corresponding C code. ModelMap uses `Output` to add a new function in the model's C source file, named `<threadNameVal>_func`. The same function is also embedded in the previous `Start` function as the `pthread` function name. The domain-specific `*_sched_param_t` C structure is passed as an argument to the `pthread` function, to set the corresponding thread scheduling parameters. In the end, the function-call subsystem's corresponding C call is retrieved via a `%<LibBlockExecuteFcnCall>()` TLC function, and embedded in an infinite `while` loop.

The DriveOS system TLC spawns the `pthreads` from the `main` function of the generated C source code. A Simulink custom file processing template [Mat22e] is used to generate the `main` function. A new `pthread` is created for every element in `all_thrd_parms`. Part of the `main` function is shown in Code Block 4.2.

```
for(i = 0; i < num_threads; i++) {
#ifdef SMARTOS
        pthread_t* new_thread = (pthread_t *) malloc(sizeof(pthread_t));
        pthread_create(new_thread, NULL, all_thrd_parms[i].threadfuncname,
                all_thrd_parms[i]);
#else
        // Linux domain
        ...
#endif
}
```

Code Block 4.2: A snippet of the ModelMap-generated `main` function in C

4.1.2 Inter-task Communication Blocks

ModelMap provides a set of blocks for intra- and inter-domain task communications. These blocks are set up as *shared memory communication channels* via the hypervisor layer of DriveOS [SW21,LXRD19]. DriveOS uses Intel VT-x extended page tables (EPTs)

to securely map host physical memory regions between communicating threads, irrespective of their domain. Both synchronous and asynchronous communications are supported across channels identified with a unique `channel_key`.

DriveOS has a set of C API functions in Linux and Quest to set up the communication channels. ModelMap implements MATLAB interfaces for these C functions [Mat22b], which are described further in the next two subsections.

A ModelMap `createChannel` Simulink block takes an integer input as the `channel_key`. It has block mask parameters to set the type and specification of the channel. For a synchronous channel, the buffer length and the size of each element must be specified. For an asynchronous channel, only the element size is needed.

4.1.2.1 Synchronous Communication

A synchronous channel is implemented as a ring buffer in DriveOS. This is useful for control data that must be communicated without loss. Channel data structures are created by OS-specific userspace libraries in both Quest and Linux domains. `syncRead` and `syncWrite` are busy-waiting calls that read and write, respectively, a message in the buffer. Busy-waiting is used for synchronous communication in Quest-V [WLMD16] and DriveOS [SW21]. This contrasts with ACRN's blocking approach [LXRD19].

Blocking or busy-waiting is problematic when reading or writing to different channels. As will be seen later in Figure 4.9, a `syncRead` on one channel may delay the execution of `syncRead` calls on other channels. To mitigate this issue, ModelMap implements `syncNWRead` and `syncNWWrite`, which are non-waiting (NW) function blocks. Calling `syncNWRead` (or `syncNWWrite`) when a channel buffer is empty (or full) immediately returns -1, otherwise it returns the size of the read (or written) message.

4.1.2.2 Asynchronous Communication

An asynchronous channel in DriveOS is implemented using Simpson's four-slot buffers [Sim90]. This is useful when the most recent (i.e., freshest) data must be communicated, while stale data is discarded. Sensor readings fall into this category of communication. `asyncRead` and `asyncWrite` Simulink blocks read and write asynchronous messages, respectively.

4.1.3 CAN I/O Blocks

CAN bus communication between sensors and actuators is commonly used in the automotive domain. DriveOS implements real-time USB-CAN I/O in its Quest domain [GCW18]. The CAN I/O API of Quest is accessed via MATLAB's C-interface function blocks, including `canChannelSetup`, `canRead`, and `canWrite`. `canChannelSetup` has a block parameter that sets the CAN bus baud rate with an option from 10 kbit/s to 1Mbit/s.

4.1.4 Timing Blocks

ModelMap implements several Simulink timing blocks. MATLAB function block `time_from_start` outputs the time in μs since the model starts running. Similarly, `time_since_last_called` outputs the time in μs since the last time this block was called. These timing functions use the x86 `RDTSC` instruction to measure processor clock cycles, divided by the base clock frequency, which yields accurate time in μs. These blocks are C-function interfaces in MATLAB [Mat22b].

4.1.5 Domain-specific C Code Generation

Domain-specific components of a model are designed as atomic subsystems in Simulink. ModelMap generates domain-specific C code for a selected subsystem target in Simulink with the MATLAB command `slbuild(<subsystem name>)`.

4.2 Nested Binaries

ModelMap produces an ELF binary [Lu95] format as executables for DriveOS, called a *nested binary*. Similar to a fat binary [DFS98], a nested binary contains multiple binary executables. These individual binary executables may have Application Binary Interfaces (ABIs) for different OSs. For example, DriveOS nested binary executables have binaries for both Quest and Yocto Linux.

ModelMap includes a nested binary compiler. A corresponding nested binary loader performs runtime parsing of individual binary executables within a nested binary. It then spawns a new process for every binary into a corresponding DriveOS sandboxed domain.

4.2.1 Nested Binary Format

The nested binary acts as a container for the individual raw ELF binaries. It stores the raw binary bytes in its separate ELF data sections, named `binexec` sections. There is a `binexec` section for each individual domain-specific binary. The location offsets to these data sections are stored in the ELF section header table. A nested binary also has a `metadata` ELF section to store the mapping between an individual ELF binary, saved in a `binexec` section, and a runtime domain ID for the binary. Figure 4.2 shows the organization of the nested binary sections in the ELF format.

4.2.1.1 ELF Header

The ELF header defines the target OS ABI, bitness, and other details in an ELF binary. The following fields are modified: (1) **`e_ident[EI_OSABI]:`** The target OS ABI is set to a custom value of `0x15`; (2) **`e_machine:`** The target ISA is set to `EM_386` (`0x03`) for an x86 target; (3) **`e_type:`** The object file type is set to `0x02` for an executable file; (4) **`e_ident[EI_DATA]:`** The endianness is set to little-endian.

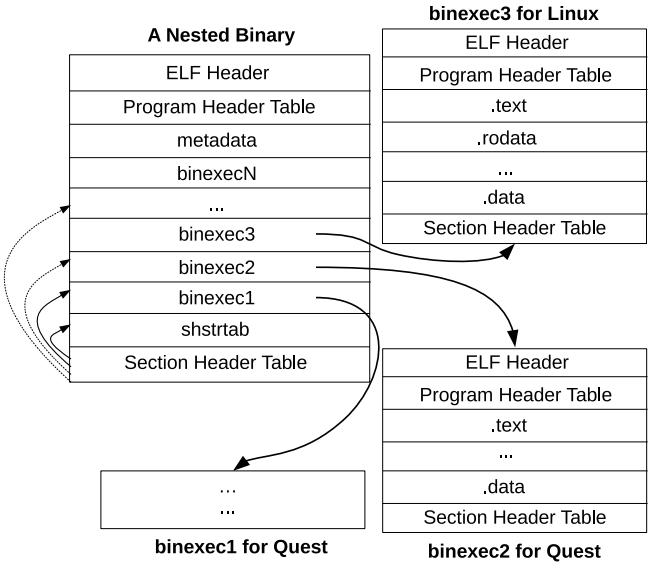


Figure 4.2: Nested Binary Sections

4.2.1.2 Program Header Table

This section has one entry to satisfy the ELF format requirement. The entry is the Program Header Table (`PT_PHDR`) itself. As individual executables in a nested binary have their respective program header table for runtime process image information, this section is not needed.

4.2.1.3 Section Header Table

This section lists all the data sections in a nested binary:

• **`binexec`:** Every nested binary has one or more `binexec` sections. These sections contain the individual binary executables in different ABIs as raw binary bytes.

The name of every `binexec` section is appended at the end with an integer numeric ID, starting from 1. This ID specifies the order in which the binaries will be spawned at runtime, where lower ID means earlier execution. Figure 4.2 shows N number of `binexec` sections: `binexec1, binexec2, ..., binexecN`, where `binexec1` will be spawned before `binexec2` and so on.

• **metadata:** The `metadata` section maps an individual binary in a `binexec` section to a domain in DriveOS. The nested binary loader uses the metadata section to spawn a new process from an individual binary in the corresponding domain. The section contains an array of `C structs` which holds a tuple of the `binexec` section name and the corresponding integer domain ID. Currently, DriveOS assigns domain ID 1 to Quest and 2 to Linux.

• **shstrtab:** This is the string table section that contains the section names, like other ELF binaries.

4.2.2 Nested Binary Compiler

ModelMap's nested binary compiler (`nested_bin_cc`) creates a nested ELF binary from multiple individual binary executable files. The compiler utilizes the `libelf` library [Kos10] to create, enumerate and organize different ELF binary sections according to the above format. The following command is used to create a nested binary:

```
nested_bin_cc <Binary File1> <Domain ID1> ...
        <Binary FileN> <Domain IDN> <Name of Nested Binary>
```

The above command combines N ELF binary executable files and saves the mapping between a binary and its runtime domain in the `metadata` section. For example, `Binary File1` is mapped to `Domain ID1`. A new nested binary is created as per the last argument. The sections of a nested binary can be inspected with `binutils` tools such as `readelf`.

4.2.3 Nested Binary Loader

The nested binary loader is also implemented with the `libelf` APIs [Kos10]. The loader runs in a DriveOS Linux domain and takes a nested binary as the first argument. It also

takes a number of command-line arguments for every individual ELF binary. The following command is used to execute a nested binary:

```
nested_loader <nested ELF binary>
        <argc1> <argv11> ... <argc2> <argv21> ...
```

Here, `argc1` is the number of command-line arguments for the `binexec1`, starting with `argv11`. Similarly, `argc2` is the argument count for `binexec2`, starting with `argv21`, and so on.

The loader parses the `metadata` section in a nested binary to read the mapping between a `binexec` section and its runtime domain. It spawns a new process with the raw bytes of a binary embedded in a `binexec` section to its respective domain. Figure 4.3 summarizes the steps to execute a binary in Linux and Quest.

4.2.3.1 Executing a binary in a Linux Domain

The nested binary loader first parses the metadata and then iterates over the sections in a nested binary to find the corresponding `binexec` sections. It checks whether the domain ID in an entry of the `metadata` section is 2. If it is, the loader employs the fork-and-exec approach in Linux to spawn the raw binary as a new process. However, as the executable is not in a file and directly available in memory, the `execve`-class of `C` functions cannot be used. Instead, the loader creates a new file descriptor for the memory location of the Linux binary with the `memfd_create` function [Lin22b]. Then, it `fork`s a new child process and calls `fexecve` [Lin22a] to execute the binary.

4.2.3.2 Executing a binary in a Quest Domain

Executing a program in a remote Quest domain requires more effort than starting an executable within Linux. The nested binary loader running in Linux first identifies the domain ID=1 for a Quest binary object. It then sends the raw binary bytes to Quest via shared

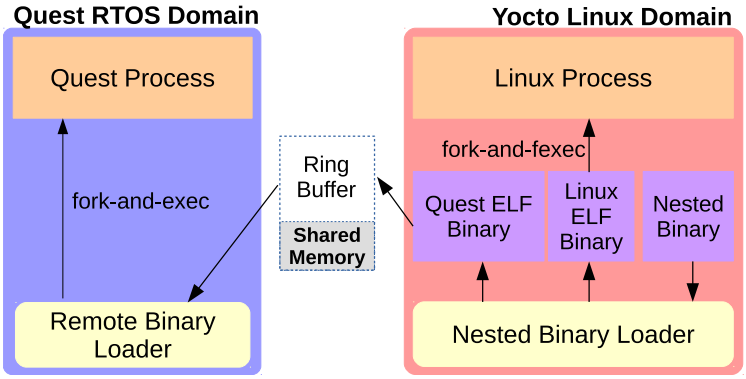


Figure 4.3: Nested Binary Loader

memory as shown in Figure 4.3. A specific shared memory region of 800KB is mapped between the Quest and Linux domains at system boot time for remote binary execution. The region is appropriately sized to accommodate Quest static binary sizes. This shared memory region works as a synchronous ring buffer channel with a single buffer slot. A remote binary loader process in Quest polls the shared region for any new binary execution request. Once the nested binary loader in Linux indicates that it has written a new program to the shared region, the remote binary loader in Quest starts reading the program and its arguments. Then, the remote Quest loader spawns a new process with a fork-and-exec mechanism.

4.3 Evaluation

ModelMap is tested with custom and real-world Simulink models. The goal is to show that the worst-case end-to-end (E2E) delays, or the maximum reaction times [DZDN+07], of the models are within their expected upper bounds after deployment in DriveOS.

ModelMap Simulink blocks are applied to the following three models: (1) a multi-domain *synthetic benchmark* with different types of inter-task communication, (2) a DriveOS *CAN Gateway* [SB07] to filter and forward CAN messages to different Quest and Yocto Linux applications, and (3) a port of an *automotive HVAC control* Simulink model

for a MotoHawk ECU to DriveOS. The functional and timing correctness of the ported HVAC model is demonstrated by the output equivalence in both MIL and HIL executions.

The three models are tested with DriveOS running on a Cincoze DX1100 industrial PC [Cin22], as described in the previous chapter. The HIL simulation setup of DriveOS in Figure 3.4 is used in these experiments as well. As it is described in Section 3.5.1, DriveOS uses two USB xHCI bottom-half handler threads in Quest, each with budget=0.1ms and period=1ms, referred to as `USBBH_rx` and `USBBH_tx`. Two USB-CAN kernel driver threads, each with budget=0.2ms and period=1ms, send (`CAN_tx`) and receive (`CAN_rx`) CAN messages.

4.3.1 Synthetic Benchmarks

Figure 4.4 shows our multi-domain Simulink benchmark model, designed using ModelMap blocks. The inter-task and CAN channel setup blocks are omitted. The model reads a CAN message in the `canReader` atomic subsystem of the Quest domain (Domain 1) from CAN channel 0 (CAN0) via `canRead`. Then, `syncWrite` forwards the data to the `procThread` atomic subsystem for processing in the Linux domain (Domain 2). This setup allows `procThread` to apply any control logic to the received message using additional Simulink blocks. For our experiments, `procThread` forwards the message to a `canWriter` atomic subsystem in Domain 1 using `syncWrite`. The `canWriter` then outputs a message on CAN channel 1 (CAN1). This model is representative of the canonical communication path between two CAN bus interfaces and separate OS domains in DriveOS.

The `canReader`, `canWriter` and `procThread` blocks in Figure 4.4 are function-call atomic subsystems, configured as periodic threads. Their function trigger ports are connected to the output ports of the `threadSetup` blocks. The `threadSetup` blocks are assigned to the Quest domain for the `canReader` and `canWriter` subsystems, and

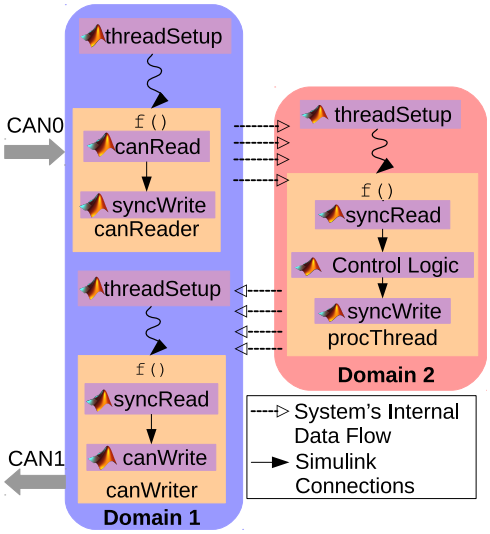


Figure 4.4: A Multi-Domain Simulink Model for Benchmark

to the Linux domain for the `procThread` subsystem. Their budgets and periods, given in Table 4.1, are derived empirically by profiling [WEE+08], and assigned in the corresponding `threadSetup` blocks. `canReader` and `canWriter` subsystems rely on Quest real-time capabilities. `procThread` is representative of a lower criticality control task that is scheduled in the Linux domain using the `SCHED_DEADLINE` policy.

Table 4.1: Budgets and Periods for the Synthetic Benchmark

Subsystem/Thread	Budget (μs)	Period (μs)	Util. (%)	# of Threads
Domain 1				
canReader	100	2000	5%	1
canWriter	100	2000	5%	1
Domain 2				
procThread	100–500	1000	10–50%	1

The Simulink model's corresponding C code and subsequent nested binaries are automatically generated. When this model is launched by ModelMap's nested binary loader, `canReader` and `canWriter` threads are spawned in Domain 1, and `procThread` is spawned in Domain 2 at runtime. This model is a classic example of a sensing-processing-actuation task pipeline [DZDN+07, BDM+17].

4.3.1.1 End-to-end Delay Performance

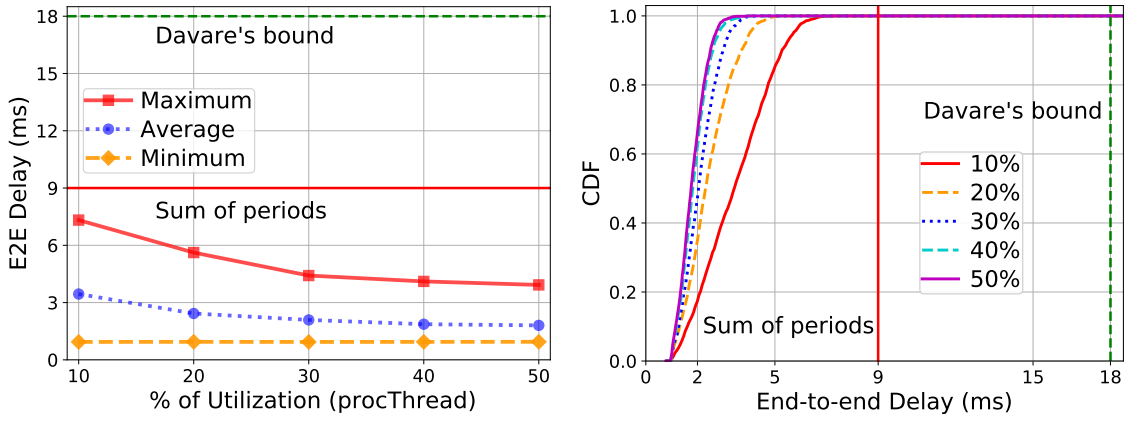
We measure the end-to-end (E2E) delay (also known as the maximum reaction

time) [DZDN+07, DBCC19] of a CAN message traversing through `canReader` →`procThread` → `canWriter` threads. The E2E delay upper bound for a pipeline of periodic real-time tasks has been theoretically analyzed before [DZDN+07, BDM+17, DBCC19, CWE18], but only for a domain-specific scheduling algorithm. However, it is important to measure the E2E delay of multi-domain applications in a centralized VMS where the time-critical software components expand beyond a single domain [SW21, BSC+21]. We perform an experimental evaluation of such multi-domain applications in this paper and use the sum of the task periods [GSW20] (*optimistic*) and Davare's upper bound [DZDN+07] (*conservative*, twice the sum of periods assuming response-time of a task $\leq$ its period) as the two target E2E delay bounds. As stated earlier, the `USBBH_rx`, `USBBH_tx`, `CAN_rx` and `CAN_tx` threads are also considered in a task chain for CAN I/O, as a CAN message has to pass through these I/O threads as well. For example, the aggregate period delay bound for Figure 4.4 will be $\big($ (1 + 1 + 1 + 1) [for the I/O system threads] + (2 + 2 + 1) [from Table 4.1] $\big)$ = 9ms.

4.3.1.2 Result Analysis

A stream of messages is sent from an Ubuntu 18.04 Linux machine to DriveOS via CAN0 on the DX1100. A corresponding message is received via CAN1 on the same Ubuntu machine. The sent and received CAN message timestamps are logged with `candump` in Ubuntu, to calculate the E2E delay. In the first set of experiments, we vary the utilization (ratio of *Runtime* and *Period*) of the `procThread` subsystem from 10 to 50% by increasing the *Runtime* parameter in the associated `threadSetup` block. Figure 4.5a shows the minimum, average and maximum E2E delays with increasing `procThread` utilization in Linux. All the E2E delays are within the target upper bounds.

Figure 4.5a shows that the maximum E2E delay is improved by 46%, as `procThread`'s utilization is increased from 10% to 50% in Linux. This coincides with



(a) Average, Minimum, Maximum

(b) Cumulative Distribution Function

Figure 4.5: Benchmark E2E Delay vs Domain 2 Task Utilization

the increased fraction of all E2E delays within the x-axis bound in Figure 4.5b. The median latency every 10 frames in Figure 4.6 is more variable for the 10% case than others. Allocating more utilization to a Linux domain subsystem not only improves the maximum E2E delay but also reduces jitter. However, CPU utilization is often limited in resource-constrained automotive systems. ModelMap's maximum E2E guarantee is crucial for time-critical control software modeling.

`procThread` in RTOS vs. Linux The next experiment compares the previous model to one where the `procThread` subsystem in Figure 4.4 is moved to Domain 1, leaving Domain 2 idle. A special BG scheduling mode in Quest is also tested. This mode gives additional CPU time to a task beyond its model-specified CPU time via *background scheduling*, if other tasks do not need anymore CPU.

Figure 4.8a shows the E2E delays when increasing the `procThread` utilization up to 30% [2], keeping its period fixed at 1ms. All E2E delays are under the target upper bounds. The maximum E2E delays for the Quest BG mode stay almost the same, as `procThread` leverages the additional CPU time. Without BG mode in Quest, the E2E delays are still well under the target bounds, but the maximum ones are worse than Linux for higher

[2] $>$30% is not possible due to the rate-monotonic scheduling bound with other tasks.

utilization. As `procThread`'s period is fixed, its priority remains same in Quest, even with higher utilization. Therefore, the maximum E2E delay does not decrease as much as it does while running in Linux, where only `procThread` is executed.

In another experiment, `procThread`'s period is increased from 1ms to 8ms, keeping its utilization fixed at 10%. The results in Figure 4.8b show that the E2E delays are increased with greater `procThread` period. If a CAN message is not handled in the same job (i.e., task instance) that it is received, it might wait for potentially more than a task's period to be transferred. Therefore, E2E delays increase with higher `procThread` periods. However, the maximum E2E delays are within the target upper bounds, except for the 8ms period in Linux where it violates the aggregate period bound. As the `procThread`

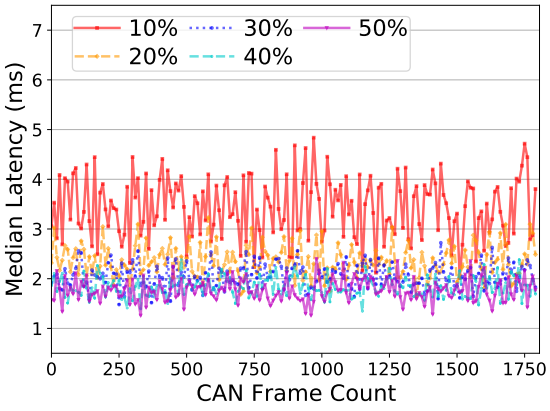


Figure 4.6: Synthetic Benchmark

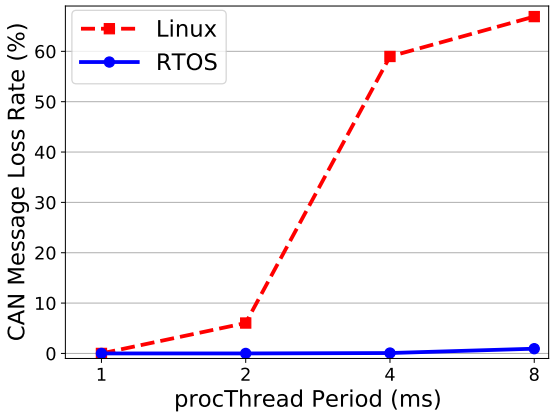
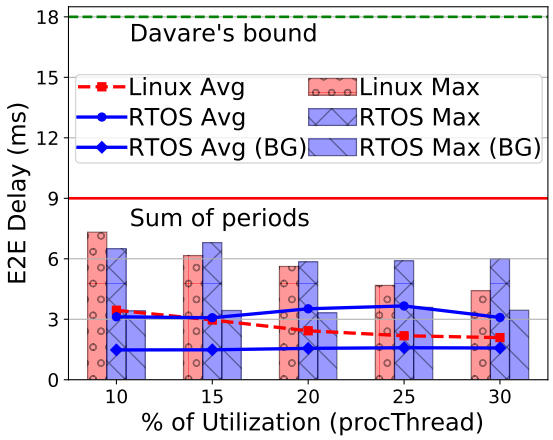
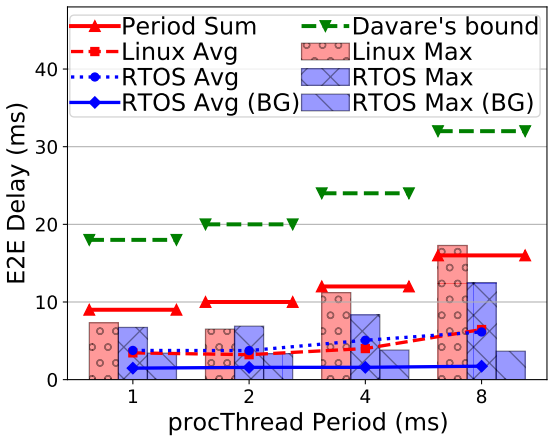


Figure 4.7: Loss Rate in Async Channel



(a) Increasing Util. (Fixed Period)



(b) Increasing Period (Fixed Util.)

Figure 4.8: Running `procThread` in Linux (Domain2) vs. Quest RTOS (Domain1)

period in the 8ms case is significantly more (4x) than the periods of `canReader` and `canWriter` threads (2ms), buffering delays increase the maximum E2E delay beyond the sum of periods. Nevertheless, all the delays are well under Davare's bound.

Asynchronous Communication Block The next experiment replaces all `syncRead` (and `syncWrite`) blocks with `asyncRead` (and `asyncWrite`) blocks in the model of Figure 4.4. In asynchronous communication, if a receiver task has a greater period than a sender task, then a message is potentially overwritten by the sender, before it is observed by the reader. The number of lost messages is important in asynchronous communications. Figure 4.7 shows the loss-rate (ratio of number of lost messages and total messages) against increasing `procThread` period, while it is run in Linux and Quest.

A stream of 1275 CAN messages are sent at 5ms intervals from the Ubuntu machine. In Figure 4.7, as long as receiving `procThread`'s period in Linux is less than the sending `canReader`'s period of 2ms, there is no data loss. The loss-rate increases with greater periods from 2ms. As `procThread`'s period goes greater than or equal to `canReader`'s period, `procThread` starts missing CAN messages. In the Quest-only model, no loss is observed until 8ms period, because the source message rate (1/5ms=200Hz) is greater than the rate of all the tasks, and they are all running with the same RMS scheduling policy. However, when `procThread` runs in Linux it is scheduled earliest-deadline first according to the `SCHED_DEADLINE` policy. As Quest tasks are scheduled in RMS order there is a potential priority mismatch, hilighting the importance of correctly setting task periods for multi-domain task models.

4.3.2 Case Study 1: CAN Gateway

A centralized VMS needs a CAN Gateway to access the various CAN buses and distribute messages to host tasks spanning different domains. DriveOS's CAN Gateway is shown as

a ModelMap Simulink model in Figure 4.9.

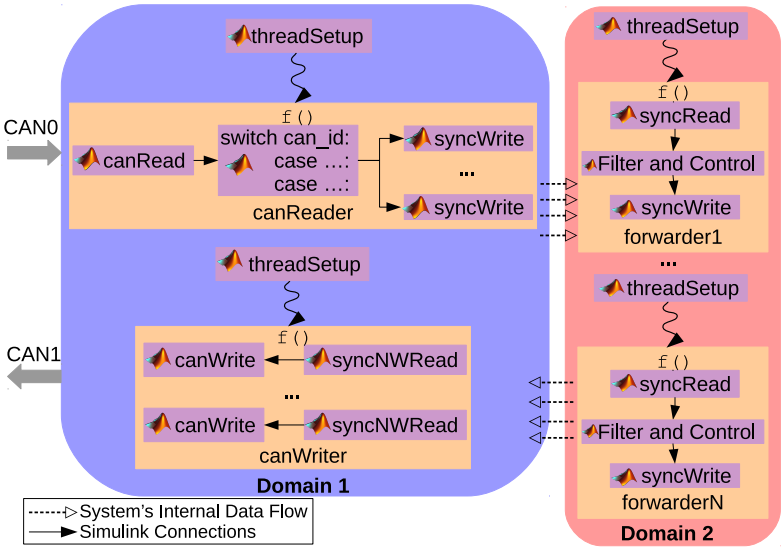


Figure 4.9: Model of a CAN Gateway

As before, CAN messages are read via CAN0 in the Quest domain (Domain 1) and forwarded to the Linux domain (Domain 2) by a `canReader` subsystem. A `canWriter` subsystem receives CAN messages from Linux to be sent out via CAN1. Unlike the previous benchmark model, there are multiple subsystems in Linux to process different categories of CAN messages based on their CAN IDs. Figure 4.9 shows N Linux subsystems (`forwarder`$\{1...N\}$) where $N = \{1, 2, 4, 8\}$ in our experiments. Each `forwarder` Linux subsystem is connected to two inter-task synchronous channels: one is to receive CAN messages from `canReader`, another is to send CAN messages to `canWriter`. If a different Linux application wants to receive (or send) a message of any particular CAN ID, it has to request it from the specific Linux `forwarder` subsystem of the CAN Gateway. For example, Instrument Cluster and In-vehicle Infotainment applications in DriveOS request CAN message transfers via specific Linux `forwarder` subsystems. For the E2E delay overhead of the CAN Gateway, `forwarder`$\{1...N\}$ pass through the CAN messages from their incoming inter-task channel (from `canReader`) to the outgoing channel (to `canWriter`).

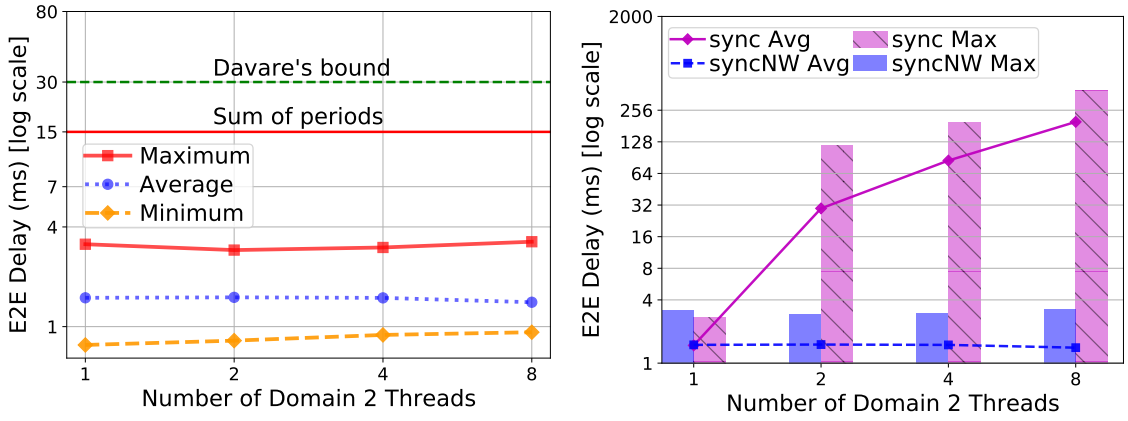
The `syncNWRead` block in `canWriter` is used to read from inter-task channels, without busy-waiting when a buffer is empty. Experiments show that `syncNWRead` significantly improves the E2E delay. `syncNWWrite` blocks are not used in `canReader`, as the Linux subsystems keep the inter-domain communication buffer free by reading out messages at a suitable rate.

Table 4.2: Budgets and Periods for CAN Gateway Case Study

Subsystem	Budget (μs)	Period (μs)	Util. (%)	# of Threads
Domain 1				
canReader	200	2000	10%	1
canWriter	300	1000	30%	1
Domain 2				
forwarder{1–8}	6400–800	8000 (fixed)	80%–10%	1-8

4.3.2.1 Result Analysis

Table 4.2 shows the budgets and periods of all the CAN Gateway tasks. The E2E delays are plotted in Figure 4.10 against increasing numbers of Linux `forwarder` threads, keeping their total utilization at 80%. For example, if two `forwarder` threads are executed, then each of them has 40% utilization. The E2E delays in Figure 4.10a exhibit low jitter and remain under the target bounds in all cases. This shows that ModelMap's CAN Gateway is able to handle multiple Linux threads in a DriveOS VMS system.



(a) Increasing # of Domain2 Threads

(b) Polling vs Non-polling Read

Figure 4.10: CAN Gateway Case Study

In other experiments, `syncNWReads` are replaced with `syncRead` blocks in the

`canWriter` subsystem. Figure 4.10b demonstrates that the polling `syncRead` block increases the E2E delay sharply as the number of threads increase. `syncNWRead` blocks are important for a scalable CAN Gateway as the busy-waiting times on synchronous channels are prohibitively large with more threads.

4.3.3 Case Study 2: Automotive HVAC Control

In this study, our electric car's existing HVAC controller, running on a MotoHawk ECU, is ported to DriveOS using ModelMap Simulink blocks. The HVAC Simulink function-call subsystem is connected to a `threadSetup` block. `threadSetup` configures the HVAC subsystem to run in the Quest domain with 0.5ms budget and 5ms period. The HVAC subsystem communicates with the Linux domain to save settings in persistent storage for when the vehicle is restarted. Cross-domain communication is omitted in this study, as it has been covered by prior experiments. Instead, the functional and timing correctness of the HVAC control is investigated.

The HVAC control receives input signals via 18 CAN IDs and sends the output signals via 7 additional IDs. CAN I/O is via two inter-task communication channels with the CAN Gateway mentioned above. The HVAC control avoids waiting on any CAN IDs by using `syncNWWrite` and `syncNWRead` blocks for message transfers. The CAN Gateway `canReader` and `canWriter` threads are set to 0.2ms budget and 4ms period.

The HIL outputs of the HVAC model after its deployment in DriveOS are compared with the MIL outputs in Simulink, using CAN data traces from our electric car. Every CAN input message is tagged with a unique and monotonically increasing integer Tag ID, which is passed through to the HVAC control's output CAN messages. The HIL and MIL signal values in the HVAC control are checked to see that they match for all Tag IDs. Due to space limits, the HIL and MIL Driver Temperature signal outputs are shown over 30 seconds in Figure 4.11. For Tag IDs 756 and 762, `driv_temp` is respectively 1 and 2 in

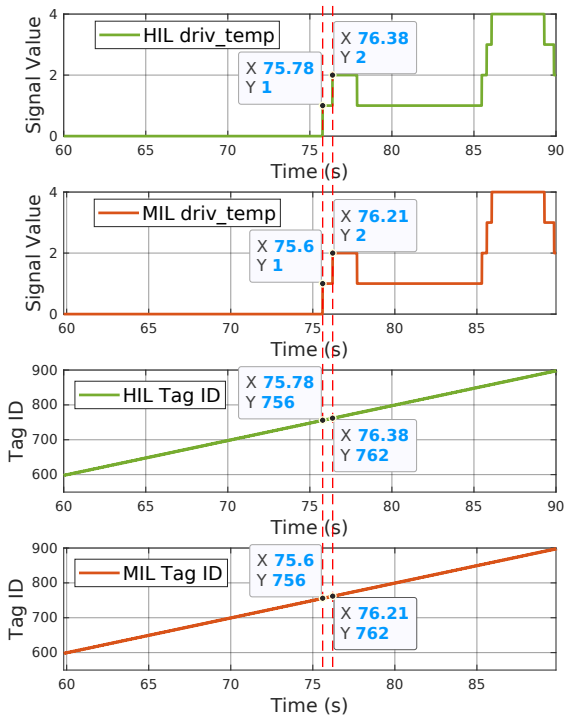


Figure 4.11: HIL and MIL: Driver Temperature Signal with Tag IDs

both MIL and HIL simulations. This is observed for all the Tag IDs and signals. The time difference on the x-axis is the DriveOS system and HVAC control overhead and contributes to the signal reaction time. Reaction times for all the signals in the HVAC control are in the similar range of 160–180 ms, which is deemed acceptable for our vehicle.

4.3.4 System Overheads

The ModelMap framework and nested binary overheads are measured with a series of microbenchmarks. All measurements are taken 20 times, and an average is presented. The x86 `RDTSC` instruction is used for timing measurements, having an overhead of 0.04μs or around 96 clock cycles, which is subtracted from all delays.

Table 4.3 presents the overheads of the ModelMap Simulink blocks in DriveOS. The `threadSetup` block takes more time in Quest than it takes in Linux, because Quest has to create a sporadic server abstraction for RMS [SSL89, DLW11]. The creation and connection to an inter-task communication channel make expensive `VMExit` [RKLM17,

Table 4.3: System Overheads for the DriveOS Simulink Blocks

	Time (μs)	
Simulink Block	**Linux**	**Quest**
threadSetup	33	190
channelCreate{Sender,Receiver}	5722	5486
channelConnect	5699	5448
(a)syncRead/Write	0.01-0.03	0.01-0.03
canChannelSetup	-	18490
canRead/Write	-	1

SW21] operations to the underlying hypervisor and take more time than reading/writing to the memory-mapped channels. CAN channel setup takes 18ms to configure the transfer rate of the USB-CAN interface. These blocks are only applied in an initial setup phase without significant runtime costs.

Nested Binary Measurements A nested binary's size is the sum of all individual binaries and 14-bytes of metadata per binary. Table 4.4 presents the overheads of executing a nested binary. Quest only supports static binaries for fast and predictable runtime, so they are typically larger than Linux dynamically-linked binaries.

Table 4.4: Nested Binary Overheads

Operation	**Time (ms)**
Extracting an individual binary from a nested binary	0.16
Forking a process via memory in Linux	0.11
Sending ˜300KB binary from Linux to Quest	7.13
Receiving ˜300KB binary in Quest and forking it	184

Chapter 5

End-to-end Scheduling of Real-time Task Pipelines in Multiprocessors

Real-time embedded and cyber-physical systems are amassed with examples of task pipelines where a series of tasks are connected by data-buffers. In the automotive domain, a sensory input is passed on to a pipeline of processing and control tasks that activate an actuation output. Such time-critical systems benefit from a real-time task pipeline model. As multiprocessor and multicore machines are being increasingly used in embedded applications, scheduling real-time task pipelines on multiprocessors needs critical investigation.

Task pipelines, or task chains, or cause-effect chains have received increased attention in recent research work [GCU+21, KBS20, CKK20, DBCC19, KBS18, SE16], partly because of their active usages in well-known software packages like ROS [CBLB19, TFG+20]. Although real-time task pipelines have long been studied [FR97, LA09b, LA09a], the application of constraints on a pipeline has received little attention [DZDN+07]. Constraints on a pipeline of periodic tasks ensure that the end-to-end properties of a pipeline are guaranteed. However, finding schedulable task runtimes and periods to satisfy the end-to-end constraints is an NP-hard problem [DZDN+07]. The traditional solvers are not convenient to be used because of their slow runtimes. They are also unsuitable in runtime scheduling, where tasks and pipeline may dynamically appear in a real-time system.

This paper presents a heuristic constraint solver algorithm for real-time task pipelines, CoPi [SW22], to derive the runtime budgets and periods of individual pipelined tasks from a list of supplied task budgets. CoPi works with two pipeline constraints: the worst-case end-to-end (E2E) delay and loss-rate. The worst-case E2E delay is the maximum time interval between the first time data appears at the first task of a pipeline, and the first time a corresponding output is produced at the last task of the pipeline. The worst-case E2E loss-rate is the number of input messages to the pipeline that do not have a corresponding output with respect to the number of input messages to a pipeline over the period of its first task. As CoPi treats pipelined tasks as asynchronous and independent tasks, data might be lost between two communicating tasks if a producer overwrites its output before a consumer has read it. The loss-rate captures how many input messages have no effect at the end of the output of a task pipeline. In addition to these constraints, CoPi uses the rate-monotonic scheduling (RMS) algorithm to schedule the tasks and the RMS utilization upper bound as another constraint.

The main idea behind CoPi is to get rid of the unnecessary delay and message losses in a pipeline. In previous work [GSS95, FRNJ08], tasks are released at some offsets, or task precedence relations are created, to mitigate the data dependency between communicating tasks. CoPi finds the suitable task runtimes and periods so that no timing and data dependencies occur at runtime between the communicating tasks.

Figure 5.1 shows a small example where CoPi meets the E2E delay upper bound by

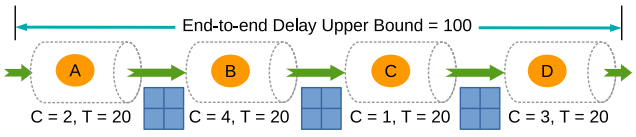


Figure 5.1: CoPi Period Derivation: Above pipeline has an end-to-end delay upper bound constraint of 100 time units. In the simplest case, CoPi divides the delay by (number of tasks + 1) [assuming input is available at arbitrary time. If input is only available at the beginning of A's period, then the upper bound could be tightened to 80 time units]. C and T are respectively the runtime budget and period of a task.

assigning suitable periods to the pipelined tasks. The example also meets the RMS utilization bound requirement. However, a tighter upper bound on the E2E delay could violate the RMS bound, and CoPi needs to find another set of appropriate task runtimes and periods. CoPi tunes the individual task runtimes and periods so that E2E delay and loss-rate are under their upper bounds, while the total utilization does not cross the RMS bound.

As CoPi meets the E2E delay and loss-rate guarantees of a pipeline, the asynchronous tasks are scheduled without any timing or data dependencies between each other. We leverage this feature of CoPi to map the tasks of *multiple* pipelines to a multiprocessor system. Figure 5.2 summarizes this main idea. We use the Worst-fit Decreasing (WFD) heuristic to map tasks to processors and also incorporate runtime task migration and scheduling parameter optimization strategies to admit dynamically appearing pipeline.

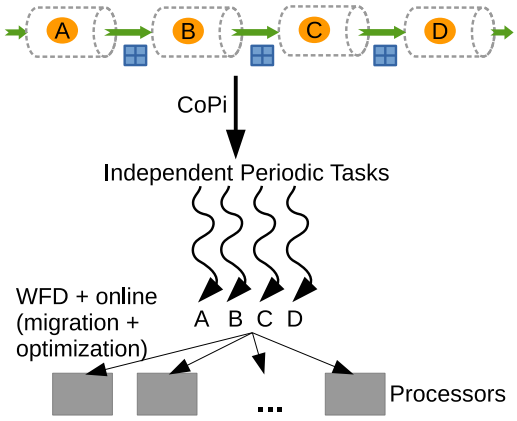


Figure 5.2: CoPi converts a task pipeline to a set of independent and asynchronous tasks. A four-slot asynchronous buffer [Sim90] is used between a pair of communicating tasks. The tasks are then mapped to a multiprocessor system.

As more real-time systems support dynamic environments such as object detection in autonomous driving [HCKK20], supporting pipeline scheduling at runtime is necessary in uniprocessor and multiprocessor architectures. Implementing a complete MINLP solver is difficult and sometimes infeasible in an OS-level scheduler. Therefore, our proposed CoPi heuristic algorithm and its use in multiprocessors are useful for practical implementation

in the future.

This chapter makes the following contributions:

1. We formally present the problem of finding suitable runtime budgets and periods of a pipeline of periodic tasks, under two pipeline constraints (E2E delay and loss-rate), and an utilization bound constraint.

2. We propose and analyze a heuristic constraint solver algorithm, CoPi, to satisfy all the constraints.

3. We demonstrate the usefulness of CoPi by using it in a multiprocessor scheduling algorithm and augment it by runtime task migration and pipeline scheduling optimization strategies to map dynamic pipelines.

4. We evaluate CoPi against open-source Mixed-integer Non-linear Programming (MINLP) solvers such as GEKKO [BHMH18], scipy [sci22] and pyomo [HWW11, BHH+21] with simulated task pipelines. The artifacts are available on `https://github.com/sohamm17/pipe_schedule`. We show that CoPi performs significantly better, an order of magnitude at the highest, in runtime, and comparably in pipeline acceptance ratio (ratio of the number of schedulable pipelines by a solver and the total number of pipelines) for randomly generated pipelines, with respect to other solvers. We have also tested CoPi with tasksets from the WATERS 2015 workshop paper [KZH15] and observe similar performance as noticed in randomly generated pipelines. Moreover, simulation experiments for multiprocessor scheduling demonstrate CoPi's usefulness in maximizing processor utilization and minimizing runtime task migrations. Finally, CoPi's derived task budgets and periods of a pipeline are shown to satisfy the end-to-end latency and loss-rate in DriveOS.

The next section describes the system model. Then, Section 5.2 formally defines the

pipeline constraints and the problem of finding schedulable task periods and budgets of a pipeline. Section 5.3 describes and analyzes CoPi. Section 5.4 explains the multiprocessor scheduling algorithm using CoPi and two pipeline acceptance improvement techniques. An evaluation with simulated task pipelines is presented in Section 5.5.

5.1 System Model

In this section, we define the task, pipeline and scheduling models in the system.

5.1.1 Task Model

A task τ in the system is a two-tuple (C, T) and asynchronous. This means that a task does not wait or block on another task for a resource. Every task has two four-slot asynchronous buffers [**?**] for its input and output and works with the most recent available data. The definitions of C and T are following:

- C: the worst-case runtime budget or capacity of a task to read a message (or *data-unit*) from its input buffer, process the data and write a message (or data-unit) to its output buffer. This is extensible to inputs or outputs of more than a single data-unit in a four-slot buffer.
- T: the period and deadline of an implicit-deadline periodic task. In every new period, τ works on new data, and generates a unique output.

C is the *initial* runtime budget to process a single data-unit by a pipelined task. Later, we show how our heuristic constraint solver algorithm CoPi adjusts the final allocated budget using C to meet the end-to-end constraints.

5.1.2 Pipeline Model

A pipeline P is represented by an ordered set of periodic tasks: $S = \{\tau_1, ..., \tau_N\}$. The cardinality of S is N. $\forall \tau_i, \tau_j \in S$, $i < j$ implies that data flows from τ_i to τ_j. Without the loss of generality, we consider unidirectional pipelines without cycles.

5.1.2.1 Overview of Pipeline Constraints

We explain the pipeline constraints formally in Section 5.2. We provide a high-level overview of them below:

- E: the worst-case end-to-end latency or delay of a pipeline i.e., the maximum time a single message takes from the input to the output of a pipeline. The input appears at any arbitrary time for the first task τ_1 of a pipeline.

- L: the end-to-end loss-rate i.e., the number of input messages that do not have a corresponding output message, for every input message to the pipeline, over the period of its first task. It is expressed as a fraction or a percentage. As the tasks work with asynchronous buffers, a certain message might be overwritten and lost due to more than one consecutive writes by a producer task before a read by a consumer task. L captures how many messages are lost per input message.

5.1.2.2 Communication Model

A task communicates with another task with a message or data unit. In practice, a message is either a sensor input like IMU data, or actuator outputs like steering control, or processed data in between inputs and outputs. In the automotive and factory automation industry, messages are also called labels [KZH15].

The tasks in a pipeline communicate with each other following an implicit communication model [HDK+17]. Therefore, a message is read from a shared input buffer at the beginning of a job of a task, and used throughout the task before writing to a shared output buffer. This ensures that a single and consistent copy of a message is used for a single job invocation of a task.

Simpson's four-slot algorithm [**?**] is used to exchange data between a pair of communicating tasks via a register-based fully asynchronous buffer. In this algorithm, two pairs of slots are maintained separately for a reader and a writer. The writer uses two control bits to indicate which pair and slot are being most recently written. The reader uses another control bit to indicate which pair it is reading. The algorithm shows that only four slots are enough for a reader and a writer to communicate asynchronously between each other [**?**, Rus02].

τ_1, the first task of a pipeline, is the source task. $\tau_1 = \tau_{src}$. τ_N, the last task of a pipeline, is the sink task. $\tau_N = \tau_{sink}$. τ_1 does not wait or block for its input data because we assume that an input is always available for τ_1. It is realistic since the source generally reads from a sensor input or digital media. In absence of new input data, the source task sends the recent available data [**?**]. The same assumption applies for the output of a sink task.

5.1.3 Scheduling Model

The system schedules all the tasks using the rate-monotonic scheduling (RMS) algorithm [LL73]. We assume that each periodic task τ_i, running in a sporadic server abstraction [SSL89] with a processor capacity reserve [MST93], will have a maximum runtime of C_i time-units in every T_i time-units, as it is implemented in an RTOS like Quest [DLW11]. We choose the RMS algortihm because it is a fixed-priority scheduling algorithm with low

runtime overhead, and many popular RTOSs already support it [B+08, BS14, DLW11].

Each task gets a priority assigned by the RMS algorithm. If two tasks of the same pipeline have the same periods, then the earlier appearing task is given higher priority. In other words, $prio(\tau_i) > prio(\tau_j)$, if $i < j$ and $T_i = T_j$. $prio(\tau_i)$ is the priority of task assigned by RMS.

If a task has already finished its work for a job invocation, it yields and does not start its next job until next period. This ensures that a single job invocation of a task does not overwrite its already written output in an asynchronous communication. Moreover, the fixed execution time tightly bounds a pipeline's end-to-end latency [GCU+21].

5.2 Pipeline Constraints

We consider constraints on the two pipeline parameters and on the total task utilization. Two pipeline parameters, end-to-end delay and loss-rate, are computed from the ordered taskset S. We first discuss a computational analysis of the parameters, and then present the constraints on them.

5.2.1 End-to-end Delay (E) Computation

The worst-case end-to-end delay of a pipeline is the maximum time for a message to appear at τ_{src} and emit from τ_{sink}. It is also known as the maximum reaction time [FRNJ08] in a cause-effect chain [BDM+16a, AUT17].

Davare *et al.* presented the first but conservative upper bound on the worst-case end-to-end delay for a pipeline of periodic tasks with arbitrary budgets and periods [DZDN+07]. If R_i is the worst-case response-time of τ_i, then the worst-case end-to-end delay is the

following:

$$E = \sum_{i=1}^{N} T_i + R_i \tag{5.1}$$

In the above equation, R_i is recursively calculated by initially estimating to be equal to the task period T_i for each task τ_i [JP86]. As Equation 5.1 is a recursive equation, the time-complexity of computing the equation depends on the wanted precision on E. Response-time calculation for fixed-priority scheduling is known to be NP-hard [ER08]. Nevertheless, a bounded computation time is preferred in a runtime task scheduling algorithm. Therefore, R_i is replaced in the above equation with T_i. If τ_i is feasibly scheduled, then R_i is less than or equal to T_i. Therefore, a faster computable version of Equation 5.1 is the following:

$$E = 2 \times \sum_{i=1}^{N} T_i \tag{5.2}$$

In offline or slower design-time analysis, Equation 5.1 is tolerable. For faster analysis and use in runtime scheduling, Equation 5.2 is preferable.

Dürr *et al.* tightened Equation 5.1 by considering task priorities between pairs of communicating tasks in a pipeline [DBCC19]. They use forward and backward cause-effect chains to derive a stricter upper bound on E2E delay. After converting the sproadic task model to the periodic task model as done in a subsequent work [GCU$^+$21], the worst-case E2E delay of a pipeline considering all the tasks are released at the critical instant [LL73] (initial release offset is 0), as proved by Dürr *et al.*, is the following:

$$E \leq T_1 + R_N + \sum_{i=1}^{N-1} max(R_i, T_{i+1} + R_i \times I) \tag{5.3}$$

In above equation, I is Iverson bracket. $I = 1$, if $(i+1)^{th}$ task has higher priority than i^{th} task. $I = 0$, otherwise.

As it is done for Equation 5.1 and 5.2, the following equation is a conservative but

faster computable version of Equation 5.3:

$$E \leq T_1 + T_N + \sum_{i=1}^{N-1} max(T_i, T_{i+1} + T_i \times I) \tag{5.4}$$

The time-complexity of Equation 5.2 and 5.4 is $\mathcal{O}(N)$. Equation 5.2 and 5.4 are useful in designing a runtime end-to-end pipeline scheduling algorithm. In this paper, Equation 5.4 is used for the uniprocessor pipeline scheduling. For multiprocessor scheduling, Davare *et al.*'s Equation 5.2 is used to avoid dependencies on task priorities in a pipeline, as a pipelined task could be mapped to any processor.

5.2.1.1 Example

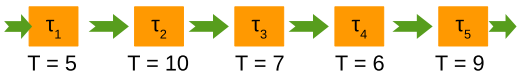


Figure 5.3: End-to-end Delay Example

Figure 5.3 shows an example pipeline. The end-to-end delays are 74 and 63, respectively with Equation 5.2 and Equation 5.4, if the tasks are scheduled with RMS. The calculation of Equation 5.4 for the example is the following:

$$\begin{aligned} E &\leq 5 + 9 + (max(5, 10 + 5 \times 0) + max(10, 7 + 10 \times 1) + \\ &\quad max(7, 6 + 7 \times 1) + max(6, 9 + 6 \times 0)) \\ &\leq 5 + 9 + (10 + (10 + 7) + (7 + 6) + 9) \\ &\leq 63 \end{aligned}$$

5.2.2 End-to-end Loss-rate (L) Computation

Data loss is an issue in systems involving sensors and actuators [ZZLN09, KXL+13, GSW20] The end-to-end loss-rate of a pipeline is the number of input messages that do not have a corresponding output, per input message to a pipeline, over the period of its first task. Suppose, the total number of input messages per period of the first task of a pipeline is I, and the number of corresponding output messages for I inputs is O, then loss-rate is defined by Equation 5.5. If O is greater than or equal to I, then no messages are lost, and the loss-rate is deemed 0. Loss-rate can also be realized in terms of Feiertag *et al.*'s concept of reachability [FRNJ08], where it is the ratio of *non-reachable* messages to the total number of input messages per period of the first task of a pipeline.

$$\begin{aligned} L &= \frac{I - O}{I}, \text{if } O \text{ ¡ } I \\ &= 0, \text{if } O \geq I \end{aligned} \tag{5.5}$$

Input (and output) messages are usually generated (and sent) from a sensor (and to an actuator), associated to an I/O buffer. However, an input may not come from an external buffer or input device and may just be generated by the source task of a pipeline. In that case, the generated messages are considered to be the inputs to a pipeline.

To calculate the loss-rate, we assume that the pipelined tasks are feasibly scheduled using a real-time scheduling algorithm following the scheduling model described in Section 5.1.3. For every pair of producer-consumer tasks ($\tau_p \rightarrow \tau_c$) in a pipeline, the consumer could either oversample ($T_c \leq T_p$) or undersample ($T_c > T_p$) its input from its corresponding producer. Based on the relationships between the periods of all the producer-consumer pairs starting from the source task to the sink task, the end-to-end loss-rate of a pipeline is calculated. The calculation is explained later in this section.

5.2.2.1 Sampling Ratio

We define the sampling ratio $f_{\tau_p \to \tau_c}$ of a producer-consumer pair ($\tau_p \to \tau_c$) as the number of output messages of τ_c per unique input message of τ_p. According to the task and scheduling model, a task generates a single message in their runtime budget per period and retires until its next invocation. Therefore, $f_{\tau_p \to \tau_c}$ is calculated from the producer's period divided by the consumer's period:

$$f_{\tau_p \to \tau_c} = \frac{T_p}{T_c} \tag{5.6}$$

Then, the loss-rate of a producer-consumer pair is $(1 - f_{\tau_p \to \tau_c}) = (1 - \frac{T_p}{T_c})$, if $f_{\tau_p \to \tau_c} \geq 1$, 0 otherwise. Examples are given later in the section.

Oversampling In case of an oversampling consumer ($T_p \geq T_c$), the data from the producer will be overrepresented in the output by the consumer. For example, consider $\tau_p = (C_p = 2, T_p = 40), \tau_c = (C_c = 1, T_c = 10)$. τ_p runs for 2 time-units in every 40 time-units and reads, processes, and writes a single input message. τ_c does the same in 1 time-unit in every 10 time-units. Therefore, τ_c will emit the same output 4 times for a unique input of τ_p. Hence, the sampling ratio is $\frac{T_p}{T_c} = 4$. Therefore, the *oversampling ratio* is: $O_{\tau_p \to \tau_c} = f_{\tau_p \to \tau_c} = \frac{T_p}{T_c} \geq 1$. The loss-rate is 0 in this case as sampling ratio is more than 1. This means that no messages are lost in this producer-consumer pair.

Undersampling The case of an undersampling consumer ($T_p < T_c$) is more nuanced because data might be lost. The data from a producer might be overwritten before a consumer has read it, as the consumer has larger period. For example, consider $\tau_p = (C_p = 1, T_p = 10), \tau_c = (C_c = 5, T_c = 40)$. τ_p takes 1 time-unit in every 10 time-units to read, process and finally output a message for τ_c. τ_c takes 5 time-units in

Table 5.1: Sampling Ratio Calculation Rules

Rules to Calculate Lower Bound on the Sampling Ratio of a resultant pipeline P_y after adding a new τ_{new} task to a pipeline P_x.

Rule	$P_x(\tau_1 \rightarrow ... \rightarrow \tau_{N_x})$	τ_{new}	Lower Bound on new Sampling Ratio (f_y)
1	Undersampled	Oversampled	f_x
2	Undersampled	Undersampled	$f_x \times \frac{T_{N_x}}{T_{new}}$
3	Oversampled	Oversampled	
4	Oversampled	Undersampled	

every 40 time-units to read a single τ_p's message, process and write its own single output message. Therefore, τ_p will run 4 times and produce 4 unique messages in 40 time-units. However, τ_c only runs once in 40 time-units and works with only 1 of 4 messages produced by τ_p. Therefore, the sampling ratio is $\frac{T_p}{T_c} = \frac{1}{4}$. Therefore, *undersampling ratio*: $U_{\tau_p \rightarrow \tau_c} = f_{\tau_p \rightarrow \tau_c} = \frac{T_p}{T_c} < 1$. Hence, the loss-rate here is $(1 - \frac{1}{4})$ = $\frac{3}{4}$ or 75%.

Pipeline Sampling Ratio Let's consider a pipeline $P_x = \{S_x\}$. Subscript x is used to distinguish a pipeline. P_x has two communicating tasks ($S_x = \{\tau_1, \tau_2\}, N_x = 2$) i.e., a single producer-consumer pair ($\tau_1 \rightarrow \tau_2$). The *sampling ratio* is f_x = $\frac{T_1}{T_2}$. The whole pipeline is *oversampled* if $f_x \geq 1$ and *undersampled* if $f_x < 1$.

Now, consider that P_x is extended by adding a new task τ_{new} at the end of P_x. A new pipeline P_y is thus formed whose ordered taskset S_y is $\{\tau_1, \tau_2, \tau_{new}\}$, and length is $N_y = N_x + 1 = 3$. τ_{new} could be oversampling or undersampling compared to the last task in P_x, τ_2 or τ_{N_x}.

We want to calculate a lower bound on the sampling ratio to derive an upper bound on the loss-rate. We define 4 rules to calculate a lower bound on the sampling ratio of a pipeline. The rules are summarized in Table 5.1 and proved below:

Rule 1 *If a pipeline P_x is undersampled, adding an oversampling task whose period is smaller than the period of P_x's last task τ_{N_x}, does not change the lower bound of the resultant pipeline's sampling ratio.*

Proof. Let's take an undersampling pipeline P_x. Its sampling ratio is $f_x = \frac{B}{A}$. It produces B output messages for every A input message to a pipeline. $B < A$.

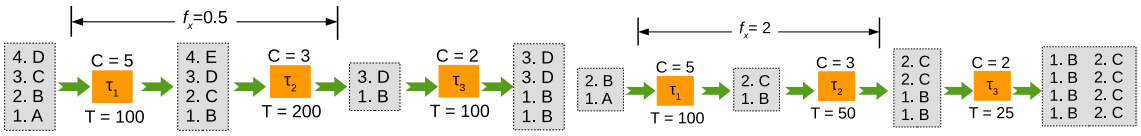
A new oversampling τ_{new} task is added to P_x to form pipeline P_y. Therefore, τ_{new}'s period is lesser than or equal to the period of P_x's last task τ_{N_x}. $T_{new} \leq T_{N_x}$.

For every new input message to τ_{new}, $O_{\tau_{N_x} \rightarrow \tau_{new}}$ =($\frac{T_{N_x}}{T_{new}} \geq 1$) number of output messages are produced by τ_{new}. Therefore, for every B outputs from P_x to τ_{new}, $(B \times O_{\tau_{N_x} \rightarrow \tau_{new}})$ outputs are produced by τ_{new}.

However, $(B - A)$ number of messages are already lost in the pipeline P_x. The oversampling task τ_{new} cannot recover those messages. Therefore, $(B \times O_{\tau_{N_x} \rightarrow \tau_{new}})$ messages just represent the oversampled messages produced by task τ_{new}. The number of unique output messages per input message of the pipeline remains same. Therefore, P_x's sampling ratio (f_x) remains the lower bound of P_y's sampling ratio. ■

Example Consider the example given in Figure 5.4a. $P_x = \{S_x = (\tau_1, \tau_2)\}$. $N_x = 2$. P_y is formed by adding τ_3 to P_x. τ_1's input is given at the left most side. Each line represents a unique message with from 1 to 4 as the unique IDs of the messages. τ_1 reads one message in its single job invocation, increases the ASCII value of the input and writes to its output buffer. τ_2 and τ_3 reads an input message and just passes through to the output buffer.

P_x's sampling ratio f_x is $\frac{T_1}{T_2}$ = 0.5. We can see that τ_2 emits 1^{st} (`B`) and 3^{rd} (`D`) message, although it receives `B,C,D,E` as inputs. τ_2 emits one out of every two input messages. Now, we add τ_3 after τ_2. τ_3 is oversampling with respect to τ_2, exemplifying *Rule 1*. As τ_3 runs twice frequently than τ_2, it replicates one input message two times in its output. Therefore, it emits `B,B,D,D` for `B,D` inputs coming from τ_2. However, the repetitions do not recover the lost messages `A,C`. Therefore, the lower bound of the sampling ratio of the new pipeline P_y remains the sampling ratio of pipeline P_x. In this case, that is 0.5.



(a) Rule 1 **(b)** Rule 3

Figure 5.4: Examples of Pipeline Sampling Ratio Calculation

Rule 2 *If an undersampling task τ_{new} is added at the end of an undersampled pipeline P_x, then the resultant sampling ratio is lower bounded by the undersampling ratio of P_x, f_x, multiplied by the undersampling ratio of the last task of P_x and the new task .*

Proof. Let's take the same undersampling pipeline P_x from *Rule 1*. $f_x = \frac{B}{A}$ and $B < A$. An undersampling τ_{new} task is added to P_x to form pipeline P_y. Therefore, τ_{new}'s period is greater than the period of P_x's last task τ_{N_x}. $T_{new} > T_{N_x}$.

For every new input message to τ_{new}, $\frac{T_{N_x}}{T_{new}} = U_{\tau_{N_x} \to \tau_{new}} (< 1)$ outputs are produced by τ_{new}. Therefore, for every B inputs to τ_{new}, only $B \times U_{\tau_{N_x} \to \tau_{new}}$ messages are produced. Hence, the sampling ratio of the newly formed pipeline P_y: $f_y = \frac{B \times U_{\tau_{N_x} \to \tau_{new}}}{A} = f_x \times \frac{T_{N_x}}{T_{new}}$. f_y provides the lower bound for the new pipeline P_y. ■

Example Consider the same example given in Figure 5.4a but with period of τ_3 = 400. In this case. τ_3 emits only the 1^{st} (B) message. Therefore, the sampling ratio is $0.5 \times \frac{200}{400}$ = 0.25.

Rule 3 *If an oversampling task τ_{new} is added at the end of an oversampled pipeline P_x, then the resultant sampling ratio is lower bounded by f_x multiplied by the oversampling ratio of the last task of P_x and the new task.*

Proof. Let's consider an oversampling pipeline P_x. $f_x = \frac{B}{A}$ and $B \geq A$. An oversampling task τ_{new} is added to P_x to form pipeline P_y. $T_{new} \leq T_{N_x}$. For every B outputs from P_x to τ_{new}, $(B \times O_{\tau_{N_x} \to \tau_{new}})$ outputs are produced by τ_{new}.

Therefore, the sampling ratio of the newly formed pipeline P_y: $f_y = \frac{B \times O_{\tau_{N_x} \rightarrow \tau_{new}}}{A} = f_x \times \frac{T_{N_x}}{T_{new}}$. f_y is the lower bound on the sampling ratio of P_y. ■

Example Consider the example in Figure 5.4b which is similar to Figure 5.4a but with different periods. The sampling ratio after adding τ_3 at the end of $P_x(S_x = \{\tau_1, \tau_2\})$ is 4. Here, 4 accurately represents the oversampling ratio.

Rule 4 *If an undersampling task τ_{new} is added to an oversampled pipeline P_x, the resultant sampling ratio is f_x multiplied by the undersampling ratio of P_x's last task and the new task .*

Proof. Let's consider the same oversampling pipeline from the above rule, P_x. $f_x = \frac{B}{A} \geq 1$. An undersampling task τ_{new} is added to P_x to form pipeline P_y. $T_{new} > T_{N_x}$. For every B input messages to τ_{new}, it produces only $(B \times U_{\tau_{N_x} \rightarrow \tau_{new}})$ messages that is less than B number of messages, as $U_{\tau_{N_x} \rightarrow \tau_{new}} < 1$. We already know that P_x produces f_x output messages for every single input to it.

Therefore, the sampling ratio of the newly formed pipeline P_y: $f_y = f_x \times U_{\tau_{N_x} \rightarrow \tau_{new}} = f_x \times \frac{T_{N_x}}{T_{new}}$. f_y also provides the lower bound on the sampling ratio of P_y. ■

Example Consider the example in Figure 5.4b but with T_3 = 200. The sampling ratio between τ_2 and τ_3 is $\frac{50}{200} = \frac{1}{4}$. Therefore, τ_3 is able to output only 1 message for every 4 input messages coming from τ_2. So, it will only output `B` in the example. Therefore, the lower bound on the full pipeline's sampling ratio is $f_x \times \frac{1}{4} = 2 \times \frac{1}{4} = \frac{1}{2}$. As we can see that two messages (`A,B`) were input to the pipeline, but only 1 message was output.

Following the above rules, the sampling ratio of a pipeline could be recursively derived by calculating the sampling ratio from the source task of a pipeline until the sink, treating a next task in a pipeline as τ_{new}. The first producer-consumer pair is considered a pipeline

P_x. Then, a new task is added at the end of P_x pipeline, and a new sampling ratio is calculated. By the end of the pipeline where there are no more tasks, end-to-end sampling ratio is computed.

5.2.2.2 Upper Bound on Loss-rate from Sampling Ratio

The upper bound on a pipeline P's loss-rate is expressed in terms of its sampling ratio f as follows:

$$\begin{aligned} L &= 0, \text{if } f_P \geq 1 \\ &\leq 1 - f_P, \text{if } f_P < 1 \end{aligned} \tag{5.7}$$

5.2.3 Formalization of Pipeline Constraints

Equations 5.4 and 5.7 show the upper bounds of the end-to-end delay and loss-rate of a pipeline. A task's runtime budget C to read, process and write 1 message is usually determined by performing a worst-case execution time (WCET) analysis [WEE+08]. A pipeline is then constructed by chaining up these periodic tasks. End-to-end delay and loss-rate constraints are applied on a pipeline to guarantee a certain level of quality of service. The unknown variable here is the periods of the individual tasks of a pipeline, and thus, the problem is to find the task periods. The constraints are summarized below:

1. E should be upper bounded by a constant E^{UB}, meaning that the worst-case end-to-end delay computed from Equation 5.4 should not exceed E^{UB}.

2. L should be upper bounded by L^{UB}, meaning that the loss-rate computed from Equation 5.7 should not exceed L^{UB}.

3. The total pipeline utilization should be within a constant upper bound of U^{UB}. This

constraint comes from the scheduling model, in our case, the RMS bound (See Section 5.1.3).

Constraint 1, 2 and *3* for a pipeline are respectively formalized in Equation 5.8, 5.9, and 5.10

$$T_1 + T_N + \sum_{i=1}^{N-1} max(T_i, T_{i+1} + T_i \times I) \leq E^{UB} \tag{5.8}$$

$$L \leq L^{UB} \tag{5.9}$$

$$\sum_{\forall \tau_i \in S} \frac{C_i}{T_i} \leq U^{UB} \tag{5.10}$$

5.2.4 Problem Statement

Given a list of pipelined tasks' budgets ($C_i, \forall \tau_i \in S$) and a set of constant upper bounds (E^{UB}, L^{UB}, U^{UB}) as constraints, the challenge is to find suitable $T_1, T_2, \ldots, T_N$, such that the constraints are satisfied,

To be precise, Equation 5.8, 5.9 and 5.10 need to be satisfied to find a set of suitable periods. This is a constraint programming problem that is in the class of integer nonlinear programming problems, assuming integer task periods. It is a nonlinear programming problem because of Equation 5.10 where the period is in the denominator. The problem is known to be NP-hard [HKLW10].

5.2.4.1 Budget Adjustment

Until now, we have considered the task runtime budget to be a constant C in the optimization problem. This requirement could be relaxed by allowing increment of task budgets in integer multiples of C. If a task's budget is decreased from the initial budget C, it would be unable to read, process and write 1 message or data-unit, assuming that a fragment of 1 data-unit is invalid.

When a task τ's budget is increased from C to MC (M is an integer constant greater than 1), it means τ processes M messages in its single job invocation. τ's input and output buffer sizes are also increased to pass M messages in each slot of the four-slot buffer.

Additional Constraint The constraints for the budget adjustment are the following:

$$M_i \geq 1 \tag{5.11}$$

$$\forall_{\tau_i \in S}\ FC_i = M_i \times C_i \tag{5.12}$$

FC_i is the final allocated runtime budget of τ_i. M_i is called a budget multiplier.

Our heuristic constraint solver algorithm CoPi uses the budget adjustment mechanism to bound the other pipeline constraints. The details are in Section 5.3.2.2. However, these budget adjustment constraint equations are not supplied to the open-source solvers, by default. Therefore, they use only Equations 5.8, 5.9 and 5.10, unless otherwise specified.

Loss-rate Recalculation after Budget Adjustment Sampling ratio formula is extended from the Equation 5.6 to accommodate the budget adjustment by CoPi:

$$f_{\tau_p \to \tau_c} = \frac{T_p}{T_c} \times \frac{M_c}{M_p} \tag{5.13}$$

M_c and M_p are respectively the consumer and producer budget multipliers from Equation 5.11. New loss-rate is calculated from the adjusted sampling ratio.

5.2.4.2 MINLP Solvers

We have modeled the pipeline constraints in three open-source Mixed-Integer Non-Linear Programming (MINLP) solvers in Python: GEKKO [BHMH18], pyomo [HWW11, BHH+21] and scipy [sci22]. We compare their performances to our heuristic constraint

solver CoPi's performance in terms of the number of accepted task pipelines in Section 5.5. We have also considered GNU Linear Programming Kit [GNU21], Google-OR Tools [Goo21b] and other solvers, but they lack integer nonlinear programming features, and the above ones suffice for the purpose of this work.

5.3 CoPi: Pipeline Constraint Solver Heuristic

This section explains our heuristic constraint solver algorithm for uniprocessor scheduling.

5.3.1 CoPi's Objective and Approach

The primary objective of CoPi, our constraint solver heuristic for end-to-end scheduling of a real-time task pipeline, is to avoid unnecessary delay and data loss among the communicating tasks. Once the data-dependencies between the pipelined tasks are handled by tuning the task parameters, all the tasks run independent of each other without waiting for job release and completion times [BDM+16a, CKK20].

Given the initial task budgets, the upper bounds of the pipeline parameters (E2E delay and loss-rate) and the RMS utilization bound, CoPi derives the task periods and new runtime budgets. Gerber *et al.* also proposed a similar approach of deriving task periods, offsets and other parameters from the end-to-end constraints, albeit on task precedence relations [GSS95]. We utilize the core idea of deriving suitable budgets and periods from the end-to-end requirements, so that the pipelined tasks could be independently executed.

For a multiprocessor system, runtime task migrations are feasible because of CoPi's conversion of pipelined tasks to independent asynchronus tasks (see Figure 5.2). Although process to core mapping and migration should also consider cache, memory and other microarchitectural properties, it should be handled at the system implementation level, and needed properties could be added to an extended task and scheduling model.

5.3.2 CoPi Heuristic Algorithm

Pseudocode of CoPi is provided in Algorithm 5.1. It finds the suitable periods of a pipeline of N tasks under all the constraints constraints (*Constraint 1, 2* and *3* from Section 5.2.3).

CoPi takes the initial task runtime budgets (C in the model and `budgets` in Algorithm 5.1) and the desired upper bound on the end-to-end delay and loss-rate (E^{UB} and L^{UB} in the model, and `e2e_ub` and `lr_ub` in Algorithm 5.1) as its inputs. `budgets` are given in the same order as in the ordered taskset S in the pipeline model. α and β are CoPi's internal tuning parameters that are also taken as inputs and explained later.

5.3.2.1 Stage 1

Line 8–11 in Algorithm 5.1 show Stage 1. CoPi starts by setting all the task periods to be the same in Line 9: $eq\_period = \frac{e2e\_ub}{N+1}$, where N is the pipeline length. By trying to assign the same equal period to all the tasks, CoPi tries to eliminate any loss between the pipelined tasks. Thus, any loss-rate upper bound constraint is satisfied, as loss-rate is 0 for equal task periods.

To satisfy the end-to-end delay constraint, $e2e\_ub$ is divided by $(N+1)$ instead of N. As per the scheduling model in Section 5.1.3, the earlier appearing task in a pipeline is given higher priority in RMS algorithm for tasks with equal periods. Therefore, the upper bound on end-to-end delay following Dürr *et al.*'s Equation 5.4 is: $E \leq \Big((N+1) \times eq\_period\Big)$ [DBCC19]. Thus, the end-to-end delay constraint is implicitly satisfied by the choice of equal task periods of $(\frac{e2e\_ub}{N+1})$: $E \leq \Big((N+1) \times eq\_period\Big) = e2e\_ub$.

As both the pipeline constraints are satisfied, CoPi checks the schedulability of the task pipeline with the RMS utilization bound constraint in Line 10.
`utilization_bound_test(taskset)` is briefly the following: $\big(\sum_{i=1}^{N} \frac{C_i}{T_i} \leq n \times (2^{\frac{1}{n}} - 1)\big)$.

Algorithm 5.1 CoPi Pipeline Constraint Solver Algorithm

Input: $budgets[N]$ - Budgets of N Pipelined Tasks in an ordered sequence from source to destination
Input: $e2e\_ub$ - upper bound of end-to-end delay
Input: lr_ub - upper bound of end-to-end loss-rate
Input: $util\_ub$ - upper bound on processor utilization. Used if less than the RMS utilization bound.
Input: α - The multiplicative scaling factor
Input: β - The divisive scaling factor
Output: If schedulable: an ordered taskset with budgets and periods, else: not schedulable.
`// Stage 1`
$eq\_period = \frac{e2e\_ub}{N+1}$; $taskset = [(b, eq\_period)$ for b in $budgets]$
if `utilization_bound_test(taskset)` **then return** $taskset$
end if
`// Stage 2`
$scaled\_period = \alpha \times eq\_period$; $taskset = [(b, scaled\_period)$ for b in $budgets]$
while $True$ **do**
    $one\_pipe\_changed = False$
    for $i = 0$ to $N - 2$ **do**
        $producer = taskset[i]$; $consumer = taskset[i+1]$
        if $producer.budget < \frac{producer.period}{\beta}$ and $consumer.budget \times \beta < consumer.period$ **then**
            $producer = (producer.budget, \frac{producer.period}{\beta})$
            $consumer = (\beta \times consumer.budget, consumer.period)$
            if `utilization_bound_test (taskset)` **then**
                $one\_pipe\_changed = True$
                if `total_e2e_delay(taskset)` $\leq e2e\_ub$ and `loss_rate(taskset)` $\leq lr\_ub$ **then return** $taskset$
                end if
            end if
        end if
    end for
    if not $one\_pipe\_changed$ **then** break
    end if
end while
`// Stage 3`
for $i = N - 1$ to 0 **do**
    $cur\_task = taskset[i]$; $cur\_budget = cur\_task.budget$; $cur\_period = cur\_task.period$
    $init\_budget = cur\_task.init\_budget$
    while $\frac{cur\_budget}{\beta} \geq init\_budget$ **do**
        $cur\_budget = \frac{cur\_budget}{\beta}$; $cur\_period = \frac{cur\_period}{\beta}$
    end while
    $taskset[i] = (cur\_budget, cur\_period)$
    if `utilization_bound_test(taskset)` and `total_e2e_delay(taskset)` $\leq e2e\_ub$ and `loss_rate(taskset)` $\leq lr\_ub$ **then return** $taskset$
    end if
end for
return none

For a large value of $e2e\_ub$, Stage 1 itself returns a schedulable pipeline, as the total utilization of the pipelined tasks would be small. For smaller and tighter values of $e2e\_ub$, CoPi moves on to the Stage 2.

5.3.2.2 Stage 2

Line 12–30 in Algorithm 5.1 show the Stage 2. In this stage, CoPi tries adjusting the task periods to bring down the total utilization while satisfying the end-to-end delay and loss-rate constraints. In order to do so, in step 1, it scales up all the task periods by a constant factor (α) to reduce the total task utilization. However, that increases the end-to-end delay and violates the constraint. Then in step 2, CoPi considers all the producer-consumer pairs one by one. It scales down the period of a producer by a constant integer factor (β) and scales up the runtime budget of a corresponding consumer by the same factor β, in an effort to bring down the end-to-end delay and keep the loss-rate under constraint.

Rate-matching Heuristic Before explaining Stage 2 in Algorithm 5.1, we explain how CoPi adjusts the budget and periods. To reiterate from Section 5.2.4.1, when CoPi changes a task τ's runtime budget from C to $(\beta \times C)$, it implies that the task now reads, processes and writes β number of messages or data-units in a single job invocation.

Consider an example given in Figure 5.5. A pipeline is shown with $S = \{A, B\}$ and their initial runtime budgets of 2 and 4, respectively. Imagine after step 1 of stage 2, CoPi assigns period of 80 to both the tasks. This pipeline is shown at the top of Figure 5.5. The end-to-end delay of this pipeline is 240, as calculated using Equation 5.4.

Then, CoPi divides the producer A's period by β and also multiplies the consumer B's budget by β, with β=2. The resultant pipeline is at the bottom of Figure 5.5. The end-to-end delay is reduced to 160 from 240 time units. In the new pipeline, B will consume 2 messages in its single job invocation, with a runtime budget of 8. A will still produce

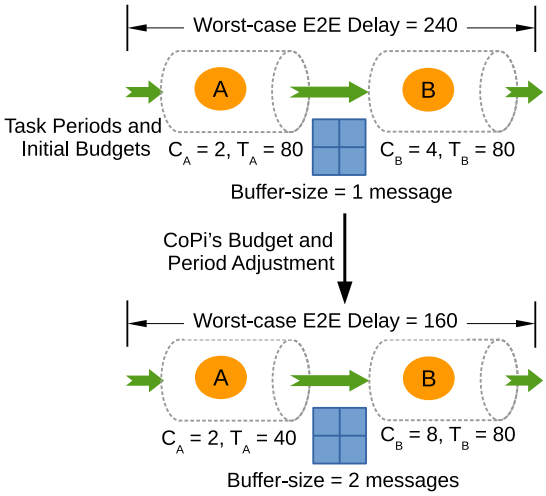


Figure 5.5: Period and Budget Adjustment Example

1 message in its runtime because its budget is unchanged. But A's period is halved. So it will run twice within a single period of B. Therefore, the capacity of the four-slot asynchronous buffer is increased to store 2 messages from A. In its single job invocation, B will now consume those 2 messages. Therefore, the final sampling rate of the pipeline is still $(\frac{40}{80} \times \frac{2}{1}) = 1$.

By adjusting the budgets and periods, end-to-end latency is reduced while keeping the loss-rate to 0. Although the total utilization increases, it is already much smaller because of scaling the period by α. This is the reason CoPi succeeds in scheduling task pipelines under the constraints.

CoPi enforces that task runtime budgets are never decreased from C, as fraction of a message is invalid. Also, increments to budgets are only in integer multiples of C.

Explanation of Stage 2 In this stage, CoPi first multiplies the equal period from stage 1 by a factor α (> 1) in Line 13. α is empirically chosen from a range of values and given as an input to CoPi. Stretching the period by an α factor helps in lowering the pipelined tasks' total utilization and keeps it under the RMS bound. However, it violates the end-to-end delay constraint. So CoPi tries reducing the task periods, satisfying the other constraints.

Starting from the first pair of producer and consumer: producer's period is divided by β (> 1) and consumer's budget is multiplied by β, as long as the periods are greater than the task budgets (Line 16–20). This reduces the end-to-end latency while keeping the producer-consumer rate-matched for minimal data-loss.

Moving forward, CoPi checks whether utilization bound test is satisfied in Line 21. If it passes that constraint, then end-to-end delay and loss-rate constraints are checked in Line 23. If all the constraints are satisfied, a schedulable pipeline with new budget and period assignments to its tasks is returned.

If a constraint is violated, the algorithm moves on to the next pair in the pipeline and repeat the steps until it covers all the producer-consumer pairs in the pipeline. After completing an iteration of going through all the producer-consumer pairs of a pipeline, CoPi again starts from the first pair for another iteration in Line 16. If no pair could be tuned for an iteration, tracked by the $one\_pipe\_changed$ boolean variable, CoPi moves on to stage 3.

5.3.2.3 Stage 3

Lines 31–41 in Algorithm 5.1 show CoPi's Stage 3. The objective of this stage is to further reduce the E2E delay, while keeping the total task utilization same and loss-rate under its upper bound.

In this stage, CoPi scales down the budgets and periods of the tuned consumers of stage 2. The stage starts from the sink or the last task of a pipeline and goes until the source task. It divides both periods and budgets of a task by β, as long as the budget is more than or equal to the initial budget of the task to process a single message. In each iteration, the constraints are checked in Line 39 and if they are satisfied, a feasible task pipeline is returned. Otherwise, CoPi declares the pipeline unschedulable.

5.3.2.4 Discussion

CoPi runs Stage 1 only once for a pipeline, and it runs Stage 2 and 3 multiple times with different α values. In all of our experiments, we fix $\beta = 2$ that is empirically chosen, while we test Stage 2 and 3 with α in a range of 1.01 and 2. Next, we establish a lower bound on α in Equation 5.14 to minimize runtime overhead in Section 5.3.2.5. The steps of incrementing α and its higher bound can also be used to control the runtime overhead of the algorithm.

5.3.2.5 Lower Bound on α

The equal period derived in stage 1 is $T_{eq} = \frac{E^{UB}}{N+1}$. Therefore, the total task pipeline utilization is $\sum_{i=1}^{N} \cdot \frac{C_i}{T_{eq}}$. CoPi moves to second stage because this total utilization is not under the utilization bound of RMS. So, $\sum_{i=1}^{N} \frac{C_i}{T_{eq}} > B$, where $B = n \times (2^{\frac{1}{n}} - 1)$.

In step 1 of stage 2, CoPi scales the equal period by α. Therefore, the total utilization after scaling must be less than or equal to B. Otherwise, CoPi will not be able to adjust the task budgets and periods in its next steps. So, we can write:

$$\sum_{i=1}^{N} \frac{C_i}{\alpha \times T_{eq}} \leq B$$

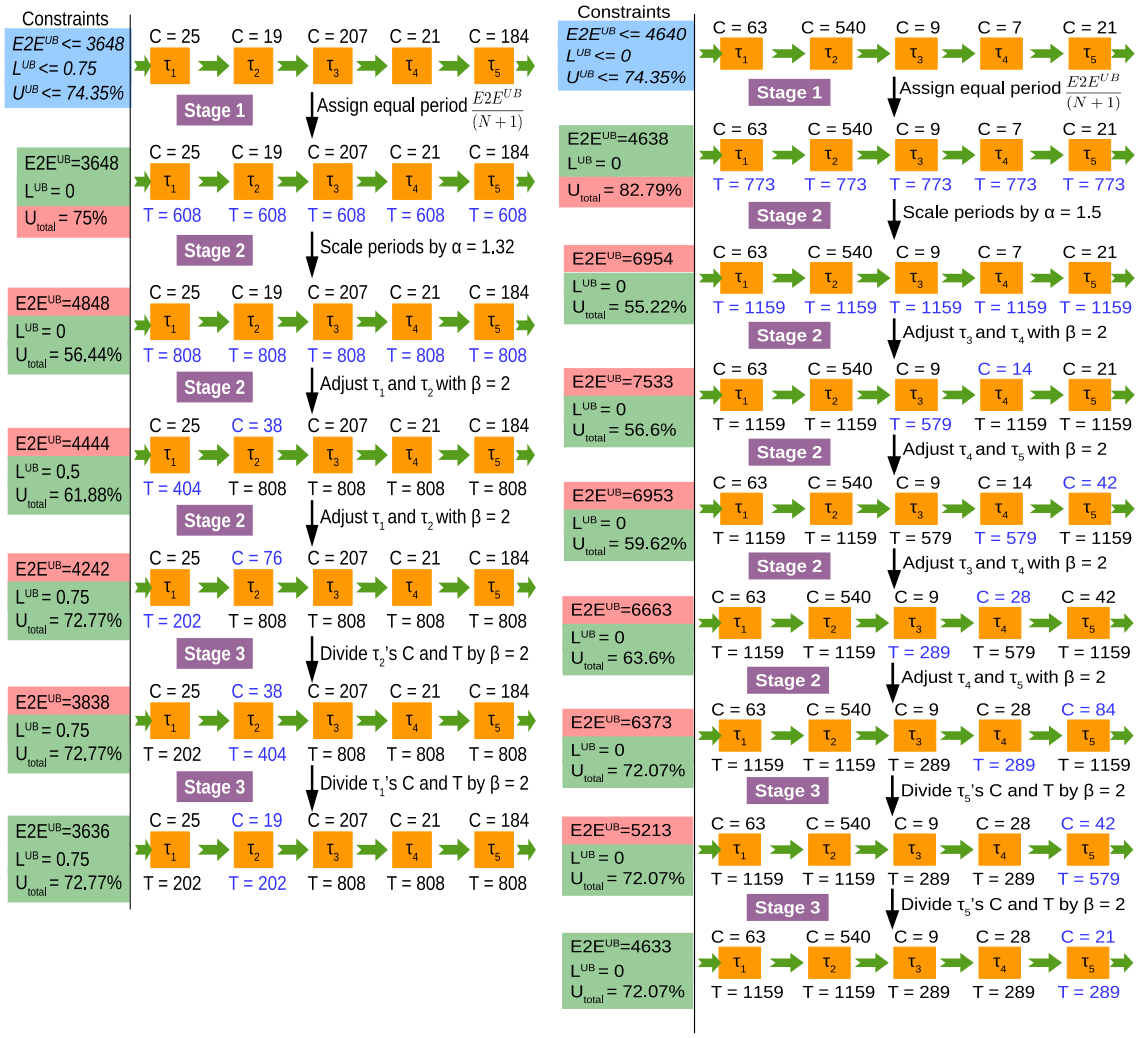
$$\sum_{i=1}^{N} \frac{C_i}{\frac{\alpha \times E^{UB}}{N+1}} \leq B$$

$$\frac{N+1}{\alpha \times E^{UB}} \sum_{i=1}^{N} C_i \leq B$$

$$\frac{N+1}{B \times E^{UB}} \sum_{i=1}^{N} C_i \leq \alpha$$

$$\frac{N+1}{N \times (2^{\frac{1}{N}} - 1) \times E^{UB}} \sum_{i=1}^{N} C_i \leq \alpha \text{ (for RMS)} \tag{5.14}$$

Equation 5.14 provides a starting value of α to CoPi with the above formula.



(a) CoPi Example 1 **(b)** CoPi Example 2

Figure 5.6: CoPi Examples

5.3.3 Examples

Figure 5.6a shows an example iteration of CoPi where it schedules a pipeline of 5 tasks. The constraints are shown at the top left. The utilization bound is the RMS bound for 5 tasks. We show a successful iteration where α is set to 1.32. The pipeline parameters are shown on the left at each stage. Their colors indicate if a parameter constraint is satisfied at a stage (green – satisfied, red – unsatisfied). Figure 5.6b shows another such example but with 0% loss-rate constraint or no data loss.

5.3.4 Execution Time Complexity of CoPi

For a pipeline of length N, stage 1 runs in $\mathcal{O}(N)$ time. Stage 2 iterates over the length of a pipeline multiple times, depending on the constraints. Stage 2 starts with a lower total pipeline utilization and goes up to the RMS utilization bound. Since it divides the task periods and multiplies the budgets by a constant β, the number of iterations in stage 2 is some constant. It is calculated by a function dependent on logarithm base α of task budgets, e2e delay upper bound and utilization bound. As all of them are constant, stage 2 approximately takes $\backsim \mathcal{O}(K \times N)$ (K is some constant, N is length of a pipeline), because it checks all the producer-consumer pairs in a pipeline.

Finally, stage 3 checks all the tasks in a pipeline and runs in $\mathcal{O}(N)$. Tuning every task in stage 3 also takes similar logarithmic function and could be assumed to be a constant $\backsim \mathcal{O}(J)$. Therefore, stage 2 and 3 together take approximately $\mathcal{O}((K + J) \times N)$, where $\mathcal{O}(K + J)$ represents the hardness of the constraints.

These stages are run for a limited range of α values. For example, if we test all α values between 1.01 and 2 with a 0.01 step increment, stage 2 and 3 are then run for 100 times. We could also find a feasible schedule before that. So we assume the total number of stage 2 and 3 runs to be a constant I. However, the lower bound of α has an inverse relationship with N. Therefore, the range of α values are higher for a larger N. So, I leans to $\backsim \mathcal{O}(N)$.

CoPi overall takes $\mathcal{O}\big(I \times (K + J) \times N\big)$, where $\mathcal{O}((K + J) \times I)$ is dependent on the constraints. Overall, CoPi's complexity is linear to quadratic with respect to a pipeline's length. Thus, CoPi is a good candidate as a helper heuristic to a scheduling algorithm at runtime. Section 5.5.2 presents an experimental analysis of CoPi's runtime overhead.

5.4 Multiprocessor Pipeline Scheduling

In multiprocessor scheduling, we utilize CoPi's feature of providing a set of independent and asynchronous tasks that are chained in a pipeline. Thus, the asynchronous tasks are free to be mapped to any available processor. Even runtime task migrations are possible, as there are no data, timing or priority dependencies between tasks after CoPi has derived the task parameters. We use Equation 5.2 by Davare *et al.* to derive the upper bound on the worst-case end-to-end delay, in contrast to Equation 5.8 used for uniprocessor, to avoid any priority dependencies between the tasks. In this way, a task could be mapped to any available processor as long as the tasks complete their jobs within their periods. Hence, CoPi uses the previous two Equations 5.9 and 5.10 from Section 5.2, and the Equation 5.15 given below in this case.

$$2 \times \sum_{i=1}^{N} \times T_i \leq E^{UB} \tag{5.15}$$

This multiprocessor pipeline scheduling algorithm demonstrates the benefits and application of the constraint solving approach and CoPi. CoPi is coupled with a traditional and well-known multiprocessor scheduling heuristic for this purpose. Although multiprocessor scheduling is an NP-hard problem [HvdVV94], there are well-known heuristics to map a set of tasks to a number of processors in polynomial time. We use the worst-fit decreasing (WFD) heuristic. In WFD heuristic, the algorithm maintains a sorted list of the pipelined tasks based on their utilization values and a sorted list of the processors based on the available utilization, both in decreasing order. Then, it maps a task from the sorted task list to a processor from the sorted processor list. After CoPi provides a set of tasks, the algorithm uses WFD to map a pipeline to the available processors. In addition, the algorithm implements a few other heuristics to improve the runtime acceptance ratio of new pipelines, that are explained in Section 5.4.1 and 5.4.2.

Algorithm 5.2 Multiprocessor Pipeline Scheduling

```
 1: Input: pipeline - Budgets of N Pipelined Tasks in ordered sequence from Source to Destination
 2: Input: e2e_ub - upper bound on end-to-end delay
 3: Input: loss_ub - upper bound on loss-rate
 4: Input: α - The multiplicative scaling factor
 5: Input: β - The divisive scaling factor
 6: Output: True, if a pipeline is accepted in the multiprocessor system, else False
 7: util_ub = get_total_util_from_all_procs()
 8: q_pipeline = CoPi(pipeline[budgets], e2e_ub, loss_ub, util_ub)
 9: if q_pipeline is None then
10:     Try reducing utilization of an already admitted pipeline.
11:     Then go back to Line 7.
12:     Do the above for only a limited number of time.
13: end if
14: iter = 0
15: while iter ≤ num_core do
16:     if WFD_FIT(q_pipeline) is not successful then
17:         // try migration
18:         Sort the processors with decreasing order of available utilization
19:         for each processor p in the above sorted list do
20:             Sort the mapped tasks in decreasing order of utilization in this processor
21:             for each task t in the above sorted list do
22:                 for each processor q among all processors do
23:                     if t's utilization ≤ q's available utilization and p ≠ q then
24:                         unmap t from p and map to q
25:                         break out of for loop at Line 19
26:                     end if
27:                 end for
28:             end for
29:         end for
30:         Increment iter
31:     else return True
32:     end if
33: end while
34: return False
```

The algorithm uses a partitioned RMS scheduling for separate processors, where each processor runs its mapped tasks following the RMS policy. RMS utilization bound [LL73] is checked for task acceptance to a particular processor. The algorithm tracks each processor's available utilization by initially approximating it to 0.69 – the RMS bound for infinite number of tasks.

The multiprocessor scheduling algorithm is sketched in Algorithm 5.2. An auxiliary scheduling driver code externally initializes the number of processors and their utilizations. Then, the driver calls Algorithm 5.2 to map a pipeline with constraints to available

processors. Algorithm 5.2 takes a pipeline and its constraints as input and returns `True` if the pipeline is accepted. Otherwise, it returns `False`.

Algorithm 5.2 works as follows. It first gets the sum of available utilization in all the processors in Line 7. This available utilization is fed to CoPi along with a new pipeline's task budgets and constraints in Line 8. CoPi either returns a pipeline with schedulable budget and period assignments, or returns `None` if the pipeline was unschedulable. For a schedulable pipeline, the algorithm tries mapping individual tasks to processors. A task is able to mapped to any available processor, since CoPi generates independent tasks.

Then, the algorithm uses `WFD_FIT` function for the WFD heuristic to map new pipelined tasks to processors in Line 16. It first determines whether all the pipelined tasks could be mapped, by checking the individual tasks one by one, before actually mapping the tasks to processors. Only if it could map all the tasks in a pipeline to the available processors, it maps them to the processors.

When WFD is not able to map all the tasks, the algorithm tries migrating tasks from one processor to another in Line 17– 29. The migration strategy is described later in Section 5.4.1.

Moreover, when CoPi is first called in Line 8, it may not return a feasible schedule because of not meeting the utilization bound. Such an infeasible schedule may occur due to unoptimized pipelines already admitted in the system. As the system starts with more available utilization, CoPi is initially run with a higher and relaxed utilization bound constraint. So it may have returned unoptimized pipelines because they were already schedulable with higher utilizations. In such cases, the multiprocessor scheduling algorithm optimizes an admitted pipeline in Line 10–12. The strategy is explained more in Section 5.4.2.

5.4.1 Runtime Task Migration

When the multiprocessor scheduling algorithm fails to schedule a new pipeline in its available processors even after having spare utilization, it explores the possibility of migrating already mapped tasks to make room for a new pipeline. Line 17– 29 show it in Algorithm 5.2. The algorithm first sorts the processors in decreasing order of available utilization. For each processor in the sorted list, it sorts the mapped tasks in decreasing order of task utilization. It picks a task from this sorted list of mapped tasks, and migrates it to the first available processor that can accommodate the task.

As soon as a task is migrated, the algorithm tries to schedule the new pipeline with WFD heuristic. We do this to minimize the number of total task migrations in the system, because migrations have practical runtime overhead. For a new pipeline, the algorithm only tries migrating M tasks at most, if M is the number of total processors. Thus, we limit the number of migration attempts per new pipeline to bound the time to find a schedulable mapping.

In summary, we employ task migration to admit more pipelines at runtime by creating smaller utilization holes in processors. However, task migration should be carefully administered and minimized as it is associated with non-negligible overhead and potential disruptions for admitted pipelines. Nevertheless, predictable migration [LWCM14] enables admission of new pipelines in a multiprocessor system. Our evaluation results in Section 5.5.4 demonstrates the benefit of migrations in terms of the number dynamically accepted pipelines.

5.4.2 Runtime Pipeline Optimization (RPO)

We provide another improvement technique for multiprocessor scheduling where we optimize an already admitted pipeline by reducing its total task utilization. When CoPi fails to

find schedulability for a new pipeline, Algorithm 5.2 picks an already admitted pipeline. It sends the old admitted pipeline's initial task budgets, old timing constraints and a *stricter* utilization bound constraint than the pipeline is currently using, to CoPi. For our experiments, Algorithm 5.2 asks CoPi to reduce a pipeline's current utilization by 5% at a time. If CoPi is able to find new task budgets and periods for the admitted pipeline with the new utilization bound constraint, the algorithm unmaps all the tasks of the pipeline from all the processors. It then remaps with new task parameters following the WFD heuristic. This strategy reduces the total processor utilization at runtime and also makes room for a new pipeline. Our evaluation shows that this strategy yields a higher number of pipeline admissions. However, *RPO* implementation in a working system should ensure that the task unmapping and mapping are done at a safe timing point. If a task τ_i is running, then the scheduler may wait T_i time before new runtime and budget can be applied. Therefore, the scheduler needs to wait $(\sum_{i=1}^{N} T_i)$ time units in the worst-case. An RPO implementation in an RTOS needs to be aware of such delays. The full analysis and implementation details of RPO are out of scope of the paper and left for future work. In this paper, we show the benefits of *RPO* in admitting new pipelines at runtime with simulated experiments in Section 5.5.4.

5.5 Evaluation

We evaluate CoPi and the multiprocessor scheduling algorithm on top of it by running simulated experiments. The artifacts of the experiments are available on `https://github.com/sohamm17/pipe_schedule`. We first generate individual task utilizations using the standard UUnifast algorithm [BB05]. Then, we generate the task budgets by multiplying the utilization with a random value chosen from uniform distribution between 100 and 1000. These budgets are used as the initial task runtime budgets, C in

the model. The constraint solvers including CoPi use the task budgets to solve the E2E delay, loss-rate and utilization bound constraints.

Before going into the analysis of evaluation results, we explain a couple of parameters related to the E2E delay to standardize its relationship to the task budgets. These parameters are used throughout this section.

1. **Latency Budget Gap (LBG):** It is the ratio of a supplied upper bound on the E2E latency (E^{UB}) and the summation of all the task budgets in a pipeline P.

$$\text{Latency Budget Gap: } LBG = \frac{E^{UB}}{\sum_{i=1}^{N} C_i} \tag{5.16}$$

   LBG depicts how much time-unit (or gap) is there for the task periods, that is beyond the sum of task budgets, with respect to E2E delay. As noted in Equation 5.2 and 5.4, the periods contribute to the E2E delay. Therefore, LBG intuitively depicts the hardness of the E2E delay constraint.

   For different pipelines, the task budgets are different. Therefore, their end-to-end delay upper bounds (E^{UB}) are also expected to be different. This makes it difficult to compare how a solver performs for different pipelines with different task budgets for the end-to-end delay constraint. LBG standardizes the relationship between the task budgets and E^{UB}. Thus, performance against randomly generated task pipelines are compared for different solvers with different values of LBGs.

   A higher LBG means that the upper bound on the E2E delay is greater, and the E2E delay constraint is more relaxed. So, finding a schedulable pipeline is more probable with a higher LBG, because of more number of possible solutions. Conversely, smaller LBG means tighter E2E delay constraint, as the gap between E^{UB} and the sum of task

budgets is small.

2. **Normalized LBG (NLBG):** It is the ratio of the LBG and the length of a pipeline. NLBG normalizes LBG with respect to the pipeline length.

$$\text{Normalized LBG: } NLBG = \frac{LBG}{N} \tag{5.17}$$

We can only compare the pipelines of same length with LBG. With NLBG, pipelines of different lengths are compared (e.g., Figure 5.9). A higher NLBG, like LBG, also increases the probability of finding a schedulable pipeline, and vice-versa.

The loss-rate is already expressed in terms of percentage, so we do not need any other standardized parameter for it.

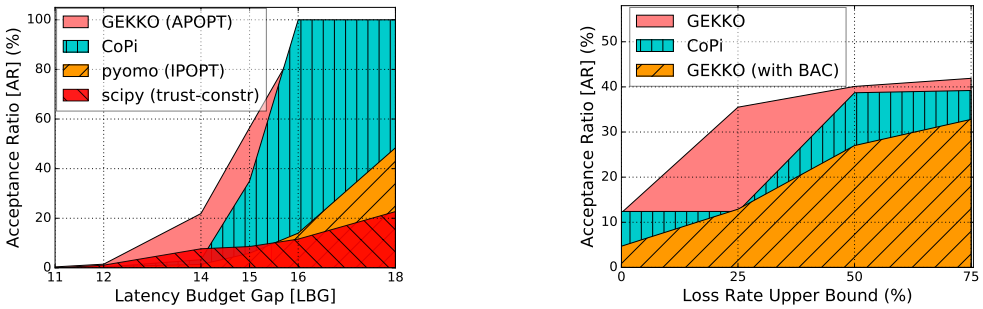
We run all the experiments with Python 3.6 on an 64-bit Linux (Ubuntu 16.04) machine featuring a Core i5-4210 processor. For every experiment, we report the average value against 1000 randomly generated task pipelines. We choose $\beta = 2$ in all cases, whereas α is iterated starting from 2 and then decreasing in steps of 0.01.

5.5.1 Uniprocessor Acceptance Ratio

5.5.1.1 Only End-to-end Delay Constraint

The first experiment compares the pipeline acceptance ratios (ARs) of open-source constraint solvers to CoPi's AR, only under the end-to-end delay constraint. Figure 5.7a shows the percentage of pipelines with 10 tasks, that are schedulable for uniprocessor RMS utilization bound against increasing LBG. LBG is varied from an acceptance ratio of 0% to 100% for most solvers.

The GEKKO Optimization Suite [BHMH18] with its APOPT solver [apo22] dominates other modeling packages and CoPi. Although pyomo [HWW11, BHH+21] uses



(a) Solvers under E2E delay constraint **(b)** Solvers under both pipeline constraints (LBG = 15)

Figure 5.7: Uniprocessor Pipeline Acceptance Ratios (N = 10)

a well-known IPOPT method [WB06] for MINLP problems, its implementation of the *Disjunction* properties are still in development [Pyo21]. Hence, its acceptance ratio is worse, but the performance improves with higher *LBG* values. `scipy` is a more generalized mathematical and optimization python package, from which we use the *trust-constr* [BHN99, LNP98] constraint minimization approach. As it is a local minimizer, its solution is dependent on the initial suggested value. Because of limitation in its current implementation [sci21], it performs poorly for the same initial value that is provided to GEKKO and pyomo.

CoPi's AR is worse than GEKKO's, but better than other solvers. It reaches 100% AR at LBG = 16 when GEKKO also reaches near 100% AR. Its performance is similar for smaller (≤ 12) and larger (≥ 15) LBG values. This experiment shows that CoPi performs comparably with respect to the other MINLP solvers. The performance of the MINLP solvers may well be further improved with more iterations, commercial solvers and perhaps better modeling techniques, but MINLP solvers are not suitable for runtime scheduling because of their slow execution time performances. We show this in Section 5.5.2.

5.5.1.2 Both E2E Delay and Loss-rate Constraints

We now focus on the performance of GEKKO and CoPi, as other solvers do not perform as good as these two. For next set of experiments, we apply both the pipeline constraints

- E2E delay and loss-rate. Apart from running GEKKO with the existing constraints, we also run it with the Budget Adjustment Constraint (BAC) (described in Section 5.2.4.1) to investigate whether it is able to utilize a rate-matching domain knowledge. AR is plotted in Figure 5.7b for increasing loss-rate upper bounds against a fixed LBG = 15 and N = 10.

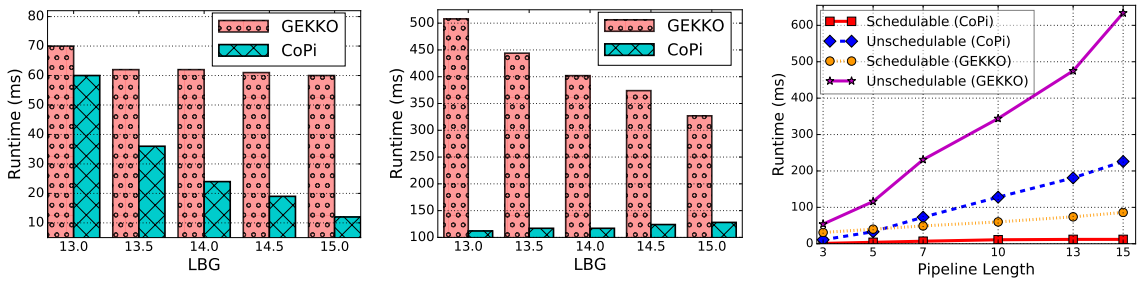
The graph shows that CoPi performs comparably to GEKKO. As the upper bound on the loss-rate increases, AR also improves for both the solvers. GEKKO (with BAC) performs worst because BAC adds more variables to the solver. It exhausts the number of iterations with more variables. GEKKO performs much better without BAC and is not able to exploit a rate-matching heuristic like CoPi does.

5.5.2 Solver Runtime Overhead

In the above experiments, we show that CoPi and GEKKO perform similarly against the E2E delay and loss-rate and RMS utilization bound constraints. In next experiments, we investigate the execution times of both the solvers to examine their capabilities in runtime scheduling of task pipelines.

Figure 5.8a plots the runtime of CoPi and GEKKO for schedulable pipelines against an increasing LBG for pipeline length of 10. We only plot for LBG = [13, 15] because the acceptance ratio is significant in this range of LBG for both GEKKO and CoPi. It shows that the runtime is comparable for both the solvers for a stricter LBG. As LBG increases, the E2E delay constraint is more relaxed. In these cases, CoPi is able to find a schedulable pipeline more quickly than GEKKO is capable of doing. For an LBG = 15, GEKKO takes on average almost 5 times more than CoPi.

Figure 5.8b plots the execution times for unschedulable pipelines. It shows that even for a unschedulable pipeline, GEKKO keeps searching for feasible task parameters longer



(a) Runtime for schedulable pipelines (N = 10)

(b) Runtime for unschedulable pipelines (N = 10)

(c) Runtime vs. pipeline length ($NLBG$ = 1.5)

Figure 5.8: Runtime Overhead under end-to-end delay constraint

time before retiring, whereas CoPi responds *at least* 2 times faster.

Figure 5.8c shows runtime overhead of CoPi and GEKKO with respect to increasing pipeline length. As explained in Section 5.3.4 for CoPi, its runtime increases with increasing length of pipeline (against a fixed $NLBG$ = 1.5). The relationship between runtime and pipeline length is nearly linear. GEKKO's runtime for both schedulable and unschedulable pipelines are greater than CoPi's runtime for all pipeline lengths. More importantly, GEKKO's runtimes grow faster with pipeline length than CoPi's runtimes do.

Table 5.2: Runtime overhead for both pipeline constraints (N = 10)

Constraints	**Schedulable (ms)**		**Unschedulable (ms)**	
	GEKKO	CoPi	GEKKO	CoPi
E2E Delay (LBG = 15)	61	**12**	327	**128**
E2E Delay + Loss-rate ($L^{UB} \leq 0\%$)	205	**104**	966	**130**
E2E Delay + Loss-rate ($L^{UB} \leq 25\%$)	191	**105**	1437	**132**
E2E Delay + Loss-rate ($L^{UB} \leq 50\%$)	187	**102**	1801	**135**
E2E Delay + Loss-rate ($L^{UB} \leq 75\%$)	200	**107**	2237	**131**

In the next experiment, we evaluate the effect of both the constraints on runtime overhead for GEKKO and CoPi. Table 5.2 summarizes the result of the experiment with a pipeline length of 10 and two constraints (LBG = 15 fixed , L^{UB} varied). CoPi always has lower runtime overhead compared to GEKKO. The performance of GEKKO degrades significantly after adding both the constraints for failed pipelines from 327ms to as much

as 2237ms, whereas CoPi takes similar time to fail to schedule a pipeline. The reason is that CoPi checks the loss-rate constraint every time where the E2E delay constraint and utilization bound are checked. Hence, the failing time does not increase. However, the runtimes for schedulable pipelines does increase for CoPi after adding the loss-rate constraint on top of E2E delay, because it discards all the results where loss-rate is greater than the given upper bound.

5.5.3 Performance Insights of CoPi

This section delves into more details about CoPi's performance and its optimizations.

5.5.3.1 Pipeline Length and $NLBG$

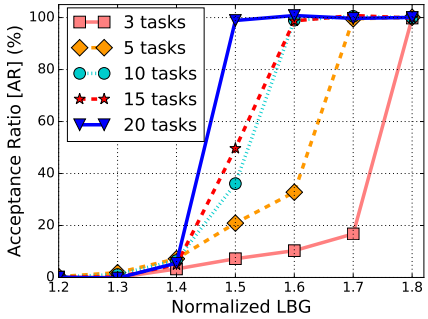


Figure 5.9: NLBG vs. pipeline lengths

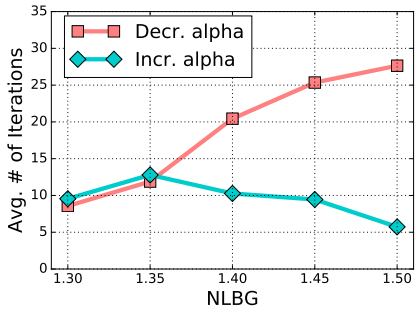


Figure 5.10: Number of iterations in CoPi against two α iteration strategies

Figure 5.9 shows CoPi's AR with increasing NLBG for different pipeline lengths only under the E2E delay constraint. After certain NLBGs, the acceptance ratio jumps to 100% for all pipeline lengths because: 1) assigning fixed and same period of $\frac{E^{UB}}{N+1}$ meets the E2E delay constraint requirement, 2) as individual task utilizations reduce with greater periods, the utilization bound constraint is also satisfied.

The NLBG value, after which most pipelines are schedulable, is dependent on the pipeline length. For example, Figure 5.9 shows that all pipelines are schedulable for

pipeline lengths of 20 for $NLBG \geq 1.5$. However, for pipeline lengths of 5, all pipelines are schedulable for $NLBG \geq 1.7$. This threshold NLBG is higher for shorter pipelines because there are fewer available pipelined tasks to tune. CoPi gets fewer opportunities to distribute the periods from the E2E delay, before the utilization bound constraint is violated.

5.5.3.2 Effectiveness of Stage 2 and 3

Table 5.3: Performance Insights CoPi

(a) Stage 2 and 3 Acceptation Ratio under *strict* NLBG

Pipeline Length	NLBG	AR (%)
3	1.3	0.8
	1.4	2.2
	1.5	7.4
	1.6	11.1
5	1.3	2.1
	1.4	6.5
	1.5	22
	1.6	31.8
10	1.3	2.5
	1.4	6.7
	1.5	7.2
	1.6	35.5
15	1.3	1.1
	1.4	1.7
	1.5	4.8
	1.6	49

(b) Acceptance Ratio Improvement with Larger Utilization Bound for Harmonic Tasksets ($N = 10$)

NLBG	RMS Utilization Bound	AR (%)
1.1	$\leq n \times (2^{\frac{1}{n}} - 1)$	0
	≤ 1	20.4
1.2	$\leq n \times (2^{\frac{1}{n}} - 1)$	0.3
	≤ 1	67.6
1.4	$\leq n \times (2^{\frac{1}{n}} - 1)$	3.3
	≤ 1	98.7
1.5	$\leq n \times (2^{\frac{1}{n}} - 1)$	35
	≤ 1	100
1.6	$\leq n \times (2^{\frac{1}{n}} - 1)$	100
	≤ 1	100

With smaller and stricter NLBG, it is harder to find a schedulable pipeline. Table 5.3a shows the AR by CoPi's Stage 2 and 3 with varying pipeline length and strict NLBG values to show its effectiveness of those two stages. Together with Figure 5.9, it demonstrates that CoPi's Stage 2 and 3 could find more feasible pipelines for stricter NLBGs for any pipeline length. For example, CoPi's Stage 2–3 AR is 31.8% for $NLBG = 1.6$ and $N = 5$, whereas CoPi's overall AR is also 31.8% for the same pipeline length and NLBG

(see Figure 5.9). It shows that the Stage 2 and 3 contributed to all the pipeline acceptances for $NLBG = 1.6$ and $N = 5$. Moreover, CoPi's optimizations are more useful for longer pipelines because the Stage 2 and 3 are able to tune more number of tasks in a pipeline to satisfy the constraints. It can be seen that Stage 2 and 3 schedule as many as 49% pipelines ($N = 15, NLBG = 1.6$) in Table 5.3a.

5.5.3.3 Utilization Bound

CoPi checks the Liu-Layland RMS utilization bound to determine whether a pipeline is schedulable in a uniprocessor. However, RMS utilization could be relaxed to 1 if all the tasks are harmonic [KM91]. Exploiting this RMS scheduling property, Table 5.3b shows that CoPi is able to schedule more tasks, even with very strict NLBGs.

5.5.3.4 Other Variations

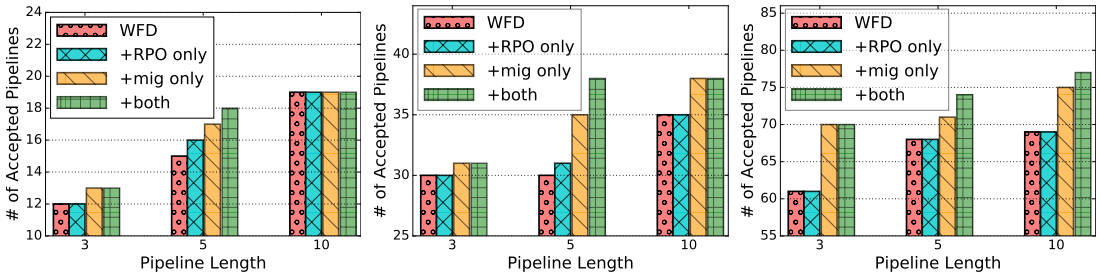
In a set of experiments, we compare two strategies of iterating over α within a fixed range: incrementing and decrementing. We investigate which one reduces the number of CoPi optimization loop iterations of Stage 2 and 3 combined. For α's increment, we start from the value derived by Equation 5.14 and increase until α = 2 to limit the number of iterations. For decrement, we start from a higher α value (2 in the experiment) and decrease until 1.01, or the point when scaling up periods by α does not meet the utilization bound test.

Figure 5.10 shows the number of iterations for schedulable pipelines of length 10. As $NLBG$ increases from 1.3 to 1.5, incrementing α takes smaller numbers of iterations as we find feasible period assignments more quickly from a higher starting value of α. The range of searching a desired α decreases. Decrementing α is thus not preferable. Both of these techniques have similar acceptance ratio.

5.5.4 Multiprocessor Performance

In this section, we investigate the performance of our multiprocessor scheduling algorithm coupled with CoPi. We measure the number of pipeline acceptances for dynamically appearing pipelines in a simulated environment that models a runtime scheduling scenario.

We experiment with 2, 4 and 8 processors. We feed 50, 100, and 200 pipelines respectively to the 2, 4, and 8 processors. For each of these experiments, pipelines are fed one after another to simulate dynamically appearing pipelines. After every 5, 10 and 20 pipelines respectively for 2, 4 and 8 processors, all the pipelines are unmapped from the processors to simulate ephemeral pipelines. We also vary the pipeline length.



(a) 2 Processors **(b)** 4 Processors **(c)** 8 Processors

Figure 5.11: Number of Accepted (Schedulable) Simulated Pipelines in Multiprocessors

Figure 5.11 shows the number of schedulable pipelines for different pipeline lengths and number of processors. Here, WFD stands for the worst-fit decreasing heuristic, *mig* is runtime task migration (described in Section 5.4.1), RPO is runtime pipeline optimization (described in Section 5.4.2). We show the performance when WFD is solely used without migration and RPO, WFD with individually RPO (+*RPO only*) and migration (+*mig only*), and with both of them together (+*both*). On a high level, the experiment reveals that proposed runtime optimizations accommodate more dynamically appearing pipelines.

Individually WFD with only RPO has limited impact in accepting more pipelines in a number of cases ((N=3, 10 M=2), (N=3, 10, M=4), (N=all, M=8), where M is num-

ber of processors). Because of the limitations of the WFD heurisic, RPO alone cannot help significantly in accommodating more pipelines, unless already admitted tasks create utilization holes for new pipelines after optimization. Nevertheless, applying both run-time strategies together (+*both*) results into more pipeline admissions in all processors and pipeline lengths. After task migrations create bigger available utilization holes in processors, RPO helps in accommodating more pipelines.

In Figure 5.11a, both migration and RPO are not able to accommodate more pipelines for N = 10 in 2 processors. As the pipeline length is longer, the algorithm cannot accommodate all the tasks of a single pipeline in just 2 processors. So the number of pipeline admissions does not improve.

5.5.4.1 Processor Utilization

Table 5.4 tabulates the normalized (per processor) utilization at the end of the experiment for varying pipeline lengths. It shows that RPO indeed decreases the processor utilization on average. But it does not adjust the *available utilization holes* in processors for new tasks. Adding migration with RPO improves the per processor utilization in addition to admitting more pipelines.

The RMS utilization bound is 69%, when the number of tasks tends to infinity. Rest

Table 5.4: Multiprocessor Utilization

Pipeline Length	Strategy	Normalized Utilization Per Processor (%)
3	WFD	51.2
	+ RPO only	50.5
	+ migration only	54.8
	+ both	54.2
5	WFD	54.9
	+ RPO only	55
	+ migration only	60.8
	+ both	62.46
10	WFD	61.6
	+ RPO only	60.8
	+ migration only	65
	+ both	64.4

of the CPU is usually given to lower-priority background tasks. So our multiprocessor algorithm actually keeps the processor utilization to a respectably high level, as displayed in Table 5.4.

5.5.4.2 Task Migrations

Table 5.5 shows the number of average migrations which resulted in successful scheduling of new pipelines. Number of migrations increases with more processors, as the algorithm limits the number of migrations per new candidate pipeline by the number of processors. Overall, this experiment shows that only a few migrations are needed to accommodate new pipelines in the system, and the average numbers of migrations are much smaller than their limits.

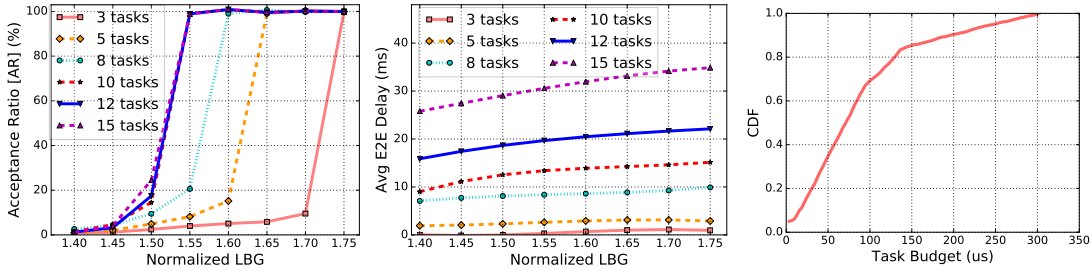
Table 5.5: Migrations in Multiprocessor

Processors	Strategies	Average Migrations
2	WFD + migration only	0.1
	WFD + migration + RPO	0.13
4	WFD + migration only	0.67
	WFD + migration + RPO	0.67
8	WFD + migration only	2.7
	WFD + migration + RPO	3.17

5.5.5 Case Study with an Industry Benchmark

We have tested CoPi with a benchmark from the WATERS 2015 workshop paper [KZH15], provided by Bosch, for uniprocessor scheduling. We calculate the worst-case execution time of a task by multiplying a random average case execution time (ACET) with a random WCET factor. The random ACET and WCET factor are chosen from the task distribution provided in Table III of the paper. We consider pipeline lengths from 3 to 15 because each runnable from the dataset might be re-implemented as a pipelined task.

Figure 5.12 summarizes the result of this experiment. Figure 5.12a shows the accep-



(a) Acceptance Ratio for Different Pipeline Lengths

(b) E2E Delay for Schedulable Pipelines

(c) CDF of WCETs

Figure 5.12: Experiments with Dataset from Bosch [KZH15]

tance ratio among 1000 randomly generated tasksets for each case. The graph is similar to Figure 5.9, but focuses on NLBG between 1.4 to 1.7 for more fine-grained data points. Figure 5.12b captures the average E2E delay for schedulable pipelines in `ms`, where each runnable's WCET is in μs granularity. Figure 5.12c shows a CDF of the worst-case execution times for 3 tasks pipelines. Other pipeline lengths have similar CDF. In Figure 5.12b, the E2E delays for smaller pipelines are under 10`ms` which is usually expected in automotive industries. The E2E delays for longer pipelines go up to 30 ms depending on the NLBG. As longer pipelines might be utilized for comparatively lower frequency workloads, slightly longer E2E delays are tolerable.

5.5.6 Case Study in DriveOS

As a real-world case study, we generate the task budgets and periods of a pipeline of 5 tasks with CoPi and create a model with those parameters in ModelMap. The model binds 5 tasks to 5 real-time periodic threads in the DriveOS' Quest RTOS domain which schedules the tasks with RMS algorithm. The task communicates with each other via `asyncRead` and `asyncWrite` Simulink blocks. Table 5.6 summarizes the result.

The end-to-end delays in Table 5.6a are computed for different NLBGs. The task budgets and periods are adjusted by CoPi based on the NLBG values. These NLBGs are higher than the ones in Section 5.5.3 because the other tasks including the I/O threads

Table 5.6: Experimental Results in Quest RTOS domain of DriveOS

(a) End-to-end Delay (Loss-rate = 0%)

NLBG	CoPi E2E Delay Upper Bound	Maximum E2E Delay in DriveOS
2.3	4815	2796
2.4	4864	2861
2.5	4857	2837
2.6	5423	3023
2.7	5635	3174

(b) End-to-end Loss-rate

CoPi Loss-rate Upper Bound	Loss-rate in DriveOS
0	0
25	9
50	28
75	47

reduce the available CPU utilization. The utilization values of the pipelines are in the range of 41–44%. The second column shows the theoretical E2E delay of a pipeline using Dürr *et al.*'s formula [DBCC19] after CoPi has found a schedulable set of task budgets and periods. The third column is the observed maximum E2E delay of a pipeline in DriveOS. This shows that the worst-case E2E delays in DriveOS are within their theoretical upper bounds for a set of task budgets and periods derived by CoPi. We also repeat the same experiments for different expected loss-rates of 0%, 25%, 50% and 75%. Table 5.6b shows that the observed end-to-end loss-rates in DriveOS are under their theoretical upper bounds as derived by CoPi.

Chapter 6

Conclusion

This thesis presents a vehicle operating system framework which consists of multiple OS domains on a centralized hardware platform for mixed-criticality automotive applications. As part of the framework, we have introduced the DriveOS integrated VMS which is being used for a production-grade electric car. DriveOS uses a real-time separation kernel to host an in-house RTOS, Quest, and Yocto Linux as guest OS domains. As proof of concept, a fast and predictable CAN gateway and a longitudinal controller are implemented as real-time services. These services support three vehicle applications in Linux, namely IC, IVI and OpenPilot ADAS. A secure, predictable and low-overhead `shmcomm` module facilitates the shared memory communication between real-time services in Quest and applications in Linux. DriveOS is tested using timing-based metrics against a standalone Linux that is currently found in many infotainment systems in the automotive industry. Experiments show that the maximum E2E delay for a USB-CAN-dependent controller loop is 12 times more in Linux than in DriveOS. DriveOS also achieves 24% more throughput for a vehicles CAN messages than a standalone Linux.

We also present ModelMap for model-based multi-domain application development and deployment in DriveOS. ModelMap consists of Simulink blocks for binding real-time threads of control, inter-task communication and CAN I/O. It supports the generation of nested binary executables to encapsulate and execute DriveOS applications. Experiments

show that custom and real-world DriveOS Simulink models, designed using ModelMap, have predictable E2E delays, in keeping with the requirements of a high performance electric vehicle.

Furthermore, this thesis explores the real-time task pipeline model to guarantee end-to-end properties of connected ECU software tasks. To this end, we present a non-linear optimization problem to find suitable task budgets and periods under a pipeline's end-to-end constraints and scheduling utilization bound. We propose CoPi, a heuristic constraint solver algorithm, which tunes task periods and budgets to minimize a pipeline's end-to-end delay and loss-rate. It essentially converts the pipelined real-time tasks to independent periodic tasks. We explain CoPi with examples and analyze with simulated experiments. Experiments show that CoPi performs favorably in terms of task pipeline acceptance ratio, compared to open-source MINLP solvers like GEKKO [BHMH18]. CoPi has an order of magnitude better runtime than GEKKO. CoPi is better suited for OS-level scheduling than an MINLP solver. We also demonstrate the benefits of CoPi in multiprocessor scheduling for dynamically appearing pipelines with fewer task migrations and more pipeline admissions with simulated tasksets. Moreover, we show that CoPi's task budgets and periods for a pipeline of periodic tasks meet the theoretical E2E properties of a pipeline when it is deployed in DriveOS.

6.1 Future Work

Plans are underway to migrate more ECU functions to DriveOS with the ModelMap framework, to support real-time torque vectoring and battery management. Figure 6.1 shows an overview of the current and possible future components in DriveOS. The userspace threads and processes are inter-connected via shmcomm channels. The CAN gateway in Quest could be connected to domain-specific filter threads in different other DriveOS

domains. A domain filter will forward CAN messages to other applications based on a security policy. In addition, real-time (RT) services like HVAC could be securely exposed to domain-specific distributor threads which would mediate access to these critical services from other applications in a domain. Moreover, services within a single domain could communicate with each other via shmcomm channels without a cross-domain connection. For example, a Linux IVI application may need to exchange map updates with an ADAS service in Linux. Similarly, a real-time power management module in Quest may turn off other vehicle devices via the CAN gateway. The shmcomm channels could be leveraged to construct a virtual vehicle function network.

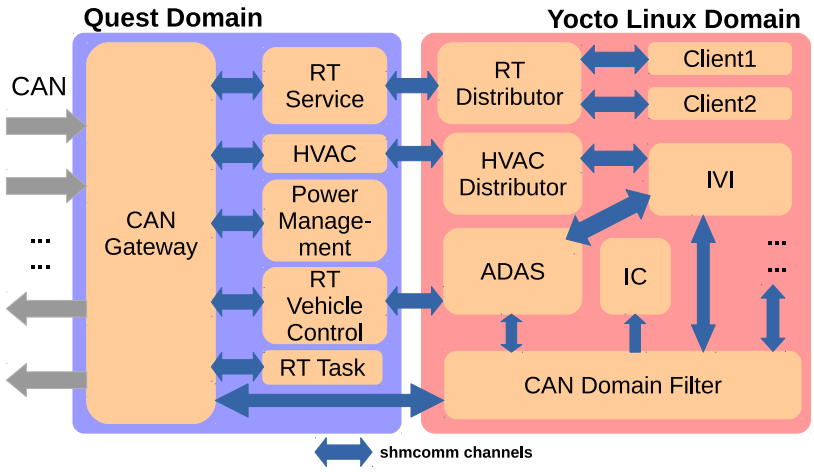


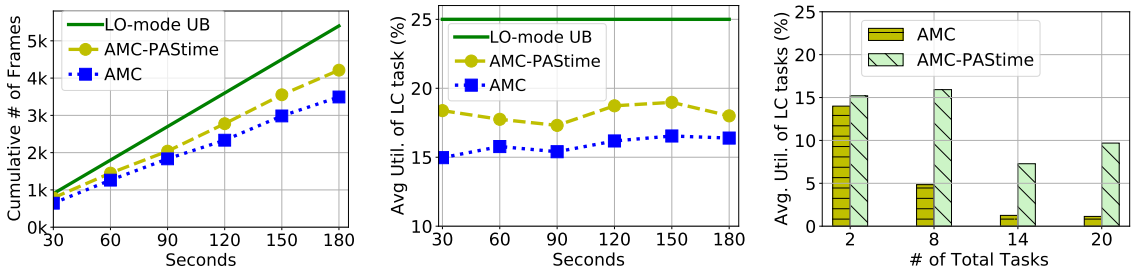
Figure 6.1: More Functions in DriveOS

Future work will also use Quest's real-time USB framework to integrate camera devices as part of an enriched autonomous vehicle framework. Additional real-time virtual device interfaces will be provided via shmcomm to allow Linux and Android to implement new ADAS services. The shmcomm module could be interfaced with existing application frameworks such as ROS [QCG+09] for publisher-subscriber communication. ROS applications in Linux domain can access real-time I/O in Quest via the shmcomm interfaces.

DriveOS currently isolates critical and non-critical tasks into separate OS domains using a partitioning hypervisor [WLMD16]. Other system design techniques could be ex-

plored for isolated application domains [Sin18]. As more functions are consolidated in DriveOS, a single domain may itself include a mixed-criticality taskset [Ves07]. For example, a Linux domain in DriveOS can include a video player as part of the IVI functions and an object detector as part of an ADAS service. The video player would then be considered low-criticality, and the object detector would be high-criticality task in Linux. In such a scenario, a mixed-criticality scheduling algorithm is necessary to ensure that the low-criticality tasks have enough CPU runtime for its minimum service-level guarantees while a high-criticality task meets its deadlines. Our progress-aware scheduling algorithm, called PAStime [SWG20], improves the Adaptive Mixed-criticality [BBD11] scheduling to attain better Quality-of-service (QoS) in low-criticality tasks. It inserts checkpoints in a high-criticality object detector application at compile-time. At runtime, it gives more CPU time to the low-criticality tasks if the high-criticality tasks progresses as expected at the inserted checkpoints. Thus, the low-criticality gets more time to run and achieves better QoS. PAStime is integrated into a uniprocessor LITMUS^{RT} [CLB^{+}06], a variant of Linux. Figure 6.2 shows the improved frame-rate and CPU utilization of a low-criticality MPEG video decoder, while a co-running high-criticality object detector application meets all its deadlines. A LITMUS^{RT} sandbox in DriveOS with PAStime runtime policy could be integrated to enable intra-domain mixed-criticality scheduling. More such cross-domain and intra-domain real-time and mixed-criticality task scheduling policies [Sin18] are possible directions for future work.

The end-to-end scheduling of real-time task pipelines has applicability in many cyber-physical systems, even beyond the automotive domain. Therefore, pipeline scheduling needs to be adopted in application frameworks. ModelMap already takes advantage of CoPi to provide predictable pipeline properties in the Quest RTOS domain of DriveOS. Further, the nested binary interface could be extended to integrate CoPi across different



(a) Cumulative # of Frames **(b)** Average Utilization **(c)** Varying # of Tasks

Figure 6.2: Video Decoder Performance

DriveOS domains. In the future, application frameworks such as ROS [QCG+09] and micro-ROS [mic22] can also use CoPi and its multiprocessor scheduling to parameterize and schedule a predictable pipeline of periodic tasks in RTOSes like FreeRTOS [Fre22b].

Furthermore, a model-driven pipeline programming language will be investigated on top of the ModelMap framework. Model-based design languages like Simulink provide a rich and intuitive interface to build modular and composable software task pipelines [GCW18, GSW20]. Data communication and its constraints could be treated as first-class properties in the language. End-to-end guarantees of a task pipeline in Simulink could be verified in simulation under a provided task scheduling algorithm. Since ModelMap demonstrates predictable end-to-end properties of the Simulink models in DriveOS, the language interface would offer a verifiable guarantee of the end-to-end properties both at the design time and runtime.

Bibliography

[AGK+02] Luca Abeni, Ashvin Goel, Charles Krasic, Jim Snow, and Jonathan Walpole. A Measurement-based Analysis of the Real-time Performance of Linux. In *Proceedings. Eighth IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 133–142, 2002.

[Ahs13] Kamrul Ahsan. Trend Analysis of Car Recalls: Evidence from the US Market. *International Journal of Managing Value and Supply Chains*, 4(4):1, 2013.

[Ape22] Apex.ai. Customer Success Story: Toyota's Woven Planet, 2022. `https://www.apex.ai/toyota-woven-planet`.

[Apo21] ApolloAuto. Apollo Automotive. https://github.com/ApolloAuto/apollo, 2021.

[apo22] APOPT - Advanced Process OPTimizer. `https://apopt.com/`, 2022. Accessed: 2022-May-27.

[Arg22] Argo.ai. Argo is reimagining the human journey, 2022. `https://www.argo.ai/`.

[AUT17] AUTOSAR. Specification of Timing Extensions. page 219, 2017.

[AUT22] AUTOSAR. AUTomotive Open System ARchitecture, 2022. `http://www.autosar.org`.

[Avi85] Algirdas Avizienis. The N-version Approach to Fault-tolerant Software. *Software Engineering, IEEE Transactions on*, (12):1491–1501, 1985.

[B+08] Richard Barry et al. FreeRTOS. *Internet, Oct*, 2008.

[BB05] Enrico Bini and Giorgio C Buttazzo. Measuring the Performance of Schedulability Tests. *Real-Time Systems*, 30(1-2):129–154, 2005.

[BBD11] S. K. Baruah, A. Burns, and R. I. Davis. Response-Time Analysis for Mixed Criticality Systems. In *2011 IEEE 32nd Real-Time Systems Symposium*, pages 34–43, November 2011.

[BC08] Enrico Bini and Anton Cervin. Delay-Aware Period Assignment in Control Systems. In *2008 Real-Time Systems Symposium*, pages 291–300, November 2008.

[BDDK18] Ondrej Burkacky, Johannes Deichmann, Georg Doll, and Christian Knochenhauer. Rethinking Car Software and Electronics Architecture. *McKinsey & Company*, 2018.

[BDF+03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. *ACM SIGOPS OSR*, 37(5):164–177, 2003.

[BDM+16a] Matthias Becker, Dakshina Dasari, Saad Mubeen, Moris Behnam, and Thomas Nolte. MECHAniSer - A Timing Analysis and Synthesis Tool for Multi-Rate Effect Chains with Job-Level Dependencies. In *7th International Workshop on Analysis Tools and Methodologies for Embedded and Real-Time Systems (WATERS)*, page 6, 2016.

[BDM+16b] Matthias Becker, Dakshina Dasari, Saad Mubeen, Moris Behnam, and Thomas Nolte. Synthesizing Job-Level Dependencies for Automotive Multi-Rate Effect Chains. In *2016 IEEE 22nd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 159–169. IEEE, 2016.

[BDM+17] Matthias Becker, Dakshina Dasari, Saad Mubeen, Moris Behnam, and Thomas Nolte. End-to-end timing analysis of cause-effect chains in automotive embedded systems. *Journal of Systems Architecture*, 80:104–113, 10/2017.

[BDS19] Ondrej Burkacky, Johannes Deichmann, and Jan Paul Stein. Automotive Software and Electronics 2030: Mapping the Sector's Future Landscape. *McKinsey & Company*, 2019.

[BGL+20] Hamza Bourbouh, Pierre-Loïc Garoche, Thomas Loquen, Éric Noulard, and Claire Pagetti. CoCoSim, a Code Generation Framework for Control/Command Applications: An Overview of CoCoSim for Multi-Periodic Discrete Simulink Models. In *the 10th European Congress on Embedded Real Time Software and Systems*, Toulouse, France, 2020.

[BHH+21] Michael L. Bynum, Gabriel A. Hackebeil, William E. Hart, Carl D. Laird, Bethany L. Nicholson, John D. Siirola, Jean-Paul Watson, and David L. Woodruff. *Pyomo–Optimization Modeling in Python*, volume 67. Springer Science & Business Media, third edition, 2021.

[BHMH18] Logan Beal, Daniel Hill, R Martin, and John Hedengren. GEKKO Optimization Suite. *Processes*, 6(8):106, 2018.

[BHN99] Richard H Byrd, Mary E Hribar, and Jorge Nocedal. An interior point algorithm for large-scale nonlinear programming. *SIAM Journal on Optimization*, 9(4):877–900, 1999.

[BKÁ+18] Philipp Berger, Joost-Pieter Katoen, Erika Ábrahám, Md Tawhid Bin Waez, and Thomas Rambow. Verifying Auto-generated C Code from Simulink. In *the International Symposium on Formal Methods*, pages 312–328. Springer, 2018.

[Bla21] BlackBerry. BlackBerry QNX Software Is Now Embedded In Over 195 Million Vehicles, 2021. `https://www.blackberry.com/us/en/company/newsroom/press-releases/2021/blackberry-qnx-software-is-now-embedded-in-over-195-million-vehicles`.

[BMC+17] Alessio Bucaioni, Saad Mubeen, Federico Ciccozzi, Antonio Cicchetti, and Mikael Sjödin. Technology-Preserving Transition from Single-Core to Multi-core in Modelling Vehicular Systems. In *European Conference on Modelling Foundations and Applications*, pages 285–299. Springer, 2017.

[BR13] Reto Buerki and Adrian-Ken Rueegsegger. Muen-an x86/64 separation kernel for high assurance. *University of Applied Sciences Rapperswil (HSR), Tech. Rep*, 2013.

[BS95] Thomas C Bressoud and Fred B Schneider. Hypervisor-based fault tolerance. *ACM SIGOPS Operating Systems Review*, 29(5):1–11, 1995.

[BS14] Gedare Bloom and Joel Sherrill. Scheduling and Thread Management with RTEMS. *ACM Sigbed Review*, 11(1):20–25, 2014.

[BSC+21] Luca Belluardo, Andrea Stevanato, Daniel Casini, Giorgiomaria Cicero, Alessandro Biondi, and Giorgio Buttazzo. A Multi-Domain Software Architecture for Safe and Secure Autonomous Driving. In *Proceedings of the 27th IEEE RTCSA Conference*, 2021.

[BSPL21] Victor Bandur, Gehan Selim, Vera Pantelic, and Mark Lawford. Making The Case for Centralized Automotive E/E Architectures. *IEEE Transactions on Vehicular Technology*, 70(2):1230–1245, 2021.

[CBLB19] Daniel Casini, Tobias Blaß, Ingo Lütkebohle, and Björn B. Brandenburg. Response-Time Analysis of ROS 2 Processing Chains Under Reservation-Based Scheduling. In Sophie Quinton, editor, *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*, volume 133 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 6:1–6:23. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019.

[CCM+03] Paul Caspi, Adrian Curic, Aude Maignan, Christos Sofronis, Stavros Tripakis, and Peter Niebert. From Simulink to SCADE/Lustre to TTA: A Layered Approach for Distributed Embedded Applications. In *LCTES*, page 10, 2003.

[Cha21] Robert N Charette. How Software is Eating The Car. *IEEE Spectrum, June*, 2021.

[Cin22] Cincoze. DX1100. https://www.cincoze.com/, 2022.

[Cis17] Cisco. Time-Sensitive Networking: A Technical Introduction, 2017. White paper, www.cisco.com.

[CJK16] Hyun-Jun Cha, Woo-Hyuk Jeong, and Jong-Chan Kim. Control-Scheduling Codesign Exploiting Trade-Off between Task Periods and Deadlines. *Mobile Information Systems*, 2016:1–11, 2016.

[CKK+18] Jin-Kyu Choi, Kyongho Kim, Dohyun Kim, Hyunkyun Choi, and Byungtae Jang. Driver-adaptive Vehicle Interaction System for the Advanced Digital Cockpit. In *2018 20th International Conference on Advanced Communication Technology (ICACT)*, pages 307–310. IEEE, 2018.

[CKK20] Hyunjong Choi, Mohsen Karimi, and Hyoseung Kim. Chain-Based Fixed-Priority Scheduling of Loosely-Dependent Tasks. In *2020 IEEE 38th International Conference on Computer Design (ICCD)*, pages 631–639. IEEE, 10/2020.

[CLB+06] John M Calandrino, Hennadiy Leontyev, Aaron Block, UmaMaheswari C Devi, and James H Anderson. LITMUS^{RT}: A Testbed for Empirically Comparing Real-time Multiprocessor Schedulers. In *2006 27th IEEE International Real-Time Systems Symposium (RTSS'06)*, pages 111–126. IEEE, 2006.

[com20] comma.ai. OpenPilot: An Open Source Driver Assistance System, 2020. `https://github.com/commaai/openpilot`.

[COV22] COVESA. Connected Vehicle Systems Alliance (COVESA). https://www.covesa.global/, 2022.

[Cru22] Cruise. A self-driving car service, 2022. `https://www.getcruise.com/`.

[CRZC16] Wanli Chang, Debayan Roy, Licong Zhang, and Samarjit Chakraborty. Model-Based Design of Resource-Efficient Automotive Control Software. In *the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8, Nov 2016.

[CS16] Kyong-Tak Cho and Kang G Shin. Error Handling of In-vehicle Networks Makes Them Vulnerable. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1044–1055, 2016.

[CWAF14] Arquimedes Canedo, Jiang Wan, and Mohammad Abdullah Al Faruque. Functional Modeling Compiler for System-level Design of Automotive Cyber-Physical Systems. In *2014 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 39–46, 2014.

[CWE18] Z. Cheng, R. West, and C. Einstein. End-to-End Analysis and Design of a Drone Flight Controller. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2404 – 2415, Nov 2018.

[DAN+13] Dakshina Dasari, Benny Akesson, Vincent Nelis, Muhammad Ali Awan, and Stefan M Petters. Identifying the Sources of Unpredictability in COTS-based Multicore Systems. In *2013 8th IEEE international symposium on industrial embedded systems (SIES)*, pages 39–48. IEEE, 2013.

[DBCC19] 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, 2019-10-19.

[dBR06] D. de Niz, G. Bhatia, and R. Rajkumar. Model-Based Development of Embedded Systems: The SysWeaver Approach. In *the 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 231–242, April 2006.

[Del19] Deloitte. Semiconductors – the Next Wave. Opportunities and winning strategies for semiconductor companies, 2019. `https://www2.deloitte.com/content/dam/Deloitte/tw/Documents/technology-media-telecommunications/tw-semiconductor-report-EN.pdf`.

[Del20] Deloitte. Software is transforming the automotive world, 2020.

[Den17] Steve Dent. Toyota's latest infotainment system is powered by Linux. `https://www.engadget.com/2017-05-31-toyota-automotive-grade-linux-camry.html`, 2017.

[DFS98] Premkumar Devanbu, PW-L Fong, and Stuart G Stubblebine. Techniques for Trusted Software Engineering. In *Proceedings of the 20th International Conference on Software Engineering*, pages 126–135. IEEE, 1998.

[DLW11] Matthew Danish, Ye Li, and Richard West. Virtual-CPU Scheduling in the Quest Operating System. In *the 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 169–179. IEEE, 2011.

[DM03] Lorenzo Dozio and Paolo Mantegazza. Linux Real Time Application Interface (RTAI) in Low Cost High Performance Motion Control. *Motion Control*, 2003(1):1–15, 2003.

[dNAK+17] Dionisio de Niz, Bjorn Andersson, Hyoseung Kim, Mark Klein, Linh Thi Xuan Phan, and Raj Rajkumar. Mixed-criticality Processing Pipelines. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pages 1372–1375. IEEE, 2017.

[DNSV10] Marco Di Natale and Alberto Luigi Sangiovanni-Vincentelli. Moving from Federated to Integrated Architectures in Automotive: The Role of Standards, Methods and Tools. *Proceedings of the IEEE*, 98(4):603–620, 2010.

[DRC+17] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. CARLA: An open urban driving simulator. In *Proceedings of the 1st Annual Conference on Robot Learning*, pages 1–16, 2017.

[DZDN+07] Abhijit Davare, Qi Zhu, Marco Di Natale, Claudio Pinello, Sri Kanajan, and Alberto Sangiovanni-Vincentelli. Period Optimization for Hard Real-time Distributed Automotive Systems. In *2007 44th ACM/IEEE Design Automation Conference*, pages 278–283, June 2007.

[eCo22] eCos. Embedded Configurable Operating System, 2022. `http://ecos.sourceware.org/`.

[EJ09] Christof Ebert and Capers Jones. Embedded Software: Facts, Figures, and Future. *Computer*, 42(4):42–52, April 2009.

[EKI17] Jesse Edwards, Ameer Kashani, and Gopalakrishnan Iyer. Evaluation of Software Vulnerabilities in Vehicle Electronic Control Units. In *2017 IEEE Cybersecurity Development (SecDev)*, pages 83–84. IEEE, 2017.

[ER08] Friedrich Eisenbrand and Thomas Rothvoß. Static-priority Real-time Scheduling: Response time computation is NP-hard. In *2008 Real-Time Systems Symposium*, pages 397–406. IEEE, 2008.

[ERSS+20] Zeinab El-Rewini, Karthikeyan Sadatsharan, Niroop Sugunaraj, Daisy Flora Selvaraj, Siby Jose Plathottam, and Prakash Ranganathan. Cybersecurity Attacks in Vehicular Sensors. *IEEE Sensors Journal*, 20(22):13752–13767, 2020.

[Fle01] William J. Fleming. Overview of Automotive Sensors. *IEEE Sensors Journal*, 1(4), December 2001.

[Flo21] Florian Götz. The Data Deluge: What do we do with the data generated by AVs?, 2021. `https://blogs.sw.siemens.com/polarion/the-data-deluge-what-do-we-do-with-the-data-generated-by-avs/`.

[For21] Ford. SYNC. `https://www.ford.com/technology/sync/`, 2021.

[FPZ15] Bernd Finkbeiner, Geguang Pu, and Lijun Zhang, editors. *Formal Verification of Simulink/Stateflow Diagrams*, volume 9364 of *Lecture Notes in Computer Science*. Springer International Publishing, Cham, 2015.

[FR97] G. Fohler and K. Ramamritham. Static scheduling of pipelined periodic tasks in distributed real-time systems. In *Proceedings Ninth Euromicro Workshop on Real Time Systems*, pages 128–135, June 1997.

[Fre22a] FreeRTOS. SAFERTOS for Automotive. `https://www.freertos.org/FreeRTOS-Plus/Safety_Critical_Certified/SafeRTOS.html`, 2022.

[Fre22b] FreeRTOS. Symmetric Multiprocessing (SMP) with FreeRTOS. `https://freertos.org/symmetric-multiprocessing-introduction.html`, 2022.

[Fri06] J. Friedman. MATLAB/Simulink for Automotive Systems Design. In *the Design Automation Test in Europe Conference*, volume 1, pages 1–2, March 2006.

[FRNJ08] Nico Feiertag, K. Richter, J. Nordlander, and J. Jönsson. A Compositional Framework for End-to-End Path Delay Calculation of Automotive Systems under Different Path Semantics. In *Proc. IEEE RTSS Workshop*, 2008.

[FUVC13] Borja Fons-Albert, Hector Usach-Molina, Joan Vila-Carbo, and Alfons Crespo-Lorente. Development of Integrated Modular Avionics Application Based on Simulink and XtratuM. In *Data Systems In Aerospace*, volume 720, page 15, August 2013.

[GCU+21] M. Günzel, K.-H. Chen, N. Ueter, G. v. d. Brüggen, M. Dürr, and J.-J. Chen. Timing Analysis of Asynchronized Distributed Cause-Effect Chains. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2021.

[GCW18] Ahmad Golchin, Zhuoqun Cheng, and Richard West. Tuned Pipes: End-to-end Throughput and Delay Guarantees for USB Devices. In *the IEEE Real-Time Systems Symposium (RTSS)*, pages 196–207, 2018.

[Gee22] GeekWire. Backed by Amazon and newly public, Aurora accelerates self-driving tech, starting with trucks, May 2022. `https://www.geekwire.com/2022/backed-by-amazon-and-newly-public-aurora-accelerates-self-driving-tech-starting-with-trucks/`.

[Gen22] General Motors. General Motors and Red Hat Collaborate to Trailblaze the Future of Software-Defined Vehicles, 2022. `https:`

//media.gm.com/media/us/en/gm/news.detail.html/content/Pages/news/us/en/2022/may/0510-redhat.html.

[Gho22] Ghost Locomotion. Attention-Free Self-Driving, 2022. https://www.driveghost.com/.

[GNU21] GNU Project. GNU Linear Programming Kit. https://www.gnu.org/software/glpk/, 2021.

[Goo21a] Google. Android Automotive OS, 2021. https://source.android.com/devices/automotive.

[Goo21b] Google. Google OR-Tools. https://developers.google.com/optimization, 2021.

[GSC+16] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Newman Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 653–669, 2016.

[GSS95] R. Gerber, Seongsoo Hong, and M. Saksena. Guaranteeing Real-time Requirements with Resource-based Calibration of Periodic Processes. *IEEE Transactions on Software Engineering*, 21(7):579–592, July 1995.

[GSW20] Ahmad Golchin, Soham Sinha, and Richard West. Boomerang: Real-Time I/O Meets Legacy Systems. In *IEEE RTAS*, pages 390–402, 2020.

[HBGR17] Philipp Hock, Sebastian Benedikter, Jan Gugenheimer, and Enrico Rukzio. CarVR: Enabling In-Car Virtual Reality Entertainment. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, pages 4034–4044, 2017.

[HCKK20] Seonyeong Heo, Sungjun Cho, Youngsok Kim, and Hanjun Kim. Real-time object detection system with multi-path neural networks. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 174–187. IEEE, 2020.

[HDK+17] Arne Hamann, Dakshina Dasari, Simon Kramer, Michael Pressler, and Falk Wurst. Communication Centric Design in Complex Automotive Embedded Systems. page 20, 2017.

[Hil92] Dan Hildebrand. An Architectural Overview of QNX. In *USENIX Workshop on Microkernels and Other Kernel Architectures*, pages 113–126, 1992.

[HKLW10] Raymond Hemmecke, Matthias Köppe, Jon Lee, and Robert Weismantel. Nonlinear integer programming. In *50 Years of Integer Programming 1958-2008*, pages 561–618. Springer, 2010.

[HvdVV94] JA Hoogeveen, Steef L van de Velde, and Bart Veltman. Complexity of scheduling multiprocessor tasks with prespecified processor allocations. *Discrete Applied Mathematics*, 55(3):259–272, 1994.

[HWW11] William E Hart, Jean-Paul Watson, and David L Woodruff. Pyomo: Modeling and Solving Mathematical Programs in Python. *Mathematical Programming Computation*, 3(3):219–260, 2011.

[Int20] Intel. Benefits of ECU Consolidation. 2020.

[ISO11] ISO. ISO 26262-3: Road vehicles - Functional safety - Part 3: Concept phase , 2011.

[JP86] Mathai Joseph and Paritosh Pandya. Finding response times in a real-time system. *The Computer Journal*, 29(5):390–395, 1986.

[KAGS05] H. Kopetz, A. Ademaj, P. Grillinger, and K. Steinhammer. The Time-Triggered Ethernet Design. In *Proceedings of the 8th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, 18-20 May 2005.

[KBS18] Tomasz Kloda, Antoine Bertout, and Yves Sorel. Latency analysis for Data chains of Real-time Periodic tasks. In *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 360–367. IEEE, 9/2018.

[KBS20] Tomasz Kloda, Antoine Bertout, and Yves Sorel. Latency upper bound for data chains of real-time periodic tasks. *Journal of Systems Architecture*, 109:101824, 10/2020.

[KCR+10] Karl Koscher, Alexei Czeskis, Franziska Roesner, Shwetak Patel, Tadayoshi Kohno, Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, and Hovav Shacham et al. Experimental Security Analysis of a Modern Automobile. In *2010 IEEE Symposium on Security and Privacy*, pages 447–462, May 2010.

[KKR13] Hyoseung Kim, Arvind Kandhalu, and Ragunathan Rajkumar. A Coordinated Approach for Practical OS-Level Cache Management in Multi-Core Real-Time Systems. In *2013 25th Euromicro Conference on Real-Time Systems*, pages 80–89. IEEE, 2013.

[KM91] T.-W. Kuo and A.K. Mok. Load adjustment in adaptive real-time systems. In *[1991] Proceedings Twelfth Real-Time Systems Symposium*, pages 160–170, December 1991.

[Kos10] Joseph Koshy. libelf by Example, 2010. `http://people.freebsd.org/jkoshy/download/libelf/article.html`.

[KTM+18] Shinpei Kato, Shota Tokunaga, Yuya Maruyama, Seiya Maeda, Manato Hirabayashi, Yuki Kitsukawa, Abraham Monrroy, Tomohito Ando, Yusuke Fujii, and Takuya Azumi. iAutoware on board: Enabling Autonomous Vehicles with Embedded Systems. In *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)*, pages 287–296. IEEE, 2018.

[Kva22] Kvaser. https://www.kvaser.com/product/kvaser-usbcan-pro-5xhs/, 2022.

[KW07] Robert Kaiser and Stephan Wagner. Evolution of the pikeos microkernel. In *First International Workshop on Microkernels for Embedded Systems*, volume 50, 2007.

[KXL+13] Linghe Kong, Mingyuan Xia, Xiao-Yang Liu, Min-You Wu, and Xue Liu. Data loss and reconstruction in sensor networks. In *2013 Proceedings IEEE INFOCOM*, pages 1654–1662. IEEE, 2013.

[KZH15] Simon Kramer, Dirk Ziegenbein, and Arne Hamann. Real World Automotive Benchmarks For Free. page 6, 2015.

[LA09a] Cong Liu and James H. Anderson. Supporting Sporadic Pipelined Tasks with Early-Releasing in Soft Real-Time Multiprocessor Systems. In *2009 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 284–293, August 2009.

[LA09b] Cong Liu and James H. Anderson. Supporting Pipelines in Soft Real-Time Multiprocessor Systems. In *2009 21st Euromicro Conference on Real-Time Systems*, pages 269–278, July 2009.

[Lev22] Levin, Tim. Security company says Teslas can be unlocked and driven using a simple, inexpensive hack, May 2022. `https://www.businessinsider.com/tesla-hack-unlock-iphone-key-bluetooth-security-2022-5`.

[Lin19] FOSS Linux. BMW gets closer to adopting Linux as the mainline platform. `https://bit.ly/2XKD6kk`, 2019.

[Lin22a] Linux. fexecve - Execute Program Specified via File Descriptor, 2022. `https://man7.org/linux/man-pages/man3/fexecve.3.html`.

[Lin22b] Linux. memfd_create - Create an Anonymous File, 2022. https://man7.org/linux/man-pages/man2/memfd_create.2.html.

[LL73] 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.

[LLFC11] Juri Lelli, Giuseppe Lipari, Dario Faggioli, and Tommaso Cucinotta. An Efficient and Scalable Implementation of Global EDF in Linux. In *OSPERT*, pages 6–15, 2011.

[LNP98] Marucha Lalee, Jorge Nocedal, and Todd Plantenga. On the implementation of an algorithm for large-scale equality constrained optimization. *SIAM Journal on Optimization*, 8(3):682–706, 1998.

[LRS+19] Mengqi Liu, Lionel Rieg, Zhong Shao, Ronghui Gu, David Costanzo, Jung-Eun Kim, and Man-Ki Yoon. Virtual Timeline: A Formal Abstraction for Verifying Preemptive Schedulers with Temporal Isolation. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–31, 2019.

[LSOH07] Bernhard Leiner, Martin Schlager, Roman Obermaisser, and Bernhard Huber. A Comparison of Partitioning Operating Systems for Integrated Systems. In *International Conference on Computer Safety, Reliability, and Security*, pages 342–355. Springer, 2007.

[Lu95] Hongjiu Lu. ELF: From The Programmer's Perspective, 1995.

[LV62] Robert E. Lyons and Wouter Vanderkulk. The Use of Triple-Modular Redundancy to Improve Computer Reliability. *IBM Journal of Research and Development*, 6(2):200–209, 1962.

[LWCM14] Ye Li, Richard West, Zhuoqun Cheng, and Eric Missimer. Predictable Communication and Migration in the Quest-V Separation Kernel. In *2014 IEEE Real-Time Systems Symposium*, pages 272–283. IEEE, 2014.

[LXRD19] Hao Li, Xuefei Xu, Jinkui Ren, and Yaozu Dong. ACRN: a Big Little Hypervisor for IoT Development. In *the 15th ACM SIGPLAN/SIGOPS Intl. Conf. on Virtual Execution Environments*, pages 31–44, 2019.

[Mat22a] MathWorks. Block Target File Methods, 2022. https://www.mathworks.com/help/rtw/tlc/block-target-file-methods.html.

[Mat22b] MathWorks. Call Custom C/C++ Code from the Generated Code, 2022. https://www.mathworks.com/help/coder/ug/call-cc-code-from-matlab-code.html.

[Mat22c] MathWorks. Create a Basic C MEX S-Function, 2022. https://www.mathworks.com/help/simulink/sfg/example-of-a-basic-c-mex-s-function.html.

[Mat22d] MathWorks. Create Block Masks, 2022. https://www.mathworks.com/help/simulink/block-masks.html.

[Mat22e] MathWorks. Generate Source and Header Files with a Custom File Processing (CFP) Template, 2022. https://www.mathworks.com/.

[Mat22f] MathWorks. Simulink Desktop Real-time, 2022. https://www.mathworks.com/products/simulink-desktop-real-time.html.

[Mat22g] MathWorks. Spawn Task Function as Separate Linux Thread, 2022. https://www.mathworks.com/help/supportpkg/armcortexa/ref/linuxtask.html.

[Mat22h] MathWorks. Using Function-Call Subsystems, 2022. https://www.mathworks.com/help/simulink/ug/using-function-call-subsystems.html.

[MBR06] B. Meenakshi, Abhishek Bhatnagar, and Sudeepa Roy. Tool for Translating Simulink Models into Input Language of a Model Checker. In *Formal Methods and Software Engineering*, volume 4260, pages 606–620. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[Mer22] Mercedes-Benz. MB.OS is the "Next Big Thing" - Interview with Dr. Michael Hafner, 2022. https://group.mercedes-benz.com/careers/about-us/mercedes-benz-operating-system/michael-hafner.html.

[mic22] micro-ROS. micro-ROS. https://micro.ros.org/, 2022.

[MMO+95] Allen Brady Montz, David Mosberger, Sean W O'Mally, Larry L Peterson, and Todd A Proebsting. Scout: A communications-oriented operating system. In *Proceedings 5th Workshop on Hot Topics in Operating Systems (HotOS-V)*, pages 58–61. IEEE, 1995.

[MN15] Saad Mubeen and Thomas Nolte. Applying End-to-end Path Delay Analysis to Multi-rate Automotive Systems Developed using Legacy Tools. In *2015 IEEE World Conference on Factory Communication Systems (WFCS)*, pages 1–4. IEEE, 2015.

[MPM+19] Finbarr Murphy, Fabian Pütz, Martin Mullins, Torsten Rohlfs, Dennis Wrana, and Michael Biermann. The Impact of Autonomous Vehicle Technologies on Product Recall Risk. *International Journal of Production Research*, 57(20):6264–6277, 2019.

[MRR+19] Claire Maiza, Hamza Rihani, Juan M Rivas, Joël Goossens, Sebastian Altmeyer, and Robert I Davis. A Survey of Timing Verification Techniques for Multi-Core Real-Time Systems. *ACM Computing Surveys (CSUR)*, 52(3):1–38, 2019.

[MSS+18] Sanaz Mortazavi, Detlef Schleicher, Frank Schade, Carsten Gremzow, and Friedel Gcrfers. Toward Investigation of the Multi-Gig Data Transmission up to 5 Gbps in Vehicle and Corresponding EMC Interferences. In *2018 International Symposium on Electromagnetic Compatibility (EMC EUROPE)*, pages 60–65. IEEE, 2018.

[MST93] Clifford W Mercer, Stefan Savage, and Hideyuki Tokuda. Processor Capacity Reserves for Multimedia Operating Systems. Technical report, Carnegie-Mellon University, Pittsburgh, PA, School of Computer Science, 1993.

[MTS+20] José Martins, Adriano Tavares, Marco Solieri, Marko Bertogna, and Sandro Pinto. Bao: A lightweight static partitioning hypervisor for modern multi-core embedded systems. In *Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[MV13] Charlie Miller and Chris Valasek. Adventures in Automotive Networks and Control Units. *Def Con*, 21:260–264, 2013.

[MWL14] Eric Missimer, Richard West, and Ye Li. Distributed Real-Time Fault Tolerance on a Virtualized Multi-Core System. *OSPERT 2014*, page 17, 2014.

[New22] Newlib. The Newlib Homepage, 2022. `https://sourceware.org/newlib/`.

[NF15] Mitra Nasri and Gerhard Fohler. An Efficient Method for Assigning Harmonic Periods to Hard Real-Time Tasks with Period Ranges. In *2015 27th Euromicro Conference on Real-Time Systems*, pages 149–159, July 2015.

[Nie16] Georg Niedrist. Deterministic Architecture and Middleware for Domain Control Units and Simplified Integration Process Applied to ADAS, 2016. https://www.tttech.com/technologies/adas.

[Nur22] Nuro. Nuro - Less Driving. More Thriving., 2022. `https://www.nuro.ai/`.

[Nvi22a] Nvidia. DRIVE OS and SDK. https://docs.nvidia.com/drive/, 2022.

[Nvi22b] Nvidia. Hardware for Self-Driving Cars, 2022. https://www.nvidia.com/en-us/self-driving-cars/drive-platform/hardware/.

[Ope22] OpenSynergy GmbH. COQOS HYPERVISOR SDK, 2022. https://www.opensynergy.com/automotive-hypervisor/.

[OTNK18] Yutaka Onuma, Yoshiaki Terashima, Sumika Nakamura, and Ryozo Kiyohara. A method of ECU Software Updating. In *2018 International Conference on Information Networking (ICOIN)*, pages 298–303. IEEE, 2018.

[PFF+18] Claire Pagetti, Julien Forget, Heiko Falk, Dominic Oehlert, and Arno Luppold. Automated Generation of Time-Predictable Executables on Multicore. In *the 26th ACM International Conference on Real-Time Networks and Systems*, pages 104–113, France, October 2018.

[PIBM11] Roberto Pineiro, Kleoni Ioannidou, Scott A Brandt, and Carlos Maltzahn. Radflows: Buffering for predictable communication. In *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 23–33. IEEE, 2011.

[PSG+14] Claire Pagetti, David Saussié, Romain Gratia, Eric Noulard, and Pierre Siron. The ROSACE Case Study: From Simulink Specification to Multi/Many-Core Execution. In *the 19th IEEE RTAS*, pages 309–318, April 2014.

[Pyo21] Pyomo. Pyomo Documentation. https://readthedocs.org/projects/pyomo/downloads/pdf/stable/, 2021.

[QCG+09] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, Andrew Y Ng, et al. ROS: an open-source Robot Operating System. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, Japan, 2009.

[RBH+18] Debayan Roy, Michael Balszun, Thomas Heurung, Samarjit Chakraborty, and Amol Naik. Waterfall Is Too Slow, Let's Go Agile: Multi-domain Coupling for Synthesizing Automotive Cyber-Physical Systems. In *ICCAD*, pages 1–7, November 2018.

[RG14] Robert Reicherdt and Sabine Glesner. Formal Verification of Discrete-Time MATLAB/Simulink Models Using Boogie. In *Software Engineering and Formal Methods*, volume 8702, pages 190–204. Springer International Publishing, Cham, 2014.

[RIM16] RIMAC Automobili. Rimac All Wheel Torque Vectoring. *Press Release*, 2016.

[RKLM17] Ralf Ramsauer, Jan Kiszka, Daniel Lohmann, and Wolfgang Mauerer. Look Mum, No VM Exits! (Almost). *arXiv preprint arXiv:1705.06932*, 2017.

[RMF19] Federico Reghenzani, Giuseppe Massari, and William Fornaciari. The Real-time Linux Kernel: A Survey on PREEMPT_RT. *ACM Computing Surveys (CSUR)*, 52(1):1–36, 2019.

[Rus81] John M Rushby. Design and Verification of Secure Systems. *ACM SIGOPS Operating Systems Review*, 15(5):12–21, 1981.

[Rus02] John Rushby. Model checking simpsons four-slot fully asynchronous communication mechanism. *Computer Science Laboratory–SRI International, Tech. Rep. Issued*, 2002.

[SB07] Jorg Sommer and Rainer Blind. Optimized Resource Dimensioning in an Embedded CAN-CAN Gateway. In *the International Symposium on Industrial Embedded Systems*, pages 55–62. IEEE, 2007.

[SCB+13] Stephan Sommer, Alexander Camek, Klaus Becker, Christian Buckl, Andreas Zirkler, Ludger Fiege, Michael Armbruster, Gernot Spiegelberg, and Alois Knoll. RACE: A Centralized Platform Computer Based Architecture for Automotive Applications. In *2013 IEEE International Electric Vehicle Conference (IEVC)*, pages 1–6. IEEE, 2013.

[sci21] scipy. Scipy Minimize Error - GitHub Issues. `https://github.com/scipy/scipy/issues/3056`, 2021.

[sci22] SciPy: An open-source scientific computing library in Python. `https://www.scipy.org/`, 2022.

[SE16] Johannes Schlatow and Rolf Ernst. Response-Time Analysis for Task Chains in Communicating Threads. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–10, April 2016.

[SFW22] Soham Sinha, Anam Farrukh, and Richard West. ModelMap: A Model-based Multi-domain Application Framework for Centralized Automotive Systems. In *41st IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2022.

[SGEW20a] Soham Sinha, Ahmad Golchin, Craig Einstein, and Richard West. A Paravirtualized Android for Next Generation Interactive Automotive Systems. In *Proceedings of HotMobile*, pages 50–55, 2020.

[SGEW20b] Soham Sinha, Ahmad Golchin, Craig Einstein, and Richard West. Poster: A Paravirtualized Android for Next Generation Interactive Automotive Systems. In *Proceedings of the 21st International Workshop on Mobile Computing Systems and Applications*, HotMobile '20, page 100, Austin, TX, USA, March 2020. Association for Computing Machinery.

[Sim90] H.R. Simpson. Four-slot Fully Asynchronous Communication Mechanism. *IEEE Computers and Digital Techniques*, 137:17–30, January 1990.

[Sim22] Simulink. Extending Embedded and Generic Real-Time System Target Files, 2022. `https://www.mathworks.com/help/physmod/simscape/ug/extending-embedded-and-generic-real-time-targets.html`.

[Sin18] Soham Sinha. Scheduling Policies and System Software Architectures for Mixed-Criticality Computing. Technical Report, Department of Computer Science, Boston University, December 2018.

[SSL89] Brinkley Sprunt, Lui Sha, and John Lehoczky. Scheduling Sporadic and Aperiodic Events in a Hard Real-time System. Technical report, Carnegie-Mellon University, Pittsburgh, PA, Software Engineering Institute, 1989.

[SST07] Koichi Shigematsu, Takayuki Sekisue, and Kimitoshi Tsuji. The Automotive System Simulation by Using Multi Domain Modeling Technique. In *2007 European Conference on Power Electronics and Applications*, pages 1–8, September 2007.

[Sto20] Stout. 2020 Automotive Defect & Recall Report, 2020. `https://www.stout.com/en/insights/report/2020-automotive-defect-and-recall-report`.

[SW21] Soham Sinha and Richard West. Towards an Integrated Vehicle Management System in DriveOS. *ACM Transactions on Embedded Computing Systems (TECS)*, 20(5s):1–24, 2021.

[SW22] Soham Sinha and Richard West. End-to-end Scheduling of Real-time Task Pipelines on Multiprocessors. *Journal of Systems Research*, 2(1), August 2022.

[SWG20] Soham Sinha, Richard West, and Ahmad Golchin. PAStime: Progress-Aware Scheduling for Time-Critical Computing. In Marcus Völp, editor, *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*, volume 165 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 3:1–3:24, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[TE 18] TE Connectivity. The Car in the Age of Connectivity: Enabling Car to Cloud Connectivity. *IEEE Spectrum: Technology, Engineering, and Science News*, 2018.

[Tes21] Tesla. Autopilot. `https://www.tesla.com/support/autopilot`, 2021.

[Tes22] Tesla. Linux. https://github.com/teslamotors/linux, 2022.

[TFG+20] Yue Tang, Zhiwei Feng, Nan Guan, Xu Jiang, Mingsong Lv, Qingxu Deng, and Wang Yi. Response Time Analysis and Priority Assignment of Processing Chains on ROS2 Executors. In *2020 IEEE Real-Time Systems Symposium (RTSS)*, pages 231–243, December 2020.

[The21] The Linux Foundation. Automotive Grade Linux, 2021. `https://www.automotivelinux.org/`.

[The22] The RTEMS Project. Real-Time Executive for Multiprocessor Systems, 2022. `https://www.rtems.org/`.

[UOO15] Moisés Urbina, Zaher Owda, and Roman Obermaisser. Simulation Environment based on SystemC and VEOS for Multi-Core Processors with Virtual AUTOSAR ECUs. In *2015 IEEE International Conference on Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing*, pages 1843–1852. IEEE, 2015.

[vdVD04] Peter H van der Veen and Dan T Dodge. Distributed Kernel Operating System, February 24 2004. US Patent 6,697,876.

[Ves07] Steve Vestal. Preemptive Scheduling of Multi-criticality Systems with Varying Degrees of Execution Time Assurance. In *the 28th IEEE International Real-Time Systems Symposium (RTSS)*, pages 239–243, 2007.

[Way22] Waymo. Waymo - formerly the Google self-driving car project, 2022. `https://waymo.com/`.

[WB06] Andreas Wächter and Lorenz T Biegler. On the Implementation of an Interior-point Filter Line-search Algorithm for Large-scale Nonlinear Programming. *Mathematical programming*, 106(1):25–57, 2006.

[WEE+08] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, et al. The Worst-case Execution-time Problem – Overview of Methods and Survey of Tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):1–53, 2008.

[Wes22] West, Richard. Quest Operating System, 2022. `http://questos.org`.

[WHKA13] Bryan C Ward, Jonathan L Herman, Christopher J Kenna, and James H Anderson. Making Shared Caches More Predictable on Multicore Platforms. In *2013 25th Euromicro Conference on Real-Time Systems*, pages 157–167. IEEE, 2013.

[Win19] Ally Winning. Number of automotive ECUs continues to rise, May 15, 2019. https://www.eenewsautomotive.com/news/number-automotive-ecus-continues-rise.

[Win22a] Wind River. VxWorks — Real-Time Operating System (RTOS), 2022. `https://www.windriver.com/products/vxworks`.

[Win22b] Wind River. Wind River Linux, 2022. `https://www.windriver.com/products/linux`.

[WLMD16] Richard West, Ye Li, Eric Missimer, and Matthew Danish. A Virtualized Separation Kernel for Mixed-Criticality Systems. *ACM Transactions on Computer Systems*, 34(3):8:1–8:41, June 2016.

[XCÅ16] Yang Xu, Anton Cervin, and Karl-Erik Årzén. Harmonic Scheduling and Control Co-design. In *2016 IEEE 22nd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 182–187, August 2016.

[XLXC14] Guoqi Xie, Renfa Li, Xiongren Xiao, and Yuekun Chen. A High-Performance DAG Task Scheduling Algorithm for Heterogeneous Networked Embedded Systems. In *2014 IEEE 28th International Conference on Advanced Information Networking and Applications*, pages 1011–1016, May 2014.

[Y+99] Victor Yodaiken et al. The RTLinux Manifesto. In *Proc. of the 5th Linux Expo*, 1999.

[YCSW18] Ying Ye, Zhuoqun Cheng, Soham Sinha, and Richard West. vLibOS: Babysitting OS Evolution with a Virtualized Library OS. *arXiv preprint arXiv:1801.07880*, 2018.

[YMWP14] Heechul Yun, Renato Mancuso, Zheng-Pei Wu, and Rodolfo Pellizzoni. PALLOC: DRAM Bank-aware Memory Allocator for Performance Isolation on Multicore Platforms. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 155–166. IEEE, 2014.

[YWZC16] Ying Ye, Richard West, Jingyi Zhang, and Zhuoqun Cheng. MARACAS: A Real-Time Multicore VCPU Scheduling Framework. In *2016 IEEE Real-Time Systems Symposium (RTSS)*, pages 179–190. IEEE, 2016.

[ZDB+20] Shuai Zhao, Xiaotian Dai, Iain Bate, Alan Burns, and Wanli Chang. DAG Scheduling and Analysis on Multiprocessor Systems: Exploitation of Parallelism and Dependency. In *2020 IEEE Real-Time Systems Symposium (RTSS)*, pages 128–140. IEEE, 12/2020.

[Zoo22] Zoox. Zoox - The Future is for Riders., 2022. `https://zoox.com/`.

[ZW06] Yuting Zhang and Richard West. Process-aware Interrupt Scheduling and Accounting. In *2006 27th IEEE International Real-Time Systems Symposium (RTSS'06)*, pages 191–201. IEEE, 2006.

[ZZLN09] Hongzi Zhu, Yanmin Zhu, Multicast Li, and Lionel M Ni. SEER: Metropolitan-scale traffic perception based on lossy sensory data. In *IEEE INFOCOM 2009*, pages 217–225. IEEE, 2009.

Curriculum Vitae

Contact

Soham Sinha

soham1@bu.edu

Department of Computer Science, Boston University, 111 Cummington Mall, Boston, MA 02215, USA

Education

Boston University Ph.D. Candidate, Computer Science, September 2016 – present. Thesis Advisor: Richard West

University of Alberta, M.Sc., Computing Science, September 2014 – June 2016. Thesis Advisor: Paul Lu and Di Niu

IIEST Shibpur (formerly BESU, Shibpur), B.E., Computer Sceince and Technology, Jul 2008 – Apr 2012.

Publications

Soham Sinha, Anam Farrukh, Richard West, **ModelMap: A Model-based Multi-domain Application Framework for Centralized Automotive Systems**, in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, San Diego, CA, USA, November 2022

Soham Sinha, and Richard West, **End-to-end Scheduling of Real-time Task Pipelines on Multiprocessors**, *Journal of Systems Research (JSys)*, 2(1), August 2022

Soham Sinha, and Richard West, **Towards Integrated Vehicle Management in DriveOS**, in *Proceedings of the ACM SIGBED International Conference on Embedded Software (EMSOFT)*, October 08-15, 2021, (published in *ACM Transactions on Embedded Computing Systems (TECS)*, Volume 20, Issue 5s, October 2021, Article No.: 82)

Soham Sinha, Richard West, and Ahmad Golchin, **PAStime: Progress-Aware Scheduling for Time-Critical Computing**, in *Proceedings of the 32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*, July 7-10, 2020

Ahmad Golchin, Soham Sinha, and Richard West, **Boomerang: Real-Time I/O Meets Legacy Systems**, in Proceedings of the 26th *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2020)* April 21-24, 2020

Soham Sinha, Ahmad Golchin, Craig Einstein, and Richard West, **A Paravirtualized Android for Next Generation Interactive Automotive Systems**, in *Proceedings of the 21st International Workshop on Mobile Computing Systems and Applications (HotMobile 2020)*, Austin, Texas, USA, March 3-4, 2020

Soham Sinha, **Scheduling Policies and System Software Architectures for Mixed-criticality Computing**, Technical Report BUCS-TR-2018-001, Department of Computer Science, Boston University, December, 2018

Ying Ye, Zhuoqun Cheng, Soham Sinha, and Richard West, **vLibOS: Babysitting OS Evolution with a Virtualized Library OS**, 2018, arXiv:1801.07880

Soham Sinha, Di Niu, Zhi Wang, and Paul Lu, **Mitigating Routing Inefficiencies to Cloud-Storage Providers: A Case Study**, in IEEE *International Parallel and Distributed Processing Symposium (IPDPS) Workshop (DPDNS)*, May 2016, Chicago, USA

Awards

Among 3 Best Papers at EMSOFT 2021 - Towards Integrated Vehicle Management in DriveOS