

面向自动驾驶 SOAFEE 框架的容器参数优化配置方法研究

陶泓雨¹⁾ 陈钰婷²⁾ 孙景昊²⁾ 管思成¹⁾ 廖顺鑫¹⁾
李延恺¹⁾ 邓庆绪¹⁾

¹⁾(东北大学计算机科学与工程学院 沈阳 110819)

²⁾(大连理工大学计算机科学与技术学院 辽宁 大连 116024)

摘 要 SOAFEE(Scalable Open Architecture for Embedded Edge)架构作为面向嵌入式边缘的可扩展开源框架,通过容器化和微服务技术实现了云端测试与车载部署的一体化。然而,SOAFEE架构多层耦合且运行环境动态变化,使得自动驾驶系统的时序分析变得极为复杂,传统实时性分析方法难以适用。在SOAFEE架构中,容器作为功能模块的部署载体,其配置参数(如CPU、内存等)直接决定了任务的调度与执行效率,进而影响了系统整体的时序行为。研究表明,不同的容器配置对自动驾驶系统的端到端时延影响显著。如何在满足端到端时序约束的前提下自动优化容器参数配置,是SOAFEE成功应用于自动驾驶的关键挑战。本文提出了一种构建在测试之上设计(Design Upon Test, DUT)的容器参数优化方法,该方法不依赖传统的静态实时理论,而是基于实际测试获得容器在不同配置下的实时性能(子路径端到端时延),并在此基础上构建优化问题,寻求全局最优的容器配置组合。本文提出基于动态规划思想的链式组合优化算法(Chain Composition Optimization Algorithm, CCOA)求解这一问题,能够在多维资源约束条件下高效搜索高质量的容器配置方案。实验证明,在典型的Autoware自动驾驶系统场景下,本文方法能够有效保障多容器、多任务链的端到端时延。与贪心算法相比,本方法可将路径截止期满足率提升约50%;与传统混合整数线性规划(Mixed-Integer Linear Programming, MILP)的方法相比提升约44%;与Kubernetes下的垂直Pod自适应调整(Vertical Pod Autoscaler, VPA)算法相比提升约58.14%。结果表明,本文所提出的方法在满足时序约束的同时实现了资源分配优化,为SOAFEE架构下自动驾驶系统的端到端时序保障与容器资源优化配置提供了有效的解决方案。

关键词 端到端时延;容器配置;构建在测试之上的设计(DUT);自动驾驶;可扩展嵌入式边缘开放架构(SOAFEE)

中图法分类号 TP311

DOI号 10.11897/SP.J.1016.2026.01801

Container Parameter Optimization for the SOAFEE Framework in Autonomous Driving

TAO Hong-Yu¹⁾ CHEN Yu-Ting²⁾ SUN Jing-Hao²⁾ GUAN Si-Cheng¹⁾ LIAO Shun-Xin¹⁾
LI Yan-Kai¹⁾ DENG Qing-Xu¹⁾

¹⁾(School of Computer Science and Engineering, Northeastern University, Shenyang 110819)

²⁾(School of Computer Science and Technology, Dalian University of Technology, Dalian, Liaoning 116024)

Abstract The Scalable Open Architecture for Embedded Edge (SOAFEE) provides a scalable open-source framework designed for embedded edge systems, integrating cloud-based testing

收稿日期:2025-10-20;在线发布日期:2026-05-22。本课题得到国家自然科学基金面上项目(Nos. 62572111, 62472063)、国家自然科学基金重大研究计划(No. 92582120)、辽宁省科技计划项目(No. 2025JH2/101330113)资助。陶泓雨,博士研究生,主要研究领域为实时嵌入式系统、实时调度、容器化。E-mail: taohy@mails.neu.edu.cn。陈钰婷,硕士研究生,主要研究领域为实时调度、容器化。E-mail: 1767822875@mail.dlut.edu.cn。孙景昊(通信作者),博士,教授,主要研究领域为信息物理系统、实时嵌入式多智能体系统、多核并行计算与实时调度。E-mail: jhsun@dlut.edu.cn。管思成,硕士研究生,主要研究领域为实时调度、ROS 2。廖顺鑫,硕士研究生,主要研究领域为实时调度、端到端时延优化。李延恺,博士研究生,主要研究领域为实时嵌入式系统、实时调度、端到端时延优化。邓庆绪,博士,教授,中国计算机学会(CCF)高级会员,主要研究领域为实时嵌入式系统、可重构计算、物联网。

with in-vehicle deployment through containerization and microservices. However, the multi-layer coupling and dynamically changing runtime environment of SOAFEE make timing analysis in autonomous driving systems extremely complex, rendering traditional real-time analysis techniques less applicable. In the SOAFEE architecture, containers serve as deployment units for functional modules, and their configuration parameters (e. g. , CPU, memory) directly determine task scheduling and execution efficiency, thereby affecting the system's overall timing behavior. Studies show that different container configurations have a significant impact on the end-to-end latency of autonomous driving systems. Thus, how to automatically optimize container configuration parameters while satisfying end-to-end timing constraints has become a key challenge for the successful application of SOAFEE in autonomous driving. To address this challenge, we propose a Design Upon Test (DUT) approach for container parameter optimization. Unlike traditional static real-time theories, this approach relies on empirical testing to obtain real-time performance data (i. e. , subpath end-to-end latency) of containers under different configurations. Based on these measurements, we formulate an optimization problem to search for the globally optimal combination of container configurations. A Chain Composition Optimization Algorithm (CCOA) based on dynamic programming is proposed to efficiently explore high-quality configuration solutions under multi-dimensional resource constraints. Experimental results in a representative Autoware autonomous driving system demonstrate that the proposed method effectively ensures end-to-end timing constraints for multi-container and multi-task chains. Compared with the greedy algorithm, the proposed method improves the path deadline satisfaction rate by approximately 50%. In comparison with the traditional mixed-integer linear programming (MILP) approach, it achieves an improvement of approximately 44%. Furthermore, compared with the Kubernetes Vertical Pod Autoscaler (VPA) algorithm, the improvement reaches approximately 58.14%. These results indicate that the proposed approach not only satisfies timing constraints but also achieves optimized resource allocation, providing an effective solution for end-to-end timing assurance and container resource optimization in SOAFEE-based autonomous driving systems.

Keywords end-to-end latency; container configuration; design upon test (DUT); autonomous driving; scalable open architecture for embedded edge (SOAFEE)

1 引 言

随着汽车行业加速迈向软件定义汽车 (Software Defined Vehicle, SDV) 时代, 车辆功能日益依赖软件, 尤其是 L5 级自动驾驶车辆的软件规模预计到 2030 年将达到 10 亿行^[1]。这一增长推动了车企引入 OTA (Over-The-Air) 升级机制, 以实现快速功能迭代和持续增值服务。然而, 频繁的软件更新带来了新的挑战, 尤其在自动驾驶系统中, 如何在保证实时性和安全性的同时加速迭代成为亟待解决的问题^[1]。自动驾驶控制系统通常要求在 100 ms 内响应传感器数据以确保及时决策^[2], 而传统道路测试无法全面验证系统安全性, 且验证成本高昂^[3], 这

促使了“云端验证+边缘部署”策略的发展。为应对这些挑战, SOAFEE (Scalable Open Architecture for Embedded Edge) 架构应运而生, 旨在通过云原生技术、容器化和微服务等, 实现在云端开发与车载软件部署的一体化^[4]。SOAFEE 架构通过容器化技术实现轻量级虚拟化, 隔离软件服务并保持接近原生性能, 推动了软件交付和更新的加速^[5]。然而, SOAFEE 架构的多层耦合特性 (应用层、ROS 2 中间件、宿主操作系统等) 引入了调度延迟、资源争用和不确定性抖动, 使得系统时序行为异常复杂^[6]。

面对这种复杂的多层架构, 经典实时系统理论在分析和保障系统时序时面临根本性挑战。首先, 基于最坏情况执行时间 (Worst Case Execution Time, WCET) 的分析依赖严格的系统假设, 难以

覆盖数据变化和中间件通信等多种动态因素^[5,7]。其次,传统单层调度策略(如EDF、RM)假设系统调度环境稳定且可预测,无法应对多容器任务链中的性能耦合与并发干扰^[8]。更关键的是,容器化微服务架构中,每个功能模块的资源配置(如CPU和内存)直接影响执行时延,进而影响整个任务链的端到端响应时间。单纯依赖CPU利用率等单一指标,难以准确刻画整体时延行为^[9]。根本原因在于多容器任务链中存在非线性性能耦合,单个容器负载波动可能引发任务链全局时延的显著变化^[10]。因此,必须基于端到端任务模型,综合考虑容器资源配置与任务链特性,以有效评估整体时序。

针对上述挑战,本文提出一种构建在测试之上的设计(Design Upon Test, DUT)的容器配置参数优化方法。该方法从端到端系统视角出发,通过测试驱动与组合优化算法弥补传统静态分析的不足。具体来说,DUT方法以端到端模型为基础,将自动驾驶系统抽象为有向无环图(DAG)进行描述^[8]。通过系统化测试采集各容器在不同资源参数(如CPU、内存)配置下的任务子路径执行时延,建立容器配置与模块时延间的映射模型。本文通过组合各子路径时延推断关键路径时延是否满足约束,并在全局范围内寻找最优容器配置组合,以确保关键路径时延满足要求的同时最小化资源分配,实现时延安全与资源配置的协同优化。为高效求解该模型,本文提出了链式组合优化算法(Chain Composition Optimization Algorithm, CCOA),在多任务链依赖和多资源约束下,能够有效选择最优容器配置。与依赖严格假设和静态分析的传统方法不同,DUT方法基于真实测试数据,更加贴近容器化系统的实际运行特性,为SOAFEE多层架构环境下的自动驾驶系统提供了可靠的时延保障。

本文的主要贡献如下:(1)提出构建在测试之上的设计的容器参数优化方法,通过系统测试测量不同配置下的子路径时延,建立容器配置与时延的映射;(2)设计链式组合优化算法(CCOA),能够在多任务链相互依赖与多资源约束条件下有效选取容器配置组合方案,兼顾任务实时性与资源使用效率;(3)在Autoware自动驾驶框架上进行了验证,实验结果表明该方法可有效保障多任务链端到端时延,提升系统的资源使用效率,为SOAFEE架构下自动驾驶系统的实时性与可靠性保障提供了一条可行的新思路。

2 相关工作

近年来,随着云原生与容器化技术的快速发展,容器化逐渐成为智能驾驶和车载系统软件交付与部署的核心机制。在此背景下,如何在保证系统性能的同时实现高效的容器资源自动化配置优化^[11],并保障系统的实时性,成为研究的核心问题。围绕这一问题,现有研究主要可分为三类:一是聚焦自动驾驶端到端时延分析,二是容器化自动驾驶系统的时序优化研究,三是容器资源参数的自动化配置与调优。

(1)自动驾驶系统的端到端时序分析与优化。在自动驾驶领域,研究者主要从系统建模与调度机制出发,探索端到端时延的可控性与分析建模。Kim等人^[12]针对ROS 2的端到端时延问题,提出CROS-RT跨层优先级调度框架,保障高优先级链端到端时延并构建分析模型推导端到端时延上界以保障可控性。Teper等人^[13]针对自动驾驶场景下的多执行器ROS 2系统,提出覆盖其所有原生通信类型的端到端时延上界分析方法与基于约束规划的ROS 2系统最优配置优化方案,以优化自动驾驶系统的实时性与响应能力。Yano等人^[14]针对自动驾驶系统Autoware on ROS 2提出了一个多截止时间有向无环图(DAG)调度模型,以为复杂的自动驾驶软件栈提供更精确的端到端时序保证。针对自动驾驶端到端时延保障问题,Sun等人^[15]将自动驾驶系统建模为多速率有向无环图,提出集成框架联合分析任务可调度性与关键链端到端时延。这类研究聚焦于自动驾驶系统的时序分析与调度机制,为端到端时延的理论建模和验证提供了重要基础,但多数成果针对的是非容器化或假定执行环境理想化的场景,对容器化带来的资源隔离粒度、内核、中间件交互延迟等实现层面因素关注不足。

(2)自动驾驶场景容器化环境下的时序优化研究。随着容器技术在自动驾驶领域的普及,部分研究开始专门关注容器化部署对系统实时性能的影响,并探索在容器化环境下优化端到端时延的方法。这些工作为ROS 2在容器中的实时应用奠定了基础并提供了直接优化思路。例如,Sollfrank等人^[16]定量评估了Docker对工业实时应用的性能影响,证实容器化引入的额外延迟微小且线性,为ROS 2的容器化部署提供了可行性依据。Betz等人^[2]基于ROS 2/Autoware提出了容器化微服务架构,并系统评估了容器部署对端到端时延的影响,揭示了其在提升性能方面的潜

力。De Marchi 等人^[17]则研究了在 Docker 与 Kubernetes 环境下 ROS 2 节点分布对通信延迟的影响,并通过协议绑定机制有效降低了延迟。更进一步,Betz 等人^[18]针对容器化 ROS 2 自动驾驶软件,整合了优先级配置、通信优化、容器-执行器绑定及周期调整等多种策略,显著降低了端到端时延。总体来看,这类研究多聚焦于通信路径、优先级、IPC 与执行器层面的优化等,为容器化 ROS 2 的时序改进提供了可行方法,但通常未将容器级资源配置作为系统性决策变量加入建模与优化流程。

(3)容器资源参数的自动化配置与调优。针对多容器环境下资源争用及人工调参困难的问题,Cai 等人^[19]提出了 conTuner 框架,将容器参数调优转化为离线-在线协同搜索问题:离线阶段对 CPU、内存、IO 等多维时序特征进行聚类分析,并预生成最优配置;在线阶段通过匹配历史类别并进行微调,实现容器间资源自动化配置与性能优化。针对大规模微服务应用部署后配置参数难以持续优化的问题,Somashekar 等人^[20]提出自动化配置调优框架 OPPerTune,通过引入 HybridBandits 算法以同时处理数值和分类配置参数,并结合 Autoscope 组件自动确定调优实例的范围,减少了手动配置的工作量和潜在错误,实现了高效且可靠的参数优化。这类方法提升了多容器环境下的协同能力与运行性能,但其主要目标在于提高资源利用效率和降低部署开销,对严格的实时性保障考虑不足。

在 Kubernetes 场景下,研究者普遍关注集群层面的资源分配问题。Liu 等人^[21]针对 Kubernetes 场景下的容器放置问题,提出了一种多目标多因子进化算法,通过同时优化多个目标实现了资源的高效利用和负载的均衡分配。曹成成等人^[22]为解决容器化环境中资源利用率低和能耗高的问题,提出了容器多目标资源放置优化 CMR 算法,在 CPU、内存与带宽三维资源约束下,通过匹配度评估实现兼顾能耗与资源利用率的容器最优放置。Jansen 等人^[23]针对 Kubernetes 复杂配置空间,提出 Columbo 框架。该框架通过离线推理方法分析工作负载部署流水线,利用分析模型和基准测试定位性能瓶颈,并基于规则库推荐容器配置参数,有效缩短了工作负载的部署时间。IBM 研究团队提出的 ConfAdvisor 框架^[24]针对容器镜像默认配置性能差及调参门槛高问题,通过自动检测和修正容器镜像与运行时配置优化单应用性能,并结合 Kubernetes 插件动态优化集群参数配置,提供可扩展的规则开发支持与调优建议,提升了集群范围内

容器应用的部署效率和运行性能。针对 Kubernetes 集群在复杂动态负载下资源调度不足的问题,梁拓等人^[25]提出了一种结合长短期记忆网络(LSTM)与遗传算法的智能调度策略,利用 LSTM 模型对未来 CPU 与内存等资源需求进行预测,以应对不同场景下的负载波动,并通过遗传算法优化资源配额,提升云端容器平台的整体资源利用率。虽然这类方法提升了容器云平台的弹性伸缩与整体效率,但其优化目标依然偏向于集群级性能指标,而对端到端任务链的时序保障关注不足。

综上所述,现有工作在容器参数优化方面重点关注容器性能提升、资源利用率最大化和集群效率优化等,而对于 SOAFEE 架构下多容器多任务链端到端时延协同保障仍缺少系统性探索。为此,本文提出一种构建在测试之上设计(DUT)的容器参数优化方法。通过系统化测试获得每个容器在不同配置下的实时性能,进而在全局范围内寻求满足时延约束且资源使用最小的最优配置组合。同时,本文提出链式组合优化算法(CCOA)高效求解此问题。与现有方法相比,本文将容器参数优化与自动驾驶任务链的端到端时延紧密结合,并在 Autoware 框架上进行验证。实验表明,该方法可有效提升多容器多任务链的端到端时序保障能力。

3 相关背景

基于 SOAFEE 的自动驾驶系统采用多层级软件架构,具体分为应用层、中间件层和操作系统层,如图 1 所示。其中,应用层主要包括环境感知、路径规划与控制执行等功能模块及其间的数据流交互,代表性框架包括 Autoware^[26]与 Apollo^[27]。中间件层将应用层的功能模块封装至容器,并为每个功能模块提供基本运行时环境、数据传输协议与接口抽象等服务,典型代表包括 ROS 2^[28]与 CyberRT^[29]。操作系统层则提供进程调度、内存分配、中断响应等底层资源管理服务,使自动驾驶系统能够充分利用异构多核车载硬件资源。接下来详细介绍应用层和中间件层的核心机制及工作原理,操作系统层的细节机制超出本文讨论范围,故不再赘述。

自动驾驶应用层核心架构。以开源自动驾驶框架 Autoware 为例,自动驾驶系统应用层核心功能模块包括传感、定位、地图、规划、感知、控制、车辆及系统^[30],如图 2 所示。其中,传感模块采集来自车载传感器(如激光雷达、摄像头等)的多源原始数据,为系

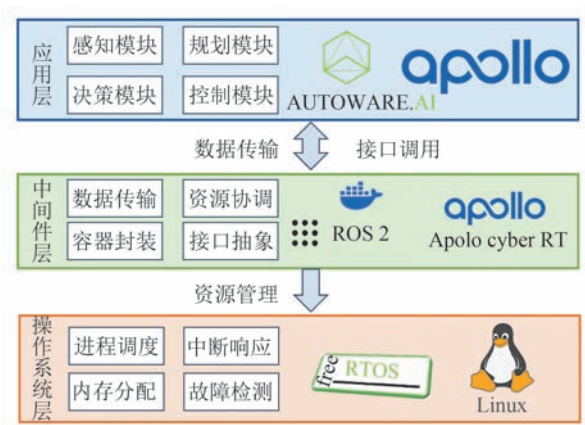


图1 自动驾驶系统多层级软件架构

统提供原始环境信息输入。定位模块融合多传感器设备数据,实时估算车辆精确位置与状态,确保系统具备全局定位能力。地图模块提供高精度语义地图和道路拓扑信息,并结合定位结果实现车辆在地图上的准确投影。感知模块识别并跟踪环境中的动态与静态障碍物,实现对周围场景的理解。规划模块基于感知与地图信息生成可行、安全的行为决策与运动轨迹。控制模块将规划轨迹转化为可执行的转向、加减速等控制命令。车辆模块连接车辆底盘,负责状态读取与控制指令的物理执行。系统模块综合管理调度、模块通信与资源分配,保障各功能模块稳定运行。

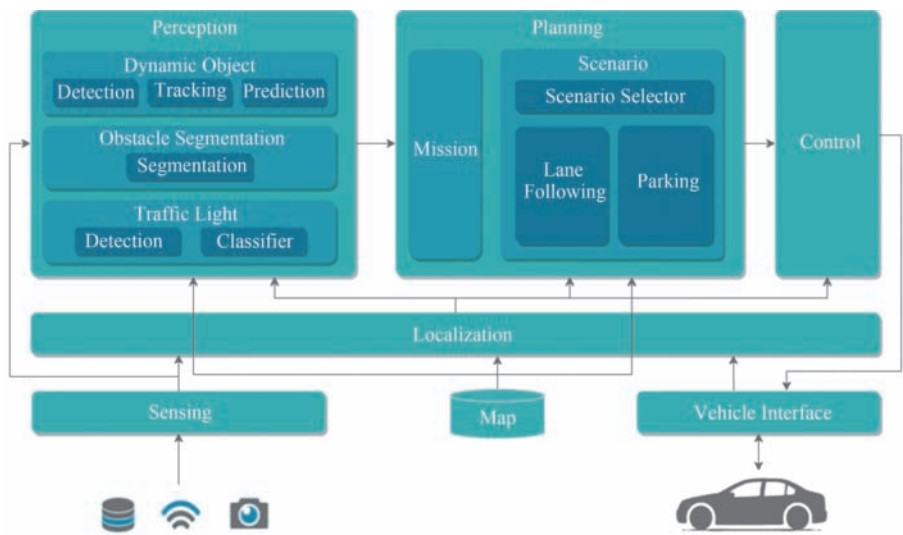


图2 Autoware 核心功能模块图^[31]

在自动驾驶系统中,每个模块由一组功能相关的 ROS 2 节点实现,这些节点协同完成模块任务。Autoware 节点示意图如图 3 所示,其中相同颜色节点表示同一功能模块。具体来说,系统将一个功能模块解耦为多个功能子模块,并将每个子模块的计算任务封装为独立的 ROS 2 节点^[32]。模块节点间通过 ROS 2 消息机制实现任务触发与数据传递。根据触发方式的不同,ROS 2 节点可分为时间触发和事件触发^[33]。当一个节点为时间触发时,其通过定时器以固定周期执行,常用于传感器数据采集与状态刷新等周期性任务。如图 3 所示,虚线框中的 LiDAR 驱动节点以 100 ms 为周期定时释放作业。当一个节点为事件触发时,其由特定前驱节点触发,在接收到特定输入数据时立即释放作业执行相应任务。例如,图 3 感知模块(绿色)中的 euclidean cluster 节点为事件触发,其前驱节点 low height crop box filter 为 euclidean cluster 节点的触发节点。当 low

height crop box filter 节点完成点云裁剪操作后,将结果传给后继节点 euclidean cluster,触发其释放作业。如果一个节点为事件触发且有多多个前驱节点,则指定其中一个前驱节点作为该节点的主触发节点。在节点协同中,当一个节点有多多个前驱节点时,该节点被称为融合节点(如图 3 中圆角节点为典型融合节点)。融合节点会将其所有的前驱节点数据进行融合,生成新的输出数据。融合节点可以是时间触发,也可以是事件触发。以时间触发融合节点为例,图 3 黄色圆角节点 behavior path planner 每隔 100 ms 从其前驱节点 stop filter、twist2accel 和 external velocity limit selector 中读取最新数据,将停车约束、加速度信息与速度限制值进行融合,生成一条符合系统要求的行为路径。事件触发融合节点则由其主触发节点数据驱动。例如,图 3 绿色节点中的 obstacle pointcloud based validator 将 roi detected object fusion 节点作为主触发节点,obstacle pointcloud

参数对端到端时延的影响具体如下：(1)CPU 核心数量：分配的 CPU 核心数量直接决定了容器内部计算任务的处理能力，核心数越多，容器能够执行的线程或进程数量越多，有利于降低任务排队时间与计算时延，减少端到端时延。然而，核心资源的过度分配可能引发上下文切换频繁，增加调度开销，影响系统整体的运行效率与时延稳定性。(2)CPU 核心绑定策略：核心绑定策略对容器调度的稳定性与处理时序产生显著影响。将关键路径上的容器绑定到特定的、隔离的 CPU 核心上，可以减少上下文切换开销和核心间缓存同步延迟，降低关键路径的时延。但是绑定不当则可能导致资源争抢和性能波动，增加路径端到端时延开销。(3)内存容量：内存大小直接影响数据在节点间传输过程中的缓存能力与内存访问延迟。充足的内存能够提高缓存命中率，减少对底层存储的访问，降低数据访问开销与节点处理时延，从而减少端到端时延。然而内存不足会导致页面频繁换入换出，大幅增加数据访问延迟，甚至造成消息丢失或导致传输时延激增。(4)I/O 带宽：I/O 带宽是影响容器间通信效率的关键，决定了数据在跨容器子路径之间的传输速率。带宽受限会造成通信链路拥塞，增加消息传输过程中的排队与重传延迟，影响系统整体实时性。(5)输入频率：容器的输入频率影响任务链的处理负载和时延。高输入频率会增加节点处理和通信的并发需求，容器资源不足会导致任务积压，延长端到端时延。过低频率可能造成资源空闲，降低系统效率。为优化端到端时延，必须针对各关键子路径所在容器，依据其计算及通信特性进行精细化资源配置。

SOAFEE/EWAOL 资源隔离与调度机制。上述容器配置参数在 SOAFEE 架构中通过其参考实现 EWAOL 得到实现。EWAOL 构建了一个分层的资源管控体系，旨在将云原生范式安全延伸至汽车边缘侧，核心挑战在于满足功能安全与实时性要求。

在 CPU 隔离与绑定方面，EWAOL 采用由底向上的策略。在操作系统层，通过 Linux 内核启动参数 `isolcpu` 将指定的物理 CPU 核心从通用调度器中隔离，专用于高实时性任务。在容器运行时层，EWAOL 采用的轻量级运行时 `crun` 会利用 `cgroup` 的 `cpuset` 子系统，通过 `--cpuset-cpus` 参数将容器严格绑定到预隔离的 CPU 核心集合上。这种绑定确保了关键容器独占计算资源，避免了跨核心上下文切换与缓存同步带来的非确定性延迟。

在内存分配方面，EWAOL 通过 `crun` 调用

`cgroup` 的 `memory` 子系统，为每个容器设置内存使用上限和内存预留，防止容器间因内存溢出而相互干扰，也为实时任务提供基本的资源保障。此外，编排层（如 `k3s`）通过声明式的资源请求与限制定义，将应用需求转化为具体的资源约束配置。

该分层管控机制本身也引入了时序不确定性。中间件层将应用逻辑转化为资源请求，操作系统层最终执行调度与分配，这条交互链路会引入额外的调度延迟。叠加中间件自身组件占用 CPU、I/O 资源形成的底层干扰，会直接导致容器响应时间波动，影响自动驾驶等核心场景的实时需求。

4 系统模型

本文将自动驾驶系统建模为有向图 $G=(V, E)$ 。其中，节点集合 $V=\{v_1, v_2, \dots, v_n\}$ 表示自动驾驶系统中的 ROS 2 节点，每个节点 v_i 具有执行时间 τ_i ，表示该节点从触发到完成输出的处理时延。边集合 $E \subseteq V \times V$ 描述节点间的数据依赖关系，每条边 $(v_i, v_j) \in E$ 关联一个缓冲区 $B(v_i, v_j)$ ，用于存储前驱节点 v_i 发布的消息，后继节点 v_j 从中取出消息并触发作业释放。节点集合 V 由时间触发节点集合 V_T 和事件触发节点集合 V_E 组成，即 $V = V_T \cup V_E$ 。时间触发节点 $v_i \in V_T$ 以固定周期 T_i 释放作业，并在释放时从关联的缓冲区读取前驱节点数据；事件触发节点 $v_j \in V_E$ 仅由其指定的单一触发节点 $Trig_j$ 触发并释放作业。对于任一节点 v_i ，将其前驱节点集合定义为 $Pred(v_i)$ ，将后继节点集合定义为 $Succ(v_i)$ 。特殊地，将无前驱节点的节点称为源节点，无后继节点的节点称为汇聚节点。

上述节点和边的定义刻画了系统中 ROS 2 节点的触发与依赖关系，接下来介绍路径的概念以描述节点触发到结果输出的完整执行链路。当节点 v_i 触发时，会产生一个任务实例（作业） j_i ，表示一次具体的执行操作。作业 j_i 执行完成后，其输出数据通过边关联的缓冲区传递至后继节点 v_j ，触发释放作业 j_j 。作业从源节点开始，沿有向图向汇聚节点传播，所形成的一条连续节点序列称为完整路径 $l = \{v_1, v_2, \dots, v_q\}$ ，其中任意相邻节点满足 $(v_i, v_j) \in E$ 。将该路径在最坏情况下从源节点 v_1 触发作业到汇聚节点 v_q 完成作业所经历的总时间定义为最坏端到端时延 $T_{path, l}$ ，并将其需满足的时延约束定义为截止时间 Ddl_l 。本文的目标是基于测试对容器化自动

驾驶系统基础软件参数(CPU、内存)进行配置,以确保各关键路径时延约束得以满足。注意到,自动驾驶系统(即有向图 G)被划分为不同功能模块并被封装到独立容器中,因此完整路径可根据经过的模块划分为若干子路径(详细内容如下文所述)。为保证各关键路径的时延约束,必须基于各子路径的最坏响应时间推导出完整路径的最坏端到端时延,为优化系统参数与最优配置选择提供可靠依据。

前文已将自动驾驶系统建模为有向图,并明确了保障关键路径的端到端时延约束是系统的核心目标,鉴于系统实际由多个容器化功能模块组成,接下来详细介绍功能模块对应的功能图模型,为后续子路径的划分与端到端时延的推导提供支撑。本文将系统有向图 G 按节点功能划分为若干子图,每个子图作为一个连通分支,代表一个功能模块(如感知、规划、控制等),共同构成功能图模型 $G_M=(M, E_M)$ 。集合 $M=\{M_1, M_2, \dots, M_N\}$ 表示 N 个功能模块,每个功能模块 $M_k \in M$ 封装一组ROS 2节点,将其建模为功能子图 $M_k=(V_k, E_k)$,其中 $V_k \subseteq V$ 为功能模块 M_k 包含的节点集合, $E_k=\{(v_i, v_j) \in E | v_i, v_j \in V_k\}$ 表示模块内节点间的连接关系。模块间的通信依赖建模为边集合 $E_M \subseteq M \times M$,由跨模块的节点边映射而来。具体来说,对于任意两个模块 $M_i, M_j \in M$,若存在跨模块的边 $(v_a, v_b) \in E$,其中 $v_a \in M_i$ 且 $v_b \in M_j$,则定义边 $(M_i, M_j) \in E_M$,表示模块 M_i 的节点 v_a 的输出为模块 M_j 的节点 v_b 输入。

为明确模块的功能起点和终点以及跨模块任务触发的时间边界,本文引入入口节点和出口节点的概念。对于模块 M_k ,如果时间触发节点 $v_i \in V_k$ 不存在前驱节点,或时间触发节点 $v_i \in V_k$ 存在位于模块外部的前驱节点 v_j ,则将节点 v_i 定义为模块 M_k 的一个入口节点。模块所有的入口节点构成入口节点集合 I_k ,其被来自系统外部事件触发或前驱模块的输出触发,是模块功能开始执行的起点。如果节点 v_i 不存在后继节点,或存在位于模块外部的后继节点 v_j ,则将节点 v_i 定义为模块 M_k 的出口节点。所有的出口节点构成出口节点集合 O_k ,其输出将作为有关后继模块的输入,用于触发跨模块的任务执行;若无后继模块,则该节点的输出标志整个任务链完成。

基于上述功能图模型,将完整路径根据所经过的模块子图划分为若干子路径,并定义其关键属性。具体来说,若路径 l 依次经过功能模块

$\{M_1, M_2, \dots, M_N\}$,则将其划分为子路径集合 $l=\{l_1, l_2, \dots, l_N\}$,其中子路径 l_i 表示任务在模块 M_i 内部完成的执行流程。每个子路径 l_i 具有以下关键属性:释放周期为 T_i^l ,释放时间 a_i^l ,最坏响应时间 R_i^l 与完成时间 f_i^l 。释放周期为 T_i^l 表示子路径 l_i 的触发间隔,因该子路径的执行始于入口节点的触发,故 T_i^l 值与该子路径的入口节点 v_{in} 周期 T_{in} 一致,即 $T_i^l=T_{in}$ 。子路径释放时间 a_i^l 指子路径 l_i 入口节点触发作业的时间,对应入口节点每次释放任务实例的时刻(如第 j 个实例释放时间为 $a_i^l(j)$)。子路径最坏响应时间 R_i^l 表示最坏情况下子路径 l_i 从入口节点触发到出口节点完成输出所需总时延。需要指出的是,在实际的SOAFEE容器化环境中,子路径执行过程还受到中间件调度、内核调度以及共享资源争用等因素的影响,从而导致实际观测到的响应时间相较于理想隔离条件下的测试结果出现一定程度的放大。为刻画上述干扰,本文引入干扰修正因子 ϵ ,并重新定义干扰修正后的子路径最坏响应时间为 $R_i^l(1+\epsilon)$,其中 ϵ 基于实验数据或经验估算表示由系统底层干扰引起的相对时延放大比例,反映了共置容器间干扰引起的响应时间增幅。子路径完成时间 f_i^l 则指子路径完成全部执行并输出数据的时间,由释放时间与最坏响应时间叠加得到,第 j 个实例完成时间即为 $f_i^l(j)=a_i^l(j)+R_i^l$ 。需要注意的是,本文后续仅分析同一序列的任务实例,故将释放时间与完成时间统一简记为 a_i^l 与 f_i^l 以简化后续时延推导过程中的表述。

在明确子路径的划分方式与关键属性后,接下来通过这些属性推导路径最坏端到端时延的构成,以子路径实时性能判断关键路径是否满足系统时延约束。基于上述划分,路径最坏端到端时延 $T_{path,l}$ 可由各子路径干扰修正后的最坏响应时间 $R_i^l(1+\epsilon)$ 与子路径的等待读取时间 W_i^l 组成:

$$T_{path,l} = \sum_{i=1}^N (R_i^l(1+\epsilon) + W_i^l) \quad (1)$$

其中,等待读取时间 W_i^l 表示子路径 l_i 入口节点为读取前驱子路径的输出结果而产生的等待时间。具体而言,由于路径中各子路径之间存在严格的依赖关系,如果子路径 l_{i+1} 入口节点在定时器触发时未能从缓冲区读取到前驱子路径 l_i 的输出数据,即前驱子路径 l_i 尚未执行完成,则子路径 l_{i+1} 入口节点必须持续等待,直到在后续周期触发点能够成功读取到数据后才释放并执行子路径 l_{i+1} 的作业,即后继子

路径 l_{i+1} 入口节点的最早释放时间需不早于前驱子路径的完成时间。

如图4所示,子路径入口节点释放时间为 a_i^l ,最坏完成时间为 f_i^l 。基于此,后继子路径 l_{i+1} 入口节点的最早释放时间为 $a_{i+1}^l = \left\lceil \frac{f_i^l}{T_{i+1}^l} \right\rceil \cdot T_{i+1}^l$,其中 T_{i+1}^l 为子路径 l_{i+1} 释放周期。此时,子路径 l_{i+1} 入口节点等待前驱子路径 l_i 结果输出而产生等待读取时间,其值为最早释放时间与前驱子路径 l_i 完成时间的差值,即 $W_{i+1}^l = a_{i+1}^l - f_i^l$ 。

由上文可知路径最坏端到端时延由子路径最坏

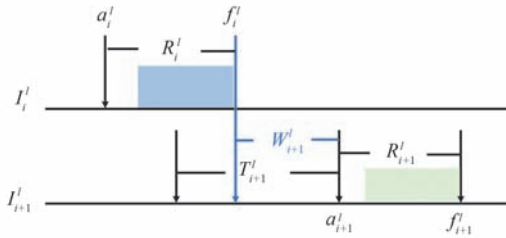


图4 子路径触发与等待读取时间示意图^[36]

响应时间与等待读取时间决定,接下来考虑最坏情况下的等待时间,以推导出路径最坏端到端时延的保守上界。在实际运行时,每段子路径入口节点都可能等待一个完整周期才能触发作业,此时路径最坏端到端时延可推导为保守上界形式,即 $T_{path_l} = \sum_{i=1}^N (R_i^l + T_i^l)$ 。这是因为前驱子路径 l_i 完成时间 f_i^l 恰好错过子路径 l_{i+1} 入口节点当前触发时间点,入口节点必须延迟一个周期才能触发执行,并在执行最坏响应时间 R_{i+1}^l 后产生结果。这种情况可能在路径中所有相邻子路径间重复出现,各子路径入口节点的触发可能刚好错过调度点而被延迟一个周期。如图5所示,路径 l 被划分为在不同模块中的多个子路径 $\{l_1, l_2, l_3\}$,每个子路径入口节点作业释放周期为 T_i^l ,作业释放时间为 a_i^l ,最坏响应时间为 R_i^l ,完成时间为 f_i^l ,其中 $i=1,2,3$ 。在最坏情况下,每个子路径入口节点都错过触发时间点,故路径最坏端到端时延等于各子路径最坏响应时间与入口节点作业释放周期之和。

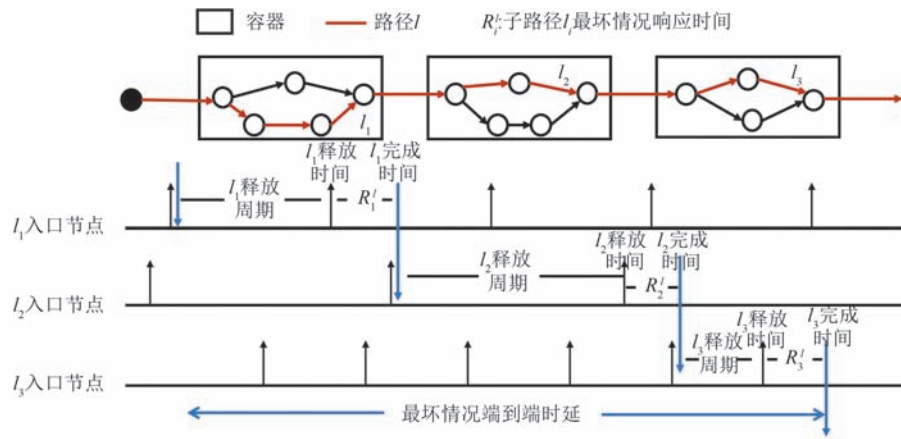


图5 最坏端到端时延计算示意图^[37]

基于对端到端时延的分析,子路径最坏响应时间同样是路径总时延的决定性因素,而其直接依赖于功能模块所在容器的资源配置,为准确预测端到端时延并进行资源优化配置,需建立功能模块到具体容器资源的映射关系。具体来说,将各个功能模块(即子图)映射到独立的容器 $C_i \in Cont$ 中运行,每个容器 C_i 对应一个功能模块 M_i ,并为其分配CPU核心数量和内存资源。在自动驾驶系统中,容器的资源分配方案直接影响模块内任务的调度效率与执行速度,从而影响容器中各子路径响应时间。不同的资源分配方案将导致子路径响应时间变化,进而

影响整个路径的端到端时延。

本文研究目标是通过离线阶段为容器配置合理资源并在系统运行时加载,在满足自动驾驶关键路径时延约束与资源约束的前提下,实现整体资源消耗最小化。容器资源配置直接影响子路径最坏响应时间,并与等待读取时延共同作用于关键路径端到端时延。基于此,容器配置优化问题可形式化为在满足端到端时延约束下的资源最小化问题。即在保证所有关键路径最坏端到端时延不超过其截止期的前提下,最小化系统的总资源消耗。然而,求解该问题需精确建立配置-时延映射。传统实时理论依赖

最坏情况执行时间(WCET)等确定性模型,其在动态容器化环境中因假设过强而失效。

为此,本文提出构建在测试之上的设计(DUT)框架。其核心是采用数据驱动方法,通过系统测试直接表征容器时序行为,构建可靠的配置-性能经验模型,为后续优化提供依据。

5 DUT 求解框架

DUT方法的核心在于承认时序行为本质上是随机的,任务执行时间是容器配置与干扰环境的函数。我们通过系统化测试对函数进行经验评估,建立容器资源配置与容器内子路径最坏响应时间之间的映射关系,以指导容器资源优化分配,实现在满足关键路径时延约束的同时最小化系统资源消耗。接下来将围绕测试方法与优化模型两个方面,系统阐述DUT方法的求解框架,为后续算法实现与容器配置优化提供基础。

在该方法中,首先需获取不同配置下容器响应时间的准确数据,本文在完整的容器化自动驾驶系统的环境中采用单容器配置测试方法获取配置-时延表。测试阶段采用单容器测试以获取在相对隔离条件下的容器时延基准值,用于刻画容器配置与子路径最坏响应时间之间的基本映射关系。具体来说,该方法通过在系统真实运行状态下,仅调整目标容器的资源配置(CPU核心数和内存容量),而其余容器均保持default设置,不再进行额外配置。针对目标容器 C_i ,预设若干候选资源配置方案,形成配置集合 $K_i = \{k_i^1, k_i^2, \dots, k_i^{|K_i|}\}$,其中 $k_i^j = (Core_i^j, Mem_i^j)$ 表示容器 C_i 在第 j 个配置下所分配的CPU核心数量与内存容量。系统在每个候选配置 k_i^j 下运行并收集容器内各子路径 l_i 的最坏响应时间 $R_i^l(k_i^j) = F(l_i, k_i^j)$,其中 $F(l_i, k_i^j)$ 表示子路径时延 l_i 与容器配置 k_i^j 之间的映射关系。为确保实验数据的稳定性和可靠性,每种候选配置均进行5000次重复测试。从所有有效测试值中提取99%分位数,并进一步叠加干扰时延修正(干扰修正因子基于实验测量设为0.3),从而得到该子路径在对应配置下的最坏响应时间。完成当前目标容器的测试后,依次更换其他容器并重复上述过程,最终,对所有容器及其所有候选配置重复上述过程,形成完整的配置-时延表。表1为该表的一个示例片段。

基于配置-时延表与端到端时延公式,可将容器

表1 配置-时延表示例

CPU	内存	入口节点	出口节点	最坏响应时间
1	0.5	FrontLidarDriver	PointsTransformerFront	88.15 ms
1	0.5	RearLidarDriver	PointsTransformerRear	77.47 ms

配置优化问题转为约束条件下的系统资源最小化数学模型。该模型的目标是在满足系统实时性和资源总量限制的前提下,选择一组容器配置实现整体资源利用的最小化。

为建模容器资源开销,定义 $x_i^j \in \{0, 1\}$ 表示容器 C_i 是否选用第 j 个配置 k_i^j ,当且仅当容器选择配置 k_i^j 时, $x_i^j = 1$ 。此时,容器 C_i 所分配的CPU核心数量与内存容量分别为 $Core_i^j$ 和 Mem_i^j 。在此基础上,本文的优化目标是 minimized 系统总资源开销,即归一化的加权CPU与内存消耗之和,如公式(2)所示:

$$\min Cost_{sys} = \alpha \cdot \frac{\sum_i \sum_{j \in K_i} x_i^j \cdot Core_i^j}{Core_{max}} + \beta \cdot \frac{\sum_i \sum_{j \in K_i} x_i^j \cdot Mem_i^j}{Mem_{max}} \quad (2)$$

其中, α 和 β 分别表示CPU与内存的权重系数, $Core_{max}$ 和 Mem_{max} 分别表示系统可分配的核心总数与内存总量。

该问题需要满足以下条件:

$$\sum_{j \in K_i} x_i^j = 1, x_i^j \in \{0, 1\}, \forall i \in Cont \quad (3)$$

$$\sum_{i \in Cont} \sum_{j \in K_i} x_i^j \cdot Core_i^j \leq Core_{max} \quad (4)$$

$$\sum_{i \in Cont} \sum_{j \in K_i} x_i^j \cdot Mem_i^j \leq Mem_{max} \quad (5)$$

$$T_{path,l} \leq Ddl_l, \forall l \in P_{crit} \quad (6)$$

公式(3)表示每个容器 C_i 只能选择配置集 K_i 中的一个配置 k_i^j ;公式(4)和(5)要求所有容器配置总资源(CPU与内存)消耗均不能超过系统资源上界;公式(6)要求关键路径 l 的最坏端到端时延 $T_{path,l}$ 不能超过其对应截止时间 Ddl_l ,其中 P_{crit} 为关键路径集合,需注意 $T_{path,l}$ 是基于单容器测试获取的子路径最坏响应时间结合端到端时延公式计算得出,确保时延约束与实测数据直接关联。

与传统先设计后测试的方法不同,DUT方法避免了传统静态分析对严格假设的依赖。DUT方法以测试为核心,通过系统运行测试得到实际响应时间数据,并在此基础上进行优化设计。该方法充分考虑系统实际运行特性,使容器配置优化结果更贴近真实系统性能。

6 链式组合优化算法

为满足关键路径端到端时延约束与整体实时性要求,同时最小化总资源消耗,需要在全局范围内为所有容器联合选择最优配置。为求解该问题,本文提出基于动态规划思想的链式组合优化算法(CCOA),以在多维资源约束条件下探索高质量的

容器配置方案。算法首先将各容器候选配置抽象为统一的基本数据结构,通过拼接操作更新资源消耗与时延状态,并使用优化策略减少状态空间,直至构建覆盖全系统的最优容器配置组合。下面将详细介绍算法的基本数据结构定义、拼接操作、整体流程设计及优化策略,并对算法复杂度进行分析。为便于后续模型与算法描述,本文将涉及的主要符号及其含义统一汇总如表2所示。

表2 符号说明表

符号	含义说明	符号	含义说明
$G=(V,E)$	自动驾驶系统的有向图模型	T_i^l	子路径 l_i 的释放周期
$V=\{v_1,v_2,\cdots,v_n\}$	ROS 2 节点集合	$a_i^l(j)/a_i^l$	子路径 l_i 入口节点第 j 个实例释放时间
τ_i	节点 v_i 的执行时间	R_i^l	子路径 l_i 最坏响应时间
$E\subseteq V\times V$	边集合,表示节点间数据依赖关系	$f_i^l(j)/f_i^l$	子路径 l_i 上第 j 个实例完成时间
$B(v_i,v_j)$	边 $(v_i,v_j)\in E$ 的缓冲区	W_i^l	子路径 l_i 的等待读取时间
V_T,V_E	时间、事件触发节点集合	$Cont$	容器集合
T_i	时间触发节点 v_i 的触发周期	C_i	第 i 个容器
$Pred(v_i),Succ(v_i)$	节点 v_i 前驱/后继节点集合	K_i	容器 C_i 的配置集合
j_i	节点 v_i 的任务实例(作业)	k_i^j	容器 C_i 的第 j 个配置
l	端到端路径	$Core_i^j,Mem_i^j$	配置 k_i^j 下容器 C_i 分配的 CPU、内存
P_{crit}	关键路径集合	$R_i^l(k_i^j)$	配置 k_i^j 下子路径 l_i 的最坏响应时间
$T_{path,l}$	路径 l 最坏端到端时延	$F(l_i,k_i^j)$	子路径时延 l_i 与容器配置 k_i^j 的映射关系
Ddl_l	路径 l 的截止期	$x_i^j\in\{0,1\}$	容器 C_i 是否选择配置 k_i^j 的决策变量
M_k	第 k 个功能模块	$Core_{max},Mem_{max}$	系统可分配的核心总数与内存总量
$M_k=(V_k,E_k)$	模块 M_k 的子图模型	L_i	容器 C_i 中所有潜在子路径集合
I_k,O_k	模块 M_k 的入口/出口节点集合	$D(k_i^j)$	L_i 在配置 k_i^j 下的最坏响应时间集合
l_i	路径 l 在模块 M_i 内部的子路径	$d_p(k_i^j)$	潜在子路径 $p\in L_i$ 在配置 k_i^j 下的最坏响应时间

基本数据结构定义。为建立容器配置与时延特征之间的映射关系,首先定义基本数据结构及其对应的时延向量表示。假设每个容器 C_i 定义的入口节点集合为 I_i ,出口节点集合为 O_i ,并有配置 k_i^j 所对应的核心数 $Core_i^j$ 和内存容量 Mem_i^j ,则将该配置扩展为基本数据结构 $k_i^j=(Core_i^j,Mem_i^j,I_i,O_i)$ 。在此基础上,容器 C_i 的所有基本数据结构形成集合 $K_i=\{k_i^1,k_i^2,\cdots,k_i^{|K_i|}\}$,所有容器的结构形成初始结构集合 $K=\{K_1,K_2,\cdots,K_N\}$ 。容器 C_i 入口节点集合 I_i 与出口节点集合 O_i 的所有潜在入口-出口对构成潜在子路径集合 $L_i=\{p=(v_{in},v_{out})|v_{in}\in I_i,v_{out}\in O_i\}$, p 表示容器内部一条从入口节点 v_{in} 到出口节点 v_{out} 的潜在子路径。对于容器 C_i 的配置 $k_i^j\in K_i$,其关联向量定义为 $D(k_i^j)=\{d_p(k_i^j)|p\in L_i\}$,其中 $d_p(k_i^j)$ 表示容器 C_i 中潜在子路径 $p\in L_i$ 在配置 k_i^j 下的最坏响应时间。若子路径 $p\in L_i$ 为任务图中实际存在,则 $d_p(k_i^j)$

取该配置下实测所得的子路径最坏响应时间;若子路径并不存在于任务图中,则该分量取 $+\infty$,表示该子路径在该任务图下不可达。容器 C_i 所有结构对应的向量集合可表示为 $D(K_i)=\{D(k_i^1),D(k_i^2),\cdots,D(k_i^{|K_i|})\}$,进一步地,系统所有容器的结构集合 $K=\{K_1,K_2,\cdots,K_N\}$ 对应的时延向量集合 $D(K)=\{D(K_1),D(K_2),\cdots,D(K_N)\}$ 。

拼接计算操作。为在系统任务图中刻画容器间的组合关系与路径时延的累积效应,本文在系统模型的基础上引入拼接计算操作,用于对沿数据流方向相邻的容器结构进行递归组合,该操作本质上是对 DAG 图中可达子路径集合的扩展。具体来说,若两个容器和在数据流方向上连通,即存在边 $(v_a,v_b)\in E$,其中 $v_a\in O_i,v_b\in I_n$ (对称方向同理),则容器对应的基本数据结构 $k_i^j=(Core_i^j,Mem_i^j,I_i,O_i)$ 和 $k_n^m=(Core_n^m,Mem_n^m,I_n,O_n)$ 在任务图上构成可达

关系,可进行拼接计算,生成新的数据结构 $k_{\text{new}} = (Core_{\text{new}}, Mem_{\text{new}}, I_{\text{new}}, O_{\text{new}})$ 。

在资源维度上,容器资源配置保持独立且不可共享,新结构的资源消耗等于两个子结构资源需求的和,即 $Core_{\text{new}} = Core_i^j + Core_n^m$, $Mem_{\text{new}} = Mem_i^j + Mem_n^m$ 。在节点集合维度上,拼接操作通过消除已建立跨容器依赖的中间边界节点,将跨容器的数据依赖关系转为组合结构的内部关联关系,从而实现组合结构的节点更新。具体而言,入口节点集合仅保留在拼接后无前驱节点的节点,出口节点集合仅保留无后继节点的节点,更新规则形式化表示如公式(7)(8)所示。

$$I_{\text{new}} = (I_i \cup I_n) \setminus B_{\text{in}} \quad (7)$$

$$O_{\text{new}} = (O_i \cup O_n) \setminus B_{\text{out}} \quad (8)$$

其中, $B_{\text{in}} = \{v_i \in I_n \mid \exists v_j \in O_i, (v_j, v_i) \in E\}$ 与 $B_{\text{out}} = \{v_i \in O_i \mid \exists v_j \in I_n, (v_i, v_j) \in E\}$ 分别表示跨容器依赖被内部化的入口和出口节点。如图6中容器入口节点集合分别为 $I_i = \{a, b\}$ 和 $I_n = \{e, f\}$, 出口节点集合分别为 $O_i = \{c, d\}$ 和 $O_n = \{g, h\}$, 根据上述规则更新后得到的新入口节点集合为 $I_{\text{new}} = \{a, b, f\}$, 新出口节点集合为 $O_{\text{new}} = \{d, g, h\}$ 。若相反方向可进行拼接计算操作,即 O_n 与 I_i 相连,则也可按对称规则进行更新。

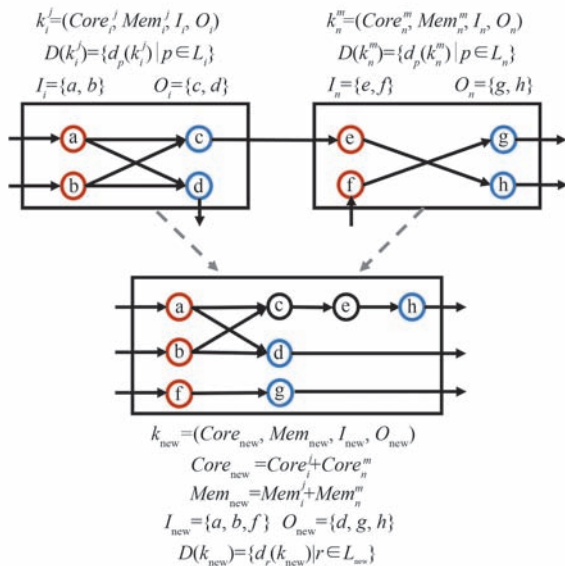


图6 基本数据结构拼接过程示意图

在时延维度上,拼接计算操作对应于可达子路径集合的扩展,其时延更新为路径端到端时延的逐段累积,如公式(1)所示。设 L_i 和 L_n 分别为容器 C_i

和 C_n 的潜在子路径集合,则拼接后结构的潜在子路径集合 L_{new} 由四类路径构成:(1)完全位于容器 C_i 内部的子路径;(2)完全位于容器 C_n 内部的子路径;(3)由 C_i 内子路径与 C_n 内子路径经跨容器依赖边拼接形成的跨容器子路径;(4)入口-出口不可达路径。基于系统模型中端到端时延公式(1),即路径时延为子路径最坏响应时间与子路径入口节点作业释放周期之和,拼接后结构的时延向量结构值 $D(k_{\text{new}}) = \{d_r(k_{\text{new}}) \mid r \in L_{\text{new}}\}$ 更新规则可统一表示为公式(9)。

$$d_r(k_{\text{new}}) = \begin{cases} d_p(k_i^j) + d_q(k_n^m) + T_b & r \text{ 由 } p \in L_i \text{ 和 } q \in L_n \text{ 跨容器拼接组成} \\ d_p(k_i^j) & r = p \in L_i \text{ 且未经过 } C_n \\ d_q(k_n^m) & r = q \in L_n \text{ 且未经过 } C_i \\ +\infty & r \text{ 不经过 } C_i \text{ 与 } C_n \end{cases} \quad (9)$$

其中,跨容器拼接要求存在 $p = (v_{\text{in}}, v_a) \in L_i$, $q = (v_b, v_{\text{out}}) \in L_n$, 并且 $(v_a, v_b) \in E$; T_b 表示容器 C_n 中子路径 q 的入口节点 v_b 的周期,对应子路径 q 最大等待读取时延;若子路径未经过任意一个容器,则对应分量为 $+\infty$ 。该定义保证 $d_r(k_{\text{new}})$ 中任一分量均不小于系统模型下对应路径的实际最坏响应时间,因此拼接计算在结构组合过程中始终保持对端到端时延的保守估计。

CCOA算法流程。在明确拼接规则后,算法整体流程可分为状态初始化、拼接扩展及剪枝优化三个阶段,通过固定流程实现从初始结构到最优配置组合的构建。CCOA算法流程如图7所示。

在算法初始阶段,将所有容器的每种配置均抽象为独立的基础数据结构 k_i^j , 并关联子路径向量,形成初始结构集合,如算法1第1行所示。算法首先选择起始容器 C_{start} 并将其所有状态结构 K_{start} 作为起始节点,通过逐步执行拼接操作扩展子路径,直至形成覆盖入口节点集合到出口节点集合的完整解,如算法1第2~10行所示。CCOA算法整体流程如图7所示。

在迭代拼接扩展阶段,每次从候选集合中依次取出候选结构并尝试与所有可拼接的结构进行拼接(算法1第4~7行),并进行约束检查。在拼接 $k_i^j = (Core_i^j, Mem_i^j, I_i, O_i)$ 与 $k_n^m = (Core_n^m, Mem_n^m, I_n, O_n)$ 的过程中,除连通性检查外,还需检查拼接后的资源消耗是否满足约束条件,即总资源消耗不超过系统上限,如公式(10)所示(算法1第6行)。

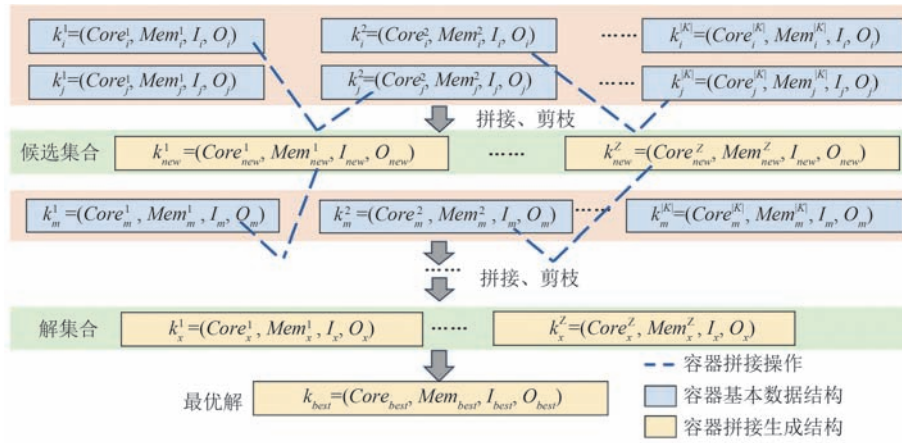


图7 CCOA算法流程示意图

$$\begin{cases} Core_i^j + Core_m^m \leq Core_{\max} \\ Mem_i^j + Mem_m^m \leq Mem_{\max} \end{cases} \quad (10)$$

此外,还需检查拼接后子路径的实时性约束。由于同一子路径可能被多条端到端路径共享,且这些路径具有不同的截止期,因此在拼接时需确保该子路径的累计时延在所有经过它的路径上均不违反实时性约束。设子路径 r 被部分关键路径集合 $P_{sub} \subseteq P_{crit}$ 共享,则拼接后的累计时延 $d_r(k_{new})$ 必须满足 P_{sub} 中所有端到端路径 l 的截止期约束 Ddl_l ,实时性约束可表示为公式(11) (算法1第8行)。

$$d_r(k_{new}) \leq \min \{ Ddl_l \mid l \in P_{sub} \} \quad (11)$$

若上述约束均满足,则允许拼接并生成新状态 k_{new} ,将其加入临时集合(算法1第9~10行),并对临时集合进行优化剪枝。随着拼接的推进,候选结构数量可能呈指数级增长。为避免状态空间过大,算法在临时集合中进行剪枝筛选(算法1第11行),基于启发式评价函数为每个保留的结构计算得分,并按得分对候选结构排序。为了控制搜索空间,仅保留排名靠前的 Z 个结构,形成下一轮迭代的候选集合(算法1第12行)。

算法1. 链式组合优化算法

输入:任务图 G ,结构集合 K 及向量集合 $D(K)$,资源上

限 $Core_{\max}, Mem_{\max}$,路径集合 P_{crit} 及截止期 Ddl_l

输出: 最小资源开销的最优容器配置组合 K_{opt}

1. $Candidate \leftarrow K_{start}, K^* \leftarrow \emptyset$
2. WHILE $Candidate \neq \emptyset$ DO
3. $Temp \leftarrow \emptyset$
4. FOR $k \in Candidate$ DO
5. FOR $k_{next} \in$ 可拼接结构集合 DO
6. IF $Update(Core, Mem) \leq Core_{\max}, m \leq Core_{\max}, Mem_{\max}$ THEN
7. $d_r(k_{new}) \leftarrow delay_merge(k, k_{next})$

8. IF 子路径 p 满足 $Ddl_l, l \in P_{sub}$ THEN
9. $k_{new} \leftarrow combine(k, k_{next})$
10. $Temp \leftarrow Temp \cup \{k_{new}\}$
11. $Optimize(Temp)$
12. $Candidate \leftarrow TopZ(Temp)$
13. FOR $k^* \in Temp$ DO
14. IF $IsFullCoverage(k^*)$ THEN
15. $K^* \leftarrow K^* \cup \{k^*\}$
16. $K_{opt} \leftarrow argmax_{k \in K^*} Judge(k)$
17. RETURN K_{opt}

在不断拼接扩展与剪枝优化阶段后,算法逐步生成了覆盖系统全局入口到出口的候选结构集合,为保证获得完整的端到端解,需要定义算法的终止条件。具体来说,当某一结构 k^* 的入口集合覆盖系统全局入口、出口集合覆盖系统全局出口时,说明该结构已形成完整的端到端可达解,将其加入解集合。算法继续迭代,直至候选集合为空,最终得到候选解集合 K^* (算法1第13~15行)。

在得到候选解集合 K^* 后,需要在多路径实时约束下筛选出全局最优方案,引入基于时延安全裕度的评估函数选出最优方案,实现多路径实时约束下的全局优化(算法1第16行)。具体来说,首先对路径时延进行归一化处理,定义路径安全裕度如公式(12)所示。

$$Sec_l(k) = \frac{Ddl_l - d_l(k)}{Ddl_l} \in [0, 1], k \in K^* \quad (12)$$

安全裕度的归一化处理主要基于两方面考量。首先,不同路径的端到端时延与截止时间往往量级不同,直接使用绝对时延差难以进行跨路径比较,而归一化将所有路径的安全裕度映射至统一的 $[0, 1]$ 区间,从而支持多路径场景下的公平评估与综合权

衡。其次,归一化后的值可直观解释为路径相对于截止期的安全余量,数值越大表明路径对时延波动的容忍能力越强,从系统设计角度体现了时序行为的鲁棒性。基于此,为每条路径 l 赋予相对权重 w_l ,构建加权安全裕度指标作为全局评价准则如下。

$$Judge(k) = \sum_l w_l \cdot Sec_l(k), k \in K^* \quad (13)$$

$Judge(k)$ 是各路径安全裕度进行加权求和构造候选解的评估指标。其中,路径权重 w_l 用于刻画不

同路径在系统中的相对重要性或关键程度。该指标不仅反映了候选解在全局范围内对时延约束的满足程度,同时也能够优先保障关键路径的实时性能,从而避免仅通过局部最优或单一路径裕度做出次优选择。

为直观说明拼接过程中不同容器配置下潜在子路径时延的构成,图8以三个容器为例,展示了在不同配置下容器内部潜在子路径及其对应时延向量,并示意了链式拼接过程中容器结构与时延状态的逐步扩展与更新。

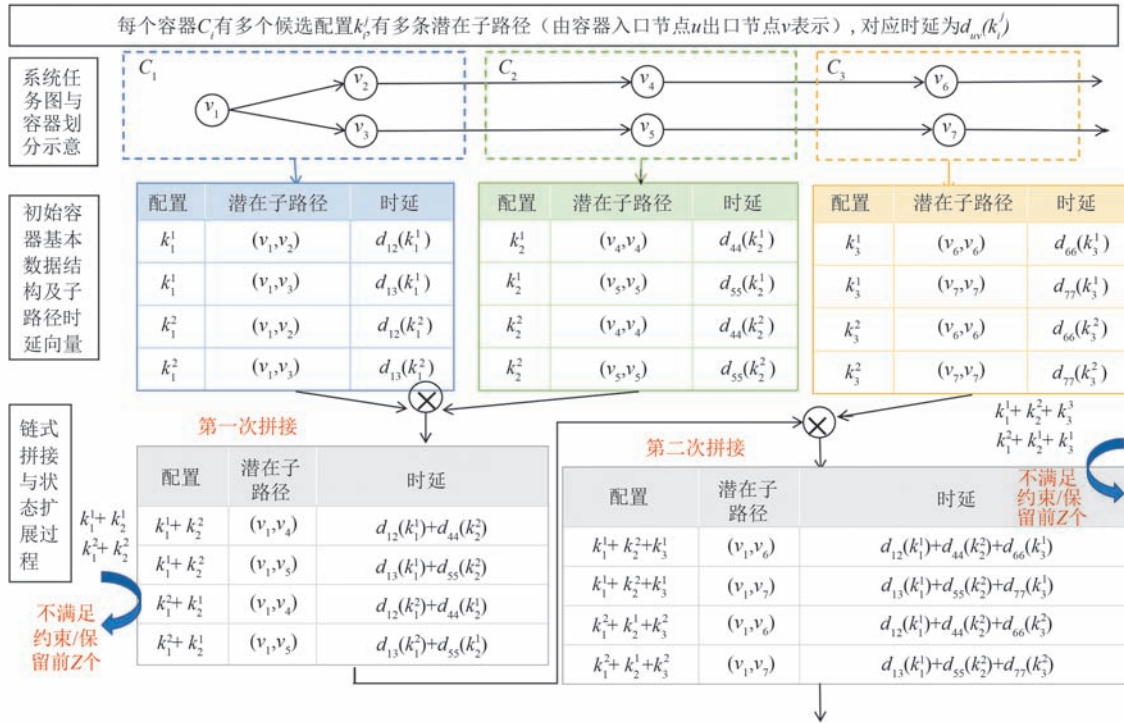


图8 容器结构与子路径时延的拼接示例

优化策略。在多容器链式组合优化配置的拼接过程中,为避免状态空间过大和计算复杂度过高,本文引入优化策略,从不同角度对候选解进行剪枝与筛选,以保证搜索过程满足实时性与资源约束。在筛选候选解时,先将临时集合按入口节点集合与出口节点集合进行分组,对于每组 $((I_k, O_k)$ 相同),采用多目标Pareto支配原则^[38]进行剪枝,将支配关系定义在多维属性向量上如公式(14)所示,将支配关系形式化定义为公式(15)。

$$Attr(k) = (Core(k), Mem(k), D(k)) \quad (14)$$

$$k_1 \text{ 支配 } k_2 \Leftrightarrow \forall \dim, Attr_{\dim}(k_1) \leq Attr_{\dim}(k_2) \quad (15)$$

$$\wedge \exists \dim', Attr_{\dim'}(k_1) < Attr_{\dim'}(k_2)$$

其中, $\dim$ 表示向量的每一维,包括CPU核心、内存及子路径时延向量。 $\forall \dim$ 表示 k_1 在所有维度

上都不高于 k_2 , $\exists \dim'$ 表示 k_1 至少存在某一维度严格优于 k_2 。具体来说,如果结构 k_1 在资源消耗 $(Core_1, Mem_1)$ 和时延向量 $D(k_1)$ 维度上均不高于 k_2 ,且至少在一维上严格低于 k_2 ,则支配解 k_1 支配被支配解 k_2 ,此时可以直接舍弃 k_2 。若两个解(如 k_1 与 k_3)互不支配,即在某些维度上 k_1 优于 k_3 而其他维度上 k_3 优于 k_1 ,则两者均为非支配解。非支配解集合构成Pareto前沿,需保留这些解以维护候选解的多样性。以CPU核心与内存消耗(使用越少越优)这两维度为例 $Attr(k) = (Core(k), Mem(k))$,支配剪枝示意图如图9所示。其中蓝色区域为结构 k_3 的支配区域,在此区域内的所有解(k_4 和 k_5)在CPU核心和内存维度上的消耗均不优于 k_3 ,故被剪枝舍弃; k_2 被 k_1 支

配,同样被剪枝舍弃。最终 k_1 和 k_3 作为非支配解,共同构成 Pareto 前沿并被保留。

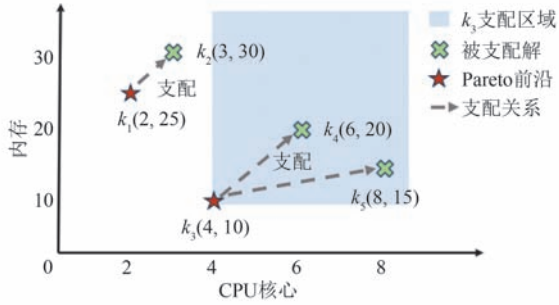


图9 Pareto多目标支配剪枝示意图

对于互不支配的候选解,算法通过启发式评价函数 $Score()$ 对其进行评分筛选,以在多维资源约束和实时性要求下快速收敛到高质量的解。评价函数综合考虑资源使用情况与时延安全程度,以满足系统资源需求与实时安全性。如前所述, α 和 β 反映系统资源的紧迫程度,在此基础上 Z 的选取规则如下:首先,量化配置 k_{new} 的资源开销如下。

$$Cost(k_{new}) = \alpha \cdot Core_{new} + \beta \cdot Mem_{new} \quad (16)$$

为比较不同配置资源使用情况,定义资源利用评分如公式(17)所示。

$$Res(k_{new}) = 1 - \frac{Cost(k_{new})}{\alpha \cdot Core_{max} + \beta \cdot Mem_{max}} \quad (17)$$

该指标取值范围为 $[0, 1]$,其值越大表示资源占用越小,系统在资源开销方面的表现越优。其次,由于拼接过程中子路径归属的端到端路径未知,导致路径截止期与权重不可知,因此引入时延估计函数 $EstDelay()$,以局部信息为基础评估候选结构的时延性能。 $EstDelay(k)$ 反映了入口节点到出口节点可能路径时延情况,其基于结构 k_{new} 的向量 $D(k_{new}) = \{d_p(k_{new}) | p \in L_{new}\}$ 设计,定义如下。

$$EstDelay(k_{new}) = \frac{\sum_{p \in L_{new}} d_p(k_{new})}{|L_{new}|} \quad (d_p(k_{new}) \neq +\infty) \quad (18)$$

在此基础上定义启发式评价函数 $Score(k_{new})$ 如下。

$$Score(k_{new}) = (1 - \gamma) \cdot Res(k_{new}) + \gamma \cdot \frac{1}{1 + EstDelay(k_{new})} \quad (19)$$

其中,权衡参数 $\gamma \in [0, 1]$ 用于在资源利用效率与时延安全性之间进行灵活调节。该设计将两目标统

一至同一评价框架中:当 γ 取值较大时,算法优先保障时延安全裕度,倾向选择干扰敏感度低、资源更充裕的保守安全配置;当 γ 取值较小时,算法转向资源效率优先,逼近资源消耗最小但时延裕量紧张的高效配置。通过调节 γ ,算法能够适应不同的系统约束与应用需求,从而在迭代过程中引导搜索方向,逐步逼近全局最优。最终,算法根据 $Score$ 对候选结构排序,保留前 Z 个最优结构进入下一轮迭代。

针对入口节点集合和出口节点集合不同的情况,采用上界剪枝策略进一步减少无效搜索^[39]。对于每一个候选结构 $k_{candidate}$,先计算其累计资源消耗与当前拼接产生的端到端时延向量 $D(k_{candidate})$ 。同样用多维属性向量 $Attr(k)$ 表示结构状态,在搜索过程中,算法维护一个当前最优结构 k_{best} 的属性上界向量 $(Core(k_{best}), Mem(k_{best}), D(k_{best}))$ 。若 $k_{candidate}$ 的CPU核心与内存消耗以及时延均不优于当前最优解 k_{best} ,则 $k_{candidate}$ 后续拼接无法生成更优解。此时提前舍弃结构 $k_{candidate}$,以避免无效搜索。同样以CPU核心与内存资源维度为例,最优性剪枝示意图如图10所示。橘色区域为剪枝区域,在这个范围内的解(k_4)在两个维度上均超过了最优解,无法产生更优的解,故将此区域的解进行剪枝。黄色区域为潜在更优区域,在此区域内的解(k_1, k_2 和 k_3)至少在一个维度优于 k_{best} ,将这些解保留进行后续拼接以获取更优解。

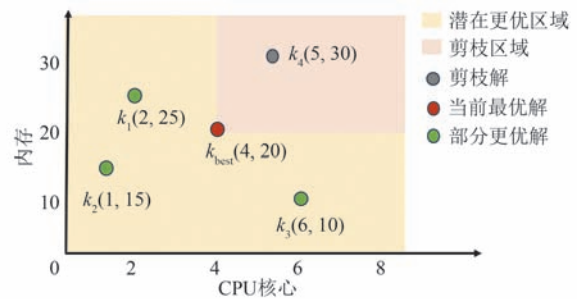


图10 最优性上界剪枝示意图

算法性能分析。下面对CCOA算法的时间复杂度进行理论分析。该算法所求解的容器资源配置优化问题,本质上是在满足多维资源约束与多重时序要求下,为每个容器从候选配置集合中选择一个配置,以最小化系统的总资源消耗。CCOA算法在多容器、多资源、多路径约束下的容器资源配置优化问题中,能够在可接受的计算开销内获得高质量近似解。其整体最坏情况时间复杂度为

$$O(Z^2 \cdot N \cdot |K|^2 |L|).$$

其中, N 为容器系统数量, $|K|$ 为容器最大候选配置数, $|L|$ 为系统潜在端到端路径数量最大值, Z 为每轮迭代保留的候选解数量。

CCOA 算法时间复杂度主要取决于主循环的迭代次数以及每次迭代中候选解生成、剪枝与筛选的操作开销。初始化阶段生成 $N \cdot |K|$ 个基础结构, 每个结构关联一个维度为 $|L|$ 的时延向量。随后, 算法通过逐步拼接容器结构, 最多需要进行 $N-1$ 轮迭代以覆盖所有端到端路径。在第 i 次迭代中 ($1 \leq i \leq N-1$), 当前候选集的大小至多为 Z 。在扩展阶段, 每个候选解最多与下一个容器的 $|K|$ 种配置进行拼接, 生成的新的新候选解数量上界为 $P = O(Z \cdot |K|)$ 。由于每次拼接需要更新一个 $|L|$ 维的时延向量, 因此该阶段时间复杂度为 $O(Z \cdot |K| \cdot |L|)$ 。在 *Pareto* 剪枝阶段, 需对 P 个新解进行多目标非支配排序, 其属性维度为 $M = 2 + |L|$ (包括 CPU、内存与时延向量分量)。在最坏情况下, 该过程时间复杂度为 $O(M \cdot (Z|K|)^2)$, 即 $O((2 + |L|) \cdot (Z|K|)^2)$ 。在启发筛选阶段, 算法对经剪枝保留的解进行启发式评分, 并选取前 Z 个解作为下一轮迭代的候选集合。该阶段的复杂度主要来自评分计算与排序操作, 分别为 $O(Z \cdot |K| \cdot |L|)$ 与 $O(Z \cdot |K| \cdot \log(Z|K|))$ 。综上所述, 每轮迭代的主导复杂度为 *Pareto* 剪枝阶段的 $O((2 + |L|) \cdot (Z|K|)^2)$ 。考虑到算法整个过程最多需要执行 $N-1$ 轮迭代, 算法的整体最坏情况时间复杂度为 $O(Z^2 \cdot N \cdot |K|^2 |L|)$ 。

通过对复杂度理论的分析, 可以看出, CCOA 算法通过限制候选集与启发式筛选, 有效控制了指数级增长的搜索空间, 从而在保证解质量的同时维持了可接受的计算开销。在实际应用中, 尤其在典型自动驾驶系统中, 任务流的拓扑结构通常是稳定且有限的, 路径数量 $|L|$ 可预测且可控。总体而言, CCOA 算法在实际部署中具备良好的可行性, 能够在多容器、多资源、多路径约束下提供高质量解。

可扩展性分析。尽管 CCOA 算法在最坏情况下的时间复杂度与容器数量 N 和路径数量 $|L|$ 相关, 但其在典型自动驾驶系统中具备良好的可扩展性。首先, 实际系统规模受到任务结构的显著约束。在当前主流开源自动驾驶框架 Autoware 的典型部署

中, 核心容器数量通常较少, 关键任务路径也较为有限, 自动驾驶任务链通常由少数关键路径主导, 路径数量 $|L|$ 远小于理论上的组合爆炸上限, 且容器间耦合多呈现链式或树状, 而非全连接网状, 降低了搜索复杂度。此外, 算法可通过调节参数在解的质量与计算开销之间灵活权衡。实验表明, CCOA 算法能够在 Autoware 真实部署规模下获得最优解, 满足当前实际部署的工程需求。

对于未来可能出现的超大规模系统, 可通过分层-分治策略进行优化。先基于功能域划分进行集群级资源配额, 再在各子域内并行运行 CCOA 算法。此外, 也可结合元启发式算法进行近似优化, 或构建历史配置库支持增量学习, 提升求解效率。

7 实验结果

为验证本文提出的容器配置优化算法 (CCOA) 在自动驾驶 SOAFEE 框架下的适用性与有效性, 实验基于 Autoware Reference System 搭建多功能模块的多容器测试环境。该系统包含 6 个功能模块, 每个模块封装为独立容器, 模块间的功能依赖关系构成有向任务图, 此架构与当前主流 Autoware 自动驾驶系统的容器化部署方式一致。实验硬件采用 ARM 架构的 AVA 开发板, 软件环境为 SOAFEE 架构的 EWAOL (Edge Workload Abstraction and Orchestration Layer) 参考实现^[40]。AVA 开发板作为通用多核 CPU 参考平台, 与实际车载异构多核平台在硬件结构等方面存在差异, 因此实验结果主要反映了方法在通用 ARM 环境下的性能。本文提出的容器配置优化方法基于系统测试建立的容器资源配置与端到端时延映射关系, 其优化框架不依赖特定硬件架构, 具有较好的跨平台可迁移性。实验中容器通过 Linux Cgroups 机制实现容器间的资源限制与隔离, 且禁用 Hypervisor 层以避免额外性能干扰。

(1) 配置集生成。为保证实验的公平性与可重复性, 在实验前为每个容器生成多样化的候选配置集。综合考虑 Autoware Reference System 的实际运行特性, 兼顾资源紧迫性与任务实时性要求, 先基于系统空闲资源阈值确定每个容器 CPU 与内存的上下限范围, 再在该范围内生成多组候选配置, 既覆盖低资源占用的轻量化方案, 也包含高性能配置, 以满足不同任务负载下的实时性需求。通过分析容器内

部任务子图,确定入口节点集合与出口节点集合,形成对应的基本结构数据集合。在每组候选配置下测试容器内部各子路径的端到端时延,得到不同配置下的潜在路径时延集合。结合系统资源上限及实际运行规律,可观察到当CPU与内存分配超过一定阈值时,时延改善呈现边际递减趋势。基于这一特性,将CPU核心分配范围设置为1至3核,内存分配范围设置为0.5至5 GB,并通过组合生成不同配置方案,以在资源利用效率与任务实时性间取得平衡。

(2)关键路径。根据Autoware Reference System的实际任务流,在功能模块任务图中定义了5条关键路径,每条路径覆盖从传感器数据采集、点云处理、定位与规划到车辆控制的完整执行流程。这一路径数量的选择能够有效覆盖Autoware运行中的核心需求,并与实际部署场景的规模相匹配。这些路径包括前、后激光雷达感知路径、融合定位路径、全局规划路径以及车辆控制路径,能够全面表征系统的主要任务流。为贴近真实应用场景,依据任务的功能安全等级和关键性为每条路径设置了各自的时延约束与相对权重。例如,直接关系到安全的车辆控制路径被赋予了更严格的时延约束和更高的权重。

(3)基准算法。为全面验证所提方法的有效性,选取贪心算法、混合整数线性规划(MILP)以及Kubernetes下的垂直Pod自适应调整(Vertical Pod Autoscaler, VPA)算法作为对比基准。贪心算法针对每个容器单独选择最优配置,通过贪心评价函数对容器的各候选配置进行排序,从而为系统选取每个容器的最优组合,形成一组整体容器配置方案。评价函数综合衡量该配置对各条子路径时延的改善效果以及所需CPU和内存资源,生成综合评价价值。评价价值越高,表明该配置在降低各条子路径时延的同时,更高效地利用了系统资源,即在减少时延的同时消耗更少的CPU与内存资源。MILP将容器配置优化问题形式化为多维资源约束下的优化问题,目标是在满足端到端时延与系统资源约束的前提下,最小化总体资源消耗,约束条件包括CPU与内存上限以及多关键路径截止期约束。VPA算法则通过分析容器的历史资源使用情况,为每个容器生成推荐的CPU和内存配置,包括目标推荐值、下界和上界,旨在为容器提供合理的资源配置建议,以平衡性能与资源效率。相较之下,本文提出的CCOA算法从全局出发,综合考虑多关键路径间的资源争用与时延要求,旨在满足所有路径实时性约束的同

时实现系统总资源消耗的最小化,力求在多关键路径场景下实现整体性能的最优平衡。

(4)性能指标。实验在不同算法下使用相同的候选配置集,通过算法选取一组容器配置方案并在实际系统中部署执行,性能评价指标主要包括路径端到端时延满足率与资源分配率。路径端到端时延满足率反映各关键路径在给定资源配置下的实时性表现,时延超过截止期则认定该路径实时性失效,其可作为自动驾驶系统功能安全流程中实时性验证的重要参考指标;资源分配率评估容器配置对CPU和内存的占用情况,其值越小说明系统资源利用越高,反之则表示资源占用较大,可能存在浪费的风险。通过综合分析端到端时延与资源分配率,可以全面评估不同算法在多路径、多容器场景下的性能表现。实验将不同算法下求得的解方案实际部署运行,记录每条关键路径的实际时延,计算对应的路径端到端时延满足率与资源分配率,以验证算法在保证资源约束的同时是否能够保障多路径端到端时延约束,并在多路径协调下实现全局优化。为保证实验结果的稳健性,每组配置方案均重复多次运行,以消除偶然波动对性能指标的影响。

(5)CCOA算法。为验证CCOA算法在自动驾驶SOAFEE框架下的有效性,本实验围绕多路径端到端时延约束评估算法在多容器场景下的性能。实验目标是在给定资源上限约束、多路径截止期约束及路径权重的前提下,从候选配置集中选取高质量可行解,以在满足系统实时性和资源限制的同时实现整体资源利用最小化。算法在搜索过程中引入启发式评价函数进行筛选,其中CPU与内存的权重系数 α 和 β 均为0.5以平衡两类资源的贡献;参数 γ 用于调控资源利用率与端到端时延之间的权衡,其值越大表示越注重时延安全性;参数 Z 表示每轮迭代保留的候选结构数量,直接影响搜索空间规模和计算时间。

首先,实验测试了不同 γ 和 Z 的组合下的算法性能,由于自动驾驶系统更关注端到端时延保障, γ 取值范围设置在相对较高的区间。为控制搜索空间与成本, Z 值最大不得超过配置集大小的一半。实验评价指标包括求解时间与路径截止期满足率,求解时间指算法在多次运行下的平均耗时(单位:秒)。实验结果如表3所示,其中“-”表示该参数组合下未找到可行解。结果如图11所示。

从实验结果来看,随着 Z 值的增加,搜索空间扩

表 3 CCOA 算法求解时间对比					
γ	$Z=3$	$Z=5$	$Z=8$	$Z=10$	$Z=15$
0.9	0.886 s	4.719 s	24.365 s	49.283 s	166.71 s
0.8	-	-	20.066 s	43.921 s	148.65 s
0.7	-	-	17.125 s	38.735 s	148.62 s
0.6	-	-	17.555 s	38.947 s	135.61 s
0.5	-	-	17.415 s	38.731 s	136.26 s

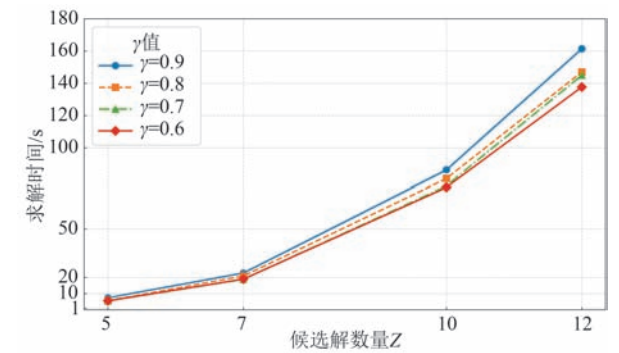


图 11 不同 γ 和 Z 下算法求解时间对比

大,算法保留更多候选结构,导致求解时间显著增长。这表明 Z 值直接决定搜索空间规模与计算开销之间的权衡。在低 Z 值,低 γ 值下无法找到可行解,说明算法倾向选择资源消耗较低的候选解,而未能严格满足端到端时延约束。这一现象说明,在小搜索空间和低 γ 值条件下,保证解的存在性需要增加候选解数量或适当放宽路径时延约束。另一方面, γ 值在评价函数中调节时延与资源利用率的权重。较高 γ 值使算法更注重保障端到端时延,因此能够在较小 Z 值下快速收敛并得到可行解;随着 γ 逐渐减小,系统对时延的紧迫性降低,算法更倾向于资源最小化,可能无法满足时延约束。求解时间随着 γ 值减小略有下降并趋于稳定。尽管最大求解时间达到了166.7 s,但由于优化过程为离线容器配置阶段,该耗时在可接受范围内,并不影响系统的实时运行性能。

其次,算法在不同 γ 和 Z 的组合下生成多种配置方案,以对比不同参数组合下的方案。结果如表4所示,其中 C_i 表示第 i 个容器, C 表示CPU资源(核数), M 表示内存资源(GB)。实验结果表明, Z 值对方案稳定性有明显影响,如图12所示。

当 Z 值较大时(如 $Z=10、15$),算法保留的候选结构数量充足,搜索空间几乎覆盖所有有效解。因此,无论 γ 如何变化,最终求得的配置方案均保持一致,说明在候选结构充足的情况下,算法具有较高的全局稳定性。当 Z 值为中等水平(如 $Z=8$)时,候选

表 4 CCOA 算法求解所得配置方案		
序号	配置方案	对应配置组合
1	C1: C=3.0, M=1.0	$\gamma=0.9, Z=3$
	C2: C=2.0, M=0.5	
	C3: C=2.0, M=0.5	
	C4: C=2.0, M=0.5	
	C5: C=2.0, M=0.5	
	C6: C=1.0, M=2.5	
2	C1: C=3.0, M=1.0	$\gamma=0.9, Z=5$
	C2: C=2.0, M=1.0	
	C3: C=2.0, M=1.0	
	C4: C=2.0, M=0.5	
	C5: C=2.0, M=1.0	
	C6: C=1.0, M=2.5	
3	C1: C=3.0, M=1.0	$\gamma=0.9, Z=8$
	C2: C=2.0, M=1.0	$\gamma=0.9, Z=10$
	C3: C=1.0, M=0.5	$\gamma=0.8, Z=8$
	C4: C=2.0, M=1.5	$\gamma=0.8, Z=10$
	C5: C=3.0, M=0.5	$\gamma=0.7, Z=10$
	C6: C=1.0, M=2.5	$\gamma=0.6, Z=10$
4	C1: C=3.0, M=0.5	$\gamma=0.7, Z=8$
	C2: C=2.0, M=1.0	$\gamma=0.6, Z=8$
	C3: C=1.0, M=0.5	$\gamma=0.5, Z=8$
	C4: C=2.0, M=1.5	
	C5: C=3.0, M=0.5	
	C6: C=1.0, M=2.5	
5	C1: C=3.0, M=1.0	$\gamma=0.9, Z=15$
	C2: C=2.0, M=1.0	$\gamma=0.8, Z=15$
	C3: C=1.0, M=0.5	$\gamma=0.7, Z=15$
	C4: C=2.0, M=2.0	$\gamma=0.6, Z=15$
	C5: C=3.0, M=0.5	$\gamma=0.5, Z=15$
	C6: C=1.0, M=2.5	

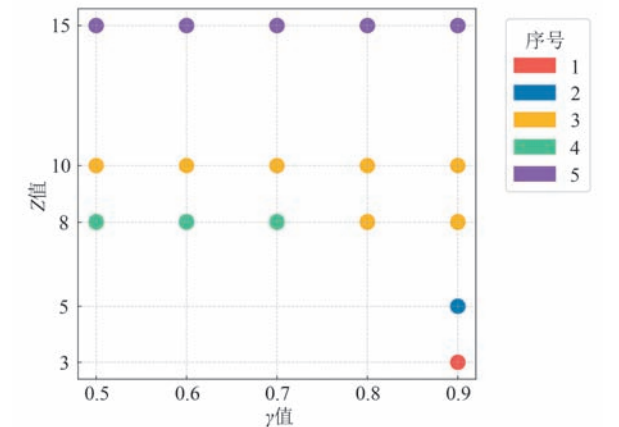


图 12 不同 γ 和 Z 下的配置方案

结构数量相对有限, γ 的变化会对最终求得的方案产生影响,但整体仍保持一定稳定性。例如,在 $Z=8$ 时, $\gamma=0.9、0.8$ 的配置方案一致,而 $\gamma=$

0.7、0.6、0.5的方案也一致,表明算法在不同 γ 下会形成稳定的方案组合。在较小的 Z 值下,候选结构有限,低 γ 值可能无法找到可行解,这表明算法在小搜索空间中对 γ 较为敏感。总体来看, γ 较大时,算法对端到端时延更加敏感,能够在不同 Z 值下快速收敛并生成高质量可行解。

为进一步验证算法在实际运行环境中的有效性,实验将表4中的配置方案(序号如上)部署到系统中,测试各关键路径端到端时延是否满足截止期,以评估算法在资源约束下的端到端时延安全性。各方案下实际运行结果如图13所示,截止期满足率如表5所示。

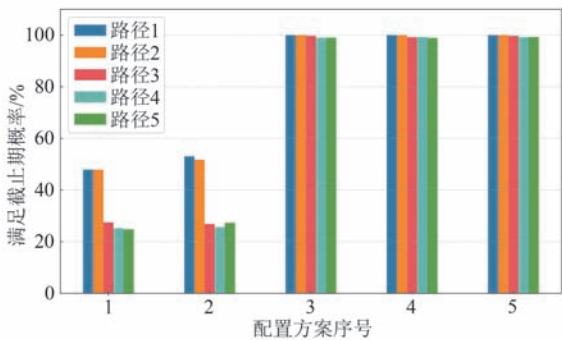


图13 不同配置方案下各路径满足截止期概率

从系统实际运行结果可看出,随着 Z 值的增加,配置方案在各路径满足截止期的概率显著提高。当 $Z \geq 8$ 时,方案在所有路径的截止期满足率接近或达到100%,充分验证了CCOA算法在保证端到端时延约束方面的有效性。相比之下,较小 Z 值的方案在各路径上截止期满足率明显下降,表明候选结构数量不足时,算法在实际运行中可能无法完全满足时延约束。在同一配置下,由于各路径权重递减,系统优先保证权重较高的前两条关键路径,因此各路径的截止期满足率基本呈现递减趋势。

综合考虑求解时间与截止期满足率,实际运行时优先选择在保证各关键路径截止期满足率接近100%的前提下,求解时间较短且候选结构充足的

配置方案。具体而言, $Z=8$ 的两种方案均能在满足端到端时延约束的同时保持适中的求解时间,是高质量配置方案的理想选择。在 $Z=8$ 的情况下,高 γ 值方案的加权资源分配率为86.36%。低 γ 值方案下加权资源分配率为85.1%。两者在资源约束与实时性性能上基本一致,进一步验证了算法在不同参数组合下的稳定性和适用性。

(6)对比实验。为进一步对比验证所提CCOA算法的有效性,本文以贪心算法、混合整数线性规划(MILP)和Kubernetes下的VPA算法作为对比基准。贪心算法通过贪心评价函数对每个容器的候选配置进行排序,逐个选择最优配置,形成整体容器配置方案。MILP方法通过建立整数线性约束模型,将容器资源分配、路径截止期及系统资源上限统一建模,使用PuLP调用求解器进行全局优化求解。Kubernetes下的VPA算法通过分析容器的历史资源使用模式,为每个容器生成推荐的CPU和内存配置,旨在提供合理的资源配置建议以平衡性能与资源效率。贪心算法和MILP以最小化系统总资源消耗为核心目标,而VPA则侧重于基于历史使用数据的资源生成合理化目标值推荐;CCOA算法同样以最小化系统总资源消耗为目标,但从全局视角综合考虑多关键路径间的资源争用与时延耦合关系。上述算法均用于分析不同资源分配方案对端到端时延的影响。

为确保在所有参数组合下均能生成可行解且求解时间合理,同时考虑实时系统对时延的敏感性,CCOA算法选用 $Z=8$ 且 $\gamma=0.9$ 方案(方案序号为3)作为对比对象,其他算法中有关 γ 值统一设置为0.9以保持一致性。候选配置集在四种算法上保持一致。在求解时间方面,贪心算法求解时间在0.1秒左右,MILP求解时间约0.4秒,均能够快速生成配置方案,而VPA算法作为推荐式方法,其推荐结果依赖充足的离线历史资源使用数据,算法自身求解时间可忽略不计。

四种算法生成的容器配置方案如表6所示,为

表5 不同方案下载止期满足率

方案序号	路径1	路径2	路径3	路径4	路径5	截止期满足率均值
1	47.92%	47.90%	27.46%	25.20%	24.78%	34.65%
2	53.01%	51.73%	26.86%	25.62%	27.30%	36.90%
3	100%	100%	99.72%	98.97%	98.96%	99.53%
4	100%	100%	99.17%	99.22%	98.87%	99.25%
5	100%	100%	99.71%	99.16%	99.23%	99.62%

进一步分析其性能,将这些方案部署到实际系统中,并记录关键路径的截止期满足率和系统资源分配率,如图14所示。

表6 四种算法求解方案

算法	容器配置方案
CCOA-Z(8)	C1: C=3, M=1.0
	C2: C=2, M=1.0
	C3: C=1, M=0.5
	C4: C=2, M=1.5
	C5: C=3, M=0.5
	C6: C=1, M=2.5
贪心	C1: C=2, M=0.5
	C2: C=2, M=0.5
	C3: C=2, M=0.5
	C4: C=2, M=0.5
	C5: C=2, M=0.5
	C6: C=1, M=0.5
MILP	C1: C=2, M=1.5
	C2: C=2, M=1.0
	C3: C=1, M=0.5
	C4: C=2, M=1.0
	C5: C=2, M=1.0
	C6: C=1, M=1.0
VPA	C1: C=2.3, M=0.5
	C2: C=2, M=1.0
	C3: C=1.3, M=0.5
	C4: C=2, M=1.0
	C5: C=1.9, M=0.5
	C6: C=0.8, M=0.5

实验结果表明,CCOA算法在五条关键路径上的平均截止期满足率为99.53%,显著优于贪心算法的49.62%、MILP方法的55.5%以及VPA算法的41.39%,这表明CCOA算法能够在严格的多路径约束下高效保障端到端时延要求。从机制上看,CCOA算法在候选解生成过程中考虑多路径时延要求并使用全局启发式评价策略,能够合理分配系统资源,保证整体系统的实时性。尽管CCOA算法的资源分配率相对较高,但仍低于预计分配资源总量,在可接受范围内实现了合理的资源利用。在实时系统中,任务的安全性和时延保障比资源消耗更为关键,因此CCOA算法在资源与实时性之间达到了较优平衡。

相比之下,贪心算法针对单个容器仅追求资源效率选择最优配置,通过局部评价函数对候选配置,形成整体方案。虽然其求解时间仅约0.1秒,但由于缺乏全局优化考虑,导致截止期满足率明显下降,

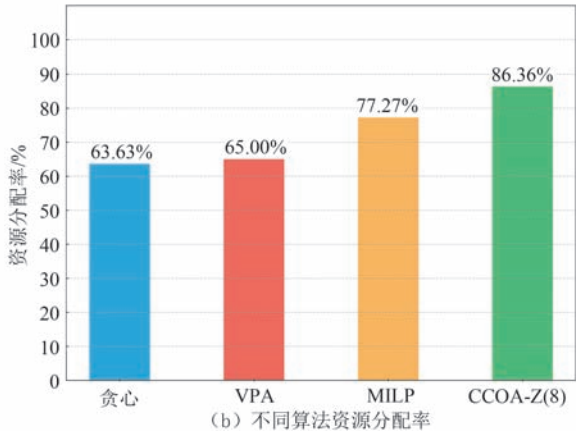
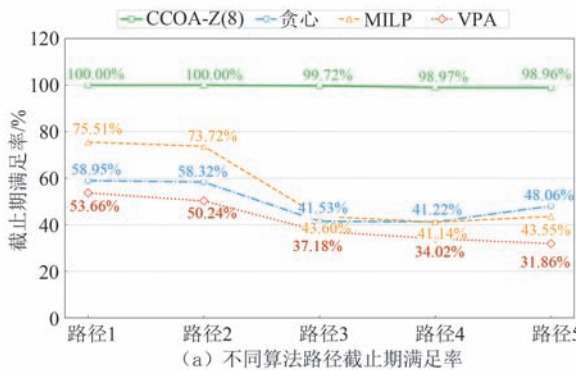


图14 四种算法实验结果对比

仅为49.62%。其资源分配率均值为63.63%。贪心算法消耗资源较少,但实时性保障能力有限。实验结果进一步说明,局部最优解未能考虑全局优化,其在多路径端到端自动驾驶场景下难以满足严格的实时性需求。

实验结果显示,MILP方法在前两条高权重路径上能够较好满足截止期,但在后三条低权重路径上满足率大幅下降;尽管资源分配率略优于CCOA,但在实际运行中却表现出较差的稳定性和安全性。原因在于,MILP模型的目标是最小化系统总资源消耗,时序约束仅作为可行解的边界条件。求解器倾向于找到资源恰好够用的配置方案,因此理论上资源利用率最优,但每条路径时延裕量非常有限。在资源受限的情况下,MILP会优先保障高权重路径,导致低权重路径的时延保障能力不足,且在实际系统中,任何微小的性能抖动或容器间干扰都可能导致任务时延超出截止期。相比之下,CCOA算法通过权重参数 γ 在资源利用率与时延安全裕度之间进行显式权衡。选用高 γ 值时,算法倾向于选择那些时延远低于截止期、但更加安全的配置方案,即便这些方案会消耗更多资源。实验结果表明,尽管CCOA的总体资源利用率略比MILP高,

但 CCOA 对所有路径的时延保障更均衡,系统在实际运行中更稳健。

VPA 算法的平均截止期满足率为 41.39%,资源分配率为 65%。VPA 虽能够基于容器的历史资源使用模式生成资源推荐配置,但其本质上属于单容器独立的资源合理化推荐方法,仅从局部资源利用率角度进行优化,缺乏对多关键路径时延耦合关系、任务依赖关系以及全局端到端时延的感知。尽管 VPA 算法相比于贪心算法为容器分配了较多资源,但其配置决策完全不具备时延感知与全局优化能力。与之不同的是,贪心算法虽同样属于局部优化方法,但其配置选择面向容器时延特征进行排序,在资源分配上仍具备一定的实时性偏向,因此获得了优于 VPA 的截止期满足率。在多路径且端到端时延敏感的自动驾驶场景下,VPA 无法实现全局资源协同优化,难以有效保障系统实时性。

综上所述,CCOA 算法在资源分配量比贪心算法增加 22.73% 的情况下,路径截止期满足率显著提高 49.91%;与 MILP 方法相比,CCOA 在仅增加 9.09% 资源的情况下,提升了 44.03% 的路径截止期满足率;与 VPA 算法相比,将路径截止期满足率提升了约 58.14%。贪心算法仅追求单个容器资源效率,局部最优未考虑全局优化,无法联合优化满足端到端时延的严格实时性需求;MILP 算法的目标是最小化资源,倾向于找到资源恰好够用的配置方案,但动态变化的实际运行环境中任何微小的性能抖动或容器间干扰都可能导致任务时延超出截止期,无法保证配置方案下的可靠性和稳定性;VPA 算法虽基于历史资源使用数据的角度进行资源配置,但本质上是单容器资源感知推荐方法,缺乏系统全局视角与端到端时延感知能力。这表明,CCOA 算法在保证多路径端到端实时性的同时,实现了合理的资源分配,表现出优于贪心和 MILP 方法的整体性能。因此,CCOA 算法适用于面向 SOAFEE 框架下自动驾驶系统的实际部署,能够在保证实时性和资源高效利用之间实现平衡,满足复杂多路径场景下的调度需求。

8 总 结

本文针对 SOAFEE 框架下自动驾驶系统中多容器多任务链的端到端实时性保障与资源优化问题,提出了一种基于 DUT 的容器参数优化方法,并在此基础上设计了链式组合优化算法(CCOA)。该

方法通过实际测试,测得每个容器在不同配置下的实时性能,寻求所有容器配置方案的最优组合,满足系统实时性能,实现了在多维资源约束下的高质量容器配置选择。基于 Autoware 自动驾驶框架对 CCOA 算法进行了系统测试,并与贪心算法及经典 MILP 方法进行对比分析。结果表明,CCOA 算法在端到端路径实时性保障方面表现优异且资源利用合理,显著优于贪心算法、MILP 方法与 VPA 推荐配置方法。这验证了 DUT 方法在 SOAFEE 架构下实现自动驾驶系统多容器多任务链端到端时序保障与资源优化配置的有效性,为实际部署提供了可行方案。

未来的工作将围绕以下几个方向展开,以进一步深化与拓展本研究。首先,将精细化干扰建模并适应不同场景,通过设计面向不同驾驶场景的测试,提取场景特征对应的干扰系数,从而构建更贴合动态环境的多容器干扰模型。其次,优化提升 CCOA 算法的求解效率,探索基于机器学习或元启发式的高效近似算法,以支撑更大规模任务图的实时配置优化。再次,从离线优化扩展至在线自适应场景,构建一个离线配置与在线决策相结合的两阶段优化框架,离线阶段预计算出多套最优的容器配置方案,并构建场景-配置映射库。在线运行时实时判定当前运行状态并结合数据对最匹配的计算配置进行微调,实现开放环境下实时性保障与资源效率的平衡。最后,进一步探索与功能安全流程的结合。例如结合 ISO 26262 功能安全标准,将容器配置优化结果与系统级安全分析方法相结合,在保证关键路径实时性的基础上,为自动驾驶系统的功能安全验证提供支持,实现实时性能优化与功能安全保障的协同设计。

作者贡献声明 陶泓雨、陈钰婷为共同第一作者,东北大学与大连理工大学为共同第一单位。

参 考 文 献

- [1] China Association of Automobile Manufacturers. White Paper on Software-defined Vehicle Industry Ecosystem Innovation (V1.0). <http://www.caam.org.cn>
- [2] Betz T, Wen L, Pan F, et al. A containerized microservice architecture for a ROS 2 autonomous driving software: An end-to-end latency evaluation//Proceedings of the IEEE 30th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA). Taipei, China, 2024: 57-66

- [3] RAND Corporation. Autonomous Vehicles Cannot Be Test-Driven Enough Miles to Demonstrate Their Safety; Alternative Testing Methods Needed. <https://www.rand.org/news/press.html>
- [4] Nitin Dahad. SOAFEE architecture for embedded edge enables software defined cars. <https://www.ednasia.com/soafee-architecture-for-embedded-edge-enables-software-defined-cars/>
- [5] Struhár V, Behnam M, Ashjaei M, et al. Real-time containers: A survey//Proceedings of the 2nd Workshop on Fog Computing and the Internet of Things (Fog-IoT 2020). Schloss Dagstuhl, Germany, 2020: 7:1-7:9
- [6] China Automotive Technology and Research Center. Research Report on In-vehicle Operating System Architecture
- [7] Wilhelm R, Engblom J, Ermedahl A, et al. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 2008, 7(3): 1-53
- [8] Rahman J, Lama P. Predicting the end-to-end tail latency of containerized microservices in the cloud//Proceedings of the IEEE International Conference on Cloud Engineering (IC2E). Prague, Czech Republic, 2019: 200-210
- [9] Qian W, Zhao H, Chen T, et al. Learning Unified System Representations for Microservice Tail Latency Prediction. *arXiv preprint*, arXiv:2508.01635. 2025
- [10] Gan Y, Liang M, Dev S, et al. Sage: Practical & scalable ML-driven performance debugging in microservices//Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. Virtual, 2021: 1-17
- [11] Wu Y W, Zhang Y, Wang T, Wang H M. Development exploration of container technology through Docker containers: A systematic literature review perspective. *Journal of Software*, 2023, 34(12): 5527 - 5551 (in Chinese)
(吴逸文, 张洋, 王涛, 王怀民. 从 Docker 容器看容器技术的发展: 一种系统文献综述的视角. *软件学报*, 2023, 34(12): 5527-5551)
- [12] Kim S, Song J, Lee K, et al. Cros-Rt: Cross-layer priority scheduling for predictable inter-process communication in Ros 2//Proceedings of the 31st IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS). Irvine, USA, 2025: 202-214
- [13] Teper H, Betz T, Gunzel M, et al. End-to-end timing analysis and optimization of multi-executor ROS 2 systems//Proceedings of the 30th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS). Hong Kong, China, 2024: 212-224
- [14] Yano A, Azumi T. Work-in-progress: Multi-deadline DAG scheduling model for autonomous driving systems//Proceedings of the IEEE Real-Time Systems Symposium (RTSS). Taipei, China, 2024: 451-454
- [15] Sun J, Duan K, Li X, et al. Real-time scheduling of autonomous driving system with guaranteed timing correctness//Proceedings of the 29th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS). San Antonio, USA, 2023: 185-197
- [16] Sollfrank M, Loch F, Denteneer S, et al. Evaluating docker for lightweight virtualization of distributed and time-sensitive applications in industrial automation. *IEEE Transactions on Industrial Informatics*, 2020, 17(5): 3566-3576
- [17] De Marchi M, Bombieri N. Orchestration-aware optimization of ROS 2 communication protocols//Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE). Valencia, Spain, 2024: 1-6
- [18] Betz T, Teper H, Ebner D, et al. End-to-end latency optimization for containerized ROS 2 autonomous driving software. *IEEE Access*, 2025, 13: 112654-112672
- [19] Cai L, Qi Y, Wei W, et al. Improving resource usages of containers through auto-tuning container resource parameters. *IEEE Access*, 2019, 7: 108530-108541
- [20] Somashekar G, Tandon K, Kini A, et al. OPPerTune: Post-deployment configuration tuning of services made easy//Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI). Santa Clara, USA, 2024: 1101-1120
- [21] Liu R, Yang P, Lv H, et al. Multi-objective multi-factorial evolutionary algorithm for container placement. *IEEE Transactions on Cloud Computing*, 2021, 11(2): 1430-1445
- [22] Cao C, Li Z. Research on the optimization algorithm of multi-target resource placement in cost-sensitive container. *Intelligent Computer and Applications*, 2020, 10(4): 50-53+56 (in Chinese)
(曹成成, 李志聪. 代价敏感的容器多目标资源放置优化算法. *智能计算机与应用*, 2020, 10(4): 50-53+56)
- [23] Jansen M, Talluri S, Doekemeijer K, et al. Columbo: A reasoning framework for Kubernetes' configuration space//Proceedings of the 16th ACM/SPEC International Conference on Performance Engineering (ICPE). Toronto, Canada, 2025: 1-13
- [24] Chiba T, Nakazawa R, Horii H, et al. Confadvisor: A performance-centric configuration tuning framework for containers on Kubernetes//Proceedings of the IEEE International Conference on Cloud Engineering (IC2E). Prague, Czech Republic, 2019: 168-178
- [25] Liang T, Wang L, Xu Z. Research on intelligent scheduling strategy of container resources based on LSTM and genetic algorithm. *Computer Science and Application*, 2024, 14(12): 132-141 (in Chinese)
(梁拓, 王利众, 许震. 基于 LSTM 和遗传算法的容器资源智能调度策略研究. *计算机科学与应用*, 2024, 14(12): 132-141)
- [26] Molina E, Lazaro O, Sepulcre M, et al. The AUTOWARE framework and requirements for the cognitive digital automation//Proceedings of the 18th IFIP WG 5.5 Working Conference on Virtual Enterprises (PRO-VE 2017). Vicenza, Italy, 2017: 107-117
- [27] Fan H, Zhu F, Liu C, et al. Baidu apollo em motion planner. *arXiv preprint arXiv:1807.08048*, 2018
- [28] Macenski S, Foote T, Gerkey B, et al. Robot operating system 2: Design, architecture, and uses in the wild. *Science Robotics*, 2022, 7(66): eabm6074

- [29] Baidu. Cyber RT Document. <https://cyber-rt.readthedocs.io/en/latest/>
- [30] Behere S, Tömgren M. A functional reference architecture for autonomous driving. *Information and Software Technology*, 2016, 73: 136-150
- [31] The Autoware Foundation. Autoware concepts. <https://autowarefoundation.github.io/autoware-documentation/main/design/autoware-concepts/>
- [32] Al-Batati A S, Koubaa A, Abdelkader M. ROS 2 in a nutshell: A survey. Preprints, 2024-10
- [33] Sobhani H, Choi H, Kim H. Timing analysis and priority-driven enhancements of ROS 2 multi-threaded executors// *Proceedings of the IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. San Antonio, USA, 2023: 106-118
- [34] Yang Y, Azumi T. Exploring real-time executor on ROS 2// *Proceedings of the IEEE International Conference on Embedded Software and Systems (ICESS)*. Las Vegas, USA, 2020: 1-8
- [35] Lee S, Park H S, Chae J, et al. Probabilistic Latency Analysis of the Data Distribution Service in ROS 2. arXiv preprint arXiv: 2508.10413, 2025
- [36] Dürr M, Brüggem G V D, Chen K H, et al. End-to-end timing analysis of sporadic cause-effect chains in distributed systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 2019, 18(5s): 1-24
- [37] Davare A, Zhu Q, Di Natale M, et al. Period optimization for hard real-time distributed automotive systems// *Proceedings of the Design Automation Conference (DAC)*. San Diego, USA, 2007: 278-283
- [38] Zhu S, Xu L, Goodman E D, et al. A new many-objective evolutionary algorithm based on generalized Pareto dominance. *IEEE Transactions on Cybernetics*, 2021, 52(8): 7776-7790
- [39] He H, Daumé III H, Eisner J. Learning to search in branch-and-bound algorithms// *Proceedings of the 28th International Conference on Neural Information Processing Systems*. Volume 2. 2014: 3293-3301
- [40] SOAFEE. Edge Workload Abstraction and Orchestration Layer (EWAOL) Documentation. <https://meta-ewaol.docs.soafee.io/en/latest/>



TAO Hong-Yu, Ph. D. candidate. Her research interests include real-time embedded systems, real-time scheduling, and containerization.

SUN Jing-Hao, Ph. D. , professor. His research interests include cyber-physical systems, real-time embedded multi-agent systems, multicore parallel computing, real-time scheduling, and optimization.

GUAN Si-Cheng, M. S. candidate. His research interests include real-time scheduling and ROS 2.

Background

The Scalable Open Architecture for Embedded Edge (SOAFEE) integrates cloud-based validation and in-vehicle deployment through containerization and microservices, providing a technological pathway for rapid iteration and functional upgrades in autonomous driving applications. However, under SOAFEE's multilayer heterogeneous architecture, traditional real-time analysis methods are often too idealized to be reliably applicable. Since container configuration directly affects end-to-end latency, it becomes a critical breakthrough point for achieving real-time guarantees in autonomous driving systems. This paper focuses on



CHEN Yu-Ting, M. S. candidate. Her research interests include real-time scheduling and containerization.

LIAO Shun-Xin, M. S. candidate. His research interests include real-time scheduling and end-to-end latency optimization.

LI Yan-Kai, Ph. D. candidate. His research interests include real-time embedded systems, real-time scheduling, and end-to-end latency optimization.

DENG Qing-Xu, Ph. D. , professor. His research interests include real-time embedded systems, reconfigurable computing, and Internet of Things (IoT).

the problem of container configuration optimization and end-to-end timing assurance for multi-container task chains under the SOAFEE architecture.

Existing studies have explored container configuration and tuning from various perspectives. Frameworks such as conTuner, OPPerTune, and ConfAdvisor, as well as strategies based on evolutionary algorithms, machine learning, and predictive modeling, have been proposed to improve resource scheduling and performance optimization. However, these approaches mainly target general-purpose computing clusters and

cloud environments, paying insufficient attention to safety-critical autonomous driving systems, which are subject to strict end-to-end timing constraints. Particularly under the SOAFEE framework, how to jointly model and optimize container configurations and task-chain timing behavior remains an open challenge.

To address this gap, this paper proposes a Design Upon Test (DUT)-based container parameter optimization method. The approach systematically establishes the mapping between container configuration parameters (e. g. , CPU cores, memory capacity) and execution delay through empirical testing. On this foundation, a Chain Composition Optimization Algorithm (CCOA) is developed to solve the multi-container configuration optimization problem under resource and timing constraints. Experimental results show that the proposed method improves the deadline satisfaction ratio by over 40% compared with greedy, Mixed-Integer Linear Programming (MILP)-based and VPA approaches. Unlike traditional static-assumption-based models, our method builds on real measurement data, making it more representative of the operational characteristics of containerized systems under SOAFEE. It effectively guarantees real-time performance on critical paths while improving overall resource utilization.

Our research group has been deeply engaged in real-time embedded systems, multicore real-time scheduling, network optimization, and parallel computing, with related works published in leading journals and conferences, including the Chinese Journal of Computers, IEEE Real-Time Systems

Symposium, Real-Time and Embedded Technology and Applications Symposium, Euromicro Conference on Real-Time Systems, International Conference on Embedded Software, Design Automation Conference, IEEE Transactions on Parallel and Distributed Systems, IEEE Transactions on Mobile Computing, IEEE Transactions on Computers and ACM Transactions on Design Automation of Electronic Systems, etc.

This work was supported in part by the National Natural Science Foundation of China (No. 62572111, No. 62472063), the Major Research Plan of the National Natural Science Foundation of China (No. 92582120) and Liaoning Provincial Science and Technology Program(2025JH2/101330113). Within these projects, this paper addresses a specific but essential subproblem: how to achieve real-time performance assurance for containerized task chains in modern multilayer embedded edge systems. In particular, the work focuses on the configuration optimization of multi-container execution chains under resource and end-to-end timing constraints. This problem constitutes an application-oriented extension of real-time dataflow graph scheduling, where abstract scheduling and timing analysis are coupled with practical container parameters, empirical execution-delay measurements, and SOAFEE-based deployment environments. Therefore, the proposed DUT-based modeling method and CCOA algorithm provide a concrete solution component for the broader research objective of improving real-time scheduling and timing assurance in complex embedded software systems.