Silverline: Lightweight Virtualization and Orchestration of Distributed Systems

Arjun Ramesh[1] Tianshu Huang[1] Emily Ruppel[3] Dakshina Dasari[4] Behnaz Pourmohseni[4]
Fedor Smirnov[4] Marco Giani[4] Paolo Pazzaglia[4] Charles Shelton[3] Nuno Pereira[2]
Arne Hamann[4] Dirk Ziegenbein[4] Anthony Rowe[13]

[1]Carnegie Mellon University [2] ISEP/P.Porto and INESC-TEC [3]Robert Bosch LLC [4]Robert Bosch GmbH

{arjunr2,tianshu2,agr}@andrew.cmu.edu
{nap}@isep.ipp.pt
{emily.ruppel,charles.shelton}@us.bosch.com
{dakshina.dasari,behnaz.pourmohseni,fedor.smirnov,
marco.giani, paolo.pazzaglia,arne.hamann,dirk.ziegenbein}@de.bosch.com

***Abstract*—We introduce Silverline, a novel framework for lightweight virtualization and orchestration of distributed real-time systems. Leveraging WebAssembly (Wasm) for robust sandboxing and multi-language (polyglot) capabilities, Silverline decouples applications from their platforms through distinct manifests, enabling a centralized orchestrator to optimize resource allocation and deploy Wasm modules seamlessly across the edge-cloud continuum. It features a split data and control plane with orchestration sidecars, allowing applications to use native communication protocols and respond autonomously to network changes. We evaluate our framework in two real application contexts: an industrial automation use-case and an automotive body electronics demonstrator. Through micro-benchmarks and end-to-end testing, we demonstrate Silverline's potential for managing real-time workloads in diverse heterogeneous ecosystems.**

***Index Terms*—Distributed Systems, Real-time, WebAssembly, Edge Computing, Orchestration, Industrial Automation, Automotive Systems**

I. Introduction

Being anchored to the physical world, embedded systems are often inherently distributed. Yet, to curb system complexity, designers often gravitate towards centralized designs such as the simple thin-client architectures seen in Programmable Logic Controllers (PLCs) or automotive Electronic Control Units (ECUs). However, as computing and networking capabilities continue to scale, a more distributed design would more naturally provide improved flexibility and reliability to meet the increasing compute demand of embedded workloads.

Unlike their centralized counterparts, distributed embedded systems present a multitude of challenges. Given the intricate interactions between distributed programs — real-time constraints, concurrency, and even physical I/O, these systems are difficult to deploy, monitor, and debug, which is further exacerbated by the limited observability of embedded systems. The integration of heterogeneous devices adds another layer of complexity: these devices often belong to distinct ecosystems, requiring specialized knowledge and tools to manage diverse targets – e.g. ARM microcontrollers, RISC-V single-board computers, and x86 servers – effectively.

To help manage these complexities, many have proposed extending cloud orchestration frameworks to the edge [25, 38, 49, 52, 54, 88]. However, traditional cloud-centric architectures struggle to meet the specific requirements of embedded and edge systems:

(1) Many cloud-inspired systems use containers or virtual machines as a virtualization primitive that are challenging to configure, cannot be embedded into existing domain-specific runtimes (e.g industrial automation or automotive applications), and have difficulty scaling down to application-class or smaller processors. Additionally, they fall short in meeting the demands of timing-sensitive applications deployed on resource-constrained devices due to their slow startup times and large memory footprints (in the MB range) respectively.

(2) Cloud infrastructure also lacks fine-grained resource controls (e.g., CPU, memory, and network) – along with the required abstractions – which are necessary to support real-time requirements. This is well understood in monolithic designs but remains a challenge in distributed systems, especially across versatile software stacks.

(3) Finally, distributed embedded ecosystems typically operate under assumptions of unpredictable environments and unreliable networks (e.g. 5G) where standard cloud-based reliability mechanisms fall short. They also often use *domain-specific* communication interfaces (e.g., CAN, TSN) that are challenging to abstract across in a uniform manner.

To address these challenges, we propose Silverline[1] , a distributed computing framework featuring lightweight virtualization and orchestration to support reliable mixed-criticality

[1]Silverline is a pun on "every cloud has a silver lining," representing the optimistic, lightweight edge that complements cloud computing systems.

distributed applications spanning the edge computing stack. We incorporate the following solutions to address the needs of edge-cloud systems:

(1) **Lightweight Virtualization using WebAssembly**: We base our framework on WebAssembly (Wasm) [41], a polyglot bytecode-based virtualization technology with provable sandboxing [58] that has seen strong adoption in the cloud [23, 30, 35, 75], and increasing interest in edge applications ranging from microcontrollers and IoT devices [55] to automotive [27, 70] and industrial applications [87]. To facilitate communication amongst isolated Wasm sandboxes, we design a protocol-agnostic *channel* API which allows for abstraction over a wide variety of communication backends used in edge and embedded ecosystems.

(2) **Resource Controls and Orchestration**: Instead of applications that are tightly coupled to their underlying platforms, users provide decoupled application and platform manifests. *Application manifests* capture properties such as end-to-end timing or reliability requirements, while *platform manifests* specify available resources (e.g., CPU type, memory, networking) and physical I/O. An orchestrator then uses these manifests to provision the system and deploy Wasm modules to edge Host instances, optimizing them under schedulability constraints.

(3) **Split Data and Control Plane Orchestration**: Given the variety of application domains supported by embedded systems, there is no "silver bullet" communication solution. For this reason, we separate the platforms' control messages (e.g., startup, shutdown, and dispatch) into a simple Publish/Subscribe (Pub/Sub) messaging system. The applications are then free to communicate using their own native protocols. To provide applications with the ability to react to changes in their network on their own timescales, Silverline deploys *sidecars* – Wasm modules containing delegated orchestration logic – that can autonomously react to changes at extremely low latencies, improving reactivity of failover mechanisms. The explicit separation between a best-effort control plane timescale for the orchestrator and a real-time data plane timescale for the sidecars enables Silverline to offer both global optimization of long-term objectives (e.g., costs or resource load) and real-time reaction to dynamic events without jeopardizing the deployed applications. These features allow users to provision dynamic mode changes across applications that would normally be static due to safety and/or real-time requirements.

Contributions This paper presents the design of the Silverline architecture as well as several practical implementations of the same. Three core contributions are realized in this work:

(1) **Real-Time Aware Edge Orchestration**: Split data and control planes decouple real-time data communication of applications from non real-time orchestration control. With orchestration logic removed from the real-time critical path, data-plane *sidecars* in Silverline offer capabilities for pre-staged updates and quick failover reactions in real-time without orchestrator involvement, thus standing out from prior approaches like TinyLink [40] or dspace [34].

(2) **Adaptable Communication Abstraction for Wasm**: The *channel* API in Silverline facilitates a generalized abstraction for inter-module communication, allowing Silverline to handle diverse and complex communication flows present in existing software infrastructure. This distinguishes it from existing Wasm abstractions for embedded systems, such as WASI extensions (e.g., HTTP [5], I2C [10]) or OmniWasm [14], which are tailored to specific protocols or chain-based application pipelines respectively.

(3) **Integration Across Diverse, Practical CPS Stacks**: The flexible abstraction of Silverline across programming languages with Wasm, communication interfaces with channels, and high-level application requirements with application/platform manifests, facilitates end-to-end integration in complex real-world CPS stacks. This paper presents two convincing case studies of integrating Silverline effectively in deeply embedded *industrial automation* (Section VI) and *automotive* (Section VII) system stacks.

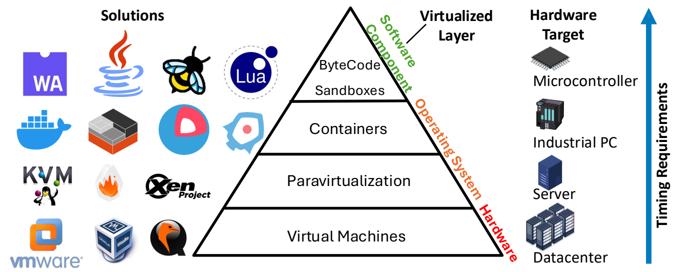

Fig. 1: The Virtualization Pyramid. As less of the computing stack is virtualized, virtualization techniques can deploy to smaller hardware at the cost of application portability effort.

II. Background

Today, systems are custom-built, tied to specific hardware, operating systems, or network physical layers that are difficult to debug, port, and upgrade. Silverline addresses these shortcomings by building upon multiple concepts related to virtualization and orchestration pioneered in part by the cloud computing community. In this section, we provide background on these topics from an edge computing perspective.

A. Lightweight Virtualization

Virtualization allows software to run seamlessly and securely across diverse operating systems and/or ISAs. There are a multitude of virtualization options available to abstract different levels of the hardware/software stack, illustrated in Fig. 1. The challenge for heterogeneous Cyber-Physical Systems (CPS), however, is identifying a solution that fits within the memory, performance, and startup time budget of the hardware targets. A uniform virtualization solution

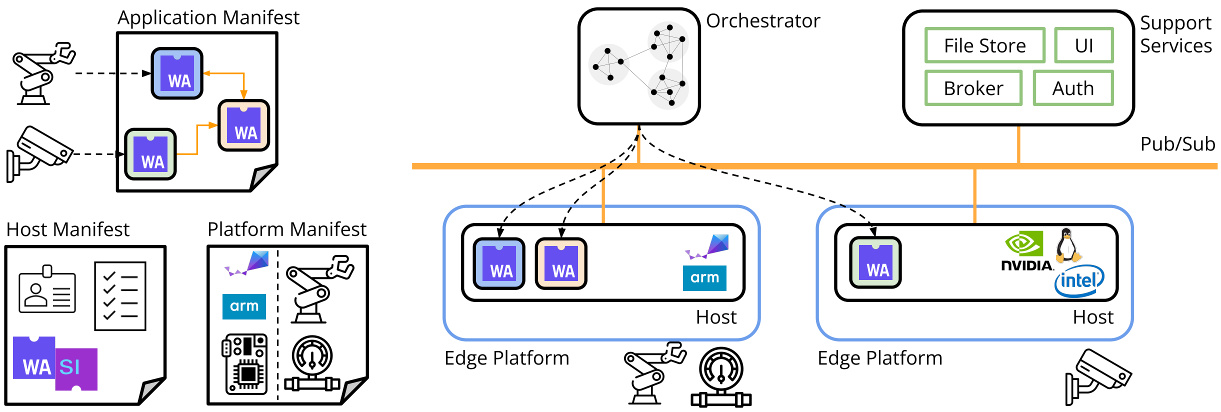

Fig. 2: The Silverline Architecture: An Orchestrator manages Host instances distributed across platforms with resources described in host and platform manifests. Applications are described by application manifests. A set of supporting services for communication, authentication, file storage and a visualization dashboard complete the system.

for edge-cloud applications should offer improved reliability, scalability, and development benefits – analogous to those of cloud microservice applications – while coexisting with existing software infrastructure.

Bytecode virtualization is particularly attractive for small devices to enable cross-platform compatibility and isolation by transforming applications into a platform-independent bytecode with guaranteed isolation at the programming language level [47, 74, 82]. By sandboxing only the minimal application logic and sharing underlying libraries and the OS, bytecode virtualization limits the binary size of the applications and may run on unmodified middlewares, which is critical for embedded targets. The proposed Silverline framework employs WebAssembly bytecode, a.k.a. Wasm [41], which is particularly interesting for CPS systems due to its low memory footprint, strict formal specification, language independence, and functional determinism, along with the ability to explicitly define custom domain-specific system interfaces callable from the virtualized software.

B. *Orchestration*

In the cloud, management of resources and applications is typically done with centralized orchestration frameworks like Kubernetes [24], Docker Swarm [48], and AWS Elastic Container Service (ECS) [72]. These were developed to manage stateless compute clusters in a best-effort context. However, unlike general-purpose cloud workloads, CPS applications are often subject to strict timing constraints for sensing actuation flows in safety-critical environments, demanding stronger guarantees than just best effort. Moreover, such applications must adhere to rigorous reliability, integrity, and safety requirements dictated by the application domain (e.g., automotive or industrial automation). These requirements exceed the capabilities of current cloud orchestration frameworks, which *warrants a solution that meets the distinct requirements of reliable mixed-criticality applications in CPS.*

III. Silverline Ecosystem Overview

Silverline aims to streamline the development of reliable, distributed, mixed-criticality applications on heterogeneous edge platforms using Wasm as a lightweight virtualization solution. Unlike cloud-based frameworks that usually use standard networking interfaces, Silverline decouples control messages for orchestration from the underlying data plane that connects applications, which helps build heterogeneous communication networks that can support real-time requirements.

System Architecture Silverline is composed of several software *Host instances* dispatched across hardware edge platforms managed by a centralized *Orchestrator* as shown in Fig. 2. The Host instances serve as specialized module executors: thin wrappers around a Wasm engine that register with the Orchestrator and provide an interface to the rest of the system with platform-level resource isolation. They also periodically update the Orchestrator about their resource status (e.g., cpu loads) and the state of their managed modules. The Orchestrator is responsible for handling the deployment requirements of the Silverline applications and implementing a satisfactory global deployment across Host instances that meets those constraints. A *Pub/Sub bus* serves as the control plane backbone of Silverline, allowing communication between all components. Silverline uses specific topics on this bus to provide access control, notifications for Host updates/configuration, and to shepherd profile information from all components. Furthermore, Silverline is easily extensible with core support services (authentication, logging, or dashboard) implemented as plug-in components connected to the Pub/Sub bus.

Application Model Silverline defines applications as a set of communicating software *modules* in the form of Wasm binaries. Inter-module communication in Silverline is designed to be flexible, allowing applications to use the baseline Pub/Sub backend or leverage arbitrary domain-specific communication. This is enabled through the use of an abstract

```
"cpu": {
  "arch": "X86_64", "bits": 64,
  "model": 107,
  "family": 15,
  "cores": 4,
  "class": null,
  "l1i_size": 262144,
  ...
  "l3_size": "512 KB",
  "qos": {"rt": 1, "cfs": 3},
  "cpufreq": "999.9290 MHz"
}
```

Fig. 3: Sample Platform Manifest CPU Section; some fields are omitted for brevity.

communication interface called *channels* that is protocol- and platform-independent. Silverline applications written against the channel API allow developers to customize the channel back-end implementation based on domain-specific use cases, as demonstrated in our Section VI case study. Wasm modules individually possess no knowledge of the data flow among the application modules. Wasm modules are configured at the application level through a user-provided *application manifest* which enables flexible adaptation of inter-module communication by relinking channel *endpoints*. The application manifest also includes application modules' deployment requirements (deadlines or QoS) which are used by the Orchestrator to correctly deploy the application.

IV. Silverline Implementation

Our Silverline implementation consists of three main components: application and resource descriptions, a *Host* that controls the local state of the platform, and an *Orchestrator* that handles the global control and deployment state.

A. Application and Resource Description

Describing the behavior of a distributed application can be a tedious and error-prone process [65], which is further exacerbated in systems with timing constraints. We aim to strike a balance between simplicity and specificity of resource controls using a custom *manifest* format detailing applications and resource reservations (memory, CPU, network), where applications or platforms are treated as best effort by default in the absence of specified reservations.

Platform Manifest Each platform has a *platform manifest* that describes its hardware and system features, used to inform the platform's Host instance(s) and the Orchestrator about the platform's capabilities. This manifest is automatically generated on each platform and contains a section for each resource — CPU, memory, network, and optional resources such as transducers and accelerators.

Host Manifest A single hardware platform may support multiple Host instances, so Host manifests (Fig. 4) provide instance-specific identification and platform information. Host instances are identified by a unique ID (uuid) and a user-friendly name. The Host feature information includes the system APIs it supports, a platform ID, and "Tags" (e.g., ctrlX) utilizable by the Orchestrator for deployment decision. The

```
{
   "uuid":"5f937916-d29d-4f66-801e-3d69f57728e2",
   "name":"edge_1",
   "apis":[
      "sl:channels",
      "wasm:wasi_preview1"
    ],
   "metadata":{
      "tags":[ "edge_node" ],
      "platform_id":"IPC1"
    }
}
```

Fig. 4: Sample Host Manifest for an edge node that supports both channels and WASI APIs.

```
name: "app1"
modules:
- name: "process"
  file: "process.wasm"
  trigger:
    type: "periodic"
    periodInNs: 50000000
  inputChannels:
    active: true
    channels:
      - name: "x"
        endpoint: "ep1"
  outputChannels:
    active: true
    channels:
    - name: "y"
      endpoint: "ep2"
      payloadSizeInBytes: 20
```

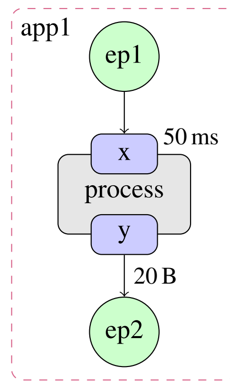

Fig. 5: Sample application manifest for a simple application `app1` with a single module `process` that triggers every 50 ms and produces a 20 B output flow.

Host sends this manifest to the Orchestrator when it registers with the latter.

Topology Manifest Deployment decision-making requires knowledge of the network configuration and topology, which is specified through a *topology manifest*. Although we currently enforce a fixed, unchanged topology manifest for a single deployment, future iterations of Silverline could enable the Orchestrator to dynamically capture the network topology information from network controllers as supported by the Centralized Network Configuration (CNC) in IEEE 802.1Q [44].

Application Manifest Applications are modeled as a set of interconnected Wasm actor modules that communicate over dedicated channel interfaces supported by the Silverline ecosystem (e.g. MQTT or CAN). The requirements and reservations for these modules and communication channels are specified in an *application manifest* (see Fig. 5), which are capable of defining complex applications consisting of multiple modules, multiple channels with different payload sizes, and different activation modes for data transfer (periodic or event-based).

B. The Host

The Silverline Host (Fig. 6) is a runtime software layer deployed on a platform in a Silverline deployment, which executes application code (Wasm modules), manages resource isolation mechanisms, and implements the I/O interfaces of

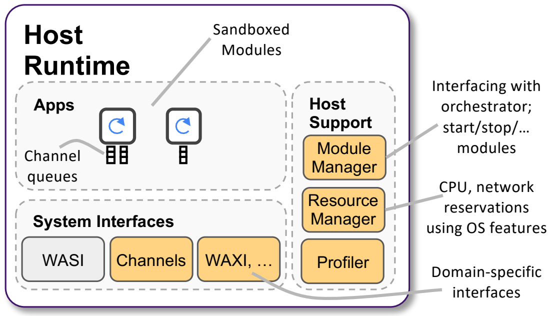

Fig. 6: Silverline Host Components. Silverline Host additions in orange: System interfaces for domain-specific access, multi-module management, resource management and profiling.

Silverline. Our reference implementation is a thin wrapper around an off-the-shelf Wasm engine — the WebAssembly Micro-Runtime (WAMR) [12], which has leading performance [80], suitably low memory overhead, and support for a diverse set of operating systems used in edge contexts including microcontrollers. Using Wasm as our computing substrate offers in-process memory sandboxing, control flow integrity for executing modules, and protection from remote-code injection attacks by design since the execution state of Wasm modules are not addressable directly by the module itself. For safety-critical applications, trusted Wasm engines can be used with provable guarantees through safety certification or formal verification [16, 51]. Most engines also allow programs to communicate directly with the host platform using the popular WebAssembly System Interface (WASI) [9] API.

Temporal Isolation Although the Wasm engine provides spatial isolation across the hosted modules, embedded real-time applications also often require temporal isolation. To address this, the Silverline Host implementation adds capabilities for performing underlying hardware resource isolation and monitoring. The Orchestrator uses the relevant platform manifest of the Host to decide relevant reservations on deployment, including scheduler parameters (e.g., CPUs, budget, deadlines). The Host then enforces these Orchestrator-assigned isolation properties using underlying OS interfaces. CPU isolation is enforced in Linux with `cgroups` and `SCHED_DEADLINE`. Network isolation is enforced through *channels* and Time-Sensitive Networking (TSN) [46] on supported synchronized platforms. Network-isolated Host instances are configured by the Orchestrator to use strictly assigned, non-overlapping time slots based on a global schedule generated by the Orchestrator using the application and topology manifest constraints, preventing interference between different Silverline applications. TSN's time-aware shaper and preemption mechanisms [45] also enforce isolation from non-Silverline applications which may be using the same communication network.

Silverline Host System Interfaces To allow communication between modules, the Host exposes the *channel* API: a Pub/Sub interface with a file-descriptor-like API. Using channels for communication allows network monitoring, while also being protocol- and platform-independent, allowing developers to use any implementation backend of choice. Our reference implementation uses a Paho-MQTT [3] backend with an additional *loopback* optimization in the Host. Loopback improves the communication efficiency of co-located Wasm modules on the Host by performing direct message delivery to the target message queue, bypassing the MQTT bus.

Additionally, modules running on Silverline Host instances can also use standard system interfaces (WASI [9]) and extensions supporting domain-specific application, linkable through WAMR, to access non-standard devices and libraries. We demonstrate Host extensibility for two domains – industrial automation and automotive systems – below.

Domain Extension: Industrial Automation with WAXI

We demonstrate the embedding of the Silverline Host into the ctrlX CORE [68] (Bosch Rexroth's flagship hardware) ecosystem for industrial automation. CtrlX offers various functions for control, PLC, motion, IoT capabilities, and fieldbus interfacing (e.g. over EtherCAT) for many industrial drives in a single platform. It runs ctrlX OS, an industry-grade Linux with the RT-preempt patch [32] and features a custom automation scheduler and runtime, which handle synchronous execution and monitoring of real-time tasks using a priority driven scheduler.

With ctrlX domain extensibility, Silverline applications can tap into the existing feature-rich ctrlX automation framework unmodified, allowing reuse of its underlying data broker and scheduler runtime used in many deployed industrial systems. Control engineers are now offered the flexibility to develop and deploy untrusted distributed Wasm modules in a language/platform agnostic manner alongside other ctrlX applications, while sandboxing the modules from taking down the entire ctrlX CORE.

I/O Model To use external interfaces or communicate among each other, ctrlX applications use the ctrlX Datalayer [66], a data broker middleware that provides secure, managed, and uniform access to real-time and best-effort data for all applications. Within the Datalayer, data across the system is identified via hierarchical path addresses in an object tree. Communication between participants is then transparently optimized: applications that reside within the same node use shared memory (when in the same process space) or named pipes to communicate in real-time with low latency, while remote applications communicate using TCP.

WAXI The Silverline Host implementation for ctrlX features a custom WAXI extension, a Wasm interface to enable access to the ctrlX Datalayer to communicate with field I/O devices. Comprising of 55 C-functions and approximately 5000 lines of C++ library code, WAXI's complexity intrinsically arises from the fact that a large core of functions are required to enable fully featured applications to target Wasm. The WAXI interface provides complete host objects management, performing necessary allocation and deallocation opaquely, hence preventing developer-induced memory leaks and crashes from improper assumptions of object lifetime. It also manages ownership of objects, allowing modules to request exclusive access rights that are safely enforced by the WAXI library. We utilize this WAXI-extended Host to interface with an industrial-grade inverted pendulum machine (see Sec. VI)

Scheduler integration The Silverline ctrlX Host integrates as a dynamically-loadable bundle within the ctrlX automation framework, which is built on Apache Celix OSGI [21] and uses a priority-driven scheduler for task management. This design bridges Celix's modularity with WAXI to enable seamless module interaction with the ctrlX automation environment. Under ctrlX scheduling, a schedulable task comprises of a set of callables which can be invoked in a user-defined order. Each hosted Wasm module in a Silverline Host is a callable. These Wasm modules effectively function as *reactors* [56]: units of a program triggered periodically by the ctrlX automation framework and run under control of the ctrlX's real-time scheduler.

Domain Extension: Automotive Systems with VSS

Modern vehicles that include a wide range of devices spanning high-end processors running driver assistance functions to embedded microcontrollers (MCUs). To target such heterogeneous automotive ecosystems, we built a compressed variant of the Silverline Host for MCUs, called Host-MCU, which runs on Zephyr RTOS [33]. Host-MCU features all the necessary control-plane handling components of Host required to integrate into Silverline, but consumes as little as 128 KB of RAM and 1 MB of storage, designed ideally for MCUs. The Host-MCU exposes vehicle data to Wasm modules through an implementation of the emerging Vehicle Signal Specification (VSS) [26] interface – a standardized format for referring to vehicle-relevant data ranging from seat position to vehicle speed. We implement a limited subset of these data points via a read/write/subscribe interface.

Host-MCU focuses on judicious, non-dynamic memory allocation – a vital requirement in safety critical embedded systems – by statically allocating buffers for Wasm guest memory used by WAMR. Likewise, we implement VSS and channels using statically defined topic or data point names that are mapped to 32-bit integers rather than hierarchical strings, minimizing their parsing overhead. Host-MCU enabled devices must interface with a gateway component running on another node, e.g. a more powerful microprocessor, to convert to-and-from these statically defined data points and the more flexible formats used by other nodes. As we demonstrate in Section VII, the Silverline Host-MCU combined with a uniform VSS API allows the same Wasm binary to run on a wide range of ECUs, from MCUs to high-end microprocessors, without recompilation.

C. Orchestrator

Silverline's Orchestrator globally monitors and manages the configuration of the available Host instances and the placement and life cycle of the applications and Wasm modules.

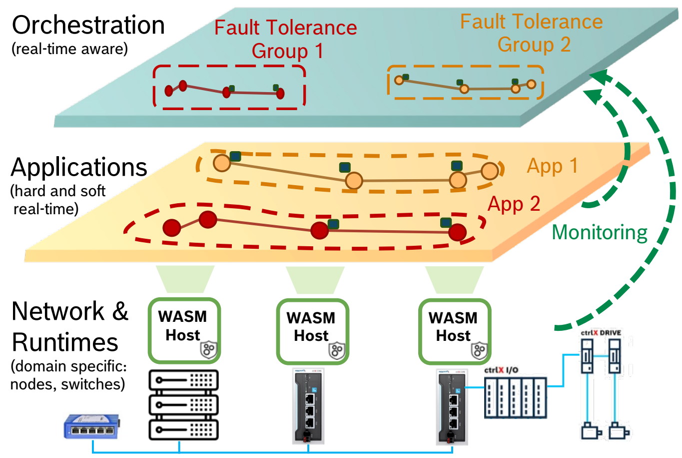

Fig. 7: Silverline separates the control and data planes to enable real-time aware orchestration and monitoring without affecting communication between modules.

Responsibilities When the user requests an application deployment, the Orchestrator finds a module placement that satisfies its requirements without violating the requirements of currently deployed applications. After deploying modules using deployment commands issued to each assigned Host, the Orchestrator monitors the system for runtime events which would necessitate adjusting module placement and/or resource configuration to maintain deployment guarantees. The Orchestrator uses a backtracking-based mapping algorithm and also is designed in a manner to be able to plug-in additional deployment algorithms.

Fulfilling Timing Requirements The Silverline Orchestrator in tandem with the Silverline Host enforces resource budget reservation for modules if requested in the application manifest. This support can be extended to provide application-level timing guarantees (e.g. end-to-end latency) using Compositional Performance Analysis (CPA) — an approach frequently used for timing analysis in heterogeneous and distributed systems [43].

APIs The Orchestrator provides two distinct APIs to facilitate internal communication within the Silverline framework and external interaction with user(s):

(1) The internal Silverline communication API defines the Host-to-Orchestrator messages used to inform the Orchestrator about the status of each Host (e.g. registration, resource usage, and state of hosted modules), as well as the Orchestrator-to-Host messages used to instruct Host instances to deploy, delete, or update hosted modules.
(2) The external REST API receives requests for the deployment, update, and deletion of applications and provides an endpoint for querying different aspects of system state.

Separation of Control and Data Planes Silverline logically *splits* orchestration services into a control and a data plane (Fig. 7). Services in the control plane fulfill non-real-time-critical responsibilities such as Host configuration, high-level resource management, and initial application deployment. These services, often involving complex optimization and analysis, fall outside the real-time constraints of core application functionality, thus are ideally implemented in the centralized Orchestrator.

In contrast, services in the data plane comprise real-time-critical responsibilities such as module failover due to resource failures. Implementing such services directly in the centralized Orchestrator introduces several serious challenges in providing low-latency real-time and reliable guarantees:

(1) The Orchestrator becomes a single point of failure for real-time flows,
(2) The Orchestrator's complexity is significantly constrained by its obligation to meet tight real-time guarantees, and
(3) The Orchestrator must be deployed in close proximity to Host instances to minimize network latency.

In Silverline, these problems are instead addressed by delegating all data-plane orchestration services to Host-deployed *sidecar* modules.

Sidecars Sidecars are highly-cohesive infrastructure Wasm modules that are deputized by the Orchestrator to deliver specific monitoring and reaction services in close proximity to applications. Sidecars are planned, verified, and proactively deployed by the Orchestrator, possessing the ability to provide timely reactions to critical runtime events (using lightweight if-then decision making procedures). With sidecars residing in the real-time data plane as specialized modules, they are operationally decoupled from the non real-time Orchestrator, allowing them to react quickly to Host-local events.

Sidecars are primarily provisioned to independently carry out orchestration-related functions for Host instances when the Orchestrator is temporarily unreachable due to network and/or system failures. In Section V-C, we demonstrate how sidecars can be used to realize transparent network disconnect detection and failover mechanisms, presenting experimental results that highlight the benefits of implementing such time-critical services using sidecars. The separation of concerns between the Orchestrator and sidecars additionally increase flexibility in Orchestrator locations, enabling it to be hosted as a managed non real-time service in a public cloud.

V. Component Evaluation

To showcase the features of Silverline components, we evaluate the operation of the Host and Orchestrator in a controlled, experimental software environment with a focus on timing and jitter. We discuss more holistic case study evaluations of Silverline in Sec. VI and VII for concrete safety-critical CPS applications targetting industrial automation and automotive systems.

A. Intrinsic Host Overheads

The Host uses the WAMR [80, 85] engine off-the-shelf and Paho-MQTT client [3], both of which have been thoroughly benchmarked in literature [15, 53, 80]. However, these are not intrinsic to the Host itself, which conceptually allows for any Wasm engine or communication backend. As such, we focus our evaluation on the overhead of the channel interface, which is intrinsically tied to the Host for control and optionally data messages. We compare the time taken to communicate using our underlying MQTT client with the call overhead of our channel interface (Fig. 8) using a simple ping benchmark on a heterogeneous set of devices. Our channels interface proves to be efficient – nearly an order of magnitude lower overhead relative to the MQTT communication backend – and can especially be capitalized upon by co-located modules on a single Host with loopback, bypassing MQTT altogether.

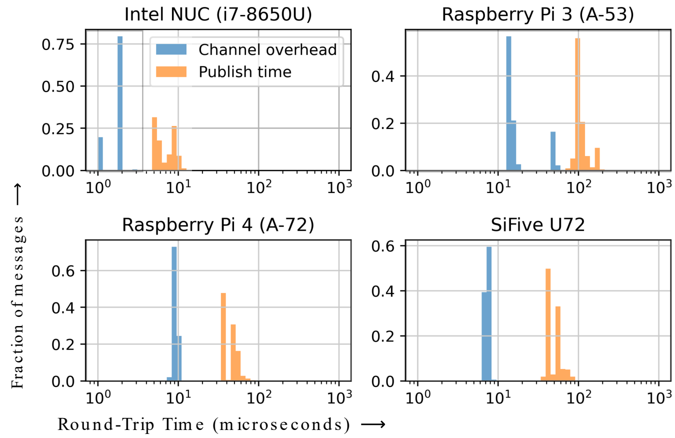

Fig. 8: Overhead of channel compared to Paho-MQTT publish overhead. Channels add a minimal overhead (≈ 10%) to the publishing pipeline.

B. Zero-Downtime Update

The ability to perform non-disruptive rolling module updates is a crucial feature in many CPS domains where application downtime is costly: for example, halting industrial production lines — even for a few seconds — causes significant financial losses. As such, the Silverline Orchestrator enables users to perform a *Zero-Downtime Update (ZDU)* of a deployed module while maintaining continuous operation of the application. A ZDU occurs in two phases:

(1) **Staging Phase**. The new module is deployed as a *shadow* with its output channels set as inactive, preventing interference with the existing module. This staging enables validation of new module behavior and uninterrupted operation by eliminating startup delays during update.
(2) **Switchover Phase**. The user prompts a switchover, activating the output channels of the *shadow*, transparently substituting the existing module with the new module in the application. This occurs almost instantaneously.

Experimental Setup We demonstrate ZDU using a simple application with one module that publishes to a specific topic every 100 ms. Our setup consists of two platforms, each with a Host, and one platform with the Orchestrator, broker, and a Pub/Sub logger, all connected in a star topology. The

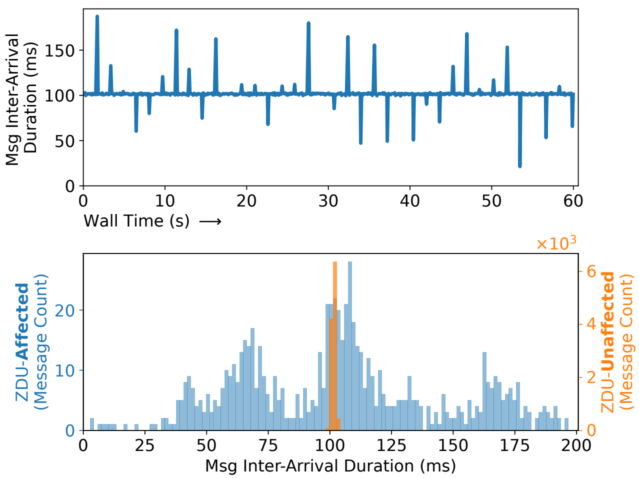

Fig. 9: Time-Series (top plot) and histogram (bottom plot) of the inter-arrival time of the output messages of the module observed under periodic ZDU events.

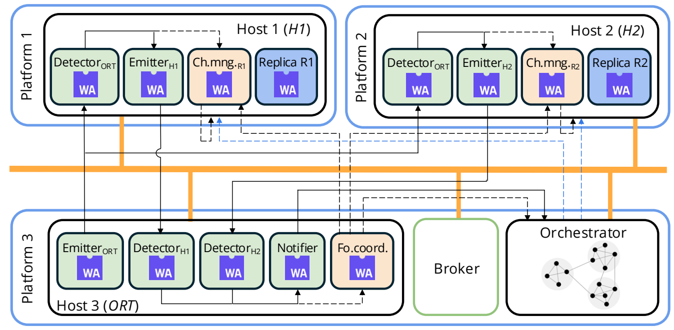

Fig. 10: Failover experimental setup, showing the deployment of Silverline components, replica module instances (blue), sidecars realizing disconnect detection (green, solid edges), and sidecars realizing failover (orange, dashed edges).

experiment begins by deploying the module on one Host followed by a ZDU to switch to the updated module on the other Host 500 ms after staging. This ZDU (swapping the active module between Host instances) is repeatedly performed every 1.5s for 20 minutes, resulting in 740 ZDU iterations and ∼12,000 messages published by the active module.

Timing Impacts To evaluate the impact of ZDU from the perspective of potential consumers of the updated module's output data, we measure the inter-arrival time of control messages at the logger (Fig. 9). We observe that spikes represent 7% of all messages that are affected by ZDU and exhibit inter-arrival times in the range 0–200 ms, while this value stays within the range 100–102 ms for the remaining unaffected messages, 93%. We conclude that from an application perspective, ZDU induces minimal jitter in the system (order of milliseconds), which is suitable for supporting rapid prototyping and updates of actively deployed systems.

C. Transparent Fault Tolerance and Failover

Silverline provides transparent, rapid failover support using sidecars to provide applications with resilience to platform disconnections and network failures. For failover support (requested per module via the application manifest), the Orchestrator deploys additional module replicas and specialized sidecars. The transparent failover support consists of three key aspects: *module replication*, *disconnect detection*, and *failover*.

Module Replication The Orchestrator deploys the module redundantly on different Host instances according to the number of replicas requested in the application manifest. The replicas are configured as hot standbys, where one instance is designated as *primary* and the other instances serve as *backup(s)* deployed with their output channels set to inactive.

Disconnect Detection The Orchestrator deploys two sidecars – the *emitter* and the *detector* – on each Host to detect network disconnections or node failures. The emitter sidecar publishes periodic *heartbeat* messages to the Orchestrator at a pre-configured rate, while the detector sidecar acts as a watchdog, listening for the heartbeat from the Orchestrator. Rather than adding capability for disconnect detection directly in the Orchestrator, we use a designated Host called ORT that is co-located on the Orchestrator's platform and hosts additional instances of *the same sidecars* (Fig. 10). The ORT hosts exactly one emitter sidecar as well as one detector sidecar *per monitored platform* and tracks heartbeats between interacting Host instances. A *notifier* module on ORT interprets the dis/reconnect messages produced by the co-located detector sidecars and forwards alerts to the Orchestrator.

Failover The failover mechanism is realized using two additional types of sidecars – the *failover coordinator* and the *channel manager* – which allows it to complete reliably within tight timing bounds after detecting the disconnection of the primary replica. The failover coordinator sidecar, deployed on ORT, tracks the connection state of each Host and assigns new primary modules based on the policy defined in the application manifest, notifying the Orchestrator of any changes. To prevent duplicate channel outputs, a *channel manager* sidecar is deployed to each Host hosting a replica, which both activates the new primary replica's channels and deactivates any replica's channels when disconnected from the Orchestrator.

Experimental Setup We aim to measure the *disconnect-reaction time*, i.e., the time between detecting a disconnection and the activation of a backup. Our setup follows Fig. 10, where we first deploy an application comprised of a single module requesting failover support with 2 replicas, R1 (primary) and R2 (backup). Repeated failovers are then forced on the system every three seconds by emulating the disconnect/reconnect of platforms 1 and 2, swapping the primary role between R1 and R2. To validate our *sidecar-based* failover implementation, we compare its disconnect-reaction time to that of an *Orchestrator-based* failover implementation, where the failover coordinator functionality is implemented directly within the Orchestrator. We run the experiment for 20 minutes,

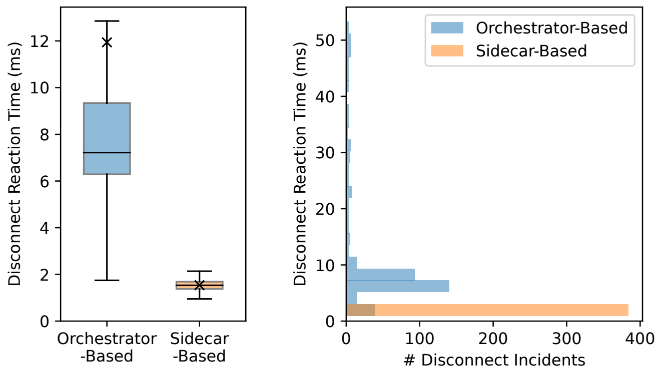

Fig. 11: Disconnect reaction fidelity of the two proposed failover variants. Sidecar-based failover has much quicker reaction times (left plot) and lower jitter (right plot) than orchestrator-based failover.

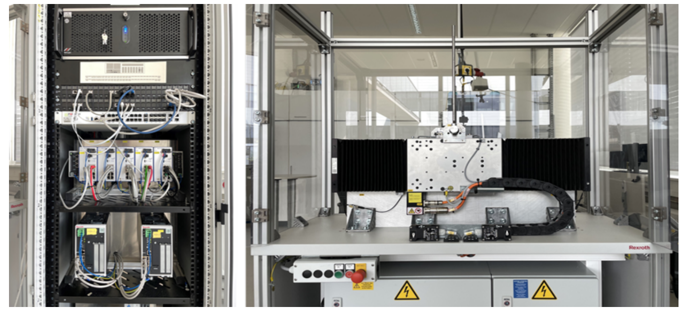

Fig. 12: Inverted pendulum demonstrator: edge server (left) and hardware (right).

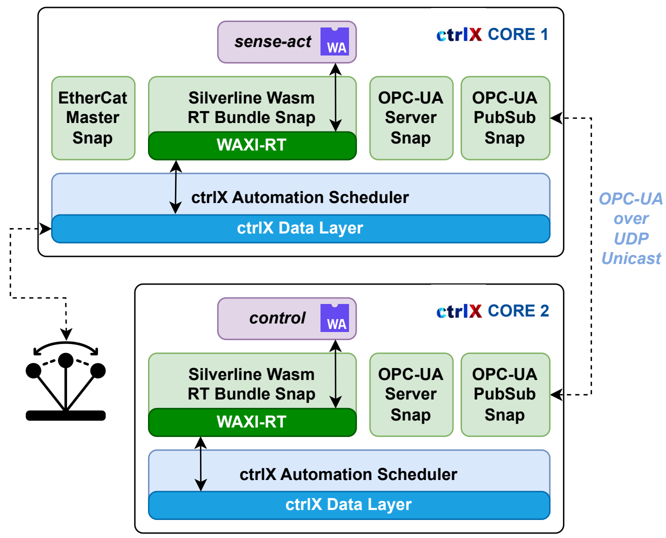

Fig. 13: Software stack for the distributed inverted pendulum application. The *control* and *sense-act* are deployed on separate ctrlX cores with communication enabled by OPC-UA

resulting in a total of 380 failover processes.

Reactivity Measurements As shown in Fig. 11, sidecar-based failover incurs a significantly lower reaction time (1.5 ms vs. 11.9 ms on average) and minimal jitter (0.9–2.1 vs. 1.7–53.3 ms), whose effect can be attributed to the Orchestrator's computational complexity and its workload variability, respectively. Thus, sidecar-based reaction mechanisms are highly effective in CPS systems where timing is a crucial factor. Note that in our setup, the Orchestrator is located in close proximity to the application, benefiting from low network latency. The measured overheads will likely be exacerbated in real deployment settings, where the Orchestrator may need to be deployed on a remote cloud server that induces increased network latency into the disconnect-reaction timing loop, further motivating sidecar-based reaction mechanisms.

VI. Industrial Automation Case Study

Conventional Programmable Logic Controllers (PLCs) are the de facto standard to meet the hard real-time requirements of complex industrial control in automated production processes. While hardware based PLCs provide high performance and reliability, they limit system flexibility and ultimately incur increased operational costs [37] (including expensive PLC runtime license costs). To demonstrate the feasibility of Silverline for Software-Defined Manufacturing (SDM), we developed a Host for the ctrlX CORE (detailed in Sec. IV-B) and constructed an inverted pendulum process, a common process used for testing control algorithm effectively, using industry grade components (Fig. 12).

Pendulum Operation The key problem in the inverted pendulum controller is to swing the pendulum up and maintain a stable, upright position. While the goal is relatively straightforward, accomplishing this reliably requires accurately accounting for real-world physics, which has strict timing requirements and is surprisingly susceptible to even minor changes in environmental conditions (e.g. temperature, humidity, machine wear). Using the EtherCAT fieldbus protocol, the controller issues commands to the pendulum system's cart, the mount plate on a linear drive [69], and reads the angle of the pendulum rod from a rotary encoder. The pendulum controller has two distinct modes of operation:

(1) **Swing-Up Mode:** The pendulum starts in this mode. The control algorithm moves the pendulum rod from its initial downward position to an upright one using an energy-based controller [93].
(2) **Balancing Mode:** Once the rod is upright, this mode balances it vertically using a Linear-Quadratic Regulator (LQR) controller to adjust the position and velocity of the cart. This mode is very susceptible to timing jitter.

A. Distributed Pendulum Application

The original monolithic program carries out the following tasks perodically in each PLC cycle:

(1) Read operational state from the safety controller.
(2) Read the current position, angle, and drive status word from the EtherCAT slaves.

(3) Calculate the next drive velocity using the pendulum controller.
(4) Write the resulting drive command to the drive controller, which will act on the pendulum at the end of the cycle.

Modularizing the PLC application We modularize this monolithic PLC application for a distributed deployment (Fig. 13) by porting it to C and splitting it into two components compilable independently to Wasm:

(1) A *sense-act* module which is deployed on a ctrlX field device (ctrlX CORE-1) and interacts with the pendulum (via the Datalayer) with a 1 ms period; and
(2) A *control* module which executes the pendulum control algorithm on an edge node hosting a virtual ctrlX OS, (ctrlX CORE-2) connected over a network

All tasks are executed on a fixed priority scheduler within the Celix Scheduling framework of ctrlX. Communication between the distributed modules is handled over OPC-UA [67] (Open Platform Communications - Unified Architecture), an industrial automation protocol for machine-to-machine communication. We use the Rexroth supplied OPC-UA Pub-Sub and OPC-UA Server snaps [67] (Version 3.2) that expose Datalayer information directly to the OPC-UA address space, allowing applications to simply read or write to the local Datalayer while external communication is handled by the underlying OPC-UA clients and servers. Our WAXI-extended Silverline Host is provides Wasm modules with access to this Datalayer and therefore also allows Silverline applications to communicate via OPC-UA.

OPC-UA Configuration We configured the reader and writer groups of OPC-UA using UA-Expert [13], which configures and selects the ctrlX Datalayer elements that will be used for publish and subscribe actions across the two ctrlX nodes over UDP Unicast. OPC-UA Pub/Sub allows the assignment of priority levels to published messages at the writer group level. In order to ensure that the sensing and actuation messages are prioritized, we set up a VLAN connection for the pendulum traffic, published the OPC-UA messages with the highest priority and communicated over the VLAN with this priority.

B. Round-Trip Delay Measurement

The total round trip time spans the duration between sensor value readings to when actuation is sent back to the EtherCAT-Master, shown in Fig. 14. This notably consists of the following periodic tasks:

- The EtherCAT-Master triggers every *2 ms* to read/write sensor/actuation values respectively to the Datalayer.
- The OPC-UA publisher sends messages every *1 ms* via UDP unicast from ctrlX1 to ctrlX2 (lack of clock synchronization induces additional delays).
- The OPC-UA subscriber also polls every *1 ms* – a higher frequency than the EtherCAT-Master to account for possible data subscription delays in the network — and may forward the expected message to the application in either the current or next cycle when it repolls depending on message arrival time.
- The sense-act module reads sensor values from the Datalayer and also writes back actuation commands (determined by the control module) every *1 ms*

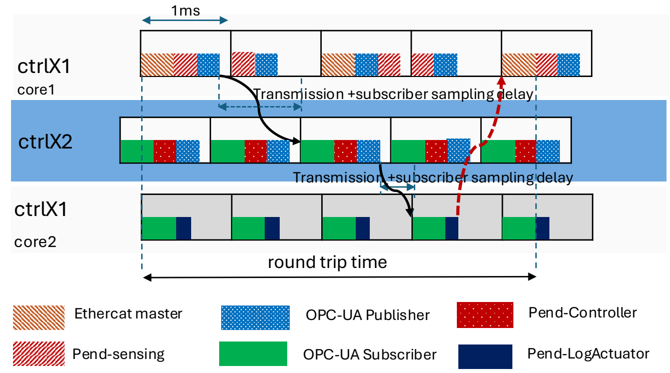

Fig. 14: Timing chart for the RTT measurement (with a possible flow sequence)

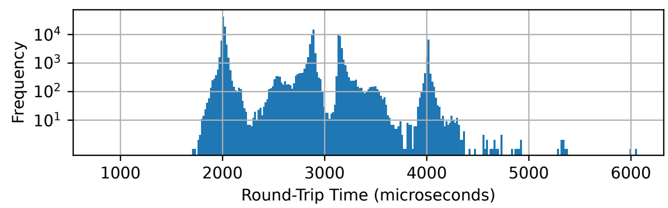

Fig. 15: Histogram of RTT measurements of the Pendulum controller input-output flow over OPC-UA (note the log scale on the y-axis)

As seen in Fig 15, the total Round-Trip Time (RTT) $\leq$ 4 ms for nearly 90% of the samples, with a dominant time spent in the communication stack and in network delays. The *pend-sense-act* module on resource-constrained ctrlX1 only has a mean execution time of 350μsecs with a 10% variability, while the control module on ctrlX2 has a mean execution time of 100μsecs with a very small variability. Specifically, the long tail can be attributed to the effects of polling based subscription (adding up on both sides), the execution jitter of the higher priority Ethercat-Master and the non-synced ctrlX nodes. All these are functions of a typical OPC-UA implementation (which could be optimized), but most importantly RTT is bounded and still proves sufficient to execute both the swing-up and balancing control algorithms of the pendulum on the edge node.

Impact of Controller Design Our controller features a combination of energy-based control and state-feedback control based on [22]. The control is executed at double the frequency of the EtherCAT master to prevent "dead" period of actuation, since the existing values in the Datalayer are used for actuation in the absence of controller update. The control system, partly due to its design, behaves as weakly-hard-real time rather than hard real-time. In particular, the controller is designed to be robust towards uncertain and variable delays, allowing it to handle a moderate range of RTTs (Fig. 15).

Applicability We were able to demonstrate control-function offloading to the edge using an *off-the-shelf closed-source communication stacks (like OPC-UA) and an existing industrial automation software stack*. On the basis of our RTT results, we believe the framework can be used in real-time applications with deadlines in the tens of milliseconds range such as mobile robots, automated guiding vehicles on the factory floors, process automation in chemical/pharmaceutical plants [11, 19], or for data intensive predictive analysis with closed-loop feedback to the production system.

Furthermore, the integration of Silverline into the ctrlX ecosystem enables flexibility, safety, and productivity gains for application developers and machine operators. With Wasm, developers can now break the shackles of conventional PLC programming with structured text and expensive commercial PLC runtime licenses, allowing development across multiple language of choice. The test and deployment lifecycle can also be streamlined since machine operators can safely execute any (potentially buggy) Wasm module in a sandboxed Silverline Host. All calls to the underlying platform must pass through the Wasm runtime, adding another layer of isolation and safety.

With dynamically loadable tasks in Silverline, developers now also need not recreate a new ctrlX app for every update. They can simply focus on writing their function, compile it to a Wasm module, and deploy it on a Silverline Host running on a ctrlX device. New languages do need to expose the API definitions of the domain-specific interface used, which can easily be performed using the `bindgen` tool. We exposed the WAXI interface to the Rust, allowing us to develop ctrlX applications from a memory-safe language.

VII. Automotive Case Study

In this case study, we developed the Host-MCU (see Sec. IV-B for details) for Silverline to run a fully portable application binary on two resource-constrained embedded ECUs with different hardware architectures and operating systems. Developers would typically use very different software toolchains to program these ECUs, but Silverline unifies this development.

Use Case Description We used Silverline in conjunction with a VSS API and the Kuksa Vehicle Databroker [31] to build a sunroof control application that runs on both a pre-production zone controller (ZCU) and a more powerful vehicle computer (VCU) (Fig. 16). The sunroof controller is deployed as a Wasm module (with a 100 ms period) which reads in the position of a control knob, applies an EMA filter [63], and writes the output to a motor that controls the position of a sunroof. The ZCU is an eight-core Arm Cortex-R52 microcontroller running Zephyr RTOS [33], and the VCU is a quad-core Arm Cortex-A53 microprocessor running Apertis Linux [64]. To mimic a zonal E/E architecture, the VCU serves as a gateway to the broader internet and hosts heavyweight

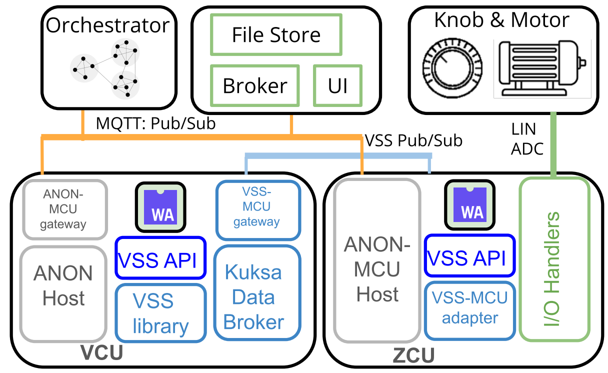

Fig. 16: In the logical Silverline to VSS setup, the separate Pub/Sub buses allow for easy integration of Silverline while maintaining portability between the VCU and ZCU.

services, communicating via automotive Ethernet to the ZCU, which interfaces directly to the control knob and motor.

To achieve portable access to VSS data, we expose the same subset of VSS API calls to Wasm modules on both the ZCU and VCU. The VSS library implementations are customized to either the ZCU or VCU because their operating systems and physical connections differ. The VCU runs a Silverline Host, the Kuksa databroker, and two gateways that reduce the overhead of supporting channels and gRPC calls respectively, on the ZCU. As in the WAXI use case, the Host is extended with VSS libraries and a VSS interface for Wasm modules. The ZCU runs the Host-MCU and communicates via the VSS-MCU gateway on the VCU to transfer data from local sensors and actuators to the Kuksa databroker. To demonstrate complete binary portability, the Wasm modules are interpreted instead of being compiled to native machine code [89]. The size of the resulting Wasm binary is only 406 B, whereas the native binary is 1.1 kB.

Discussion The sunroof controller's runtime is dominated by I/O– the controller's reads and writes to datapoints managed by Kuksa require more time than filtering the signal from a control knob. The average application slowdown in Silverline vs native on the VCU is minimal, because the virtualized portion of the application is primarily responsible for coordinating information provided by the underlying VSS library. Specifically, the average round-trip time is 4.86 ms using WebAssembly, versus 4.79 ms with a native binary. Furthermore, the Wasm module is portable, allowing seamless mobility from the VCU to the ZCU without recompilation. This case study is a promising first step that demonstrates the feasibility of portable application binaries across different architectures and operating systems in the automotive domain.

VIII. Related Work

Virtualization Virtualization in the form of full OS isolation with virtual machines or containerization has become a staple in modern compute infrastructures, including in real-time distributed systems [39, 59, 60]. This provides strong isolation but comes with significant resource overheads and startup costs, hindering scalability down to smaller platforms. In recent years, Wasm's lightweight portability and security have enabled adoption in edge computing and serverless contexts with heterogeneous platforms and high-performance execution engines [6]–[8, 36]. Systems literature has extensively studied Wasm performance [76, 84, 86] and optimizations [77, 78] – at the time of writing, Wasm runtimes show an average 2x slowdown over native code [29] with nearly-native compiler optimizations for certain ISAs such as ARM64 [90].

The extended Berkeley Packet Filter (eBPF), has also been leveraged for general purpose virtualization in microcontrollers [91, 92]. However, Wasm targets a more general-purpose ecosystem and has verified isolation properties that provide a much greater degree of confidence for the user than eBPF [50, 62, 81].

Programming Distributed Systems Popular approaches for programming distributed systems include the message-passing interface (MPI) [83] for efficient parallel computation in high-performance environments, remote procedure call (RPC) frameworks like gRPC [4] and Apache Thrift [2] for seamless remote function execution, and dataflow programming models exemplified by Apache Beam [1] for building complex data processing pipelines. Among these, the actor-based model stands out as a particularly attractive option for easily programming stateful CPS applications. The actor model (also used in [42], Ray [61], or Lingua Franca [57]) is employed by Silverline (when used in a standalone generic deployment) with an aim towards simplicity, allowing developers to focus on the logic of components without worrying about their deployment. However, when Silverline is embedded in an existing runtime, as shown with the ctrlX integration, it follows the reactor model.

End-to-End Latency in Distributed Systems Analyzing the end-to-end latency of processing chains in distributed systems is the subject of an extensive body of research for different systems and assumptions [17, 18, 20, 28, 71, 73, 79]. The Silverline Orchestrator features a modular resource analysis component that receives a system description to assess a specified deployment's end-to-end performance.

Orchestration There are many recent attempts to support edge-cloud application orchestration, most of them focusing on extending existing Cloud frameworks like Swarm [48] and Kubernetes [24] to the Edge. Furthermore, there are efforts that focus on designing resource-constrained lightweight runtimes [38]; reducing the state storage consistency guarantees in Kubernetes clusters [49]; performing device discovery at the Edge [25]; extending Kubernetes to the Edge with a compatible network and runtime infrastructure [88]; or developing an energy and interference-aware scheduler [52]. With the Silverline orchestrator and the split data-control plane the focus is on fault-tolerance, enabling zero-down time updates and meeting end-to-end latency requirements of distributed real-time applications.

IX. Conclusion

We introduced Silverline, a framework for transparently and efficiently orchestrating edge-cloud applications using WebAssembly (Wasm) as a lightweight virtualization technology. Silverline relies on isolation mechanisms across the distributed system to execute applications with end-to-end timing and/or reliability requirements. Our evaluation in industrial automation and automotive domains shows that Silverline adds minimal overhead to distributed communication and optimizes software workflows for developers. With abstractions over communication protocols (MQTT, OPC-UA via ctrlX Datalayer) and extensible interfaces (WAXI, VSS), Silverline seamlessly integrates with domain-specific ecosystems and supports full binary virtualization with negligible overhead. We showed that Silverline's design approach improves application reliability with transparent failover and facilitates features such as zero-downtime updates.

Acknowledgement

We extend our sincere thanks to the German Federal Ministry for Economic Affairs and Climate Action (BMWK) for supporting this research project 13IK001ZF "Software-Defined Manufacturing for the automotive and supplying industry https://www.sdm4fzi.de/". Additionally, we extend our thanks to Michael Farb and Ivan Liang for their support in designing and implementing many prototypes and components (e.g. dashboards, metrics, and multiple awesome system services).

References

[1] "Apache Beam Overview," 2024, accessed: 2024-05-21. [Online]. Available: https://beam.apache.org/documentation/patterns/overview/
[2] "Apache thrift homepage," 2024, accessed: 2024-05-21. [Online]. Available: https://thrift.staged.apache.org/
[3] "Eclipse paho c client library for the mqtt protocol," 2024, accessed: 2024-05-23. [Online]. Available: https://github.com/eclipse/paho.mqtt.c
[4] "grpc - a high-performance, open source universal rpc framework," 2024, accessed: 2024-05-21. [Online]. Available: https://www.grpc.io
[5] "Wasi http," 2024, accessed: 2025-05-03. [Online]. Available: https://github.com/WebAssembly/wasi-http
[6] "Wasmer," 2024, accessed: 2024-05-21. [Online]. Available: https://wasmer.io/
[7] "Wasmtime," 2024, accessed: 2024-05-21. [Online]. Available: https://github.com/bytecodealliance/wasmtime
[8] "Wavm," 2024, accessed: 2024-05-21. [Online]. Available: https://wavm.github.io/
[9] "Webassembly system interface (wasi)," 2024, accessed: 2024-05-23. [Online]. Available: https://github.com/WebAssembly/WASI
[10] "Wasi i2c," 2025, accessed: 2025-05-03. [Online]. Available: https://github.com/WebAssembly/wasi-i2c
[11] 5G Alliance for Connected Industries and Automation (5G-ACIA), "5g for connected industries and automation," 2018, accessed: 2024-11-06. [Online]. Available: https://www.5g-acia.org/publications/5g-for-connected-industries-and-automation
[12] B. Alliance, "Webassembly micro runtime (wamr)," 2024, accessed: 2024-05-23. [Online]. Available: https://github.com/bytecodealliance/wasm-micro-runtime
[13] U. Automation, *UaExpert—A Full-Featured OPC UA Client*, 2022, https://www.unified-automation.com/products/development-tools/uaexpert.html/.
[14] M. Bai, R. Pan, and G. Parmer, "Omniwasm: Efficient, granular fault isolation and control-flow integrity for arm microcontrollers," in *2024 IEEE 30th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2024, pp. 239–251.

[15] M. Bender, E. Kirdan, M.-O. Pahl, and G. Carle, “Open-source mqtt evaluation,” in *2021 IEEE 18th Annual Consumer Communications & Networking Conference (CCNC)*. IEEE, 2021, pp. 1–4.

[16] J. Bosamiya, W. S. Lim, and B. Parno, “{Provably-Safe} multilingual software sandboxing using {WebAssembly},” in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 1975–1992.

[17] A. Bouillard and E. Thierry, “An algorithmic toolbox for network calculus,” in *INRIA*, 2007.

[18] M. Boyer, J. Migge, and M. Fumey, “Pegase – a robust and efficient tool for worst-case network traversal time evaluation on afdx,” in *SAE Aerotech*, 2011.

[19] N. Brahmi, O. N. C. Yilmaz, K. W. Helmersson, S. A. Ashraf, and J. Torsner, “Deployment strategies for ultra-reliable and low-latency communication in factory automation,” 2015, presented at IEEE Globecom. [Online]. Available: https://www.ericsson.com

[20] D. Casini, T. Blass, I. Lütkebohle, and B. B. Brandenburg, “Response-time analysis of ros 2 processing chains under reservation-based scheduling,” *Dagstuhl Artifacts Ser.*, vol. 5, pp. 05:1–05:2, 2019. [Online]. Available: https://api.semanticscholar.org/CorpusID:195825437

[21] A. Celix, “Apache celix bundles,” 2024. [Online]. Available: https://celix.apache.org/docs/2.3.0/celix/documents/README.html

[22] D. Chatterjee, A. Patra, and H. K. Joglekar, “Swing-up and stabilization of a cart–pendulum system under restricted cart track length,” *Systems & control letters*, vol. 47, no. 4, pp. 355–364, 2002.

[23] Cloudflare, “Webassembly on cloudflare workers,” 2018. [Online]. Available: https://blog.cloudflare.com/webassembly-on-cloudflare-workers/

[24] CNCF, *Kubernetes*, 2022, https://kubernetes.io/.

[25] CNCF and M. Inc, *Project Akri*, 2022, https://github.com/project-akri/akri.

[26] COVESA, “Vehicle signal specification,” 2024, accessed: 2024-05-23. [Online]. Available: https://covesa.github.io/vehicle_signal_specification/

[27] M. Danzeisen, “Truly portable vehicle applications using webassembly & wasi,” April 2023, cOVESA All Member Meeting. [Online]. Available: https://wiki.covesa.global/display/WIK4/COVESA+All+Member+Meeting+~+April+25-27%2C+2023

[28] D. Dasari, M. Becker, D. Casini, and T. Blaß, “End-to-end analysis of event chains under the qnx adaptive partitioning scheduler,” in *2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2022, pp. 214–227.

[29] F. Denis, “Performance of webassembly runtimes in 2023,” 2023. [Online]. Available: https://00f.net/2023/01/04/webassembly-benchmark-2023/

[30] Fastly, “Compute@Edge,” 2022. [Online]. Available: https://docs.fastly.com/products/compute-at-edge

[31] E. Foundation, “Eclipse kuksa,” 2023, accessed: 2024-05-23. [Online]. Available: https://projects.eclipse.org/projects/automotive.kuksa

[32] L. Foundation, “Real-Time Linux,” 2020. [Online]. Available: https://wiki.linuxfoundation.org/realtime/start

[33] ——, “Zephyr project,” 2024, accessed: 2024-05-23. [Online]. Available: https://www.zephyrproject.org/

[34] S. Fu and S. Ratnasamy, “dspace: Composable abstractions for smart spaces,” in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, ser. SOSP ’21. New York, NY, USA: Association for Computing Machinery, 2021, p. 295–310. [Online]. Available: https://doi.org/10.1145/3477132.3483559

[35] P. K. Gadepalli, S. McBride, G. Peach, L. Cherkasova, and G. Parmer, “Sledge: A serverless-first, light-weight wasm runtime for the edge,” in *Proceedings of the 21st International Middleware Conference*, ser. Middleware ’20. New York, NY, USA: Association for Computing Machinery, 2020, p. 265–279. [Online]. Available: https://doi.org/10.1145/3423211.3425680

[36] ——, “Sledge: A serverless-first, light-weight wasm runtime for the edge,” in *Proceedings of the 21st international middleware conference*, 2020, pp. 265–279.

[37] O. Givehchi, J. Imtiaz, H. Trsek, and J. Jasperneite, “Control-as-a-service from the cloud: A case study for using virtualized plcs,” in *2014 10th IEEE Workshop on Factory Communication Systems (WFCS 2014)*, May 2014, pp. 1–4.

[38] T. Goethals, F. DeTurck, and B. Volckaert, “Extending kubernetes clusters to low-resource edge devices using virtual kubelets,” *IEEE Transactions on Cloud Computing*, pp. 1–1, 2020.

[39] K. Govindaraj and A. Artemenko, “Container live migration for latency critical industrial applications on edge computing,” in *2018 IEEE 23rd international conference on emerging technologies and factory automation (ETFA)*, vol. 1. IEEE, 2018, pp. 83–90.

[40] G. Guan, B. Li, Y. Gao, Y. Zhang, J. Bu, and W. Dong, “Tinylink 2.0: integrating device, cloud, and client development for iot applications,” in *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, ser. MobiCom ’20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3372224.3380890

[41] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, “Bringing the web up to speed with webassembly,” *SIGPLAN Not.*, vol. 52, no. 6, p. 185–200, jun 2017. [Online]. Available: https://doi.org/10.1145/3140587.3062363

[42] P. Haller and M. Odersky, “Scala actors: Unifying thread-based and event-based programming,” *Theoretical computer science*, vol. 410, no. 2-3, pp. 202–220, 2009.

[43] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst, “System level performance analysis - the symta/s approach,” 2005. [Online]. Available: https://api.semanticscholar.org/CorpusID:37682

[44] IEEE, “802.1 q-2018-ieee standard for local and metropolitan area networks bridges and bridged networks,” 2018.

[45] ——, “802.1 qbu standard for local and metropolitan area networks bridges and bridged networks. amendment: framepreemption,” 2018.

[46] ——, “802.1 qbv standard for local and metropolitan area networks bridges and bridged networks. amendment: enhancements for scheduled traffic,” 2018.

[47] R. Ierusalimschy, L. H. De Figueiredo, and W. Celes Filho, “The implementation of lua 5.0.” *J. Univers. Comput. Sci.*, vol. 11, no. 7, pp. 1159–1176, 2005.

[48] D. Inc, *Docker Swarm*, 2022. [Online]. Available: \url{https://docs.docker.com/engine/swarm}

[49] A. Jeffery, H. Howard, and R. Mortier, “Rearchitecting kubernetes for the edge,” in *Proceedings of the 4th International Workshop on Edge Systems, Analytics and Networking*, ser. EdgeSys ’21. New York, NY, USA: Association for Computing Machinery, 2021, p. 7–12. [Online]. Available: https://doi.org/10.1145/3434770.3459730

[50] J. Jia, R. Sahu, A. Oswald, D. Williams, M. V. Le, and T. Xu, “Kernel extension verification is untenable,” in *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, 2023, pp. 150–157.

[51] E. Johnson, E. Laufer, Z. Zhao, D. Gohman, S. Narayan, S. Savage, D. Stefan, and F. Brown, “Wave: a verifiably secure webassembly sandboxing runtime,” in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 2940–2955.

[52] K. Kaur, S. Garg, G. Kaddoum, S. H. Ahmed, and M. Atiquzzaman, “Keids: Kubernetes-based energy and interference driven scheduler for industrial iot in edge-cloud ecosystem,” *IEEE Internet of Things Journal*, vol. 7, no. 5, pp. 4228–4237, 2020.

[53] H. Koziolek, S. Grüner, and J. Rückert, “A comparison of mqtt brokers for distributed iot edge computing,” in *Software Architecture: 14th European Conference, ECSA 2020, L’Aquila, Italy, September 14–18, 2020, Proceedings 14*. Springer, 2020, pp. 352–368.

[54] H. Lee, S. Noghabi, B. Noble, M. Furlong, and L. P. Cox, “Bumblebee: Application-aware adaptation for edge-cloud orchestration,” in *2022 IEEE/ACM 7th Symposium on Edge Computing (SEC)*, 2022, pp. 122–135.

[55] B. Li, H. Fan, Y. Gao, and W. Dong, “Bringing webassembly to resource-constrained iot devices for seamless device-cloud integration,” in *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*, 2022, pp. 261–272.

[56] M. Lohstroh, “Reactors: A deterministic model of concurrent computation for reactive systems,” Ph.D. dissertation, EECS Department, University of California, Berkeley, Dec 2020. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2020/EECS-2020-235.html

[57] M. Lohstroh, C. Menard, S. Bateni, and E. A. Lee, “Toward a lingua franca for deterministic concurrent systems,” *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 20, no. 4, pp. 1–27, 2021.

[58] A. E. Michael, A. Gollamudi, J. Bosamiya, C. Disselkoen, A. Denlinger, C. Watt, B. Parno, M. Patrignani, M. Vassena, and D. Stefan, “Mswasm: Soundly enforcing memory-safe execution of unsafe code,” *arXiv preprint arXiv:2208.13583*, 2022.

[59] A. Moga, T. Sivanthi, and C. Franke, “Os-level virtualization for industrial automation systems: are we there yet?” in *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, ser. SAC ’16.

New York, NY, USA: Association for Computing Machinery, 2016, p. 1838–1843. [Online]. Available: https://doi.org/10.1145/2851613.2851737

[60] R. Morabito, "Virtualization on internet of things edge devices with container technologies: A performance evaluation," *IEEE Access*, vol. 5, pp. 8835–8850, 2017.

[61] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan *et al.*, "Ray: A distributed framework for emerging {AI} applications," in *13th USENIX symposium on operating systems design and implementation (OSDI 18)*, 2018, pp. 561–577.

[62] S. Narayan, C. Disselkoen, T. Garfinkel, N. Froyd, E. Rahm, S. Lerner, H. Shacham, and D. Stefan, "Retrofitting fine grain isolation in the firefox renderer," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 699–716. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/narayan

[63] A. Project, "Moving average filters," 2024, accessed: 2024-05-23. [Online]. Available: https://www.alglib.net/time-series/moving-average-filter.php

[64] ——, "Apertis: Collaborative os platform for products," 2024, accessed: 2024-05-23. [Online]. Available: https://www.apertis.org/

[65] A. Rahman, S. I. Shamim, D. B. Bose, and R. Pandita, "Security misconfigurations in open source kubernetes manifests: An empirical study," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 4, pp. 1–36, 2023.

[66] B. Rexroth, *CtrlX Datalayer*, 2022, https://docs.automation.boschrexroth.com/doc/778876125/ctrlx-data-layer/latest/en/.

[67] ——, *CtrlX OPC-UA*, 2022, https://docs.automation.boschrexroth.com/iirds/cdp-metadata.boschrexroth.de\~iiDC~Product-ctrlX-OPC-UA-Pub-Sub-App/.

[68] ——, *CtrlX World ecosystem around ctrlX AUTOMATION*, 2022, https://apps.boschrexroth.com/microsites/ctrlx-automation/en/ctrlx-world/.

[69] ——, "Compact converter hcs01.1e-w0005-a-03-b-et-ec-nn-l4-nn-fw," 2024, accessed: 2024-05-23. [Online]. Available: https://store.boschrexroth.com/Automation/Drives/IndraDrive/HCS01-Single-Axis-drive/COMPACT-CONVERTER_R911331601

[70] F. Scheidl, "Webassembly: Paving the way towards a unified and distributed intra-vehicle computing-and data-acquisition-platform?" in *2020 AEIT International Conference of Electrical and Electronic Technologies for Automotive (AEIT AUTOMOTIVE)*. IEEE, 2020, pp. 1–6.

[71] J. Schlatow and R. Ernst, "Response-time analysis for task chains in communicating threads," in *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2016, pp. 1–10.

[72] A. W. Services, *AWS Elastic Container Service (ECS)*, 2024, https://aws.amazon.com/ecs/.

[73] W. Shao, B. Ye, H. Wang, G. Parmer, and Y. Ren, "Edge-rt: Os support for controlled latency in the multi-tenant, real-time edge," in *2022 IEEE Real-Time Systems Symposium (RTSS)*, 2022, pp. 1–13.

[74] H. Sharaf, I. Ahmad, and T. Dimitriou, "Extended berkeley packet filter: An application perspective," *IEEE Access*, vol. 10, pp. 126 370–126 393, 2022.

[75] Shopify, "How shopify uses webassembly outside of the browser," 2020. [Online]. Available: https://shopify.engineering/shopify-webassembly

[76] B. Spies and M. Mock, "An evaluation of webassembly in non-web environments," in *2021 XLVII Latin American Computing Conference (CLEI)*, 2021, pp. 1–10.

[77] W. Stackenäs, "An evaluation of webassembly pre-initialization for faster startup times," Master's thesis, KTH, School of Electrical Engineering and Computer Science (EECS), 2023.

[78] R. Szewczyk, K. Stonehouse, A. Barbalace, and T. Spink, "Leaps and bounds: Analyzing webassembly's performance with a focus on bounds checking," in *2022 IEEE International Symposium on Workload Characterization (IISWC)*, 2022, pp. 256–268.

[79] H. Teper, M. Günzel, N. Ueter, G. von der Brüggen, and J.-J. Chen, "End-to-end timing analysis in ros2," in *2022 IEEE Real-Time Systems Symposium (RTSS)*, 2022, pp. 53–65.

[80] B. L. Titzer, "Whose baseline compiler is it anyway?" in *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2024, pp. 207–220.

[81] A. VanHattum, M. Pardeshi, C. Fallin, A. Sampson, and F. Brown, "Lightweight, modular verification for WebAssembly-to-native instruction selection," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. ACM, pp. 231–248. [Online]. Available: https://dl.acm.org/doi/10.1145/3617232.3624862

[82] B. Venners, "The java virtual machine," *Java and the Java virtual machine: definition, verification, validation*, vol. 12, 1998.

[83] D. W. Walker and J. J. Dongarra, "Mpi: a standard message passing interface," *Supercomputer*, vol. 12, pp. 56–68, 1996.

[84] W. Wang, "Empowering web applications with webassembly: Are we there yet?" in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021, pp. 1301–1305.

[85] ——, "How far we've come–a characterization study of standalone webassembly runtimes," in *2022 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2022, pp. 228–241.

[86] ——, "How far we've come – a characterization study of standalone webassembly runtimes," in *2022 IEEE International Symposium on Workload Characterization (IISWC)*, 2022, pp. 228–241.

[87] C. Woods and A. Chhokra, "An end-to-end toolchain for evaluating webassembly runtimes for cps-iot use cases," October 2022, webAssembly Research Day 2022. [Online]. Available: https://www.cs.cmu.edu/~wasm/wasm-research-day-2022.html

[88] Y. Xiong, Y. Sun, L. Xing, and Y. Huang, "Extend cloud to edge with kubeedge," in *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, 2018, pp. 373–377.

[89] J. Xu, L. he, X. Wang, W. Huang, and N. Wang, "A fast webassembly interpreter design in wasm-micro-runtime," 2021, accessed Mar. 10, 2023. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/webassembly-interpreter-design-wasm-micro-runtime.html

[90] Z. Yedidia, "Lightweight fault isolation: Practical, efficient, and secure software sandboxing," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2024, pp. 649–665.

[91] K. Zandberg and E. Baccelli, "Minimal virtual machines on iot microcontrollers: The case of berkeley packet filters with rbpf," in *2020 9th IFIP International Conference on Performance Evaluation and Modeling in Wireless Networks (PEMWN)*, 2020, pp. 1–6.

[92] K. Zandberg, E. Baccelli, S. Yuan, F. Besson, and J.-P. Talpin, "Femto-containers: lightweight virtualization and fault isolation for small software functions on low-power iot microcontrollers," in *Proceedings of the 23rd ACM/IFIP International Middleware Conference*, 2022, pp. 161–173.

[93] K. Åström and K. Furuta, "Swinging up a pendulum by energy control," *Automatica*, vol. 36, no. 2, pp. 287–295, 2000. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0005109899001405