EMDRIVE Plattformkonzept für verteilte heterogene Automotive-Echtzeit Rechennetzwerk-Architekturen

Abschlussbericht

SIEMENS AG

Version 1.0

Bundesministerium für Bildung und Forschung

EMDRIVE ist ein BMBF-Initiativprojekt im Zukunftsfonds „Automobilindustrie“.
Es referenziert auf die European Processor Initiative „EPI“ des Horizon 2020 Forschungs- und Innovationsprogramms der Europäischen Union.

ZUSAMMENFASSUNG

Projekttitel	Konzeption und RT-kompatible Erweiterung von Zentralrechenplattformen und Embedded Compute Netzwerken für zukünftige hochautomatisierte Fahrzeuge
Projektkurzbezeichnung	Mannheim EMDRIVE
Teilvorhaben	Effizientes Rechnen und Kommunizieren im Sensor-to-Edge-System für die Fertigung von komplexen mechatronischen Systemen (Automobilen)
Projektnummer:	
Förderkennzeichen	16ME0451
Datum Projektstart	01/02/2022
Projektlaufzeit	36 Monate
Dokument	Abschlussbericht
Projektpartner	Siemens AG
Eingereicht am:	
Verbreitungsebene	Public

Änderungshistorie

Version	Datum	Autor	Kommentar
0.1			Initiales Dokument

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Über die Jahrzehnte hat sich die Automobilbranche deutlich verändert. Zunächst bestimmt durch Produktentwicklungen in der Fahr- und Assistenztechnik, komplexere mechanische Systeme und höhere Sicherheitsanforderungen, wird sie nun ergänzt durch variable Antriebssysteme und steigende Individualisierungs- und Anpassungsmöglichkeiten nach Geschmack der Kundinnen und Kunden. Zusammen mit wachsendem Druck resultierend aus unvorhersehbaren Markteffekten, Krisen und globaler Konkurrenz, muss sich folglich auch die Produktion anpassen und insbesondere schnell auf Veränderungen reagieren können. Mit diesem Szenario beschäftigt sich die Siemens AG, in Bezug auf die Fertigung von hochkomplexen mechatronischen Systemen im Automobilbereich.

Ein möglicher Lösungsansatz ist der Übergang oder die Ergänzung von Linienfertigung in mit modularen Fertigungssystemen, insbesondere bezogen auf Mittel- und Kleinserienfertigung oder auf die Fertigung von selten verwendeten Bestandteilen. Diese modularen Fertigungssystemen basieren auf dem Inselkonzept. Sie setzen sich aus Aktorik in Form von Robotern oder Fertigungsmaschinen sowie Sensorik und Netzwerkanbindung aus. Den Transport der Werkstücke übernehmen Autonomous Guided Vehicles (AGVs), die das Förderband der Linienfertigung ersetzten.

Diese Inseln ermöglichen eine dynamische Adaption der Fertigungsmöglichkeiten durch Hinzufügen von zusätzlichen, Herausnehmen von überflüssigen oder einer Anpassung von einzelnen Zellen oder Fertigungsabläufen. Dies ist einfach möglich, da, im Gegensatz zur Linienfertigung, der Fertigungsablauf nicht baulich sequenzialisiert und somit veränderlich ist. Folglich bieten Fertigungsinseln eine flexible und skalierbare Lösung.

Gleichzeitig bringt dieses Konzept jedoch einige Herausforderungen mit sich. Für die Aktuatorik stellen die unregelmäßigen und sich in Auslastung deutlich unterscheidenden Betriebsbedingungen eine Verwendung dar, für die diese oft primär nicht konzeptioniert sind. Dies erfordert eine genauere Zustandsüberwachung (Condition Monitoring). Dieses basiert typsicherweise auf der Überwachung von Maschinen und deren Abweichung von zu erwartenden Zuständen. Aufgrund der benötigten hohen Messauflösung bereiten die dynamischen Umgebungsbedingungen hier jedoch große Probleme. Einflüsse von sich in der Nähe befindlichen Zellen können nicht eingeplant und kompensiert werden, da dies die dynamische Natur des neuen Fertigungskonzepts verbietet.

Um dieses Problem zu adressieren, untersucht die Siemens AG im Mannheim EMDRIVE Projekt im Teilvorhaben „Effizientes Rechnen und Kommunizieren im Sensor-to-Edge-System für die Fertigung von komplexen mechatronischen Systemen (Automobilen)“ die Herausforderungen und potenzielle Lösungswege, die die dynamische Rekalibrierung in sich verändernden Fertigungen ermöglicht. Aufgrund der hohen Komplexität dieses

Rekalibrierungsprozesses kann das Anpassen der eingesetzten Quality of Sensing (QoS) Modelle nicht auf dem Mikrocontroller durchgeführt werden. Ein verteiltes Sensor-to-Edge System bietet Abhilfe, welches den Lernprozess eines oder mehrerer Mikrocontroller übernehmen auf dem Edge Device ausführen kann. Dieses bietet gegebenenfalls dedizierte Hardware, wie beispielsweise KI-Beschleuniger. Wann diese Auslagerung nötig ist, wird dynamisch zur Laufzeit entschieden.

2 Ziele von Siemens

Siemens verfolgt im Teilvorhaben „Effizientes Rechnen und Kommunizieren im Sensor-to-Edge-System für die Fertigung von komplexen mechatronischen Systemen (Automobilen)“ das übergeordnete Ziel die Qualität der von Sensoren erzeugten Daten durch eine dynamische Betriebsstrategie für die IoT-basierte Sensor-to-Edge-Systeme entscheidend zu verbessern.

Zu Projektbeginn wurden folgende Ziele festgelegt:

- Ziel 1: Steigerung der Genauigkeit von Sensorinformationen durch dynamische Betriebsführung des Sensor-to-Edge-Systems
    - Z1.1: Einsatz von komplexeren Verfahren auf dem Edge Gerät ermöglichen eine höhere Auflösung und damit eine höhere Genauigkeit
    - Z1.2: Der Ort der Berechnung wird anhand von Qualitätsmerkmalen und einem konfigurierbaren Kostenfaktor bestimmt
- Ziel 2: Reduktion der Fehlerrate um 50% und damit eine Steigerung der Vertrauenswürdigkeit der reduzierten, eingesetzten Verfahren zur Anomalieerkennung im Sensor-to-Edge-System mit dynamischer Lastverteilung durch Anreichung der Daten mit Qualitätsinformationen („Quality of Sensing“)
- Ziel 3: Integration und Validierung der entwickelten Verfahren in dem Sensor-to-Edge-System zu Steigerung der Datenqualität und damit zu einer optimierten, dynamischen Betriebsführung. Nachweis der optimierten Funktionen mittels optimierter Hardware und Software.

Die Arbeitsschwerpunkte von Siemens lagen in AP3, AP4 und AP6. Die initiale Planung wurde dabei mit folgenden zusammenfassenden Schwerpunkten realisiert.

- AP3: Erarbeitung von Konzepten zur dynamischen Betriebsstrategie, Metriken zur Auslösung, technische Umsetzung der Migration und Anpassung an das industrielle Umfeld
- AP4: Hardware- und Softwarearchitektur der Laufzeitumgebungen für Mikrocontroller zur Befähigung zur dynamischen Ausführung und Anpassung an das industrielle Umfeld

- AP6: Konzeption und use-case in der modernen Automobilproduktion („Use-case Automotive Produktion“)

Darüber hinaus liefert Siemens wichtige Beiträge zu AP1 (Anforderungen an künftige Rechnerarchitekturen in Automotive / Industrie), AP2 (Enhanced Embedded Computing Systemarchitektur) und AP7 (Simulation, Test & Validation - Demonstration)

3 Zielerreichung

Siemens hat in EMDRIVE alle Ziele erreicht. Technische Details hierzu sind in den folgenden Kapiteln beschrieben. Als wichtigste Erkenntnis jedoch ergab sich, dass die entsprechenden Technologien zwar in anderen Kontexten existieren und verbreitet sind, im Anwendungsfall für Mikrocontroller aber weitaus weniger ausgereift sind oder gar nicht existieren. Entsprechend ergaben sich viele Herausforderungen, die zunächst die Erforschung der Basistechnologie und deren Anwendbarkeit in der Industrie nötig machten. Somit konnte für einige Ziele eine Quantifikation nicht vorgenommen werden, da die Konzeptions- und Designphase weitaus länger dauerte. Dies ermöglichte andrerseits einen deutlich größeren wissenschaftlichen Beitrag in Form von erarbeiteten Basismethoden, der in einigen Publikationen ersichtlich ist. Qualitative Nachweise zur Zielerreichung konnten dennoch erbracht werden.

- Ziel 1: Eine höhere Genauigkeit konnte durch die Integration von QoS Daten erreicht werden. Die Verlässlichkeit der Daten ist höher, so lassen sich genauere Auswertungen erzielen. Dies wird im in der Einleitung beschriebenen Kontext einer sich dynamisch ändernden Fertigung jedoch erst möglich, wenn ein Edge Gerät eine Anpassung der QoS Modelle ermöglicht. Ein verbessertes, auf Domänenwissen aufbauendes Caching erlaubt außerdem den Einsatz von komplexeren Programmen im Fertigungsumfeld, womit ebenfalls genauere Ergebnisse produzierbar sind.
- Ziel 2: Die Anreicherung durch der Sensor Daten durch QoS Daten trägt ebenso zur Steigerung der Vertrauenswürdigkeit bei. Aufgrund der Erforschung grundlegender Technologien wurde eine qualitative Einordnung einer quantitativen vorgezogen.
- Ziel 3: Ein Nachweis der Plausibilität der vorgeschlagenen Konzepte lassen sich im Demonstrationsszenario erkennen. Dieses wurde in enger Abstimmung mit der Vorfeldentwicklung der Automatisierungsbranche innerhalb Siemens entwickelt und anschließend prototypisch umgesetzt.

4 Grundlagen

4.1 Echtzeitbetriebssysteme und Interruptbehandlung

Betriebssysteme bilden die Grundlegende Abstraktionsschicht zwischen Hardware und Software. Sie beinhalten das Management von Hardware-Ressourcen wie CPU, Speicher und Peripherie, inklusive deren zeitlichen Belegung. Die Zeitliche Planung, das Scheduling, ist dabei Kernbestandteil. Sie umfasst die Anordnung von Threads sowie das Behandeln von asynchronen Unterbrechungen (Interrupts). Kann ein Betriebssystem Garantien gegenüber zeitlichem Determinismus geben, spricht man von einem Echtzeitbetriebssystem (Real-Time Operating System, RTOS). Dieser Begriff wird häufig jedoch auch für Betriebssysteme mit geringem Fußabdruck in Bezug auf Speicher und Laufzeit oder für Betriebssysteme speziell ausgerichtet auf eingebettete Systeme verwendet.

Interrupts sind essenziell für Betriebssysteme und eingebettete Systeme, da sie eine effiziente Handhabung von Ereignissen wie Hardware-Signalen, Timer-Abläufen oder Ein- und Ausgabeoperationen ermöglichen. Durch Interrupts kann der Prozessor andere Aufgaben ausführen und wird nur dann unterbrochen, wenn tatsächlich ein Ereignis eintritt, das eine sofortige Reaktion erfordert. Dies vermeidet die Notwendigkeit des fortwährenden Abfragens von Ereignissen (Polling) und steigert die Gesamteffizienz des Systems.

Allerdings bringen Interrupts auch signifikante Herausforderungen mit sich. Da sie jederzeit auftreten können, muss die Synchronisation und Datenkonsistenz gewährleistet sein, insbesondere wenn mehrere Prozesse oder Threads auf gemeinsame Ressourcen zugreifen. Ohne adäquate Synchronisationsmechanismen können Dateninkonsistenzen oder Race Conditions entstehen. Zudem kann die Systemreaktionsfähigkeit beeinträchtigt werden, wenn während der Ausführung einer Interrupt-Service-Routine (ISR) weitere Interrupts deaktiviert sind. Dies ist oft notwendig, um Datenkonsistenz zu gewährleisten, führt aber dazu, dass das System für kurze Zeit nicht auf andere wichtige Ereignisse reagieren kann. In Echtzeit- und Mikrocontrollersystemen kann dies besonders kritisch sein.

Um diesen Herausforderungen zu begegnen, folgt die Interruptbehandlung und Synchronisation auf Betriebssystemebene einem strikten Prioritätsmodell mit drei Prioritätsstufen: E_0, $E_{1/2}$ und E_1. Jede Stufe E_i unterliegt dabei spezifischen Regeln. Erstens kann eine Aufgabe auf Stufe E_i jederzeit von einer Aufgabe auf einer höheren Prioritätsstufe E_j unterbrochen werden ($j > i$), was sicherstellt, dass zeitkritische Aufgaben sofort ausgeführt werden können. Zweitens wird eine Aufgabe auf Stufe E_i nicht von Kontrollflüssen auf derselben oder einer niedrigeren Prioritätsstufe E_k unterbrochen ($k \leq i$), wodurch verhindert wird, dass weniger wichtige Aufgaben kritische Prozesse stören. Drittens laufen Kontrollflüsse auf derselben Stufe E_i sequenziell ab und werden vollständig ausgeführt, bevor sie beendet

werden (Run-to-Completion), was die Notwendigkeit komplexer Synchronisationsmechanismen reduziert.

Um die Auswirkungen von deaktivierten Interrupts während der ISR-Ausführung zu minimieren und die Systemreaktionsfähigkeit zu erhöhen, wird ein zweistufiger Ansatz eingesetzt. Zunächst wird ein Prolog auf Stufe E_1 ausgeführt, der zeitkritische Operationen durchführt, wie das unmittelbare Auslesen oder Erfassen von Daten nach einem Interrupt. Während dieser Phase sind Interrupts oft deaktiviert oder eingeschränkt, um die Datenkonsistenz zu gewährleisten. Anschließend kann der Prolog einen Epilog auf Stufe $E_1/_2$ auslösen, der die Verarbeitung der erfassten Daten übernimmt. Durch die Verlagerung des Epilogs auf die niedrigere Prioritätsstufe $E_1/_2$ können Interrupts wieder aktiviert werden, wodurch das System in der Lage ist, auf neue Ereignisse zu reagieren.

Die Epiloge auf Stufe $E_1/_2$ werden sequentiell verarbeitet. Anstehende Epiloge werden vom Betriebssystem in eine Warteschlange gestellt und nacheinander abgearbeitet. Dies stellt sicher, dass alle Epiloge vollständig abgeschlossen werden, bevor die Kontrolle zurück zur Anwendungsebene E_0 wechselt. Die sequentielle Verarbeitung auf derselben Prioritätsstufe eliminiert die Notwendigkeit zusätzlicher Synchronisationsmechanismen, da keine zwei Epiloge gleichzeitig ausgeführt werden. Die Daten, die zwischen Interrupt-Handlern (Prolog und Epilog) und Anwendungstasks ausgetauscht werden, befinden sich logischerweise auf der Epilog-Ebene $E_1/_2$. Durch die Regeln des Prioritätsmodells und die sequentielle Abarbeitung ist die Datenkonsistenz gewährleistet, ohne dass zusätzliche Sperren oder Synchronisationsmaßnahmen erforderlich sind [1]. In der Praxis wird dieser zweistufige Ansatz zur Interruptbehandlung häufig eingesetzt. Ein prominentes Beispiel ist die Implementierung in Linux-Systemen, wo der Prolog als "Top Half" und der Epilog als "Bottom Half" bezeichnet wird [2].

4.2 Quality of Sensing

In vielen Fällen gehen Entwickler und Anwender von einer unverfälschten und vollständigen Datenerfassung aus. Tatsächlich existieren jedoch vielfältige Störfaktoren, die eine präzise Messung mit Sensoren erschweren. Zu den häufigsten Herausforderungen zählen:

- Verlorene und verspätete Daten: Datenpakete werden nicht oder verspätet übertragen, wodurch zeitkritische Entscheidungen erschwert werden.
- Messfehler und Drifts: Unterschiedliche Einflüsse können die Sensorwerte verfälschen. Dabei können Abweichungen vom realen Wert auftreten, die sich über die Zeit verstärken.
- Sensorausfälle: Ein Defekt oder eine Störung der Hardware kann zu verfälschten oder unbrauchbaren Werten führen.

Da viele Systeme implizit von fehlerfreien Daten ausgehen, setzen sich frühzeitige Messfehler als Fehlinformationen in nachfolgenden Verarbeitungsschritten fort. Dies kann zu falschen Entscheidungen in Prozesssteuerungen, zu ungewollten Prozessunterbrechungen oder zu verpassten Gelegenheiten für Optimierungen führen. Angesichts dieser Unsicherheiten erweitert das als Quality of Sensing (QoS) bezeichnete Vorgehen die Sensorrohdaten um aussagekräftige Qualitätsindikatoren. Indikatoren zur Qualitätsüberwachung von Sensordaten schaffen eine Basis für belastbare Entscheidungen, indem sie die Verlässlichkeit der Messwerte sichtbar machen. Werden Abweichungen rechtzeitig erkannt, lassen sich Fehlerquellen schnell beheben oder alternative Datenquellen nutzen. Mit vordefinierten Qualitätsdimensionen wird eine durchgehende Überwachung und Steuerung ermöglicht, sodass Messfehler oder Ausfälle frühzeitig erkannt werden. Darüber hinaus liefert die Anreicherung der Messdaten mit zusätzlichen Metainformationen (z.B. Zeitstempel oder Fehlerraten) wertvolle Anhaltspunkte für eine präzisere Interpretation und Bewertung der Daten. Anstelle reiner Messwerte erhält das nachgelagerte System somit eine Kombination aus Messinformationen und zugehörigen Qualitätsparametern. So kann bei Bedarf entschieden werden, ob bestimmte Werte zur Prozesssteuerung verwendet werden oder ob sie womöglich unzuverlässig sind. Um QoS systematisch zu erfassen, wird häufig zwischen folgenden vier übergeordneten Kategorien unterschieden.

4.2.1 Timing (Rechtzeitigkeit)

Diese Kategorie beschreibt, wie zeitnah Daten zur Verfügung gestellt werden und ob sie den aktuellen Zustand des Messobjekts noch angemessen widerspiegeln. Dadurch lässt sich überprüfen, ob Daten zu spät ankommen oder ob bestimmte Ereignisse zeitlich nicht erfasst wurden. Nicht immer ist Echtzeitverarbeitung erforderlich, doch muss die Datenbasis zu dem Zeitpunkt verfügbar sein, an dem sie benötigt wird.

4.2.2 Accuracy (Genauigkeit)

Die Genauigkeit eines gemessenen Werts im Vergleich zur tatsächlichen Realität steht hier im Mittelpunkt. Wenn ein Referenzsystem vorliegt, lässt sich aufzeigen, in welchem Umfang Abweichungen auftreten. Fehlerquellen wie systematische Drifts, unzureichende oder fehlerhafte Kalibrierung oder zufällige Ausreißer können so identifiziert werden.

4.2.3 Completeness (Vollständigkeit)

Vollständigkeit bezieht sich darauf, dass alle benötigten Attribute im Datensatz tatsächlich belegt sind. Fehlende oder leere Werte gelten als Qualitätsmangel, da sie wichtige Informationen für nachfolgende Verarbeitungsschritte nicht bereitstellen. Die Relevanz der Vollständigkeit kann je nach Anwendung variieren, da nicht alle Felder gleich kritisch sind.

4.2.4 Consistency (Konsistenz)

Umfasst die Widerspruchsfreiheit innerhalb eines Datensatzes mit sich selbst oder im Vergleich mit anderen Datenquellen. Ein Widerspruch könnte beispielsweise auftreten, wenn

zwei Sensoren dieselbe Größe messen, aber stark divergierende Werte liefern, die nicht durch gewöhnliche Messtoleranzen erklärbar sind. Hohe Datenkonsistenz ist ein Hinweis auf die Plausibilität der Messungen.

Um die Qualität von Sensordaten zuverlässig zu erfassen, müssen Messwerte bereits auf Sensorebene um zusätzliche Informationen angereichert werden. Im Rahmen der Determination wird daher systematisch erfasst, in welchem Zustand ein Sensor seine Messwerte erzeugt und welche Faktoren (z. B. Umgebungseinflüsse oder Kalibrierzustände) das Ergebnis beeinflussen könnten. Diese Prozessschritte können sowohl innerhalb der Sensoreinheit als auch in nachgelagerten Auswertungseinheiten (Edge, Cloud) erfolgen.

Nach Anreicherung einzelner Messwerte mit zusätzlicher Information entsteht oft der Bedarf, die Datenqualität über mehrere Sensorquellen hinweg einheitlich zu bewerten. Ziel dieser Fusion ist es, aus den diversen Einzelinformationen eine Gesamtbewertung der Datenqualität zu errechnen, die den Zustand des gesamten Sensornetzwerks abbildet. In der Praxis kommen hierzu Methoden wie gewichtete Mittelwerte oder Plausibilitätsprüfungen basierend auf historischen Daten zum Einsatz. Darüber hinaus können Machine-Learning-Verfahren wie Support Vector Machines oder Fuzzy-Logik-Systeme zum Einsatz kommen, die einzelnen QoS-Indikatoren zusammenführen, wobei die jeweiligen Gütekriterien gezielt gewichtet werden. Durch die Fusion lassen sich redundante, aber potenziell fehlerbehaftete Werte besser klassifizieren. Beispielsweise kann ein System automatisch entscheiden, welchen Sensor es vorrangig nutzen soll, wenn mehrere Messquellen dasselbe Signal liefern, diese jedoch unterschiedliche Zuverlässigkeitswerte aufweisen. Indem Anomalien oder inkonsistente Daten herausgefiltert oder zumindest gekennzeichnet werden, bleibt die Qualität des gesamten Datenbestands auf einem höheren Niveau und bildet somit eine stabilere Basis für Folgeprozesse.

Aus den gewonnenen QoS-Informationen kann ein System in Echtzeit oder in nachgelagerten Analysen Maßnahmen ableiten. So ist es beispielsweise möglich, bei auffälligen Messwerten Alarmmeldungen zu generieren oder automatische Korrekturverfahren anzustoßen. Auch lassen sich Prozessparameter dynamisch anpassen, wenn die Güte der Eingangssignale schwankt. Erhöhte Fehlerraten können etwa zu einer höheren Abtastfrequenz oder zu einer vorsorglichen Umschaltung auf alternative Sensoren führen. Darüber hinaus bietet eine granulare Aufschlüsselung der Datenqualität in die Kategorien Timing, Accuracy, Completeness und Consistency einen gezielten Ansatz zur Prozessoptimierung. Tritt eine zu lange Signallatenz auf, könnte dies an mangelhafter Konnektivität liegen, was sich durch den Einsatz schnellerer Schnittstellen beheben lässt. Werden systematische Abweichungen sichtbar, kann dies ein Hinweis auf fehlerhafte Kalibrierung sein. Fehlende Werte machen auf Ausfälle oder Unterbrechungen aufmerksam, während unvereinbare Datensätze eine tiefergehende Analyse der Sensor- oder Systemlogik erforderlich machen. Durch die gezielte

Bewertung dieser Kriterien können Prozesse langfristig stabiler, sicherer und effizienter gestaltet werden. [3], [4], [5], [6], [7], [8], [9]

4.3 Devicetrees

Die Devicetree-Spezifikation [10] ermöglicht ein gemeinsames Verständnisses von Hardware-Schnittstellen in Computersystemen, insbesondere in eingebetteten Systemen, was essenziell wichtig ist. Ein Devicetree ist eine hierarchische Baumstruktur, in der jeder Knoten ein Gerät oder eine Komponente repräsentiert. Diese Knoten enthalten Eigenschaften (Properties), die die Merkmale des jeweiligen Geräts beschreiben, einschließlich Registeradressen, Konfigurationsparameter und weiterer hardwarebezogener Informationen.

Zusätzlich können Devicetrees auch Interruptquellen, -controller und -nummern als Knoteneigenschaften darstellen. Die Knoten werden über ihre Bezeichnungen (Labels) oder hierarchischen Pfade identifiziert, was eine eindeutige Zuordnung innerhalb der Struktur ermöglicht.

Obwohl dies nicht explizit in der Spezifikation festgelegt ist, spiegelt die hierarchische Struktur des Devicetrees implizit die verschiedenen Ebenen der Hardware wider: von der CPU über das System-on-a-Chip (SoC) bis hin zu den Peripheriegeräten. Die ersten beiden Ebenen, CPU und SoC, gelten als unveränderlich, da Änderungen an ihnen physische Modifikationen der Leiterplatte erfordern würden. Im Gegensatz dazu ermöglicht die Peripherieebene eine flexible Anpassung, da hier Sensoren, Aktoren und andere externe Geräte problemlos ausgetauscht oder hinzugefügt werden können.

In Zephyr, dem in diesem Projekt eingesetzten Echtzeitbetriebssystem, dient der Devicetree ausschließlich der Softwareentwicklung und Kompilierung während der Prototyping-Phase. Auf dem Gerät gibt es kein entsprechendes Laufzeitäquivalent; der Devicetree wird zur Compile-Zeit herangezogen, um die Hardwarekonfiguration in die Software zu integrieren und so die Funktionalität des Systems sicherzustellen.

4.4 WebAssembly

WebAssembly (Wasm) [11], [12] wurde im Jahr 2017 von den führenden Browserherstellern als leistungsfähige Alternative zu JavaScript eingeführt. Wasm ist ein niedrigstufiges, formal definiertes binäres Instruktionsformat, das eine schnelle und portable Codeausführung in den Vordergrund stellt. Wesentliche Merkmale sind dabei die Speichersicherheit durch Isolation und die Unabhängigkeit von spezifischer Hardware.

Auf LLVM basierende Toolchains ermöglichen die Generierung von Binärdateien aus verschiedenen Programmiersprachen wie C/C++, Rust und anderen. Diese Flexibilität hat dazu

geführt, dass Wasm über die Webentwicklung hinaus in Bereichen wie eingebetteten Systemen und Edge Computing Anwendung findet.

WebAssembly-Bytecode kann entweder Ahead-of-Time (AoT) oder Just-in-Time (JIT) in native Maschineninstruktionen übersetzt oder direkt interpretiert werden. In allen Fällen wird eine dedizierte Laufzeitumgebung (Runtime) benötigt. Da der CPU-Befehlssatz durch die Runtime abstrahiert wird, ist Wasm-Bytecode von Natur aus portabel über verschiedene Hardwareplattformen hinweg. Diese Abstraktion führt jedoch zu einem Laufzeit-Overhead, der je nach Optimierung der Runtime (z. B. JIT- oder AoT-Kompilierung) und der spezifischen Implementierung auf eingebetteten Geräten variiert.

In der Fachliteratur [13], [14], [15], [16], [17], [18], [19] werden für Mikrocontroller Overhead-Faktoren im Bereich von 1,14 bis 36,15 im Vergleich zur nativen Ausführung berichtet. Die geringsten Overheads werden nur durch AoT-Kompilierung erreicht, während Interpreter typischerweise einen Overhead-Faktor von über zehn aufweisen. Auch leistungsstärkere Edge-Geräte zeigen ähnliche Overhead-Werte.

Als stackbasierte virtuelle Maschine arbeiten Wasm-Instruktionen mit einem Operandenstack, auf dem Werte abgelegt, manipuliert und wieder entnommen werden können. Gemäß der Wasm-1.0-Spezifikation und ihrer Implementierung in der WebAssembly Micro Runtime (WAMR) [20], [21] besteht ein Wasm-Programm aus einem Modul, das statische Komponenten wie Bytecode, Funktionsdefinitionen, Tabellen und globale Variablen enthält. Bei der Instanziierung werden eine Modulinstanz und eine Ausführungsumgebung erzeugt, in der Symbole und der lineare Speicher (der in Wasm verwendete Speicherbereich) initialisiert werden. Die Ausführungsumgebung verwaltet einen Aufrufstack, den Operandenstack und Referenzen sowohl auf die Modulinstanz als auch auf die aktuelle Funktion.

Wasm bietet Mechanismen zur Interaktion mit dem Host-System durch das Importieren von Funktionen. Importierte Funktionen werden ähnlich wie interne Funktionen verwendet, ermöglichen jedoch den Übergang vom Wasm-Bytecode zu nativem Code. Die Verantwortung für die Gewährleistung der Speicherisolation liegt dabei beim Host-System. Die WebAssembly System Interface (WASI) [22] hat sich als Standard für die Definition von Systemschnittstellen etabliert und bietet Zugriff auf das Dateisystem sowie andere niedrigstufige Interaktionen für Nicht-Web-Umgebungen. WASI und andere sich entwickelnde Schnittstellenspezifikationen zielen darauf ab, die Portabilität und Nutzbarkeit von Wasm für eingebettete Anwendungen zu verbessern.

Wasm zeichnet sich durch eine robuste Speicherisolation aus, die durch sein Speichermodell und Typsystem erreicht wird. Der Wasm-Bytecode kann ausschließlich auf seinen eigenen, dedizierten linearen Speicher zugreifen – seinen Heap – und verwendet dazu 32-Bit-Integer-Indizes. Diese Indizes fungieren als Offsets innerhalb des linearen Speichers. Das Typsystem

von Wasm verhindert Zugriffe unterhalb der minimal zulässigen Speicheradresse, während die Laufzeitumgebung sicherstellt, dass keine Zugriffe über die maximal erlaubte Speichergrenze hinaus erfolgen. Ein Zugriff auf den Speicher des Host-Systems oder anderer Programme ist somit ausgeschlossen.

Zeiger (Pointer) werden in Wasm als 32-Bit-Offsets in globale Tabellen dargestellt, wie zum Beispiel die Funktionstabelle. Durch diese Repräsentation entfallen dereferenzierbare Typen, was das Risiko von unautorisierten oder schädlichen Speicherzugriffen erheblich reduziert.

Die Konsistenz der Datentypen wird durch statische Überprüfungen gewährleistet, die bereits während der Kompilierung stattfinden. Der Kontrollfluss des Programms wird geschützt, indem der Aufrufstack (call stack), der Informationen über Funktionsaufrufe und Rücksprungadressen enthält, vom Operandenstack (operand stack) getrennt gehalten wird, der für die Ausführung von Berechnungen genutzt wird. Kontrollinstruktionen sind so konzipiert, dass sie nur eingeschränkt auf den Stack zugreifen können, um stackbasierte Angriffe wie Buffer Overflows oder unautorisierte Veränderungen des Programmflusses zu verhindern.

Die formale Spezifikation von Wasm unterstützt die Mechanisierung, also die formale Verifikation von Programmen, und erleichtert somit die Überprüfung der Korrektheit von Bytecode und Typsystem [23], [24]. Dies fördert die Entwicklung von verifizierten Laufzeitinterpretern, die mathematisch nachgewiesen korrekt arbeiten [25]. Allerdings stellt die Verifikation von Ahead-of-Time (AoT) und Just-in-Time (JIT) Kompilierungen weiterhin eine Herausforderung dar [26], da diese Techniken zur Laufzeit Code generieren und ausführen, was zusätzliche Komplexitäten und potenzielle Sicherheitsrisiken mit sich bringt.

Obgleich aktuelle industrielle Implementierungen von Wasm-Laufzeitumgebungen häufig die Leistung über die formale Korrektheit stellen, ist zu erwarten, dass zukünftige Entwicklungen diese Lücke schließen werden. In dieser Arbeit gehen wir daher, wo relevant, von der Verwendung einer korrekten und vertrauenswürdigen Laufzeitumgebung aus.

5 Anwendungsszenario (AP6)

Das Anwendungsszenario der Siemens AG im Mannheim EMDRIVE Projekt bezieht sich auf die hochgradig automatisierte Fertigung von komplexen mechatronischen Systemen in der Automotive-Industrie und entstand in enger Abstimmung mit den Vorfeldentwicklungen innerhalb der Automatisierungsbranche bei Siemens. Das Szenario entstand in zwei Schritten, wobei sich der erste Laboraufbau am Bahnsektor orientierte, da dort bereits eine existierende Testumgebung verfügbar war. Eine schnelle Erzeugung und Auswertung von Sensordaten war somit möglich und erlaubte eine erste Evaluierung der Notwendigkeit der dynamischen

Betriebsführung (siehe Kapitel 6.3.1). Diese erste Ausrichtung diente vor allem dazu, zentrale IoT-Konzepte wie die Sensor2Edge-Kommunikation zu erproben.

In Abbildung 1 ist die Gesamtarchitektur von Siemens dargestellt. Hierbei wird Sensorik and ein zu überwachendes Objekt angebracht und die Sensordaten über ein Gateway an ein oder mehrere High-Power Computer (HPC) und ggf. in die Cloud gesendet. Sie stellt eine typische Architektur dar, die so in fast jeder Anwendung vorhanden ist. Eine Übertragung in andere Anwendungsgebiete war somit zu jeder Zeit möglich.

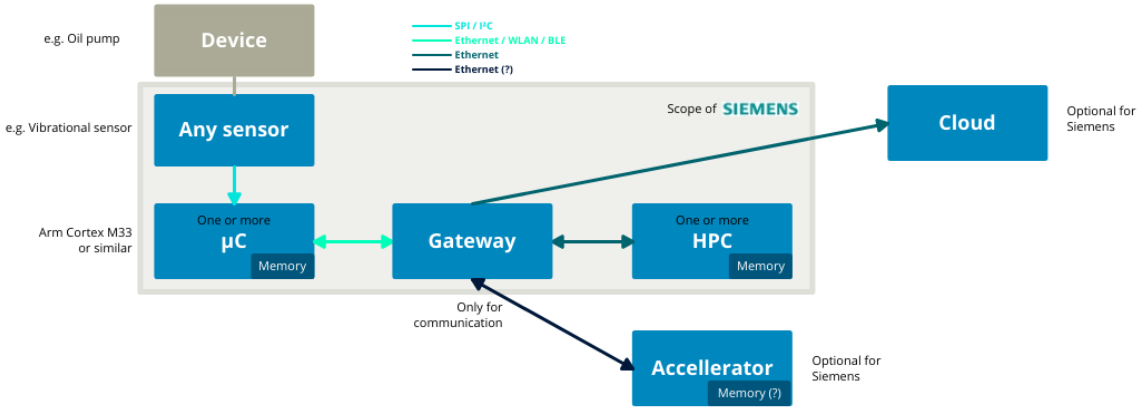

Abbildung 1: Sensor2Edge Architektur von Siemens mit Sensorik auf Edge- und Cloud-Übertragung.

Dieser konzeptionelle Übertrag erfolgte im zweiten Schritt. Hier wurde die dargestellte Architektur auf die zukünftige Automobilfertigung angewendet. Dies geschah in Zusammenarbeit mit dem Siemens Autonomous Factory Lab (AFL). In diesem wird unter anderem gezeigt, wie intelligente Anlagen selbständig auf sich ändernde Randbedingungen und Produktionsziele reagieren können, ohne dass aufwendige Umrüstungen oder manuelle Interventionen erforderlich sind. Sie bildet somit die optimale Grundlage für die Erforschung der dynamischen Betriebsführung in industriellen Anlagen.

Die dafür verwendeten Aufbauten sind modular konzipiert und erlauben eine prozessnahe Konfiguration der Hardware und Software. Dies begründet sich aus sich ändernden Produktionsbedingungen. Durch weltweite Engpässe in Lieferketten und unbeständige Marktsituationen durch geopolitische Änderungen kann sich die Nachfrage jederzeit ändern. Dies betrifft nicht nur die Quantität des Endprodukts, sondern insbesondere seine Konfiguration. Da diese außerdem im Laufe der letzten Jahre deutlich komplexer geworden ist, ist auch die Reaktion und dynamische Anpassung immer komplexer geworden. Gleichzeitig ist auch das Angebot von Vorprodukten und Bestandteilen von diesen Schwankungen betroffen. Als Folge daraus bieten sich die beschriebenen, modularen

Konzepte zur Fertigung an. Dies beinhaltet einerseits die Hardware, andererseits jedoch auch die Software. Sich dynamisch anpassende Hardware muss in deutlich höheren Maßstäben auch adaptive Software einsetzen. Die sich ändernden Umgebungen, Materialien, Formen und Produktionsschritte haben insbesondere auch Konsequenzen auf die Sensorik, da andere Verhaltensweisen der Produktion und Einflussfaktoren der Umgebung auftreten. Software muss sich diesen anpassen können und dynamisch zur Einsatzzeit die Algorithmen anpassen und nachladen.

Dieses Szenario liegt den in diesem Projekt erarbeiteten Softwarebausteinen sowie dem Demonstrator im AFL zugrunde. Als Hardwaregrundlage zur Anwendung der Softwarebausteine diente ein aus dem vorangegangenen Förderprojekt GEMIMEG II stammender, spezieller Schaltschrank, der ursprünglich zur Qualitätsbestimmung von Sensordaten im Rahmen des Quality of Sensing entwickelt wurde. Dieser Schaltschrank wurde auf die Anforderungen einer Industrial-Automotive-Production-Umgebung umgebaut, um sich für den hochgradig automatisierten Einsatz in Fertigungsstraßen zu eignen.

Des Weiteren wurde der SSI-Multisensor verwendet. Er basierend auf einem ARM-Cortex M33 (siehe Kapitel 6.1) und vereint mehrere Sensoren, unter anderem einen LSM6DLS MEMS-Akzelerometer inklusive Gyroskops sowie einen BMP280 Temperatursensor. Im Mannheim EMDRIVE Projekt wurde für diesen Multisensor ein neuer Chip integriert sowie die Portierung auf das Zephyr Betriebssystems (siehe Kapitel 6.2.1) für dieses Sensorsystem vorgenommen. Der LSM6DSL ermöglicht die Erfassung von Beschleunigungs- und Gyroskopdaten. Diese dienen in Produktionsumgebungen der Zustandsüberwachung von Maschinen, Bauteilen und Prozessen. Da die Signalverarbeitung in der Vibrationsdatenanalyse insbesondere von äußeren Faktoren wie Transferpfade und Montierung abhängt, ist es essenziell, dass diese Faktoren bekannt sind. Siemens hat bereits Erfahrung mit dem LSM6DSL Gyroskop. Entsprechende Hardwareparameter wurden in mehrjähriger Forschung ermittelt, weshalb das Sensorsystem auch im EMDRIVE Projekt Einsatz finden soll. Ein zugehöriger Treiber findet sich im ZephyrRTOS.

Abbildung 2 zeigt den Prozess, der die in diesem Projekt erarbeiteten Konzepte zur dynamischen Betriebsführung exemplarisch veranschaulicht. Ein Sensor nimmt Daten einer bestimmten elektrischen Maschine oder eines Fertigungsschritts auf. Wie dargestellt, sind diese aufgrund der modularen Hardware jedoch besonders von Störeinflüssen beeinträchtigt. Um dennoch sicherzustellen, dass in die erfassten Daten verlässlich sind, wurde ein Quality-of-Sensing-(QoS)-Modell entwickelt, das sich auf die Genauigkeit von MEMS-Sensorik konzentriert. Dabei geht es vor allem um die anwendungsspezifische Frage, wie man Sensordaten kontinuierlich auf ihre Qualität überprüft und sofort auf auffällige Abweichungen reagieren kann. In regelmäßigen Abständen wird der Zustand der Sensorik erfasst, damit statistische Anomalien und zufällige Fehler frühzeitig aufgedeckt werden, was gerade bei Motoren, Getrieben oder anderen hochbelasteten Komponenten in der Fertigung

unbeachtet sind. Die entsprechenden Analysen laufen typsicherweise direkt im Sensorsystem, um die Drahtloskommunikation möglichst wenig auszulasten, ebenfalls eine kritische Eigenschaft im industriellen Umfeld.

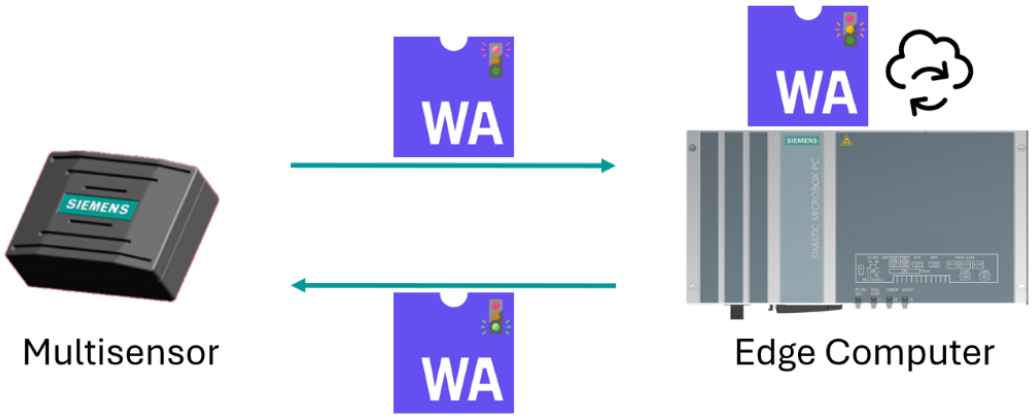

Abbildung 2: Dynamische Betriebsführung mittels WebAssembly Container zwischen Microcontroller (links) und Edge Computer (rechts) auf Grundlage von Quality of Sensing Modellen.

Im präsentierten Anwendungsfall detektiert der QoS Algorithmus eine Abweichung. Um festzustellen, ob diese ein Defekt oder eine valide Änderung der modularen Hardwarekonfiguration ist, muss diese Information typischerweise durch Externe geliefert werden, beispielsweise durch die Fertigungsplanenden. Sie könnte jedoch auch durch erweiterte Datenauswertung erlangt werden. Ist die Änderung eine valide Konfigurationsänderung, muss das QoS Modell neu trainiert werden, jedoch ohne die Sensordatenqualitätsauswertung zu unterbrechen. Andernfalls ist eine Interpretation der Daten später nicht mehr möglich. Die QoS Algorithmen werden daher zunächst an ein verfügbares Edge-Gerät (HPC) ausgelagert. Der Sensor schaltet in den Streaming-Modus, eine Variante, die die Rohdaten direkt an den HPC liefert, dafür jedoch die Drahtloskommunikationskanäle stark auslastet. Das Edge-Gerät hat nun die Möglichkeit, die alten QoS Modelle weiter auszuführen. Die Modelle sind zwar unpassend, jedoch liefern sie wenigstens gewisse Erkenntnisse. Gleichzeitig lernt das Edge-Gerät die QoS Modelle neu an die veränderte Situation an. Anschließend werden die QoS Modelle zurück auf den Sensor gespielt. Die QoS Modelle sind dabei in WebAssembly (Wasm) Containern gekapselt (siehe Kapitel 4.4), was diese Migration erst ermöglicht.

Dieser Prozess beinhaltet die kritischen Schritte der dynamischen Betriebsführung. Zusammengesetzt ermöglichen die im Folgenden präsentierten Softwarebausteine die

Ausführung dieses Prozesses und beleuchten die einzelnen Schritte und deren Herausforderungen im Detail. Diese sind insbesondere

1. Die Erkennung zur Auslösung einer Übertragung (Migration) von Software(teilen), hier realisiert durch die Abweichung der QoS Werte
2. Die Ausführung der QoS Modelle auf verschiedener Hardware (Mikrocontroller und HPC)
3. Die Migration von Softwareteilen zwischen den Geräten

Diese Inhalte werden in den folgenden Kapiteln erläutert.

Um den beschriebenen Prozess und die erarbeiteten Softwarebausteine anschaulich darzustellen, wurde in Mannheim EMDRIVE ein Video zu Demonstrationszwecken erstellt. Im ersten Teil des Video-Demonstrators erhält man zunächst einen Überblick über das AFL von Siemens, das sich durch modulare Industrie-Stationen wie Förderbänder und Roboterarme auszeichnet. Diese flexible Struktur, vergleichbar mit modernen Anlagen in der Automobilbranche, bietet ein ideales Umfeld, um innovative Konzepte unter realitätsnahen Bedingungen zu erproben. Abbildung 3 illustriert die im AFL aufgezeichneten Bild- und Videosequenzen und verdeutlicht das Zusammenspiel der unterschiedlichen Komponenten.

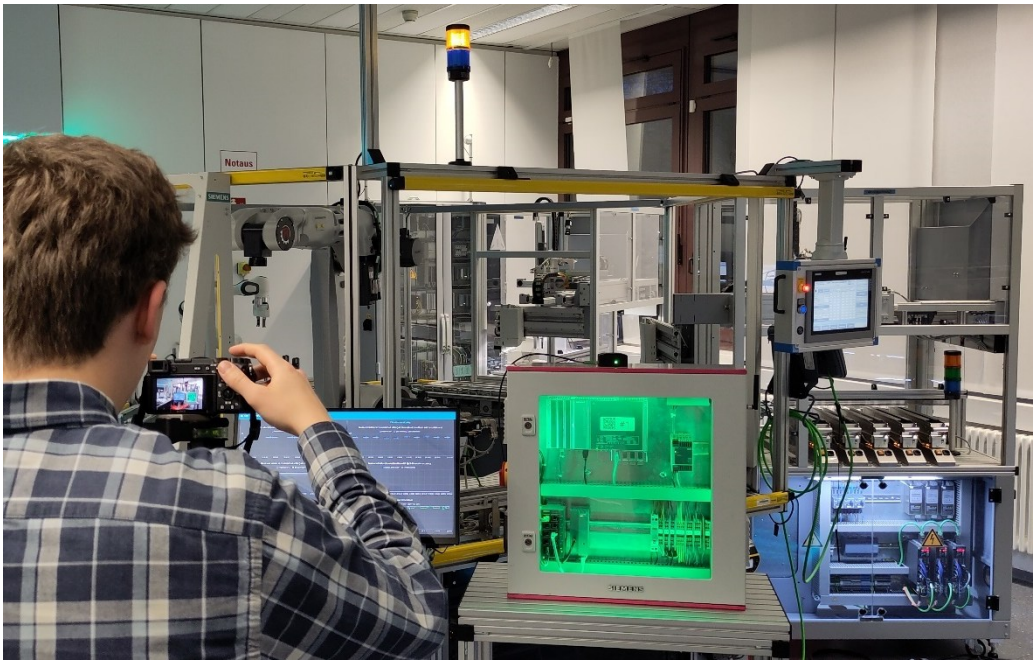

Abbildung 3: Bild- und Videoaufnahmen der Automotive Use-Case Demonstration im Autonomous Factory Lab.

Gleich zu Beginn des Videos ist das modulare Förderband zu sehen, in dessen Zentrum ein Roboterarm positioniert ist. Auf der linken Seite befindet sich ein 3-Achsen-Positionierungs-Präzisionsroboter, der Bausteine auf Transportwagen lädt, damit diese anschließend auf dem Fließband befördert und weiterverarbeitet werden können. An diesem schwenkbaren

Positionierungsarm ist der SSI-Multisensor angebracht. Im nächsten Ausschnitt wird gezeigt, wie genau der SSI-Sensor montiert ist und wie seine Status-LED zunächst blinkt und dann konstant leuchtet. Dieses Aufleuchten signalisiert eine erfolgreiche drahtlose Verbindung mit dem Edge Gerät, das im Schaltschrank verbaut ist. Abbildung 4 zeigt den montierten SSI-Sensor mit aktiver Verbindung.

Abbildung 4: SSI Multisensor auf dem 3-Achsen-Positionierungs-Präzisionsroboter montiert mit aktiver Verbindung zum Industrial Edge Gerät.

Auf dem Sensor selbst startet die Datenakquise automatisch. Diese beinhalten ein speziell für diese Situation kalibriertes QoS-Modell, wobei sich neben dem Roboterarm befindende Förderband bewusst in Ruhe befand. Das Modell berechnet fortlaufend die Datenqualität und reichert jeden übermittelten Messwert um entsprechende Qualitätsinformationen an.

Die darauffolgende Szene rückt den Schaltschrank mit dem Edge Gerät in den Vordergrund. Er wird in Grün ausgeleuchtet, was eine zufriedenstellende QoS-Datenqualität darstellt. Gleichzeitig ist zu sehen, wie der Positionierungsarm mitsamt dem SSI-Sensor Bausteine auf einen Transportwagen befördert. Ist das Förderband eingeschaltet, wird also eine bewusst für die QoS Modelle unbekannte Situation geschaffen, wechselt die Beleuchtung des Schaltschranks auf Rot, sobald sich der Arm bewegt. Dies veranschaulicht eine fehlerhafte Kalibrierung, die beispielsweise durch die Änderung der Umgebung entstehen kann.

Im Anschluss veranschaulicht das Video das dynamische Betriebsführungskonzept wie in Abbildung 2 gezeigt. Der Wasm-Container, der auf dem Multisensor läuft, wird zum Edge Computer übertragen, um dort aktualisiert zu werden. Dieses Update kann durch maschinelles Lernen unterstützt werden. Während die neue Version generiert wird, bleibt die bisherige Container-Version auf dem Edge Computer weiterhin aktiv. Sobald das Update

abgeschlossen ist, wird der neue Container automatisch auf den Sensor zurückgespielt, ohne dass ein Neustart notwendig ist. Dadurch bleibt die QoS-Analyse während des gesamten Vorgangs erhalten und wird (nahezu) nahtlos auf unterschiedlichen Geräten fortgeführt.

Im Hauptteil des Videos wird ein Blick auf das QoS-Dashboard geworfen, das auf einem separaten Monitor die aktuellen Qualitätsparameter in Echtzeit visualisiert. Gleichzeitig ist zu sehen, dass der Schaltschrank weiterhin in Rot leuchtet, was den noch unzureichend kalibrierten Zustand der QoS-Modelle widerspiegelt. Zur Veranschaulichung wird nun der Updateprozess des WebAssembly-Containers manuell über die Konsole eingeleitet, während im Einsatzfall eine automatisierte Aktualisierung zu bevorzugen ist. In diesem Schritt wird der veraltete Container vom Sensor auf das Industrial Edge Gerät drahtlos übertragen und dort neu trainiert. Während die Übertragung, das neue Training des Containers und die anschließende Rückübertragung auf den Sensor ablaufen, leuchtet der Schaltschrank zur Statusanzeige durchgehend Blau. Damit wird signalisiert, dass sich das System in einem Updatezustand befindet.

Sobald der neue Container vollständig eingespielt und die QoS-Parameter erneut auf dem Sensor berechnet werden, befindet sich das Systemwieder in einem stabilen und kalibrierten Zustand, veranschaulicht durch die grüne Beleuchtung. Beim erneuten Bewegungsablauf des Positionierungsarms werden nun die aktualisierten QoS-Werte des neuen Modells übertragen, wodurch das laufende Förderband nicht mehr als Fehler interpretiert wird.

Abschließend zeigt das Video noch einmal das gesamte Setup des AFL mit seinen Komponenten, wie in Abbildung 3 gezeigt.

6 Wissenschaftlich-technische Arbeitsinhalte

6.1 Hardwarekomponenten (AP4)

Um sicherheitsrelevante (Security) bezogene Aspekte betrachten zu können, muss die Hardwareplattform entsprechende Komponenten aufweisen. Dies beinhaltet insbesondere gesonderte Hardware-Sicherheitsmodule (Trusted Execution Environment (TEE) oder Trusted Platform Module (TPM)) nötig. Sie ermöglichen eine Abkapselung von bestimmten Programmteilen, z.B. der Ver- und Entschlüsselung oder Zertifikatserstellung und bilden so einen vertrauenswürdigen Startpunkt (Root of Trust). Aufgrund nichtfunktionaler Anforderungen wie Hardwarekosten und Energieverbrauch wurde ein ARM Cortex M-33 Prozessor von STMicro als Evaluationsplattform ausgewählt (STM32U575).

6.2 Softwarekomponenten (AP4)

6.2.1 Echtzeitbetriebssystem

Echtzeitbetriebssysteme (Real-Time Operating Systems, RTOSes) bilden die softwaretechnische Grundlage der meisten Projekte für ressourcenbeschränkte Systeme. Sie abstrahieren die Hardware und verteilen Ressourcen, insbesondere Speicher und Rechenzeit. Außerdem stellen sie allgemeine Treiber zur Interaktion mit Peripheriegeräten zur Verfügung.

Da sie die unterste Softwareschicht bilden, ist die Berücksichtigung von Security hier essenziell. Fehlen entsprechende Mechanismen auf dieser Ebene, sind sie oft schwer oder gar nicht nachrüstbar. Diese Relevanz wird durch gesetzliche Bestimmungen betont, insbesondere im Cybersecurity Act [27]. Durch die Prinzipien „Security by Design“ und „Security by Default“ soll die Cybersecurity von Anfang an berücksichtigt werden, weshalb der Auswahl des RTOs erweiterte Aufmerksamkeit gewidmet wurde.

Dazu wurden verschiedene RTOSes verglichen. Zur Auswahl standen bekannte und weit verbreitete Betriebssysteme, namentlich Zephyr RTOS, FreeRTOS und Mentor Nucleus. Die Vergleichskategorien in Bezug auf Security wurden dabei ausgewählt durch den IEC62443-4-2 Standard. Daher wurde für den weiteren Projektverlauf Zephyr ausgewühlt. Dies begründet sich durch bessere Dokumentation, besseres Tooling, Offenlegung des Quellcodes und einer Nähe zu etablierten Linux-Betriebssysteme, was die Adaption von bestehendem Code einfacher macht. Des Weiteren ist die Anzahl an unterstützter Hardware besonders groß und Treiber werden als Industry-Grade angepriesen.

Tabelle 1 stellt die Bewertung der verschiedenen Anforderungen und deren Erfüllung durch das jeweilige Betriebssystem dar. Hieraus lässt sich schlussfolgern, dass alle der aufgeführten Kandidaten den meisten Anforderungen erfüllen. Insgesamt zeigt sich, dass sich die Kandidaten nur in Details unterscheiden.

Security Requirement according to IEC62443-4-2	Mentor Nucleus	FreeRTOS	Zephyr
Identification of human users (CR 1.1)	L.e.	L.e.	L.e.
Identification of devices (CR 1.2)	Yes*	Yes*	Yes
Strength of authentication through passwords (CR 1.7)	L.e.	L.e.	L.e.
PKI certificates (CR 1.8)	Yes*	Yes*	Yes
Unsuccessful login attempts (CR 1.11)	L.e.	L.e.	L.e.
Usage control of radio links (CR 2.2)	No	No	No
Mobile code (CR 2.4)	No	No	No
Auditable events (CR 2.8)	Yes	Yes	Yes
Storage capacity for event records (CR 2.9)	L.e.	L.e.	Yes
Behavior in the event of processing errors (CR 2.10)	No	No	No
Timestamps (CR 2.11)	Yes	Yes	Yes
Use of physical diagnosis and test interfaces (CR 2.13)	No	No	No
Protection against malicious code (CR 3.2)	Yes	Yes	Yes
Input validation (CR 3.5)	No	No	No
Predetermined states of the outputs (CR 3.6)	No	No	No
Protection of test information (CR 3.9)	No	No	No
Support for Updates (CR 3.10)	No	Yes	Yes
Physical tamper security and detection (CR 3.11)	Yes*	Yes*	Yes*
Provision of manufacturer trust anchors (CR 3.12)	Yes*	Yes*	Yes*
Provision of operator trust anchors (CR 3.13)	Yes*	Yes*	Yes*
Integrity of boot processes (CR 3.14)	Yes	Yes	Yes
Protection against denial of service events (CR 7.1)	Yes	Yes	Yes
Secure incident management process	No	No	Yes
Secure coding guideline	No	No	Yes

Legende: *: 3rd-Party Bibliothek, L.e.: Low-effort, < 4 Wochen Implementierung

Tabelle 1: Sicherheitsrelevante (Security) Aspekte der Echtzeitbetriebssysteme

6.2.2 Containerisierung als Grundlage für dynamische Betriebsführung

Zielausrichtung des Mannheim EMDRIVE Projekts für Siemens war die Untersuchung der dynamischen Betriebsführung, also eine dynamische Allokation von freien Geräten. Bevor diese Problemstellung erörtert werden kann, müssen Programme (Workloads) zunächst überhaupt auf verschiedenen Endgeräten ausführbar sein, ohne neu kompiliert werden zu müssen. Des Weiteren müssen sie unterbrechbar sein sowie zwischen Geräten einfach verschoben werden können. Da die ausgeführten Programme aus der Sicht des Endgeräts unbekannt sind, müssen sie außerdem als nicht vertrauenswürdig behandelt werden. Eine Schlüsseltechnologie, die diese Herausforderungen adressiert, ist die Containerisierung.

Ein Container bezeichnet dabei eine standardisierte Einheit, die eine Anwendung zusammen mit all ihren Abhängigkeiten in einem isolierten Bereich ausführt. Durch die Nutzung von Laufzeitumgebungen (runtime) oder Betriebssystem-Virtualisierungstechniken können Container ausgeführt werden, während sie gleichzeitig eine getrennte Umgebung für die Anwendung bereitstellen. Dies ermöglicht es, Anwendungen konsistent über verschiedene Umgebungen hinweg auszuführen, da die Container die gleichen Laufzeitbedingungen beibehalten, unabhängig vom zugrundeliegenden System. Eigenschaften von Containern sind unter anderem ihre Leichtgewichtigkeit im Vergleich zu traditionellen virtuellen Maschinen, da sie keinen vollständigen Gastbetriebssystem-Overhead mit sich bringen, sowie ihre Portabilität und Skalierbarkeit. Durch die Isolation bieten sie zudem Sicherheit und Stabilität, indem sie die Auswirkungen von Änderungen oder Fehlern auf den jeweiligen Container begrenzen.

Eingebettete Systeme (constrained devices, embedded systems), also Geräte mit eingeschränkten Ressourcen, stellen hier eine besondere Herausforderung für die Implementierung von Container-Technologien dar. Die erforderliche Laufzeitumgebung, die benötigt wird, um Container auszuführen, muss auf diesen Geräten besonders ressourcenschonend sein. Gleichzeitig bringen Container, trotz ihres vergleichsweisen geringen Overheads, immer noch eine gewisse Zusatzbelastung in Bezug auf Laufzeit, Varianz in der Ausführungszeit (Jitter) und Speicher mit, welche auf diesen Systemen nicht vernachlässigt werden kann.

Deshalb wurden verschiedene Mechanismen zur Isolation verglichen. Diese sind in Tabelle 2 abgebildet. Zur Auswahl standen WebAssembly (siehe Kapitel 4.4), eBPF, Java, MicroPython, LLVM Intermediate Representation (LLVMIR) und runc. eBPF ist ein Bytecodeformat, das im Linux-Kernel ausgeführt werden kann, um Programmteile zu beschleunigen. Java beinhaltet eine Programmiersprache sowie eine eigenes Bytecodeformat, welches durch die Java Virtual Machine (JVM) ausgeführt werden kann. Die Portierung der Python Programmiersprache und das Stellen einer entsprechenden Laufzeitumgebung wurde im MicroPython Projekt durchgeführt. LLVMIR ist eine Zwischensprache des LLVM-Compilerframeworks, wobei

ebenfalls Interpreter hierfür existieren. Runc ist die Container-Laufzeit auf der beispielsweise Docker beruht.

Technology	Isolation	Portability	Polyglot	Restrictions
WebAssembly	Type-safety, restricted memory	Good runtime support	LLVM-based	System interoperability
eBPF	Type-safety, validator	Very few runtimes	LLVM-based	Programming restrictions
Java	Type-safety, Runtime checks	Good runtime support	No	Licensing problems
MicroPython	None	Good runtime support	No	Unspecified and changing bytecode
LLVMIR	None	Very few runtimes	LLVM-based	Many opcodes, frequent changes
runc	cgroups, namespaces, seccomp	Good runtime support	Good tooling	Resource intensive

Tabelle 2: Vergleich verschiedener Isolationstechniken

Die wichtigsten, qualitativen Vergleichskriterien sind ebenfalls in Tabelle 2 dargestellt. Diese beinhalten die Isolation zwischen dem Hostsystem und dem Gastprogramm, die Portabilität, also die Möglichkeit ein Programm auf verschiedenen Hostsystemen auszuführen, die Polyglot-Eigenschaften, die beschreiben, wie gut sich der Container aus verschiedenen Programmiersprachen generieren lässt sowie Restriktionen, der die jeweilige Technologie unterliegt.

Prinzipiell zeigte sich, dass die Isolationseigenschaften zwar durch sehr unterschiedliche Technologien sichergestellt werden, diese jedoch entweder stark ausgeprägt und gut oder nicht vorhanden sind. Dies begründet sich durch die Zielsetzung der jeweiligen Technologie, die entweder Isolation anstrebt und somit gute Ergebnisse liefern muss, oder dies nicht verfolgt. Die Portabilität ist maßgeblich durch die Breite der Laufzeitumgebungen definiert. Sind viele vorhanden, die auf verschiedensten Plattformen agieren können, ist auch die Portabilität des Containers hoch. Hier zeigen sich positive Ergebnisse bei Wasm, MicroPython, und runc. Die Quellsprachunterstützung ist bei allen LLVM-basierten Technologien gut.

Der Hauptaspekt für eine weitere Eingrenzung sind jedoch die Restriktionen, die jede Technologie mit sich bringt. Eine Anwendung von runc konnte sofort ausgeschlossen werden, da keine Laufzeitumgebung für die ausgewählte Hardwareplattform (Cortex-M33, siehe Kapitel 6.1) existiert. Da außerdem eine Unix-Umgebung für die Isolationsmechanismen benötigt wird, ist eine Ausführung auf Mikrocontrollern momentan nicht möglich. LLVMIR sowie MicroPython sind aufgrund ihrer Bytecode-Struktur ebenfalls nicht geeignet. Beide dienen dazu, für ein sehr spezifisches Projekt Programmfluss darzustellen. Dies äußert sich in einer spezialisierten Form, die viele, für den Mikrocontrollerkontext unnötige Informationen

enthält, instabil ist und/oder viel Speicher benötigt. Des Weiteren ändert sich der Bytecode oft und ist teilweise unspezifiziert, was eine dauerhafte und verlässliche Nutzung im industriellen Kontext unmöglich macht. Die Nutzung von Java ist aufgrund lizenztechnischer Probleme weitgehend eingeschlafen, weshalb sich sowohl die Nutzung an sich als auch die Zukunftssicherheit sowie die Unterstützung als problematisch gestalten.

eBPF und WebAssembly zeigten sich zunächst als sehr gute Kandidaten, weshalb weitere Untersuchungen durchgeführt wurden. Abbildung 5 zeigt den Vergleich von Ausführungszeiten verschiedener CRC-Algorithmen, die in typischen Anwendungen in der drahtlosen Sensorkommunikation verwendet werden. Die Laufzeiten zeigen Standardabweichungen deutlich unter einem Prozent, was auf gute Jitter-Eigenschaften schließen lässt. Die Ausführungszeiten der Algorithmen dargestellt in Wasm sind mit 32 ms ungefähr 20% langsamer als die mit rBPF dargestellten. Allerdings sind die quantitativen Einblicke dieser Messung nur von nebensächlicher Bedeutung, was die Auswahl eines Container Kandidaten anbelangt.

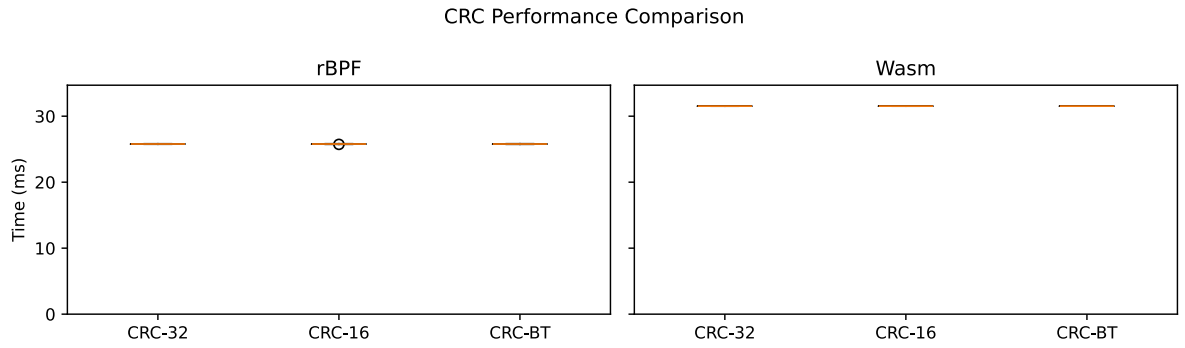

Abbildung 5: Laufzeitvergleich rBPF und Wasm Bytecodes

Zielführender in Hinblick auf die Auswahl einer Containertechnologie für Mikrocontroller war der Testaufbauprozess. Bei der Kompilierung in das eBPF-Bytecodeformat traten unerwartete Komplikationen auf. Beispielsweise unterstützt das eBPF-Format keine Gleitkommazahlen (floating point units nach IEEE 754), da diese zu Indeterminismus führen können. Diese sind jedoch häufig in der Sensorik anzutreffen. Des Weiteren ergaben sich große Probleme mit Funktionsaufrufen und Rekursion, was die Programmierung erheblich erschwert. Laufzeitumgebungen waren ebenso wenig zu finden. Außerdem müssen standardmäßig alle Schleifen (for, while) statische obere Grenzen haben. Auch dies ist nicht immer gegeben. Lediglich die rBPF [28] Implementierung spezialisiert für Mikrocontroller konnte ausgeführt werden. Diese beschränkt sich auf einen Interpreter. Im Gegensatz hierzu steht Wasm, das durch aktiv entwickelte Werkzeuge deutlich einfacher handhabbar ist, keine Einschränkungen an die Semantik hat sowie einen interpretierten, einen Just-in-Time und einen Ahead-of-Time kompilierten Modus anbietet (siehe Kapitel 4.4). Infolgedessen beschränkt sich das Projekt im Folgenden auf WebAssembly als Containerisierungstechnologie und die Anwendung des

zugehörigen Bytecodes.

Runtime	Creator / maintainer	Supports AoT	Supports JIT	Supports Embedded	Written in
WAMR	Bytecode Alliance	Yes	Yes	Yes	C
wasm3	*unknown*	No	No	Yes	C
wasmi	Parity	No	No	Yes	Rust
wasmtime	Bytecode Alliance	Yes	Yes	No	C
wasmer	Wasmer	Yes	Yes	No	Rust

Tabelle 3: Vergleich verschiedener Wasm Laufzeitumgebungen

Tabelle 3 zeigt populäre Laufzeitumgebungen zur Zeit des Projektstarts. Diese beinhalten die WebAssembly Micro Runtime (WAMR), wasm3, wasmtime und wasmer. Die verschiedenen Lösungen wurden hinsichtlich ihres Funktionalitätsumfangs und ihrer Eignung für beschränkte Systeme ausgewählt. Hier bietet WAMR die beste Option. Unterhalten durch die Bytecode Alliance dient WAMR neben wasmtime als Referenzimplementierung für den WebAssembly Standard. Des Weiteren wurde WAMR mit Fokus auf Konfigurierbarkeit, kleinen Fußabdruck und Geschwindigkeit implementiert. Sie bietet einen Interpreter, einen Just-In-Time-Compiler sowie eine Ahead-of-Time Compiler und somit den vollen Funktionsumfang, wie Wasm Bytecode ausgeführt werden kann. Um sich Flexibilität im Fortlaufen des Projekts offenzuhalten, wurde sich daher für diese Laufzeitumgebung entschieden.

6.2.3 WASM-IO: Hardwareinteraktion in Containern

Mikrocontrollersysteme werde in der Regel eingesetzt, um mit der physikalischen Welt zu interagieren, also Sensorik oder Aktorik zu steuern. Dies erfolgt meist durch Treiber im Betriebssystem. Da Containerisierung jedoch darauf aufbaut, die Interaktion mit dem Host-System zu minimieren, ist ein Nutzen von Host-Treibern nicht ohne weiteres möglich. Insbesondere in Wasm sind hierzu zwar Konzepte angedacht, jedoch noch nicht generalisiert und spezifiziert. Dies zeigt sich auch in Tabelle 2 in der limitierten Systeminteraktion für Wasm. Des Weiteren beinhaltet dies nur Treiber, die bereits im System vorhanden sind. Um eine tatsächliche dynamische Betriebsführung und somit eine dynamische Adaption zu ermöglichen, ist dies jedoch nicht ausreichend. Ändert sich die Ausführungsplattform durch ein Verschieben von Programmen zwischen zwei Geräten, so ist die Hardwarekonfiguration unterschiedlich. Ein Kernaspekt der erarbeiteten Ergebnisse in EMDRIVE war daher die Adressierung dieses Problems, welche im Entwurf von WASM-IO mündet. WASM-IO stellt ein Framework zur Verfügung, welches Peripherie in Wasm zugreifbar macht. Die folgenden

Inhalte wurden in einem Preprint [29] und einem Konferenzbeitrag sowie Journalartikel [30] veröffentlicht. Das folgende Kapitel beruht maßgeblich auf diesen Inhalten.

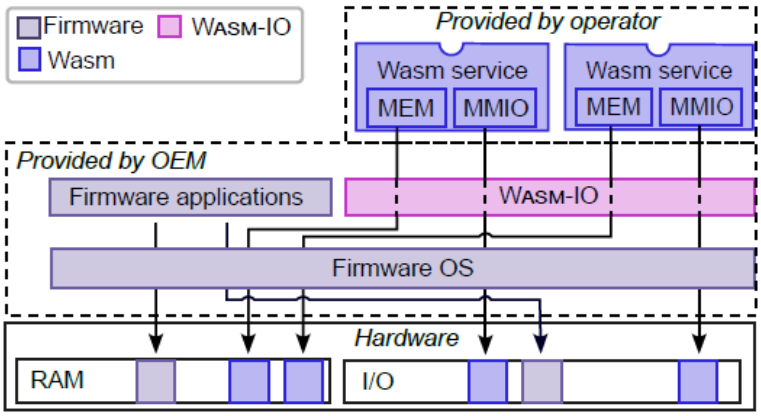

Abbildung 6: Architektur von Wasm-IO

Architektur. Die Architektur von Wasm-IO basiert auf Originalgeräteherstellern (OEMs) von welchen eingebetteten Geräten wie beispielsweise Sensoren produziert werden. Initial stellen OEMs eine Firmware bereit, die ein Betriebssystem und Firmware-Anwendungen bereitstellen, wobei letztere die anfängliche Logik beinhalten, die vom Gerät ausgeführt wird. Jede Softwarekomponente wird mit zeitlichen Schranken geliefert und unterliegt einer statischen Verifikation und Zertifizierung.

Um die Geräte funktional zu erweitern, können Betreiber (operators) Dienste (services) laden, die nicht Teil der Firmware sind. Daher muss der Code dieser Dienste als nicht vertrauenswürdig betrachtet werden. Das Ausführen dieses Codes wird durch Wasm-IO, einer Zwischenschicht auf Kernel-Ebene, ermöglicht. Die inhärenten Sandboxing-Eigenschaften von Wasm werden dabei verwendet, wobei eine Wasm-Laufzeitumgebung durch Wasm-IO bereitgestellt wird. Da die Laufzeitumgebung Teil der vertrauenswürdigen Computing-Basis ist, nehmen wir ihre formale Korrektheit an. Dies ermöglicht es, Services im Kernel-Modus auszuführen, ohne das Systems zu beeinflussen.

Die zuvor genannten Dienste werden als Wasm-Binärdateien bereitgestellt und im Folgenden als Wasm-Services bezeichnet. Jeder Wasm-Service umfasst Geschäftslogik, Treiberlogik oder beides. Er ist single-threaded und von anderen Wasm-Services sowie dem Betriebssystem isoliert. Ein Betriebssystem-Thread führt die Wasm-Laufzeitumgebung aus, die wiederum das Wasm-Binärprogramm ausführt. Die Laufzeitumgebung stellt temporäre, isolierte RAM-Bereiche bereit, in denen der Wasm-Service operieren kann. Wasm-IO weist jedem Dienst bei Bedarf exklusiv Memory-Mapped I/O (MMIO)-Regionen zu. MMIO ist eine Technik, bei der Hardware-Peripheriegeräte durch Zuweisung von Speicheradressen auf ihre Register zugegriffen wird, wodurch ein direkter Zugriff im Adressraum des Prozessors ermöglicht wird.

Beim Kompilieren eines Quellprogramms nach Wasm bleibt der Kontrollfluss erhalten. Strukturen, wie beispielsweise Treiberregister, werden beibehalten, wenn der Adresswert dem Compiler unbekannt ist, um eine Konstantenfaltung zu verhindern. Dies wird durch die Methode zur Spezifikation von Peripherieanforderungen erreicht, welche später vorgestellt wird.

Allerdings gibt es einige kleinere Unterschiede in der Semantik. Insbesondere fehlt in Wasm ein Mechanismus zur Darstellung von volatilen Variablen sowie Prinzipien der Speicher-Konsistenz. Beides sind Konzepte, die in C/C++-basierten Treibern üblich sind. Dies stellt jedoch keine Einschränkung dar. Compileroptimierungen werden auf einer Zwischenrepräsentation durchgeführt, typischerweise der LLVM Intermediate Representation (LLVM IR), bevor der Code nach Wasm konvertiert wird. Folglich existieren die Konzepte von Volatilität und Speicher-Konsistenz noch. Da Interpreter das generierte Wasm-Binary direkt nutzen, bleibt die Reihenfolge der Speicherzugriffe erhalten. Für nachfolgende Kompilierungsprozesse, wie Just-in-Time (JIT) und Ahead-of-Time (AoT) Kompilierung, kann das volatile Verhalten durch die Verwendung von atomaren Zugriffen emuliert werden [31], obwohl diese noch standardisiert werden müssen und keine semantisch gleichwertige Operation darstellen.

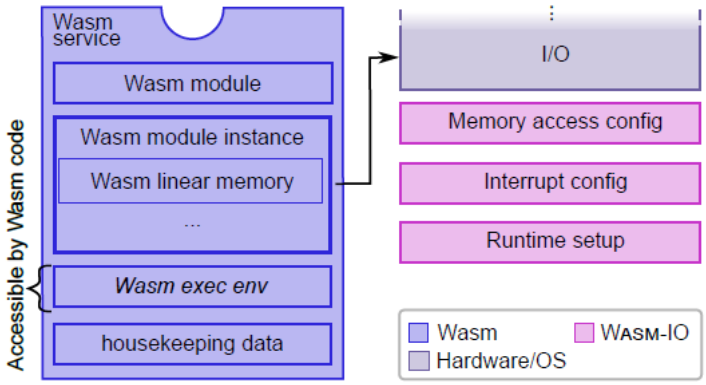

Abbildung 7: Speicherarchitektur von Wasm-IO

Die relevanten Teile des Aufbaus von Wasm-IO sind in Abbildung 7 dargestellt. Das Runtime-Setup wird von Wasm-IO genutzt, um den Initialisierungsprozess der Wasm-Laufzeitumgebung zu verwalten. Jeder Service besteht aus den durch WAMR definierten Bestandteilen module, module instance und execution environment (siehe auch Kapitel 4.4) und einigen Verwaltungsdaten. Die memory access configuration beinhaltet die Zuweisung von MMIO-Regionen auf einen Wasm-Service.

Die interrupt config ermöglicht asynchrone Interaktionen, also die Behandlung von Interrupts. Der nicht vertrauenswürdige Code des Operators, der als Wasm-Binärdateien bereitgestellt wird, kann nur auf Speicherbereiche innerhalb des execution environment

zuzugreifen. Insbesondere sind die Wasm-Services nicht in der Lage, die Konfigurationen der Peripheriegeräte zu modifizieren.

Synchrones I/O. Um von verschiedenen Wasm-Services auf verschiedene Peripheriegeräte zuzugreifen, müssen mehrere Teilprobleme gelöst werden: (a) Die Sicherstellung der Verfügbarkeit der vom Wasm-Service benötigten Peripheriegeräte auf dem jeweiligen Endgerät, (b) ihre hardwareunabhängige Einbindung und (c) die Nutzung innerhalb des Wasm-Services entsprechend des Treiberlayouts.

a) *Sicherstellung der Verfügbarkeit*: Wasm-IO fordert vom OEM bereits in der Designphase eine explizite Erlaubnis zur Nutzung eines Peripheriegeräts innerhalb eines Wasm-Services. Dies gewährleistet die Trennung zwischen Firmware-Treibern und Treibern der Wasm-Services. Somit wird sichergestellt, dass Peripheriegeräte jederzeit exklusiv zugewiesen sind.
   Hierfür wurde ein neues Devicetree-Binding eingeführt. Während der Firmware-Entwicklung müssen OEMs alle Knoten identifizieren, beispielsweise ungenutzte Busse oder universelle digitale Ein-/Ausgänge (GPIOs), an die Peripheriegeräte angeschlossen werden können. Diese Knoten werden explizit in einem Knoten des neuen Bindings referenziert. Ein Build-Tool generiert daraufhin eindeutige, unveränderliche Namen, die aus dem unveränderlichen Pfad oder dem Label der Knoten abgeleitet sin. Zusammen mit ausgewählten Eigenschaften wie Interrupt-Nummern und Registeradressen bilden diese Namen den statischen Teil der memory access configuration. Dabei sollten Knotenlabels wie „spi1“ anstelle vollständiger Pfade zu verwenden, um die plattformübergreifende Kompatibilität zu verbessern.
b) *Plattformunabhängige Einbindung*: Ein Treiber für ein neues Peripheriegerät wird typischerweise gegen eine Hardware-Abstraktionsschicht (Hardware Abstraction Layer, HAL) programmiert. Die HAL dient dazu, hardwareabhängige Details zu verbergen und stellt Platzhalter für jede Registeradresse bereit, um eine plattformunabhängige Entwicklung der Treiberlogik zu ermöglichen. Diese Platzhalter werden zur Compile-Zeit mithilfe einer spezifischen Konfiguration für die jeweilige Hardwareplattform aufgelöst. Das resultierende Binärprogramm enthält folglich nur physische Adressen.
   Wasm-IO baut auf diesem bewährten Ansatz auf. Anstatt jedoch die Platzhalter direkt auf physische Adressen aufzulösen, führt Wasm-IO für jeden Platzhalter Dummy-Register ein, die mit einem eindeutigen Registerlabel verknüpft sind. Diese Dummy-Register verbleiben im kompilierten Wasm-Binärprogramm, wenn ein Wasm-Service erstellt wird.
c) *Verwendung innerhalb Wasm*: Beim Laden des Wasm-Services wird jedes Dummy-Register mithilfe der memory acces configuration auf eine physische Adresse abgebildet. Um die physikalische Adresse herleiten zu können, wird ein Verfahren

analog zum Ausdrücken der verfügbaren Register verwendet. Neuen Peripherie kann virtuell an die unveränderlichen Elemente im Devicetree angefügt werden. Das Build-Tool generiert dann konsistent die gleichen Labels. Diese Labels werden anschließend in der custom section des Wasm-Services als Abhängigkeiten eingebettet. Zur Ladezeit können diese im Wasm-Service mit dem statischen Anteil der memory access configuration abgeglichen werden, um die physischen Adressen oder Interrupt-Nummern bereitzustellen.

Beim Laden eines Wasm-Services überprüft Wasm-IO, ob alle vom Service angeforderten Register bereitgestellt werden. Sollte eine der angeforderten Abhängigkeiten nicht verfügbar sein, wird der Wasm-Dienst abgelehnt, um potenzielle Laufzeitfehler zu verhindern.

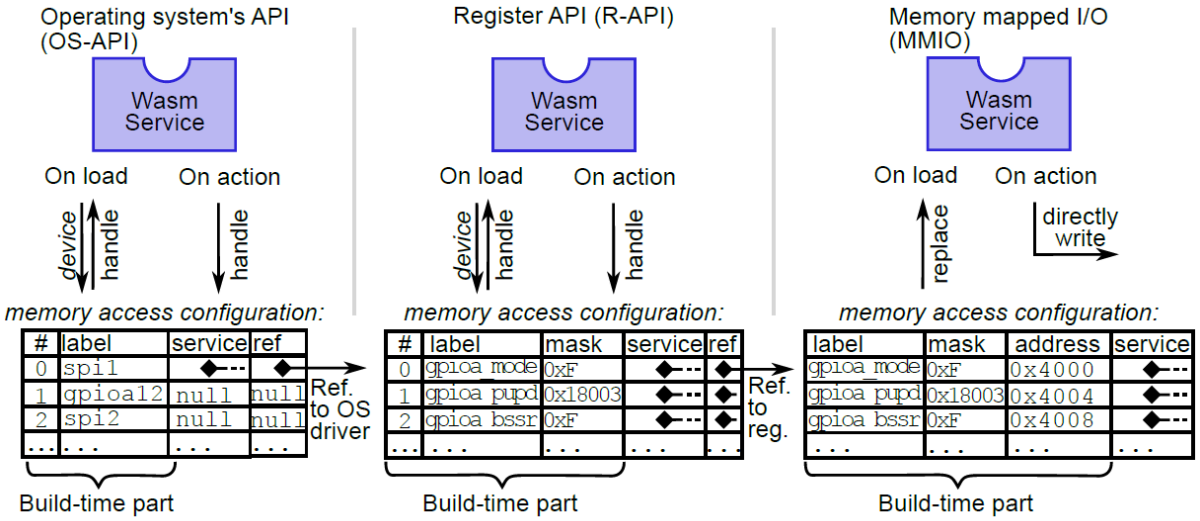

Abbildung 8: Verschiedene Peripheriezugriffsmethoden implementiert in Wasm-IO

Da Unterstützung für die Interaktion mit Peripheriegeräten in Wasm fehlt, bietet Wasm-IO eine Referenz- und vier neuartige Methoden für synchronen Peripheriezugriff: OS-API, R-API, MMIO sowie R-API und MMIO mit Beschleunigung durch Direct Memory Access (DMA). Diese werden im Folgenden detailliert erläutert.

Die Betriebssystem-API (OS-API) orientiert sich an bestehender Literatur [32], [33], [34] und dient als Referenzgrundlage, um einem Wasm-Service den Zugriff auf die Peripherie-Treiber-API des Betriebssystems zu ermöglichen. Um diese API zu nutzen, muss der Wasm-Service gegen die bestehende Betriebssystem-Treiber-API programmiert werden. Geräteabhängigkeiten, die im Wasm-Binärcode eingebettet sind, verweisen auf den Devicetree-Knoten statt auf dessen Register, um die Existenz des Geräts sicherzustellen. Beim Laden fordert der Wasm-Service einen Geräte-Handle an, der der gewünschten Treiberinstanz entspricht, anstatt Dummy-Register aufzulösen. Diese Methode wird auf der

linken Seite von Abbildung 8 dargestellt. Wird die Anfrage genehmigt, wird der Handle in der memory access configuration gespeichert.

Zur Laufzeit ruft der Wasm-Service für jede I/O-Aktion die interne Betriebssystem-API auf, beispielsweise `spi_write(...)`. Wasm-IO überprüft anhand der memory access configuration, ob der Service die notwendigen Berechtigungen für den Handle besitzt. Wenn der Zugriff gewährt wird, wird die Anfrage an den entsprechenden Treiber weitergeleitet. Da dieser Ansatz vollständig auf dem betriebssystemspezifischen Treiber und dessen Konfiguration beruht, erlaubt die OS-API keinen beliebigen Peripheriezugriff. Daher dient sie lediglich als Referenz.

Um den Zugriff auf beliebige Peripheriegeräte zu ermöglichen, wurde die neuartige Register-API (R-API) eingeführt. Dieser Ansatz verfeinert das vorgestellte Konzept einer API und erlaubt die direkte Manipulation von Geräte-Registern. Im Gegensatz zur OS-API, bei der jeder Geräte-Handle einem Betriebssystemtreiber entspricht, repräsentiert in der R-API jeder Handle direkt mit ein spezifisches Register. Dadurch kann ein Service mit jedem Gerät interagieren, sofern die entsprechenden Registeradressen bekannt und bereitgestellt sind.

Die erforderlichen Adressen werden mithilfe von Dummy-Register-Labels ermittelt, wie in diesem Kapitel beschrieben. Die Interaktion mit einem Register erfolgt somit durch den Aufruf einer Funktion statt durch die Dereferenzierung eines Zeigers, wie es in C-ähnlichen Sprachen üblich ist. Dieses Design ermöglicht es Wasm-IO, jeden Registerzugriff zu überprüfen, indem die memory access configuration überprüft wird. Zudem erlaubt es die Nutzung von Programmiersprachen, die kein Referenzkonzept haben. Diese Methode wird in der Mitte von Abbildung 8 veranschaulicht.

Für eine feingranulare Kontrolle wendet Wasm-IO Bitmasken auf die Werte vor dem Zugriff an, beispielsweise beim Manipulieren von GPIO-Registern, da diese Zugriff auf mehrere I/O-Pins bieten. Diese Masken werden während des Firmware-Build-Prozesses aus dem initialen devicetree bezogen und zusammen mit den Registeradressen in der memory access configuration gespeichert. Die Register sind nach ihrer Nutzungsfrequenz sortiert. In einem der spezifischen Evaluationsfälle, in dem Wasm-IO im Usermode (untrusted scenario) arbeitet, ist der API-Aufruf einem Systemaufruf gleichzusetzen.

Wie die spätere Evaluierung zeigt, weisen sowohl die OS-API als auch die R-API beträchtliche Latenzen auf. Um dieses Problem zu adressieren, wurde eine deutlich schnellere Variante entwickelt, indem echtes Memory-Mapped I/O (MMIO) in Wasm ermöglicht wird. Hierfür wurden die Speicher- und Ladeoperationen innerhalb der Wasm-Laufzeitumgebung modifiziert.

Die Wasm Runtime führt bei der Ausführung einer Speicher- oder Ladeanweisung eine Prüfung der Adresse durch. Da I/O-Registeradressen (als nicht-vorzeichenbehaftete

Ganzzahlen interpretiert) außerhalb des linearen Speichers liegen, würde diese Prüfung zwangsläufig zu einem Fehler führen. Im Fehlerfall ermittelt Wasm-IO, ob die Adresse einer freigegebenen Registeradresse entspricht, und genehmigt gegebenenfalls die entsprechende Anweisung. Dieser Ansatz wird auf der rechten Seite von Abbildung Abbildung 8 dargestellt. Ähnlich wie beim R-API-Ansatz verwendet die MMIO-Methode ebenfalls Masken, um den Zugriff auf bestimmte Bits innerhalb eines Registers einzuschränken. Im Usermode nutzt Wasm-IO einen Systemaufruf, um von der Registeradresse zu lesen oder sie zu ändern.

Wie in der Evaluation gezeigt, ist die Hauptursache für erhöhte I/O-Latenzen in jedem der Ansätze der Kontextwechsel (ggf. Userspace, nach Wasm, nach Kernel). Um dieses Problem zu lösen, wird der DMA-Controller (Direct Memory Access) verwendet. Zudem wurde das Konzept des Conveyor-Memory eingeführt, der virtuell an den linearen Speicher des Wasm-Moduls angefügt wird. Die standardmäßige Wasm-Zugriffsprüfung wurde modifiziert, sodass sie gegen eine verschobene Grenze testet, die sowohl den Wasm-linearen Speicher als auch das Conveyor-Memory umfasst.

Beim Auflösen von Register-Labels zu ihren physischen Adressen schreibt Wasm-IO diese Adressen in die DMA-Konfiguration, anstatt die Dummy-Register zu ersetzen. Die Dummy-Register werden hingegen auf Adressen innerhalb des Conveyor-Memory abgebildet, was sicherstellt, dass jeder Speicherzugriff des Wasm-Service innerhalb seines zugänglichen Speicherbereichs bleibt. Die indirekte DMA-Konfiguration verhindert DMA-Angriffe.

Während der Ausführung von Wasm pollt der DMA-Controller kontinuierlich die Register und den Conveyor-Memory und aktualisiert alle Änderungen. Da der DMA-Controller im privilegierten Modus arbeitet, kann er sowohl auf dem MMIO Bereich im Kernel als auch auf den Wasm Bereich, potenziell im Usermode, zugreifen. Der inhärente Overhead der Wasm-Interpretation mit mehreren Zyklen pro Wasm-Anweisung sorgt dafür, dass die Latenzen, die durch die Aktualisierung durch den DMA-Controller entstehen, während der Laufzeit keinen Einfluss haben.

Asynchrones I/O (Interrupts). Wie in Kapitel 4.1 beschrieben, wird in modernen Betriebssystemen die Interrupt-Verarbeitung in zwei Phasen unterteilt: Die Prolog-Phase ermöglicht die Kommunikation mit dem Gerät, d.h. den Empfang von Daten, während die Epilog-Phase die eigentliche Datenverarbeitung durchführt.

Entsprechend werden Wasm-Services auf der Ebene E_0 ausgeführt, was ein Unterbrechen ihrer Ausführung ermöglicht, während Prolog oder Epilog auf den Ebenen E_1 bzw. $E_{1/2}$ ausgeführt werden. Wird dieses Prolog-Epilog-Modell jedoch naiv auf das System angewendet, so ergeben sich erhebliche Isolationsschwachstellen, da der Wasm-Service beliebigen Code ausführen kann. Nicht vertrauenswürdiger Code wird im Prolog mit der höchsten Priorität und deaktivierten Interrupts ausgeführt. Dadurch führt jeder Programmfehler unvermeidlich zu einem Geräteausfall, beispielsweise erfordert eine Division

durch Null bei deaktivierten Interrupts externe Hardware für einen Reset. Zudem wird der Systemdeterminismus durch die ungewisse Ausführungszeit und -beendigung aufgehoben.

Um dieses Problem zu lösen, bietet Wasm-IO einen allgemeinen Wasm-IO-Prolog an, der Daten vom Gerät anstelle des Wasm-Services abruft. Um plattformunabhängige Interrupts zu ermöglichen, gelten dieselben Konzepte wie für die bereitgestellten Register. OEMs definieren während der Designphase die Methode zum Quittieren von Interrupt-Flags. Wasm-Services können sich registrieren, um Interrupt-Anfragen (IRQs) zu empfangen, indem sie ein Interrupt-Label zusammen mit einer Prioritätsstufe angeben.

Dies behebt die im Prolog inhärente Schwachstelle. Allerdings können mehrere Peripheriegeräte eine einzelne Interrupt-Leitung teilen, was zur Ausführung aller mit dieser Leitung verbundenen Wasm-Epiloge führt, unabhängig ob diese für die jeweiligen Peripheriegeräte berechtigt sind oder überhaupt ausgeführt werden sollen. Darüber hinaus muss die Ausführung von Firmware-Epilogen Vorrang vor Wasm-Epilogen haben. Daher wird durch den Wasm-IO-Epilog eine zusätzliche vertrauenswürdige Instanz bereitgestellt, die diese Filterung und Priorisierung durchsetzt. Der Wasm-IO Epilog arbeitet auf Ebene $E_{1/2}$, wobei die Wasm-Epiloge auf eine neue Prioritätsebene $E_{1/4}$ verschoben wurden, die zwischen dem Wasm-IO-Epilog und dem Wasm-Service liegt. Diese Änderung resultiert in dem neuen Prolog-Epilog-Modell von Wasm-IO. Die Wasm-IO-Epiloge können den Wasm-Epilogen übergeordnet sein und somit Kontrolle über die Wasm-Epiloge ausüben. Firmware-Epiloge verbleiben weiterhin auf Ebene $E_{1/2}$.

Um den Datenempfang auf Prolog-Ebene zu ermöglichen, kann ein Wasm-Service Daten registrieren, die zusammen mit dem entsprechenden Interrupt empfangen werden sollen. Hat der Wasm-Service die entsprechende Berechtigung zum Zugriff auf diese Daten, werden sie gepuffert. Vor der Ausführung eines Wasm-Epilogs kopiert der Wasm-IO-Epilog die Daten in den linearen Speicher des Wasm-Services, wobei die Dummy-Register entsprechend modifiziert werden. Da der Wasm-Service dann auf dieser Kopie arbeitet, entfällt die Notwendigkeit für den Wasm-Epilog, eine Ebene unterhalb von $E_{1/4}$ zu betreten, um die Daten abzurufen. Dies führt maßgeblich dazu, die Interrupt-Latenzen für die Wasm-Services zu reduzieren, insbesondere bei aktiviertem User-Modus. Programmatischer Datenempfang muss jedoch im Wasm-Epilog erfolgen.

Darüber hinaus erleichtert dieser Ansatz mit dem überarbeiteten Datenmodell eine zeitnahe und deterministische Interrupt-Synchronisation. Ein weiterer Vorteil der Beschränkung eines Wasm-Services auf die Ebenen E_0 und $E_{1/4}$ ist die Vermeidung einer expliziten Interrupt-Synchronisation, wie in Kapitel 4.1 beschrieben.

Der Interruptmechanismus von Wasm-IO wird erreicht, indem der Wasm-IO-Prolog direkt in die Interrupt-Vektortabelle geschrieben wird. Beim Eintreffen eines Interrupts sequenzierte der Wasm-IO-Prolog die IRQs, indem er sie in die Epilog-Warteschlange einfügte. Der

höchstpriore Betriebssystem-Thread wartete auf Einträge in dieser Warteschlange. Anschließend puffert der Wasm-IO-Epilog die Daten im Interrupt-Datenpuffer und ruft die Wasm-Epiloge in der Reihenfolge ihrer Priorität auf.

Beim Registrieren eines Interrupts gibt ein Wasm-Service den Index der entsprechenden Funktion in der internen Funktions-Tabelle von Wasm über die von Wasm-IO bereitgestellte API an. Zur Verwendung des Datenkopiermechanismus zur Minimierung der Latenz muss außerdem die Datenquelle definiert werden, aus der kopiert werden sollte. Für die R-API musste auch der Zielort im linearen Speicher zusammen mit der Datenquelle angegeben werden. Der Wasm-IO-Epilog kopierte die Daten dann direkt an diesen angegebenen Ort. Bei MMIO wurden die Daten in einen Puffer kopiert. Dieser Puffer befindet sich im Benutzerraum, wenn sich Wasm-IO im Benutzerraum befindet.

Jedes Mal, wenn die Wasm-Epiloge aufgerufen wurden, wird für jede Wasm-Modulinstanz eine neue Ausführungsumgebung erstellt. Dadurch bleibt der Kontext des aktuellen Wasm-Services erhalten, während globale Daten über die Wasm-Modulinstanz gemeinsam genutzt werden können, einschließlich globaler Variablen und linearem Speicher. Infolgedessen arbeitete der ausführende Thread außerhalb von Wasm.

Evaluation. Um Wasm-IO evaluieren, wurden Overheads und Worst-Case-Ausführungszeiten (Worst-Case Execution Times, WCETs) bestimmt. Overheads erlauben eine Bewertung in Bezug auf die Praktikabilität und Anwendbarkeit des Ansatzes. Die Analyse oberer Schranken für Ausführungszeiten ermöglichen qualitative Aussagen bezüglich des Determinismus im Gesamtsystem.

Die Evaluation erfolgt durch externe Messung der längsten beobachteten Ausführungszeiten sowie der Aufzeichnung der entsprechenden Instruktionspfade. Durch die Kombination dieser Pfade mit dem Quell- und Maschinencode wird die notwendige Worst-Case-Pfadkomponente für die WCET-Analyse, im Folgenden bezeichnet als Instruktionsanzahl, ermittelt. In Verbindung mit einem umfassenden Kostenmodell können diese Instruktionsanzahlen zur Bestimmung der WCETs genutzt werden. Der Fokus lag dabei auf der Analyse von Schleifengrenzen und der Abschätzung des Echtzeitverhaltens sowie dessen Skalierung basierend auf diesen Grenzen unter Einbeziehung der Zeitmessungen.

Es werden zwei Szenarien evaluiert. Im ersten Szenario wird angenommen, dass der Interpreter vertrauenswürdig ist, beispielsweise durch statische Verifikation. In diesem Fall wird die Ausführung von Wasm-Services, die von der im Kernel operierenden Wasm-IO runtime interpretiert werden, mit traditionellen Isolationstechniken verglichen. Speziell erfolgt ein Vergleich mit der Adressraumtrennung unter Verwendung des Usermodes von Zephyr, der die Memory Protection Unit als Hardware-Schutzmechanismus einsetzt.

Im zweiten Szenario wird die Laufzeitumgebung nicht als vertrauenswürdig angesehen. Es ist naheliegend, dass die Ausführung von Programmen in einer Umgebung, die sowohl durch Wasm als auch durch den Usermode isoliert ist, zu suboptimaler Leistung führt im Vergleich zur Ausführung ausschließlich im Usermode. Daher wird die Ausführung im Usermode verglichen, wobei speziell der Einfluss der DMA-Beschleunigung, sowohl mit als auch ohne deren Verwendung, evaluiert wird.

Die Evaluierung wurde auf einem STM32WL55JC-Evaluationsboard durchgeführt, das über eine Cortex-M4-CPU verfügt, die mit 48 MHz arbeitet. Das Board ist mit 64 KiB SRAM ausgestattet. Die Peripheriebusse arbeiten ebenfalls mit 48 MHz. Um mögliche Beeinflussung durch die Messmethode zu vermeiden, wird ein Saleae Logic Pro 16 Logic Analyzer mit einer Abtastrate von 250 MHz für externe Messungen verwendet, anstatt Softwareinstrumentierung zu verwenden.

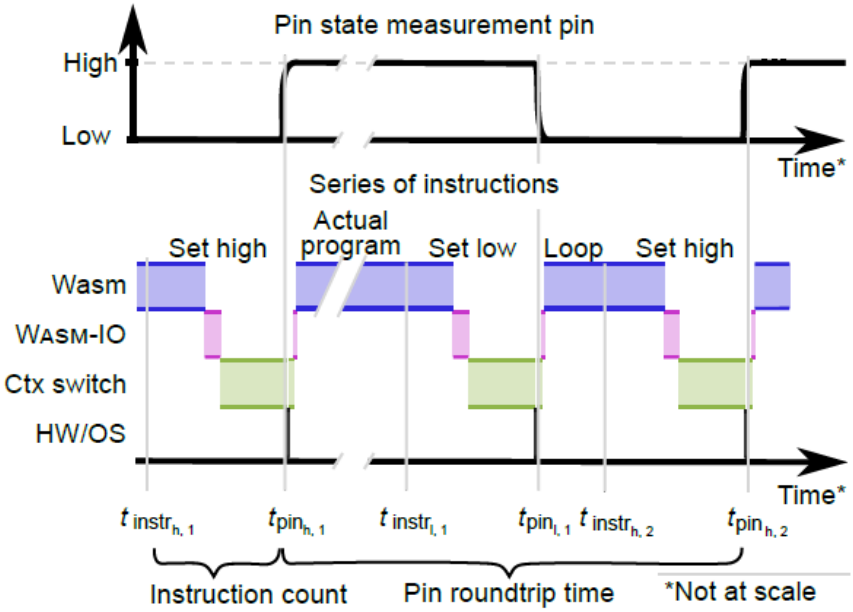

Abbildung 9: Messvorgang zur Bestimmung der zeitlichen Kosten für synchrones I/O

Es wird die Umlaufzeit (roundtrip time) gemessen, die erforderlich ist, um einen GPIO-Pin (den measurement pin) zu aktiv und anschließend inaktiv zu setzen, um die mit synchronen Peripherieinteraktionen verbundenen Kosten zu analysieren. Der Messvorgang wird in Abbildung 9 illustriert. Der Wasm-Service führt die Softwareinstruktionen auf Wasm-Ebene aus, um den Pin zu ändern, z. B. durch Aufruf der R-API oder durch Schreiben in eine Speicheradresse. Folglich wird der Wasm-Kontext verlassen, abhängig vom gewählten Ansatz. Wasm-IO verifiziert den Speicherzugriff und wechselt bei Erfolg in das Betriebssystem (Kernel). Falls die Wasm-IO runtime nicht vertrauenswürdig ist und Wasm-IO im Usermode arbeitet, erfolgt der Wechsel über einen Systemaufruf. Anschließend erfolgt die Manipulation des Registers zunächst auf Softwareebene und dann auf Hardwareebene. Danach kehrt jede

aufgerufene Funktion zurück. Dieser Rückkehrprozess ist nicht vernachlässigbar, insbesondere auf der Betriebssystemebene (im Usermode) und der Wasm Ebene, da umfangreiches Kopieren vorgenommen wird. Das Setzen des Pins auf den inaktiven Zustand folgt dem gleichen Verfahren.

Die folgende Messung konzentriert sich auf die Dauer zwischen der ersten I/O-Instruktion in Wasm und dem extern beobachtbaren Ereignis, also der Signalflanke. Das actual program in Abbildung 9 wird daher ausgelassen (NOOP). Es werden nur die WOETs (Worst Observed Execution Times) präsentiert. Wichtig zu beachten ist, dass die gemessenen Instruktionsanzahlen nicht direkt den Zeitmessungen entsprechen. Obwohl sie eine detailliertere Kategorisierung der Instruktionen bieten, fehlen ihnen Rückkehrpfade und Hardwarelatenzen.

Die Ausführung von synchronen I/O-Interaktionen unter Verwendung der Zephyr-API nativ im Kernel ergab eine Zeit von 4 µs. Da jedoch die Ausführung von nicht vertrauenswürdigem Code im Kernel keinerlei dynamischer Betriebsführung entspricht und somit nicht Ziel des Vorhabens war, wird dies von weiteren Diskussionen ausgeschlossen.

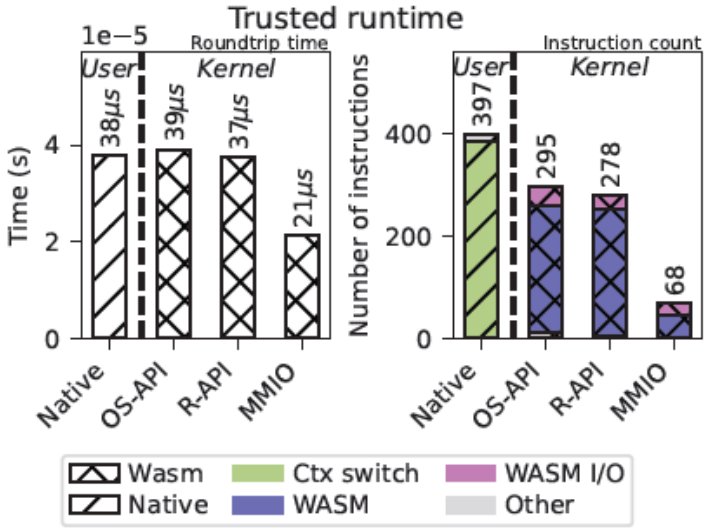

Abbildung 10: Ausführung von synchronem I/O mit Wasm-IO im Kernel

Vertrauenswürdige Laufzeitumgebung. Abbildung 10 zeigt die Ergebnisse einer vertrauenswürdigen Laufzeitumgebung, wobei die Wasm-Services im Kernelmode ausgeführt werden. Die Roundtrip-Zeit wurde für alle drei vorgestellten Ansätze gemessen: die OS-API, die R-API und MMIO. Die Zephyr-API, die die Isolation des Usermodes nutzt, dient als Referenz und wird als Native bezeichnet. Wie dargestellt, sind die OS-API und die R-API in der Dauer ungefähr äquivalent zur Zephyr-API. Die MMIO-Lösung dagegen ist wesentlich schneller als die anderen, einschließlich der Referenz.

Eine Analyse der Instruktionsanzahlen auf der rechten Seite der Abbildung zeigt, dass die verhältnismäßig lange Ausführungszeit der nativen Ausführung hauptsächlich auf Kontextwechsel zurückzuführen ist. Die OS-API- und R-API-Zahlen weisen eine beträchtliche Anzahl von Instruktionen im Zusammenhang mit der Wasm runtime auf, hauptsächlich resultierend aus der Ausführung importierter Funktionen. Dieser Prozess beinhaltet die Auflösung von Funktionszielen, die Erstellung neuer Wasm-Call-Stack-Frames und die Konvertierung von Funktionsargumenten. Im Gegensatz dazu beinhaltet der MMIO-Ansatz nur die grundlegende Wasm-Speicherzugriffsprüfung, was zu deutlich reduzierter Ausführungszeit und geringerer Instruktionsanzahl führt. Der Anteil von Wasm-IO resultiert aus der Geräte-Handle-Suche oder der Adressprüfung. Die leichten Abweichungen zwischen OS-API und R-API können auf den zusätzlichen Kernel-Level-Aufruf der Zephyr-API bei der OS-API zurückgeführt werden.

Die Messkonfiguration ist für die Diskrepanzen zwischen Zeitmessungen und Instruktionsanzahlen (OS-API und R-API vs. Zephyr-API) verantwortlich. Wie in Abbildung 9 gezeigt, umfassen die Zeitmessungen die Rückkehrpfade der I/O-Anweisungen. Während diese Rückkehrpfade im Kontext der Zephyr-Implementierung vernachlässigbar sind, haben sie signifikante Auswirkungen auf die Wasm-Implementierungen, wo das Freigeben von Call-Frames, Kopieren von Rückgabewerten und das Ausführen von Schleifen innerhalb von Wasm zusätzliche Instruktionen erfordern. Obwohl die Zeitmessungen diese Feinheiten erfassen, spiegeln die entsprechenden Instruktionsanzahlen sie nicht wider.

Die Analyse der Echtzeitfähigkeiten für synchrone Peripherieinteraktionen betont die erhebliche Abhängigkeit von der Wasm runtime. Diese Abhängigkeit wird jedoch als Bestandteil des Kostenmodells betrachtet. Eine Untersuchung der aufgezeichneten Spuren zeigt ferner, dass der Beitrag von Wasm-IO zu den Worst-Case-Instruktionsanzahlen signifikant geringer ist als der der Wasm runtime. Manuelle Codeanalysen enthüllen ein lineares Verhalten, das aufgrund der Suche durch die freigegebenen Register über alle drei Methoden hinweg mit der Anzahl der bereitgestellten Register skaliert. Die aktuelle Implementierung ist durch ein konfigurierbares Maximum begrenzt.

Unvertrauenswürdige Laufzeitumgebung. Im zweiten Szenario wird die Wasm runtime nicht als vertrauenswürdig angesehen und läuft daher im Usermode. Abbildung 11 zeigt die Roundtrip-Zeiten links und die Instruktionsanzahlen rechts. Wie erwartet sind die WOETs deutlich schlechter als bei der direkten Nutzung der Zephyr-API aus dem Usermode (38 µs). Zudem ist die MMIO-Lösung langsamer als alle anderen Optionen aufgrund der zusätzlichen Verwaltungsdaten, die beim Systemaufruf in den Kernel kopiert werden müssen.

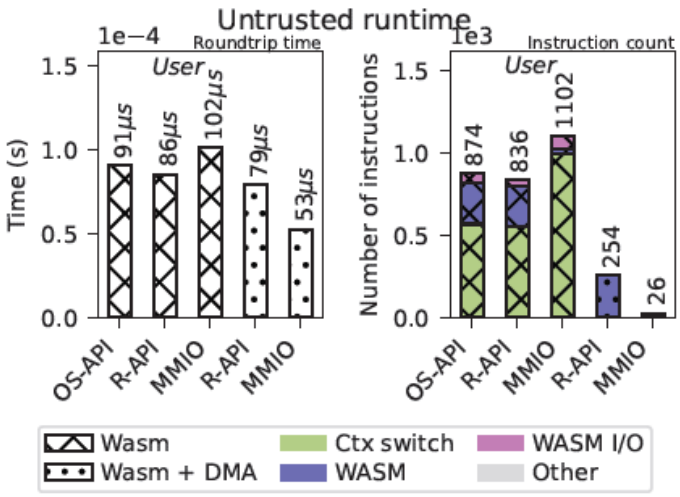

Abbildung 11: Ausführung von synchronem I/O mit Wasm-IO im Usermode

Die durch DMA beschleunigten Ansätze zeigen eine bessere Leistung im Vergleich zu den Nicht-DMA-Versionen, übertreffen jedoch nicht die Zephyr-API im Usermode. Der Hauptfaktor für den Overhead ist der Kontextwechsel. Die Instruktionsanzahlen für die DMA-Beschleunigung weisen auf eine geringere CPU-Belastung hin, da die DMA-Einheit die Verantwortung für das Kopieren übernimmt.

Obere Grenzen hängen von den Eigenschaften des Kontextwechsels ab, insbesondere vom Kopieren von Teilen der Wasm-Strukturen in den Kernelraum, welche eine konstante Größe aufweisen. Der Beitrag von Wasm-IO bleibt unverändert. Für den DMA-Ansatz muss das Kostenmodell den externen Controller, einschließlich der Buskommunikation, berücksichtigen.

Abbildung 12 zeigt den Messvorgang zur Bestimmung der Interrupt-Latenzen. Die Messung beginnt mit der steigenden Flanke am Eingangspin des Interrupts (interrupt source pin), die einen Interrupt auslöst. Dies startet den Wasm-IO-Prolog. Anschließend werden der Wasm-IO- und der Wasm-Epilog ausgeführt, lesen die bereitgestellten Daten und aktivieren den Messpin, um den Datenempfang zu signalisieren. Die Zephyr-Referenz führt das gleiche Verfahren durch und nutzt eine native Callback-Funktion, um den Pin nach dem Laden der Daten zu aktivieren. Zur Berechnung der Interrupt-Latenz wird die Differenz zwischen der Triggerflanke und der Empfangsflanke genommen und die zuvor gemessene Roundtrip-Zeit des Pins subtrahiert. Die grauen Kästchen am unteren Rand von Abbildung 12 zeigen die Einschränkungen des Messverfahrens zur Instruktionszählung. Sie markieren die Anweisungen für Rescheduling und Thread-Erstellung, die nicht in den

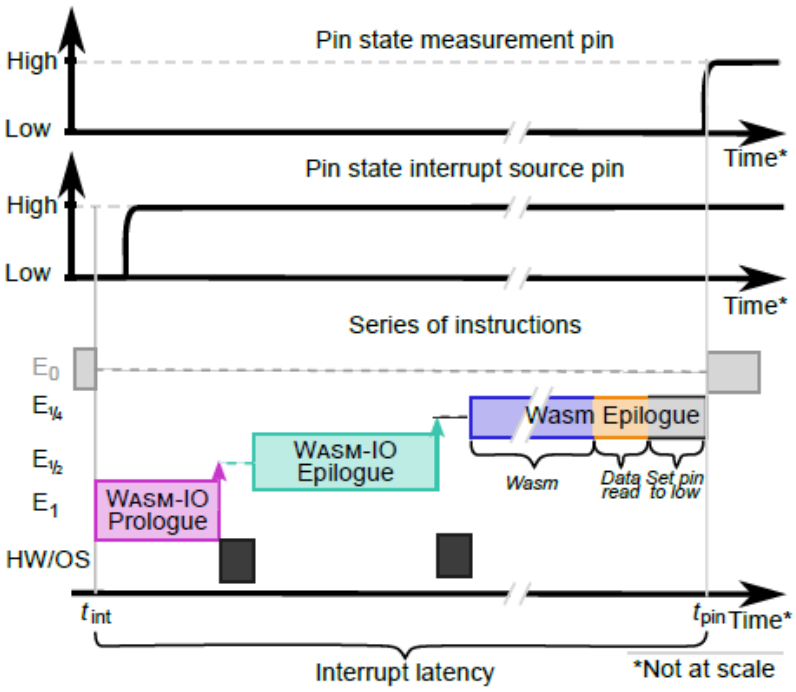

Abbildung 12: Messvorgang für asynchrones I/O

Instruktionszählungsergebnissen enthalten sind. Zudem bietet die OS-API keine Funktionen für beliebigen Datenzugriff und ist daher von der Messung ausgeschlossen.

Vertrauenswürdige Laufzeitumgebung. Abbildung 13 zeigt die Ergebnisse für die vertrauenswürdige Laufzeitumgebung. Das linke Diagramm zeigt, dass alle Interrupt-Latenzen, einschließlich der Referenz, vergleichbar sind. Rechts zeigen die Instruktionsanzahlen, dass die Dauer des Prologs über alle Ansätze hinweg konsistent und im Vergleich zu anderen Komponenten relativ kurz ist. Die Zeit, die das System nicht-responsiv im verbringt, Prolog beträgt weniger als 25 µs, während der Epilog für die Wasm-Ansätze

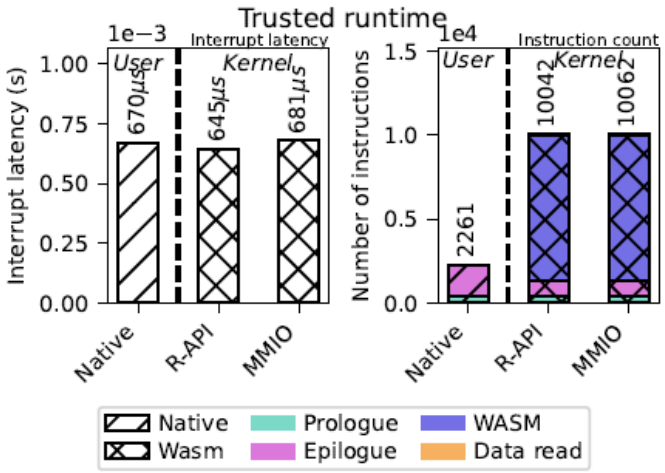

Abbildung 13: Interruptlatenzen und Instruktionen mit Wasm-IO im Kernel

weniger als 63 µs benötigt. In allen Fällen stehen die meisten Instruktionen im Zusammenhang mit dem Übergang in die geschützte Domäne. Der Epilog für die Zephyr-Referenz dauert länger, da ein zusätzlicher User-Thread initialisiert werden muss, um den registrierten Callback auszuführen.

Die oberen Latenzgrenzen hängen erneut vom Betriebssystem, dessen Kontextwechsel- und Scheduling-Implementierung und dem Aufruf der Wasm runtime ab. Diese Aspekte wurden im Zuge dieser Arbeit nicht untersucht. Der Beitrag von Wasm-IO für jeden Wasm-Service ist direkt proportional zur Anzahl der registrierten Wasm-Epiloge und der Anzahl der Datenobjekte, die zum Kopieren in jeden linearen Speicher registriert sind. Es gibt konfigurierbare Maxima, die beide einschränken.

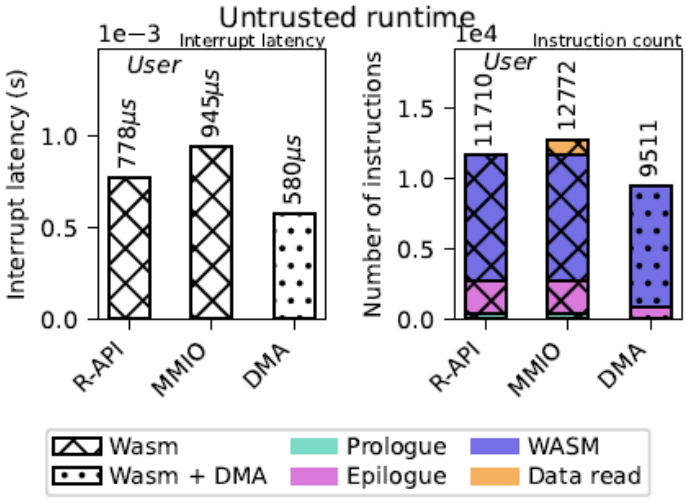

Abbildung 14: Interrupt-Latenzen mit Wasm-IO im Usermode

Unvertrauenswürdige Laufzeitumgebung. Im Fall einer nicht vertrauenswürdigen Laufzeitumgebung sind die beobachteten Latenzen höher, wie in Abbildung 14 bestätigt wird. Obwohl der DMA-beschleunigte Ansatz schneller ist als die hier vorgestellten für die vertrauenswürdige Laufzeitumgebung, hängen die Echtzeitgrenzen erneut vom externen Hardwaremodul ab, das ein detailliertes Kostenmodell erfordert. Weder der Wasm-IO-Prolog noch der Wasm-IO-Epilog sind von der Vertrauenswürdigkeit der Laufzeitumgebung beeinflusst.

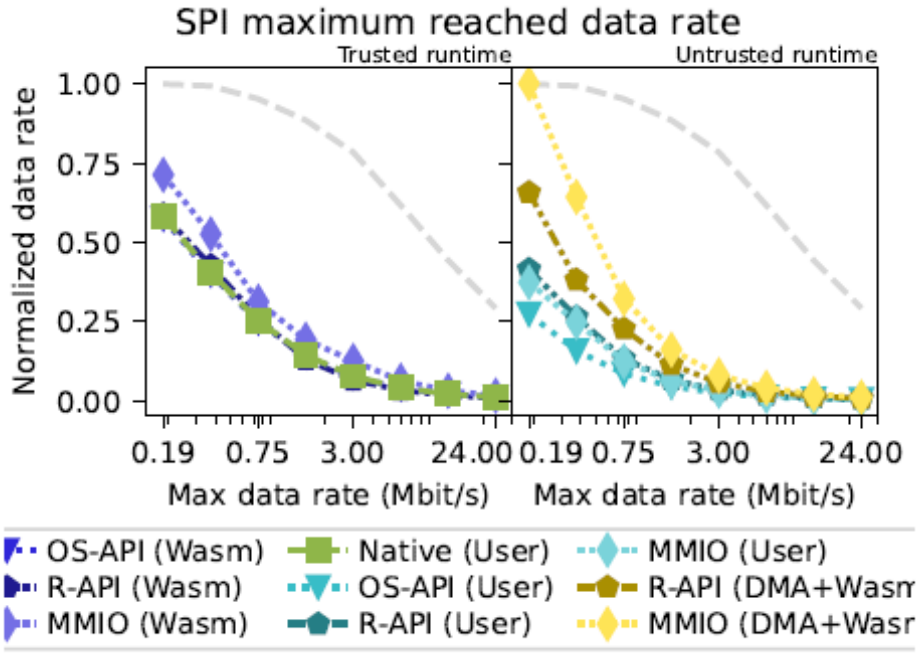

Abbildung 15: Normierte Datenrate über konfigurierter Datenrate

SPI Fallstudie. Um die Anwendbarkeit von Wasm-IO zu evaluieren, wurde ein exemplarischer Hardwaretreiber in Wasm entwickelt, der einen SPI-Bustreiber implementiert. SPI ist ein weit verbreiteter Standard mit einer etablierten Referenzimplementierung im Betriebssystem sowie verfügbaren Decodern für Verifikations- und Messzwecke. Grundlegende Funktionen wie ein SPI-Treiber sind allerdings typischerweise Teil des Betriebssystems und daher in der Praxis vermutlich nicht als Wasm-Service implementiert. Hier wurde er dennoch verwendet, um einen aussagekräftigen Vergleich herstellen zu können, verglichen mit dem integrierten Betriebssystemtreiber.

Es wurden 1 KiB Daten verwendet, unterteilt in 16-Bit-Wörtern. Unterschiedliche Datenraten wurden konfiguriert, die von 18,75 kHz bis zu 24 MHz reichen, was der Hälfte der CPU-Taktfrequenz entspricht. Für die Messung wurde der in Abbildung 9 dargestellte Prozess eingesetzt. Der Messpin wurde vor Beginn der SPI-Transaktion auf High gesetzt und nach Abschluss wieder zurückgesetzt; das heißt, das actual program aus Abbildung 9 repräsentiert die SPI-Transaktion. Anschließend wurde die GPIO-Roundtrip-Zeit subtrahiert.

Abbildung 15 zeigt die Ergebnisse. Die gemessene Datenrate wird für jede konfigurierte Datenrate auf der y-Achse dargestellt, wobei die x-Achse die konfigurierten Datenraten zeigt. Die Werte wurden relativ zur konfigurierten Rate normalisiert. Die graue gestrichelte Linie ohne Markierungen zeigt die Ausführung mit der nativen Zephyr-API im Kernelmode und gibt die erreichbare maximale Datenrate an. Selbst diese performanteste Implementierung erfährt bereits bei 37,5 kHz, der zweitniedrigsten Datenrate, eine Reduktion der Datenrate

auf 99,5 %. Diese Reduktion ist auf die relativ niedrige CPU-Taktfrequenz im Vergleich zur hohen SPI-Taktfrequenz zurückzuführen.

Die Werte der Trusted Runtime stimmen mit den vorigen Messungen überein. Wie erwartet zeigt der MMIO-Ansatz die beste Performance und übertrifft dabei auch die OS-API. Dennoch erreicht keiner dieser Ansätze mehr als 71 % der möglichen Datenrate, selbst bei Konfiguration der niedrigsten Datenrate. Interessanterweise zeigt die OS-API im Untrusted-Fall eine geringere Performance als sowohl die R-API als auch MMIO, was den vorherigen Messungen widerspricht. Diese Diskrepanz ist auf unterschiedliche Implementierungen der SPI OS-API und GPIO OS-API zurückzuführen. Erstere kopiert Referenzen auf den Wasm-Speicher, was eine zusätzliche Kopie des linearen Wasm-Speichers erfordert. Letztere hingegen kopiert den Wert eines Integers (High oder Low) über den Stack des OS-Threads an das Betriebssystem, was deutlich schneller ist.

Die DMA-Lösungen zeigen deutlich bessere Ergebnisse, da sie die SPI-Register ohne Beteiligung der CPU kopieren können und somit Übergänge zwischen den Schutzdomänen für jedes Byte vermeiden. Dies ermöglicht zumindest für 18,75 kHz die volle Datenrate im MMIO-Ansatz.

Insgesamt zeigen die Ergebnisse, dass die Instrumentierung von I/O zwar machbar ist, jedoch mit einem beträchtlichen, aber bekannten Overhead verbunden ist. Der MMIO-Ansatz übertrifft die traditionelle Usermode-Isolation bei synchronem I/O und in der SPI-Fallstudie, erfordert jedoch eine vertrauenswürdige Laufzeitumgebung. Asynchrones I/O zeigt im Trusted-Fall über alle Lösungen vergleichbare Overheads. Der Untrusted-Fall offenbart erhebliche Performanceeinbußen, die eine praktische Anwendung erschweren. Die DMA-Beschleunigung verringert den gesamten Overhead, geht jedoch mit der Notwendigkeit eines zunehmend komplexeren Hardwarekostenmodells einher.

6.2.4 Erweiterung des LSM6DSL Treibers in Zephyr

Wie in Kapitel 6.2.1 beschrieben, wurden im Zuge des Projekts verschiedene Echtzeitbetriebssysteme (ZephyrRTOS, Mentor Nucleus und FreeRTOS) verglichen, wobei sich letztlich für Zephyr entschieden wurde. Diese Entscheidung begründete sich auch durch ausführliche Dokumentation und das bessere Tooling, welches die überdurchschnittlich hohe Unterstützung von Peripherie und Sensorik einschließt, einschließlich des in Kapitel 5 eingeführten LSM6DLS Beschleunigungssensors.

Im ersten Projektjahr wurde angenommen, dass die Treiber der Dokumentation und der Reputation folgend entsprechend gut sind und sich auch somit unverändert nutzen lassen. Dies hat sich leider als unzutreffende Annahme erwiesen, weshalb im Zuge des Projekts die Treiberinfrastruktur neu und abweichend vom durch Zephyr propagierten Design konzipiert, integriert und implementiert werden musste.

Der LSM6DSL Sensor ist konfigurierbar auf Abtastraten mit bis zu 6,6 kHz. Tests mit dem nativen ZephyrRTOS Treiber ergaben jedoch in der Praxis maximale Abtastraten im Bereich einiger 100 Hz. Abbildung 16 zeigt links den nativen Treiber, wobei der Datenverlust ab 833 Hz deutlich ansteigt. Der exakte Schwellwert hängt stark von der Implementierung und Konfiguration ab, ist jedoch für die Datenauswertung schlicht inakzeptabel.

Weiter unterstützt der ZephyrRTOS Treiber nur Bruchteile der LSM6DSL Funktionalität, wobei insbesondere der „FIFO-Mode“ nicht implementiert ist. Dieser könnte eine schnellere Ausführung ermöglichen, da er den Hardwarepuffer des LSM6DSL verwendet. Ein Betrieb ohne diesen ist in der Praxis nicht möglich, da eine Synchronisation der Erzeugungszeitpunkte der Daten durch den Sensor sowie der Abfragezeitpunkte durch das Mikrocontrollersystem technisch nicht möglich ist. Doppelungen oder das Fehlen von Daten ist somit weder verhinderbar noch überhaupt detektierbar, wodurch eine kontinuierliche Messung von Vibrationen, eine rechtzeitige Datenerfassung aus dem Sensor und eine lückenlose Datenauswertung nicht möglich sind. Der native ZephyrRTOS Treiber setzt weiterhin standardmäßig einige Vorfilter, insbesondere Tiefpassfilter, was sich nicht abändern lässt, die Datenanalyse aber offensichtlich erheblich beeinflusst.

Für das von Siemens angedachte Datenstreaming wurden daher einige Anpassungen vorgenommen. Zunächst wurde eine verbesserte Komprimierung eingeführt, indem die 16 Bit Sensorwerte auch durch die passenden Datentypen repräsentiert werden. Die Standardimplementierung nutzte hier 2x32 Bit repräsentiert werden was pro Beschleunigungsachse zu einer Vergrößerung der zu sendenden Daten um einen Faktor von vier führt.

Des Weiteren wurden erweiterte Konfigurationsoptionen hinzugefügt, so dass sich nun der Hardwarepuffer des LSM6DLS konfigurieren und verwenden lässt. Das polling-basierte Ausführungsmodell wurde gänzlich verworfen. Der Sensor wird nun im Interrupt-Modus betrieben und meldet sich, sobald der Puffer bis zu einem bestimmten Schwellwert gefüllt ist. Anschließend können die Daten asynchron weiterverarbeitet werden. Dazu werden sie auf den von Zephyr zur Verfügung gestellten *zbus* geschrieben. Dieser stellt einen virtuellen Kommunikationsbus zur Kommunikation zwischen verschiedenen Softwarebestandteilen zur Verfügung und operiert nach dem publish-subscribe-Prinzip. Die anschließende CBOR-Datenkodierung ermöglicht besonders datensparende Formate, wenn Sender und Empfänger sich auf ein Datenformat geeinigt haben.

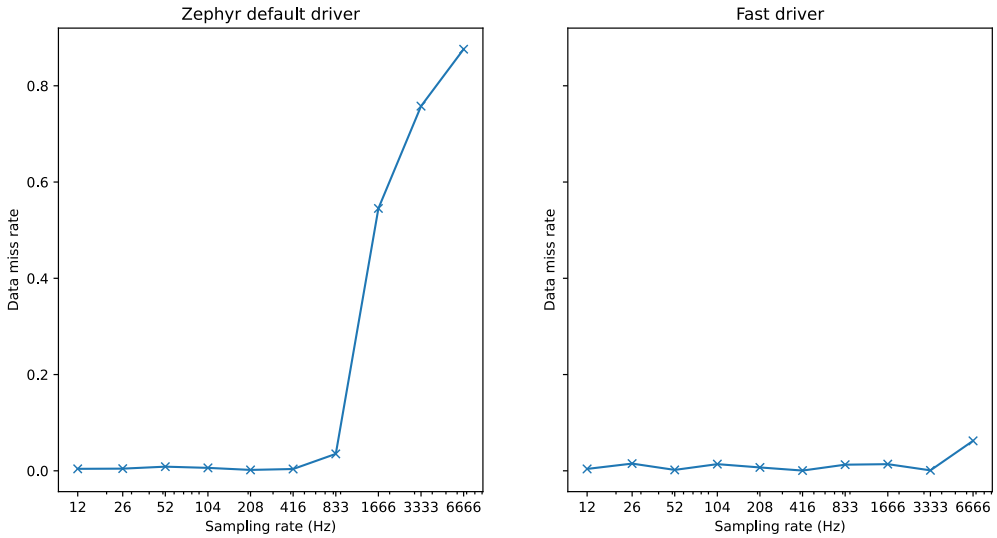

Abbildung 16: Datenverlust bei verschiedenen Datenraten, links mit dem nativen und rechts mit den durch Siemens neu entworfenen Fast driver.

Abbildung 16 zeigt den Vergleich der Datenverluste des Zephyr Treibers sowie des neu implementierten LSM6DSL fast drivers in seiner neuen Architektur. Zusammen mit der deutlich besseren Komprimierung und der somit schnelleren Übertragung, konnte die Ausführungsgeschwindigkeit deutlich erhöht werden. Eine Abtastrate von 3,3 kHz ist somit möglich, bei weiterer Verbesserung und Reduktion der parallelen Aktivitäten zeichnet sich auch eine Datenrate von 6,6 kHz als möglich ab.

6.3 Dynamische Betriebsführung (AP3)

In Mannheim EMDRIVE wird von Siemens die dynamische Betriebsführung in industriellen Systemen untersucht. Dynamische Betriebsführung beschreibt dabei die flexible und adaptive Allokation von Ressourcen innerhalb eines verteilten Systems, insbesondere die Zuweisung von Geräten und das Verschieben von Programmen zwischen diesen Geräten. In modernen IoT-Architekturen und industriellen Anwendungen ist es oft notwendig, dass Softwarekomponenten nicht statisch auf bestimmten Hardwareeinheiten verbleiben, sondern je nach Auslastung, Verfügbarkeit oder anderen betrieblichen Anforderungen dynamisch neu zugeordnet werden können. Dadurch wird eine effiziente Ressourcennutzung ermöglicht, die Skalierbarkeit des Systems erhöht und die Fähigkeit gesteigert, auf Veränderungen in der Umgebung oder im Systemzustand zu reagieren. Um dies in einem Demonstrator sichtbar zu machen, mussten zunächst verschiedene Betriebsmodi erörtert werden, bevor eine Ausführung auf unterschiedlichen Geräten implementiert werden konnte.

6.3.1 Notwendigkeit einer dynamischen Betriebsführung

In der Datenakquise im industriellen IoT-Bereich erfassen Sensoren erfassen Umgebungsdaten (Sense), die verarbeitet werden, um darauf basierend Aktionen auszulösen (Act). Dieser kontinuierliche Kreislauf ermöglicht es, Systeme in Echtzeit an veränderte Bedingungen anzupassen und effiziente Steuerungsmechanismen zu implementieren oder verlässliche Aussagen bezüglich predictive Maintenance zu tätigen.

Durch die fortlaufende Erfassung von Sensordaten entstehen Zeitreihendaten, die den zeitlichen Verlauf der Messgrößen abbilden. Diese Daten sind essenziell für Analysen, Prognosen und die Identifikation von Mustern oder Anomalien im Systemverhalten. Aufgrund begrenzter Ressourcen und eingeschränkter Übertragungsmöglichkeiten aufgrund hohem Stromverbrauchs und Verstopfung der physikalischen Datenkanäle können jedoch nicht alle Daten kontinuierlich übertragen werden. Hier kommt das Snapshotting zum Einsatz: Die Sensordaten werden in Zwischenspeichern mit einer festen (ggf. zeitlichen) Größe gespeichert und können anschließend als Datenmenge übertragen werden. Diese Snapshots ermöglichen es, relevante Informationen zu sammeln, ohne die verfügbaren Kommunikationskanäle zu überlasten oder den Energieverbrauch unverhältnismäßig zu erhöhen.

Nach der Datenerfassung werden verschiedene Eigenschaften der aufgezeichneten Signale berechnet, sogenannte Features oder auch Key Performance Indikatoren (KPIs). Diese können statistische Kennwerte Frequenzkomponenten oder andere charakteristische Merkmale der Messkurven sein. Anhand dieser Features erfolgt eine Kategorisierung oder Klassifizierung der Datenabschnitte. Beispielsweise können bestimmte Muster in den Beschleunigungsdaten auf Verschleiß oder Fehlfunktionen hinweisen, sodass präventive Wartungsmaßnahmen eingeleitet werden können. Die tatsächliche Klassifikation erfolgt dann anhand eines oder mehrerer Features.

Eine leitende Vermutung, die dem Förderprojekt zu Grunde lag, war, dass eine Erweiterung auf mehr oder andere Featurekombinationen die Klassifizierungsgenauigkeit verbessert. Um diese jedoch ausführen zu können, ist eine höhere Rechenleistung nötig, beispielsweise durch ergänzende Geräte neben den IoT Sensoren. Um diese Vermutung, und somit die Begründung zur dynamischen Betriebsführung, zu bestätigen, wurden zunächst Laborversuche mit Zeitreihendaten durchgeführt.

Predictive Maintenance Anwendungen gehen häufig mit Temperatur- und Vibrationsmessungen einher. Temperaturveränderungen weisen jedoch in der Regel eine hohe Trägheit auf, weshalb niedrige Abtastraten ausreichend sind und wenige Daten anfallen. Sogar aufwändige Nachverarbeitungsschritte können hier durchgeführt werden, da der

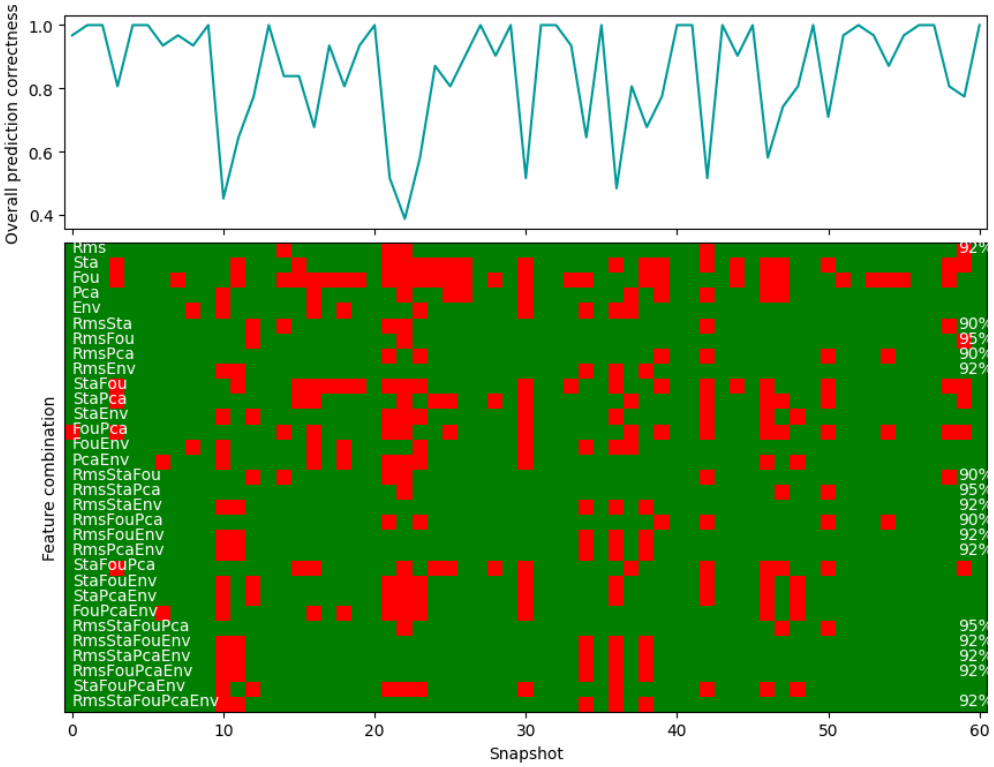

Abbildung 17: Genauigkeit verschiedener Features und Kombinationen

Prozessor des Sensors viel Zeit bis zum nächsten Messpunkt hat. Im Gegensatz dazu stehen Vibrationsdaten. Hier sind Abtastraten im Kiloherzbereich typisch, was den Sensing-Prozess an sich oft schon schwierig gestaltet. Folglich wurden für die Laborversuche Vibrationsmessungen ausgewählt, die an einem Motor stattfinden sollte. Aufgrund der vorhandenen Ausrüstung und Expertise wurde daher als Zwischenschritt der Motor einer Ölpumpe eines Zugfahrgestells ausgewählt.

Abbildung 17 zeigt 60 ausgewählte Snapshots einer Ölpumpe mit verschiedenen Feature-Kombinationen. Dazu wurden gängige Features aus der Literatur ausgewählt und anschließend kombiniert. Grüne Bereiche zeigen eine korrekte Klassifizierung, rote Bereiche eine fehlerhafte. Im Graphen darüber ist die Vorhersagegenauigkeit dargestellt. Wie zu erkennen ist, sind einige Snapshot-Gruppen zu finden, bei welchen die Erkennungsrate niedrig ist. Dies liegt unter anderem an der kleinen Messfenstergröße, die durch den kleinen Arbeitsspeicher des Mikrocontrollers begrenzt ist, gleichzeitig aber vorübergehende Störeinflüsse wie Luftblasen im Öl und externe Erschütterungen übermäßig stark gewichtet. Eine Auslagerung an ein Gerät mit mehr verfügbarem Arbeitsspeicher deutet sich hier bereits an.

Anhand von Abbildung 17 lässt sich außerdem auch noch erkennen, dass insbesondere Featurekombinationen, die eine Fouriertransformation oder eine Hauptkomponentenanalyse (PCA) enthalten, gut performen. Diese wiederum sind

arbeitsspeicherlastige Algorithmen, welche ebenfalls von einer Auslagerung an ein Edge-Gerät profitieren.

6.3.2 Quality of Sensing als Initiator

Die kontinuierliche Erfassung von Sensordaten bildet die zentrale Grundlage für die Entscheidungsprozesse in einer automatisierten Fertigungsumgebung. Gerade in der Automotive-Industrie, wo Bauteile wie Motoren, Getriebe und andere mechatronische Komponenten einer hohen Belastung unterliegen, entscheidet die Güte dieser Sensorik darüber, ob frühzeitig auf Verschleiß, Vibrationen oder strukturelle Veränderungen reagiert werden kann. Hier setzt das Konzept des Quality of Sensing (QoS) an, das in der vorliegenden Systemarchitektur eine Schlüsselrolle übernimmt, um die Betriebsführung dynamisch zu gestalten. Im Kern verfolgt QoS das Ziel, den Zustand und die Genauigkeit von Sensoren zur Laufzeit zu bewerten. Dazu werden Messdaten wie Beschleunigung, die von einem Beschleunigungssensor erfasst werden, kontinuierlich aufgezeichnet und anhand verschiedener statistischer und spektraler Verfahren analysiert. Diese Datengrundlage

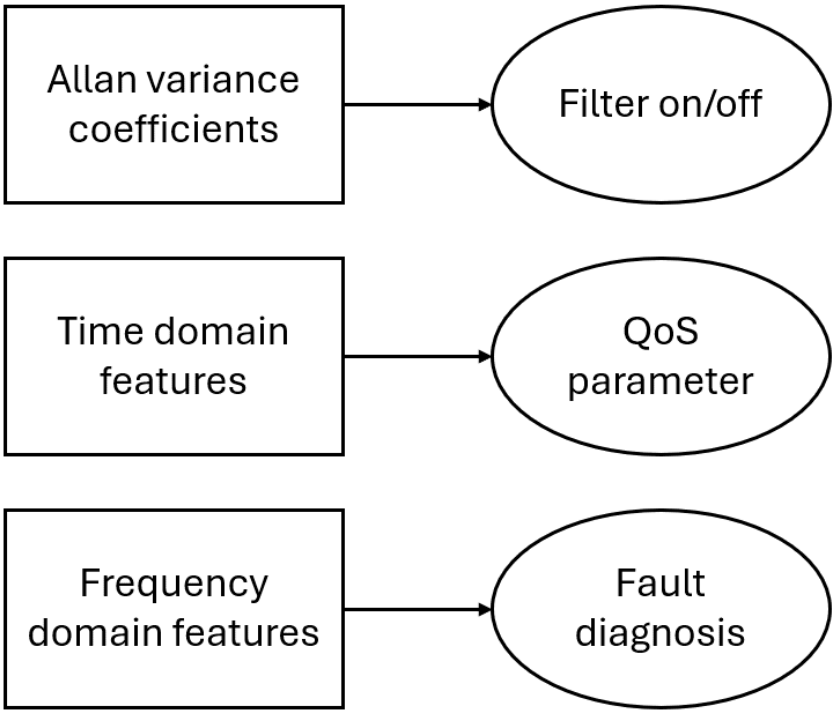

Abbildung 18: Hierarchische Anordnung der eingesetzten Methoden zu Genauigkeitsbestimmung von MEMS-Sensorik

erlaubt Rückschlüsse sowohl auf das eigentliche Messverhalten des Sensors als auch auf dessen Einbettung in den Prozess, sodass Abweichungen oder sich abzeichnende Fehler umgehend auf Sensorebene erkannt werden können. Hierbei kommen Methoden wie Allan-Varianz, Power Spectral Density (PSD) und Wavelet-Analysen zum Einsatz, um unterschiedliche Arten von Rauschen und Abweichungen isoliert zu betrachten. Es werden

Kenngrößen wie Bias Error, Bias Instability, Angle beziehungsweise Velocity Random Walk, Quantization Noise und Rate Random Walk betrachtet. Außerdem statistische Kenngrößen von Zeitreihen wie Formfaktor, Scheitelfaktor, Schiefe, Wölbung und Effektivwert miteinbezogen. Werden für einen oder mehrere dieser Werte kritische Schwellen überschritten, kann die zuständige Steuerungslogik eine Anpassung der Betriebsmodi oder Wartungs- und Updateprozesse veranlassen.

Abbildung 18 veranschaulicht die hierarchische Anordnung der Methoden, die sowohl auf einem eingebetteten Mikrocontroller als auch auf einem Edge-Gerät oder in der Cloud ausgeführt werden können. Zunächst wird die Allan-Varianz berechnet, um gegebenenfalls Filter zu aktivieren oder zu deaktivieren und damit Weißes Rauschen zu unterdrücken. Die Allan-Varianz ist ein Verfahren, um unterschiedliche Arten von Rauschen in einem Messsignal zu erkennen und zu quantifizieren. Weisen die Allan-Varianz-Koeffizienten auffällige Ausschläge auf, werden anschließend die relevanten Kenngrößen aus den Zeitreihen bestimmt und mithilfe von Übertragungsfunktionen zu einem QoS-Parameter fusioniert. Winkelzufallswanderung (Angle Random Walk), Bias-Stabilität (Flicker-Rauschen) oder Rate-Random-Walk, im zeitlichen Verlauf zu identifizieren und damit eine präzisere Beurteilung der Signalqualität zu erhalten. Hierfür wird das gesamte Datenmaterial in Zeitintervalle (Cluster) unterteilt, für die jeweils ein Mittelwert bestimmt wird. Diese sogenannten Cluster-Mittelwerte dienen als Basis zur Berechnung der Allan-Varianz und der daraus abgeleiteten Allan-Abweichung. In der Praxis entsteht daraus eine charakteristische Allan-Kurve, die auf der horizontalen Achse die jeweilige Clusterlänge abträgt und auf der vertikalen Achse die Allan-Abweichung des Signals. Die unterschiedlichen Rauschanteile lassen sich anhand typischer Steigungen (Slopes) im Verlauf der Allan-Kurve erkennen. Quantisierungsrauschen zeigt sich in der Regel mit einer Steigung von −1, das weiße Rauschen (Random Walk) mit −0,5, während sich Flicker-Rauschen als Bias-Stabilität bei einer Steigung von 0 bemerkbar macht. Ein Brown’sches Rauschen (Random Walk) der Rate zeigt sich oft durch eine Steigung von +0,5, wohingegen eine Rate-Rampe (z. B. rosa Rauschen) mit einer Steigung von +1 identifiziert werden kann.

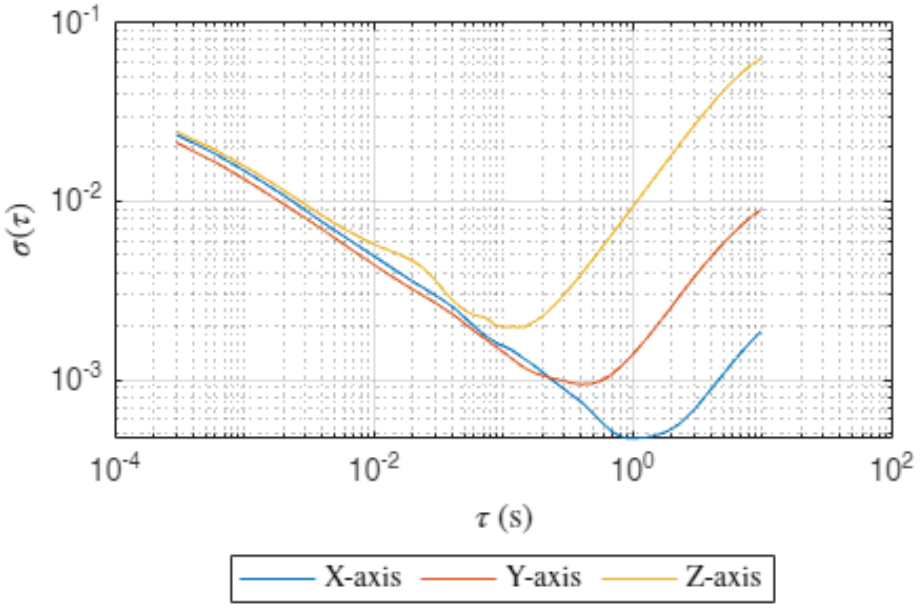

Abbildung 19: Allan-Kurven der drei Achsen des LSM6DSL Beschleunigungssensors

In Abbildung 19 zeigt die Ergebnisse der Rauschanalyse mithilfe der Allan-Varianz, wobei für jede Achse eines dreiachsigen Beschleunigungssensors die Allan-Deviation berechnet wurde. Hierbei kam ein Datensatz von 10^4 Kilo-Samples zum Einsatz, in der sich der Beschleunigungssensor in Ruhe befindet. Die X-Achse der Abbildung zeigt die Zeitverschiebung τ in Sekunden, während die Y-Achse die Allan-Divergenz (Allan-Deviation) repräsentiert. Deutlich erkennbar ist, dass sich die drei Achsen in ihrem Verlauf unterscheiden. In der Darstellung entspricht die blaue Kurve der X-Achse, die orangefarbene Kurve der Y-Achse und die gelbe Kurve der Z-Achse des Beschleunigungssensors. Anhand der Steigungen (Slopes) können die verschiedenen Noise-Anteile pro Achse ermittelt und bewertet werden. Durch die gezielte Betrachtung dieser Steigungen wird ersichtlich, in welchen Zeitbereichen sich beispielsweise weißes Rauschen (rote Linie) besonders stark ausprägt, während in anderen Segmenten eventuell ein 1/f-Rauschen oder 1/f²-Rauschen dominiert. Die genauere Analyse offenbart somit, welche Rauschmechanismen für die einzelnen Achsen charakteristisch sind und inwieweit sie sich in Intensität und Verlauf unterscheiden.

Deuten diese Koeffizienten auf einen Fehler oder Verschleiß hin, folgt eine Fault Diagnosis, die den Frequenzbereich einbezieht und dabei insbesondere Beschädigungen des Außenrings von Lagern, Schläge oder mechanische Lockerheiten erkennt. Abbildung 20 zeigt exemplarisch zwei Spektrogramme, die im Rahmen des Nyquist-Shannon-Abtasttheorems bis zu 1,6 kHz über eine Dauer von 1 s abdecken. Im linken Spektrogramm ist im Normalzustand mit einer Grundfrequenz von 100 Hz kaum weißes Rauschen zu erkennen. Im Gegensatz dazu

weist das rechte Spektrogramm eine deutliche Zunahme weißen Rauschens auf, die sich in einer höheren Intensität über alle Frequenzbereiche bemerkbar macht, während die Grundfrequenz von 100 Hz selbst noch weitgehend unverrauscht bleibt. Bei weiterer Verstärkung dieses weißen Rauschens könnte allerdings das Signal-Rausch-Verhältnis kritisch beeinträchtigt werden.

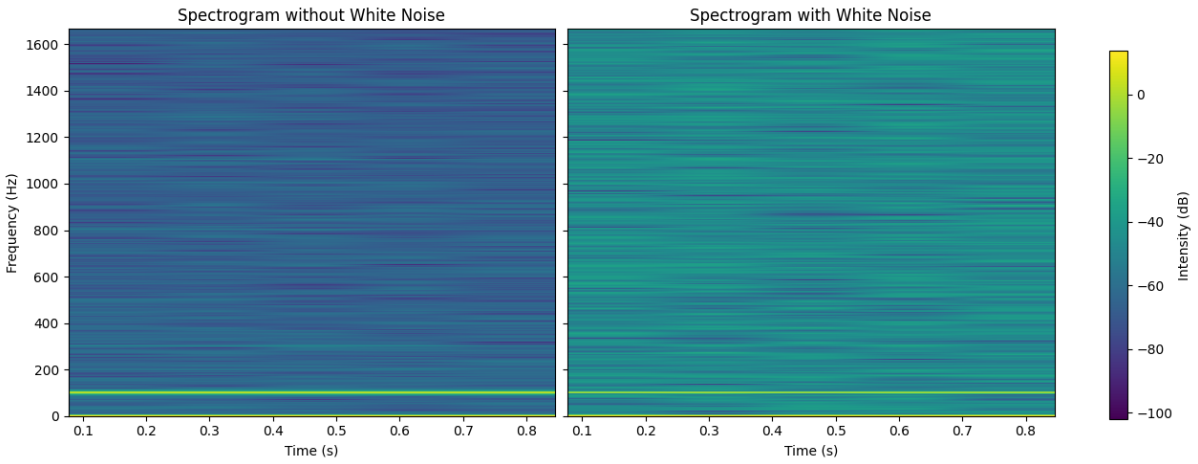

Abbildung 20: Spektrogramm zur Rauschquellenuntersuchung des Beschleunigungssensors auf einem Prüfstand mit der Grundfrequenz von 100 Hz mit und ohne weißes Rauschen.

Auf diese Weise wird eine dynamische Betriebsführung initiiert, die nicht nur starr auf festgelegte Produktionsabläufe zurückgreift, sondern das tatsächliche Zustandsbild des Sensors laufend einbezieht. Damit dieser Kreislauf aus Sensorbeobachtung und Betriebsanpassung in Echtzeit funktioniert, ist eine Systemarchitektur nötig, die sowohl auf Firmware-Ebene als auch in der Edge-Infrastruktur die relevanten Daten weiterleitet und verarbeitet.

Diese Algorithmen laufen je nach Anforderung entweder lokal auf dem Sensor oder werden in Containern an eine Edge-Plattform ausgelagert, wo sie mit aktuellen und historischen Daten verglichen werden können. Die Fähigkeit zur flexiblen Anpassung beruht auf der Container-basierten Organisation der Auswertungs- und Überwachungssoftware. Neue Algorithmen oder Parameter können im laufenden Betrieb integriert werden, sodass der Sensorstatus als aktiver Stellhebel für die gesamte Fertigung dient. Wird beispielsweise eine Abweichung im Messverhalten erkannt oder zeichnet sich eine Schwingungsinstabilität ab, lassen sich speziell angepasste Modelle einspielen und so das System dynamisch an die neuen Bedingungen anpassen. Das System reagiert flexibel, sobald bestimmte Schwellenwerte oder Anomalien registriert werden, und kann etwa Aufgaben auf eine andere Einheit verteilen. Der QoS-Ansatz ist somit mehr als bloße Fehlerdetektion. Er steht im Zentrum der dynamischen

Betriebsführung, in der Entscheidungen unmittelbar auf Analysen fundierter Messdaten basieren. Das Ergebnis ist ein fehlertoleranteres Fertigungssystem, das ohne große Ausfallzeiten auf neue Produktionsbedingungen, Komponenten und Qualitätsansprüche reagieren kann und damit einen Fortschritt gegenüber klassischen starren Fertigungskonzepten bietet.

6.3.3 Nimblenet: Dynamisches Ausführen in Verteilten Systemen

Um die dynamische Betriebsführung möglichst effizient in verteilten Systemen anwenden zu können, müssen auch Programme effizient migriert und versendet werden. Das hier erarbeitete Nimblenet Framework wurde im 10th International Workshop on Serverless Computing vorgestellt [35]. Das folgende Kapitel basiert auf dieser Publikation.

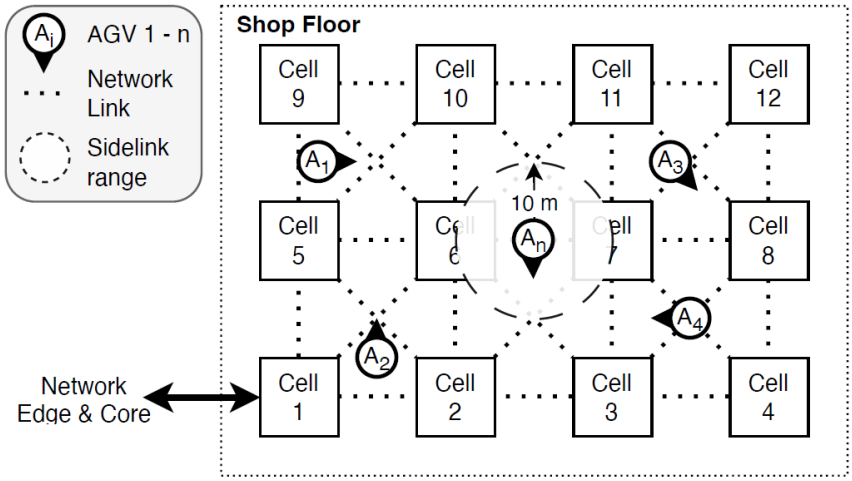

Abbildung 21: Zellbasiertes Fertigungssystem mit AGVs und Netzwerksetup

Wie bereits in Kapitel 5 beschrieben, sind zukünftige Produktionsumgebungen zellbasiert, statt der aktuell weitgehend vertretenen Linienfertigungen, wie in Abbildung 21 dargestellt. Diese Inselfertigungen sind deutlich flexibler in ihrer Verwendung. Autonome Transportfahrzeuge (AGVs) bewegen sich durch die Produktionshalle und bringen Materialien. Beim Erreichen der Produktionsinseln leiten sie außerdem die notwendigen Fertigungsschritte ein. Jede Zelle enthält Geräte (Knoten, Nodes) welche miteinander vernetzt sind, angefangen bei Mikrocontrollern mit begrenztem Speicher bis hin zu leistungsstarken PLCs.

Die Vielfalt der eingesetzten Hardware erschwert die Implementierung einheitlicher Softwarelösungen. Zudem führt die steigende Anzahl von Sensoren und Aktoren zu einer erhöhten Netzwerklast, was zu Engpässen und Beeinträchtigungen des Produktionsprozesses führen kann und einen effizienteren Mechanismus zur Übertragung von Programmen benötigt.

Dabei stellen sich insbesondere die Herausforderung, die begrenzte verfügbare Bandbreite in drahtlosen industriellen Netzwerken nicht zu überbeanspruchen. Insbesondere können Machine-Learning-Klassifizierer hohe Datenraten verursachen oder Echtzeitübertragungskanäle in zeitkritischen Netzwerken (Time Sensitive Networks, TSN) belegen. Ineffiziente Lastverteilungen können zu Netzwerküberlastungen führen, die den Produktionsprozess beeinträchtigen oder sogar zum Stillstand bringen. Daher ist eine effiziente Lastverteilung unter Nutzung von domänenspezifischem Wissen über die physische Umgebung unerlässlich, um Überlastungen zu vermeiden.

Um die Auslastung des Netzwerks zu minimieren, ist der Einsatz angepasster Protokolle erforderlich, die eine effiziente Migration ermöglichen und die Gesamtgröße der übertragenen Programme reduzieren. Dies kann durch Erhöhung der Informationsdichte, beispielsweise durch Datenkompression, oder durch Ausschluss redundanter Daten erreicht werden. Der erarbeitete Mechanismus konzentriert sich auf Letzteres, indem er vorhandenes domänenspezifisches Wissen nutzt, um die Effizienz des Caches zu steigern.

Ansatz. Bei der Analyse von Steuerungsanwendungen für Produktionsinseln wurden drei Hauptbereiche identifiziert, in die Algorithmen unterteilt werden können. Der hardwaredefinierte Bereich umfasst taskspezifische Konfigurationen von Geräten wie Sensoren und Aktoren, die eng an die Architektur der jeweiligen Hardware und deren Peripherie gebunden sind. Der datendefinierte Bereich wird durch die Art der gesammelten, erzeugten oder ausgewerteten Daten bestimmt. Dabei handelt es sich um Algorithmen, die sich aus verschiedenen physikalischen Aspekten und Dimensionen ableiten, wie beispielsweise Zeitreihenanalysen, die auf eine Vielzahl von Sensoren angewendet werden können, Bildverarbeitung oder Trajektorienberechnung. Diese Algorithmen basieren hauptsächlich auf mathematischen Berechnungen und finden auf unterschiedlichen Geräten Anwendung. Der prozessdefinierte Bereich bezieht sich auf werkstückspezifische Details, die oft physikalische Modellierungen der Werkstücke oder deren Qualitätsklassifizierung beinhalten.

Innerhalb des speziellen Umfelds in industriellen kann eine räumliche Abhängigkeit festgestellt werden. Da hardwaredefinierte Algorithmen an die Gerätetechnik gebunden sind, sind sie auch mit der Position jedes Geräts in der Werkshalle verknüpft. Daher sollten Programme, die auf diesen Algorithmen basieren, lokal auf dem jeweiligen Gerät zwischengespeichert werden. Datendefinierte Algorithmen sind intrinsisch mit den physikalischen Daten verbunden. Diese Verbindung ergibt sich aus dem Zusammenspiel zwischen dem Werkstück, dem Prozess selbst und den Fähigkeiten der Produktionsinsel, wobei die letzten beiden voneinander abhängig sind. Folglich sind die Eigenschaften diesen Produktionsinseln zugeordnet und sollten über deren Geräte verteilt werden. Beispielsweise basiert die Gaserkennung in chemischen Prozessen typischerweise auf Variationen elektrischer Eigenschaften, die unabhängig vom spezifischen Sensor sind. Die

Datenauswertungsalgorithmen können daher mit anderen Programmen geteilt werden, da häufig mehrere Gassensoren erforderlich sind. Ähnlich bestimmt die Kombination aus Prozess und Produktionsinsel, ob die Qualitätskontrolle auf Methoden wie der Analyse elektromagnetischer Wellen, der Vibrationsspektralanalyse oder bildbasierten Techniken basiert, die grundlegende Matrix- oder Vektoroperationen oder Bilderkennung nutzen. Daher ist ein Caching auf Ebene der Produktionsinsel vorteilhaft. Schließlich resultieren die prozessdefinierten Funktionen aus der Abfolge der Schritte am Werkstück und sind somit mit dem Werkstück und temporär mit dem AGV verbunden. Da die Fertigung so effizient wie möglich sein muss, wird angenommen, dass kritische Produktionsprozesse, einschließlich des Weges des Fahrzeugs durch die Produktionsinseln, optimiert sind. Folglich werden aufeinanderfolgende Schritte wahrscheinlich in nahe beieinander liegenden Produktionsinseln durchgeführt, was ein Caching in Zellclustern motiviert.

Diese räumliche Abhängigkeit ermöglichen ein besonders effizientes Caching. Da die Arbeitslasten inhärent unbekannt sind, ist eine statische Verteilung nicht praktikabel, weshalb ein heuristischer Ansatz für die Verteilung der zu übertragende Programme zur Laufzeit zwischen den Produktionsinseln und den Knoten der Fertigungshalle verwendet wird, abhängig von ihrem jeweiligen Bereich. Dieser Ansatz ermöglicht eine nahezu optimale Caching-Performance durch Nutzung der domänenspezifischen Trennung. Dieses Caching-Verhalten ist besonders vorteilhaft in Fertigungsprozessen mit geringen Variationen zwischen den Chargen oder Einheiten.

Um dieses Caching zu ermöglichen, enthält Nimbelnet eine Toolchain, die Programme in mehrere voneinander abhängige Unterprogramme zu unterteilt. Bei der Ausführung eines Programms wird jeder Aufruf einer Abhängigkeit zunächst an die Nimblenet Laufzeitumgebung übermittelt. Diese Laufzeitkomponente löst die Abhängigkeit automatisch auf, lädt Programmteile von anderen Geräten nach und führt sie aus, einschließlich der Übergabe von Parametern und Rückgabewerten. Diese Unterprogramme können aus dem Programmcode abgeleitet werden, indem Funktionen aus Bibliotheken sortiert und

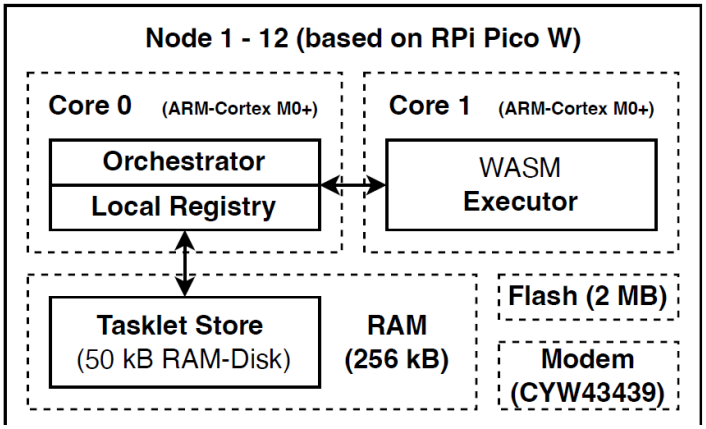

Abbildung 22: Architektur der Nimblenet Laufzeitumgebung auf einem Gerät (Node)

zusammengefasst werden. Da die Anzahl der Bibliotheksfunktionen begrenzt ist, basiert dieser Ansatz auf einer statischen Zuordnung, die die Trennung von Funktionen in Bereiche wie Steuerungen, Zeitreihendatenauswertung, zustandsändernde Datenauswertung und andere ermöglichen. Diese können dann mit den vorgestellten Caching-Bereichen verbunden werden.

Implementierung. Um eine verbesserte Migration zu ermöglichen, wurde die Nimblenet Laufzeitumgebung implementiert, bestehend aus einem lokalen Cache, einem Scheduler und mehreren Ausführungseinheiten (Executors), wie in Abbildung 22 gezeigt. Die Geräte (Knoten) sind in einem Mesh-Netzwerk organisiert, wobei Verbindungen zwischen räumlich nahen Knoten bestehen.

Wenn ein Knoten, beispielsweise ein Sensor oder eine Steuerungseinheit, eine Datenübertragung oder -verarbeitung initiieren muss, überprüft er, ob das benötigte Unterprogramm im lokalen Cache vorhanden ist. Ist das Unterprogramm verfügbar, wird es als reserviert markiert, um seine zukünftige Ausführung sicherzustellen und zu verhindern, dass es aus dem Cache entfernt wird. Fehlt es im Cache, wird das Unterprogramm von anderen Knoten oder einem zentralen Register abgerufen.

Nachladen. Das Nachladen des Unterprogramms beginnt mit einer Broadcast-Anfrage an die direkten Nachbarn (Time-to-Live, TTL = 1). Sollte kein Nachbar positiv antworten, erfolgt eine zweite, globale Broadcast-Anfrage. Der Knoten mit der geringsten Roundtrip-Zeit wird ausgewählt. Bevor der Programmcode übertragen wird, werden Metadaten, einschließlich der Größe des Unterprogramms, ausgetauscht. Der lokale Cache wird überprüft, um festzustellen, ob ausreichend Speicherplatz für den Programmcode vorhanden ist. Falls nicht, werden unnötige Unterprogramme entfernt. Anschließend wird der Code übertragen, lokal zwischengespeichert und als reserviert gekennzeichnet.

Ausführung. Nach dem Erhalt kann das Unterprogramm von einer geeigneten Ausführungseinheit ausgeführt werden. Verschiedene Executors werden unterstützt, sofern sie der erforderlichen Programmierschnittstelle (API) entsprechen. Dazu zählt auch ein "native" Executor, der Funktionen ausführen kann, die speziell an bestimmte Hardware angepasst sind oder bei Bedarf mit dem System interagieren. Die Executors basieren auf kooperativem Scheduling, bei dem sich verschiedene Prozesse die Rechenzeit teilen.

Ruft ein Unterprogramm während seiner Ausführung eine Abhängigkeit auf, wird die Kontrolle an die Laufzeitumgebung übertragen. Diese speichert den aktuellen Zustand, bereitet die Argumente vor (Marshalling) und initiiert die Ausführung des abhängigen Unterprogramms auf dem Gerät, wobei Unterprogramme bei Bedarf nachgeladen werden.

Im Falle des Wasm Executors werden zur Ausführung von Abhängigkeiten die Importfunktion von Wasm über eine Wrapper-Funktion genutzt, welche den Ausführungszustand speichert. Aufgrund der begrenzten Ressourcen einiger IoT-Geräte wird nur der Zustand des Unterprogramms erhalten, was die Wiederverwendung derselben Ausführungseinheit ermöglicht. Der Zustand umfasst den Aufrufstack, den Operandenstack, den Heap und globale Variablen. Ausgewählte Laufzeitdaten werden ebenfalls gespeichert, um den Programmneustart zu beschleunigen.

Die Wrapper-Funktion signalisiert anschließend einen Fehlerzustand an den WAMR-Executor, wodurch die Wasm-Ausführung sofort beendet wird. Das Unterprogramm wird in derselben Ausführungseinheit geladen. Um zum ursprünglichen Unterprogramm zurückzukehren, wird der gespeicherte Wasm-Zustand wiederhergestellt. Der Aufrufstack wird angepasst, indem der letzte Stackframe entfernt und der Befehlszeiger dekrementiert wird, sodass der Aufruf der Abhängigkeit erneut aufgenommen werden kann und direkt die Ergebnisse der Abhängigkeit liefert.

Löschung. Nachdem das Unterprogramm ausgeführt wurde, wird seine Reservierungsmarkierung entfernt. Anschließend greift eine Löschungsstrategie (Eviction), die vom Systemumfeld abhängt. Für die Evaluierung wurde aufgrund ihrer Einfachheit die LRU-Strategie (Least Recently Used) verwendet, die die Anwendbarkeit des vorgestellten Ansatzes effektiv veranschaulicht.

Abbildung 23: Nimblenet constrained Testbed mit 12 Raspberry Pi Pico W

Evaluation. Zur Evaluierung von Nimblenet wurde ein praxisnahes Beispiel ausgewählt. In einer Fertigungshalle wurden IoT-Sensorknoten im Abstand positioniert. Nähert sich ein AGV, wird ab einem Schwellwert eine Verbindung hergestellt. Periodisch werden verschiedene Unterprogramme für Sensorik auf diesem Knoten ausgeführt.

Für die Validierung des Ansatzes wurden zwei reale Testumgebungen eingesetzt: (1) eine „high-end“ Umgebung mit dreizehn Edge-Knoten, wobei jeder Knoten durch einen Raspberry Pi Zero 2 W repräsentiert wird, und (2) ein „constrained Testbed“ mit zwölf Knoten, die jeweils durch einen Raspberry Pi Pico W dargestellt werden. Die entsprechenden Geräte sind in Abbildung 23 dargestellt. Die erstgenannten Geräte verfügen über einen 1 GHz BCM2835-Prozessor mit 512 MB RAM. Die zweiten Geräte sind auf zwei ARM Cortex M0+ Kerne mit 133 MHz und 256 kB RAM begrenzt. Die hier präsentierte Evaluierung wurde auf der

ressourcenbeschränkten Testumgebung (2) mit kleiner Unterprogrammgröße durchgeführt. Bei der Durchführung derselben Evaluierung auf der leistungsstarken Testumgebung (1) wurden ähnliche Ergebnisse erzielt. Des Weiteren wurde jedes Evaluationsszenario auf dem eigens entwickelten Simulator ausgeführt und getestet.

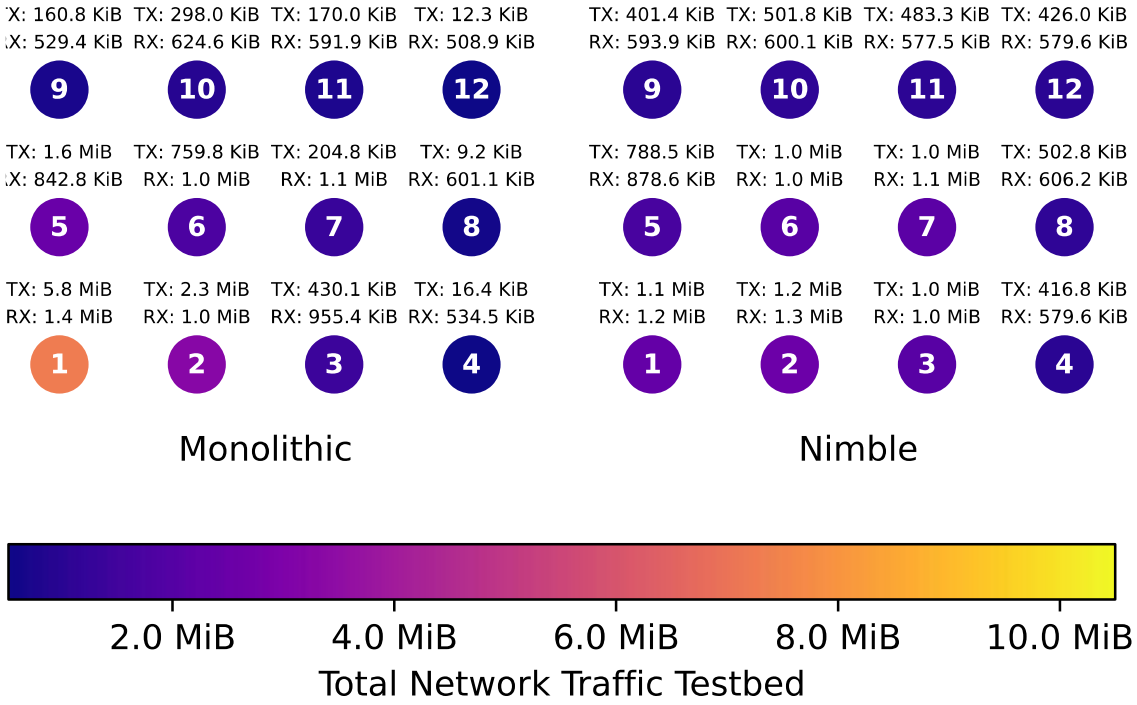

Abbildung 24: Verteilung des Netzwerkverkehrs

Verteilung des Netzwerkverkehrs. Die kombinierten Netzwerkverkehrsmessungen, bestehend aus gesendeten (TX) und empfangenen (RX) Bytes, wurden für jeden Knoten erfasst, der sowohl den Nimblenet Caching-Mechanismus als auch monolithische Anwendungen über einen Zeitraum von einer Stunde ausführt. Die Ergebnisse aus den Simulationen in Abbildung 24 sind links dargestellt, während die der realen Testumgebung rechts zu sehen sind. Die Messungen weisen eine vier- bis fünffache Reduzierung der übertragenen Bytes bei den Knoten auf, die dem Rand des Netzwerks am nächsten liegen, wenn der Nimblenet-Mechanismus verwendet wird, verglichen mit monolithischen Methoden. Dies gilt sowohl für die Simulation als auch die Testumgebung. Der Netzwerkverkehr ist gleichmäßiger über alle Knoten verteilt, was die Last von den Randknoten nimmt und das Netzwerk so gleichmäßiger ausnutzen kann. Die Gesamtmenge der Daten bleibt in beiden Szenarien unverändert, da identische Unterprogramme verwendet werden.

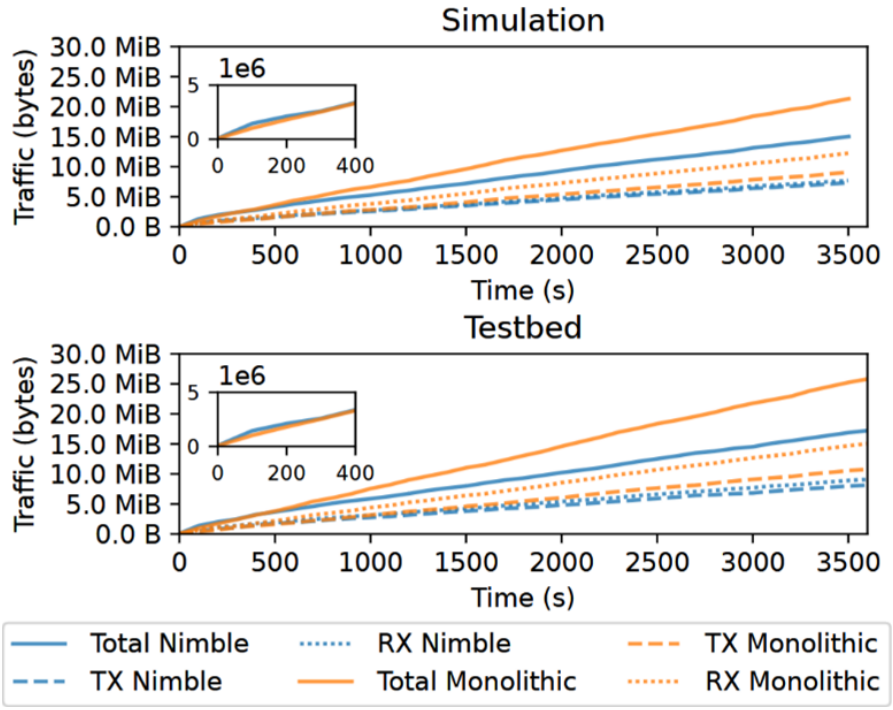

Abbildung 25: Nezwerkauslastung pro Zeit

Gesamte Netzwerklast Die zeitliche Entwicklung der Netzwerklast wurde sowohl in der Testumgebung als auch in der Simulation aufgezeichnet und sind in Abbildung 25 zu sehen. Anfangs ist die Gesamtlast für die Nimblenet-Lösung moderat erhöht, was auf den zusätzlichen Overhead durch Routing und Abfragen zurückzuführen ist. Ein Gleichgewicht wird in der Simulation bei t = 300 s und in der Testumgebung bei t = 400 s erreicht. Ab diesem Zeitpunkt wird die Effizienz der On-Device-Caches und der Caching-Strategien deutlich. Der in den Simulationen beobachtete Trend zeigt sich auch in den Testbeds. Allerdings weist die Testumgebung einen leichten Anstieg der gesamten Netzwerklast im Vergleich zur Simulation auf, bedingt durch zusätzliche Verwaltungsdatenpakete und Paketverluste aufgrund von Hardwarebeschränkungen in der realen Umgebung.

Antwortzeit. Die durchschnittliche Antwortzeit für alle abgeschlossenen Unterprogramme innerhalb eines gleitenden Fensters von 100 s wurde ermittelt. Trotz nahezu identischer Ausführungszeiten der Unterprogramme ermöglicht der Nimblenet-Mechanismus ein schnelleres Laden im Vergleich zur monolithischen Methode, da die Vorteile des Cachings genutzt werden. Dies ist besonders vorteilhaft bei einer signifikanten Verzögerung von etwa 100 ms pro Hop im Mesh-Netzwerk oder wenn die Verbindungsgeschwindigkeit eingeschränkt ist, wie bei größeren Unterprogrammen beobachtet. Die geringe durchschnittliche Ausführungsverzögerung des Caching-Mechanismus von etwa 30 ms

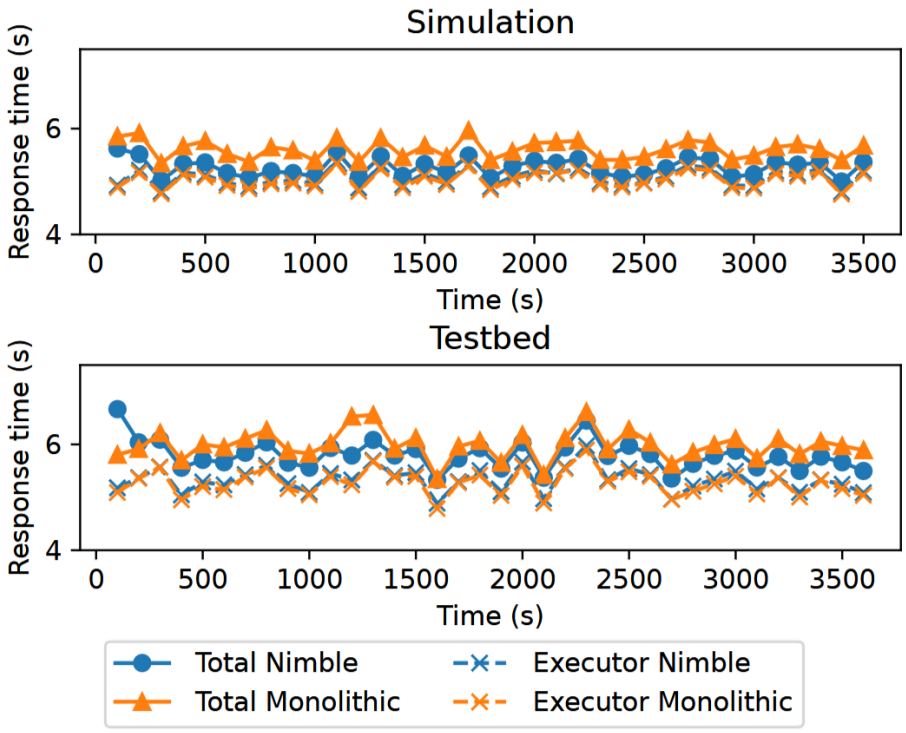

Abbildung 26: Antwortzeiten im zeitlichen Verlauf

resultiert aus zusätzlichen Code-Ladevorgängen und -Initialisierungen. Die Ergebnisse passen konsistent zu den vorher präsentierten Messreihen, die denselben Break-Even-Punkt zwischen 300 und 400 s in der realen Testumgebung zeigen. Sowohl die Simulation als auch die tatsächliche Ausführung in der Testumgebung zeigen ein ähnliches Anfangsmuster, bei dem die gesamten Antwortzeiten signifikant ansteigen, bis die meisten Aufgaben im System verbreitet sind. Nach ungefähr 400 s in der Testumgebung werden die Verbesserungen der Antwortzeit durch die Beschaffung bei unmittelbaren Nachbarn und das lokale Caching sichtbar, und der Caching-Mechanismus übertrifft den monolithischen Ansatz.

Zusammenfassend zeigt die Evaluierung die Effektivität des vorgeschlagenen Nimblenet Caching-Mechanismus für die Migration in Industrie- und Fertigungsanlagen. Durch die Reduzierung der Netzwerklast und die Verbesserung der Antwortzeiten trägt der Mechanismus zur Optimierung der Netzwerkleistung bei, was eine effektive dynamische Betriebsführung ermöglicht.

7 Projektergebnisse

Im Rahmen von Mannheim EMDRIVE wurden mehrere bestehende Konzepte erweitert sowie neue Software- und Systemkonzepte erarbeitet. Diese wurden mehrfach durch Publikationen veröffentlicht und werden im Folgenden noch einmal zusammengefasst.

7.1 Konzepte und Software

Durch die implementierten QoS-Strategien wurden erhebliche Fortschritte in der Überwachung und Anpassung der Fertigungsprozesse erreicht. Die fortlaufende Berechnung der Allan-Varianz ermöglichte eine detaillierte Charakterisierung der verschiedenen Rauschanteile in den Beschleunigungssignalen, wodurch insbesondere Weißes Rauschen, Flicker-Rauschen und Random Walk präzise identifiziert wurden. Die Analyse enthüllte signifikante Unterschiede in den Rauschkennwerten der einzelnen Achsen des dreiachsigen Beschleunigungssensors, was auf unterschiedliche Sensitivitäten oder potenzielle Kalibrierungsabweichungen hindeutete. Durch die Integration der Fault Diagnosis konnten Anomalien wie mechanische Lockerheiten oder Lagerdefekte frühzeitig detektiert werden, was anhand der Spektrogramme deutlich wurde, in denen eine Zunahme des weißen Rauschens und spezifischer Frequenzkomponenten beobachtet wurde. Die daraus abgeleiteten QoS-Parameter führten zu einer adaptiven Anpassung der Betriebsmodi, indem beispielsweise Filter dynamisch aktiviert oder Wartungsprozesse eingeleitet wurden. Dies resultierte in einer messbaren Reduktion von Ausfallzeiten und einer Steigerung der Gesamtanlageneffektivität, was die Effektivität des entwickelten QoS-Konzepts für eine dynamische und resiliente Fertigungssteuerung unterstreicht.

Des Weiteren wurde im Projekt gezeigt, wie Wasm-IO das Integrieren von Low-Level-Gerätetreibern in Wasm ermöglicht. Hardwareanforderungen wurden plattformübergreifend ausgedrückt und in Wasm Binärdateien eingefügt. Wasm-IO ermöglicht außerdem synchrones I/O und implementiert dazu verschiedene Ansätze, welche im Projekt evaluiert wurden. Zudem wurde die Unterstützung für Interrupt-Verarbeitung in Wasm eingeführt. Durch die Erweiterung des traditionellen Systemmodells um eine zusätzliche Prioritätsebene E1/4, die explizit für Wasm-Services reserviert ist, wird der zeitliche Einfluss eines Wasm-Services vom restlichen System entkoppelt. Die Evaluierung zeigt, dass Hardware-Interaktionen und Interrupt-Latenzen akzeptabel sind, gleichzeitig eine starke zeitliche Isolation und die Vorteile der Containerisierung ermöglichen und so die zukünftige, dynamische Fertigung erst möglich macht.

Mit Nimblenet wurde ein neues Modell zur Ausführung verteilter Programme in der zellbasierten Fertigung erarbeitet. Programme werden dabei partitioniert und in verschiedene Unterprogramme aufgeteilt. Verschiedene Executor (Native, Python und WebAssembly) und eine neuen Methode zur kooperativen Zusammenarbeit zwischen Laufzeitumgebung ermöglicht die Unterbrechung der Ausführung der Teilprogramme und die Übergabe des Kontrollflusses an die Laufzeitumgebung, gefolgt vom Nachladen anderer Programmteile. So können Programme partitioniert ausgeführt werden. Außerdem wurde gezeigt, wie Caching-Domänen in industriellen Umgebungen effizient konstruiert werden können. Die räumlichen Abhängigkeiten ermöglichen ein Caching auf verschiedenen Ebenen: gerätelokal für Hardwarespezifika, fertigungsinsellokal für ähnliche Algorithmen und mit dem

Werkstück wandernd für werkstückspezifische Eigenschaften. Die Implementierung und Evaluierung auf kleinen Geräten wie dem Raspberry Pi Zero sowie durch Simulationen zeigt eine bessere Netzwerklastverteilung sowie, nach dem Einschwingverhalten, schnellere Antwortzeiten und weniger Netzwerkkommunikation. Erst mit diesen verbesserten Eigenschaften ist die Anwendbarkeit der dynamischen Fertigung skalierbar, da die sonst erhöhte Netzwerkkommunikation eine steigende Komplexität unrentabel macht.

7.2 Demonstratoren

Abbildung 27: Foto der beiden Tischdemonstratoren. Links ist der QoS Demonstrator mit Sensor, Motor, Anzeige und Schaltschrank zu sehen, rechts der Migrationsdemonstrator mit Sensor und Edge Gerät sowie Monitoren zur Darstellung der jeweiligen Workloads.

Um die gewonnenen Erkenntnisse in einen praktischen Kontext einzubetten und die Relevanz zu demonstrieren, wurde außerdem viel Zeit auf die Demonstration an einem Beispielaufbau aufgewendet. Im Zuge des Projekts entstand ein Tischdemonstrator, sowie die Erweiterung eines zweiten Tischdemonstrators, aus dem GEMIMEG Förderprojekt. Zusätzlich wurde ein Video produziert, welches bereits in Kapitel 5 zum Einstieg beschrieben wurde.

Der QoS Demonstrator ist in Abbildung 27 links zu sehen. Er besteht aus einem Sensor, einem Motor, einem Bildschirm sowie einem Schaltschrank mit integriertem Edge Gerät. Der SSI-Multisensor ist oben auf dem Motorengehäuse verschraubt und ermöglicht so, die durch den Motor erzeugten Vibrationen mittels des verbauten MEMS-Sensors zu protokollieren. Der Motor lässt sich in verschiedenen Betriebsgeschwindigkeiten betreiben, wodurch sich

verschiedene Umgebungssituationen und Betriebsmodi simulieren lassen. Berechnungen lassen sich entweder auf diesem Multisensor oder auf dem im Schaltschrank verbauten Edge Gerät ausführen. Der Schaltschrank bietet außerdem die weitere Infrastruktur wie Spannungsversorgung, Sicherheitsbauteile, WiFi Infrastruktur sowie eine Status Anzeige durch verbaute LEDs. Das QoS Dashboard, im Foto auf dem Bildschirm zu sehen, stellt die verschiedenen QoS Werte in unterschiedlichen Darstellungen dar. Dabei lässt sich zwischen detaillierten Anzeigen und kumulierten Anzeigen in unterschiedlicher Granularität wechseln. Schaltschrank und Bildschirm stammen aus dem GEMIMEG II Projekt. Für das Mannheim EMDRIVE Projekt wurde der Motor ergänzt. Der Chip des Sensors wurde durch den eingeführten ARM Cortex-M33 Core ersetzt sowie eine vollständig andere Programmierung des Gesamtsystems, nun basierend auf dem vorgestellten Zephyr RTOS. Des Weiteren wurden sowohl in das Edge Gerät als auch in den Multisensor eine vollständige Wasm Laufzeitumgebung integriert, im Falle des Sensors inklusive Wasm-IO.

Abbildung 27 zeigt rechts außerdem den finalen Migrationsdemonstrator. Oben ist das Edge Gerät zu sehen, ein Siemens SIMATIC IPC520A Industrie-PC. Unten, am Ende des Leuchtbands befindet sich der Mikrocontroller, welcher das Multisensorsystem repräsentiert. Ein an der Rückseite versteckter Rechner agiert als emulierter Sensor und sendet die Daten wahlweise direkt an den Mikrocontroller oder das Edge Gerät. Als repräsentatives Programm wird eine Schnelle Fouriertransformation (FFT) durchgeführt, welche auf beiden Geräten, als Wasm Container gekapselt, ausgeführt werden kann. Ein extern gegebenes Signal, was das Erreichen des QoS Schwellwert repräsentiert, stößt die Migration des Wasm Containers von einem zu anderen Gerät an. Die beiden Bildschirme stellen dabei ständig die aktuelle Ausgabe des jeweiligen Geräts graphisch dar. Dies ermöglicht es, die Migration nachzuverfolgen.

8 Verwertung

Im Mannheim EMDRIVE Projekt wurde die dynamische Betriebsführung im industriellen Umfeld untersucht. Somit konnten erfolgreich erste Schritte in der Entwicklung einer Basistechnologie gemacht werden. Dabei wurden zwei Doktoranden sowie vier Masterandinnen und Masteranden mit den Inhalten ausgebildet und teilweise finanziert. Außerdem wurden die folgenden Verwertungen generiert bzw. an ihnen mitgewirkt:

Publikationen

1. M. Kreutzer, M. L. Seidler, V. Pazmino Betancourt, und J. Becker, „Work-in-Progress: Integrating WebAssembly into Service-Oriented Architectures for Edge Systems“, in *Proceedings of the International Conference on Embedded Software*, in EMSOFT ’23. New York, NY, USA: Association for Computing Machinery, Jan. 2024, S. 17–18. doi: 10.1145/3607890.3608456.

2. K. Müller, M. Seidler, P. Ulbrich, und N. Franchi, „NimbleNet: Serverless Computing for the Extreme Edge in Factory Environments“, in *Proceedings of the 10th International Workshop on Serverless Computing*, in WoSC10 ’24. New York, NY, USA: Association for Computing Machinery, Dez. 2024, S. 19–24. doi: 10.1145/3702634.3702953.
3. M. Kreutzer, M. Seidler, K. Dudzik, V. P. Betancourt, und J. Becker, „Migration of Isolated Application Across Heterogeneous Edge Systems“, in *2024 IEEE 8th International Conference on Fog and Edge Computing (ICFEC)*, Mai 2024, S. 64–70. doi: 10.1109/ICFEC61590.2024.00014.
4. M. Seidler, A. Krause, und P. Ulbrich, „Extending Lifetime of Embedded Systems by WebAssembly-based Functional Extensions Including Drivers“, 10. März 2025, *arXiv*: arXiv:2503.07553. doi: 10.48550/arXiv.2503.07553.
5. M. Seidler, A. Krause, und P. Ulbrich, „Wasm-IO: Enabling Low-Level Device Interaction in WebAssembly for Industry Automation“, in ACM Trans. Embedd. Comput. Syst. (TECS), Januar 2026, doi: 10.1145/3760387
6. P. Schmidt *u. a.*, „EMDRIVE Architecture: Embedded Distributed Computing and Diagnostics from Sensor to Edge“, in *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, März 2024, S. 1–6. doi: 10.23919/DATE58400.2024.10546796.

9 Zusammenfassung und Ausblick

Im Rahmen des Mannheim EMDRIVE Projekts hat Siemens wegweisende Basistechnologie im Bereich der dynamischen Betriebsführung von IoT-basierten Sensor-to-Edge-Systemen untersucht. Das Teilvorhaben fokussierte sich auf die Verbesserung der Datenqualität in der Fertigung komplexer mechatronischer Systeme, insbesondere im automobilen Umfeld. Dazu wurde ein Anwendungsfall der modularen Fertigung untersucht und durch die dynamische Betriebsführung erweitert. Konzeptionell wurde die Erkennung und Auslösung zur dynamischen Betriebsführung erarbeitet, ausgelöst durch eine Änderung in der Sensordatenqualität, beispielsweise durch die sich regelmäßig ändernde Umgebung. Des Weiteren wurden die Basistechnologien zur Umsetzung der dynamischen Betriebsführung untersucht und eingeführt. Verschiedene Containerisierungstechnologien wurden untersucht, um eine Ausführung auf unterschiedlicher Hardware zu ermöglichen. WebAssembly (Wasm), eine mögliche Art der Containerisierung, wurde erweitert um die Ansteuerung von Peripheriegeräten, was den Einsatz auf ressourcenbeschränkten Systemen erst möglich macht. Des Weiteren wurden speziell für die Industrie optimierte Cachingverfahren entwickelt, die, auf Basis von Wasm Container, die Kommunikationsraten senken – ein kritisches Ziel bei sich ständig erhöhender Intelligenz im Feld.

Die wissenschaftliche Bedeutung des Projekts spiegelt sich in mehreren Publikationen wider, wobei insbesondere die Ausbildung von zwei Doktoranden sowie vier Masterandinnen und Masteranden hervorzuheben ist. Die erarbeiteten Basistechnologien und -methoden bilden die notwendige Grundlage für dynamische Betriebsführung, wobei sie sich andererseits auch auf dieses Beschränken. Eine ausgiebige Evaluation der Auswirkungen auf nicht-funktionale Eigenschaften wie Auslastung oder Energieverbrauch wurde im Projekt nur qualitativ und exemplarisch durchgeführt, da dies ein gesondertes Forschungsgebiet ist. Insbesondere die Zuweisung von Arbeit auf ein bestimmtes Gerät und die zugehörigen Rahmenbedingungen bilden ein komplexes Optimierungsproblem und müssen in Zukunft untersucht werden. Des Weiteren wurde durch die Untersuchungen im Projekt klar, dass eine Anwendung von Wasm auf Mikrocontrollern insbesondere abhängig von dessen Echtzeitverhalten ist. Auch ist bisher keinerlei Literatur zu finden, weshalb weitere Nachforschungen betrieben werden muss.

Insgesamt hat das Mannheim EMDRIVE Projekt wertvolle Einblicke in die Grundlagen zur Realisierung einer dynamischen Betriebsführung im industriellen Umfeld geliefert und so einen Beitrag zur zukünftigen Fertigung von Automobilen und damit dem Wirtschaftsstandort Deutschland beigetragen.

Literaturverzeichnis

[1] W. Schröder-Preikschat, „Betriebssystemarchitektur“, gehalten auf der Vorlesung Betriebssystemtechnik, Erlangen, 2016.
[2] R. Love, *Linux kernel development*, 3rd Aufl. Addison-Wesley, 2010.
[3] K. Fizza und others, „A survey on evaluating the quality of autonomic internet of things applications“, *IEEE Communications Surveys & Tutorials*, Bd. 25, Nr. 1, S. 567–590, 2023.
[4] Perez-Castillo und others, „Data quality best practices in IoT environments“, in *2018 11th international conference on the quality of information and communications technology (QUATIC), coimbra, portugal*, 2018, S. 272–275.
[5] T. Ruhland und others, „Data quality in IoT temperature sensor systems: Demonstrated on time-dependent temperature fluctuations“, *IEEE Sensors Journal*, Bd. 24, Nr. 16, S. 25960–25971, 2024.
[6] T. Engel, „GEMIMEG-II — How metrology can go digital...“, *Measurement Science and Technology*, Bd. 34, Nr. 10, S. 104002, Juli 2023.
[7] T. Engel, „Method and sensing system for determining an overall quality indicator qox for a sensor based measurement“, EP4425352, 4. September 2024
[8] A. P. Vedurmudi, J. Neumann, M. Gruber, und S. Eichstädt, „Semantic description of quality of data in sensor networks“, *Sensors*, Bd. 21, Nr. 19, S. 6462, 2021.
[9] H. Y. Teh, A. W. Kempa-Liehr, und K. I.-K. Wang, „Sensor data quality: a systematic review“, *Journal of Big Data*, Bd. 7, Nr. 1, S. 11, Feb. 2020, doi: 10.1186/s40537-020-0285-1.
[10] *Devicetree Specification*, 28. Juni 2023. Zugegriffen: 31. März 2023. [Online]. Verfügbar unter: https://github.com/devicetree-org/devicetree-specification/releases/v0.4
[11] *WebAssembly Specification 1.0*, Specification, 19. April 2022. Zugegriffen: 7. September 2023. [Online]. Verfügbar unter: https://www.w3.org/TR/wasm-core-2/
[12] A. Haas *u. a.*, „Bringing the web up to speed with WebAssembly“, in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, in PLDI 2017. New York, NY, USA: Association for Computing Machinery, Juni 2017, S. 185–200. doi: 10.1145/3062341.3062363.
[13] R. Liu, L. Garcia, und M. Srivastava, „Aerogel: Lightweight Access Control Framework for WebAssembly-Based Bare-Metal IoT Devices“, in *2021 IEEE/ACM Symposium on Edge Computing (SEC)*, Dez. 2021, S. 94–105. doi: 10.1145/3453142.3491282.
[14] J. Ménétrey, M. Pasin, P. Felber, und V. Schiavoni, „Twine: An Embedded Trusted Runtime for WebAssembly“, in *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, Apr. 2021, S. 205–216. doi: 10.1109/ICDE51399.2021.00025.
[15] G. Peach, R. Pan, Z. Wu, G. Parmer, C. Haster, und L. Cherkasova, „eWASM: Practical Software Fault Isolation for Reliable Embedded Devices“, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Bd. 39, Nr. 11, S. 3492–3505, Nov. 2020, doi: 10.1109/TCAD.2020.3012647.
[16] S. Wallentowitz, B. Kersting, und D. M. Dumitriu, „Potential of WebAssembly for Embedded Systems“, in *2022 11th Mediterranean Conference on Embedded Computing (MECO)*, Juni 2022, S. 1–4. doi: 10.1109/MECO55406.2022.9797106.

[17] W. Wang, „How far we've come – a characterization study of standalone WebAssembly runtimes", in *2022 IEEE international symposium on workload characterization (IISWC)*, Nov. 2022, S. 228–241. doi: 10.1109/IISWC55918.2022.00028.

[18] E. Wen und G. Weber, „Wasmachine: Bring IoT up to Speed with A WebAssembly OS", in *2020 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, März 2020, S. 1–4. doi: 10.1109/PerComWorkshops48775.2020.9156135.

[19] S. Yoo, M. Park, und C. Yoo, „A Step to Support Real-Time in Virtual Machine", in *2009 6th IEEE Consumer Communications and Networking Conference*, Jan. 2009, S. 1–7. doi: 10.1109/CCNC.2009.4784876.

[20] „The WAMR memory model", WAMR. Zugegriffen: 6. September 2023. [Online]. Verfügbar unter: https://bytecodealliance.github.io/wamr.dev/blog/the-wamr-memory-model/

[21] „Understand the WAMR stacks", WAMR. Zugegriffen: 3. Oktober 2023. [Online]. Verfügbar unter: https://bytecodealliance.github.io/wamr.dev/blog/understand-the-wamr-stacks/

[22] D. Gohman *u. a.*, *WebAssembly/WASI: v0.2.2*, 3. Oktober 2024. doi: 10.5281/zenodo.13888155.

[23] C. Watt, „Mechanising and verifying the WebAssembly specification", in *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, in CPP 2018. New York, NY, USA: Association for Computing Machinery, Jan. 2018, S. 53–65. doi: 10.1145/3167082.

[24] C. Watt, X. Rao, J. Pichon-Pharabod, M. Bodin, und P. Gardner, „Two Mechanisations of WebAssembly 1.0", in *Formal Methods*, M. Huisman, C. Păsăreanu, und N. Zhan, Hrsg., Cham: Springer International Publishing, 2021, S. 61–79. doi: 10.1007/978-3-030-90870-6_4.

[25] C. Watt, M. Trela, P. Lammich, und F. Märkl, „WasmRef-Isabelle: A Verified Monadic Interpreter and Industrial Fuzzing Oracle for WebAssembly", *Proc. ACM Program. Lang.*, Bd. 7, Nr. PLDI, S. 110:100-110:123, Juni 2023, doi: 10.1145/3591224.

[26] W. Zaeske, S. Friedrich, T. Schubert, und U. Durak, „WebAssembly in Avionics : Decoupling Software from Hardware", in *2023 IEEE/AIAA 42nd Digital Avionics Systems Conference (DASC)*, Okt. 2023, S. 1–10. doi: 10.1109/DASC58513.2023.10311207.

[27] *Regulation (EU) 2019/881 of the European Parliament and of the Council of 17 April 2019 on ENISA (the European Union Agency for Cybersecurity) and on information and communications technology cybersecurity certification and repealing Regulation (EU) No 526/2013 (Cybersecurity Act) (Text with EEA relevance)*, Bd. 151. 2019. Zugegriffen: 12. März 2025. [Online]. Verfügbar unter: http://data.europa.eu/eli/reg/2019/881/oj/eng

[28] K. Zandberg und 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)*, Dez. 2020, S. 1–6. doi: 10.23919/PEMWN50727.2020.9293081.

[29] M. Seidler, A. Krause, und P. Ulbrich, „Extending Lifetime of Embedded Systems by WebAssembly-based Functional Extensions Including Drivers", 10. März 2025, *arXiv*: arXiv:2503.07553. doi: 10.48550/arXiv.2503.07553.

[30] M. Seidler, A. Krause, und P. Ulbrich, „Wasm-IO: Enabling Low-Level Device Interaction in WebAssembly for Industry Automation“, *TECS*, doi: 10.1145/3760387.

[31] Heejin Ahn, J. Bastien, D. Gohman, und D. L. Schuff, „WebAssembly/tool-conventions: C++ volatile support“, GitHub. Zugegriffen: 5. März 2025. [Online]. Verfügbar unter: https://github.com/WebAssembly/tool-conventions/issues/60

[32] R. Kurte, „Embedded WASM“, GitHub. Zugegriffen: 29. März 2023. [Online]. Verfügbar unter: https://github.com/embedded-wasm

[33] B. Li, H. Fan, Y. Gao, und W. Dong, „Bringing webassembly to resource-constrained iot devices for seamless device-cloud integration“, *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*, 2022, doi: 10.1145/3498361.3538922.

[34] R. G. Singh und C. Scholliers, „WARDuino: a dynamic WebAssembly virtual machine for programming microcontrollers“, *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*, 2019, doi: 10.1145/3357390.3361029.

[35] K. Müller, M. Seidler, P. Ulbrich, und N. Franchi, „NimbleNet: Serverless Computing for the Extreme Edge in Factory Environments“, in *Proceedings of the 10th International Workshop on Serverless Computing*, in WoSC10 ’24. New York, NY, USA: Association for Computing Machinery, Dez. 2024, S. 19–24. doi: 10.1145/3702634.3702953.