Bei diesem Beitrag handelt es sich um einen wissenschaftlich begutachteten und freigegebenen Fachaufsatz („reviewed paper“).

doi.org/10.37544/1436-4980-2026-06-66
Datum der Einreichung: 20.03.2026
Datum der Annahme: 14.04.2026
Datum der Veröffentlichung: 27.07.2026

Mit White-Box-Tests versteckte Latenzquellen in Container-Runtimes finden

Benchmarking von Container-Runtimes für vPLCs

M. Walker, R. Neumann, M. Richter, M. Fischer, A. Lechler, A. Verl

ZUSAMMENFASSUNG Container-Technologien wie Docker und Podman bieten Vorteile für die Bereitstellung virtualisierter Steuerungen (vPLCs). Deren Isolationsmechanismen können jedoch das Echtzeitverhalten beeinflussen. Bisherige Studien nutzen Black-Box-Tests, die keine Rückschlüsse auf konkrete Ursachen ermöglichen. Dieser Beitrag stellt eine White-Box-Methodik vor, mit der einzelne Mechanismen systematisch bewertet werden, um Konfigurationsentscheidungen für echtzeitkritische Containeranwendungen zu treffen.

STICHWÖRTER
Virtualisierung, vPLC, Steuerungstechnik

Benchmarking container runtimes for vPLCs – Identifying hidden sources of latency in container runtimes using white-box testing

ABSTRACT Container technologies such as Docker and Podman offer advantages for deploying virtualized programmable logic controllers (vPLCs). However, their isolation mechanisms can affect real-time behavior. Previous studies have used black-box testing, which does not allow conclusions to be drawn about specific causes. This paper presents a white-box methodology that systematically evaluates individual mechanisms to inform configuration decisions for real-time-critical container applications.

1 Einleitung

Die industrielle Automatisierung unterliegt zunehmendem Anpassungsdruck. Traditionelle Steuerungssysteme sind stark durch die enge Kopplung von Hardware und Software limitiert und lassen sich nur schwer an veränderte Produktionsanforderungen anpassen. Neue Konzepte wie Software-defined Manufacturing (SDM) zielen darauf ab, diese Bindung aufzulösen und Produktionsfunktionen über Software zu realisieren [1, 2]. Durch die Trennung von Hardware und Steuerungssoftware können Funktionen unabhängig von spezifischer Hardware bereitgestellt und dynamisch angepasst werden [3]. Ein vielversprechender Ansatz zum Erreichen dieser Flexibilität ist die Virtualisierung von Steuerungsanwendungen [4, 5]. Dabei werden Steuerungsfunktionen als Software-Services auf allgemeinen Rechenressourcen ausgeführt [6]. Eine leichtgewichtige Form der Virtualisierung sind Software-Container, die zunehmend in der industriellen Automatisierung eingesetzt werden [7]. Container ermöglichen, bei richtiger Systemkonfiguration, die konsolidierte Ausführung mehrerer Echtzeitanwendungen auf derselben Hardware und erlauben eine flexible Bereitstellung, Skalierung und Wartung von Steuerungssoftware [8, 9].

Um Steuerungsmodule in Form von Containern in Automatisierungssystemen einzusetzen, ist es notwendig, die Performance verschiedener Container-Runtimes zu kennen. Benchmarking von Container-Runtimes liefert die Grundlage, passende Runtimes für entsprechende Echtzeitanforderungen in virtualisierten Steuerungsumgebungen auszuwählen. Dieser Beitrag untersucht systematisch die Performance unterschiedlicher Container-Runtimes anhand darunterliegender Mechanismen und liefert damit die Basis zur Vorhersage der Echtzeit-Performance von containerisierten Services in der Automatisierungstechnik.

2 Grundlagen

Die Nutzung von Containern zur Isolation von Echtzeitapplikationen ist für Anwender attraktiv, da diese deutlich weniger Ressourcen als virtuelle Maschinen beanspruchen. Dies ist vor allem darauf zurückzuführen, dass Container sich das Betriebssystem des Container-Hosts teilen und jeweils nur die für die Ausführung der Anwendung benötigten Komponenten bereitstellen. Daraus folgt, dass die Performance der containerisierten Anwendungen stark von der Performance des Hostsystems abhängig ist.

Um eine Verwaltung und Ausführung von Containern mit unterschiedlichen Systemen zu ermöglichen, hat die Open Container Initiative (OCI) [10] die benötigten Komponenten in Spezifikationen beschrieben. Dies bietet eine Interoperabilität zwischen verschiedenen Implementierungen und ermöglicht es, Container zwischen den Implementierungen auszutauschen und mithilfe unterschiedlicher Runtimes auf einer Vielzahl von Plattformen auszuführen.

Die Spezifikationen der OCI sind in eine Image-Spezifikation [11], eine Distribution-Spezifikation [12] sowie eine Runtime-Spezifikation [13] aufgeteilt. Die Image-Spezifikation beschreibt den strukturellen Aufbau sowie den Inhalt eines Container-Images, sodass dieses genutzt werden kann, um aus dem Image eine lauffähige Container-Instanz zu erstellen.

In der Distribution-Spezifikation [12] wird der Aufbau der Programmschnittstelle zur Verteilung von Inhalten innerhalb

eines Containerökosystems beschrieben, sodass Werkzeuge verschiedener Hersteller diese einheitliche Schnittstelle zum Austausch von Inhalten verwenden können. Die Spezifikation legt beispielsweise fest, wie Inhalte in eine Registry geladen und von dieser an anderer Stelle wieder bezogen werden können. Im Zuge dessen werden die notwendigen Schritte für den Up- und Download sowie für die Benennung und Identifikation von Inhalten beschrieben. Hierbei wird außerdem der Mechanismus, mit welchem Containerverwaltungsprogramme Inhalte innerhalb der Registry auffinden können, beschrieben.

Die OCI-Runtime-Spezifikation [13] legt den Schwerpunkt auf die Ausführung einer Container-Instanz. In dieser wird der Lebenszyklus eines Containers sowie die zulässigen Kommandos, zum Beispiel zum Starten und Stoppen des Containers in seinem Lebenszyklus, beschrieben. Außerdem wird festgelegt, wie die Konfiguration des Containers vorgenommen werden kann und welche Eigenschaften die Runtimes auf verschiedenen Systemen erfüllen müssen. Weiterhin wird definiert, wie diese konfiguriert beziehungsweise deren Verhalten angepasst werden kann. Die Modularisierung der Container-Engines (**Bild 1**) erlaubt es, die Komponenten flexibel durch andere Implementierungen auszutauschen.

So bieten „Docker“ [14] und „Podman“ [15] jeweils eigene Kommandozeilenapplikationen, mit welchen der Anwender Container erstellen und verwalten kann. Diese kommunizieren mittels einer spezifizierten API (Application Programming Interface) über ein Socket mit den darunterliegenden Komponenten der Container-Engines. Wie in Bild 1 zu erkennen ist, sind die Container-Engines nicht fest an eine spezifische Runtime gebunden, sondern können durch andere, OCI-kompatible Implementierungen ersetzt werden. Dies erlaubt eine einfache Anpassung des Systems an die anwendungsspezifischen Anforderungen des Nutzers. Die jeweiligen Implementierungen der Runtimes wie runc, crun und youki stellen die OCI-kompatible Schnittstelle zum Starten von auf dem aus der Registry bereitgestellten Image basierenden Containern bereit. Intern können diese wiederum Bibliotheken wie „ibcontainer” verwenden, um mit dem Kernel zu kommunizieren und diesen für den Start des Containers einzurichten. Die Zwischenschichten „containerd” [16] beziehungsweise „conmon” [17] überwachen die laufenden Container und bereiten im Fall von containerd zusätzlich das System für den Start der Container durch die Runtime vor, indem etwa die Containerinstanz aus dem Image erzeugt wird.

Im Gegensatz zu virtuellen Maschinen, die jeweils ein eigenes Gastbetriebssystem besitzen, verwenden sowohl Docker als auch Podman die vom Linux-Kernel des Hostsystems bereitgestellten Technologien, um Container sowohl gegeneinander als auch gegen den Host zu isolieren. Zu diesen gehören Namespaces, Control Groups (Cgroups), Capabilities (CAP), Secure Computing (Seccomp), AppArmor, SELinux und Change Root (Chroot). Bei der Umsetzung der Isolation unterscheiden sich die Container-Engines meist in der Auswahl der Standardkonfiguration, welche die Performance der Engines beeinflussen kann.

Namespaces werden dazu verwendet, jedem Container, beziehungsweise allen Containern im selben Namespace, eine eigene Sicht auf das System zu bieten, sodass die Prozesse so ausgeführt werden, als würden nur diese auf dem System laufen. Die Verwaltung von Systemressourcen eines Prozesses erfolgt mittels Cgroups. Mit diesen kann festgelegt werden auf welche Ressourcen ein Prozess, oder Container, zugreifen darf und wie viel der angegebenen Ressource verwendet werden darf.

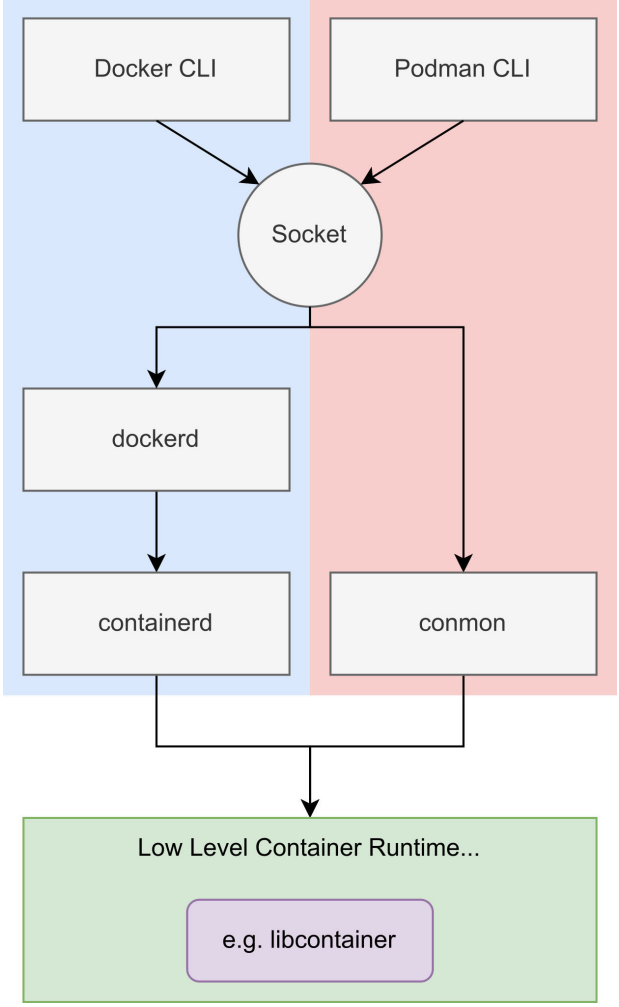

Bild 1 Modularisierung der Container-Engines. Pfeile: Alternative Implementierungen. *Grafik: ISW*

Die Technologien AppArmor und SELinux bieten Mandatory Access Control (MAC). Mit MAC kann der Zugriff eines Containers auf Dateien und Systemressourcen eingeschränkt werden. Um die möglichen Systemaufrufe eines containerisierten Prozesses kontrollieren zu können, werden Systemaufruffilter auf Basis von Seccomp und AppArmor verwendet. Um zudem den Zugriff des containerisierten Prozesses auf das System einzuschränken und aus Sicht des Containers auf einem eigenen root-Dateisystem auszuführen, wird mit Chroot ein Subverzeichnis des Hostsystems als Root-Dateisystem des Containers festgelegt.

Um einer Anwendung innerhalb des Containers die Nutzung des Netzwerks zu erlauben, bieten sowohl Podman als auch Docker mehrere Mechanismen, die über die Konfiguration des Containers ausgewählt werden können. Häufig genutzte Varianten sind das Host- sowie das Bridge-Netzwerk, wobei bei ersterem die Container direkt in das Netzwerk des Hostsystems integriert werden und nicht weiter isoliert sind. Bei Nutzung eines Bridge-Netzwerks können auf dem Host eine oder mehrere Netzwerkbridges erstellt und mit Containern verbunden werden. Somit sind Container, welche mit unterschiedlichen Bridges ver-

bunden sind, gegeneinander isoliert. Container, die mit derselben Bridge verbunden sind, können jedoch miteinander kommunizieren, als wären sie im selben Netzwerk. Um in diesem Fall zusätzlich eine Kommunikation zwischen dem Hostnetzwerk und den Containern, die mit der Bridge verbunden sind, zu erlauben, legt Docker iptables-Regeln an. Diese leiten eingehende Pakete mit Network Address Translation (NAT) an die Bridge, oder die gewünschten Container, weiter. Da die Einrichtung der Netze und Regeln unter Linux als privilegierte Operation erweiterte Rechte benötigt, bietet *Podman* zusätzlich die Möglichkeit der User-Space-Netzwerkimplementierung „pasta" [18]. Mit pasta können Netzwerke für die Container ohne root-Rechte realisiert werden. Dafür stellt pasta den Containern die Netzwerkinterfaces bereit und routet die Pakte zwischen den Teilnehmern des Netzwerks.

Alle vorgestellten Mechanismen können Einfluss auf das Ausführungsverhalten einer Anwendung nehmen. Beispielsweise verursachen die Systemaufruffilter Seccomp und AppArmor bei jedem Systemaufruf zusätzliche Latenzen. Dies führt sowohl bei Echtzeit- als auch bei Nicht-Echtzeit-Systemen zu veränderter Performance in der Anwendungsausführung und Kommunikation. Prinzipbedingt verwenden alle Container-Engines dieselben Isolationsmechanismen. Da sich allerdings die verwendeten Policen und die genaue Auswahl der Mechanismen im Detail unterscheiden, entstehen Unterschiede in der Performance und Echtzeitfähigkeit verschiedener Container-Runtimes, die wie nachfolgend dargestellt wird, von verschiedenen Publikationen beobachtet und mit Black-Box-Verfahren identifiziert wurden.

3 Stand der Technik

Die Virtualisierung von Anwendungen durch Container hat in den letzten Jahren stark an Bedeutung gewonnen, vor allem mit Blick auf Flexibilität, Portabilität und Ressourcenkonsolidierung. Zahlreiche Arbeiten befassen sich mit der Performance von Container-Technologien in Nicht-Echtzeit-Szenarien, etwa hinsichtlich der Auslastung der Central Processing Unit (CPU), Input/Output-Durchsatz (I/O-Durchsatz) oder Netzwerkkommunikation. Im Kontext der industriellen Automatisierung stellt sich aber die Frage, inwieweit containerbasierte Virtualisierung auch für Echtzeitanforderungen geeignet ist, wie sie etwa bei virtuellen speicherprogrammierbaren Steuerungen (englisch: Virtual Programmable Logic Controllers, vPLCs) auftreten.

Zunächst werden Arbeiten betrachtet, die container-basierte Ausführung mit nativer Ausführung vergleichen. Der Datendurchsatz bei Zugriffen auf das Dateisystem von einem bis vier Docker-Containern im Vergleich zur nativen Ausführung wird von *Dordevic et al.* [19] betrachtet. Dabei veränderte sich das Ergebnis bei einem Container verglichen mit nativer Ausführung kaum. Der Datendurchsatz verschlechterte sich allerdings stark bei höherer Containeranzahl. Von *Casalicchio et al.* [20] wird der durch Docker-Container verursachte Overhead hinsichtlich der Ausführungszeit CPU-intensiver Workloads sowie des I/O-Durchsatzes untersucht. Hierzu wird die Performance containerisierter Anwendungen mit nativer Ausführung verglichen. Zentrale Erkenntnis dieser Arbeit ist, dass die Performance-Messung von Containern komplex und die Auswahl geeigneter Werkzeuge nicht trivial ist. Dies liegt vor allem daran, dass unterschiedliche Messwerkzeuge auf verschiedenen Systemebenen ansetzen – etwa innerhalb des Containers oder auf Betriebssystemebene – und somit unterschiedliche Metriken liefern. Beispielsweise kann die vom Container angeforderte CPU-Quote von der tatsächlich genutzten CPU-Zeit abweichen. Die Arbeit analysiert jedoch nicht die Ursachen möglicher Performanceeinbußen durch Virtualisierungsoverhead, sondern stellt primär Werkzeuge und Metriken zur Benchmarking-Durchführung vor.

Ein weiterer Forschungsschwerpunkt liegt im Vergleich verschiedener Container-Engines, insbesondere Docker und Podman, und deren Einfluss auf die Systemperformance [21–24]. Von *Dordevic et al.* [21] wird die Performance bei Dateisystemzugriffen von Docker- und Podman-Containern analysiert und mit nativer Ausführung verglichen. Dabei zeigt sich, dass Podman geringfügig bessere Ergebnisse erzielt, während der Overhead gegenüber nativen Zugriffen in beiden Fällen insgesamt gering bleibt. Von *Voulgaris et al.* [22] wird der CPU-Bedarf von Machine-Learning-Anwendungen in Docker- und Podman-Containern untersucht, um Kostenunterschiede im Cloud-Betrieb zu quantifizieren. Bei geringer Containeranzahl ergeben sich keine signifikanten Unterschiede zwischen den beiden Engines. Bei horizontaler Skalierung zeigt Docker jedoch eine leicht effizientere CPU-Nutzung. Von *Gamess et al.* [23] werden Docker- und Podman-Container zusätzlich mit LXC-Containern verglichen. Dabei zeigt sich, dass sich Docker und Podman hinsichtlich der Ausführungszeit nicht unterscheiden, während LXC-Container eine deutlich höhere Ausführungszeit aufweisen.

Die genannten Arbeiten fokussieren sich auf die Performance-Analyse verschiedener Container-Engines und untersuchen unterschiedliche Aspekte wie Netzwerk-, Datei- und CPU-Overhead. Die Messungen erfolgen dabei überwiegend als Black-Box-Tests auf Container-Ebene. Eine detaillierte Analyse und Verifikation der zugrunde liegenden Ursachen für beobachtete Performance-Unterschiede erfolgten aber nicht, wie in **Tabelle 1** dargestellt.

Suo et al. [25] untersuchen den Kommunikations-Overhead verschiedener Container-Kommunikationsmechanismen sowohl innerhalb eines einzelnen physischen Knotens als auch zwischen mehreren Knoten. Auf einem einzelnen Knoten werden die Kommunikationsmodi None unter Verwendung des Loopback-Interfaces, Bridge, Container-Modus, bei denen sich mehrere Container denselben Netzwerk-Namespace teilen, und Host-Modus verglichen. Dabei zeigt sich, dass vor allem die Docker-Bridge mit einem hohen Kommunikations-Overhead verbunden ist.

Für Multi-Host-Setups werden zusätzlich Host-Modus, NAT, Overlay-Networking und Routing analysiert. Ergebnisse zeigen, dass Overlay-Networking einen höheren Overhead aufweist als Host-Modus, Routing und NAT. Bei der Auswahl des Kommunikationsmechanismus muss zudem eine Abwägung zwischen Performance, Sicherheit und Flexibilität erfolgen. So ist die Unterstützung bestimmter Netzwerkprotokolle beim Routing eingeschränkt, und NAT-basierte Ansätze ermöglichen nur begrenzt dynamische Container-Netzwerke.

Im Folgenden werden Arbeiten betrachtet, die sich mit den zugrunde liegenden Container-Runtimes befassen. *Kumar et al.* [26] vergleichen die Performance der Container-Runtimes runc und Kata-Containers. Hierzu wird containerd jeweils mit einer der beiden Runtimes konfiguriert und anhand verschiedener Benchmarks evaluiert. Untersucht werden unter anderem Bootzeiten, Ressourcennutzung, I/O-Performance sowie Speicher-, CPU- und Netzwerkverhalten. In allen betrachteten Kategorien zeigt runc eine bessere Performance als die Kata-Containers-Runtime. Von *Espe et al.* [27] wird die Performance von runc und runsc, der

Tabelle 1 Vergleich Stand der Technik im Bereich Benchmarking von Container-Runtimes.

Referenz	Vergleich von	Untersuchter Aspekt	Echtzeit	Black-Box-Tests
[20]	Docker	Ausführzeit-Overhead Docker-Container vs. nativ	Nein	Ja
[21]	Docker, Podman	Performance-Overhead bei Zugriff auf das Filesystem	Nein	Ja
[22]	Docker, Podman	CPU-Bedarf für einzelne und mehrere Container	Nein	Ja
[23]	Docker, Podman, LXC	Ausführzeit von Berechnungen	Nein	Ja
[25]	Docker	Kommunikationsoverhead	Nein	Nein
[26]	Runc, kata containers runtime	CPU-Bedarf, I/O-Performance, Speicherbedarf	Nein	Ja
[27, 28]	Runc, crun, runsc	Speicher- und CPU-Bedarf	Nein	Ja
[32]	Docker, Podman	Verpasste Deadlines, Jitter	Ja	Ja
[29]	Docker	Networking-Verzögerungen, Round Trip Time, CPU-Verzögerungen	Ja	Ja

Runtime von gVisor, analysiert. Beide Low-Level-Runtimes werden ebenfalls über containerd konfiguriert. Die Ergebnisse zeigen, dass runsc im Vergleich zu runc einen höheren CPU- und Speicher-Overhead verursacht. Zu ähnlichen Ergebnissen kommen auch *Bhaia et al.* [28], die zusätzlich die Runtime crun in ihre Untersuchung einbeziehen. Während runc und crun vergleichbare Performance aufweisen, zeigt runsc einen signifikant höheren Overhead.

Alle bisher betrachteten Arbeiten adressieren ausschließlich Nicht-Echtzeit-Szenarien. Die Frage, inwieweit Container-Technologien auch für vPLCs geeignet sind, wurde bisher nur punktuell untersucht. vPLCs kombinieren klassische PLC-Funktionalität mit einer containerbasierten Runtime, weshalb neben vorhersagbaren Ausführungszeiten insbesondere deterministische Kommunikationslatenzen entscheidend sind.

In *Sollfrank et al.* [29] wird die Anwendbarkeit von Container-Virtualisierung für weiche Echtzeit-Automatisierungsanwendungen am Beispiel von IEC 61499 vPLCs analysiert. Dabei werden auftretende Zeitverzögerungen und die Netzwerkkommunikation zwischen containerisierten Anwendungen bewertet. Messungen der Round-Trip-Time (RTT) von UDP (User Datagram Protocol)-basierter Client-Server-Kommunikation über ein Ethernet-Netzwerk zeigen signifikante Verzögerungen, da die Standard-Docker-Bridge verwendet wird. Als vielversprechender Ansatz zur Reduzierung deterministischer Latenzen wird Time-Sensitive Networking (TSN) vorgeschlagen. Ein Orchestrierungskonzept für Echtzeit-Container mit TSN wird von *Sollfrank et al.* [30] und *Oechsle et al.* [31] vorgestellt.

Von *Kebaier et al.* [32] wird die Auswirkung von Docker- und Podman-Containern auf die IEC 61131–3 vPLCs „CoDeSys" und „OpenPLC" untersucht. Die Ergebnisse zeigen, dass Podman-Container im Mittel einen geringeren Jitter aufweisen. Allerdings wurden Ausreißer bei der Ausführungszeit beziehungsweise beim Jitter nicht betrachtet, welche aber für Echtzeitanwendungen kritisch sein können. Eine detaillierte Analyse möglicher Ursachen für die beobachteten Performance-Unterschiede erfolgte nicht. Der Daemon-freie Ansatz von Podman wird aber als mögliche Ursache genannt.

Diese Befunde zeigen, dass die Performance-Unterschiede durch Container-Runtimes auf vPLCs bisher nur punktuell und überwiegend unter „weichen" Echtzeitbedingungen untersucht wurden. Eine systematische Analyse der Ursachen für Performance-Einbußen, insbesondere zu unterschiedlichen Container-Engines, Runtimes und Netzwerkmodi, sowie die Evaluierung deterministischer Latenzen für harte Echtzeitanforderungen bleibt bisher offen.

4 Benchmark zur Identifikation des Einflusses von Container-Isolationsmechanismen

Bild 2 zeigt die Vorgehensweise zur systematischen Identifikation von Störeinflüssen hinsichtlich des Echtzeitverhaltens durch Isolationsmechanismen von Container-Runtimes. Das Vorgehen erfolgt in zwei parallelen Strängen mit jeweils drei Schritten. Im ersten Strang werden Benchmark-Anwendungen identifiziert, die zur Beurteilung des Einflusses von Isolationsmechanismen auf spezifische Funktionalitäten von Echtzeitanwendungen dienen. Im zweiten Strang werden spezifische Isolationsmechanismen anhand derer Verwendung in Container-Runtimes identifiziert.

4.1 Die Benchmark-Anwendungen

Strang 1 (vergleiche Bild 2 oben) gliedert sich in die Schritte Funktionsanalyse, Systemcall-Identifikation und die Ableitung von Benchmark-Anwendungen. Im Zuge der Funktionsanalyse wird untersucht, welche Funktionalitäten Echtzeitanwendungen typischerweise vom Betriebssystem beanspruchen. Die Identifikation der Funktionen erfolgt anhand der Analyse einer typischen IEC 61131-Anwendung beziehungsweise vPLC. Diese besteht aus mehreren, zyklisch oder periodisch mittels Timer ausgeführten Tasks, die durch Mutexe oder lockfreie Algorithmen den Zugriff auf geteilte Ressourcen wie Shared-Memory synchronisieren.

Die Kommunikation über die Prozess- und Rechnergrenzen hinweg erfolgt typischerweise über Interprozess-(IPC)- und/oder Netzwerkkommunikation. So basieren viele industrielle Feldbusse auf Ethernet. Die typischen Funktionen umfassen somit Funktionen zum Datenaustausch über das Netzwerk, wie etwa mittels Echtzeit-Ethernet wie EtherCAT (Layer 2 Ethernet-Frames) oder UDP über das Internet Protocol (IP). Auch IPC über UDP inner-

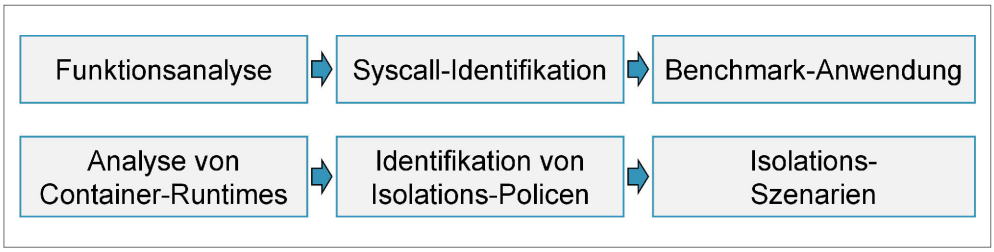

Bild 2 Vorgehensweise zur Konzeption des Benchmarks-Frameworks. *Grafik: ISW*

Tabelle 2 Identifizierte Syscalls.

Systemaufruf	Funktion	Anwendung in Echtzeitsystemen
nanosleep	Schlafenlegen eines Tasks für eine bestimmte Zeit	Periodische Echtzeittasks
send/recv	Senden und Empfangen von Daten über IPC und Netzwerk	Datenaustausch zwischen Speicherprogrammierbarer Steuerung und numerischer Steuerung. Kommunikation über Feldbus
sendmsg/recvmsg	Senden und Empfangen von Daten über IPC und Netzwerk	Datenaustausch zwischen Speicherprogrammierbarer Steuerung und numerischer Steuerung. Kommunikation über Feldbus
epoll/poll/select	Demultiplexing der Kommunikation	Event-basierte Reaktion auf mehrere Eingangskanäle zum Beispiel für Event-Loops.
mq_send/mq_recv	Message über Message-Queue senden/empfangen	Einfacher Datenaustausch zwischen Prozessen/Tasks
gettime	Aufzeichnung von Zeitstempel	Aufzeichnung von Zeitstempel

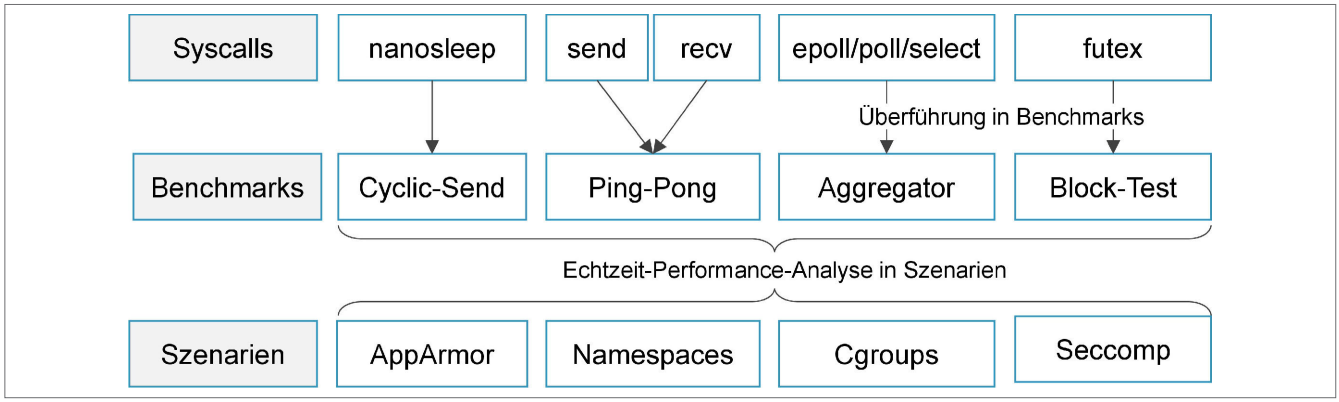

Bild 3 Vorgehensweise zur Identifikation von Latenzquellen durch Benchmark-Szenarien-Permutation. *Grafik: ISW*

halb eines Geräts, Message-Queues und Unix Domain Sockets (UDS) werden typischerweise genutzt. Timer gehören ebenfalls zu den Echtzeit-Funktionen.

Funktionen, die nicht zeitkritisch sind, aber für Echtzeit-Anwendungen relevant sind, werden nicht betrachtet, da deren Verzögerung durch Isolationsmechanismen für deren Performance irrelevant ist. Zu diesen Funktionen gehören das Setzen von Prioritäten, das Abschalten von Swapping, also der Auslagerung von Daten aus dem Arbeitsspeicher auf die Festplatte, sowie die Festlegung von CPU-Affinitäten für Echtzeit-Tasks.

Im zweiten Schritt werden die entsprechenden zugrundeliegenden Systemaufrufe (englisch: Syscalls) aus den in Schritt 1 identifizierten Funktionen abgeleitet. Für die periodische Ausführung von Tasks wird der Systemaufruf clock_nanosleep und zum Zeitnehmen der Aufruf clock_gettime verwendet. timer_create und timer_settime dienen der Realisierung von Timern. Der Systemcall futex ist die Basis für Implementierungen von Semaphoren Mutexe. Mittels der Systemcalls send, recv, sendmsg und recvmsg werden Daten zwischen Prozessen mittels IPC oder das lokale Loopback-Netzwerk und über das Netzwerk ausgetauscht. Durch select, poll und epoll erfolgt das Demultiplexing von eingehenden Datenquellen, also dem Horchen auf mehrere Nachrichtenquellen, wie etwa UDP-Datenströme. Mittels der mq_send und mq_recv werden Daten über Message-Queues zwischen Prozessen und/oder Tasks ausgetauscht. **Tabelle 2** zeigt die Liste an beschriebenen Systemaufrufen.

Für die Bewertung der Echtzeitfähigkeit werden im dritten Schritt komplementäre Benchmark-Anwendungen entwickelt, die zum einen die Systemaufrufe aus Tabelle 2 sowie die typischen Kommunikations- und Scheduling-Muster von Steuerungsanwendungen abbilden. Im Folgenden wird im Beitrag nur auf die Benchmarks für nanosleep und send/recv eingegangen (vergleiche Reihen 1 und 2 in **Bild 3**).

„Cyclic-Benchmark" (**Bild 4**): Diese Anwendung simuliert eine zyklische Steuerungsaufgabe mit fester Periodendauer. Ein Prozess führt in einer Schleife `clock_nanosleep()` mit absoluter Zeitangabe aus und misst die Differenz zwischen geplanter und tatsächlicher Aufwachzeit (Wakeup Latency). Zusätzlich wird ein UDP-Paket mit Zeitstempeln versendet, um die Netzwerkübertragungslatenz (Message Latency) zu erfassen. Das Nachrichten-

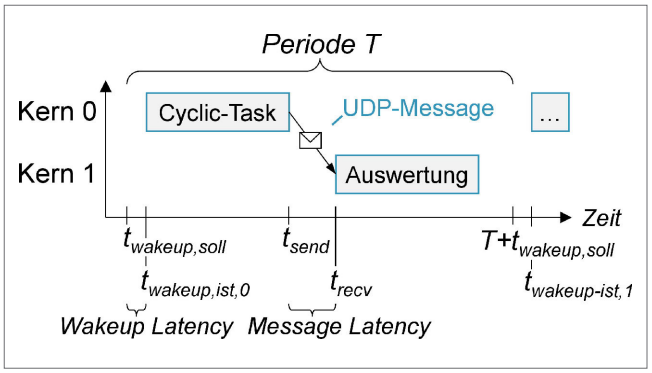

Bild 4 Cyclic-Benchmark. *Grafik: ISW*

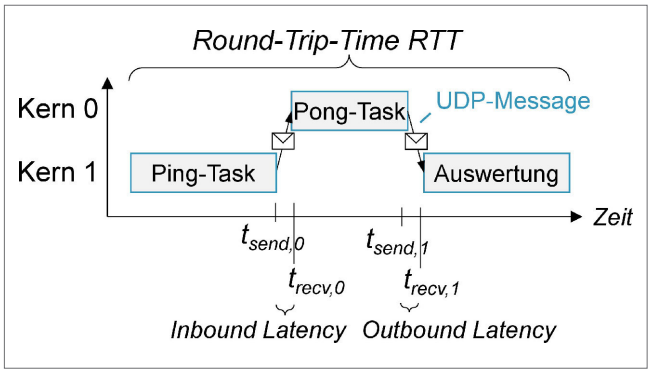

Bild 5 Ping-Pong-Benchmark. *Grafik: ISW*

format umfasst 24 Bytes mit vier 64-Bit-Zeitstempeln ($t_{wakeup,soll}$, $t_{wakeup,ist}$, t_{send} und t_{recv}) in Big-Endian-Kodierung.

„Ping-Pong-Benchmark" (**Bild 5**): Diese Anwendung misst die RTT einer Anfrage-Antwort-Kommunikation über UDP. Der Ping-Prozess sendet einen Zeitstempel, der Pong-Prozess antwortet mit dem Originalzeitstempel sowie Empfangs- und Sendezeitstempel. Aus den vier Zeitstempeln werden Outbound-Latenz (Anwendung zur Auswertung), Inbound-Latenz (Auswertung zur Anwendung) und die Gesamtlaufzeit berechnet.

„Unix Domain Socket (UDS)-Benchmark" (analog zu Bild 5): Analog zum Ping-Pong-Benchmark, aber über UDS (AF_UNIX, SOCK_DGRAM). Dieser Benchmark erfasst die IPC-Latenz ohne Beteiligung des Netzwerk-Stacks und dient als Vergleichswert für netzwerkbasierte Kommunikation.

Alle Benchmark-Anwendungen wurden in Rust implementiert und nutzen folgende Echtzeit-Konfigurationen:

- First In First Out (FIFO) für das Echtzeit-Scheduling mit Priorität 90,
- CPU-Affinität zur Vermeidung der Migration von Tasks zwischen CPU-Kernen,
- mlockall() zur Verhinderung von Page Faults und
- CLOCK_MONOTONIC mit clock_gettime() für die Zeitstempelerfassung.

4.2 Die Isolations-Szenarien

Um dedizierte Aussagen über den Einfluss spezifischer Isolationsmechanismen auf bestimmte Systemaufrufe zu identifizieren, werden in Strang 2 (siehe Bild 2 unten) Isolationsszenarien (siehe Reihe 3 in Bild 3) definiert. Dabei sind Black-Box-Szenarien und White-Box-Szenarien mögliche Ansätze zur Definition der Szenarien. Black-Box-Szenarien sind die Default-Policen der Container-Runtimes, also diejenigen, die ohne gesonderte Einstellungen angewendet werden. White-Box-Szenarien sind einzelne Isolationsmechanismen, wie etwa Memory CGroups, Network-Namespaces oder spezifische AppArmor-Profile.

Mittels Black-Box-Tests können zwar Latenzeinflüsse, die durch die Default-Policen unterschiedlicher Runtimes entstehen, aufgedeckt werden, die Frage nach der Ursache bleibt aber unbeantwortet. Mit den White-Box-Tests, die nachfolgend eingesetzt werden, kann genauer untersucht werden, welcher konkrete Mechanismus zu den erhöhten Latenzen führt. Nachfolgend werden die White-Box-Szenarien näher beschrieben. Für Black-Box-Tests sei auf die Literatur in Kapitel 3 verwiesen.

Im Gegensatz zu Black-Box-Ansätzen, bei denen eine vollständige Container-Umgebung als unteilbare Einheit vermessen wird, verfolgt der hier vorgestellte White-Box-Ansatz wie oben beschrieben das Ziel, einzelne Isolationsmechanismen unabhängig voneinander zu charakterisieren. Dazu werden die von Container-Runtimes verwendeten Isolationsmechanismen als Szenarien extrahiert und programmatisch, ohne Beteiligung einer Container-Runtime, aktiviert.

Als Referenz für die Szenarien dient ein Baseline-Szenario, bei dem keine Isolationsmechanismen aktiv sind. Alle weiteren Szenarien fügen einen oder eine definierte Kombination von Isolationsmechanismen hinzu, sodass deren inkrementeller Overhead ermittelt werden kann. Die Szenarien sind in **Tabelle 3** beschrieben.

4.3 Messverfahren

Für jedes Szenario werden 300 000 Iterationen mit einem Zyklus von einer Millisekunde durchgeführt, was einer Messdauer von etwa fünf Minuten entspricht. Zwischen den Szenarien liegt eine Beruhigungsphase von drei Sekunden. Vor jeder Messung werden Prozesse beendet und Cgroup-Strukturen zurückgesetzt, um Wechselwirkungen zu vermeiden. Die Rohdaten werden als CSV-Dateien mit Nanosekunden-Zeitstempeln gespeichert. Aus diesen werden Mittelwert, Median, Standardabweichung sowie die Perzentile p99, p99.9 und Maximum berechnet.

4.4 Exemplarische Messergebnisse

Die Testsystem-Konfiguration umfasst einen Linux-Kernel der Version 6.12 mit PREEMPT_RT. Die CPU ist ein Intel Xeon W-11555MRE mit sechs Kernen, wobei die Kerne 0–3 über isolcpus vom allgemeinen Scheduler isoliert sind. **Bild 6** zeigt einen exemplarischen Auszug aus den Messergebnissen (p99 und Maximum) für den Ping/Pong-Test mit UDP.

Jeder Balken repräsentiert eines der 19 White-Box-Szenarien. Eine deutliche Auswirkung zeigt das Netns-Extra-Hop-Szenario, bei dem ein zusätzlicher Netzwerk-Namespace mit Bridge eingefügt wird. Die Inbound-Übertragungslatenz steigt von 14,33 µs auf 16,71 µs. Um Vergleichbarkeit zu schaffen, wird in Szenario 18_cpu_stress das System mit CPU-Stress versehen. Dabei steigen die Latenzen auf 20,34 µs. Dies verdeutlicht, dass Network-Namespaces zwar zu deutlich erhöhter Latenz führen, allerdings im Vergleich zu System-Stress geringen Einfluss haben.

Tabelle 3 Die Szenarien im Überblick.

Nr.	Szenario	Beschreibung der Isolationsmechanismen	Relevanz für RT-Systeme
01	baseline	Ausschließlich Network-Namespaces werden verwendet; keine zusätzlichen Isolationsmechanismen	Dient als Referenzmessung für alle weiteren Szenarien
02	cgroup_cpuset	CPU-Affinität wird über den Cpuset-Controller der Cgroup-v2-Hierarchie konfiguriert	Messung des Cpuset-Accountings im Scheduler-Pfad
03	cgroup_mem	Der Memory-Controller begrenzt den verfügbaren Arbeitsspeicher auf 512 MB	Memory-Accounting verursacht Overhead bei jeder Allokation und jedem Page-Fault
04	cgroup_cpuset_mem	Kombination aus Cpuset- und Memory-Controller zur simultanen Ressourcenbegrenzung	Erfassung des kumulativen Overheads mehrerer Cgroup-Controller
05	cgroup_cpu_quota	Der CFS-Bandwidth-Controller limitiert die CPU-Zeit auf 95% der verfügbaren Kapazität	Throttling an Periodengrenzen kann periodische Latenzspitzen verursachen
06	seccomp_trivial	Ein minimaler BPF-Filter mit zwei Instruktionen wird auf alle Systemaufrufe angewendet	Quantifizierung des Basis-Overheads durch Seccomp-BPF-Interpreter-Eintritt
07	seccomp_heavy	Ein umfangreicher BPF-Filter mit 204 Instruktionen simuliert das Standard-Seccomp-Profil von Docker	Messung der Skalierung des Overheads mit der BPF-Programmkomplexität
08	apparmor	Das AppArmor-LSM wird im Enforce-Mode aktiviert mit einem All-Allow-Profil	LSM-Hook-Overhead wird bei jeder sicherheitsrelevanten Kernel-Operation gemessen
09	netns_extra_hop	Eine zusätzliche Linux-Bridge und ein weiterer Network-Namespace werden in den Netzwerkpfad eingefügt	Messung der zusätzlichen Latenz durch mehrfache Network-Stack-Traversierung
10	all_light	Kombination aus Cpuset-Controller, trivialem Seccomp-Filter und AppArmor	Repräsentiert eine leichtgewichtige, container-ähnliche Isolationskonfiguration
11	all_heavy	Kombination aus Cpuset-, Memory- und CPU-Quota-Controller sowie umfangreichem Seccomp-Filter und AppArmor	Repräsentiert den vollständigen Isolations-Stack typischer Container-Runtimes
12	pid_namespace	Ein separater PID-Namespace wird für die RT-Anwendung erstellt	Overhead durch PID-Translation bei Task-Struktur-Zugriffen
13	mount_namespace	Ein privater Mount-Namespace isoliert die Dateisystem-Sicht der Anwendung	Overhead bei VFS-Pfadauflösung durch Namespace-Indirektion
14	user_namespace	Ein User-Namespace mit Root-zu-Root-Mapping wird verwendet	Overhead durch UID/GID-Credential-Überprüfungen bei Berechtigungsprüfungen
15	cgroup_io_latency	Der io.latency-Controller der Cgroup-v2-Hierarchie wird aktiviert	Block-I/O-Accounting-Overhead bei Cgroup-Kontextwechseln
16	net_cls_cgroup	Der net_cls-Controller markiert Netzwerkpakete mit einer Cgroup-spezifischen Class-ID	Overhead durch Per-Packet-Lookup der Cgroup-Class-ID
17	numa_membind	Der Speicher wird explizit an einen entfernten NUMA-Knoten gebunden	Latenzeinbuße durch Cross-NUMA-Speicherzugriffe
18	cpu_stress	Die RT-Anwendung wird unter starker CPU-Last durch konkurrierende Prozesse ausgeführt	Messung der Shared-Resource-Contention (LLC, Speicherbandbreite)
19	uts_ipc_namespace	Separate UTS- und IPC-Namespaces isolieren Hostname und Inter-Prozess-Kommunikationsressourcen	Overhead durch nsproxy-Indirektion bei relevanten Systemaufrufen

Abkürzungen:
BPF: Berkeley Packet Filter – Bytecode-basiertes Filtersystem im Linux-Kernel
CFS: Completely Fair Scheduler – Standard-Prozess-Scheduler in Linux
Cgroup: Control Group – Kernel-Mechanismus zur Ressourcenbegrenzung und -überwachung
GID: Group Identifier – numerische Gruppen-Identifikation
IPC: Inter-Process Communication – Mechanismen zur Kommunikation zwischen Prozessen
LLC: Last Level Cache – letzte Cache-Ebene vor dem Hauptspeicher
LSM: Linux Security Module – Framework für Sicherheitserweiterungen im Kernel
NUMA: Non-Uniform Memory Access – Speicherarchitektur mit variablen Zugriffszeiten
RT: Real-Time – Echtzeitverarbeitung mit garantierten Latenzgrenzen
UID: User Identifier – numerische Benutzer-Identifikation
UTS: Unix Timesharing System – Namespace für Hostname und Domainname
VFS: Virtual File System – Abstraktionsschicht für Dateisysteme im Kernel

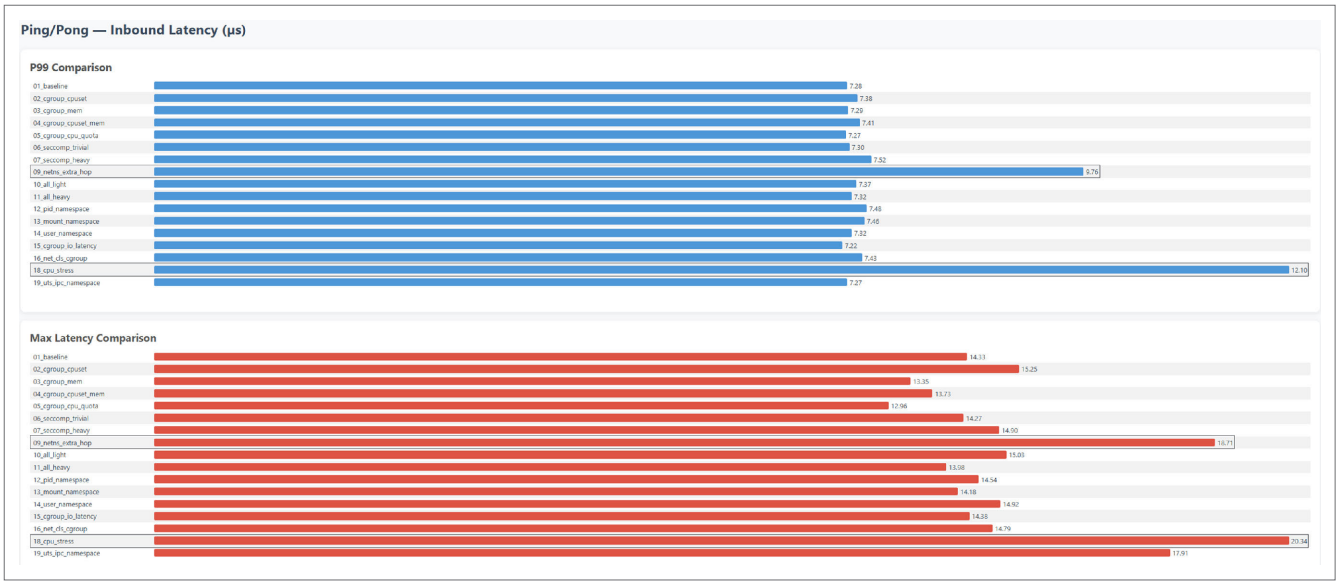

Bild 6 Auszug aus den Ergebnissen des Ping-Pong-Tests mit UDP (User Datagram Protocol) unter verschiedenen White-Box-Szenarien. Das Docker-ähnliche bridge-basierte Netzwerk-Setup führt zu erhöhten Latenzen der UDP-Pakete. *Grafik: ISW*

4.5 Fazit

Mittels Black-Box-Tests und Mikrobenchmarks können Unterschiede zwischen verschiedenen Runtimes hinsichtlich spezifischer Echtzeit-Funktionen identifiziert werden. Dabei bleibt aber die Frage nach der konkreten Ursache unbeantwortet. Vor diesem Hintergrund wurde in diesem Beitrag ein Ansatz für White-Box-Tests entwickelt.

Zusammengefasst lässt sich festhalten, dass mit dem White-Box-Ansatz identifiziert werden konnte, welches Isolationsfeature welchen Einfluss auf welchen Systemaufruf hat. Im vorliegenden Beispiel ist dies der send/recv-Systemaufruf, der durch zusätzliche Hops durch Netzwerk-Namespaces höhere Latenz erfährt. Der Source-Code der Benchmark-Suite ist öffentlich unter [33] abrufbar.

5 Zusammenfassung und Ausblick

Die vorliegende Arbeit stellt eine Methodik zum Benchmarking von Container-Runtimes für virtualisierte PLCs vor. Diese geht über bisherige Black-Box-Ansätze hinaus. Durch die isolierte Betrachtung einzelner Kernel-Mechanismen wie Namespaces, Cgroups, Seccomp-Filter und AppArmor können deren Auswirkungen auf die Echtzeitfähigkeit differenziert quantifiziert werden. Die entwickelten Benchmark-Anwendungen messen den Scheduling-Jitter periodischer Aufgaben sowie die Netzwerk-Übertragungslatenzen.

Die experimentellen Ergebnisse zeigen, dass nicht alle Isolationsmechanismen für Echtzeitanwendungen gleichermaßen problematisch sind. Während Namespace-Isolationen für PID, Mount und User keinen messbaren Overhead verursachen, führen zusätzliche Netzwerk-Hops zu einer signifikanten Latenzerhöhung von bis zu 26 %. Für die Praxis bedeutet dies, dass eine sorgfältige Analyse der tatsächlich benötigten Isolationsmechanismen wichtiger ist als ein pauschaler Verzicht auf Containerisierung.

Die vorgestellte Benchmarking-Methodik lässt sich in mehrere Richtungen erweitern. Eine naheliegende Erweiterung besteht darin, weitere White-Box-Tests aus den Standardrichtlinien von Container-Runtimes wie containerd oder CRI-O abzuleiten und verschiedene Low-Level-Runtimes wie runc und crun zu vergleichen. Außerdem sollte der Einfluss von Kernel-Versionen und PREEMPT_RT-Konfigurationen systematisch untersucht werden, um die Übertragbarkeit der Ergebnisse auf unterschiedliche Systemkonfigurationen zu validieren.

FÖRDERHINWEIS

Diese Arbeit wurde von der Deutschen Forschungsgemeinschaft (DFG) gefördert – Projektnummern 544536759 und 549319560.

Literatur

[1] Ellwein, C.; Neumann, R.; Verl, A.: Software-defined Manufacturing: Data Representation. Procedia CIRP 118 (2023), pp. 360–365, https://doi.org/10.1016/j.procir.2023.06.062

[2] Oechsle, S.; Walker, M.; Fischer, M. et al.: Real-Time Capable Architecture for Software-Defined Manufacturing. In: Kiefl, N.; Wulle, F.; Ackermann, C. et al. (eds): Advances in Automotive Production Technology – Towards Software-Defined Manufacturing and Resilient Supply Chains. SCAP 2022. ARENA2036, pp. 3–13, https://doi.org/10.1007/978-3-031-2793-1_1

[3] Neumann, R.; Walker, M.; Neubauer, M. et al.: Towards Uniform and Consistent Data Modelling of Resources in Distributed Industrial Control Systems. Procedia CIRP 138 (2026), pp. 304–309, https://doi.org/10.1016/j.procir.2026.01.053

[4] Neubauer, M.; Reiff, C.; Walker, M. et al.: Cloud-based evaluation platform for software-defined manufacturing. at – Automatisierungstechnik 71 (2023) 5, pp. 351–363, doi.org/10.1515/auto-2022–0137

[5] Giani, M.; Walker, M.; Neubauer, M. et al.: A sensitivity analysis approach to reduce economic risks in the design of edge-cloud automation systems in industry. Procedia CIRP 128 (2024), pp. 526–531, https://doi.org/10.1016/j.procir.2024.03.033

[6] Neumann, R.; Arnim, C. von; Neubauer, M. et al.: Requirements and Challenges in the Configuration of a Real-Time Node for OPC UA Publish-Subscribe Communication. 2023 29th International Conference on Mechatronics and Machine Vision in Practice (M2VIP), Queenstown,

New Zealand, 2023, pp. 1–6, doi.org/10.1109/M2VIP58386.2023.10413399
[7] Struhár, V.; Behnam, M.; Ashjaei, M. et al.: Real-Time Containers: A Survey. 2nd Workshop on Fog Computing and the IoT (Fog-IoT 2020). Open Access Series in Informatics 80 (2020), pp. 7:1–7:9, 7:1-7:9, https://doi.org/10.4230/OASIcs.Fog-IoT.2020.7
[8] Walker, M.; Fischer, M.; Lechler, A. et al.: Evaluation of Isolation and Communication Mechanisms for Real-Time Containers. 2023 IEEE 32nd International Symposium on Industrial Electronics (ISIE), Helsinki, Finland, 2023, pp. 1–8, doi.org/10.1109/ISIE51358.2023.10228012
[9] Arnim, C. von; Walker, M.; Lechler, A. et al.: Isolating Shared Resources for Time-Triggered Networking of Containerized Applications. 2023 29th International Conference on Mechatronics and Machine Vision in Practice (M2VIP), Queenstown, New Zealand, 2023, pp. 1–6
[10] The Linux Foundation: Open Container Initiative. Internet: opencontainers.org/. Zugriff am 15.04.2026
[11] Open Container Initiative: Image Format Specification v1.1.1. Internet: github.com/opencontainers/image-spec/releases/tag/v1.1.1. Zugriff am 15.04.2026
[12] Open Container Initiative: Distribution Specification v1.1.1. Internet: github.com/opencontainers/distribution-spec/releases/tag/v1.1.1. Zugriff am 15.04.2026
[13] Open Container Initiative: Runtime Specification v1.3.0. Internet: github.com/opencontainers/runtime-spec/releases/tag/v1.3.0. Zugriff am 15.04.2026
[14] Docker Inc: Get Started. Internet: docs.docker.com/get-started/. Zugriff am 15.04.2026
[15] LF Projects LLC: Getting Started with Podman. Internet: podman.io/docs. Zugriff am 15.04.2026
[16] containerd: containerd. Internet: github.com/containerd/containerd. Zugriff am 15.04.2026
[17] Containers: conmon. An OCI container runtime monitor. Internet: github.com/containers/conmon. Zugriff am 15.04.2026
[18] passt.top: pasta: Pack A Subtle Tap Abstraction. Internet: passt.top/passt/about/#pasta-pack-a-subtle-tap-abstraction. Zugriff am 15.04.2026
[19] Dordevic, B.; Gojak, D.; Davidovic, N. et al.: File system performance comparison of native operating system and Docker container-based virtualization. 8th IcETRAN 2021, Ethno village Stani (2021), pp. 495–500
[20] Casalicchio, E.; Perciballi, V.: Measuring Docker Performance. ICPE '17: ACM/SPEC International Conference on Performance Engineering, L'Aquila Italy, 2017, pp. 11–16
[21] Dordevic, B.; Timcenko, V.; Lazic, M. et al.: Performance comparison of Docker and Podman container-based virtualization. 2022 21st International Symposium INFOTEH-JAHORINA (INFOTEH), East Sarajevo, Bosnia and Herzegovina, 2022, pp. 1–6
[22] Voulgaris, K.; Kiourtis, A.; Karabetian, A. et al.: A Comparison of Container Systems for Machine Learning Scenarios: Docker and Podman, pp. 114–118, doi.org/10.1109/CompAuto55930.2022.00029
[23] Gamess, E.; Parajuli, M.: Containers Unleashed: A Raspberry Pi Showdown Between Docker, Podman, and LXC/LXD, pp. 34–43, doi.org/10.1109/SoutheastCon52093.2024.10500266
[24] Kanthed, S.: Docker vs. Podman: Architecture Differences and Their Impact on Modern Workflows. International Journal of Multidisciplinary Research and Growth Evaluation 4 (2023) 4, pp. 1139–1146, https://doi.org/10.54660/.IJMRGE.2023.4.4.1139–1146
[25] Kun Suo; Yong Zhao; Wei Chen et al.: An Analysis and Empirical Study of Container Networks. IEEE INFOCOM 2018 – IEEE Conference on Computer Communications, Honolulu, HI, USA (2018), pp. 189–197, https://doi.org/10.1109/INFOCOM.2018.8485865
[26] Kumar, R.; Thangaraju, B.: Performance Analysis Between RunC and Kata Container Runtime. Proceedings of IEEE CONECCT 2020, Bangalore, India, 2020, pp. 1–4, doi: 10.1109/CONECCT50063.2020.9198653
[27] Espe, L.; Jindal, A.; Podolskiy, V. et al.: Performance Evaluation of Container Runtimes. Proceedings ofthe 10th International Conference on Cloud Computing and Services Science (CLOSER 2020) (2020), pp. 273–281. doi.org/10.5220/0009340402730281
[28] Bhaia, N.; Hung, L.-H.; Cordingly, R. et al.: Understanding Container Isolation. Middleware '23: 24th International Middleware Conference, Bologna Italy, 2023, pp. 7–12
[29] Sollfrank, M.; Loch, F.; Denteneer, S. et al.: Evaluating Docker for Lightweight Virtualization of Distributed and Time-Sensitive Applications in Industrial Automation. IEEE Transactions on Industrial Informatics 17 (2021) 5, pp. 3566–3576, https://doi.org/10.1109/TII.2020.3022843
[30] Walker, M.; Imhoff, N.; Neumann, R. et al.: Methodology for the Combined Orchestration of Real-Time Containers and Time-Sensitive Network. Procedia CIRP 138 (2026), pp. 292–297, https://doi.org/10.1016/j.procir.2026.01.051
[31] Oechsle, S.; Frick, F.; Walker, M. et al.: Endpoint Architecture for Distributed Real-Time Applications Based on TSN, 2024, pp. 41–48
[32] Kebaier, N. B.; Geib, B.; Lyczkowski, E. et al.: Real-Time Performance Evaluation of Containerized Virtual PLCs: A Comparative Study of Docker and Podman for Industrial Automation. 2025 IEEE 30th International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD), Tempe, AZ, USA, 2025, pp. 1–6, https://doi.org/10.1109/CAMAD67323.2025.11229890
[33] Walker, M.; Neumann, R.; Richter, M. et al.: IsoBench – realtime-container-benchmark. Internet: github.com/wlkrm/realtime-container-benchmark, doi.org/10.5281/zenodo.19072487. Zugriff am 22.04.2026

Moritz Walker, M.Sc.
moritz.walker@isw.uni-stuttgart.de
Rebekka Neumann, M.Sc.
Matthias Richter, M.Sc.
Marc Fischer, M.Sc.
Dr.-Ing. Armin Lechler
Prof. Dr.-Ing. Alexander Verl
Institut für Steuerungstechnik der Werkzeugmaschinen und Fertigungseinrichtungen (ISW) der Universität Stuttgart
Seidenstr. 36, 70174 Stuttgart
www.isw.uni-stuttgart.de