# Echtzeitfähigkeit von Containerlösungen am Beispiel Docker

## Performance-Analyse der Echtzeitfähigkeiten von Linux-basierten Container-Lösungen auf ARM

Michael Schnelle, Mixed Mode

**Echtzeitbetriebssysteme**

Als Echtzeitsystem bezeichnet man im Allgemeinen ein System, das auf ein Ereignis innerhalb einer endlichen und vorhersagbaren Zeitspanne reagieren muss. Solche Systeme stellen folglich nicht nur logische, sondern auch zeitliche Anforderungen an ein Ergebnis. Echtzeit bedeutet dabei nicht unbedingt schnelles Handeln, sondern das Einhalten der gesetzten Zeitschranken und deterministisches Verhalten. Echtzeitsysteme werden dabei in zwei Kategorien eingeteilt: weiche und harte Echtzeitsysteme.

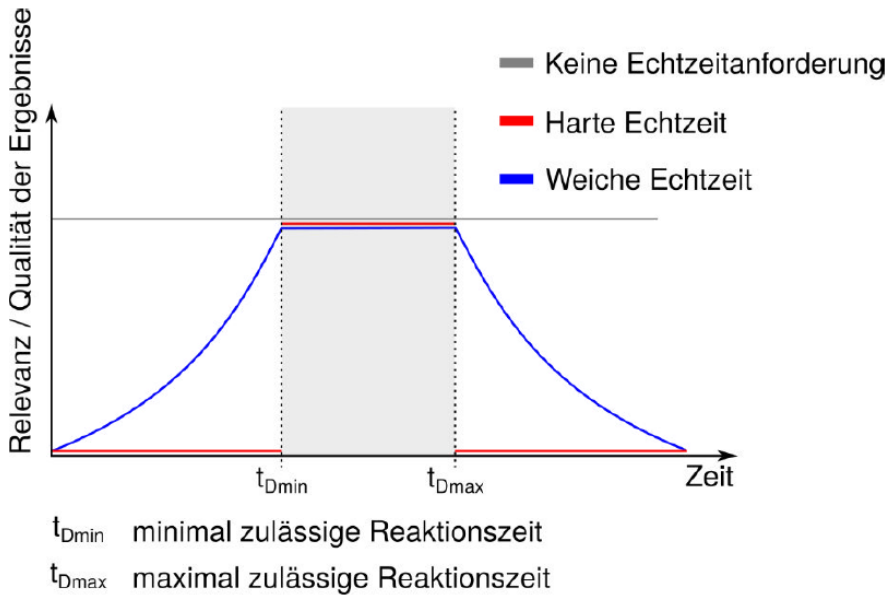


Abbildung 1 Nutzenfunktion bei Echtzeitsystemen

Bei einem weichen Echtzeitsystem dürfen die gesetzten Zeitschranken verletzt werden. Dennoch hat das gelieferte Ergebnis eine Bedeutung für das System. Hierbei hängt es vom Anwendungsfall ab, inwieweit der Nutzen des Ergebnisses unter dem verspäteten Auftreten leidet. Bei harter Echtzeit wiederum müssen die gesetzten Zeitschranken zu jedem Zeitpunkt eingehalten werden. Wird diese Regel verletzt, hat das Ergebnis keinen Wert für das System.

**Linux als Echtzeitbetriebssystem**

Linux wurde ursprünglich nicht für den Einsatz als Echtzeit- oder Embedded Betriebssystem entwickelt, doch machte die breite Hardwareunterstützung Linux sehr bald für beide Bereiche interessant. Die ersten Ansätze Linux echtzeitfähig zu machen, sahen dabei eine Dual-Kernel-Architektur vor. Bei diesen Lösungen wird

zusätzlich ein kleiner Echtzeitkernel auf dem System betrieben. Dieser Kernel führt Linux als einen niedrig priorisierten Prozess aus. Bekannte Lösungen solcher Ansätze sind RTAI und Xenomai.

Darüber hinaus ist Linux mittlerweile sogar in der Standardausführung bereits für weiche Echtzeitsysteme geeignet. Diese Tatsache geht auf die Entwicklung des Preempt-RT-Patches zurück, der den Linux-Kernel selbst hart echtzeitfähig macht. Aktuell arbeiten die Entwickler daran, den Patch vollständig in den Mainline-Kernel zu integrieren. Der Patch konzentriert sich vor allem darauf, den Linux-Kernel vollständig unterbrechbar zu gestalten, weil dies ein wichtiges Konzept für Echtzeitbetriebssysteme ist.

Der Standard-Kernel unterstützt aktuell drei Unterbrechungs-Modelle:
`No Forced Preemption (Server),` bei dem ein Task nur bei Rückkehr aus einem Systemaufruf oder bei einem Interrupt unterbrochen werden kann. `Voluntary Kernel Preemption (Desktop)`, bei dem noch zusätzliche Unterbrechungspunkte eingeführt wurden. Oder `Preemptible kernel (Low-Latency-Desktop),` bei dem der Kernel jederzeit unterbrochen werden kann, außer er befindet sich in einem kritischen Bereich.

Der Preempt-RT-Patch führt zwei weitere Unterbrechungspunkte ein: `Preemptible Kernel (Basic-RT)` und `Fully Preemptible (RT)`. Letzterer gestaltet den Linux-Kernel schließlich bis auf einige wenige kritische Abschnitte vollständig unterbrechbar. Hierzu werden Mechanismen wie `sleeping Spinlocks` und `rt_mutexes` eingesetzt. Außerdem werden Interrupt Routinen als `Threaded Interrupts` ausgeführt, die es ermöglichen, dass ein höher priorisierter Userspace-Prozess eine ISR unterbrechen kann.

**Virtualisierung**
Im Rahmen der Container-Virtualisierung wird fälschlicherweise oftmals von leichtgewichtigeren virtuellen Maschinen gesprochen, obgleich die grundlegenden Techniken gänzlich verschieden sind:
Bei der Vollvirtualisierung kommt eine Softwarekomponente zum Einsatz, um die logische Schicht zwischen dem Gastsystem und der Hardware zu bilden. Diese Softwarekomponente wird `Hypervisor` oder `Virtual Machine Monitor` genannt. In der Praxis unterscheidet man `Type-1- und Type-2 Hypervisor`. (Mandl, 2014)

Ein `Type-1-Hypervisor` arbeitet direkt auf der Hardware des Systems und agiert als minimales Betriebssystem, das für das Erstellen und Verwalten der einzelnen virtuellen Maschinen zuständig ist. Der `Hypervisor` benötigt somit Treiber für den Hardwarezugriff. Der `Type-2-Hypervisor` wird innerhalb eines Betriebssystems als Anwendungsprogramm ausgeführt. Der Zugriff auf die Hardware erfolgt dabei über die Treiber des Hostsystems.

Bei der Container-Virtualisierung (Betriebssystemvirtualisierung) werden Prozesse allein auf Betriebssystemebene isoliert. Dazu erstellt der Kernel des Host-Systems virtuelle Prozessräume, in denen die Prozesse ablaufen können. Die isolierten Prozesse bekommen den Eindruck, dass ihnen eine komplette Rechnerumgebung zur

Verfügung steht, obwohl sie in Wirklichkeit isolierte User-Space-Instanzen eines darunterliegenden Betriebssystems sind. Der Vorteil besteht darin, dass durch die gemeinsame Nutzung eines Kernels kein großer Performance Einbruch zu erwarten ist. Gleichzeitig ergibt sich dadurch aber der Nachteil, dass alle isolierten Instanzen auf dem Kernel des Hostsystems angewiesen sind und deshalb keine unterschiedlichen Betriebssysteme betrieben werden können. (C.Arnold, 2012)

Der Linux Kernel selbst stellt mehrere Mechanismen zur Betriebssystemvirtualisierung dar. Durch die Verwendung von Prozessbäumen können Prozesse gegenseitig abgeschottet werden. Die Container-Virtualisierung macht sich dieses Prinzip zu nutzen, um einzelne Container anzulegen.
Container-Lösungen benutzen diesen Mechanismus z.B. für die folgenden System-Ressourcen:

- `Mount-Namespace`: Dem Namespace kann ein eigenes Wurzelverzeichnis bzw. private Einbindungen zugewiesen werden.
- `PID-Namespace`: Bei der Erstellung eines Containers legt die Container-Engine einen neuen PID-Namespace an und führt die zu isolierenden Prozesse in diesen Namespace über. Der erste im Container ausgeführte Prozess erhält hierbei die PID 1. Jeder weitere Prozess im Container ist nun diesem untergeordnet. Der Container selbst ist auf seinen PID-Namespace eingeschränkt, das Hostsystem sieht weiterhin alle Prozesse.

Mit `control groups (cgroups)` bietet der Linux Kernel einen Mechanismus, Prozessen oder Containern nur einen Bruchteil bestimmter Ressourcen zuzuteilen. Prozesse werden hierbei verschiedenen Kontrollgruppen zugeteilt, deren Zugriff auf Systemressourcen limitiert werden kann.

**Container-Lösungen**
Das Konzept von isolierten Prozessinstanzen unter Linux gibt es schon seit 2001. Praktikabel wurde es erst die die Einführung der vorgestellten Kernel Features `cgroups` und `namespaces`. Die erste Implementierung die diese Mechanismen verwendete war LXC (Linux Container). Eine weitere Implementierung stellt Docker dar.
Docker verwendet eine Server-Client-Architektur, bestehen aus dem `Docker Client` und dem `Docker Daemon`. Der Benutzer interagiert bei der Erstellung von Containern mit dem `Docker Client`, welcher Befehle an den `Docker Daemon` weiterreicht. Dieser übernimmt anschließend alle weiteren Aufgaben, wie die Verwaltung der Systemabbilder (`Docker Images`) und das Anlegen von `Docker Containern`. Client und Daemon müssen nicht zwangsläufig auf demselben System laufen. Docker setzte anfangs auf externe Containerlösungen wie `libvirt` oder `LXC`, um mit dem Linux-Kernel zu interagieren. Ab Version 0.9 wurde diese Funktion durch eine eigene Implementierung ersetzt. Das virtualisierte Betriebssystem wird in einem Image zusammengefasst, welches entweder in einer privaten oder öffentlichen Registry gespeichert und verwaltet werden kann. Ein laufendes Image nennt man einen Container.

**Testkonzept**
Als Hardware-Grundlage wurde `BeagleBone Black` mit einem `Sitara AM335x Conrtex-A` der Firma Texas Instruments verwendet. Auf dieser Basis

soll die Echtzeitfähigkeit der Container-Lösungen Docker und LXC überprüft werden. Hierzu wurden verschiedene Testszenarien Ausgewählt:

`Cyclictest` ist ein Programm, das häufig im Zusammenhang mit dem `Preempt-RT Patch` genannt wird. Es wurde im Rahmen der Entwicklung des Patches geschrieben und kann dazu verwendet werden, die Echtzeitfähigkeit eines Systems einzuschätzen. Hierzu erstellt das Programm einen Thread, der die aktuelle Systemzeit misst und sich anschließend für ein definiertes Zeitintervall schlafen legt. Sobald der so gestellte `Timer` abgelaufen ist und der `Thread` wieder vom `Scheduler` ausgewählt wird, nimmt der `Thread` wieder die Systemzeit und kann aus den gemessenen Zeiten einen Wert für die `Scheduling-Latenz` berechnen.

```
clock_gettime(&now)
next = now + interval
while(!shutdown){
        clock_nanosleep(&next)
        clock_gettime(&now)
        latency = calcdiff(now, next)
        next += interval
}
```

Abbildung 2 Cyclictest

Das Programm wird auch von OSADL eingesetzt, um die Echtzeitfähigkeit des `Preempt-RT Patches` zu testen. Das Ergebnis wird in Form eines Histogramms ausgegeben. Zusätzliche Informationen sind die minimale, durchschnittliche und auch die maximale Latenz. Unten stehend sind exemplarisch die Histogramme für zwei `BeagleBone Blacks` aus der „OSADL Testfarm“ dargestellt. Dabei ist das linke mit dem `Preempt-RT Patch` ausgestattet und das rechte mit einem Standard-Kernel.

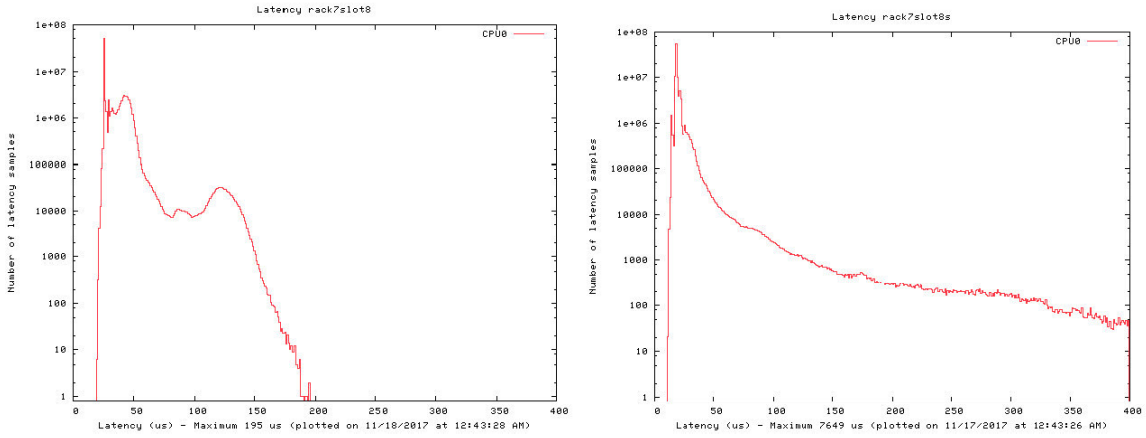


Abbildung 3 Vergleich der Latenzen - Preempt RT Kernel und Standard Kernel

Bei einer zweiten Echtzeitanwendung, der `Interrupt-Stoppuhr`, soll die Messung für die Echtzeitfähigkeit nicht wie bei `cyclictest` direkt auf dem Testsystem, sondern über eine externe Messeinrichtung stattfinden. Die Grundidee dabei ist das Nachstellen einer oft anzutreffenden Situation, in der ein Echtzeitsystem in einem definierten Zeitrahmen auf ein externes Signal reagieren muss. Die Verarbeitung und Reaktion auf das externe Signal soll dabei in einer User-Space-Anwendung erfolgen, die man einmal nativ und einmal innerhalb der Container-Lösung betreibt.

Konkreter definiert soll mit einem Mikrocontroller ein externes Triggersignal erzeugt werden, dass im Linux-Kernel eine Interrupt-Service-Routine auslöst, die einen GPIO-Pin auf high setzt und anschließend die Echtzeitanwendung im User-Space aufweckt. Diese Anwendung soll dann den entsprechenden GPIO-Pin wieder auf low setzen. Der geplante Signalverlauf von der Anregung bis zur Antwort ist unten stehend skizziert.

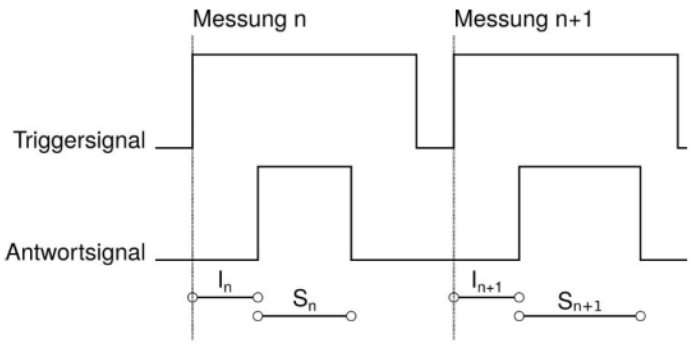


Abbildung 4 Interrupt Stoppuhr

Die Zeitspanne I steht dabei jeweils für die Interrupt-Latenz also der Zeit zwischen dem Auslösen des Triggersignals und dem Beginn der entsprechenden ISR auf dem Testsystem. Die Zeitspanne S wiederum repräsentiert die Zeit die zwischen der ISR und der Antwort der User-Space-Anwendung vergeht.

Der Mikrocontroller soll zu Beginn der Messung einen Zähler starten und sowohl die steigende, als auch die fallende Flanke des Antwortsignals detektieren. Dabei ist jeweils der aktuelle Wert des Zählers zu speichern. So kann gleichzeitig die Interrupt-Latenz als auch die Interrupt-zu-Prozess-Latenz ermittelt werden. Anschließend sollen die gemessenen Werte noch über die serielle Schnittstelle an einen Rechner für die spätere Verarbeitung weitergesendet und eine neue Messung gestartet werden.

Das Prinzip dieser Messung ist ein häufig anzutreffendes Testszenario bei der Bewertung von Echtzeitsystemen und wird z.B. von OSADL in Form der `Latency Box` durchgeführt.

Für die Auswertung und den Vergleich der Echtzeitfähigkeit der Container-Lösungen wird auch hier vor allem die Interrupt-zu-Prozess-Zeit interessant sein, da es hier zu Verzögerungen durch den Betrieb der Anwendung innerhalb eines Containers kommen könnte.

An dieser Stelle sei noch angemerkt, dass dieses Testverfahren auf den ersten Blick redundant wirken mag, da der `cyclictest` einen ähnlichen Wert ermittelt. Allerdings werden externe Interrupts und Timer-Interrupts im Kernel anders abgearbeitet. Dies führt dazu, dass andere Stellen des Kernel-Codes beteiligt sind und das Zeitverhalten daher unterschiedlich sein kann.

**Durchführung**

Bei der Testdurchführung wurden zwei verschiedene Lastszenarien betrachtet. Im ersten Fall wurden die Tests ohne zusätzliche Last auf dem `BeagleBone Black` durchgeführt. Anschließend wurde das System während der Durchführung der Tests zusätzlich belastet. Beide Programme wurden unter verschiedenen Lastszenarien mit 10 Millionen Messungen durchgeführt.

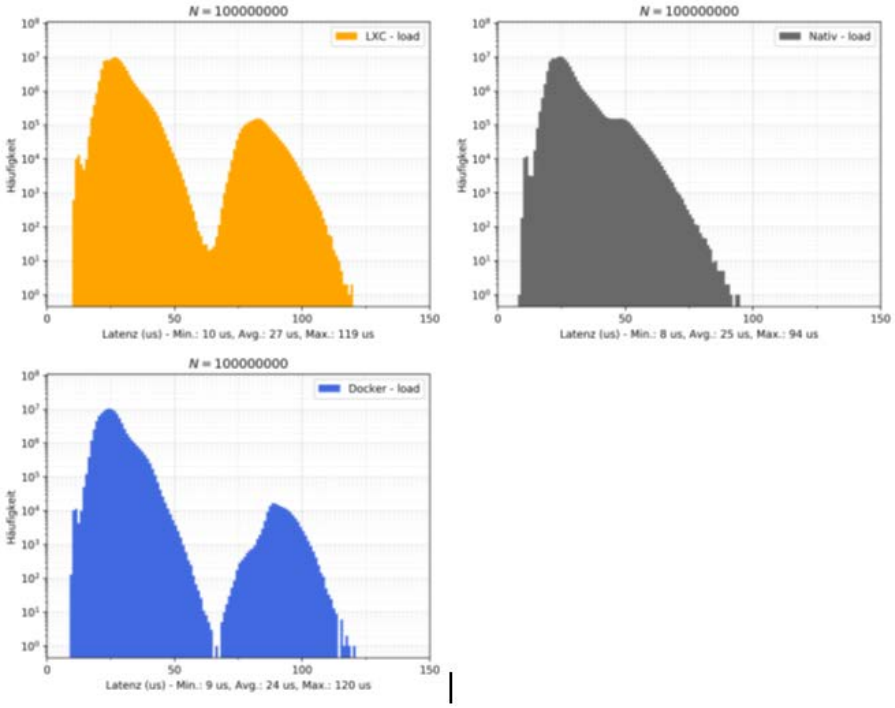


Abbildung 5 Cyclictest - Vergleich

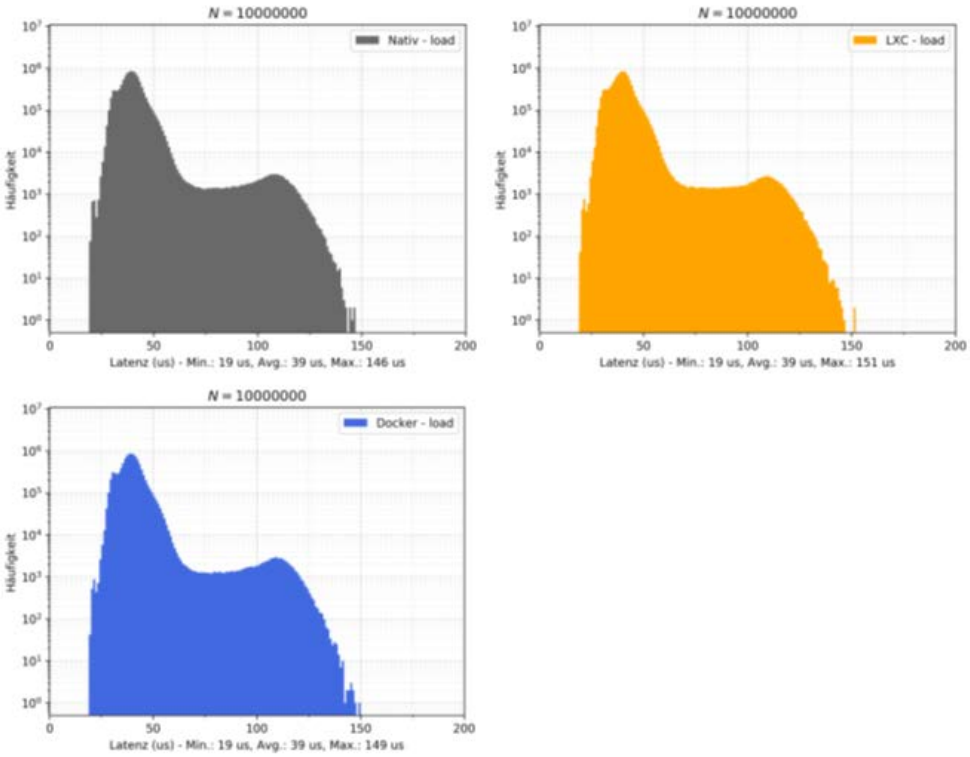


Abbildung 6 Interrupt Stoppuhr Vergleich

**Fazit**

„Hypervisor-gestützte VM-Virtualisierung und Containerbasierte Anwendungsvirtualisierung liegen vom Konzept her wie Tag und Nacht auseinander“[Quote], so die allgemeine Einschätzung. Dem kann vollstens zugestimmt werden. Ein Container ist nicht einfach eine leichtgewichtige virtuelle Maschine. Eher kapselt er Systemprozesse, deren Sichtweite durch das Betriebssystem eingegrenzt ist.

Weiter lässt sich festhalten, dass der grundsätzliche Betrieb einer Echtzeitanwendung in einem Linux-Container möglich ist. Dem Container sind dabei allerdings auch gewisse Rechte zuzuteilen, die vor allem im Hinblick auf die Gewährleistung eines sicheren Betriebs bedenklich sind. Auch wenn es aufgrund des beschränkten Umfangs der Messungen nicht gelungen ist, gänzlich unanfechtbare Ergebnisse festzuhalten, kann vermutet werden, dass auch in zukünftigen Tests ähnliche Ergebnisse erzielt werden.

**Literaturverzeichnis**

C.Arnold, M. J. (2012). *KVM Best Practices.* dpunkt.
Mandl, P. (2014). *Grundkurs Betreibssysteme.* Wiesbaden: Springer Verlag.

**Autor**

Der Referent Michael Schnelle besitzt einen Abschluss als Master in Software-Engineering. Mittlerweile verfügt er über mehrere Jahre Entwicklungserfahrung mit den Schwerpunkten Applikationsentwicklung, und „Security“ in unterschiedlichen Schattierungen. Unter anderem führte er dabei Risikoanalysen durch, entwickelte Lösungen zur Abwehr schadhafter Anfragen auf einer Socialmedia-Plattform, für ein hochsicheres Signalisierungssystems für Notrufeinheiten, sowie zur Generierung einer sicheren Linux/ Debian-Distribution. In seinem aktuellen Projekt wirkt er bei der Entwicklung von Fahrgastinformations-Systemen für Verkehrsunternehmen mit, mit den Schwerpunkten verteilte Systeme, Microservices und Testautomatisierung.

**Kontakt**

Email: michael.schnelle@mixed-mode.de