Studienarbeit
Virtio Paravirtualisierung für Palacios VMMs in L4Re

Johannes Steinmetz

April 2012

Mit Hilfe der vom V3VEE Projekt (v3vee.org) entwickelten Bibliothek Palacios ist es einfach, Vollvirtualisierungsmonitorprogramme auf verschiedene Betriebssysteme zu portieren, die moderne Prozessorfunktionen zur Virtualisierung ausnutzen. Diese Studienarbeit erweitert die experimentelle Portierung der Bibliothek auf das L4Re Betriebssystem um die Möglichkeit der Verwendung von paravirtualisierenden Gerätemodellen, welche die in Linux vorhandene Implementierung des Virtio-Frameworks benutzen.

Betreuung durch:

Dipl.-Inf. Alexander Warg

Institut für Systemarchitektur, Lehrstuhl Betriebssysteme

Technische Universität Dresden

Inhaltsverzeichnis

1 Einleitung **4**

2 Grundlagen und Stand der Technik **4**
2.1 Virtualisierung 4
2.1.1 Virtualisierung in der x86 Architektur 6
2.1.2 Hardware- und virtuelle Geräte 7
2.1.3 Geräte in Para- und Vollvirtualisierung 7
2.2 Palacios VMM Bibliothek 9
2.2.1 Geräte in Palacios 9
2.3 Fiasco und L4Re 9
2.3.1 L4 Familie 9
2.4 Mikrokern Fiasco 10
2.4.1 L4 Runtime Environment 10
2.5 Palacios in L4Re 11
2.5.1 Virtualisierung in L4Re 11
2.6 Virtio 12
2.7 Virtqueue Transportmechanismus 13
2.8 Darstellung als PCI Gerät 13
2.8.1 Virtio Geräte aus Sicht des Gastes 13
2.8.2 Virtio Interrupts 16

3 Entwurf **16**
3.1 Gerätemodelle in Palacios 16
3.2 Paravirtualisierung in Palacios 17
3.3 Gast-Linux zu Virtio Schnittstelle 18
3.4 Palacios zu L4Re Schnittstelle 19

4 Implementierung **23**
4.1 Mögliche Implementierungskandidaten 23
4.1.1 virtio_ballon 23
4.1.2 virtio_net 24
4.1.3 virtio_block 24
4.1.4 virtio_console 24
4.2 virtio_console in Linux im Detail 25
4.2.1 Control Messages 26
4.3 Schnittstellenimplementierung 27
4.3.1 Anbindung des VMM an Vcon 27
4.3.2 Palacios Schnittstelle 27
4.4 Interrupt-Thread 28
4.5 PCI Gerätemodell 29
4.6 Gast Linux 29

5 Leistungsbewertung und Ausblicke **29**
5.1 Schlüsselstellen der Implementierung 30

1 Einleitung

Virtualisierung hat einen festen Platz im Werkzeugkasten von Serverbetreibern und Systementwicklern. Im Umfeld von Mikrokern basierten Betriebssystemen ist Virtualisierung ein geeignetes Mittel, um einzelne Anwendungen einzusetzen, die für die Verwendung in üblichen Betriebssystemen wie GNU/Linux oder Microsoft Windows programmiert sind. Denn auch in kleinen Betriebssystemen mit Spezialausrichtung auf Sicherheit oder Echtzeitbetrieb sind komplexe Anwendungen gefragt wie beispielsweise ein Emailprogramm oder ein Webbrowser. Diese Programme schöpfen die zur Verfügung stehenden Softwarebibliotheken der Standard-Desktopbetriebssysteme so sehr aus, dass eine native Portierung auf die jeweiligen Mikrokern basierten Systeme mitunter zu aufwändig ist. Ein für diese Anwendungen in einer virtuellen Maschine bereitgestelltes Standardbetriebssystem kann hingegen eine effiziente und flexible Möglichkeit sein, eine solche Anwendung in Mikrokernsystemen zu realisieren. Für das Mikrokern basierte Betriebssystem der Betriebssystemeforschungsgruppe an der TU Dresden, L4Re mit dem Fiasco Mikrokern, existiert eine neue Virtualisierungslösung, die es ermöglicht, auf der x86 Architektur unmodifizierte Betriebssysteme bereitzustellen. Den virtuellen Maschinen werden über Hardwaregerätemodelle Kommunikationsmöglichkeiten eröffnet, die auf verschiedene Art und Weise an die Möglichkeiten des umgebenden Hostbetriebssystems anknüpfen können.
Diese Arbeit untersucht mit Virtio eine spezielle Schnittstelle zur Verbindung von L4Re Host- und Linux-Gastsystem. Zur Orientierung wird dabei zunächst ein Überblick über die technischen Grundlagen gegeben. Dazu ist die Vorstellung des Konzeptes der Virtualisierung vorgesehen und einige Konzepte der x86 Architektur in Hinblick auf die Einbindung von Hardwaregeräten werden grob umrissen. Es werden drei Systeme zur x86 Virtualisierung unter L4Re soweit vorgestellt, um Ihre Möglichkeiten im Bezug auf virtuelle Gerätemodelle darzustellen. Besonders liegt der Schwerpunkt auf dem im praktischen Teil verwendeten neuen Virtualisierungsmonitor, welcher mit dem Palacios Framework erstellt wurde. Letztendlich wird die Idee und Funktionsweise von paravirtuellen Geräten mit dem in Linux Virtualisierungen integrierten Virtio Framework vorgestellt und an der zur Arbeit gehörigen praktischen Implementierung des `virtio_console` Gerätemodells in Palacios auf L4Re beispielhaft demonstriert.

2 Grundlagen und Stand der Technik

2.1 Virtualisierung

In der Informatik werden unter Virtualisierung verschiedene Konzepte verstanden. Hier in dieser Arbeit ist mit Virtualisierung immer die Benutzung von logischen, also virtuellen Versionen einer Hardwareplattform anstelle der direkten Benutzung physischer Hardware verstanden. Dabei werden die Ressourcen des Hostsystems abstrahiert und zumindest in Teilen den logischen Versionen, den Gästen, zur Verfügung gestellt. Es entstehen virtuelle Maschinen, bei denen es sich um isolierte, effizient arbeitende Nachbildungen der physischen Hostmaschine handelt.

Virtuelle Maschinen, welche die Ausführungsumgebungen vieler Programmiersprachen darstellen, wie zum Beispiel die virtuellen Maschinen von Lua [6] oder Java [13], sind eine andere Art der Virtualisierung. Auch bei diesen enstehen isolierte Ausführungsumgebungen für Software, jedoch ohne eine Nachbildung von physisch existierenden Maschinen zu sein. Diese abstrakten virtuellen Maschinen dienen vorrangig der Portierbarkeit von Anwendungen und der saubereren Implementierung ihrer Ausführungsumgebungen über Betriebssystemgrenzen hinweg.
Ein dritter Virtualisierungsbegriff findet sich bei der Bereitstellung isolierter Schichten eines Betriebssystemkernes zum Zwecke isolierter Ressourcenverwaltung. Verschiedenen Anwendungen werden eigene Ausgaben des gesammten Nutzerlandes, sogenannte *Container*, gestellt unter der Verwendung eines gemeinsamen Betriebssystemkernes. Diese Form der Virtualisierung, die auch Containerisierung genannt wird, findet sich beispielsweise bei den Systemen *Linux Containers (lxc)*, *OpenVZ* und *FreeBSD Jail*[15].
Der Virtualisierungsbegriff im Sinne dieser Arbeit lässt sich nach Popek und Goldberg [17] von Emulation und Simulation dadurch abgrenzen, dass nicht eine zum Hostsystem völlig verschiedene Plattform vorgespiegelt wird, sondern die Mehrheit der Instruktionen der Gastsysteme direkt vom Prozessor des Hostsystems abgearbeitet wird und somit eine zum Hostsystem gleichartige Plattform dargestellt wird. Abgesehen von Differenzen durch unterschiedlich zur Verfügung stehende Ressourcen soll sich der im Gast ablaufende Programmcode auch auf der physischen Hostmaschine abarbeiten lassen.
Nach der Definition von Popek und Goldberg [17] hat ein Programm der direkten physischen Maschine, der virtuelle Maschinen Monitor (VMM), immer die volle Kontrolle über alle Systemressourcen in dem Sinne, dass virtuelle Maschinen ausschließlich auf die ihnen vom VMM zugeteilten Ressourcen Zugriff haben und der VMM diese Ressourcen den virtuellen Maschinen auch wieder entziehen kann. Dadurch ist Isolation der Gastmaschinen sowohl untereinander, als auch gegen die physische Hostmaschine gegeben und es offenbaren sich verschiedene Anwendungsmöglichkeiten:
Moderne Computersysteme erreichen eine Leistungsfähigkeit, mit welcher sie mehrere komplexe Probleme parallel bearbeiten können. Trotzdem ist für viele Aufgaben ein eigenes, isoliertes System erwünscht, um zu verhindern, dass Abhängigkeiten oder gegenseitige Beeinflussungen die Verfügbarkeit und Integrität einzelner Anwendungen gefährden. Virtuelle Systeme bieten die Möglichkeit die Aufgaben isoliert voneinander bis einschließlich der Betriebssystemebene zu bearbeiten, ohne auf möglichst hohe Auslastung der Hardwareplattform zu verzichten.
Außerdem bieten virtuelle Systeme eine gute Möglichkeit, Systemprogramme zu entwickeln und zu testen. Es lassen sich durch Software verschiedene Ressourcenkonfigurationen schnell und effizient darstellen und gegebenenfalls der Zustand einer Systemsoftware aus dem Hostsystem heraus beurteilen und verändern. Zu diesem Zweck ist die unter Linux laufende Virtualisierungslösung KVM beispielsweise in der Lage, die Schnittstelle zur entfernten Fehlersuche des GNU Debuggers GDB zu bedienen [18][1]. Die Kombination aus GDB und KVM kann dann leicht benutzt werden, um beispielsweise bei der Gerätetreiberentwicklung unter

[1] In [18] findet sich die Verwendung von GDB und Qemu. Verwendet man jedoch zusätzlich den Schalter `--enable-kvm` wird mit Hilfe des KVM Kernmoduls aus der Emulation Qemu ein schnellerer VMM.

Linux Fehler aufzuspüren, die im Kerncode auftreten und nicht durch ein lokales Inspektionsprogramm gefunden werden können.
Goldberg [5] gibt eine Einteilung der VMM an nach der Umgebung, in welcher sie ausgeführt werden: Mit Typ I wird dabei ein VMM bezeichnet, welcher die Kontrolle über die Ressourcen dadurch behält, dass er ohne Abstraktion auf der reinen physischen Hardware ausgeführt wird, wohingegen ein VMM vom Typ II unter der Kontrolle eines Betriebssystem ausgeführt wird.
Weiterhin können virtuelle Maschinen unterteilt werden im Grad der Genauigkeit der Nachbildung der physischen Maschine in der virtuellen Maschine: Bei einer vollständigen Virtualisierung oder Vollvirtualisierung verhält sich die virtuelle Maschine zur physischen Hostplattform so ähnlich, dass ein Gastbetriebssystem ohne Weiteres auch auf der physischen Maschine direkt laufen könnte. Die notwendigen Ressourcen werden vom VMM verwaltet und den Gästen bei Bedarf entwerder zugeteilt, oder für den Gast transparent nachgebildet. In jedem Fall läuft der Gastsystemcode in der virtuellen Umgebung, ohne durch spezielle Funktionen auf die virtuelle Maschine adaptiert zu sein.
Im Gegensatz zur Vollvirtualisierung ist bei der Paravirtualisierung die Nachbildung nicht so weitgehend: Geräteressourcen werden häufig nur über VMM spezifische Schnittstellen angeboten und die in der virtuellen Maschine laufende Software muss auf diese Umstände speziell angepasst sein. Durch diese Anpassungen werden Vereinfachungen des VMMs möglich oder leistungskritische Nachteile der Ressourcennachbildung vermieden. In Kapitel 2.1.3 gehe ich auf diese Unterscheidung genauer ein.

2.1.1 Virtualisierung in der x86 Architektur

Popek und Goldberg geben in [17] drei Kriterien an, die einen VMM charakterisieren: Erstens bietet der VMM Programmen eine Ausführungsumgebung, die weitgehend identisch zur physischen Maschine ist. Programme sollen also abgesehen von Unterschieden, die von unterschiedlicher Ressourcenverfügbarkeit herrühren, sich auf der virtuellen Maschine wie auf der physischen gleich verhalten. Zweitens sollen die Programme in der virtuellen Umgebung nur wenig langsamer ablaufen, als eine direkte Ausführung auf der physischen Maschine. Und drittens soll der VMM dabei jederzeit die volle Kontrolle über die der virtuellen Maschine zugebilligten Ressourcen haben.
Um dieses dritte Kriterium zu erfüllen, müssen die Instruktionen, die eine Veränderung der Ressourcen einer virtuellen Maschine ermöglichen, *sensitive Instruktionen*, in einem Modus ausgeführt werden, in welchem der VMM diese verhindern und gegebenenfalls verändert nachbilden kann. Da im x86 Befehlssatz jedoch unprivilegierte, aber trotzdem sensitive Befehle enthalten sind, ist, nach den von Robin und Irvine [19] ausführlich diskutierten Kriterien, die x86 Architektur nicht klassisch virtualisierbar. Robin und Irvine haben die Betrachtung im Jahr 2000 an Intel Pentium Chips bis zur Generation Pentium III durchgeführt. Heutzutage jedoch können VMMs von den speziellen Hardwarefunktionen Gebrauch machen, welche die Prozessorhersteller speziell zur möglichst effizienten Virtualisierung in die Prozessoren eingebaut haben. So bieten beispielsweise die Hersteller AMD [1] und Intel [8] in ihren Prozessoren spezielle Erweiterungen des x86 Befehlssatzes an, die dem VMM ermöglichen, den Kontrollfluss so lange an einen Gast abzugeben, wie im Gastsystem keine sensitive

Anweisungen ausgeführt werden sollen. Der Kontrollfluss setzt darauf hinter der Stelle im VMM fort, die den Eintritt in die virtuelle Maschine bewirkt hat. Es stehen dem VMM Informationen über die Ursache des Austritts zur Verfügung, um dem VMM die Behandlung der Instruktion zu ermöglichen. Dabei kann es sich etwa um eine sensitive Instruktion oder ein Interruptrequest in der Hostumgebung handeln. Durch diesen zusätzlichen Schutzmodus sind alle sensitiven Instruktionen vom VMM kontrollierbar und die Architektur damit effizient virtualisierbar. Die Erweiterungen der beiden Hersteller sind trotz ähnlicher Konzeption nicht vollständig kompatibel zueinander.

2.1.2 Hardware- und virtuelle Geräte

Stehen mit den Befehlssatzerweiterungen unter x86 nun einfache Möglichkeiten bereit, Prozessoren dieser Architektur virtuell darzustellen, benötigt eine vollständige virtuelle Maschine außerdem auch andere virtuelle Hardware. Da es sich bei der x86 Plattform meistens um Rechner handelt, die aus der PC/AT Plattform hervorgegangen sind, verlassen sich Betriebssysteme für diese Architektur zumeist auf das Vorhandensein bestimmter Geräte. Für den Softwareteil, welcher die Hardwarechips steuert, die sogenannten Gerätetreiber, stellt sich die Hardware zumeist entweder als einfache Speicherzellen mit speziellen Bedeutungen dar, oder diese Register sind in einem von normalen Speicherzugriffen separierten Adressraum an sogenannte IO-Ports eingeblendet. IO Zugriffe, und zwar sowohl in den Port- als auch den normalen Speicheradressraum, sind also prinzipiell wie normale Speicherzugriffe, die aber über Seiteneffekte die Hardwaregeräte steuern. Weitere Aspekte dazu finden sich beispielsweise in [3, Kapitel 9].
Nennenswerte, besonders wichtige Geräte sind bei PCs zum Beispiel der programmierbare Interruptleitungsmultiplexer Chip Intel 8259 (Programmable Interrupt Controller: PIC) oder der *Programmable Intervall Timer* (PIT) Intel 8254. Auf modernen PC Hauptplatinen sind diese nicht mehr als eigene Chips vorhanden, ihre Funktionalität wird aber immer noch vom Chipsatz bereitgestellt. Ein aktuelles Datenblatt eines von der Firma Intel hergestellten Chipsatzes zur Herstellung moderner Laptops spricht etwa von den Kompatibilitätsmodulen (Compatibility Modules, DMA Controller, Timer/Counters, Interrupt Controller), die Teil des sogenannten „Platform Controller Hubs“sind [7, S. 47]. Dadurch können Betriebssysteme zu älteren x86 Systemen kompatibel sein und haben eine minimale gemeinsame Basis, auf welcher die Erkennung modernerer Komponenten aufbauen kann.
Geräte können demnach in einerseits obligatorische Geräte unterteilt werden, die für Start und Funktion des Betriebssystems unentbehrlich sind, und andererseits optionale Geräte, welche lediglich die Anbindungsmöglichkeiten und Funktionen der Maschine erweitern. Die Unterscheidung, ob ein Gerät nun unbedingt notwendig ist, hängt dabei von dem in der virtuellen Maschine zu verwendenden Betriebssystem ab.

2.1.3 Geräte in Para- und Vollvirtualisierung

Vorangehend in diesem Kapitel 2.1 habe ich die Einteilung vom VMMs in Para- und Vollvirtualisierungs-VMMs erwähnt. In beiden Fällen wird durch ein VMM eine virtuelle Version des Prozessors der Maschine bereitgestellt, aber die Bereitstellung der restlichen Geräte zur

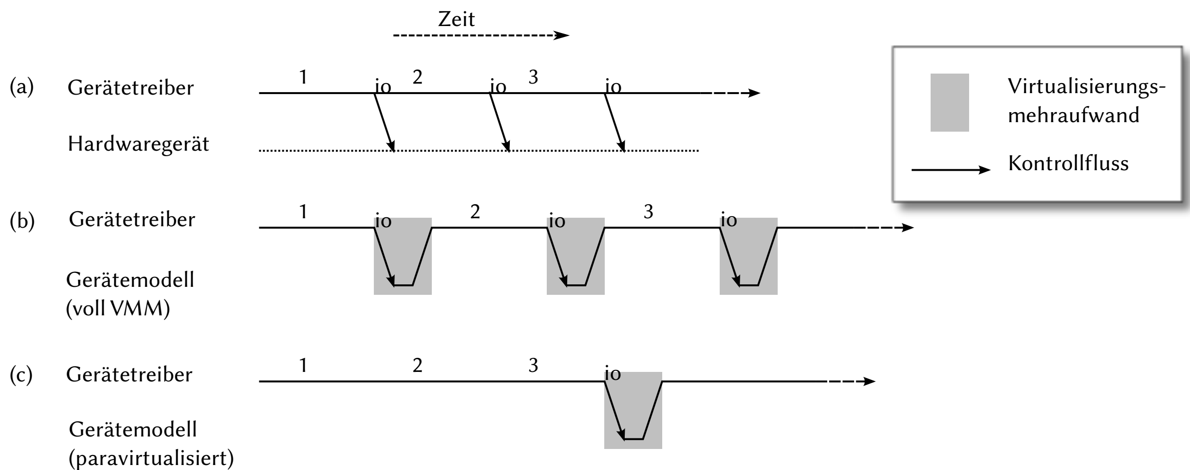

Abbildung 1: Hier ist der Mehraufwand der Virtualisierung anhand eines Gerätetreibers gezeigt. Für 3 Verarbeitungseinheiten entsteht bei (a) einem physischen Gerät kein relevanter Aufwand. In (b) einer Vollvirtualisierung erzeugt die Nachbildung eines gleichartigen Gerätes Mehraufwand, der von einem paravirtualisiertem Gerät (c) verringert wird.

Datenkommunikation mit dem virtuellen Prozessor unterscheidet sich bei Voll- und Paravirtualisierung. Laufen bei einer Vollvirtualisierung unveränderte Standardbetriebssysteme indem ihnen alle benötigten Geräte vom VMM angeboten werden, sind hingegen bei Paravirtualisierung spezielle Gerätemodelle im VMM vorhanden, die das Gastbetriebssystem nutzen kann, weil es auf den Betrieb in der virtuelle Maschine ausgelegt ist und die verwendeten Spezialschnittstellen ansprechen kann.
Im Extremfall ist das in der virtuellen x86 Maschine laufende Betriebssystem soweit angepasst, dass überhaupt keine üblichen Geräte der physischen PCs dargestellt werden und ausschließlich auf die Spezialschnittstellen im VMM zugegriffen wird. Dadurch können Flaschenhälse bei der x86 Virtualisierung durch beispielsweise häufige kleine Port-IO Operationen vermieden werden, die in der Virtualisierung, verglichen mit denen in physischen Maschinen, hohe Ausführungszeitkosten verursachen.

Abbildung 1 schildert den Leistungsvorteil gegenüber der Vollvirtualisierung. Dieser Vorteil wird jedoch im ungünstigen Fall mit dem Verlust der Ausführbarkeit von Betriebssystemen und anderen Programmen bezahlt, an denen die zur Umgehung des Flaschenhalses notwendigen Anpassungen nicht durchgeführt werden können.
Da beispielsweise zu Microsoft Windows, einem verbreiteten Betriebssystem für die x86 PC-Plattform, keine Umgebung mitgeliefert wird, die Veränderungen und Neukompilation des Betriebssystems ermöglicht, aber durch geeignete Schnittstellen Gerätetreiber hinzu geladen werden können, bietet es sich an, einen Teil der Vorteile der Paravirtualisierung in eigentlich für Vollvirtualisierung ausgelegte VMMs zu übernehmen. Dazu implementiert der VMM ein Gerätemodell, welches mit einem zugehörigen Gerätetreiber für Standardschnittstellen des Gastbetriebssystems an dieses angeschlossen wird. Dieser Mittelweg erlaubt, unmodifizierte Betriebssysteme zu starten unter der Verwendung von korrekten Gerätemodellen für zu-

mindest alle zum Starten des Betriebssystems nötigen Geräte. Leistungskritische Geräte, die während der Anwendungsabarbeitung häufig benutzt werden, ersetzt man durch möglichst optimale Paravirtualisierungsgeräte, welche virtuell an Standardschnittstellen der Hardware dargestellt werden.

2.2 Palacios VMM Bibliothek

Palacios ist eine dem V3VEE-Projekt entstammende Virtualisierungslösung, die das Ziel verfolgt, eine Bibliothek zu schaffen, die sich einfach als Typ I VMM in vorhandene Betriebssysteme integrieren lässt. [10] Der resultierende VMM nutzt die Virtualisierungserweiterungen moderner x86 Prozessoren von AMD (SVM) und Intel (VT-x). Palacios ist darauf ausgelegt, auch in kleine Betriebssystemkerne geladen werden zu können. So liegen dem Quellcode Adaptionscode für *Linux*, den *Kitten lightweight Kernel* der Sandia National Labs und den *GeekOS Kernel* der University of Maryland bei. Die letzteren beiden sind Betriebssystemkerne für traditionell angelegte, monolithische Betriebssysteme. An das umgebende Betriebssystem werden durch die Schnittstelle von Palacios nur wenige wohldefinierte Anforderungen gestellt. Um in einem Betriebssystem nun einen VMM bereitzustellen, muss lediglich eine kleine Zahl von Funktionen über eine Funktionspointersammlung bereitgestellt werden (sogenannte Callback-Funktionen oder kurz Callbacks) und mit diesen die Palacios Bibliothek initialisiert werden.

2.2.1 Geräte in Palacios

Palacios enthält einige Gerätemodelle für virtuelle Geräte. Da mit Palacios hergestellte VMMs unmodifizierte Standardbetriebssysteme ausführen können sollen, sind darunter Modelle für die wichtigsten Grundgeräte vorhanden. Doch nicht nur ein absolutes Minimum an Funktionen, sondern auch einige zum Betrieb virtueller Maschinen nützliche Zusatzgeräte sind für den praktischen Betrieb von Standardbetriebssystemen vorhanden. So gibt es beispielsweise ein Gerätemodell, welches von den standardisierten ATAPI Gerätetreibern für CDROM-Laufwerke in Betriebssystemen bedient wird, sowie verschiedene Modelle üblicher Grafik- und Netzwerkschnittstellenkarten. Damit die von Palacios VMMs verwaltete Maschinen mit mehr als einer CPU in einem virtuellen, symmetrischen Multiprozessorsystem arbeiten können, ist neben dem PIC der übliche, verbesserte Interruptmultiplexer APIC vorhanden. Im SMP-Betrieb mit mehr als einer CPU wird für jede ein eigener Thread im Hostbetriebssysten gestartet, welcher den virtuellen Kontrollfluss dieser CPU behandelt.

2.3 Fiasco und L4Re

2.3.1 L4 Familie

Der in dieser Arbeit verwendete VMM läuft in einem Mikrokern basierten Betriebssystem. Um auf die Besonderheiten hier näher eingehen zu können, sei zunächst der Mikrokern Fiasco vorgestellt. Nachdem in den späten 1980er Jahren in der Betriebssystemeforschung die Idee aufkam, die Betriebssystemkerne zu minimalisieren und Teile des Betriebssystems in den unprivilegierten Ausführungsmodus des Prozessors auszulagern, erwartete man von diesen

entscheidende Verbesserungen in der Softwaretechnik der Betriebssysteme. Die verkleinerten Kerne sollten nur Mechanismen bereitstellen, um möglichst flexible modulare Systeme mit ihnen bauen zu können [12]. Demonstrierte die erste Generation an Mikrokernen die generelle Machbarkeit solcher Systeme, zeigten sie doch in der Praxis hauptsächlich zu schlechte Ausführungsgeschwindigkeitswerte, um praktisch den Anforderungen an Betriebssysteme mit Multimedia- und Echtzeitanwendungen gerecht zu werden. 1995 schuf J. Liedtke mit L4 am *GMD Forschungszentrum Informationstechnik* einen Vertreter einer zweiten Generation von Mikrokernen, der die Geschwindigkeitsprobleme in der Kommunikation zwischen Prozessen (IPC) behob. Es folgten einige weitere Implementierungen des von L4 begründeten Mikrokerninterfaces, die die L4 Mikrokernfamilie bilden.

2.4 Mikrokern Fiasco

Fiasco, ein Mikrokern, der an an der Technischen Universität Dresden entwickelt wurde, gehört zu den weiterentwickelten Vertretern der L4 Familie.
In der aktuellen Version Fiasco.OC werden alle angebotenen Kerndienste als abstrakte Kernobjekte dargestellt. Der Kern verwaltet für alle Tasks jeweils eine Tabelle, in der Referenzen auf die Kernobjekte abgelegt sind, die die Task benutzen kann. Diese Referenzen heißen Capabilities. [4] Die Tasks selbst stellen eine Form von Fiasco Kern Objekten dar. Sie sind die Fiasco Abstraktion eines Adressraumes mit zugehöriger Kernobjektetabelle. Weiterhin für die Arbeit von Bedeutung sind noch Objekte vom Typ Thread, ein vom Fiasco Scheduler behandelter Ausführungsfaden einer Task, IPC Gate, ein Kommunikationskanal zwischen Task, und IRQ, welches Objekte zur Signalisierung von hard- oder auch softwaregenerierten Ereignissen sind.
Tasks können die Capabilities, die sie besitzen, an andere Tasks weitergeben. Über diesen Mechanismus ist ein sehr flexibles Systemdesign möglich.

2.4.1 L4 Runtime Environment

Ein Mikrokern basiertes Betriebssystem implementiert einen Großteil seiner angebotenen Funktionalität in Servern genannten Programmen. Die Funktionen, die klassische Betriebssysteme zumeist im Kern implementieren, wie zum Beispiel Dateisysteme oder Gerätetreiber, werden von diesen einzelnen Servern angeboten und sind deshalb elementarer Bestandteil des Betriebssystems, auch wenn sie wie die Anwendungsprogramme im unprivilegierten Betriebsmodus des Prozessors verarbeitet werden.
Um mit Fiasco ein System zu bauen, ist mit dem L4 Runtime Environment (L4Re) eine solche Sammlung an Servern vorhanden. Außerdem beinhaltet das L4Re einige Bibliotheken, mit deren Hilfe solche Server, aber auch andere Anwendungen in diesem Betriebssystem hergestellt werden können. Die angebotenen Funktionen beinhalten beispielsweie eine uClibc Standardbibliothek, welche auf den Mikrokernabstraktionen Teile der üblichen Funktionen zur C und C++ Anwendungsentwicklung unterstützt. Dazu gehört auch eine pthreads Abstraktion, um neue Threads in einer Task zu starten und zu synchronisieren.

2.5 Palacios in L4Re

Als Typ-I VMM zur Integration in den Kern von an Linux angelehnten Betriebssystemen, ist zunächst die Verwendung von Palacios in einem Mikrokern basierten System eine Besonderheit. Denn im Kern sind nur die nötigsten Funktionen erwünscht, die notwendig sind, um Mechanismen zu implementieren, welche das Betriebssystem braucht, um die Hardware verwalten und abstrahieren zu können. Als Typ-I VMM ist Palacios aber nicht nur mit Mechanismen ausgestattet, sondern die gesamte VMM Funktionalität, inklusive aufwändiger Gerätemodelle, wird in der Bibliothek angeboten, die normalerweise im privilegierten Ring 0 ausgeführt wird.
Für die Verwendung in L4Re ist also der VMM notwendigerweise ein Programm im unprivilegierten Modus (Ring 3 bei x86), und damit auch nicht in der Lage, privilegierte Instruktionen auszuführen. Die angebotene Callbackschnittstelle, welche die VMM-Bibliothek an das Hostsystem koppelt, ließ sich jedoch erweitern, um letztendlich alle privilegierten Instruktionen aus Palacios zu entfernen. Die notwendigen Ressourcenzugriffe werden indirekt über die bereitgestellten Hookroutinen ausgeführt. Diese lassen sich auch in L4Re darstellen; dabei behält das Mikrokern basierte System die Kontrolle über die Hardwareressourcen. Somit ist der resultierende VMM im engeren Sinne nicht mehr dem Typ I, sondern eher Typ II zuzuordnen.

2.5.1 Virtualisierung in L4Re

Im L4Re System existieren neben dem Port von Palacios bereits andere Möglichkeiten, virtuelle Maschinen zu betreiben. Zum einen gibt es mit Karma [11] einen im Rahmen einer Diplomarbeit entstanden VMM, der mit Hilfe der Virtualisierungsbefehlssatzerweiterung von AMD ermöglicht, ein speziell auf die Gastrolle hin modifiziertes Linux in der virtuellen Maschine zu starten. Der Anschluss der virtuellen Umgebung funktioniert nicht über die Vorspiegelung virtueller Versionen von üblichen Geräten, sondern verlangt dem Betriebssystem die Benutzung einer speziellen Schnittstelle, der sogenannten Hypercalls, ab. Karma stellt somit einen reinen Paravirtualisirungs VMM bereit. Dadurch kann die gewünschte Funktionalität, wie zum Beispiel eine Netzwerkanbindung der virtuellen Maschine, mit kleinem Aufwand vom VMM über die Möglichkeiten des Hostsystems realisiert werden. Doch eine Verwendung von nichtangepassten Betriebssystemen ist in diesem VMM dadurch nicht möglich.
Dem Karma VMM ging die Entwicklung von KVM-L4 [16] voraus. Bei KVM-L4 handelt es sich um eine Vollvirtualisierung, die auf dem in Linux bekannten VMM KVM [9] basiert, welches in einer Instanz von L^4Linux läuft. L^4Linux ist dabei eine Portierung des Linuxkernes in das L4 Nutzerland. Die über diesen Umweg angebotene Vollvirtualisierung zeigt gute Leistungswerte auch im Vergleich mit auf einem nativen Linux laufenden KVM.
Mit KVM-L4 ist demnach schon eine Vollvirtualisierung für L4Re vorhanden. Karma bietet als Vorteil einen geringen Ressourcenaufwand im VMM und die Fähigkeit, dem angepassten Linux eine virtuelle Multiprozessormaschine darzustellen. Eine Portierung des Palacios Framework VMMs auf L4Re kann also durchaus über geringen Ressourcenverbrauch motiviert werden. Denn wie in Abbildung 2 angedeutet wird, schlägt sich die Verwendung

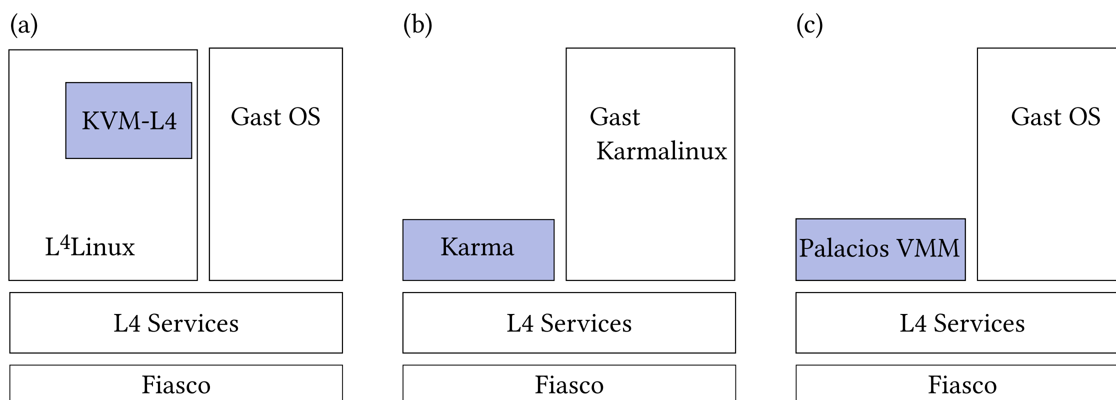

Abbildung 2: Grober Vergleich der vorgestellten VMMs. Die Architekturen von (a) Karma, (b) KVM-L4 und (c) Palacios in der L4Re Umgebung nebeneinander. Der VMM-Teil des Systems ist jeweils hervorgehoben.

des Linuxkernes auf die Gesammtkomplexität des verwendeten Codes nieder. Doch darf die Leichtgewichtigkeit auch nicht immer mit den Einschränkungen der Virtualisierung verbunden sein, welche Karma fordert. Der Bedarf an einer leichtgewichtigen Vollvirtualisierug, die zudem prinzipiell sich auch zur Virtualisierung von SMP Mehrkernsystemen eignet, ist damit eine hinreichende Motivation, mit Palacios einen dritten VMM neben den genannten zu schaffen.

2.6 Virtio

Virtio ist ein Framework zur vereinheitlichten Entwicklung paravirtualisierter Gerätetreiber in Linux. Zum Zeitpunkt der Entstehung von Virtio gab es laut Virtio Entwickler Rusty Russell im Linuxkern mindestens acht verschiedene Systeme für Virtualisierung: Xen, KVM, VMware's VMI, IBM's System p, IBM's System z, User Mode Linux, lguest und IBM's legacy iSeries. Russells Anspruch mit der Entwicklung von Virtio ist es, mit einer dünnen Schicht wichtige Gerätetreiber, etwa für virtuelle Block-, Netzwerk- und Konsolengeräte, zu abstrahieren, um sie zentral allen Visualisierungssystemen zur Verfügung zu stellen, damit unnötige Coderedundanz zu Gunsten der Qualität entfällt[20]. Die Grundlage der betreffenden Gerätemodelle sei immer ein Datentransport zwischen Hostsystem und dem Gerätetreiber im Gastbetriebssystem. Für die effiziente Implementierung eines solchen Datentransports sei ein FIFO Ringpuffer-Transportmechanismus geeignet, mit dessen Hilfe auch größere Datenmengen zwischen den Maschinen übertragen werden können, ohne überflüssige Übertritte zwischen Gast und Host zu erzeugen. Weiterhin sei außerdem nach Russell für die Bereitstellung der Paravirtualisierungstreiber ein einfach erweiterbarer Mechanismus zur gegenseitigen Bekannmachung der unterstützten Funktion in Host- und Gastteil notwendig.
Russells Lösung Virtio gliedert sich somit in zwei Teile: Einen Mechanismus zum Datentransport und einen, um diesen zu konfigurieren.

2.7 Virtqueue Transportmechanismus

Der Transportmechanismus, dessen Hauptziel der flexible und effiziente Datenaustausch zwischen Treiber im Gast und Gerätemodell im Host ist, wird durch eine Virtqueue genannte Datenstruktur abstrahiert. Über den Konfigurationsmechanismus müssen dazu die Parameter konfiguriert werden, die angeben, welche Virtqueues für ein Gerätemodell anzulegen sind, und welche Eigenschaften diese aufweisen.
Die wichtigste Funktionalität der Virtqueues ist es, mit Hilfe der Methode `virtqueue_add_buf` Puffer aufzunehmen, die dann asynchron oder nach Aufforderung durch `virtqueue_kick` im Host verarbeitet werden können. Mit `virtqueue_get_buf` ist es dem Gasttreiber möglich, fertig bearbeite Puffer wieder aus der Virtqueue zurück zu bekommen und die Ergebnisse der Verarbeitung auf der Hostseite zu verwenden. Da nun der für die Verarbeitung genutzte Puffer im Gast hinzugefügt ist, wird ist sichergestellt, dass die virtuelle Maschine ausschließlich in Speicher schreiben kann, der ihr vom VMM zugeteilt wurde.

2.8 Darstellung als PCI Gerät

Der Virtqueue-Transportmechanismus ist in Linux als Ringpuffersystem in `virtio_ring.c` zunächst abstrakt implementiert. Es bestünde die Möglichkeit, den Ringpuffer anders, angepasst auf die Gegebenheiten des verwendeten Virtualisierungsmonitors, konkret zu implementieren, etwa über eine Hypercallschnittstelle, wie sie Karma bereitstellt. Wie in Abschnitt 2.1.3 schon angedeutet, bietet es sich aber im Falle einer Vollvirtualisierung wie Palacios an, eine Betriebssystemschnittstelle zu verwenden, welche vom VMM ohnehin angeboten werden muss. Bei physischen x86 Maschinen hat die *Peripheral Component Interconnect*-Bus (PCI) Struktur eine große Verbreitung. Dass nun virtuelle Virtiogeräte zumindest in der Linux-Implementierung als PCI Geräte dargestellt werden, bietet sich aus mehreren Gründen an: Zunächst verfügen alle für die x86 Architektur verbreiteten Betriebssysteme über die Möglichkeit, mit nachgeladene Gerätetreibersoftware Geräte anzusprechen, welche über einen PCI Bus angeschlossen sind. Zweitens stehen PCI Geräten Interruptleitungen, die für den Datentransport und für die Konfiguration zur Signalisierung wichtig sind, zur Verfügung. Drittens haben PCI Geräte ihrerseits einen Konfigurationsmechanismus, über welchen sich flexibel Port-IO Register Regionen konfigurieren lassen, damit auch mehrere ähnliche Geräte automatisch ihre Konfigurationsregister konfliktfrei in den gewünschten Adressraum einblenden können.

2.8.1 Virtio Geräte aus Sicht des Gastes

Im Folgenden sei nun der gegebene Kommunikationsablauf des Gastbetriebssystems mit dem Virtio Gerät beispielhaft anhand eines virtuellen Blockgerätes dargestellt. Die Virtio Gerätetreiber kann man in ein Frontend und ein Backend unterteilen. Ich möchte dabei als Frontend den Teil des Treibers bezeichnen, der die geräteklassenspezifische, höhere Abstraktion für das Betriebssystem darstellt und als Backend den Teil, der mit Hilfe der Virtqueues und der Konfigurationsschnittstelle letztlich ein virtuelles PCI Gerät bedient.
Für das Beispiel eines Virtio Blockgerätes in Linux wird auf die Linux eigene höhere Abs-

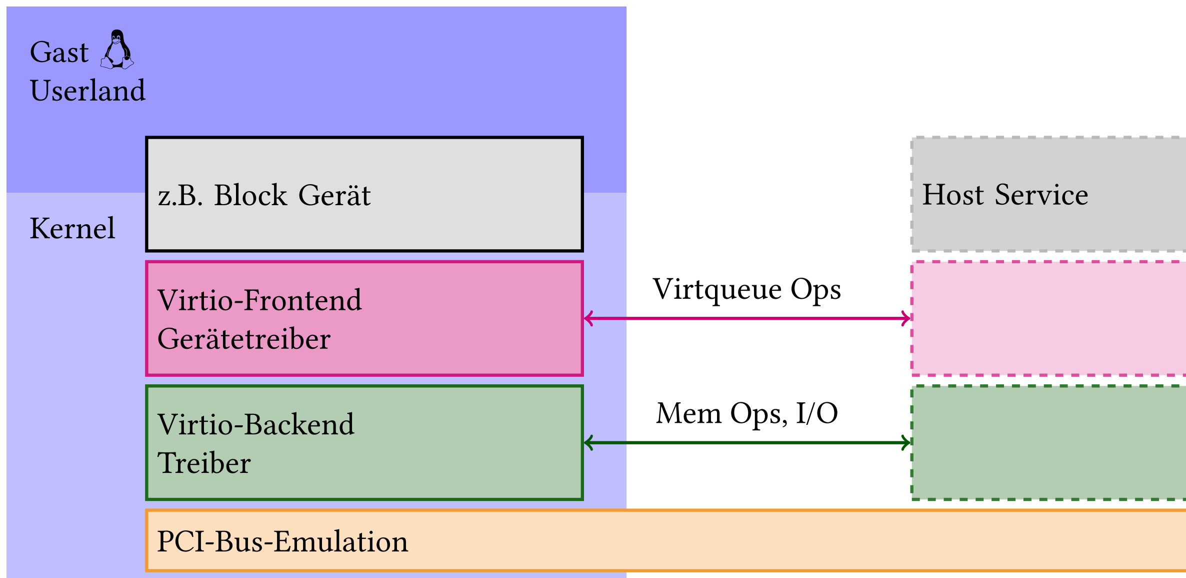

Abbildung 3: Abstraktionsschichten von Virtio Gerätetreibern in Linux Gästen
Hier am Beispiel eines Blockgerätes: Die von Linux ins Nutzerland exportierte Gerätedatei wird im Kern vom *virtio_block* Gerätetreiber über die allgemeine Virtqueue Abstraktion an das vom VMM bereitgestellte, emulierte PCI Gerät angebunden. Im VMM werden die Abstraktionsschichten ähnlich abgebaut, bis hin zu einem Service, der die Aufgabe des Virtio Gerätes im Hostbetriebssystem übernimmt.

traktion von Blockgeräten im Allgemeinen nicht näher eingegangen. Sie ist nur soweit von Belang, dass die Blockgeräteschicht einen bezeichneten Block bekannter Größe anfordern oder im Gerät persistent ablegen können möchte. Näher wird die Blockgeräteabstraktion zum Beispiel in [3, Kapitel 16] beschrieben.
Auch bei Verwendung von mehreren Virtio Geräten, werden diese PCI-Geräte in Linux von einem gemeinsamen Backend Treiber bedient. Über Zugriffe auf die Virtio Konfigurationsregister erfährt dieser Backend Treiber, wie viele Virtqueues maximal für das Gerät zu initialisieren sind, um welche Virtio Geräteklasse es sich handelt, und erhält Konfigurationsoptionen, die nur von dem entsprechenden Virtio-Frontendtreiber verstanden werden. Ein solcher Treiber kann sich mit einer Registrierung (`register_virtio_driver`) zur Verwendung der vom PCI-Backendtreiber bereitgestellten Abstraktion angemeldet haben und nun eine eventuell kleinere als die Maximalzahl der für das Gerät möglichen Virtqueues tatsächlich initialisieren lassen (find_vqueues).
Die Virtqueues selbst sind eine dreiteilige Datenstruktur: Zu jeder Virtqueue gehören eine *Descriptor-Table* (DT), der *Available Ring* (AR) und der *Used Ring* (UR). Das Gastsystem schreibt nun im Virtio-Backendtreiber in jede Zeile der DT eine Adresse, eine Länge und ein Flag, welches anzeigt, ob der Puffer zu lesen oder zu schreiben ist durch den Gegenpart des Backends im VMM. Soll der VMM in den Puffer schreiben, so ist die Länge die zur Verfügung stehende Puffergröße. Ist es hingegen ein Puffer, der vom Gast dem VMM zu lesen gegeben wurde, gibt der Längeneintrag die Größe der zu lesenden Daten in Bytes an.
Ein komplexer Datentransport kann nun aus mehreren zusammengehörigen Einträgen in der DT bestehen. Dafür gibt es außerdem ein Verkettungsfeld, den next-Eintrag, in der DT. Implizit stehen bei einem solchen Transport alle lesbaren Einträge vor den zu beschreibenden. Die beiden anderen Teile der Datenstruktur Virtqueue dienen nun dazu, den komplexen Transport zu synchronisieren. Dazu wird nun der erste der verketteten, zusammengehörigen DT Einträge im AR vom Gast referenziert. Der VMM weiß nun, dass Einträge zur Bearbeitung ausstehen, erledigt dies und referenziert darauf den erledigten Transport im UR.
Diese komplexen Anfragen zeigen, wieder auf das Blockgerätebeispiel gebracht, ihre Nützlichkeit: Der Virtio Blockgerätetreiber benötigt nur eine einzige Virtqueue als Transportmechanismus. Möchte er einen Block lesen, verkettet er drei Einträge in der DT: Erstens lesbar den Anfragenheader, der die angefragte Blocknummer enthält, zweitens einen beschreibbaren Bereich, der als Antwortheader dient und drittens einen Puffer, in welche der VMM die zu lesenden Daten schreiben kann. Der Erste der drei Einträge wird nun im AR der Virtqueue referenziert und ist damit zur Bearbeitung durch den VMM freigegeben. Da der Gast dadurch nicht blockiert, kann er einfach weitere Anfragen an das virtuelle Blockgerät stellen, bis die DT keine freien Einträge mehr aufweist. Spätestens jetzt bietet es sich an, den VMM über die zu bearbeitenden Anfragen zu benachrichtigen. Die dazu vorgesehene, *kick* genannte, Operation erledigt der Gast, indem er in das virtuelle VRING_Q_NOTIFY_PORT Register des PCI Gerätes die ID der gefüllten Queue einträgt. Durch die IO Operation wird eine Umschaltung in den VMM ausgelöst. Der VMM erhält als Grund den Schreibzugriff des VRING_Q_NOTIFY_PORT Register des Virtio PCI Gerätes. Der für diese Virtqueue zuständige Code ist über das Gerät und die Virtqueue ID eindeutig bestimmt und wird aufgerufen. Hat dieser aus einem entsprechenden Speicher die Antwortdaten und den Header dazu geschrieben, trägt der VMM jede bearbeitete Anfrage im UR ein.

2.8.2 Virtio Interrupts

Wie physische Hardwaregeräte in physischen Maschinen auch, können Virtio Geräte mit Interrupts, also Unterbrechungsroutinen, den normalen Kontrollfluss der virtuellen Maschinen unterbrechen. Da Virtio Geräte ja nur virtuell sind, müssen diese Interrupts von VMM in die PCI Bus Nachbildung der virtuellen Maschine eingebracht werden. Je nach den vorhandenen Möglichkeiten des VMMs registriert der PCI Backend Treiber sich zur Behandlung von Interrupts: Bevorzugt werden für jede Virtqueue und für Konfigurationsänderungen ein eigener Interrupt angemeldet. Dazu ist jedoch das Vorhandensein von MSI (Message Signaled Interrupts) in der PCI Emulation erforderlich, denn erst mit MSI-X kann ein PCI Gerät Auslöser für mehrere verschiedene Interrupts sein.
Schlägt der Versuch fehl, für jede der angeforderten Virtqueues eine eigene MSI-X-Vektor-Routine anzumelden, so wird versucht, allen Virtqueues eine gemeinsame Routine, sowie eine weitere Routine für Konfigurationsänderungen anzumelden.
Steht MSI-X überhaupt nicht zur Verfügung, so bleibt als ungünstigste Möglichkeit, lediglich einen PCI Interrupt zur Verarbeitung anzumelden. Dies ist eine deutlich langsamere als die MSI-X Methode, da nicht nur bei jedem Interruptereignis die betroffene Virtqueue herausgefunden werden muss, sondern auch die virtuelle Interrupt-Leitung jedes Mal bestätigt werden muss, was einen weiteren Gast-VMM Kontextwechsel benötigt.

3 Entwurf

Geräte zu paravirtualisieren, kann wie in den vorgehenden Kapiteln angedeutet, durchaus gewinnbringend zu einer Vollvirtualisierung hinzugefügt werden. Es lassen sich sowohl Leistungssteigerungen im Betrieb der virtuellen Maschinen, als auch eine Reduktion der Komplexität der Gerätemodelle erhoffen. Hinsichtlich der Hostanbindung sind dazu einige Details besonders wichtig. Ich möchte nun diese im folgenden Kapitel erläutern, und die grundlegenden Möglichkeiten einer Implementierung unter dem L4Re basierten Palacios VMM ausloten. Dazu sind zunächst die Möglichkeiten, in Palacios Gerätemodelle abzubilden, interessant.

3.1 Gerätemodelle in Palacios

Bei den in die Palaciosbibliothek integrierten Gerätemodellen handelt es sich um das virtuelle Pendant zu den in Kapitel 2.1.2 beschriebenen Hardwaregeräten auf physischen Maschinen. Die Funktionalität, welche Geräte in der Hardware physischer Maschinen bereitstellen, wird bei diesen Gerätemodellen von Softwareprogammen übernommen. Außerdem können im VMM Zugriffe auf ein Gerätemodell direkt auf die Hardware des Hostsystems weitergereicht werden, wenn der VMM das Gerät zur Benutzung durch den Gast freigibt.
Alle Gerätemodelle registrieren sich beim Gerätemananger von Palacios. Je nach Konfiguration einer virtuellen Maschine werden bei ihrer Initialisierung dann Initialisierungscallbacks

für die Gerätemodelle aufgerufen. In diesen Initialisierungsfunktionen müssen die Gerätemodelle ihre Anbindung an die virtuelle Maschine je nach Gerätetyp herstellen. Diejenigen Gerätemodelle, welche Geräten entsprechen, für die es in der x86-Architektur fest zugeteilte IO-Registerbereiche gibt, sorgen selbständig dafür, dass während ihrer Initialisierung Funktionen in eine gastpezifische Tabelle eingetragen werden, die die Behandlungsroutinen für diese Art von Ressourcenzugriff enthalten. So trägt das Gerätemodell des 8254 PIT bei seiner Initialisierung Behandlungsroutinen für die IO-Ports 0x40-0x43 in die IO-Callback Tabelle des Gastes ein. Diese Ports sind konventionell immer genau von einem PIT belegt. Daher können derartig einfache Gerätemodelle ohne weitere Registration oder dynamische Zuteilung der Ressourcenbereiche einfach angesprochen werden. Eine andere von Geräten benutzte Ressource sind Interrupts. Wie bei Geräten mit festen IO-Bereichen gibt es Interruptnummern, die den Gerätemodellen per Konvention zugeordnet sind. Auch diese werden von den Gerätemodellen einfach benutzt und über eine Funktion der Palaciosbibliothek direkt in die virtuelle Maschine induziert.
Ressourcen, die in der x86 Architektur keine implizite feste Zuteilung haben, werden in einer Konfigurationsdatei den Gerätemodellen zugeordnet. Diese wird mit in das Gastabbild übernommen und kann vom Gerätemanager von Palacios ausgelesen werden.

3.2 Paravirtualisierung in Palacios

Palacios VMMs sind zunächst auf Vollvirtualisierung ausgelegt. Vollvirtualiesierende Gerätemodelle entsprechen meistens besonders verbreiteten physischen Geräten. Damit wird erreicht, dass in viele Betriebssysteme die notwendigen Gerätetreiber schon integriert sind, oder zumindest zum Nachladen bereits zur Verfügung stehen. Gerätemodelle, die zum Standardumfang von Palacios gehören, zählen zu den verbreiteten, bekannten und zielen auf eine breite Unterstützung von Gastsystemen ab. Die Gründe für eine Erweiterung um paravirtualisierte Geräte habe ich in Kapitel 2.1.3 genannt: Spezielle Schnittstellen können Komplexität und Mehraufwand verringern und damit die Leistung der virtuellen Maschinen steigern. Dies ist besonders als Ergänzung zu den Gerätemodellen interessant, die nicht nach optimaler Leistung in der virtuellen Umgebung ausgewählt wurden, sondern wie bei den Palacios Standardgeräten nach breiter Unterstützung und einfach handhabbaren Modellen streben. Da Linux als Gastsystem auch im L4Re Umfeld gute Dienste leistet, ist für diese Ergänzung von Vorteil, auf eine bereits gastseitig in Linux intergrierte Paravirtualisierungsschnittstelle zu achten. Virtio ist eine solche Schnittstelle. Ihre klaren Ziele, eine flexible, vielseitige Grundschnittstelle für verschiedene paravirtuelle Gerätetypen zu sein, begründen die spezielle Eignung im Falle dieser Arbeit. Gegenüber den vorhandenen Gerätemodellen von meist eher älteren Standardgeräten sollte sich eine Leistungssteigerung ergeben. Durch die Einsparung der Emulation von Hardwarefunktionen unter Benutzung vorhandener Funktionen auf höherem Abstraktionsniveau ist außerdem mit einer Vereinfachung der Gerätemodelle zu rechnen.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0x00	Vendor ID		Device ID		Comand Reg		Status Reg		Rev. ID	Class Code			Cache Line	Lat. Timer	Hdr Type	BIST
0x10	Base Addr. 0				Base Addr. 1				Base Addr. 2				Base Addr. 3			
0x20	Base Addr. 4				Base Addr. 5				CardBus CIS Pointer				Subsytem Vendor ID		Subsytem Device ID	
0x30	Exp. ROM Base Addr.				Cap Ptr.	Reserved							Irq Line	Irq Pin	Min Gnt.	Max Lat.

Abbildung 4: Die standardisierten ersten 64 Byte des PCI Geräte Konfigurationsraumes bei einfachen Geräten. Die grau hinterlegten Felder sind für das virtio Konsolen Gerät von Bedeutung und im Text näher erläutert.

3.3 Gast-Linux zu Virtio Schnittstelle

Zunächst ist für die VMM Erweiterung um Virtio Geräte wichtig zu klären, auf welche Weise Virtio in Linux Gastsystemen angesprochen werden soll. Wie in Kapitel 2.8 beschrieben, kann das Backend der Virtio Gerätetreiber verschieden implementiert werden, ohne die Frontend Treiber anpassen zu müssen. Eine denkbare Möglichkeit ist daher, eine speziell für die Virtqueues zugeschnittene angepasste Hypercall Schnittstelle zu verwenden. Tatsächlich gibt es in Palacios mit `vmm_hypercall` eine Möglichkeit, Paravirtualisierungsaufrufe zu integrieren. Allerdings ist die einzige im Hauptentwicklungszweig von Linux implementierte Virtio Gast-Treiberanbindung über einen virtuellen PCI Bus als PCI Gerät im Port-IO Betrieb. Da diese Schnittstelle auch unter Palacios gut benutzbar ist und somit eine Erweiterung von Linux sowohl unnötig als auch ohne zusätzliche Gewinnaussicht ist, liegt es nahe, auch in dem vorliegenden L4Re-Palacios VMM, Virtiogeräte über die bereits in Palacios vorhandene Darstellung durch PCI Geräte in der virtuellen Maschine darzustellen.

Der PCI Standard definiert, dass PCI Geräte selbst einen Konfigurationsadressraum haben, über den verschiedene Grundeinstellungen des Gerätes ausgelesen und verändert werden können. Von den 256 Byte Konfigurationsadressraum sind die ersten 64 Bytes standardisiert [3, Kapitel 12]. Abbildung 4 zeigt eine Übersicht über alle Register in diesen 64 Bytes. Der Virtqueue Konfigurationsmechanismus könnte seine Register in den verbleibenden 192 Bytes unterbringen. Dies wird in der Linux Implementierung jedoch nicht gemacht. Die Virtqueue Konfigurationsregister werden über Port-IO Zugriffe angesprochen, deren Basisadresse im PCI Basisadressenregister 0 (BAR 0) des PCI Gerätes eingestellt ist. Die weiteren Basisadressenregister werden für Virtio Geräte nicht verwendet. Die Felder Vendor ID und Device ID des PCI Konfigurationsraumes werden mit einem konstanten Wert belegt, damit der Linux Virtio Backend Treiber sich für diese PCI Geräte registrieren kann. Einzelne Virtio Geräteklassen werden über die Subsystem IDs unterschieden.

3.4 Palacios zu L4Re Schnittstelle

Palacios ist eine stark in sich geschlossene Bibliothek, um die Portierung auf andere Betriebssysteme durch kleine Schnittstellen möglichst einfach zu halten. Die bei Palacios mitgelieferten Gerätemodelle werden direkt von dem Palacios eigenen Buildsystem mit in die Bibliothek eingebaut. Um jedoch sinnvolle Emulation aller Gerätearten für die virtuellen Maschinen anbieten zu können, ergeben sich Abhängigkeiten in das Hostbetriebssystem. So braucht beispielsweise eine Emulation von einem Festplattencontroller oder einem paravirtualisierten Blockgerät immer im Hostsystem eine Möglichkeit, die vom Gast empfangenen Datenblöcke persistent abzulegen, da dies die vom Gast erwartete Funktion eines solchen Gerätes ist. Ein Netzwerkgerät soll die Kommunikation mit anderen Netzteilnehmern ermöglichen, und eine virtuelle serielle Schnittstelle braucht im Host die Möglichkeit einen Datenstrom zu lesen oder zu schreiben. Eine denkbare Ausnahme bilden Gerätemodelle, die verschiedene virtuelle Maschinen direkt im VMM untereinander verbinden, und somit keinen Anschluss an Hostsystemschnittstellen benötigen. Für größtmögliche Isolation bietet es sich bei leichtgewichtigen VMMs im Besonderen an, auf die Möglichkeit mehr als eine virtuelle Maschine darzustellen, zu verzichten und vielmehr jede virtuelle Maschine mit einer eigenen Instanz des VMM auszustatten. Die verschiedenen virtuellen Maschinen werden damit wieder über Netzwerkgeräte verbunden, die demnach eine Anbindung an die anderen VMM Instanzen des Hostsystems benötigen.
Eine weitere ohne Schnittstelle in das Gastsystem auskommende Gerätemodellausprägung sind Scheingeräte, die nur dazu dienen, Gerätezugriffe für den Gast zu emulieren, da die Software des Gastes zwar das Gerät unumgänglich ansteuert, aber die logische Funktionalität des Gerätes im Kontext einer virtuellen Maschine nicht zwingend erforderlich ist. Ein Beispiel dafür ist etwa ein Betriebssystem, welches den Betrieb auf einer Plattform ohne Grafikkarte als Fehlerzustand ansieht und dadurch nicht wie gewünscht funktioniert, obwohl bei fertig eingerichteter virtuellen Maschinen die Systemfunktion auch ohne bildgestützte Nutzerinteraktion auskommt. Das Scheingerät, welches bei Palacios mitgeliefert ist, lässt sich also sehr gut verwenden, um nichtessentielle Hardwarezugriffe des Gastes schnell, einfach und schadlos abzuwickeln. Doch auch dieses Scheingerät wurde mit einer Außenanbindung ausgestattet, da sie im praktischen Betrieb nützlich und ohne zusätzlichen Aufwand herstellbar war: Es nutzt die in der Palaciosbibliothekenschnittstelle für vielfältige Zwecke erforderlichen Ausgabecallbackfunktionen, und kann mit diesen Zugriffe auf bestimmte IO-Ports protokollieren. Damit lassen sich IO-Zugriffe, deren Gerätemodell unbekannt oder noch nicht funktionsfähig ist, untersuchen.
Will man Gerätemodelle, die eine weitergehende Schnittstelle in die Hostumgebung erfordern, an das L4Re Hostbetriebssystem anbinden, ergeben sich dafür verschiedene Möglichkeiten: Die erste Möglichkeit ist, den VMM mit Code zu versehen, welcher die Hardwareabstraktion des Gastes möglichst unverändert durchreicht und das anzubindende Gerät fast ausschließlich außerhalb des VMMs implementiert. Besteht das Gerät im Gastprozessor beispielsweise aus mit Port-IO angebundenen Registern, können Schreib- und Lesebefehle auf diese direkt an einen Server im L4Re System durchgereicht werden. Der VMM ist lediglich soweit beteiligt, dass ihm Informationen vorliegen, welche von der Gast CPU ausgelösten Hardwarezugriffe an welchen Server im L4Re weiterzuleiten sind. In der Palacios Bibliothek

sind Funktionen vorhanden, die im VMM die Anmeldung von Behandlungsfunktionen für Hardwarzugriffe in den Gerätemodellen zulassen. Im VMM ist dafür nur eine Zuordnung von Gastspeicher und IO-Zugriffen einerseits und Interruptinjektion andereseits auf L4-IPC nötig. Es bietet sich an, dazu das vorhandene Scheingerät zu erweitern. Dieses wird dadurch zu einem Proxy-Gerätemodell und die eigentlichen Gerätemodelle können dann im Nutzerland von L4Re implementiert werden. Ein besonderer Vorteil dieser Methode ist sicher, dass die verwendete Abstraktionsebene für viele Geräteklassen allgemeingültig ist.
Doch gerade eine Stärke von Palacios, die besonders bei herkömmlichen Umgebungen von Palacios zu tragen kommt, ist die starke Kapselung der VMM Bibliothek, die mit minimalem Aufwand einen vollständigen VMM bereitstellt. Die Implementierung aller Gerätemodelle des VMM außerhalb von Palacios zu erwarten, wär ein schwerwiegender Verlust dieser auf der Eigenständigkeit der Palacios-Bibliothek beruhenden Aufwandsminimierung. In der Umgebung der herkömmlichen Palacios VMMs gibt es zudem überhaupt keine starken Anreize, Palacios zu verlassen. Denn sowohl die Bibliothek als auch der restliche VMM muss mit allen Privilegien laufen, und kann somit selbst auf Hardware zuzugreifen. Ein Weiterreichen der Anfragen der virtuellen Maschine an die physische ist dort also einfach ohne das Verlassen der Bibliothek möglich. Außerdem liegen Informationen über den gastphysischen Speicher nur in dessen Verwaltungsdatenstrukturen. Die Schnittstelle müsste also für speichereingeblendete Gerätemodelle auch um diese Informationen erweitert werden, um Gerätemodellen Zugriff auf den gastphysischen Speicher zu gewähren. Der Aufwand, den diese Schnittstelle bedeutet, wiegt demnach die Vorteile, die zudem auch noch L4 spezifisch sind, nicht auf.
Andere Möglichkeiten Gerätemodelle aus der Bibliothek heraus an den VMM und schließlich die L4 Umgebung anzubinden, sind abhängig von den jeweiligen Geräten, die dargestellt werden sollen. Denn auch die Gerätemodelle des VMM haben einen zumeist hierarchischen Aufbau, der die in den Gerätetreibern vorhandenen Abstraktionsschichten wiederspiegelt, wie sie in den Gerätetreibern von Gastsystemen wie Linux vorkommen (vgl. Abbildung 3). An den meisten dieser Abstraktionsschichtgrenzen lässt sich eine sinnvoll einsetzbare Schnittstelle an die VMM Umgebung herstellen. Es ergeben sich aus den Hierarchiebäumen der benutzten Protokollen und Abstraktionen also drei Protokollstapel, um eine einzige Ressource im Gastsystem den Anwendungen anzubieten: Die konkret verwendeten Gerätetreiberschichten, die zugehörigen Gerätemodelle im VMM und die wiederum im L4Re stattfindende Bereitstellung der Ressource. Abbildung 5 zeigt beispielhaft und schematisch dies für eine Dateiabstraktion im Gast.

Für eine Minimierung des Aufwandes und möglichst gute Leistungswerte, muss in der Implementierung der mittlere Stapel, also die VMM interne Gerätemodellhierarchie, möglichst klein gehalten werden. Allerdings findet in diesem mittleren Stapel die Adaption an das Umgebungssystem statt. Je nach bereits vorhandenen Diensten in der Umgebung und zusätzlichen Anforderungen an die Darstellung der Ressource lässt sich hier eine gewisse Mehrschichtigkeit aber nicht immer vermeiden. Die schon beschriebene Anbindungsmöglichkeit mit direktem Hardwarezugriff, also lediglich einem Proxy Gerätemodell ist hier wieder als Extrembeispiel zu nennen: Eine besonders kleine Adaptionsschicht bedeutet weniger Flexibilität, gröbere Ressourcenbeschränkungsmöglichkeiten, aber dafür sehr wenig Zusatzaufwand gegenüber der Ausführung des Gastsystems auf physischer Hardware.
Virtio-Gerätemodelle haben bisher keine direkten physischen Entsprechungen. Ihr hierar-

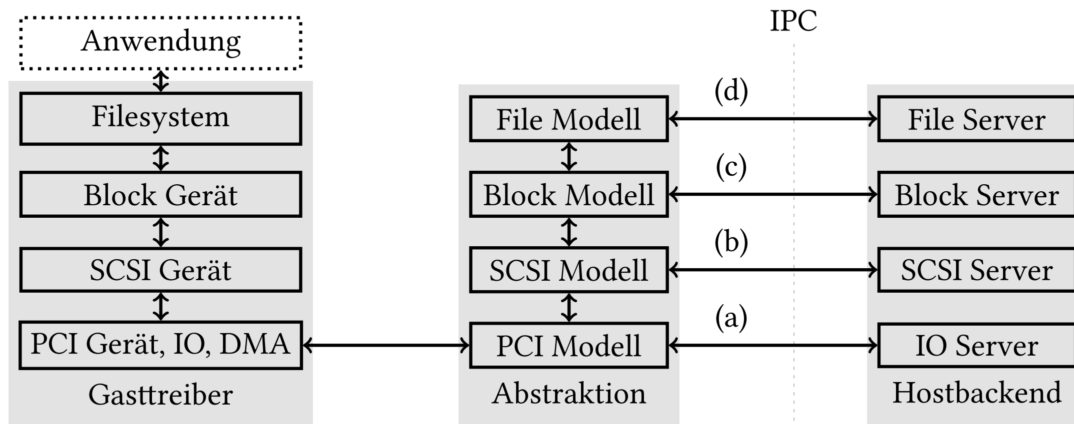

Abbildung 5: Anbindung der Gasttreiberhierarchie über Adaptionshierarchie an Hostservice Für minimalen Aufwand ist aus den Alternativen (a) bis (d) die niedrigste zu wählen, welche im Hostsystem verfügbar ist.

chisch organisierter Aufbau ermöglicht aber dennoch verschiedene Möglichkeiten der Adaption an die Abstraktionen im Hostsystem. Diese resultieren aus dem gemeinsamen abstrakten Mechanismus, der allen Virtio Gerätemodellen zur einheitlichen Kommunikation mit dem Gastsystem dient, den Virtqueues. Da es sich bei diesen bereits um eine wohldefinierte Schnittstelle handelt, geschaffen um flexibel, klein und sauber zu sein, kann es eine leistungsfähige und elegante Lösung sein, das Virtque-Backend also noch im VMM mit zu abstrahieren und die Virtqueues als Schnittstelle in das Hostsystem zu nutzen. Dieses Virtqueue Interface können alle L4 Nutzerland Server implementieren, die in L4Re Funktionen für die VMM Gäste bereitstellen sollen. Der mittlere Stapel, also der Adaptionscode ist damit sehr flach und für alle Virtiogeräte gleich. Die weitere Implementierung der einzelnen verschiedenen Virtiogeräte wird damit in die Server des L4Re ausgelagert. Es muss also im rechten der Protokollstapel diese Schnittstelle mit angeboten werden. Die Gerätemodelle im VMM müssen dazu nur über den Konfigurationsmechanismus von Virtio mit den richtigen Servern im L4Re Nutzerland verbunden werden, die diese Art von Virtiogerätemodell unterstützen. Für eine solche Schnittstelle ist ein L4 Server nützlich, bei welchem sich Virtiogerätemodellproviderserver für die Behandlung einer speziellen Virtqueue registrieren können. Der VMM muss mit einer Capability zur Kommunikation mit diesem Namensdienst gestartet werden und kann dann auf entsprechenden gastseitigen Anfragen nach Virtqueues einen für diesen Gerätetyp registrierten Server erfragen und via IPC mit dem Gerätemodell in Palacios verbinden. Abbildung 6 stellt den Protokollstapel dieses Anbindungsentwurfes schematisch dar.

Diese Art von Schnittstelle kann dann auch von L4Re Anwendungen, vor allem anderen L4Re Servern, benutzt werden. Ein fiktiver Modem-Wählverbindungsserver mit Virtqueue IPC ist zum Beispiel sowohl als Unterbau für ein Virtio basiertes, serielles Gerätemodell, als auch für ein im L4Re laufendes Terminalprogramm nützlich. Abbildung 7 illustriert ein ähnliches Beispiel mit Blockgeräten.

Einen etwas anderen Schwerpunkt liefert die Anbindung über die in Abbildung 8 einge-

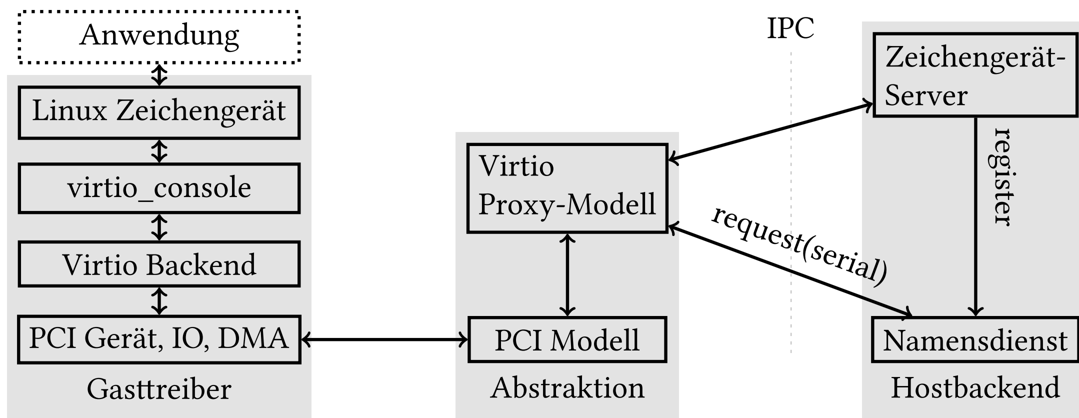

Abbildung 6: Virtio basiertes, serielles Gerät — Anbindungsvariante
In diesem Bild ist ein zeichenorientiertes Gastgerät über die *virtio_console* Abstraktion an einen Dienst im L4-Nutzerland angebunden, der eine Virtqueueschnittstelle anbietet und sie dazu über einen Namensdienst bekannt gemacht hat. Das Proxy-Gerätemodell im VMM stellt die logische Verbindung zwischen Gasttreiber und Hostserver sicher.

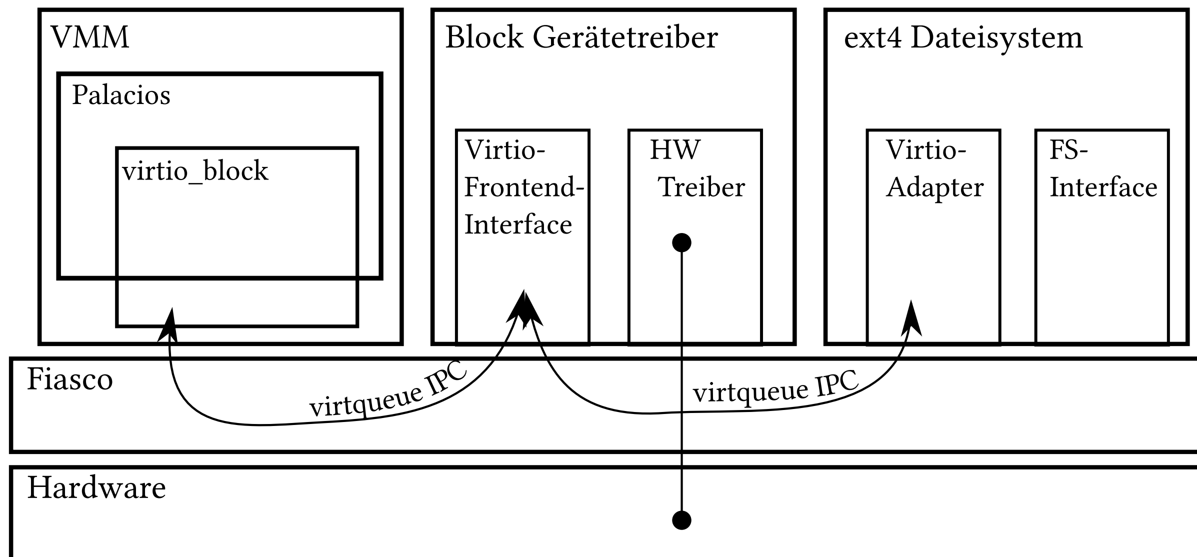

Abbildung 7: Virtqueues als Kommunikationsinterface
Nutzt man die Virtio Virtqueue-Schnittstelle als eigenständige Hardwareabstraktion ergeben sich im Mikrokernsystem auch andere Anwendungen. Hier ist die Verwendung einer Blockgeräteabstraktion mit dem VMM und einem Filesystemserver dargestellt

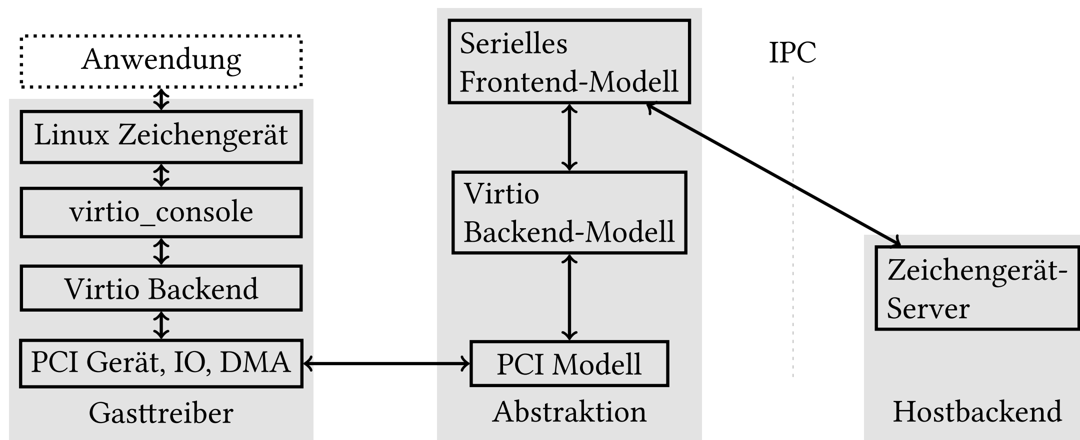

Abbildung 8: Virtio basiertes, serielles Gerät — Anbindungsvariante
In diesem Bild ist ein zeichenorientiertes Gastgerät über die *virtio_console* Abstraktion an einen Dienst im L4-Nutzerland angebunden, der eine Virtqueueschnittstelle anbietet und sie dazu über einen Namensdienst bekannt gemacht hat. Das Proxy-Gerätemodell im VMM stellt die logische Verbindung zwischen Gasttreiber und Hostserver sicher.

ordneten Protokollstapel. Der Adaptionscode ist etwas aufwändiger und durch einen gerätemodelltypenabhängige Abstraktionsschicht nicht mehr für alle Virtio Geräte gleich. Dafür können schon vorhandene Server im L4Re direkt benutzt werden. Ich habe daher mein Beispiel, das virtio_consolen-Gerätemodell, als ein derartig angebundes Gerätemodell ausgewählt. Denn bei ähnlich eingeschätztem Aufwand und ohne erkennbaren Leistungsvorteil gaben nachrangige Gründe wie die schlechtere Beherrschung der im L4Re bevorzugten Programmiersprache C++ den Ausschlag.

4 Implementierung

Das folgende Kapitel stellt Aspekte der Testimplementierung eines bestimmten Virtio Gerätemodells für L4Re dar. Dazu ist zunächst eine Auswahl aus den prinzipiell zur Verfügung stehenden Virtio Geräten zu treffen.

4.1 Mögliche Implementierungskandidaten

Die folgenden einzelnen Gerätemodelle sind in Linux schon länger gastseitig benutzbar und kommen daher prinzipiell für eine Anbindung an den L4Re-Palacios VMM in Frage. Doch nicht zu allen existieren im L4Re geeignete Services, die einfach sinnvolle Tests ermöglichen.

4.1.1 virtio_ballon

Dieses spezielle Virtio-Gerät dient zur effizienten Speicherausnutzung in Virtualisierungsumgebungen. Der Gasttreiber für dieses Gerät kann durch Allokation von physischer Spei-

cher dem Gastkernel diesen Speicher entziehen. Der Hostmonitor kann den Speicher dann anders verwenden, um beispielsweise einer momentan stärker ausgelasteten anderen virtuellen Maschine mehr Speicher zuzugestehen. Es ergibt sich also ein Mechanismus zum dynamischen Verändern der Hauptspeichergrößen der virtuellen Maschinen. Dazu baut über eine Virtqueue für jede Kommunikationsrichtung das Modell einen Befehlskanal auf, mit dessen Hilfe die Steuerung der Speichergrößenveränderung aus dem VMM heraus vorgenommen werden kann.
Da diese Gerät aus Virtio Sicht betrachtet einfach gehalten ist und die Schwierigkeiten hier vor allem im Speichermanagement unter Linux liegen, habe ich diesem Gerät keine vorrangige Bedeutung als Gerät für die Testimplementierung zugemessen.

4.1.2 virtio_net

Beim virtuellen Netzwerk-Gerät profitiert die virtuelle Maschine von der Paravirtualisierung besonders hinsichtlich der Vermeidung unnötiger Darstellung virtueller Hardware: Die Abstraktionsschicht „Netzwerkgerät “wird direkt vom Hostsystem weiter verarbeitet. Es gibt bereits eine Anbindung in Palacios, die eine Socket-artige Abstraktion von Netzwerkhardware als Callbackroutinen vom VMM verlangt. Eine solche etablierte Abstraktionsschicht für Netzwerk ist mir für L4Re nicht bekannt. Andernfalls wäre eine Netzwerkanbindung der Gäste sicher ein lohnenswertes Testobjekt für Paravirtualisierung im zu erweiternden VMM gewesen.

4.1.3 virtio_block

Das virtuelle Block-Gerät eignet sich, um der virtuellen Maschine effizient Massenspeicher zur Verfügung zu stellen. Die Vorspiegelung von bekannten Hardwaregeräten benötigt einige im Host zu behandelnden Unterbrechungen des Gast-Ausführungsfadens, die zum Teil unnötig sind: Direkt weitergereichter Massenspreicher aus der Hostumgebung vereinfacht die Ressourcenverteilung durch den VMM und vermeidet überflüssige Hardwarenachbildung. Eine einzelne Virtqueue reicht aus, um die Struktur des abstrakten Blockgerätes abzubilden, da ein Blockgerät nur sendet, wenn Daten per Anfrage erbeten wurden (siehe dazu das in Kapitel 2.8.1 gezeigte Beispiel der Virtqueue Kommunikation).

4.1.4 virtio_console

Auch die Emulation von hardwareidentischen seriellen Schnittellen in den virtuellen Maschinen ist unnötig kompliziert, wenn es nur darum geht, ein zeichenorientiertes Gerät, z.B. zur Konsolenausgabe, darzustellen. Maßnahmen, die der sicheren Übertragung auf einem physischen Kabel dienen, wie etwa die Auswahl einer geeigneten Datenwortlänge und Zeichenfrequenz, sind für Kommunikation zwischen Host und Gästen nicht erforderlich. Es liegt auf der Hand, mit Hilfe des von Virtio geschaffenen Transportmechanismus direkt den (bidirektionalen) Zeichenstrom an das Hostsystem zu übermitteln. Eine Konsole, die auch vom Linux Kernel als solche verstanden wird, ist ein Spezialfall eines solchen virtuellen seriellen Gerätes.
Zeichenströme lassen sich auch in L4Re gut verarbeiten und mit Vcon ist eine Schnittstelle in

L4Re vorhanden, die sich genau auf eine derartiges Gerätemodell aufsetzen lässt. Daher habe mich für bei der beispielhafte Realisierung eines Virtio Gerätes für die Implementierung der Gegenseite zu `virtio_console`, dem Gerätetreiber serieller Virtio Geräte, entschieden.
Die Vcon Schnittstelle wird von dem für Testbelange sehr praktischen fbterminal Server bereitgestellt, der es erlaubt, Tastatureingaben in das Gerätemodell zu schicken und Ausgaben in einem Fester des Fenstersystems darzustellen. Und nicht nur für das Hostsystem ist die Wahl von diesem Gerät eine praktische Entscheidung: Dass auch im Gastlinux nicht viele Ressourcen oder komplizierte Vorgänge nötig sind, um das Virtio Gerät anzusprechen, ist von zusätzlicher Bedeutung. Einfache Lese- und Schreiboperationen lassen sich gut auch mit dem eingeschränkten Bildschirm des Gastes zur Fehlerbeseitigung beobachten und erzeugen. Netzwerk- oder Blockgeräte hätten hier höhere Ansprüche an die Testumgebung gestellt.
Außerdem handelt es sich bei `virtio_console` um eins der Geräte, für die in Palacios bisher gar kein Code verfügbar war und welches besonders viele Virtqueues benutzt. Kommt das Blockgerät etwa mit einer einzigen Virqueue aus, werden für das implementierte serielle Gerät bei n Ports $2n+2$ Virtqueues benutzt, seit in Linux Multiport Unterstützung eingebaut ist.

4.2 virtio_console in Linux im Detail

Das als konkretes Beispiel implementierte Virtiogerätemodell zielt auf den in Linux vorhandenen `virtio_console` Gerätetreiber ab. Es ist sowohl als generelles serielles, zeichenorientiertes Gerät ausgelegt, als auch für besondere Geräte mit bildschirmartiger Darstellung der Ausgabe und Eingabe geeignet. Diese sogenannten Konsolengeräte haben wenige zusätzliche Eigenschaften und stellen einen wichtigen Anwendungsfall von paravirtualisierten seriellen Zeichengeräten dar. VMMs erlauben etwa über solche Geräte die Steuerung der Gastsysteme, auch wenn an diese keine echte Ein- und Ausgabehardware angebunden ist.
Frühere Versionen des Treibers unterstützten genau eine virtuelle, serielle Schnittstelle, die immer als Konsolengerät angesprochen wurde. Der Kompatibilität zu diesen früheren Versionen ist die weiter unten angegebene Durchnummerierung der verwendeten Virtqueues geschuldet. Denn der heute im Linux-Kern vorhandene Treiber unterstützt die sogenannte Multiport-Eigenschaft: Der Treiber stellt virtuell mehrere solcher zeichenorientierter Geräte, sogenannte Ports, bereit, die nicht unbedingt eine Konsole sein müssen. Dies ist nicht weiter besonders, denn auch physische Hardware mit derartigen Aufgaben erlaubt häufig mehr als einen einzigen Kommunikationskanal. Das für den reinen Vollvirtualisierungsbetrieb in Palacios vorgesehene Gerätemodell spiegelt etwa dem Gast die üblichen vier seriellen Schnittstellen (COM1 - COM4) mit zum 16550 kompatiblen UART Schnittstellenbausteinen vor.
Da serielle Kommunikation prinzipiell zeitgleich in jede der beiden Richtungen erfolgen kann, ist für jede eine eigene Virtqueue vorgesehen. Es werden für den Multiport Betrieb also zwei Virtqueues pro Port vorgesehen. Außerdem wird ein Konfigurationsnachrichtensystem bereitgestellt, für welches insgesamt zwei weitere Virtqueues angelegt werden. Die Konfigurationsnachrichten informieren den Gast über Anzahl, Typ (Konsole oder nicht), Bereitschaft und Namen der vorhandenen Ports.
Damit eine Kommunikation zwischen veralteten Implementierungen in entweder Host- oder

Gast-Betriebssystem und den modernen Gegenstücken möglich ist, hat man in den modernen Implementierungen die Virtqueues mit den Indices *0* und *1* weiterhin für einen Konsolen-Port, darauffolgend mit *2* und *3* die Konfigurationsnachrichtenqueues und erst darauf folgend jeweils zwei Virtqueues pro zusätzlichen Port reserviert. Dem Port mit den Virtqueues der Indices *0* und *1* kommt dabei eine Sonderbehandlung zu, da es nicht nur der einzig verfügbare in Treibern ohne Multiportunterstützung ist, sonder auch derjenige Port, der im Linux Modul möglichst früh aktiviert wird, um auf der angebotenen Konsole schon Ausgaben tätigen zu können. Dies ist vor allem in VMMs sinnvoll, die keine weitere Möglichkeit zur Logausgabe im frühen Bootstadium der virtuellen Maschine anbieten.
Bei vorhandener Multiport-Unterstützung in Host und Gast, wird von den besonderen Eigenschaften eines Konsolenports aber erst nach einer entsprechenden Ankündigung des Hostes Gebrauch gemacht, die einen Port als einen solchen Konsolenport ausweisen.

4.2.1 Control Messages

Seit Einführung der Multiportunterstützung gibt es so genannte Control-Messages im `virtio_console` Treiber. Über diese einfachen Datenpakte teilen sich beiderseits Host und Gast verschiedene Zustandsänderungen in Treiber und Gerätemodell mit. Die wesentliche Nachricht wird über das Nachrichtentypfeld ausgetauscht, von welchem es folgende Ausprägungen gibt:

DEVICE_READY Gasttreiber meldet den Zustand seiner Bereitschaft.

PORT_ADD Der Host versucht, einen neuen Port hinzuzuschalten.

PORT_REMOVE Der Host schaltet einen Port ab.

PORT_READY Gast konnte Port hinzuschalten und erwartet nun die Konfiguration durch den Host.

CONSOLE_PORT VMM teilt dem Gast mit, dass es sich um einen Konsolenport handelt.

CONSOLE_RESIZE Der Host teilt dem Gast eine Dimensionierungsänderung einer Konsole mit.

PORT_OPEN Host oder Gast teilen dem jeweils anderem Kommunikationsende mit, dass das Gerät geöffnet wurde.

PORT_NAME Der Host teilt dem Gast den Namen des betreffenden Ports mit. Ein Linux Gast stellt diesen im sysfs zur Erkennung bereit.

Weiterhin besitzen die Nachrichten außerdem eine Port-ID als Portspezifikation und ein Wert-Feld, welches den konkreten Anlass der Nachricht weiter spezifiziert. So werden beispielsweise Control-Messages mit dem Event-Typ DEVICE_READY und dem Wert „0“ als Negation einer vorausgehend signalisierten Treiberbereitschaft interpretiert.

4.3 Schnittstellenimplementierung

Die beschriebende Kapselung von Palacios im VMM macht eine wohldefinierte Schnittstelle sowohl vom in Palacios eingebauten Gerätemodell zum L4Re Verbindungscode im VMM als auch vom VMM zur L4Re Anwendungsumgebung nötig.

4.3.1 Anbindung des VMM an Vcon

Um eine sinnvolle Testumgebung mit einer möglichen Anwendung des virtio_console Gerätemodells zu erhalten, habe ich die virtuelle serielle Schnittstelle mit dem vom *fbterminal* Paket bereitgestellten virtuellen Terminal verbunden. *fbterminal* stellt auf einem Framebuffer ein Ausgabefenster dar und ermöglicht die Signalisierung von Tastenanschlägen, die getätigt wurden, während das *fbterminal*-Fenster den Fokus hat. Das *fbterminal* bietet dafür zwei Schnittstellen an: Zum einen stellt es sich als *ICU* dar, einer Interrupt Controller Unit, bei der genau eine Interruptleitung erhältlich ist. Der zugehörige Interrupt wird immer bei Eingabe in das *fbterminal*-Fenster ausgelöst. Außerhalb der Palacios Bibliothek ist im VMM ein Stück Code implementiert, welcher über die ICU sich für den angebotenen Interrupt registriert und einen eigenen Thread startet, der immer wieder aufwacht, wenn der Kern diesen über das Eintreten des Interruptereignisses benachrichtigt. Da die Tasteneingabegeschwindigkeit im Verhältnis zur Dauer der Abarbeitung des Interrupts sehr gering ist, wird nahezu für jedes einzelne Zeichen ein eigener Interrupt ausgelöst. Dass die Versendung der eingegebenen Zeichen jedes Mal einen zusätzlichen Host-Gast-Übertritt verursacht, ist bei den Häufigkeiten der Tastenanschläge nicht geschwindigkeitskritisch. Außerdem handelt es sich bei *fbterminal* um eine Implementierung der Vcon Schnittstelle. Diese stellt sowohl eine Lesefunktion bereit, mit welcher die eigentlichen Tastendrücke, die seit dem letzten Interruptereignis eingetreten sind, abgeholt werden können, als auch eine Schreibfunktion, welche einen Zeichenstrom in das *fbterminal* Fenster schreibt. Dabei werden vt100 Terminal-Steuersequenzen interpretiert.

4.3.2 Palacios Schnittstelle

Das Gerätemodell ist weitestgehend im Palacios-Teil des VMMs implementiert. Eine Übernahme dieser Codeteile für andere Portierungen von Palacios ist damit weitgehend möglich. Die Palaciosbibliothek wird, um eigenständig kompilierbar zu sein und nur wohldefinierte Abhängigkeiten in das umgebende Betriebssystem zu erzeugen, mit eigenen Make-Skripten und eigenen grundlegenden Headerdateien gebaut. Meine Schnittstelle vom Gerätemodell in der Palaciosbibliothek zum VMM muss sowohl in den Objektdateien, die vom L4Re Buildsystem gebaut werden, als auch in den Palacios-internen Objekten verwendbar sein. Sie enthält daher nur C-Grunddatentypen. Die Verwendung anderer Standarddatentypen wie `size_t` hätten zu für den Compiler inkompatiblen Redefinitionen eben dieser geführt.
Die Schnittstelle ist an der Schnittstelle zu seriellen, zeichenorientierten Gerätedateien unter Linux orientiert: Es sind Funktionen aus der Palaciosbibliothek heraus exportiert, mit denen eine solcher virtio_consolen Port geöffnet, gelesen, beschrieben und geschlossen werden kann. Da jedoch auf der Gastseite der Port ebenso bedient werden kann, habe ich Callbacks in der Port-Datenstruktur eingefügt, die bei gastseitigen Ereignissen im Host entsprechende

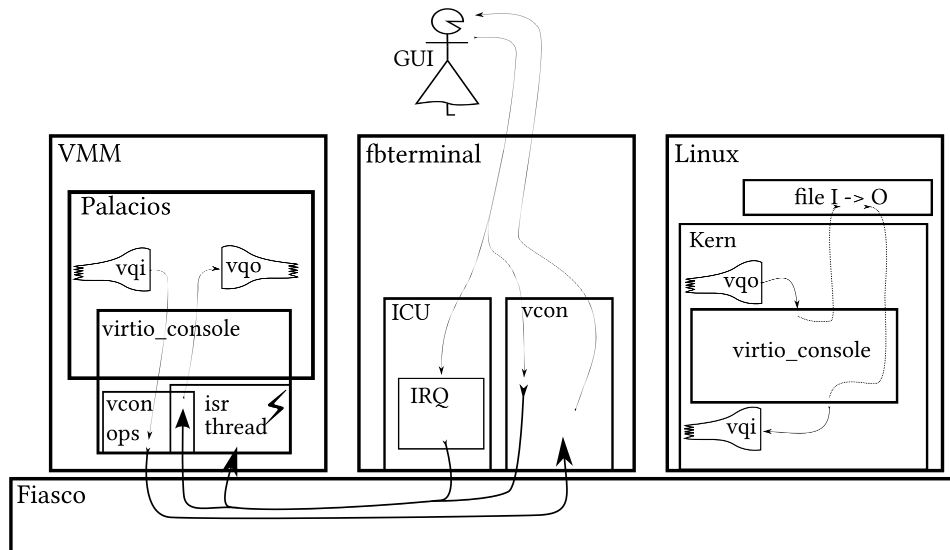

Abbildung 9: Kommunikation im Testaufbau.
Dargestellt sind alle beteiligten Komponenten: Ein Nutzer interagiert mit der *fbterminal* Anwendung, welche über Interrupt IPC und die Vcon Schnittstelle mit dem VMM kommuniziert. Über die Virtqueues vqo und vqi gelangen die Daten Schließlich in das Linux Gastsystem. Die Kommunikationsrichtung ist jeweils mit Pfeilen angegeben.

Maßnahmen ausführen können.
Es hat sich als nützlich erwiesen, zu Testzwecken bestimmten übertragenen Bytesequenzen direkt im Gerätemodell Funktionen zuordnen zu können. Damit kann beispielsweise eine Zählroutine für Gast-VMM Übertritte an- und abgeschaltet werden, die für Leistusgsvergleiche mit nichtparavirtualisierenden Gerätemodellen herangezogen werden kann.

4.4 Interrupt-Thread

Das Gerätemodell wird hier, wie in Abschnitt 4.3.1 beschrieben, mit einem eigenen Thread betrieben, welcher lediglich auf Interrupt-IPC Ereignisse vom *fbterminal* wartet und in die Gastmaschine entsprechende Tastenanschlagsinterrupts induziert. Dieses Vorgehen ist nicht unbedingt nötig. Da Threads auch einen nicht vollkommen unerheblichen Ressourcenaufwand verursachen, ist zu überlegen, den Thread einzusparen. Dies führt allerdings zu einer schlechteren Kapselung von Virtio-Frontend und -Backend. Denn die Behandlung des Hostinterrupts muss dazu beim Austritt aus dem Gast geschehen, obwohl sie nur bei der tatsächlichen Verwendung eines solchen Gerätemodells eine Rolle spielt. Da für jedes andere Virtiogerät andere Benachrichtigungen aus dem Hostsystem notwendig sind, halte ich die Kapselung der Interruptbehandlung an das Virtio Frontendmodell mit einem eigenen Thread für sinnvoller. Außerdem kann bei Mehrprozessorsystemen das Gerätemodell hier schon ähnlich physischer Hardware die Datenverarbeitung im Modell parallel zum Gast ausgeführt werden.

4.5 PCI Gerätemodell

Für die Gegenseite der Implementierung des Virtio PCI Virtqueue Unterbaus, ist die virtuelle Version eines PCI Gerätes nötig. Palacios liefert fertige Funktionen, um PCI Geräte an einem virtuellen PCI Bus der virtuellen Maschine darzustellen. Die von der Palaciosbibliothek angebotene PCI Emulation unterstützt dabei bisher keine MSI-X Erweiterung. Virtio fällt damit in den ungünstigsten Betriebsmodus zurück, bei welchem lediglich die eine emulierte Interruptleitung des virtuellen PCI Gerätes zur Verfügung steht. Dies ist auch bei der Erstellung der Behandlungsroutinen für die Zugriffe auf die virtuellen IO Ports des PCI Modells zu beachten. Die verschieden aufgeteilten IO-Regionen können sonst zu Fehlern in der Zuordnung von Offset relativ zu dem einzigen verwendeten Basisregister und Gerätemodellregister führen.

4.6 Gast Linux

Zum einfachen Erstellen des Gast Images, habe ich mir mit Buildroot [2] ein System zur möglichst automatischen Erstellung kompletter Linux Systeme ausgesucht. Diese Linux Distribution findet normalerweise in eingebetteten Systemen Anwendung, da sie vorzugsweise kleine uClibc basierte Nutzerumgebungen baut. Um Abhängigkeiten an funktionierende Blockgeräteemulation auf der virtuellen Maschine anbieten zu können, war die Größe des gebauten Root Dateisystems der Distribution insofern wichtig, als dass es möglichst in den Hauptspeicher der virtuellen Maschine geladen können werden sollte. Dies, die leichte Erweiterbarkeit und die Verfügbarkeit von x86 als Zielarchitektur machen Buildroot zur einer geeigneten Distribution für das gegebene Umfeld. Als Ausgabe habe ich eine CD-Abbild Form gewählt. Dieses wird von dem in Palacios mitgelieferten Gastabbildersteller verstanden und mit einer Konfigurationsdatei für die virtuelle Maschine zu einem Palacios spezifischen Gastabbildformat zusammengesetzt. Das iso9669 Dateisystem des CD-Abbildes enthält einen GRUB Bootloader und kann auf den meisten normalen PCs als Demonstrationssytem ohne Installation verwendet werden. Zu den mitgebauten Nutzerlandprogrammen habe ich einen Lua-Interpreter und einen VIM Texteditor gewählt, damit ich den virtio_console Treiber am laufenden System mit über einfache Kommandozeilenbefehle hinausgehenden Mitteln testen kann.

5 Leistungsbewertung und Ausblicke

Der Gewinn für Palacios unter L4Re, der durch diese Arbeit entsteht, ist in erste Linie nicht in der Leistungssteigerung oder Komplexitätsreduktion von seriellen Übertragungen zwischen Linux Gästen und dem VMM zu sehen.
Sicher sind nur sehr wenige Host-Gast Übertritte nötig. Eine über das Interface geleitete Datei von zehn Kibibyte ($10 * 2^{10} Byte$) verursacht nur etwa zehn Übertritte. Die Daten werden zwar beim Schreiben im Gastlinux vom Nutzerland in den Kernadressbereich kopiert. Da aber dieser Kopiervorgang nur dem Aufbau der Schnittstelle in Linux geschuldet ist und nicht von dem paravirtuellen Gerätemodell herrührt, kann man sagen, dass kein Kopiervorgang für die Datenübertragung in den VMM stattfinden muss. Je nach vorgespiegelter Hardware

ist bei der vollvirtualisierenden Gerätemodellversion von mehr als einem Übertritt für jedes zu übertragende Byte auszugehen. Der eingesparte fingierte Einstellungsoverhead, beispielsweise für das Setzen von Übertragungsmodus und Baudrate, ist wie auch die Kommunikation zur Konfiguration der Virtqueues nach einmaliger Ausführung für die Leistungswerte nicht ausschlaggebend. Ein reiner Leistungsgewinn durch die Verwendung von `virtio_console` gegenüber einer klassischen, emulierten seriellen Schnittstelle mit virtuellen Abbildern von UART Bausteinen ist aber auch nur von nachgeordneter Bedeutung. Denn in Anwendungen, wo die Übertragungsbandbreite ein wichtiges Ziel ist, spielen diese Geräte aktuell keine wichtige Rolle.
Der eigentliche Gewinn liegt in der Möglichkeit, Virtio Geräte überhaupt ansprechen zu können: Diese Arbeit zeigt Wege auf, die Möglichkeiten aller Virtio-Paravirtualisierungsgeräte für Palacios unter L4Re nutzbar machen. Palacios unter L4Re hat sich als praktisches VMM Framework herausgestellt, welches sich im Rahmen von L4Re durchaus erweiterbar zeigt. Für eine weitere Verbesserung der Paravirtualisierungsleistung, sollte der virtuelle PCI Bus von Palacios um MSI-X erweitert werden.

5.1 Schlüsselstellen der Implementierung

Für eine erweiterte Benutzung von Virtio unter L4Re basiertem Palacios VMM ist die Anbindung der in der Palaciosbibliothek verankerten Gerätemodelle an eine L4Re Schnittstelle ein für jeden Gerätetyp individuell zu klärendes Problem. Ich halte die Anbindung auf hohem anwendungsnahem Abstraktionsniveau für die aussichtsreichste Möglichkeit. Es ergibt sich mit der Arbeit der Wunsch, die Ausgliederung der Behandlung der Virtqueues aus dem einzelnen Gerätemodell zu schaffen, wie sie ganz ähnlich im Linux-Gastsystem als Trennung von Front- und Backend vorliegt. Zwar sind einzelne Teile schon isoliert im Code vorhanden, von der vollständige Abstraktion der Virtqueue-Schicht wurde bisher wegen mangelnder weiterer Testgerätemodellanbindungen abgesehen.

Literatur

[1] *AMD64 Architecture Programmer's Manual Volume 2: System Programming*, 2010. URL: http://support.amd.com/us/Processor_TechDocs/24593.pdf.

[2] *BuildRoot Version 2010.11*, 2010. URL: http://buildroot.uclibc.org.

[3] Corbet, J., A. Rubini und G. Kroah-Hartman: *Linux device drivers.* O'Reilly Media, Inc., 3 Auflage, 2005.

[4] *L4Re Reference Manual*, 2011. URL: http://os.inf.tu-dresden.de/L4Re/doc/main.html.

[5] Goldberg, Robert P.: *Architectural Principles for Virtual Computer Systems.* Dissertation, Harvard University, Cambridge, MA, 1973. URL: http://www.dtic.mil/cgi-bin/GetTRDoc?AD=AD772809&Location=U2&doc=GetTRDoc.pdf.

[6] Ierusalimschy, R., L.H. De Figueiredo und W. Celes: *The implementation of Lua 5.0.* Journal of Universal Computer Science, 11(7):1159–1176, 2005. URL: http://www.lua.org/doc/jucs05.pdf.

[7] *Intel® 6 Series Chipset/Intel® C200 Series Chipset: Datasheet*, 2011. URL: http://www.intel.com/content/dam/doc/datasheet/6-chipset-c200-chipset-datasheet.pdf.

[8] *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3 (3A & 3B): System Programming Guide*, 2011. URL: http://www.intel.com/Assets/PDF/manual/325384.pdf.

[9] *Kernel Based Virtual Machine* , 2011. URL: http://www.linux-kvm.org/page/Main_Page.

[10] Lange, J.R. und P.A. Dinda: *An introduction to the Palacios Virtual Machine Monitor—release 1.0.* Northwestern University, Department of Electrical Engineering and Computer Science, Tech. Rep. NWU-EECS-08-11, 2008.

[11] Liebergeld, S.: *Lightweight Virtualization on Microkernel-based Systems.* Diplomarbeit, TU Dresden Chair for Operating Systems, 2010.

[12] Liedtke, J.: *Toward real microkernels.* Communications of the ACM, 39(9):70–77, 1996.

[13] Lindholm, Tim und Frank Yellin: *Java Virtual Machine Specification.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd Auflage, 1999. URL: http://java.sun.com/docs/books/jvms/second_edition/html/Introduction.doc.html.

[14] *Linux Kernel, Quellcode Version 2.6.36*, 2010. URL: http://kernel.org/.

[15] *englische Wikipedia: Operating system-level virtualization*, 2012. URL: https://en.wikipedia.org/wiki/Operating_system-level_virtualization?oldid=469973952.

[16] Peter, M., H. Schild, A. Lackorzyński und A. Warg: *Virtual Machines Jailed.* In: *Proceedings of the 1st EuroSys Workshop on Virtualization Technology for Dependable Systems*, Seiten 18–23. ACM, 2009.

[17] Popek, Gerald J. und Robert P. Goldberg: *Formal requirements for virtualizable third generation architectures.* In: *SOSP '73: Proceedings of the fourth ACM symposium on Operating system principles*, New York, NY, USA, 1974. ACM Press. URL: http://www-users.cselabs.umn.edu/classes/Spring-2010/csci5105/papers/popek-virt-reqmts.pdf.

[18] *QEMU Emulator User Documentation - 3.12 GDB Usage* , 2011. URL: http://qemu.weilnetz.de/qemu-doc.html#gdb_005fusage.

[19] Robin, John Scott und Cynthia E. Irvine: *Analysis of the Intel Pentium's ability to support a secure virtual machine monitor.* In: *Proceedings of the 9th conference on USENIX Security Symposium - Volume 9*, Seiten 10–10, Berkeley, CA, USA, 2000. USENIX Association. URL: http://dl.acm.org/citation.cfm?id=1251306.1251316.

[20] Russell, Rusty: *virtio: towards a de-facto standard for virtual I/O devices.* SIGOPS Oper. Syst. Rev., 42(5):95–103, Juli 2008. URL: http://dx.doi.org/10.1145/1400097.1400108.