INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Bachelorarbeit

Aufbau und Untersuchung eines Virtualisierungsclusters auf der Basis der ARMv7-Plattform

Niklas Kersten

INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Bachelorarbeit

Aufbau und Untersuchung eines Virtualisierungsclusters auf der Basis der ARMv7-Plattform

Niklas Kersten

Aufgabensteller: PD Dr. rer. nat. Vitalian Danciu
Betreuer: Tobias Guggemos

Abgabetermin: 2. November 2016

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 2. November 2016

. . . . . .

(Unterschrift des Kandidaten)

Kurzfassung

Plattformen der ARMv7-Architektur, bekannt für ihren niedrigen Energieverbrauch und den verbreiteten Einsatz in eingebetteten Systemen, erhielten kürzlich eine Befehlssatzerweiterung namens *ARM Virtualization Extensions*, welche Hardwarevirtualisierung ermöglicht. Durch diese Erweiterung sind ARM-Prozessoren nun auch für den Einsatz in Virtualisierungsclustern interessant geworden.

In dieser Arbeit wird ein Überblick über die Virtualisierung auf ARMv7 mit den ARM Virtualization Extensions gegeben und die Virtualisierer Xen und KVM werden näher betrachtet. Einige Unterschiede zu Hardwarevirtualisierung auf x86-Prozessoren werden herausgestellt. Es wird ein Virtualisierungscluster auf der Basis von ARMv7-Plattformen entworfen und installiert und die dabei gemachten Erfahrungen werden geschildert. Mit dem Virtualisierungscluster werden anschließend einige Leistungsmessungen vorgenommen, welche einen schwachen bis wettbewerbsfähigen Wirkungsgrad der Virtualisierung mit KVM/ARM bescheinigen, je nach gewähltem Szenario. Ebenso wird festgestellt, dass ARMv7-Plattformen eigene Herausforderungen stellen, die im Voraus nicht unbedingt offensichtlich waren. Ein erhöhter Konfigurationsaufwand bei heterogener Hardware sowie die Unmöglichkeit, hardwarenahe Programme wie Xen ARM universell auf unterschiedlichen auf ARMv7 basierenden Systemen zu installieren, sind nur zwei davon.

Abstract

ARM-SoCs based on the ARMv7 architecture, popular for their low power consumption and widespread use in embedded systems, recently gained support for hardware virtualization through an optional instruction set extension called *ARM Virtualization Extensions*. This extension makes ARM processors suitable for use in virtualization clusters.

In this thesis an overview over virtualization on ARMv7-SoCs is given with a closer look at the hypervisors Xen and KVM as well as the differences to hardware virtualization on the x86 architecture which dominates the virtualization market so far. A virtualization cluster based on ARMv7-SoCs is designed and built and the gathered experience is documented. Benchmarks are run on the virtualization cluster and show that virtualization performance of KVM/ARM ranges from poor to competitive, depending on the scenario. Many unexpected challenges and problems were discovered along the way, increased time spendings on administration of clusters using diverse hardware and the impossibility to have an universal installer for low-level software like Xen ARM for ARMv7-based systems are only few of them.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einleitung

Virtualisierung ist ein technisches Konzept, das immer weiter an Beliebtheit gewinnt. Die dadurch ermöglichte Zentralisierung und bessere Ausnutzung von Rechnerkapazitäten senkt nicht nur die Kosten für das Betreiben von Rechenzentren, sondern ermöglicht auch neue Anwendungen, die ohne Virtualisierung nicht möglich wären. Virtualisierung ist nicht zuletzt die Grundlage von Cloud-basierten Diensten [SWM 11], die in den letzten Jahren ein großes Wachstum erfahren haben.

Bisher wurden als Basis für Virtualisierungscluster hauptsächlich x86-Prozessoren verwendet. ARMv7-Plattformen haben im letzten Jahrzehnt durch die gestiegene Nachfrage nach mobilen Geräten wie Smartphones einen deutlichen Aufschwung erfahren, da sie sich für solche Geräte aufgrund ihres geringen Energiebedarfs besser als x86-Prozessoren eignen. Für die ARMv7-Architektur wurde vor einigen Jahren eine optionale Befehlssatzerweiterung eingeführt, die Hardwarevirtualisierung ermöglicht. Mit dieser Erweiterung soll Virtualisierung auf solchen Prozessoren einfach und effizient realisierbar sein. Durch die Kombination aus Hardwarevirtualisierung und geringem Energiebedarf erscheinen ARMv7-Prozessoren nun als Konkurrenten zu den etablierten x86-Prozessoren für den Einsatz in Virtualisierungsclustern.

Derzeit existierende wissenschaftliche Arbeiten geben nur wenig Einblick in die Leistungsfähigkeit und Praxistauglichkeit von auf ARMv7-Plattformen basierenden Virtualisierungsclustern. Diese Arbeit soll einen ersten Orientierungspunkt für einen auf ARMv7-Plattformen realisierten Virtualisierungscluster darstellen.

1.1. Motivation und Zielsetzung

Einer der wichtigsten Punkte bei Virtualisierung ist die Effizienz. Wäre Virtualisierung nur ineffizient realisierbar, wäre sie unattraktiv. Effizienz bezeichnet bei Virtualisierung einerseits den Wirkungsgrad der Virtualisierung, also, wie hoch die Verluste einer virtualisierten Ausführung im Vergleich zu einer nativen Ausführung von Software sind. Andererseits bezeichnet Effizienz aber auch die Energieeffizienz der Hardware. Besonders in Rechenzentren ist Energieeffizienz von Bedeutung, da in den letzten Jahren die über die Betriebszeit der Hardware aggregierten Kosten für Energie und Kühlung in Rechenzentren deutlich zugenommen haben und teilweise die Anschaffungskosten der Hardware übersteigen [RRT$^+$ 08]. Die Energieeffizienz hängt vor allem von der Hardwareplattform, die für die Virtualisierung eingesetzt wird, ab. ARMv7-Plattformen besitzen aufgrund ihres geringen Energiebedarfs und der geringen Wärmeabgabe bei eingebetteten Systemen wie beispielsweise Smartphones einen sehr hohen Marktanteil. Mit der Einführung einer optionalen Befehlssatzerweiterung für den ARMv7-Befehlssatz, welche Hardwarevirtualisierung ermöglicht, scheinen ARMv7-Plattformen nun auch als effiziente Basis für Virtualisierungscluster geeignet zu sein. Zudem ermöglicht die Virtualisierung auf ARMv7 neue Anwendungsszenarien. Es wäre beispielsweise denkbar, dass eine virtuelle Maschine von einem Arbeitsrechner im laufenden Betrieb auf ein mobiles Gerät verschoben wird, um unterwegs damit weiterarbeiten zu können.

1.2. Fragestellung

Ziel dieser Arbeit ist es, einen Einblick in die Praxistauglichkeit und Leistungsfähigkeit von Virtualisierungsclustern basierend auf ARMv7-Plattformen zu erhalten und aufbauend darauf einen ersten Vergleich zu den im Bereich Virtualisierung etablierten x86-Prozessoren ziehen zu können. Etwas konkreter bedeutet dies, dass untersucht werden soll, welche Probleme und Herausforderungen das Betreiben eines Virtualisierungsclusters auf ARMv7-Plattformen im Vergleich zu x86-Prozessoren mit sich bringt und ob die Virtualisierung ausreichend effizient ist, um für den Einsatz in der Praxis interessant zu sein.

1.3. Verwandte Arbeiten und Einordnung dieser Arbeit

Für das Konzept der Virtualisierung erscheinen schon seit Jahrzehnten wissenschaftliche Arbeiten [PoGo 74] [RoGa 05], die als Grundlage für die moderne Virtualisierung dienen. Mit der Einführung einer Befehlssatzerweiterung für x86-Prozessoren [AdAg 06], die Hardwarevirtualisierung ermöglicht, erfuhr Virtualisierung einen neuen Aufschwung. Mittlerweile existieren zahlreiche Arbeiten zu Virtualisierung auf x86, die teils unterschiedliche Aspekte, wie beispielsweise die Implementierung und die dabei entstandenen Herausforderungen und Lösungsansätze [BDF+ 03] [KKL+ 07] von Virtualisierern schildern.

Umfangreiche Arbeiten existieren auch zu dem Einfluss von Konfigurationsparametern auf den Wirkungsgrad der Virtualisierer VMware ESXi [Lemb 10], Xen [Gebh 10], MS Hyper-V [Roma 10] und OpenVZ/Virtuozzo [Tsio 10], deren Leistungstests als Vorlage für die in dieser Arbeit ursprünglich vorgesehenen Testszenarien dienten. In [Lind 10] werden aus den Ergebnissen von empirischen Leistungsmessungen Konzepte und Ansatzpunkte zur Erhöhung des Wirkungsgrads virtueller Infrastrukturen auf Basis von x86-Prozessoren abgeleitet. Die Ergebnisse der Leistungsmessungen jener Arbeit werden bei der Leistungsevaluierung teilweise für Vergleiche zwischen ARMv7-Plattformen und x86-Prozessoren verwendet.

Bis vor einigen Jahren war Virtualisierung auf ARM-Prozessoren nur in Software möglich. Aus diesem Zeitraum existieren wissenschaftliche Arbeiten, die Softwarevirtualisierer präsentieren und evaluieren [HSH+ 08] [OKKA 10], wobei deren Fokus auf eingebetteten Systemen und mobilen Geräten liegt. Die Einführung einer Befehlssatzerweiterung [VaHe 11], die auch auf Prozessoren mit der ARMv7-Architektur Hardwarevirtualisierung ermöglicht, bildet die Grundlage jüngerer wissenschaftlicher Arbeiten. So existiert zu den Virtualisierern Xen [StCa 12] und KVM [DaNi 13] [DaNi 14] Literatur, die die Implementierungen für ARM beschreiben. Erst kürzlich wurde mit [DLL+ 16] eine ausführliche Leistungsevaluierung der Virtualisierer Xen und KVM auf ARM im Vergleich zu Xen und KVM auf x86 auf Basis eines ARMv8-Prozessors veröffentlicht, die diesen Virtualisierern auf ARMv8 einen wettbewerbsfähigen Wirkungsgrad der Virtualisierung auf dieser Plattform bescheinigt.

Auch zu Clustern und Datenzentren auf Basis von ARM-Plattformen existieren einige Arbeiten [OPD+ 12] [PdOVN 12], die vor allem die Aspekte Energieverbrauch und -effizienz im Vergleich zu auf x86-Prozessoren basierenden Clustern untersuchen und deren Ergebnis ist, dass ARM-Plattformen bei wenig rechenaufwendigen Aufgaben einen Vorteil bezüglich der Energieeffizienz besitzen, dieser aber zunehmend schwindet, je rechenaufwendiger die Aufgabe ist. Darüber hinaus wurde in [DgFKL 11] untersucht, ob Virtualisierungscluster sich auch für Hochleistungsrechenaufgaben eignen. Es wird der Schluss gezogen, dass Virtualisierungscluster für diesen Einsatzzweck administrative Vorteile bieten können, aber die Eignung von der Art der wissenschaftlichen Applikationen abhängt, da die Leistung der Applikation durch die Virtualisierung und die damit einhergehende erschwerte Softwareoptimierung aufgrund der Hardwareabstraktion durch die Virtualisierung zu stark beeinträchtigt werden könnte.

Zu Virtualisierungsclustern, die auf ARM-Plattformen basieren, gibt es derzeit nur wenig wissenschaftliche Literatur. An diesem Punkt soll diese Arbeit ansetzen. Sie soll eine erste Einschätzung bezüglich der Leistungsfähigkeit und Praxistauglichkeit von Virtualisierungsclustern auf der Basis von ARMv7-Plattformen bieten. Sie muss aber unbedingt im Kontext ihrer Zeit gesehen werden, da sowohl Hardwarevirtualisierung auf ARMv7-Plattformen als auch der Einsatz von ARMv7-Plattformen außerhalb von eingebetteten Systemen derzeit noch jung sind und dementsprechend eine Weiterentwicklung in diesen Bereichen zu erwarten ist.

1.4. Vorgehensmodell

Um die Praxistauglichkeit und Leistungsfähigkeit von auf ARMv7-Plattformen basierenden Virtualisierungsclustern zu untersuchen, ist es zunächst notwendig, die Begriffe Virtualisierung, Virtualisierungscluster und ARMv7-Plattformen zu definieren. Darauf aufbauend kann anschließend untersucht werden, wie Hardwarevirtualisierung bei der ARMv7-Architektur implementiert ist. Dies dient unter anderem dazu, erste Anhaltspunkte bezüglich der Leistungsfähigkeit der Virtualisierung auf ARMv7-Plattformen zu schaffen. Besonders interessant sind hier etwaige Unterschiede zu der Implementierung der Hardwarevirtualisierung auf x86-Prozessoren

und die sich dadurch ergebenden Vor- und Nachteile von Virtualisierung auf ARMv7-Prozessoren im Vergleich zu x86-Prozessoren. Damit wäre jedoch nur der hardwareseitige Aspekt der Virtualisierung auf ARMv7-Prozessoren abgedeckt. Da auch die Software einen wesentlichen Teil zu den Einsatzmöglichkeiten von Virtualisierung beiträgt, ist es ebenfalls notwendig, für die ARMv7-Architektur verfügbare Virtualisierer zusammenzutragen und deren Funktionsumfang und Implementierung anhand einiger Beispiele zu untersuchen. Mit den so gebildeten Grundlagen kann dann damit begonnen werden, einen auf ARMv7-Plattformen basierenden Virtualisierungscluster zu entwerfen und zu realisieren, um weitere Erkenntnisse und Erfahrungen im praktischen Einsatz zu gewinnen. Auch hier ist interessant, inwieweit sich Installation und Konfiguration von auf der PC-Plattform basierenden Virtualisierungsclustern unterscheiden, denn derzeit dient diese als Maßstab. Mit dem gebildeten Virtualisierungscluster können anschließend Messungen durchgeführt werden, um die Leistungsfähigkeit eines auf ARMv7-Plattformen basierenden Virtualisierungsclusters zu ermitteln.

Das Vorgehensmodell ist in Abbildung 1.1 grafisch dargestellt.

Diesem Modell folgend werden in Kapitel 2 zunächst grundlegende Begriffe definiert beziehungsweise Definitionen dieser Begriffe zusammengetragen und anschließend eine knappe Einführung in die ARM-Prozessorarchitektur gegeben. In diesem Kapitel erfolgt ebenfalls eine Darstellung der Funktionsweise der Hardwarevirtualisierung der ARMv7-Architektur mit den ARM Virtualization Extensions und eine Vorstellung der Virtualisierer Xen und KVM sowie ihre Implementierungen auf ARMv7. In Kapitel 3 folgt dann eine Beschreibung des für die Arbeit erstellten Virtualisierungsclusters sowie eine Auflistung von bei der Installation und Betreibung beobachteten Problemen, Hürden und Herausforderungen. Die daraus abgeleiteten Instruktionen zur Installation von Xen und KVM auf einem Banana Pi befinden sich in Anhang A beziehungsweise B. Die mit dem realisierten Virtualisierungscluster durchgeführten Leistungsmessungen werden in Kapitel 4 dargestellt. In Kapitel 5 folgt abschließend eine Zusammenfassung und kritische Auseinandersetzung über die Praxistauglichkeit und Leistungsfähigkeit von Virtualisierungsclustern auf der Basis von ARMv7-Plattformen, sowie ein Ausblick auf mögliche fortführende wissenschaftliche Arbeiten.

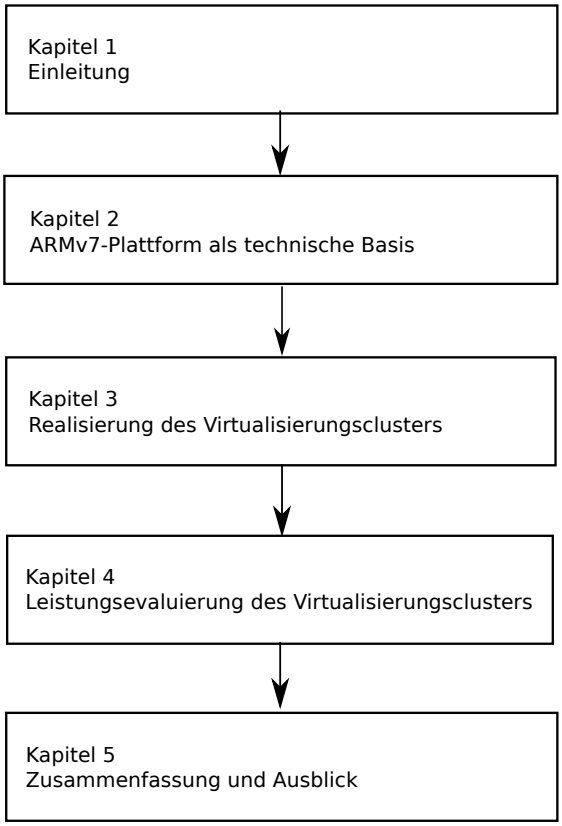

Abbildung 1.1.: Das Vorgehensmodell dieser Arbeit

2. ARMv7-Plattform als technische Basis

Der Fokus dieser Arbeit liegt auf der Evaluierung der Praxistauglichkeit und Leistungsfähigkeit eines Virtualisierungsclusters auf der Basis von ARMv7-Plattformen. Aus diesem Grund folgt in diesem Kapitel nach einer Begriffsklärung in Abschnitt 2.1 zunächst eine Einführung in die ARM-Architektur in Abschnitt 2.2. Einige Besonderheiten von ARMv7-Plattformen werden in Abschnitt 2.3 herausgestellt. Wie Hardwarevirtualisierung bei der ARMv7-Architektur funktioniert, wird dann in Abschnitt 2.4 erläutert. In Abschnitt 2.5 folgt ein Überblick über die für ARMv7-Prozessoren verfügbaren Virtualisierer, wobei die Implementierungen der Virtualisierer Xen und KVM anschließend in Abschnitt 2.5.1 beziehungsweise 2.5.2 näher betrachtet und einige Unterschiede der beiden Virtualisierer in 2.5.3 aufgeführt werden. Abschließend wird dieses Kapitel in Abschnitt 2.6 zusammengefasst.

2.1. Theoretische Grundlagen

In diesem Abschnitt werden die Begriffe Virtualisierung (Abschnitt 2.1.1), Cluster (Abschnitt 2.1.2) und Virtualisierungscluster (Abschnitt 2.1.3) erklärt oder definiert, da sie zum Verständnis der Arbeit wesentlich sind.

2.1.1. Virtualisierung

Für diese Arbeit ist lediglich die Virtualisierung von Rechnern interessant („Host-Virtualisierung“). Auch andere technische Ressourcen als Rechner können virtualisiert werden, beispielsweise könnten mehrere virtuelle Netze auf der Basis eines physischen Netzes gebildet werden. Da dies aber nicht Thema der Arbeit ist, wird anstatt des allgemeinen Begriffs der Virtualisierung nur der Begriff der Virtualisierung von Rechnern erläutert. Host-Virtualisierung ist eine Technik, die es ermöglicht, auf einem einzelnen physischen Rechner ein virtuelles Abbild eines Rechners, eine sogenannte Virtuelle Maschine (VM), zu betreiben. Es können mehrere virtuelle Maschinen parallel auf einem System betrieben werden. Diese virtuellen Maschinen können wie physische Rechner genutzt werden. Insbesondere kann ein Betriebssystem installiert und herkömmliche Anwendungen verwendet werden.

Der Einsatz von Virtualisierung bietet mehrere Vorteile. So sind die virtuellen Maschinen voneinander isoliert, sodass zum Beispiel ein Absturz oder das Installieren von Schadsoftware auf einer virtuellen Maschine in der Regel keinerlei Auswirkungen auf die anderen virtuellen Maschinen hat. Es ermöglicht auch eine bessere Ausnutzung der physischen Ressourcen, da virtuellen Maschinen basierend auf der aktuellen Systemauslastung und ihrem eigenen Ressourcenbedarf dynamisch Ressourcen zugeteilt werden können [Danc 09]. Beispielsweise könnte die physische Maschine einen Vierkernprozessor enthalten, aber eine virtuelle Maschine bekäme davon nur einen Prozessorkern zugewiesen, während die anderen drei Prozessorkerne für andere Aufgaben zur Verfügung stünden. Diese Zuteilung von Ressourcen und die generelle Überwachung und Verwaltung der virtuellen Maschinen geschieht durch einen *Virtual Machine Monitor*, auch als *Virtualisierer* oder *Hypervisor* bezeichnet. Dabei handelt es sich um ein Programm, das mit mehr Privilegien als die von ihm kontrollierten virtuellen Maschinen läuft und eine Abstraktionsschicht zwischen physischen Ressourcen und Betriebssystemen erzeugt [PoGo 74].

Gerald Popkek und Robert Goldberg legten in den 1970 Jahren folgende Kriterien für die Virtualisierbarkeit eines Rechners fest. Die grundlegende Annahme dabei war, dass ein Virtualisierer zur Verwaltung der virtuellen Maschinen eingesetzt wird [PoGo 74].

Effizienz Es entstehen nur geringe Leistungseinbußen im Vergleich zu einer *nativen*, also nicht virtualisierten, Ausführung.

Ressourcenkontrolle Der Virtualisierer besitzt die alleinige Kontrolle über die physische Maschine.

Äquivalenz Software wird in der virtuellen Maschine genauso ausgeführt wie auf der physischen Maschine. Einzige Ausnahmen bilden die Zeitverzerrung (Erklärung folgt in Abschnitt 4.1.4) und Verfügbarkeit von Ressourcen.

Zum Erreichen von Host-Virtualisierung stehen mehrere Methoden zur Verfügung [ÅFNK 11] [WRS 11]:

Binärübersetzung Kritischer Code in Gast-Betriebssystemen, d.h. Code, der gegen die Prinzipien einer Virtualisierung verstoßen würde, wird zur Laufzeit vom Virtualisierer durch Code mit gleicher Funktion ersetzt, der nicht mehr gegen die Prinzipien der Virtualisierung verstößt. Hauptsächlich ist damit Code gemeint, der das Kriterium der Ressourcenkontrolle verletzen könnte.

Paravirtualisierung Gast-Betriebssysteme werden so angepasst, dass sie mit dem Virtualisierer kooperieren. Dazu stellt der Virtualisierer über eine Schnittstelle eine abstrahierte Form der Hardware bereit. Gast-Betriebssysteme rufen den Virtualisierer bei Ressourcenanfragen auf.

Hardwarevirtualisierung Der Prozessor selbst bietet Funktionen an, um Virtualisierung zu realisieren.

Jede dieser Möglichkeiten besitzt Vor- und Nachteile. Binärübersetzung ist kompatibel mit jeder Hardware, jedoch häufig langsamer als die anderen Möglichkeiten, da die Binärübersetzung zusätzlichen Rechenaufwand bedeutet. Bei Paravirtualisierung ist es notwendig, das Gast-Betriebssystem anzupassen, was zum Beispiel bei proprietären Systemen wie Microsoft Windows nur von Hersteller und Lizenznehmern möglich ist. Hardwarevirtualisierung ist schnell und kompatibel mit unmodifizierten Gast-Betriebssystemen, benötigt aber spezielle Hardware und ist deswegen weniger flexibel [WRS 11].

Zusätzlich lässt sich zwischen zwei Arten von Virtualisierern unterscheiden [FMMG 08].

nativ (Typ 1) Der Virtualisierer läuft als unterste Schicht direkt auf der Hardware und verwaltet Systemressourcen und Gast-Betriebssysteme.

gehostet (Typ 2) Der Virtualisierer läuft innerhalb eines Host-Betriebssystems und verwaltet lediglich die Gast-Betriebssysteme, während das Host-Betriebssystem die Systemressourcen verwaltet.

2.1.2. Cluster

Ein *Cluster* ist ein Verbund von mehreren Rechnersystemen, die in diesem Kontext auch als Knoten bezeichnet werden. So entsteht ein logisch zusammenhängendes System, das von außen in vielerlei Hinsicht als ein einzelnes physisches System betrachtet werden kann. Clustering wird unter anderem genutzt, um die Rechenleistung gegenüber einer einzelnen physischen Maschine zu erhöhen oder die Ausfallsicherheit zu verbessern [BaBu 99].

2.1.3. Virtualisierungscluster

Für diese Arbeit wird der Begriff *Virtualisierungscluster* als ein Verbund von Rechnersystemen, auf denen jeweils Virtualisierer und möglicherweise eine Mehrzahl an virtuellen Maschinen laufen, definiert. Zusätzlich muss es möglich sein, laufende virtuelle Maschinen zwischen einzelnen Clusterknoten zu verschieben, ohne dass die virtuelle Maschine ihren Betrieb einstellen muss. Dies wird auch als *Live-Migration* bezeichnet.

Die Vorteile eines Virtualisierungsclusters sind vielfältig. Zum einen wird so der administrative Aufwand zentriert, was Skaleneffekte bezüglich Kühlung, Personal und Wartung hevorruft. Zum anderen ist es möglich, auf variierende Lasten dynamisch zu reagieren, da laufende virtuelle Maschinen innerhalb des Rechnerverbundes ohne große Ausfallzeit transferiert werden können [SKM 08] und so beispielsweise eine virtuelle Maschine von einem überlasteten Knoten auf einen weniger belasteten Knoten transferiert werden kann, sodass die physischen Ressourcen besser ausgenutzt werden.

2.2. ARM-Architektur - Einführung und Begriffsabgrenzung

Der Begriff Advanced RISC Machine (ARM) fasst mehrere Generationen einer Prozessorarchitektur zusammen. Die ARM-Architektur wird hauptsächlich von der britischen Firma *ARM Holdings plc* entwickelt, die die Architektur an andere Unternehmen lizensiert, welche diese gegebenenfalls modifizieren sowie auf der Architektur basierende Prozessoren produzieren und verkaufen [Furb 00] [Ryzh 06].

Als Grundlage der ARM-Architektur dient das RISC-Prinzip. Reduced Instruction Set Computing (RISC) bedeutet, dass der Prozessor nur die wichtigsten und am häufigsten gebrauchten Befehle in Hardware ausführt und weniger häufig benutzte Funktionen durch Verkettung von Maschinenbefehlen emuliert. Im Vergleich zu Complex Instruction Set Computing (CISC) impliziert RISC einen geringeren Flächenbedarf, eine einfachere Architektur und einen geringeren Stromverbrauch sowie eine schnelle Ausführung der in Hardware implementierten Befehle [Furb 00]. Nachteilig ist, dass emulierte Befehle mehr Speicherplatz benötigen, da sie vom Compiler in Ketten von in der Hardware verfügbaren Befehlen zwecks Emulation übersetzt werden müssen [Che 00].

Zu unterscheiden sind allerdings auch Befehlssatz und Architektur. Der Befehlssatz fasst alle verfügbaren Maschinenbefehle eines Prozessors zusammen. Er wird von Programmierern oder Compilern verwendet, um ein Programm für den Prozessor zu schreiben beziehungsweise zu kompilieren. Eine Architektur ist eine Implementierung eines Befehlssatzes in Hardware. Sie beschreibt also den Aufbau eines Prozessors. Die Unterscheidung zwischen Befehlssatz und Architektur ist insofern interessant, als dass ursprünglich bei x86-Prozessoren sowohl der Befehlssatz als auch die Architektur nach dem CISC Prinzip entworfen wurden. Moderne x86-Prozessoren implementieren aus Kompatibilitätsgründen immer noch einen CISC Befehlssatz, die Architektur hingegen orientiert sich mittlerweile am RISC Prinzip. Einige Befehle aus dem CISC Befehlssatz werden von modernen x86-Prozessoren lediglich durch Verkettung anderer Befehle emuliert [IJJ 09]. Blem et al. stellten fest, dass der nach außen dargestellte Befehlssatz bei modernen Prozessoren ab einer gewissen Rohleistung keine nennenswerten Auswirkungen auf Leistung und Energiebedarf hat [BMS 13]. Dies ist eine wichtige Feststellung, denn bis dahin wurden Prozessoren mit einem CISC Befehlssatz, wie beispielsweise x86-Prozessoren, im Vergleich zu Prozessoren mit RISC-Befehlssatz als energieineffizient bezeichnet.

x86-Prozessoren dominieren hauptsächlich im PC- und Servermarkt, während eingebettete Systeme wie Router, Set Top Boxen, Smartphones und Tablets der klassische Einsatzort von ARM-Prozessoren sind. In den letzten Jahren fanden ARM-Prozessoren aber auch außerhalb von eingebetteten Systemen Verbreitung. In Entwicklungsländern sind sie beispielsweise aufgrund des geringen Anschaffungspreises und geringen Stromverbrauchs ein beliebter PC-Ersatz. Auch hierzulande werden sie mittlerweile in Googles Chromebook [goo 16] als PC-Ersatz präsentiert. Auch Advanced Micro Devices (AMD), ein Unternehmen, das unter anderem Prozessoren entwickelt und bei diesen hauptsächlich für seine x86-Prozessoren bekannt ist, wird nun Prozessoren auf Basis der ARM-Architektur anbieten [amd 16].

Die Begriffe ARM-Architektur, ARM-Prozessor, ARM-Plattform und PC-Plattform können leicht zu Missverständnissen führen und werden daher im Folgenden für diese Arbeit definiert.

Der Begriff *ARM-Architektur* ist ein übergeordneter Begriff, der mehrere Generationen einer Architektur für Prozessoren zusammenfasst. Die Generationen werden durch ein angehängtes „v“ und eine Generationsnummer unterschieden. Beispielsweise werden in dieser Arbeit Prozessoren betrachtet, die auf der ARMv7-Architektur basieren.

Zu unterscheiden sind außerdem die Begriffe *ARM-Prozessor* und *ARM-Plattform*. Ein ARM-Prozessor ist ein Prozessor, der auf der ARM-Architektur basiert. Der Begriff ARM-Plattform bezeichnet einen ARM-Prozessor, der zusammen mit Hauptspeicher und gegebenenfalls weiterer Hardware auf einem einzigen Chip zusammengefasst ist. Diese Chips werden auch als System-on-Chip (SoC) bezeichnet.

Die *PC-Plattform* hingegen bezeichnet einen kompletten Rechner oder genauer eine standardisierte Plattform, die Prozessoren auf Basis der x86-Architektur einsetzt. Die PC-Plattform ist modular aufgebaut und kein System-on-Chip. Als Basis dient die Hauptplatine, auf der verschiedene Sockel, Anschlüsse und Einschübe zur Verfügung stehen. In beziehungsweise an diese werden dann Prozessoren, Hauptspeicher, Erweiterungskarten und Peripherie eingesetzt oder angeschlossen.

2.3. ARMv7-Spezifika

Durch die ursprüngliche Ausrichtung auf eingebettete Systeme gibt es bei Systemen, die ARMv7-Plattformen einsetzen, einige Unterschiede zu der PC-Plattform. Da das ARM-Ökosystem sehr vielfältig ist, also sehr viel periphere Hardware existiert, wurde zur Erkennung solcher Hardware der *Gerätebaum* eingeführt. Dieser wird in Abschnitt 2.3.1 näher behandelt.

2.3.1. Gerätebaum zur Beschreibung peripherer Hardware

ARM-Plattformen waren ursprünglich für den Einsatz in eingebetteten Systemen gedacht. Diese Systeme besitzen häufig eine Hardwarekonfiguration, die vom Hersteller im Voraus festgelegt wird und während ihrer Lebenszeit nicht mehr geändert wird. Die Software, die auf eingebetteten Systemen eingesetzt wird, ist meistens speziell an ein bestimmtes System angepasst. Aus diesen Gründen war es bisher nicht notwendig, die Kommunikation mit an die ARM-Plattformen angeschlossener peripherer Hardware zu standardisieren. Dies führte dazu, dass die Hardware, die mit ARM-Plattformen kombiniert wird, sehr vielfältig ist und viele Hersteller eigene Protokolle zur Kommunikation mit peripherer Hardware entwickelt haben. Die fehlende Standardisierung erschwert jedoch das Erkennen der angeschlossenen Hardware.

Zur Lösung dieses Problems wurde das Konzept des Gerätebaums entwickelt. Der Gerätebaum ist eine baumförmige Datenstruktur, die beinhaltet, welche Hardware auf einer gegebenen Plattform vorhanden ist und wie mit ihr zu kommunizieren ist. Der Gerätebaum wird in einer Datei abgespeichert, die vom *Bootloader*, einem Programm zum Starten eines Betriebssystems, geladen und an das Betriebssystem übergeben wird. Dieses kann anhand der Datei erkennen, welche Hardware vorhanden ist und welche Treiber geladen werden müssen. Die Erstellung des Gerätebaums erfolgt normalerweise von menschlicher Hand und ist plattformspezifisch. Es ist zu unterscheiden zwischen der Quelldatei des Gerätebaums (Dateiendung *.dts*) und der kompilierten Fassung des Gerätebaums (Dateiendung *.dtb*). Die Quelldatei ist zur Bearbeitung von Menschen gedacht und entsprechend aufgebaut. Die Quelldatei kann mit dem Compiler *dtc* in eine Binärdatei übersetzt werden. Das Betriebssystem arbeitet ausschließlich mit der Binärdatei [LiBo 08].

Für viele Geräte, wie beispielsweise auch den hier verwendeten Banana Pi, sind im Quelltext des Linux-Kerns vorkonfigurierte Gerätebäume vorhanden. Im folgenden Listing ist ein Auszug aus der Quelldatei des Gerätebaums für einen Banana Pi dargestellt. Der Auszug zeigt die Beschreibung des SD-Karten-Lesers beschrieben.

Listing 2.1: Beschreibung des SD-Karten-Lesers im Gerätebaum eines Banana Pis

```
&mmc0 {
  pinctrl-names = "default";
  pinctrl-0 = <&mmc0_pins_a>, <&mmc0_cd_pin_bananapi>;
  vmmc-supply = <&reg_vcc3v3>;
  bus-width = <4>;
  cd-gpios = <&pio 7 10 GPIO_ACTIVE_HIGH>; /* PH10 */
  cd-inverted;
  status = "okay";
};
```

2.3.2. Unterschiede beim Bootvorgang

Auch der Bootvorgang unterscheidet sich bei ARM-Plattformen grundlegend. Während bei PC-Plattformen das BIOS für das Starten eines Betriebssystems zuständig ist, wird diese Aufgabe bei ARM-Plattformen von einem Bootloader übernommen. Der Bootvorgang ist bei ARM-Plattformen nicht standardisiert und von der verwendeten Hardware abhängig. Im Folgenden wird als Beispiel der Bootvorgang der Banana Pis beschrieben, die für den in Kapitel 3 erstellten Virtualisierungscluster verwendet wurden. Der Allwinner A20 SoC der Banana Pis greift direkt nach der Zuführung von Strom auf den achten Block (Blockgröße = 1KB) der SD-Karte zu [lin 16]. Dort wird ein Bootloader-Programm in binärer Form erwartet. In der Regel ist dies *Das*

U-Boot Secondary Program Loader (SPL), ein minimaler Bootloader. In diesem Fall lädt *Das U-Boot SPL* eine vollständige Version von *Das U-Boot*, die am 40. Block beginnt. Diese vollständige Version kann dann ein Betriebssystem von gängigen Dateisystemen starten. Ein vom Nutzer zu erstellendes Skript teilt *Das U-Boot* mit, welches Betriebssystem wo auf der SD-Karte liegt, an welche Position es in den Hauptspeicher geladen werden soll und mit welchen Parametern es gestartet werden soll.

2.4. Hardwarevirtualisierung auf ARMv7-Prozessoren

Wie bei x86 ist die ARMv7-Architektur nicht klassisch virtualisierbar. Aus diesem Grund wurde für die ARMv7-Architektur eine optionale Erweiterung eingeführt, die Virtualisierung in Hardware ermöglicht. Diese wird als ARM Virtualization Extensions (ARM VE) bezeichnet [VaHe 11].

Bei x86-Prozessoren stehen 4 Privilegienniveaus zur Verfügung. Privilegienniveaus dienen der Isolation von Prozessen und ermöglichen dem Betriebssystem beispielsweise, die alleinige Kontrolle über die Hardware zu behalten. Im Falle von Hardwarevirtualisierung wird bei x86-Prozessoren zusätzlich zwischen *root* und *non-root* Modus unterschieden, um dem Virtualisierer noch höhere Privilegien als einem in Ring 0 laufenden Betriebssystem einzuräumen [UNR+ 05], wie in Abbildung 2.1 dargestellt. Bei ARM VE hingegen stehen nur 3 Privilegienniveaus zur Verfügung, welche bei ARMv7 als *Privilege Level* bezeichnet werden [arm 14]:

PL0 (Nutzermodus) In diesem Modus laufen unprivilegierte Befehle und somit der Großteil der Anwendungen.

PL1 (Kernmodus) In diesem Modus laufen privilegierte Befehle und das Betriebssystem.

PL2 (Modus für Virtualisierer) In diesem Modus läuft der Virtualisierer.

Während bei x86 mit Hardwarevirtualisierung die root bzw. non-root Modi orthogonal zu den existierenden Privilegienniveaus arbeiten, wurde bei ARMv7 lediglich ein weiteres Privilegienniveau für Virtualisierer hinzugefügt. Die Privilegienniveaus von ARMv7 sind in Abbildung 2.2 dargestellt.

2.5. Für ARMv7 verfügbare Virtualisierer

Derzeit sind bereits mehrere Virtualisierer für die ARMv7-Architektur verfügbar, welche die ARM Virtualization Extensions nutzen. Dazu gehören unter anderem:

- Xen ab Version 4.3 [xen 13]
- KVM ab Linux-Kern Version 3.9 [kvm 13]
- Jailhouse ab Version 0.5 [jai 16]
- INTEGRITY Multivisor [int 16]
- OKL 4 Microvisor [okl 16]

Xen, KVM und Jailhouse sind freie Software und kostenfrei zugänglich. Für diese Arbeit wurden Xen und KVM aufgrund der kostenlosen Verfügbarkeit für die praktische Untersuchung ausgewählt. Jailhouse wurde nicht mit einbezogen, da hier der Fokus auf eingebettete Systeme mit Echtzeitanforderungen liegt und diese für Virtualisierungscluster normalerweise irrelevant sind.

Sowohl KVM als auch Xen werden im Kontext von ARM häufig als „KVM/ARM“ bzw. „Xen ARM“ geschrieben, um die zugrundeliegende Hardwareplattform hervorzuheben und eine Abgrenzung gegenüber den x86-Varianten zu ermöglichen, die historisch bedingt oft einfach als „KVM“ bzw. „Xen“ bezeichnet werden. In dieser Arbeit wird die Architektur der Virtualisierer zur besseren Unterscheidbarkeit explizit erwähnt, sofern nicht von KVM und Xen im Allgemeinen gesprochen wird.

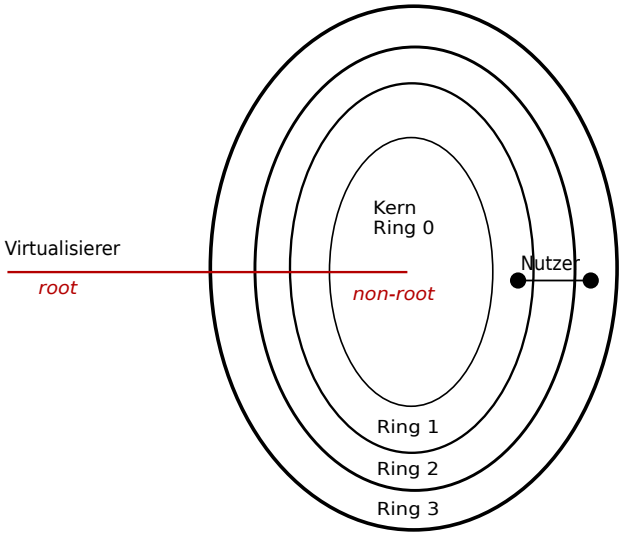

Abbildung 2.1.: Privilegienniveaus bei modernen x86-Prozessoren mit Hardwarevirtualisierung

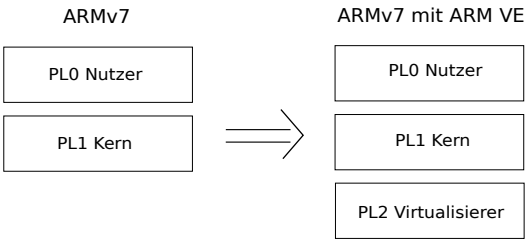

Abbildung 2.2.: Privilegienniveaus bei Prozessoren der ARMv7-Architektur ohne und mit ARM Virtualization Extensions

2.5.1. Xen auf ARM

Xen ARM ist ein nativer (Typ 1, Definition in Abschnitt 2.1.1) Virtualisierer. Die Entwickler von Xen ARM haben die Unterschiede in der Hardwarevirtualisierung zwischen ARMv7 und x86 zum Anlass genommen, den x86-Code von Xen nicht einfach auf ARMv7 zu übertragen, sondern von Grund auf neu zu strukturieren und zu säubern. Dies führte dazu, dass die Codebasis von Xen ARM lediglich ein sechstel so groß ist wie die von Xen x86_64 [Stab 16]. Das bedeutet, dass die Codebasis weniger Angriffsfläche für Sicherheitslücken bietet und möglicherweise einfacher zu warten ist.

Seit Xen ARM Version 4.4 gilt das Application Binary Interface (ABI), eine Schnittstelle, welche von Programmen zur Laufzeit genutzt werden kann, um Funktionen anderer Programme und Bibliotheken aufzurufen, als stabil. Alle folgenden Versionen von Xen ARM sollen abwärtskompatibel zu dieser ABI sein. Sie ist ab Linux-Kern Version 3.9rc1 implementiert, jedoch ist die Implementierung erst ab Linux-Kern Version 3.13-rc5 fehlerfrei [Xen 16a].

Xen ARM bietet im Gegensatz zu Xen x86, wo verschiedene Methoden zur Virtualisierung zur Verfügung stehen, nur eine Methode an, nämlich *hardwarebeschleunigte Paravirtualisierung*, vergleichbar mit PVHVM von Xen x86. Bei hardwarebeschleunigter Paravirtualisierung werden Prozessor und Hauptspeicher mithilfe der ARM Virtualization Extensions hardwarevirtualisiert und Ein-/Ausgabeanfragen auf Peripherie paravirtualisiert. Die Existenz von hardwarebeschleunigter Paravirtualisierung als einzige von Xen ARM unterstützte Form der Virtualisierung impliziert, dass Xen ARM keine QEMU-gestützte Emulation und auch keine reine Hardwarevirtualisierung anbietet [Stab 16]. Im Vergleich dazu bietet Xen x86 beispielsweise auch reine Hardwarevirtualisierung an, sodass es auf dieser Architektur möglich ist, unmodifizierte Betriebssysteme auf einer virtuellen Maschinen laufen zu lassen [Weiß 09].

Von den oben genannten Unterschieden abgesehen, arbeitet Xen ARM ähnlich wie Xen x86. Als Mikrokern verwaltet und virtualisiert Xen ARM Prozessoren, Hauptspeicher, Zeitgeber und Unterbrechungen. Die Gerätetreiber für die restliche Hardware (Sekundärspeicher etc.) und Werkzeuge („Xen Toolstack“) zur Verwaltung der virtuellen Maschinen werden von einem Gast-Betriebssystem, der sogenannten *Domäne 0* (abgekürzt „dom0“), das über höhere Privilegien als alle anderen Gast-Betriebssysteme verfügt, bereitgestellt. Die restlichen Gast-Betriebssysteme greifen mittels paravirtualisierter Gerätetreiber über die von dom0 angebotene Schnittstelle auf eine abstrahierte Darstellung der Hardware zu. Die virtuellen Maschinen, auf denen diese Gast-Betriebssysteme laufen, werden bei Xen ARM als „domU“(„U “ für „unprivilegiert“) bezeichnet [BDF+ 03]. Der Aufbau von Xen ARM ist in Abbildung 2.3 dargestellt. Xen läuft bei ARMv7 komplett auf dem Privilegienniveau PL2, sodass PL1 für Gast-Betriebssysteme und PL0 für Gast-Nutzerprogramme zur Verfügung stehen. Durch diese klare Trennung laufen Gast-Betriebssystem und Anwendersoftware größtenteils so, wie sie es ohne Virtualisierung tun würden. Dadurch wird die Anzahl der kostspieligen Kontextwechsel, also Wechseln des Prozessors zwischen unterschiedlichen Privilegienniveaus, gering gehalten. Xen ARM profitiert zwar von der geringeren Anzahl an Kontextwechseln, sodass das Wechseln von VM-Kontexten im Vergleich zu Xen x86 nur ein drittel an Prozessorzyklen benötigt, aber bei praktischen Anwendungen ist derzeit kein Leistungsvorteil nachweisbar [DLL+ 16].

Beim Booten erstellt Xen ARM einen Gerätebaum, der an dom0 übergeben wird. Der Gerätebaum wurde bereits in Abschnitt 2.3.1 behandelt. Der von Xen ARM erstellte Gerätebaum unterscheidet sich von einem herkömmlichen Gerätebaum insofern, als dass er nur die Hardware enthält, die nicht von Xen ARM selbst verwaltet wird und für die das in dom0 laufende Betriebssystem die Treiber bereitstellen muss. Im von Xen ARM erstellten Gerätebaum befindet sich auch ein Hinweis, dass die Plattform durch Xen ARM virtualisiert wird. So kann das in dom0 laufende Betriebssystem erkennen, dass es virtualisiert wird und entsprechend Treiber zur Kommunikation mit Xen ARM sowie die Schnittstelle für paravirtualisierte Treiber für andere Gast-Betriebssysteme initialisieren. Das in dom0 laufende Betriebssystem wird nicht versuchen, auf nicht existente Hardware zuzugreifen, was es erlaubt, auf Emulation vollständig zu verzichten [Stab 16].

2.5.2. KVM auf ARM

Kernel Virtual Machine (KVM) bezeichnet einen Typ 2 Virtualisierer, der in den Linux-Kern integriert ist. KVM wird zusammen mit *QEMU* eingesetzt, ein Programm, dass sowohl Hardwareplattformen emulieren

kann, als auch als *Frontend* (ein Programm, über das der Nutzer mit einem anderen Programm interagieren kann) für KVM dient. KVM und QEMU wurden als möglichst hardwareunabhängig entworfen. KVM-Varianten existieren außer für die x86-Architektur und den Generationen sieben und acht der ARM-Architektur auch für die PowerPC- und IA64-Architekturen. Diese sind, genau wie die ARM-Varianten, als „in Arbeit befindlich“ markiert. Über verschiedene Prozessorarchitekturen hinweg teilen sich diese Varianten einen großen Teil des Quelltexts von KVM bzw. QEMU [DaNi 13].

Die Virtualization Extensions für die ARMv7-Architektur erlauben es nicht, ein Betriebssystem ohne tiefgreifende Änderungen am Quelltext in PL2 laufen zu lassen, da ein Großteil der in PL1 verfügbaren Befehle nicht in PL2 verfügbar ist. Aus diesem Grund laufen Host-Betriebssystem und KVM/ARM selbst in PL1. Wie in Abbildung 2.4 dargestellt, ruft KVM/ARM von PL1 aus spezielle, in PL2 laufende, Funktionen auf, um virtuelle Maschinen zu verwalten.

Dieses Aufteilen des Virtualisierers auf zwei Ebenen ist neu und wird von den KVM/ARM-Entwicklern als *Split-mode Virtualization* bezeichnet. Der in PL1 laufende Teil von KVM/ARM wird als *Highvisor* bezeichnet, während der in PL2 laufende Teil als *Lowvisor* bezeichnet wird [DaNi 14]. Durch die Aufteilung auf zwei Privilegienniveaus sind zahlreiche Kontextwechsel zum Wechseln von VM-Kontexten nötig, sodass beispielsweise das Wechseln von einer virtuellen Maschine zu KVM bereits um den Faktor 20 langsamer als bei Xen ARM ist. Lediglich bei Eingabe-/Ausgabeanfragen benötigt Xen ARM aufgrund seiner Architektur mehr Kontextwechsel. Dall identifizierte dies als Ursache dafür, dass KVM/ARM in einer Mehrzahl an Anwendungen aus der Praxis einen höheren Wirkungsgrad als Xen ARM besitzt [DLL+ 16].

Für die 64-Bit Erweiterung der ARMv8.1-Architektur wurden die Virtualization Extensions dahingehend erweitert, dass ein Betriebssystem nun mit minimalen Änderungen in PL2 laufen kann. Es wurden auch zahlreiche andere Aspekte speziell im Hinblick auf Typ 2 Virtualisierer verbessert, wovon auch KVM in Form von einer Leistungssteigerung profitieren könnte [DLL+ 16] [Zyng 15]. Da der Fokus dieser Arbeit auf ARMv7-Plattformen liegt und auch keine entsprechende Hardware zur Verfügung stand, konnte dies nicht überprüft werden.

2.5.3. Unterschiede zwischen Xen und KVM auf ARM

Das Installieren von Xen ist auf einer ARMv7-Plattform sehr aufwendig, KVM/ARM hingegen kann mit geringerem Aufwand installiert werden. So betrug der Zeitaufwand für das Installieren und Konfigurieren des in Kapitel 3 erstellten Virtualisierungsclusters mit Xen ARM aufgrund einiger Softwarefehler und der Notwendigkeit, mehr Software selbst zu kompilieren und zu konfigurieren etwa zwei Monate, während der Virtualisierungscluster mit KVM/ARM nach vier Tagen fertig installiert und konfiguriert war (vgl. dazu auch den größeren Umfang der Installationsinstruktionen für Xen ARM in Anhang A mit denen von KVM/ARM in Anhang B). KVM/ARM besitzt jedoch den Nachteil, dass aufgrund der nicht standardisierten Hardwareschnittstellen und dem deswegen notwendigen Gerätebaum in einem heterogenen Cluster eine aus mehreren von QEMU angebotenen Hardwareplattformen als Basis gewählt werden muss. Unterscheidet sich die gewählte Beschreibung der Hardwareplattform von der physischen Hardware, so ist es möglich, dass einige Anfragen von virtuellen Maschinen durch QEMU emuliert werden müssen, was sich negativ auf die Leistung auswirken kann. Dieses Problem existiert bei Xen ARM nicht, da Xen ARM keine Emulation implementiert.

2.6. Zusammenfassung

Sowohl proprietäre als auch freie Virtualisierer bieten Unterstützung für die ARMv7-Architektur und nutzen die ARM Virtualization Extensions. Die ARM Virtualization Extensions unterscheiden sich von der Hardwarevirtualisierung der x86-Architektur insofern, als dass sie ein drittes Privilegienniveau einführen, während die Hardwarevirtualisierung der x86-Architektur einen zu den bestehenden Privilegienniveaus orthogonalen Modus für Virtualisierer nutzt. Die in Abschnitt 2.5.1 beziehungsweise 2.5.2 behandelten Implementierungen von Xen ARM und KVM/ARM zeigen auf, dass die ARM Virtualization Extensions für die ARMv7-Architektur besser für native Virtualisierer wie Xen geeignet sind. Während Xen ARM komplett im für Virtualisierer gedachten Privilegienniveau PL2 läuft, läuft KVM/ARM mit Linux als Host-Betriebssystem gezwungenermaßen sowohl in PL1 als auch PL2, da ein Host-Betriebssystem nicht ohne tiefgreifende Änderungen in

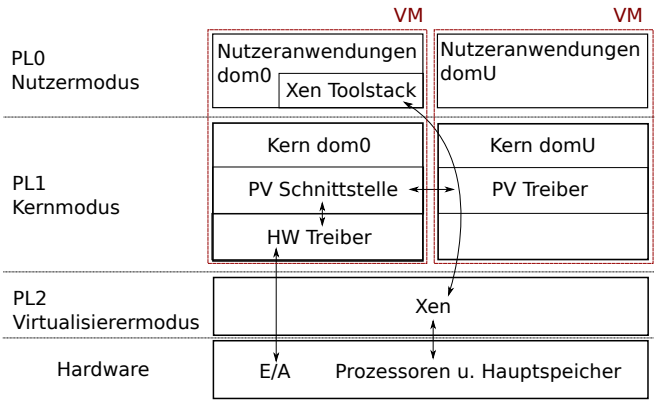

Abbildung 2.3.: Architektur von Xen ARM

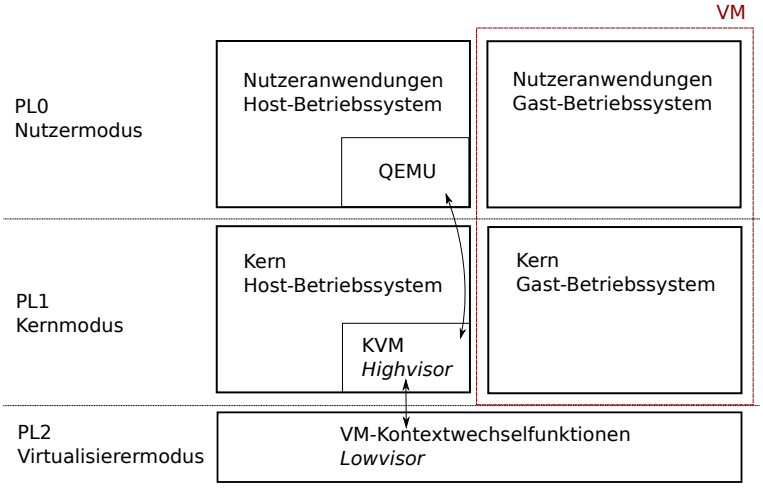

Abbildung 2.4.: Architektur von KVM/ARM

PL2 laufen kann. Dadurch entscheidet sich die Implementierung von KVM/ARM deutlich von KVM/x86, da bei KVM/x86 keine Aufteilung auf zwei unterschiedliche Privilegienniveaus vorgenommen wurde. Mit der ARMv8.1-Architektur wird dieser Unterschied allerdings zukünftig voraussichtlich aufgehoben.

Nachdem in diesem Kapitel die für Virtualisierung auf ARMv7-Plattformen erforderliche Hardware und Systemsoftware vorgestellt wurden, wird im folgenden Kapitel 3 als nächster Schritt ein Virtualisierungscluster auf der Basis von ARMv7-Plattformen entworfen und aufgebaut.

3. Realisierung des Virtualisierungsclusters

Nachdem in Kapitel 2 grundlegende Begriffe erläutert beziehungsweise definiert wurden und anschließend ein Einblick in die Hardwarevirtualisierung auf ARMv7 mittels ARM Virtualization Extensions sowie die Implementierungen von Xen ARM und KVM/ARM gegeben wurde, soll nun ein prinzipieller Aufbau eines Virtualisierungsclusters im Labormaßstab entworfen und realisiert werden. In Abschnitt 3.1 wird der Entwurf des Virtualisierungsclusters vorgestellt. Aufgrund einiger im praktischen Einsatz erkannten Probleme musste dieser später abgeändert werden. Die unternommenen Versuche und erforderlichen Änderungen sowie das Ergebnis werden in Abschnitt 3.2 präsentiert. Abschnitt 3.3 enthält Erläuterungen von Problemen und Herausforderungen, die während der Arbeit mit dem Virtualisierungscluster ersichtlich geworden sind. Eine Zusammenfassung dieses Kapitels befindet sich in Abschnitt 3.4.

3.1. Entwurf des Virtualisierungsclusters

In diesem Abschnitt wird der Entwurf des Virtualisierungsclusters präsentiert. Während die Hardwarekonfiguration beibehalten werden konnte, musste die Softwarekonfiguration aufgrund einiger Probleme im Nachhinein geändert werden. Die Änderungen und die Gründe, aus denen diese vorgenommen werden mussten, werden in Abschnitt 3.2 ausgeführt.

Der Virtualisierungscluster sollte aus drei ARM-Einplatinenrechnern, die via Ethernet miteinander verbunden sind, bestehen. Einer der Knoten sollte dabei als zentraler Knoten dienen und einen Switch für die anderen Knoten bereitstellen. Alle anderen Knoten sollten über den Switch dieses Zentralknotens miteinander kommunizieren können. Für die Knoten sollten ARM-Einplatinenrechner verwendet werden, da diese günstig und aufgrund ihres Formfaktors portabel sind. Die ARM-Einplatinenrechner mussten dem Ziel der Arbeit entsprechend unbedingt über Plattformen der ARMv7-Architektur mit den ARM Virtualization Extensions verfügen. Am Lehrstuhl für Kommunikationssysteme und Systemprogrammierung standen Einplatinenrechner der Banana Pi Produktreihe zur Verfügung. Diese sind weder besonders leistungsfähig, noch sind sie originär für den Einsatz in einem Virtualisierungscluster gedacht. Da sie aber über die ARM Virtualization Extensions verfügen und für Laborversuche als geeignet erschienen, wurden sie als Basis für den Virtualisierungscluster ausgewählt.

3.1.1. Hardwarekonfiguration

Im Folgenden wird der Entwurf der Hardwarekonfiguration des Virtualisierungsclusters beschrieben. Für die finale Version des Virtualisierungsclusters konnte die Hardwarekonfiguration des Entwurfs uneingeschränkt übernommen werden.

Knoten 1 - Banana Pi R1 - zentraler Knoten

Der Banana Pi R1 wurde als zentraler Knoten gewählt, da er über fünf externe Ethernetanschlüsse sowie einen SATA-Anschluss für Datenträger verfügt. An diesem SATA-Anschluss soll eine SSD mit 128 Gigabyte (GB) Speicherkapazität betrieben werden, welche im Netz über das Network File System (NFS)-Protokoll einhängbar ist. Auf der SSD sollen sämtliche Abbilder der virtuellen Maschinen abgelegt werden. Startet also ein beliebiger Knoten eine virtuelle Maschine, so soll er über das NFS-Protokoll auf ein auf der SSD gespeichertes Abbild einer virtuellen Maschine zugreifen. Die Vernetzung der Komponenten untereinander ist in Abbildung 3.1 dargestellt. Tabelle 3.1 zeigt die technischen Daten des Banana Pi R1.

Tabelle 3.1.: Technische Daten des Banana Pi R1 [bpi 14b]

SoC	Allwinner A20
Prozessorarchitektur	ARMv7
Hardwarevirtualisierung	Ja, ARM Virtualization Extensions
64-bit fähig	Nein
Hauptspeicher	1 GB DDR3
Stromverbrauch	weniger als 10 Watt
Ethernet-Schnittstelle	6-Wege Broadcom BCM53125 Switch, 1 GBit/s
Anschlüsse für Speichermedien	SD-Karten-Leser, USB 2.0, SATA
Eingesetzte Speichermedien	32 GB SD-Karte (Betriebssystem), 128 GB SSD (VM-Abbilder)

Tabelle 3.2.: Technische Daten des Banana Pi M1 [bpi 14a]

SoC	Allwinner A20
Prozessorarchitektur	ARMv7
Hardwarevirtualisierung	Ja, ARM Virtualization Extensions
64-bit fähig	Nein
Hauptspeicher	1 GB DDR3
Stromverbrauch	weniger als 5 Watt
Ethernet-Schnittstelle	Realtek RTL8211E, 1 GBit/s
Anschlüsse für Speichermedien	SD-Karten-Leser, USB 2.0, SATA
Eingesetzte Speichermedien	8GB SD-Karte (Betriebssystem)

Knoten 2 und 3 - Banana Pi M1

Zwei Banana Pi M1 wurden als weitere Knoten ausgewählt, da sie günstiger als der Banana Pi R1 sind und keinen integrierten Switch benötigen, aber über die gleiche ARMv7-Plattform wie der Banana Pi R1 verfügen. Die ursprüngliche Annahme dabei war, dass die gleiche Plattform den Konfigurationsaufwand reduzieren und für Homogenität sorgen würde. Später stellte sich jedoch heraus, dass bereits geringe Unterschiede in der Hardware aufgrund des Gerätebaums zusätzlichen Konfigurationsaufwand erfordern. Dies wird in 3.3.1 weiter ausgeführt. Tabelle 3.2 zeigt die technischen Daten eines Banana Pi M1.

3.1.2. Softwarekonfiguration

Folgende Software sollte auf dem Virtualisierungscluster eingesetzt werden:

Bootloader Das U-Boot

Virtualisierer Xen ARM

Linux-Kern Aktuelle stabile Version (Version 4.4)

Betriebssystem dom0 Debian 8 „Jessie“ armhf

Debian wurde als Betriebssystem gewählt, da es generell als sehr stabil gilt und vielfach auf Servern verwendet wird. Xen ARM wurde als Virtualisierer ausgewählt, da Xen x86 aufgrund seiner Leistungsfähigkeit vielfach in Virtualisierungsclustern zur Anwendung kommt.

3.2. Installationsversuche und endgültiger Virtualisierungscluster

In diesem Abschnitt werden die unternommenen Versuche zur Umsetzung des Entwurfs aus Abschnitt 3.1 geschildert und die finale Konfiguration des Virtualisierungsclusters beschrieben.

Begonnen wurde damit, Xen ARM auf einem Banana Pi M1 zu installieren. Der Banana Pi M1 wurde dem Banana Pi R1 aufgrund des fehlenden Switches vorgezogen, da die Konfiguration dadurch weniger aufwendig ist. Zunächst wurde eine Kombination aus der zu diesem Zeitpunkt aktuellen stabilen Versionen von Xen ARM (Version 4.6), Debian (Version 8) und des Linux-Kerns (Version 4.4) eingesetzt. Zum Erstellen des SD-Karten-Abbilds mit der nötigen Software wurde ein PC mit Debian 9 als Betriebssystem genutzt.

Da für ARMv7 keine vorgefertigten Installationsmedien existieren, müssen Bootloader, Linux-Kern und die Nutzerumgebung eigenständig kompiliert und konfiguriert werden. Beim Booten zeigte sich sofort, dass Xen manuell gepatcht werden muss, um ein Booten des in dom0 laufenden Linux-Kerns zu erlauben (genaueres in Abschnitt 3.3.2). Ein weiteres Problem war, dass Debian 8 in seinen Paketquellen nur über einen Xen Toolstack verfügt, der lediglich mit Xen ARM Version 4.4 lauffähig ist. Ein eigenständiges Kompilieren des Xen Toolstacks für Version 4.6 war nicht erfolgreich. Aus diesem Grund wurde auf Version 4.4 von Xen gewechselt. Interessanterweise war hier kein Patch notwendig, um ein Booten des Linux-Kerns zu ermöglichen. Mit dieser Konfiguration lief die Virtualisierung prinzipiell zwar, aber das Dateisystem von in virtuellen Maschinen laufenden Gast-Betriebssystemen wurde sofort irreparabel geschädigt, wenn es als beschreibbar eingehängt war. Aus diesem Grund erfolgte ein Umstieg auf eine Kombination aus Xen 4.6 und Ubuntu 15.10, das über einen mit Version 4.6 kompatiblen Toolstack verfügt. Bei dieser Kombination war wieder ein manueller Patch für Xen 4.6 notwendig, aber das Betreiben von virtuellen Maschinen verlief problemlos. Anschließend wurde Xen auch auf dem Banana Pi R1 installiert.

Hier zeigte sich, dass aufgrund von Unterschieden in der Hardware ein einfaches Übertragen des für den Banana Pi M1 erstellten Abbildes nicht möglich war (vgl. Abschnitt 3.3.1). Nachdem ein zweites Abbild für den Banana Pi R1 erstellt wurde, stellte sich heraus, dass Xen ARM Version 4.6 über keine Unterstützung für Live-Migration verfügt und die entsprechende Funktion bereits im Xen Toolstack deaktiviert ist. Dies war im Voraus nicht ersichtlich, denn die Übersicht über den Funktionsumfang von Xen Versionen [xen 16b] zeichnet alle Xen Varianten ab Version 4.0 mit Live-Migration aus und unterscheidet nicht zwischen unterstützten Prozessorarchitekturen. So entstand der Eindruck, dass Live-Migration von Xen ARM bereits unterstützt wäre. An dieser Stelle wurden sämtliche Versuche mit Xen ARM abgebrochen. Stattdessen erfolgte ein Wechsel zu KVM/ARM, welches Live-Migration bereits unterstützt.

Der Einsatz von KVM/ARM vereinfachte vieles. So war es nun möglich, anstatt manuell Kern, Bootloader und das Skript für den Bootloader zu konfigurieren und zu kompilieren, diese Arbeiten von den *Armbian build tools* [arm 16], einer Ansammlung von Skripten zur Erstellung von SD-Karten-Abbildern für ARM-Einplatinenrechner, übernehmen zu lassen. Innerhalb weniger Tage war der Virtualisierungscluster mit KVM/-ARM einsatzbereit. Zwar mussten auch hier aufgrund der unterschiedlichen Hardware der Knoten zwei unterschiedliche Abbilder für die SD-Karten erstellt werden, aber dank der Armbian build tools war der Aufwand gering. Die größte Schwierigkeit und den höchsten Zeitaufwand bereitete beim auf KVM/ARM basierenden Cluster die Konfiguration des im Banana Pi R1 integrierten Switches (näheres in Abschnitt 3.3.3), deren Notwendigkeit sich erst zeigte, als mehr als ein Knoten an den Banana Pi R1 angeschlossen wurde.

Für den finalen Virtualisierungscluster wurden die Hardwarekonfiguration und Vernetzung aus Abschnitt 3.1.1 und Abbildung 3.1 unverändert übernommen. Die Softwarekonfiguration des Entwurfes musste jedoch geändert werden. Der finale Virtualisierungscluster wurde mit folgender Software betrieben:

Bootloader Das U-Boot 2016.5

Virtualisierer KVM

Linux-Kern Stabile Version 4.6.2

Betriebssystem Ubuntu 16.04 LTS „Xenial Xerus“ armhf

Ubuntu 16.04 wurde im finalen Virtualisierungscluster verwendet, da die Unterstützung für Ubuntu 15.10 im Juli 2016 auslaufen sollte. Abbildung 3.2 zeigt ein Foto des Virtualisierungsclusters in der finalen Konfiguration.

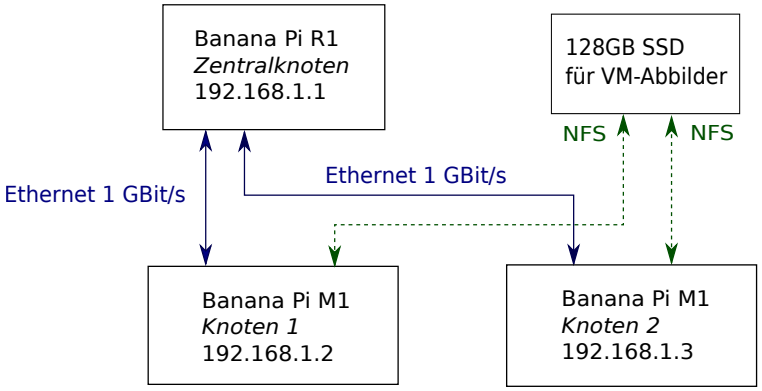

Abbildung 3.1.: Die Vernetzung der einzelnen Komponenten des Virtualisierungsclusters

Abbildung 3.2.: Foto des für die Arbeit erstellten Virtualisierungsclusters

3.3. Beobachtete Probleme und Herausforderungen

Die Installation und Konfiguration des Virtualisierungsclusters verlief nicht reibungslos, wie bereits in Abschnitt 3.2 dargestellt. Im Folgenden sind einige Probleme und Herausforderungen aufgelistet, die während der Arbeit mit dem Virtualisierungscluster aufgefallen sind und teilweise besonders im Vergleich zur PC-Plattform hervortreten. Begonnen wird mit allgemeinen Problemen von ARMv7-Plattformen in Abschnitt 3.3.1. In Abschnitt 3.3.2 werden temporäre Probleme, also Probleme, die im Lauf der Zeit vermutlich abnehmen werden, geschildert. Probleme, die spezifisch für die Hardware des für diese Arbeit erstellten Virtualisierungscluster sind, werden in Abschnitt 3.3.3 behandelt. Allen Problemen ist gemein, dass sie den Zeitaufwand für die Administration eines Virtualisierungsclusters auf der Basis von ARMv7-Plattformen im Vergleich zu einem auf der PC-Plattform basierenden Virtualisierungscluster deutlich erhöhen können.

3.3.1. Allgemeine Herausforderungen bei ARMv7-Plattformen

Durch die ursprüngliche Ausrichtung auf den Einsatz in eingebetteten Systemen existieren einige für ARMv7-Plattformen spezifische Herausforderungen, die vermutlich auch auf lange Sicht Bestand haben werden.

Gerätebaum erhöht administrativen Aufwand bei heterogenen Systemen

Der Gerätebaum wurde bereits in 2.3.1 vorgestellt. Auch wenn zwei unterschiedliche Systeme die gleiche ARMv7-Plattform nutzen, ist es möglich, dass der Gerätebaum für die Systeme individuell erstellt werden muss, wenn die periphere Hardware abweicht. Dies ist auch beim hier erstellten Virtualisierungscluster der Fall. Soll bei einem heterogenen Cluster der volle Funktionsumfang der Hardware ausgenutzt werden, so ist es unabdinglich, für jede Hardwareplattform einen eigenen Gerätebaum zu erstellen und auf den entsprechenden Knoten einzusetzen.

Als Beispiel seien hier der Banana Pi R1 und der Banana Pi M1 genannt. Beide verwenden einen Allwinner A20 SoC. Der Banana Pi R1 verwendet jedoch einen anderen Ethernetcontroller als der Banana Pi M1, da der Banana Pi R1 über fünf statt einem Ethernetanschluss verfügt und der Banana Pi R1 damit als Switch einsetzbar sein soll. Dieser Unterschied in der Hardware macht neben unterschiedlichen Gerätetreibern für die Ethernetcontroller auch unterschiedliche Gerätebäume, die die Ethernetcontroller entsprechend ihrer Eigenschaften beschreiben, erforderlich. Würde der gleiche Gerätebaum für beide Hardwareplattformen verwendet werden, wäre auf einer der beiden Hardwareplattformen der Ethernetcontroller nicht benutzbar.

Heterogene Hardware in einem Virtualisierungscluster auf der Basis von ARMv7-Plattformen bedeutet also einen höheren Konfigurationsaufwand und somit auch eine höhere Anfälligkeit für das Auftreten menschlicher Fehler.

Erhöhter Konfigurationsaufwand

Durch den nicht standardisierten Bootvorgang bei ARMv7-Plattformen (vgl. Abschnitt 2.3.2) ist das Installieren eines nativen Virtualisierers bei ARMv7-Plattformen aufwendiger als bei der PC-Plattform, da es an einer universellen Möglichkeit zur Installation von hardwarenaher Software fehlt [DaNi 14]. Während es beispielsweise bei der x86-Variante von Debian ausreicht, Xen aus den Paketquellen zu installieren und beim nächsten Boot im Bootloader *Grub* auszuwählen, ist dies bei ARMv7-Plattformen nicht möglich. Stattdessen muss der Bootloader *Das U-Boot*, der typischerweise bei ARMv7-Plattformen eingesetzt wird, gegebenenfalls manuell angepasst und kompiliert werden, sodass der Prozessor im *HYP* Modus gestartet wird. Denn nur im HYP Modus steht das für Hardwarevirtualisierung notwendige PL2 zur Verfügung. Ebenso muss ein Skript für den Bootloader erstellt und kompiliert werden, welches beinhaltet, von welcher Adresse Xen ARM und dom0 in den Hauptspeicher geladen und anschließend gestartet werden sollen. Diese Schritte sind aufwendig, fehleranfällig und die Portabilität ist eingeschränkt, da einige Details von der Hardwareplattform abhängig sein können. Auch gibt es bisher keine Installationsmedien für Betriebssysteme für ARMv7. Stattdessen müssen Betriebssystemabbilder manuell erstellt werden. Dies erschwert die Installation von Gast-Betriebssystemen in virtuellen Maschinen. Im Fall von Debian und Ubuntu kann beispielsweise mit dem Programm *debootstrap*

eines dieser Betriebssysteme installiert werden. Bei diesem Vorgehen ist das resultierende Betriebssystem aber größtenteils unkonfiguriert und somit ist möglicherweise weiterer Aufwand zum Erreichen der Wunschkonfiguration vonnöten.

In den Anhängen A und B befinden sich ausführliche Instruktionen zum Installieren von Xen ARM beziehungsweise KVM/ARM auf einem Banana Pi. Diese Instruktionen können nicht nur als Hilfestellung dienen, sondern zeigen auch den Aufwand, den das Installieren eines Virtualisierers auf ARMv7-Plattformen erfordert.

3.3.2. Temporäre Probleme

Die nachfolgenden Probleme sind temporärer Natur und es ist davon auszugehen, dass sie im Lauf der Zeit an Bedeutung verlieren werden.

Mangelnde Dokumentation

Der Umfang der verfügbaren Dokumentation für Virtualisierung auf ARM ist möglicherweise geringer für die PC-Plattform. Speziell bei den Banana Pis ist man hauptsächlich auf von Nutzern erstellte Dokumentation angewiesen. Ein weiteres Problem ist, dass Suchanfragen zu Software häufig zu x86-spezifischen Dokumenten führen, diese als solche aber im Dokument historisch bedingt oft nicht gekennzeichnet sind. Einige Instruktionen für x86 sind auf ARMv7 ohne weiteres anwendbar, andere hingegen müssen angepasst werden. Dies kann gegebenenfalls für Missverständnisse sorgen und zu Fehlern führen. Beispielsweise unterscheidet die Übersicht über den Funktionsumfang stabiler Versionen von Xen [xen 16b] bei Live-Migration nicht zwischen den Prozessorarchitekturen, sodass man davon ausgehen könnte, dass Live-Migration von Xen ARM unterstützt werden würde. Dieser Sachverhalt führte dazu, dass der Virtualisierungscluster zunächst mit Xen ARM als Virtualisierer entworfen wurde und später auf KVM/ARM gewechselt werden musste, da Live-Migration ein essentielles Kriterium eines Virtualisierungsclusters gemäß der Definition aus 2.1.3 ist. Dieses Beispiel zeigt, wie unpräzise Dokumentation das nachträgliche Anpassen eines Entwurfs für einen Virtualisierungscluster erforderlich machen kann. Auffällig ist die x86-Suchtreffer Problematik aber auch bei KVM, das in Konjunktion mit QEMU eingesetzt wird. Dort treten häufig Suchtreffer in Erscheinung, die sich lediglich mit der Emulation von ARM-Prozessoren mittels QEMU auf einem x86-Rechner befassen und somit für Virtualisierung auf ARM unerheblich sind.

Geringere Softwareverfügbarkeit

Da ARMv7-Plattformen weder im Desktop- noch im Serverbereich so populär sind wie die PC-Plattform, wird diese Plattform generell von weniger Software unterstützt [deb 16]. Es ist also zu prüfen, ob die erforderlichen Anwendungen für ARMv7 verfügbar sind und ob diese ARMv7-Varianten auch den gewünschten Funktionsumfang besitzen.

Kritische Softwarefehler in „stabilen“ Versionen

Während des Einrichtens des Virtualisierungsclusters sind zahlreiche kritische Softwarefehler aufgetreten. Beispielsweise besitzt die stabile Version 4.5 des Linux-Kerns einen Fehler, der auf dem Banana Pi eine korrekte Ethernet-Verbindung verhindert. Dieser wurde kurz darauf mit einem Patch behoben. Noch gravierender ist ein Fehler in Xen Version 4.4, der dazu führt, dass das Dateisystem von Gast-Domänen sofort irreparabel geschädigt wird, sofern es als beschreibbar eingehängt ist. Dieser Fehler ist in Xen Version 4.6 nicht mehr vorhanden. Auch die Armbian build tools, die zur Erstellung einer bootfähigen SD-Karte genutzt wurden, sind nicht fehlerfrei. So erzeugten sie bei der ersten Nutzung ein Abbild, in dessen Partitionstabelle die Werte für die Partitionsgrößen falsch waren. Dies verhinderte ein korrektes Lesen des Abbilds. Durch ein manuelles Anpassen der Partitionstabelle ließ sich das Problem beheben. Vereinzelt gibt es Berichte von anderen Nutzern mit dem gleichen Problem, aber sicher reproduziert werden konnte es noch nicht. Bei erneuter Nutzung der Skripte trat der Fehler nicht mehr auf und die Skripte funktionierten einwandfrei.

Auch wenn für jeden dieser Softwarefehler eine Lösung erarbeitet werden konnte, könnte die Existenz von diesen Fehlern ein Indikator dafür sein, dass Software aus dem Bereich Virtualisierung auf ARMv7-Plattformen derzeit zu wenig eingesetzt und getestet wird, um schwerwiegende Fehler in der Software rechtzeitig zu entdecken und so ein Auftreten solcher Fehler in sogenannten „stabilen“ Softwareversionen zu verhindern.

Die Suche nach den Fehlerursachen war teilweise sehr zeitaufwendig und hat im Fall von Xen ARM den Großteil der für die Installation benötigten Zeit beansprucht. Es sollte also für die Einrichtung eines auf ARM-Plattformen basierenden Virtualisierungsclusters ausreichend Zeit für eine eventuelle Fehlersuche und -behebung eingeplant werden.

Unfertige Software

Speziell bei der Arbeit mit Xen Version 4.6 wurde deutlich, dass Xen auf ARM noch nicht sämtliche Funktionen der x86-Variante unterstützt. Es fehlt nicht nur die Unterstützung von Live-Migration, auch einige Registerzugriffe sind noch nicht vollständig implementiert. Sie werden meistens einfach ignoriert und es wird eine Warnung über die Konsole ausgegeben.

Bei Xen ARM Version 4.6 war das Erstellen eines Patches notwendig, um Schreibvorgänge des Linux-Kerns von dom0 auf bestimmte Register zu ignorieren, da der Startvorgang des Linux-Kerns sonst von Xen ARM gestoppt wurde (für den Patch siehe Anhang A). Davon abgesehen wurde keine Beeinträchtigung des Betreibens von Virtuellen Maschinen aufgrund von nicht vollständig behandelten Registerzugriffen festgestellt.

Treiberverfügbarkeit

Die Problematik, dass Treiber nicht oder nur für bestimmte Versionen eines Betriebssystems zur Verfügung stehen, ist unter ARMv7 möglicherweise stärker ausgeprägt als bei der PC-Plattform. Eine mögliche Ursache wäre die Tatsache, dass die meisten ARM-Plattformen nach dem System-on-Chip Prinzip entworfen werden, da dies Vorteile bezüglich Kosten und Stromverbrauch bietet. Der Nachteil ist, dass die Treiber für die einzelnen Komponenten oft von einem einzelnen Hersteller abhängen. So ist der Nutzer stärker an die Treiberpolitik des Herstellers gebunden.

Beispielsweise unterstützen die vom Hersteller des Banana Pi herausgegebenen Treiber im April 2016 nur Version 3.4 des Linux-Kerns. Diese 4 Jahre alte Version bietet keine offizielle Unterstützung für Virtualisierer wie Xen ARM und KVM/ARM. Da für diese Arbeit eine neuere Version des Linux-Kerns nötig war, musste auf die Unterstützung einiger Hardwarefunktionen, wie zum Beispiel der Bluetooth-Verbindung, mangels kompatibler Treiber verzichtet werden. Glücklicherweise fehlten keine Treiber für Hardware, deren Fehlen den Einsatz in einem Virtualisierungscluster beeinträchtigt hätte.

Bei der PC-Plattform ist die Treiberproblematik leichter zu umgehen, da die Standardisierung der Plattform häufig den Einsatz von Standardtreibern erlaubt. Auch stehen vermutlich mehr Treiber als freie Software zur Verfügung, was eine Aktualisierung oder Anpassung der Treiber vereinfacht. Die modulare Bauweise der PC-Plattform vereinfacht es zudem, Komponenten zu wählen, die über die gewünschte Treiberunterstützung verfügen.

Je nach gewünschtem Einsatzzweck und der benötigten Software und Hardwarefunktionen ist also eine eingehende Prüfung der Treiberverfügbarkeit (auch mit Blick in die Zukunft) vor einer Neuanschaffung notwendig.

3.3.3. Probleme der eingesetzten Hardware

Die für den Virtualisierungscluster verwendeten Systeme stellten eigene Herausforderungen, welche jedoch nicht allgemein für alle Systeme, die auf ARMv7-Plattformen aufbauen, gelten. Dieser Abschnitt soll lediglich einen Einblick bieten, inwiefern die Auswahl des Systems zusätzliche Herausforderungen hervorrufen kann.

Aufwendige Konfiguration des Switches des Banana Pi R1

Der Banana Pi R1 wird teilweise als Router vermarktet. Tatsächlich enthält dieser aber nur einen 6-Port Switch. 5 Anschlüsse dieses Switches sind von außen zugänglich, der sechste Anschluss ist für den Banana Pi R1 selbst reserviert. Der Switch lässt sich auf verschiedene Arten betreiben. Der Banana Pi R1 baut nur eine Verbindung mit dem Switch auf, wenn der b53-Treiber für den Switch installiert ist. Dieser Treiber wird von der OpenWRT Community gepflegt und wurde bisher nicht in den Linux-Kern übernommen. Er muss also manuell hinzugefügt werden. Ist der b53-Treiber installiert, lässt sich der Switch zusätzlich mit dem Programm *swconfig* konfigurieren.

Die Konfiguration des Switches für den Einsatz in einem Virtualisierungscluster ist nicht intuitiv. Es müssen zwangsläufig Virtual Local Area Networks (VLANs) in Kombination mit einer Netzbrücke eingesetzt werden. Ein Port des Switches muss eine von den anderen Ports abweichende VLAN-ID besitzen. Über eine Netzbrücke können dann die beiden VLANs miteinander verbunden werden. Wird nur ein VLAN oder gar kein VLAN konfiguriert, sind entweder nur Punkt-zu-Punkt Verbindungen von verbundenen Knoten zum Banana Pi R1 möglich, d.h. der Switch verbindet die Ports nicht miteinander, oder der Switch verbindet die Ports, aber der Banana Pi R1 selbst kann sich nicht mit den anderen Stationen verbinden (siehe dazu auch die Netzkonfiguration in Anhang B).

Das Finden der korrekten Konfiguration und das damit im Zusammenhang erforderliche Experimentieren mit der Konfiguration des Switches lief nicht reibungslos ab. Teilweise musste der Banana Pi R1 während der Konfiguration neu gestartet werden, da sonst Softwarefehler verhinderten, dass die Netzverbindungen weiter funktionierten. Besonders störend war hierbei, dass das Betriebssystems während des Herunterfahrens manchmal abgestürzt ist. Dies ist problematisch, da deswegen eine Trennung vom Stromnetz erforderlich wird, was wiederum zur Folge haben kann, dass das Dateisystem beschädigt wird. Nach dem Trennen von der Stromzufuhr sollte also eine Fehlerüberprüfung und gegebenenfalls eine Fehlerkorrektur des Dateisystems erfolgen. Ist das Dateisystem irreparabel beschädigt, muss erneut ein Abbild eines Betriebssystems auf die SD-Karte geschrieben werden. Möglicherweise gehen dadurch kürzlich vorgenommene Konfigurationsschritte verloren. Ein irreparabel geschädigtes Dateisystem trat zwar nur einmal auf, aber die generelle Problematik erhöhte den für die Konfiguration benötigten Zeitaufwand erheblich.

3.4. Zusammenfassung

Es ist gelungen, einen Virtualisierungscluster auf der Basis von ARMv7-Plattformen mit KVM/ARM als Virtualisierer zu entwerfen, zu installieren und zu betreiben. Speziell bei Verwendung des Virtualisierers Xen ARM erhöhten zahlreiche Probleme den Zeitaufwand für die Installation deutlich. Ein Großteil aller aufgetretenen Probleme ist jedoch temporärer Natur oder hardwarespezifisch, sodass zu erwarten ist, dass mit der Zeit die Anzahl an Problemen abnehmen wird.

Als festgestellt wurde, dass Xen ARM, welches im Entwurf als Virtualisierer vorgesehen war, derzeit keine Live-Migration unterstützt und damit die Definition eines Virtualisierungsclusters aus Abschnitt 2.1.3 nicht erfüllt, wurden weitere Experimente mit Xen ARM abgebrochen. Dennoch war Xen ARM auf den Knoten des Clusters lauffähig, es fehlte lediglich die Unterstützung von Live-Migration. Auf dieser Basis können gegebenenfalls künftige Untersuchungen mit weiterentwickelten Versionen von Xen ARM, sobald diese Unterstützung für Live-Migration bieten, aufbauen. Mit KVM/ARM hingegen ist es möglich, mehrere virtuelle Maschinen auf einem Knoten zu betreiben und diese im laufenden Betrieb zwischen den Knoten zu verschieben. Wie leistungsfähig KVM/ARM auf dem erstellten Virtualisierungscluster ist, wird nun in Kapitel 4 untersucht.

4. Leistungsevaluierung des Virtualisierungsclusters

Der in Kapitel 3 erstellte Virtualisierungscluster soll in diesem Kapitel auf seine Leistungsfähigkeit untersucht werden.

Das Ziel der Messungen soll sein, den in der Praxis zu erwartenden Wirkungsgrad der Virtualisierung mit KVM/ARM möglichst gut abzubilden. Ursprünglich waren aus diesem Grund umfangreiche Messungen und diverse Testszenarien pro Komponente ähnlich zu den in den Arbeiten [Gebh 10] [Lemb 10] [Tsio 10] [Roma 10] [Lind 10] gewählten Szenarien vorgesehen, da somit einerseits eine Vielzahl an Vergleichsmessungen anderer Virtualisierer auf der PC-Plattform zur Verfügung gestanden hätte und andererseits die dortigen Szenarien eine gute Annäherung an den in der Praxis zu erwartenden Wirkungsgrad liefern sollten. Es hat sich jedoch herausgestellt, dass viele Programme für Leistungsmessungen nicht oder nicht korrekt auf ARMv7-Plattformen laufen. Entweder es existiert gar keine ARMv7-Variante eines bestimmten Programms oder die ARMv7-Variante arbeitet nicht korrekt. So wurden die Programme für die hier vorgenommenen Leistungstests letztlich nach dem Kriterium der Lauffähigkeit auf ARMv7-Plattformen ausgewählt, um wenigstens ein paar Anhaltspunkte für den zu erwartenden Wirkungsgrad zu erhalten.

Dieses Kapitel ist wie folgt aufgebaut. Zur Einleitung werden in Abschnitt 4.1 zunächst einige Begriffe und Herausforderungen der Leistungsevaluierung dargestellt. In Abschnitt 4.2 folgen Messungen des Wirkungsgrads verschiedener Komponenten. Auswirkungen von speicherintensiven Prozessen auf die Dauer von Live-Migrationen werden in Abschnitt 4.3 behandelt. In Abschnitt 4.4 werden die Ergebnisse dieses Kapitels zusammengefasst.

4.1. Grundlegende Begriffe und Herausforderungen der Leistungsevaluierung

Bevor mit den Leistungsmessungen begonnen wird, sollen zunächst einige für dieses Kapitel relevante Begriffe erklärt beziehungsweise definiert werden. Zusätzlich zu der Begriffsklärung werden noch die Herausforderungen, die Leistungsmessungen auf Rechnern generell und virtuellen Maschinen im Speziellen stellen, beschrieben.

4.1.1. Leistungsbegriff

Wenn in dieser Arbeit der Begriff *Leistung* zur Anwendung kommt, dann ist damit in einem bestimmten Zeitraum geleistete Arbeit gemeint. Somit orientiert sich diese Arbeit am klassischen Leistungsbegriff aus der Physik, der ähnlich definiert ist. Bei den folgenden Leistungsmessungen wurde also gemessen, wieviel Arbeit in einem gewissen Zeitraum von einem Programm geleistet wurde.

4.1.2. Herausforderungen bei Leistungsmessungen von Rechnern

Ein Rechner ist eine komplexe Maschine, die aus einer Vielzahl von Komponenten besteht. Jede dieser Komponenten hat einen Einfluss auf die für den Nutzer sichtbare Gesamtleistung des Rechners. Da die Komponenten eng miteinander verknüpft sind, beeinflussen sich diese in ihrer Leistung gegenseitig. Ein schneller Prozessor kann beispielsweise durch einen langsamen Hauptspeicher in seiner Leistung beeinträchtigt werden, wenn der Hauptspeicher nicht schnell genug die für die Berechnungen des Prozessors nötigen Daten liefern kann. Zudem

belasten einzelne Programme häufig eine einzelne Komponente stärker als andere Komponenten. Aus diesem Grund werden spezielle Programme für Leistungsmessungen verwendet, die den Fokus auf die Leistungsmessung einer einzelnen Komponente legen. Ist die Komponente bekannt, die ein Programm aus der Praxis hauptsächlich belastet, können auf Basis solcher Leistungsmessungen Rückschlüsse auf die in der Praxis zu erwartende Leistung des Programms auf dem jeweiligen System gezogen werden.

4.1.3. Wirkungsgrad im Kontext der Virtualisierung

In der Physik ist der Wirkungsgrad definiert als das Verhältnis von Nutzleistung zu zugeführter Leistung. Im Kontext der Virtualisierung gibt der Wirkungsgrad einer Komponente an, wieviel Leistung nach Abzug des zusätzlichen Rechenaufwands für die Virtualisierung im Vergleich zu einer nativen Ausführung, also einer Ausführung ohne Virtualisierung, zu erwarten ist. Je höher der Wirkungsgrad, desto effizienter ist die Virtualisierung der Komponente.

4.1.4. Zeitabfragenproblematik bei virtuellen Maschinen

Leistungsmessungen innerhalb einer virtuellen Maschine stehen vor einer zusätzlichen Herausforderung, die eine genaue Messung erschwert. Bei Virtualisierung kann nicht garantiert werden, wann eine Zeitabfrage eines auf einer virtuellen Maschine laufenden Gast-Betriebssystems beantwortet wird. Dies liegt daran, dass neben der virtuellen Maschine meist auch ein Host-Betriebssystem und gegebenenfalls andere virtuelle Maschinen laufen. Dadurch muss die verfügbare Rechenzeit des Prozessors aufgeteilt werden. So kann es vorkommen, dass eine virtuelle Maschine gar nicht aktiv ist, wenn eine Zeitabfrage stattfinden sollte. Weitere Ebenen von Scheduling erschweren die Problematik zusätzlich [DgFKL 11]. Konkret bedeutet dies, dass das Ergebnis einer Zeitabfrage nicht immer den eigentlichen Zeitpunkt der Abfrage angibt, sondern lediglich eine Näherung darstellt [Inc. 08]. Somit ist eine Verfälschung der Messergebnisse möglich und wahrscheinlich, da die Programme, die für die Leistungstests eingesetzt werden, notwendigerweise die Zeit messen müssen, um aus der geleisteten Arbeit und der vergangenen Zeit die Leistung zu berechnen.

Es existieren mehrere Wege, dieser Problematik zu begegnen. Lindinger beispielsweise modifizierte das Programm Linpack so, dass es über das angeschlossene Netz ein Paket an einen externen Rechner zur Zeitmessung schickte [Lind 10]. Andere Programme bieten bereits eine Client/Server-Funktionalität, sodass der Server innerhalb einer virtuellen Maschine Arbeit verrichtet und der Client außerhalb der virtuellen Maschine in einer nativen Umgebung läuft und dort eine präzise Zeitmessung vornehmen kann [Gebh 10]. In dieser Arbeit kommt, sofern nicht anders angegeben, eine dritte Methode zum Einsatz, die bereits am Lehrstuhl für Kommunikationssysteme und Systemprogrammierung angewendet wurde. Es wurde innerhalb einer virtuellen Maschine pro Programm zur Leistungsmessung ein Shell-Skript erstellt, dass das Programm mit den gewünschten Einstellungen aufruft. Aufgerufen wurde das Shell-Skript über eine *ssh-Sitzung*, wobei Secure Shell (ssh) ein Protokoll zum Abhalten von Shell-Sitzungen über ein Netz bezeichnet. Diese ssh-Sitzung wurde von einer nativen Umgebung aus gestartet. Die Dauer der ssh-Sitzung wurde durch den Shell-Befehl *time*, der die Ausführungszeit eines Befehls misst, gemessen. Da die von *time* gemessene Zeit auf einer Uhr einer nativen Umgebung basiert, ist sie nicht von der Zeitabfragenproblematik innerhalb von virtuellen Maschinen betroffen. Ein Aufruf einer solchen ssh-Sitzung sah beispielsweise so aus:

Listing 4.1:

```
{ time ssh root@192.168.1.100 "sh␣dd.sh" ; } 2> dd_messung_nativ.txt
```

Dieses Verfahren führt jedoch zu leichten Verfälschungen der Messergebnisse, da *time* auch die Zeit, die für den Verbindungsauf- und abbau der ssh-Sitzung notwendig ist, in die Messung mit einbezieht. Aus diesem Grund wurde die Anzahl der Durchläufe individuell pro Programm so gewählt, dass in etwa eine Gesamtlaufzeit von 8 Stunden pro Messung erreicht wurde. Durch eine so lange Laufzeit wurde sichergestellt, dass die für die Praxis relevantere Dauerleistung und nicht die Spitzenleistung gemessen wurde und der für die Kommunikation über ssh zusätzlich benötigte Zeitaufwand sich nur unwesentlich auf das Endergebnis ausgewirkt haben sollte. Die Authentifizierung erfolgte automatisiert über einen Schlüsselaustausch und mögliche Ausgaben der aufgerufenen Programme an die Konsole wurden so umgeleitet, dass sie nicht in der ssh-Sitzung

sichtbar waren, sodass der Kommunikationsaufwand während der Messungen auf ein Mindestmaß reduziert war.

4.2. Messungen des Wirkungsgrads der Virtualisierung mit KVM/ARM

In den folgenden Abschnitten wird der Wirkungsgrad jeweils für die Komponenten Prozessor (Abschnitt 4.2.1), Hauptspeicher (Abschnitt 4.2.2), Sekundärspeicher (Abschnitt 4.2.3) und Netz (Abschnitt 4.2.4) gemessen und ausgewertet. Diese Komponenten wurden ausgewählt, da sie normalerweise den größten Anteil an der Systemleistung tragen und sie somit für allgemeine Aussagen über die Leistungsfähigkeit am besten geeignet erscheinen (vgl. dazu auch die Arbeiten [Gebh 10] [Lemb 10] [Tsio 10] [Roma 10] [Lind 10]).

Sämtliche Messungen wurden auf dem Banana Pi R1 vorgenommen, der durch die angeschlossene SSD lokalen Zugriff auf die Abbilder der virtuellen Maschinen besaß. Somit konnte der Einfluss der bei anderen Knoten erforderlichen Kommunikation über das Netz auf die Messungen vermieden werden. Die Hardwarespezifikationen des Banana Pi und des restlichen Clusters wurden bereits in Abschnitt 3.1.1 erläutert. Sowohl in der nativen als auch virtuellen Umgebung wurde eine minimale Konfiguration von Ubuntu 16.04 als Betriebssystem verwendet. Der virtuellen Maschine wurden ein virtueller Prozessor und 256 MB Hauptspeicher zugewiesen.

4.2.1. Wirkungsgrad von rechenintensiven Prozessen

Zur Untersuchung des Wirkungsgrads von Prozessen, die hauptsächlich den Prozessor belasten, wurden die Programme *Dhrystone* und *Whetstone* verwendet. *Dhrystone* ist ein Programm, dass die Leistung eines Prozessors bei Ganzzahloperationen misst. *Whetstone* hingegen misst die Leistung eines Prozessors bei Gleitkommaoperationen. Diese Programme wurden ausgewählt, da sie bereits seit Jahrzehnten für Leistungsmessungen von Prozessoren Anwendung finden und entsprechend eine große Menge an Vergleichsdaten existiert. Sie wurden außerdem ausgewählt, da sie sich für die in Abschnitt 4.1.4 erläuterte Methode zur Messung über ssh zum Umgehen der Zeitabfragenproblematik eignen. Das Ergebnis der Messungen berechnet sich aus der gewählten Anzahl an Durchläufen geteilt durch die von *time* gemessene Zeit.

Im Falle von *Dhrystone* wurden insgesamt $7,68 * 10^{10}$ Durchläufe ausgeführt, während für *Whetstone* $4,032 * 10^5$ Durchläufe gewählt wurden. Die Ergebnisse der beiden Programme finden sich in den Tabellen 4.1 und 4.2. In den Abbildungen 4.1 und 4.2 sind die Ergebnisse grafisch dargestellt. Die erreichten Wirkungsgrade von 99,25% bei *Dhrystone* beziehungsweise 98,73% bei *Whetstone* sind sehr gute Werte. Bei der Virtualisierung von rechenintensiven Prozessen wird also nahezu die Leistung einer nativen Ausführung erreicht.

Tabelle 4.1.: Messergebnisse des Programms *Dhrystone*

Ergebnis VM	Ergebnis nativ	Wirkungsgrad
2890324,383 Dhrystones/s	2912028,418 Dhrystones/s	99,25%

Tabelle 4.2.: Messergebnisse des Programms *Whetstone*

Ergebnis VM	Ergebnis nativ	Wirkungsgrad
1492,92 MWIPS	1512,12 MWIPS	98,73%

4.2.2. Wirkungsgrad von Zugriffen auf den Hauptspeicher

Ursprünglich sollte der Wirkungsgrad von Zugriffen auf den Speicher mit dem Programm *ramspeed* ermittelt werden, da dieses in den Arbeiten [Gebh 10] [Lemb 10] [Tsio 10] [Roma 10] [Lind 10] verwendet wurde. Von diesem existiert jedoch keine ARMv7-Variante. Als Alternative wurde das Programm *sysbench* ausgewählt,

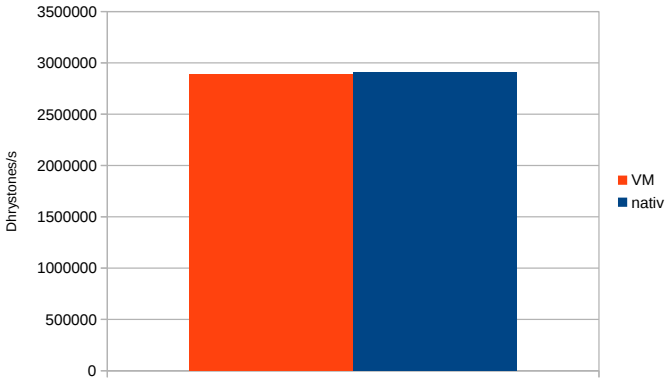

Abbildung 4.1.: Messergebnisse des Programms *Dhrystone*

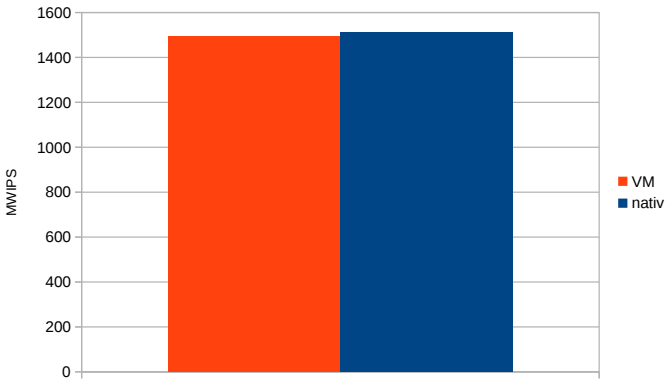

Abbildung 4.2.: Messergebnisse des Programms *Whetstone*

eine Sammlung von Werkzeugen zur Leistungsmessung, die auch Speicherzugriffe messen kann. Angedacht waren Messungen mit verschiedenen Blockgrößen der Zugriffe, da unterschiedliche Blockgrößen die Leistung beeinflussen können [Lind 10]. *Sysbench* bietet wie *ramspeed* Optionen zum Einstellen der Blockgröße. Es stellte sich jedoch heraus, dass die Messergebnisse nicht stimmen konnten, da regelmäßig ein Speicherdurchsatz von 0,0 MB/s gemessen wurde. Aus diesem Grund wurde mangels Alternative das Programm *mbw* [mbw 15], kurz für „Memory Bandwidth Benchmark“ ausgewählt. Dieses misst die Kopiergeschwindigkeit eines *Arrays*, also einer Datenstruktur, vorgegebener Größe im Hauptspeicher. Es werden in einem Lauf also Lese- und Schreibzugriffe kombiniert.

Für diese Messung wurden über ssh 160.000 Durchläufe ausgeführt. Die Größe des zu kopierenden Arrays betrug 60 MB, größere Allokationen von Speicher mit einer Anfrage waren innerhalb einer virtuellen Maschine mit *mbw* nicht möglich. Das Ergebnis der Messung wurde berechnet, indem die Anzahl der Durchläufe mit der Array-Größe multipliziert wurde und anschließend durch die von *time* gemessene Zeit geteilt wurde. Das zu kopierende Array wird von *mbw* nur einmal während des Programmstarts angelegt und nicht bei jedem Durchlauf.

Das Programm *mbw* wurde folgendermaßen aufgerufen:

Listing 4.2:

```
mbw -t0 -n 160000 60M
```

Die mit *mbw* erzielten Durchsatzraten befinden sich in Tabelle 4.3 und sind in Abbildung 4.3 grafisch dargestellt. Es wurde ein Wirkungsgrad von 97,24% ermittelt. Dieser stellt ein gutes Ergebnis dar, Virtualisierer auf der PC-Plattform erreichen sowohl bessere als auch schlechtere Werte in einem Bereich von 86,96% bis 100,63%, je nach Virtualisierer und Systemkonfiguration [Lind 10]. Es ist also auch bei der Virtualisierung von speicherintensiven Prozessen nahezu die Leistung einer nativen Ausführung zu erwarten.

Tabelle 4.3.: Speicherdurchsatz mit *mbw* Version 1.2.2

Speicherdurchsatz VM	Speicherdurchsatz nativ	Wirkungsgrad
323,20 MB/s	332,36 MB/s	97,24%

4.2.3. Wirkungsgrad von Zugriffen auf den Sekundärspeicher

Für die Messungen der Komponente Sekundärspeicher wurde zunächst das Programm *iometer* ausgewählt, ebenfalls in Anlehnung an die in Abschnitt 4.2 genannten Arbeiten. Allerdings existiert auch von *iometer* keine Variante für ARMv7. Aus diesem Grund wurde als Alternative *Flexible I/O Tester* (kurz: *fio*) gewählt. Dieses Programm ermöglicht das Messen von Datendurchsatzraten bei Schreib-/Lesevorgängen und erlaubt es, durch vielfältige Konfigurationsmöglichkeiten genau festzulegen, was gemessen werden soll. Es bietet eine Client/Server-Funktionalität ähnlich wie *iometer*, die eine genaue Zeitmessung innerhalb von virtuellen Maschinen erlaubt, wenn der Client auf einem externen Rechner in nativer Umgebung läuft. Es hat sich jedoch herausgestellt, dass *fio* nur in einer nativen Umgebung auf ARMv7-Plattformen fehlerfrei operiert. Innerhalb einer virtuellen Maschine terminiert es nicht.

Mangels Alternativen wurden letztendlich die Programme *dd* zur Messung der Leistung von Schreibzugriffen und *hdparm* zur Messung der Leistung von Lesezugriffen auf den Sekundärspeicher verwendet. Bei *dd* handelt es sich um ein simples Programm, welches lediglich Daten von einem Ort an einen anderen kopiert. Für die Messung der Leistung von Schreibzugriffen mittels *dd* wurden 2000 mal 512 MB auf den Sekundärspeicher geschrieben. Wieder wurde die Zeit mittels *time* und *ssh* gemessen und das Ergebnis aus der Anzahl der Durchläufe multipliziert mit den geschriebenen Bytes pro Durchlauf geteilt durch die gemessene Zeit ermittelt. Es wurde mit unterschiedlichen Blockgrößen experimentiert, in Testläufen mit einer geringen Anzahl an Wiederholungen ergab sich jedoch kein nennenswerter Unterschied der Ergebnisse, sodass letztendlich nur mit einer Blockgröße von 1 MB gemessen wurde. Als Datenquelle diente die Datei */dev/zero*. Hierbei handelt es sich um eine *Pseudodatei*, sie existiert nicht wirklich auf dem Sekundärspeicher. Bei Lesezugriffen auf die

Datei */dev/zero* wird eine Kette von Nullen zurückgegeben. Durch die Benutzung von */dev/zero* als Datenquelle maß *dd* bei den vorgenommenen Messungen nicht die Kopierleistung, also die Kombination von Lese- und Schreibzugriffen, sondern lediglich die Schreibleistung.

Das Programm *hdparm* wurde für die Messung der Leistung von sequenziellen Lesezugriffen verwendet. Dabei bietet *hdparm* drei Arten von Messungen an. Zum einen kann die Leserate des Puffers von Linux für den Sekundärspeicher bestimmt werden, dabei wird lediglich der Puffer gelesen und nicht der Sekundärspeicher. Zum anderen kann *hdparm* die Leserate des Sekundärspeicher messen, wahlweise unter Ausnutzen des Seitenpuffers von Linux oder mit der *O_DIRECT* Option des Linux-Kerns, sodass ein ungepuffertes Lesen erfolgt. Für diese Arbeit wurden nur die letzten beiden Arten gemessen, da erstere eher ein Indikator für die Leistungsfähigkeit von Prozessor und Speicherhierarchie als für die Leistungsfähigkeit des Sekundärspeichers ist [hdp 16], für diese Komponenten jedoch mit *Dhrystone*, *Whetstone* und *mbw* Programme gewählt wurden, die Anwendungen aus der Praxis besser annähern sollten.

Es sei angemerkt, dass sowohl *dd* als auch *hdparm* lediglich die Leistung sequenzieller Zugriffe auf den Sekundärspeicher messen. In der Praxis treten allerdings auch zufällige Zugriffe auf von einander entfernt liegende Sektoren auf. Bei diesen ist die Leistung häufig geringer [Lind 10]. Mangels geeigneter Programme zur Messung solcher Zugriffe konnten diese im Rahmen der Arbeit jedoch nicht gemessen werden.

Die virtuelle Maschine war so konfiguriert, dass sie über die *virtio* [Russ 08] Schnittstelle auf den Sekundärspeicher zugriff, da *virtio* sich zu einem de facto Standard für Ein-/Ausgabeoperationen in virtuellen Maschinen entwickelt hat und als solcher in der Praxis verbreitete Anwendung findet.

Das Programm *dd* wurde in einer Schleife mit 2000 Durchläufen wie folgt aufgerufen:

Listing 4.3:

```
dd bs=1M count=512 if=/dev/zero conv=fdatasync \
of=<Pfad zur zu schreibenden Datei auf Sekundärspeicher>
```

hdparm wurde in einer Schleife mit 2000 Durchläufen folgendermaßen aufgerufen:

Listing 4.4:

```
hdparm -t <Pfad zum Sekundärspeicher>
hdparm -t --direct <Pfad zum Sekundärspeicher>
```

Die mit *dd* erzielten Schreibraten sind in Tabelle 4.4 und Abbildung 4.4 dargestellt, während die mit *hdparm* erzielten Leseraten in Tabelle 4.5 und Abbildung 4.5 dargestellt sind. Der erreichte Wirkungsgrad von 94,81% bei sequenziellen Schreibzugriffen auf den Sekundärspeicher ist ein wettbewerbsfähiger Wert im Vergleich zu anderen Virtualisierern auf der PC-Plattform. Bei Lesezugriffen ist jedoch mit deutlichen Einbußen zu rechnen. Sowohl beim direkten als auch beim gepufferten Lesen liegt der Wirkungsgrad bei etwa 50%. Andere Virtualisierer erreichen auf der PC-Plattform mindestens einen Wirkungsgrad von 74,16%, teilweise aber auch nahezu die Leistung einer nativen Ausführung [Lind 10]. Bei Anwendungen, die häufig Leseoperationen auf den Sekundärspeicher ausführen, ist also bei einer Virtualisierung durch KVM/ARM mit deutlichen Leistungsbußen zu rechnen.

Es sei angemerkt, dass im Falle von *hdparm* die virtuelle Maschine über das NFS-Protokoll auf den lokalen Sekundärspeicher zugegriffen hat, während der Zugriff in der nativen Umgebung direkt auf den Sekundärspeicher erfolgte. Das Programm *hdparm* greift immer direkt, also unter Umgehung von Dateisystemen, auf den zu messenden Sekundärspeicher zu. Die virtuelle Maschine war jedoch so konfiguriert, dass über das NFS-Protokoll auf das VM-Abbild zugegriffen wurde, da dies im praktischen Einsatz ebenfalls so wäre. Unter Hinzunahme der Tatsache, dass die *hdparm* Messung innerhalb der virtuellen Maschine die Uhr eben dieser zur Zeitmessung verwendete, sind die Ergebnisse mit Vorsicht zu werten. Der Zugriff von *dd* auf den Sekundärspeicher erfolgte in beiden Fällen über das NFS-Protokoll, um eine bessere Vergleichbarkeit der Resultate zu gewährleisten.

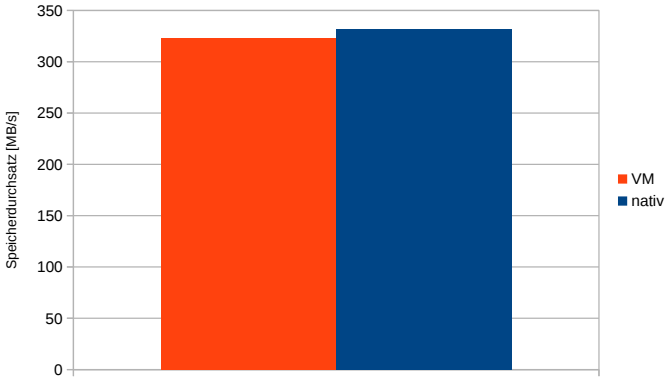

Abbildung 4.3.: Speicherdurchsatz mit *mbw*

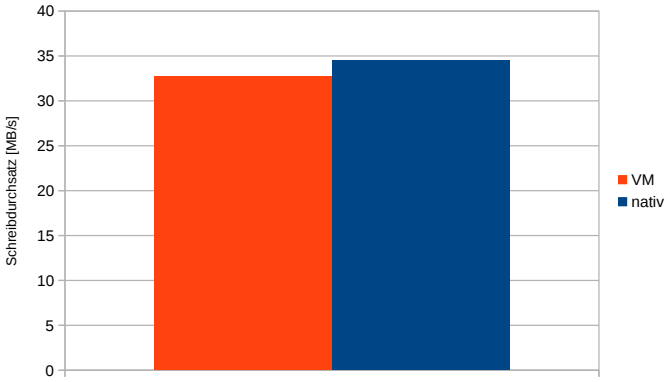

Abbildung 4.4.: Schreibdurchsatz des Sekundärspeichers mit *dd*

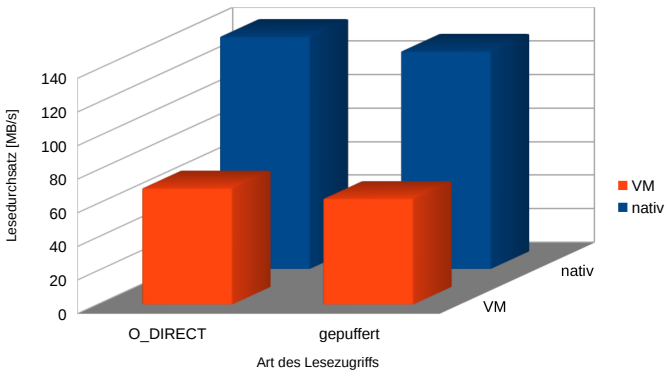

Abbildung 4.5.: Lesedurchsatz des Sekundärspeichers mit *hdparm*

Tabelle 4.4.: Sequentielle Schreibrate auf Sekundärspeicher mit *dd* Version 8.25

Schreibdurchsatz VM	Schreibdurchsatz nativ	Wirkungsgrad
32,73 MB/s	34,52 MB/s	94,81%

Tabelle 4.5.: Sequentielle Leserate auf Sekundärspeicher mit *hdparm* Version 9.48

Zugriffsart	Lesedurchsatz VM	Lesedurchsatz nativ	Wirkungsgrad
O_DIRECT	69,34 MB/s	138,38 MB/s	50,11%
gepuffert	129,59 MB/s	63,18 MB/s	48,75%

4.2.4. Wirkungsgrad von Zugriffen auf das Netz

Für die Messung des Wirkungsgrads von Netzzugriffen fiel die Wahl auf das Programm *iperf*. Das Programm *iperf* misst den Datendurchsatz im Netz und erlaubt das Konfigurieren der zu verwendenden Paketgrößen. Eine geringere Paketgröße führt meist zu einem geringeren Durchsatz. Für die hier vorgenommenen Messungen wurden die Paketgrößen 4 Kilobyte (KB), 32 KB, 256 KB, 1 MB und 4 MB in Anlehnung an [Lind 10] ausgewählt, da sie ein großes Spektrum abdecken und so bessere Aussagen über die in der Praxis zu erwartende Leistung erlauben. Als Anwendung läuft *iperf* auf Schicht 7 des *ISO/OSI-Schichtenmodells*, welches für Netzkommunikation notwendige Protokolle in Schichten mit unterschiedlichen Aufgabenbereichen einteilt und so einen modularen und flexiblen Aufbau von Netzkommunikation begünstigt. Gemessen wurde der Durchsatz unter Verwendung des Transmission Control Protocol (TCP), das auf Schicht 4, der Transportschicht, agiert. In dieser Konfiguration ähnelt *iperf* Serveranwendungen, die in der Praxis ebenfalls auf Schicht 7 agieren und häufig über TCP kommunizieren.

Ebenso wie bei Zugriffen auf den Sekundärspeicher war die virtuelle Maschine auch für Netzzugriffe so konfiguriert, dass sie die *virtio* Schnittstelle nutzte. Da *iperf* eine Client/Server-Architektur nutzt und somit unanfällig für die Zeitabfragenproblematik von virtuellen Maschinen ist, wurden sowohl in der nativen als auch in der virtuellen Umgebung jeweils 10 Durchläufe pro Paketgröße, mit anschließendem Bilden des arithmetischen Mittels, durchgeführt. Der *iperf* Client lief auf einem leistungsfähigen PC, der an das Netz des Banana Pi R1 angeschlossen war, um auszuschließen, dass der deutlich leistungsschwächere A20 SoC der Banana Pis die Leistungsmessungen beeinflussen könnte. Der *iperf* Client wurde wie folgt aufgerufen:

Listing 4.5:

```
iperf -c <IP-Adresse> -l <Blockgröße> -r
```

Die Ergebnisse der Messungen befinden sich für Lesezugriffe in Tabelle 4.6 und für Schreibzugriffe in Tabelle 4.7. Sie sind in Abbildung 4.6 beziehungsweise 4.7 grafisch dargestellt. Ein Wirkungsgrad von in etwa 50% bei lesenden Zugriffen beliebiger Blockgrößen auf das Netz ist ein sehr schlechter Wert. Andere Virtualisierer erzielen auf der PC-Plattform Werte bis zu 99% [Lind 10]. Interessant ist, dass der Wirkungsgrad bei Schreibzugriffen auf das Netz sehr von der Paketgröße abhängt. Im schlechtesten Fall beträgt er bei einer Paketgröße von 4 Kilobyte gerade einmal 26,52%. Bei 32 Kilobyte Paketgröße beträgt er bereits 84,91% und bei einer Paketgröße von 256 Kilobyte wurde vermutlich aufgrund von Schwankungen der Messergebnisse sogar ein Wirkungsgrad von über 100% erreicht. Bei noch größeren Paketen fiel der Wirkungsgrad auf etwa 90%. Bei netzintensiven Anwendungen sind also bei einer virtualisierten Ausführung Leistungsbußen zu erwarten. Je höher der Anteil an Leseoperationen auf das Netz ist, desto größer werden die Leistungseinbußen ausfallen.

Es sei angemerkt, dass die erreichten Durchsatzraten teilweise deutlichen Schwankungen von bis zu 200 MB/s unterlagen, weswegen in den Tabellen die Standardabweichung mit angegeben ist. Da der Wirkungsgrad auf Basis des Mittelwertes berechnet wurde, sind Abweichungen in der Praxis möglich.

Tabelle 4.6.: Datendurchsatz bei Lesezugriffen auf das Netz mit verschiedenen Blockgrößen mit *iperf* Version 2.0.5

Blockgröße	Durchsatz nativ	$\sigma_{Durchsatz\ nativ}$	Durchsatz VM	$\sigma_{Durchsatz\ VM}$	Wirkungsgrad
4 KB	451,60 MB/s	6,16 MB/s	847,90 MB/s	75,69 MB/s	53,26%
32 KB	444,20 MB/s	8,17 MB/s	890,30 MB/s	39,86 MB/s	49,84%
256 KB	454,80 MB/s	14,48 MB/s	866,90 MB/s	48,07 MB/s	52,46%
1 MB	478,10 MB/s	35,39 MB/s	917,40 MB/s	6,65 MB/s	52,11%
4 MB	448,20 MB/s	8,72 MB/s	878,80 MB/s	2,94 MB/s	51,00%

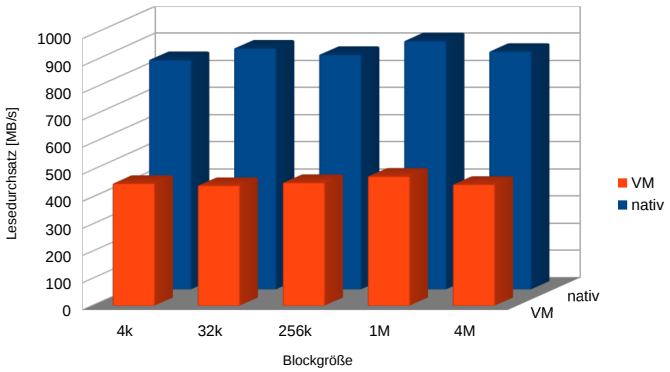

Abbildung 4.6.: Durchsatz von Lesezugriffen auf das Netz mit *iperf*

Tabelle 4.7.: Datendurchsatz bei Schreibzugriffen auf das Netz mit verschiedenen Blockgrößen mit *iperf* Version 2.0.5

Blockgröße	Durchsatz VM	$\sigma_{Durchsatz\ VM}$	Durchsatz nativ	$\sigma_{Durchsatz\ nativ}$	Wirkungsgrad
4 KB	73,86 MB/s	2,08 MB/s	278,50 MB/s	118,93 MB/s	26,52%
32 KB	347,80 MB/s	12,62 MB/s	409,60 MB/s	62,22 MB/s	84,91%
256 KB	479,20 MB/s	23,09 MB/s	467,70 MB/s	48,53 MB/s	102,46%
1 MB	505,30 MB/s	52,62 MB/s	566,50 MB/s	45,48 MB/s	89,20%
4 MB	523,40 MB/s	2,94 MB/s	588,80 MB/s	13,95 MB/s	88,89%

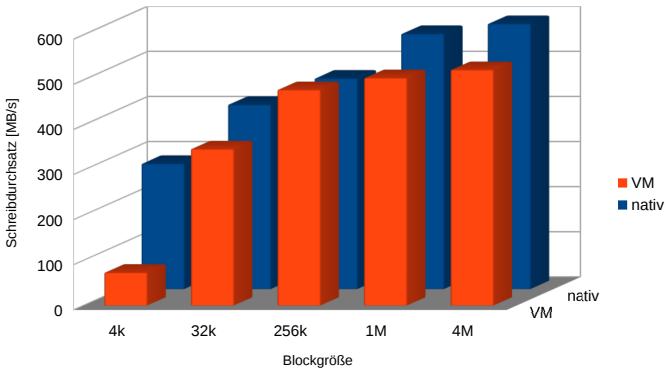

Abbildung 4.7.: Durchsatz von Schreibzugriffen auf das Netz mit *iperf*

4.3. Messung der Auswirkungen von speicherintensiven Prozessen auf die Dauer von Live-Migrationen

KVM nutzt das *pre-copy* Verfahren für die Live-Migration. Konkret bedeutet dies, dass bei der Live-Migration zunächst so viele Speicherseiten wie möglich übertragen werden. Dann werden in mehreren Iterationen die bereits übertragenen Seiten aktualisiert, wenn sich ihr Inhalt während der laufenden Migration verändert hat. Ab einem gewissen Schwellwert an verbliebenen Seiten wird die virtuelle Maschine angehalten, die letzten Seiten werden übertragen und anschließend wird die virtuelle Maschine auf dem neuen Host fortgesetzt [LuLi 07]. Speicherintensive Prozesse können die Übertragungszeit deutlich beeinflussen, wenn sie sehr viele Seiten in kurzer Zeit überschreiben, da dies immer wieder ein erneutes Übertragen der geänderten Seiten erfordert [SVTR 11]. Wie sehr sich ein speicherintensiver Prozess bei KVM/ARM auf die Migrationszeit auswirkt, soll nun untersucht werden. Dazu wird das Programm *memtester* [mem 16] verwendet. Das Programm ist ursprünglich zum Erkennen von defekten Bits des Hauptspeichers gedacht. Dazu alloziert es einen vorgegebenen Speicherbereich und wendet eine Vielzahl an Operationen auf diesen Bereich an. Da sich die Größe des zu testenden Bereichs vom Nutzer definieren lässt und das Programm in kurzer Zeit sehr viele Seiten überschreibt, eignet es sich gut für eine Messung der Auswirkungen eines speicherintensiven Prozesses auf die Migrationszeit.

Für diese Messung wurde eine spezielle virtuelle Maschine erstellt, der 100 MB Hauptspeicher zugewiesen wurden. Auf dieser virtuellen Maschine lief das gleiche Ubuntu-Abbild, dass auch für die vorhergehenden Tests verwendet worden ist. In der Basiskonfiguration belegte Ubuntu in etwa 11 MB Hauptspeicher, sofern der Nutzer keine Programme ausführte. *Memtester* wurde so ausgeführt, dass es 60 MB alloziert und auf diesen Speicherbereich diverse Operationen anwendete, sodass ein Großteil des verfügbaren Hauptspeichers konstanter Veränderung unterlag. Die Migrationszeit wurde mit dem Befehl *time* gemessen. Wie auch im Fall vorhergehender Messungen misst *time* neben der reinen Migrationszeit auch die Zeit, die der Aufbau der Verbindung über das *ssh* Protokoll kostet. Auch wenn die Zeit für den Verbindungsaufbau nicht zum gewünschten Messergebnis gehört, ist es hinnehmbar, dass diese sich auf das Messergebnis auswirkt, da auch in der Praxis ein Verbindungsaufbau zwischen den Knoten über ein Protokoll wie *ssh* erfolgen muss.

Das Programm *memtester* wurde mit folgendem Befehl innerhalb der virtuellen Maschine gestartet

Listing 4.6:

```
memtester 60M
```

Nach etwa 10 Sekunden wurde dann die Migration gestartet und die Zeit gemessen.

Listing 4.7:

```
time virsh migrate leistungstests_vm qemu+ssh://192.168.1.2/system --live
```

Es sei noch einmal erwähnt, dass das Abbild des Betriebssystems der virtuellen Maschine nicht zwischen den Knoten übertragen wird, da es auf einem im Netz für alle Knoten verfügbaren Sekundärspeicher lag. Bei der hier vorgenommenen Migration wurden lediglich die Seiten des Hauptspeichers der virtuellen Maschine und einige Statusinformationen übertragen. Da die Zeitmessung durch *time* außerhalb einer virtuellen Maschine vorgenommen wird, ist die Zeitabfragenproblematik von virtuellen Maschinen hier nicht existent. Der Befehl *virsh migrate* terminiert, wenn die Live-Migration abgeschlossen ist, sodass die Zeitmessung nicht durch ein verfrühtes oder verspätetes Terminieren verfälscht wird.

Die durchschnittlich benötigte Migrationszeit wurde aus 10 Migrationen ermittelt. Die Migrationszeit einer virtuellen Maschine mit 100 MB zugewiesenem Hauptspeicher, laufendem Ubuntu und ohne vom Nutzer gestartete Prozesse betrug im Mittel 7,4 Sekunden.

Die für eine Live-Migration benötigte Zeit mit laufendem *memtester* unterlag deutlichen Schwankungen und betrug 40 Sekunden im besten sowie etwa 5 Minuten im schlechtesten Fall. Es stellte sich jedoch heraus, dass die virtuelle Maschine nach erfolgreicher Migration nicht mehr reagierte. Gelegentlich wurden Meldungen ausgegeben, dass ein Prozess den Prozessor blockieren würde. Es war jedoch nicht möglich, die Kontrolle über das auf der virtuellen Maschine laufende Gast-Betriebssystem zurückzuerhalten. Ein vor der Live-Migration

eingerichtetes automatisiertes Senden von SIGKILL, einem UNIX-Signal, dass die sofortige Beendigung eines Prozesses erzwingt, an *memtester* nach erfolgter Migration zeigte keine Wirkung und wurde vermutlich gar nicht ausgeführt. Der gleiche Versuch wurde auch mit dem Programm *mbw*, das bereits in Abschnitt 4.2.2 zur Messung des Speicherdurchsatzes verwendet wurde, durchgeführt. Aber auch hier reagierte die virtuelle Maschine spätestens nach der zweiten Live-Migration nicht mehr. Die Live-Migration ist in beiden Fällen also als gescheitert zu betrachten und dementsprechend verbleibt diese Messung ohne endgültiges Ergebnis.

Eine mögliche Ursache für das geschilderte Verhalten könnte eine fehlende Zeitsynchronität der beiden Knoten sein [liv 16]. Es wurde jedoch mittels Network Time Protocol (NTP), einem Protokoll zur Synchronisation von Uhren in Rechnern mit einer Genauigkeit von einigen Millisekunden, sichergestellt, dass die Uhren der beiden Knoten synchron laufen. Die fehlgeschlagenen Live-Migrationen könnten also ein Indikator dafür sein, dass die Live-Migration bei KVM/ARM noch nicht gänzlich ausgereift ist. Mit weniger speicherintensiven Prozessen waren Live-Migrationen jedoch immer erfolgreich, sodass die Live-Migration nicht als gänzlich fehlerhaft bezeichnet werden kann.

4.4. Fazit

Das Ziel dieses Kapitels war es, umfangreiche Leistungsmessungen des in Kapitel 3 erstellten Virtualisierungscluster vorzunehmen. Ursprünglich war vorgesehen, dafür die Software, die in den Arbeiten [Gebh 10] [Lemb 10] [Tsio 10] [Roma 10] [Lind 10] zur Leistungsmessung der Virtualisierung verschiedener Virtualisierer auf x86 Prozessoren verwendet wurde, einzusetzen. Dadurch sollten Vergleichsmöglichkeiten geschaffen und auf die in dieser Literatur erarbeiteten theoretischen Grundlagen aufgebaut werden. Aufgrund von Inkompatibilitäten und Softwarefehlern mussten die Testszenarien in ihrem Umfang jedoch deutlich reduziert oder abgeändert werden. Teilweise musste auf für Leistungsmessungen unübliche Programme wie *dd* zurückgegriffen werden. Dieser Sachverhalt stützt die in Abschnitt 3.3.2 getroffenen Aussagen über die derzeit geringere Softwareverfügbarkeit und fehlerhafte Software auf ARMv7-Plattformen. Aus Zeitgründen musste auf Leistungsmessungen des Virtualisierungsclusters als Gesamtsystem sowie Leistungsmessungen von Praxisanwendungen verzichtet werden.

Dennoch erlauben die Messergebnisse einen ersten Eindruck der Leistungsfähigkeit von Virtualisierungsclustern auf der Basis von ARMv7-Plattformen. Grundsätzlich ist der Wirkungsgrad der Virtualisierung der Komponenten Prozessor und Hauptspeicher mit KVM/ARM mit Wirkungsgraden von über 97% im Vergleich zu Virtualisierern auf der PC-Plattform wettbewerbsfähig. Anders sieht es bei Ein-/Ausgabeanfragen aus. Während der Wirkungsgrad des Sekundärspeichers bei sequenziellen Schreibzugriffen mit 94,81% noch sehr gut ist, sind Lesezugriffe auf diesen mit Wirkungsgraden von um die 50% nicht mehr wettbewerbsfähig. Der Wirkungsgrad von Netzzugriffen ist bei Lesezugriffen mit etwas mehr als 50% ebenfalls nicht wettbewerbsfähig. Ein gemischtes Bild zeigt sich bei Schreibzugriffen auf das Netz. Bei Schreibzugriffen mit kleinen Paketgrößen wird nur ein sehr schlechter Wirkungsgrad von 26,52% erreicht. Bei größeren Paketgrößen werden jedoch deutlich bessere Wirkungsgrade von etwa 90% erreicht.

Zusammengefasst zeigt die Leistungsevaluierung, dass die Virtualisierung mit KVM/ARM bei rechen- und speicherintensiven Prozessen nahezu die Leistung einer nativen Ausführung erreicht. Anwendungen, die viele Ein-/Ausgabeanfragen sowohl auf Sekundärspeicher als auch das Netz ausführen, werden bei einer virtualisierten Ausführung jedoch deutliche Leistungseinbußen aufweisen. Dies gilt insbesondere, wenn der Anteil an Leseoperationen auf diese Komponenten einen Großteil der Operationen der Anwendung ausmacht. Einige Server- und Datenbankanwendungen besitzen einen hohen Anteil an Ein-/Ausgabeanfragen. Bei diesen Anwendungen sind bei einer virtualisierten Ausführung mit KVM/ARM große Leistungseinbußen gegenüber einer nativen Ausführung zu erwarten.

Als kritisch einzustufen sind die gescheiterten Versuche von Live-Migrationen mit in der virtuellen Maschine laufenden speicherintensiven Prozessen, welche in Abschnitt 4.3 behandelt wurden. Wenn eine virtuelle Maschine nach einer Live-Migration nicht mehr funktionsfähig ist, dann sollte die Live-Migration vermieden werden. Ohne Live-Migration entspräche der Virtualisierungscluster aber nicht mehr der in 2.1.3 aufgestellten Definition, da er einem seiner größten Vorteile beraubt wäre. Hier sollte beobachtet werden, ob Live-Migration mit neueren Programmversionen bei speicherintensiven Prozessen künftig bessere Ergebnisse erzielt.

5. Zusammenfassung und Ausblick

Ziel dieser Arbeit war es, einen Virtualisierungscluster auf Basis von ARMv7-Plattformen zu entwerfen, aufzubauen und auf seine Praxistauglichkeit und Leistungsfähigkeit zu untersuchen. Dieses Ziel wurde erreicht. Es wurde ein lauffähiger Virtualisierungscluster aus drei ARMv7-Geräten der Banana Pi Serie entworfen und aufgebaut. Live-Migrationen sind auf diesem Virtualisierungscluster mit Einschränkungen mit KVM/ARM möglich.

Der Entwurfsprozess des Virtualisierungsclusters wurde detailliert beschrieben, wobei mit den wesentlichen theoretischen Grundlagen begonnen wurde. So wurde eine Einführung in die Funktionsweise der Hardwarevirtualisierung auf ARMv7-Prozessoren gegeben sowie einige Besonderheiten des ARMv7-Ökosystems wie beispielsweise der Gerätebaum im Vergleich zu der PC-Plattform erläutert. Die Implementierungen der Virtualisierer Xen ARM und KVM/ARM wurden näher untersucht. Hier ist erwähnenswert, dass die Hardwarevirtualisierung auf ARMv7 sehr gut für die Architektur von Xen ARM geeignet ist, während sie bei KVM/ARM das neu entwickelte Konzept der *Split-mode Virtualization* notwendig macht.

Der Installationsprozess des Virtualisierungsclusters wurde ausführlich dokumentiert. Die dabei entstandenen Instruktionen zur Installation von Xen ARM und KVM/ARM auf einem Banana Pi in den Anhängen A beziehungsweise B können als Grundlage für weiterführende Arbeiten dienen. Während des Installationsprozesses ist ersichtlich geworden, dass Xen ARM derzeit nicht für einen Virtualisierungscluster geeignet ist, da die Unterstützung von Live-Migration fehlt.

Mit dem mit KVM/ARM realisierten Virtualisierungscluster wurden Messungen des Wirkungsgrads der Virtualisierung der Komponenten Prozessor, Hauptspeicher, Sekundärspeicher und Netz vorgenommen. Die Leistungsevaluierung hat ergeben, dass die Virtualisierung mit KVM/ARM im Vergleich zu Virtualisierern auf der PC-Plattform wettbewerbsfähige Wirkungsgrade erzielt. Eine Ausnahme bilden hierbei Lesezugriffe auf Sekundärspeicher und Netz, welche mit Wirkungsgraden von etwa 50% wesentlich ineffizienter als bei anderen Virtualisierern auf der PC-Plattform sind. Damit eignet sich die Virtualisierung mit KVM/ARM vor allem für rechen- oder speicherintensive Prozesse, während bei Anwendungen mit einem großen Anteil von Leseoperationen auf Sekundärspeicher oder Netz größere Leistungseinbußen zu erwarten sind. Neben den Messungen des Wirkungsgrads der Virtualisierung wurden auch die Auswirkungen von speicherintensiven Prozessen auf die Dauer von Live-Migrationen untersucht, hierbei sind jedoch fehlerhafte Live-Migrationen beobachtet worden.

Die Installation und Nutzung des Virtualisierungsclusters zeigte einige Probleme und Herausforderungen von Virtualisierung auf ARMv7-Plattformen auf. Mangelnde Dokumentation und unfertige, nicht verfügbare oder fehlerhafte Software haben den Zeitaufwand für die Installation des Virtualisierungsclusters beträchtlich erhöht und einige Anpassungen erfordert. Sie sind allerdings temporärer Natur und könnten mit der Zeit durch das Erscheinen neuer Softwareversionen und Hardwareplattformen an Bedeutung verlieren. Probleme wie die Unmöglichkeit, hardwarenahe Programme wie Xen universell auf heterogenen Hardwareplattformen mit ARMv7-Prozessoren zu installieren und der durch den Gerätebaum erhöhte administrative Aufwand bei heterogenen Clustern werden jedoch auch auf lange Sicht Bestand haben und die Attraktivität von ARMv7-Plattformen für den Einsatz in Virtualisierungsclustern möglicherweise schmälern.

Derzeit erscheint ein Einsatz von ARMv7-Plattformen in produktiv eingesetzten Virtualisierungsclustern hauptsächlich aufgrund der temporären Probleme, bei denen besonders die gescheiterten Live-Migrationen hervorzuheben sind, nicht empfehlenswert. Für den Einsatz in den Bereichen Lehre und Forschung können Virtualisierungscluster auf der Basis von ARMv7-Plattformen jedoch bereits jetzt geeignet sein. Aufgrund der geringen Anschaffungskosten sowie des geringen Energie- und Platzbedarfs stellen sie sich als Alternative zu der PC-Plattform dar, um grundlegende Einblicke in die Arbeitsweise und Flexibilität von Virtualisierungsclustern zu vermitteln sowie wissenschaftliche Untersuchungen durchzuführen.

Während dieser Arbeit wurden Ansatzpunkte für fortführende Arbeiten identifiziert. Die im Rahmen dieser Arbeit möglichen und vorgenommenen Leistungsmessungen besitzen einen zu geringen Umfang um sichere Rückschlüsse auf den in der Praxis zu erwartenden Wirkungsgrad der Virtualisierung mit KVM/ARM zu ermöglichen. Aus diesem Grund sollten in einiger Zeit Leistungsmessungen mit größerem Umfang durchgeführt werden. Der Wirkungsgrad von anderen Virtualisierern wie Xen ARM könnte ebenfalls untersucht und Vergleiche mit KVM/ARM aufgestellt werden. Da die ARM Virtualization Extensions mit der ARMv8.1-Architektur überarbeitet wurden und speziell Typ 2 Virtualisierer wie KVM/ARM damit einen höheren Wirkungsgrad als auf der ARMv7-Architektur erzielen könnten, wäre hier eine genauere Untersuchung der Unterschiede zwischen den Implementierungen und den daraus möglicherweise resultierenden Leistungsunterschieden interessant. Aber nicht nur der Wirkungsgrad der Virtualisierung ist für Virtualisierungscluster von Bedeutung, sondern auch die Energieeffizienz der Hardware. Es gibt bereits Arbeiten, die die Energieeffizienz von ARMv7-Prozessoren mit der von x86-Prozessoren verglichen haben. Es fehlt jedoch eine Betrachtung der Energieeffizienz im Kontext von Virtualisierungsclustern, wo durch Live-Migration im Vergleich zu herkömmlichen Clustern neue Möglichkeiten zur Reduzierung des Energiebedarfs existieren.

A. Shell-Instruktionen zum Installieren von Xen ARM auf einem Banana Pi

```
# Hier wird beschrieben, wie Xen auf einem Banana Pi M1 installiert und
# konfiguriert wird.
# Die einzelnen Schritte unterscheiden sich für einen Banana Pi R1 (Lamobo R1)
# nicht, mit der Ausnahme, dass Datei-, VM- und Hostnamen angepasst werden
# sollten und dass es notwendig ist, für den Banana Pi R1 einen Linux-Kern, der
# den b53-Treiber für den Switch enthält, zu kompilieren. Der b53-Treiber ist
# noch nicht im Linux-Kern integriert (Stand Version 4.6) und muss manuell
# gepatcht werden. Diese Schritte können die Armbian Skripte übernehmen, dort
# gibt es in compile.sh eine Option, nur den Linux-Kern zu kompilieren.

# Diese Anleitung basiert zum Teil auf:
# https://mirage.io/wiki/xen-on-cubieboard2
# http://www.sourcediver.org/blog/2014/06/19/running-arch-linux-arm-with-xen-on-cubieboard-2

# Compiler für ARM und benötigte Programme auf Debian x86 installieren.
sudo apt-get install gcc-arm-linux-gnueabihf \
        bc build-essential git device-tree-compiler ncurses-dev minicom

# Arbeitsverzeichnis anlegen.
mkdir virt-arm
cd virt-arm

# Variable, die bei Weiterreichen an make Kreuzkompilieren für ARM aktiviert.
CROSS_COMPILE=arm-linux-gnueabihf-
# Variable, die zusätzliche Optionen für make enthält.
MAKEOPTS="-j$(nproc)"

###################################
# -- U-Boot - Bootloader
###################################
# U-Boot Quellcode herunterladen.
git clone git://git.denx.de/u-boot.git
cd u-boot
# Auf stabile Version wechseln.
git checkout tags/v2016.05
# Automatisches Einstellen der Parameter für den Banana Pi
make CROSS_COMPILE=$CROSS_COMPILE Bananapi_config

# Hiernach sollten die Werte
# CONFIG_CPU_V7_HAS_NONSEC=y
# CONFIG_CPU_V7_HAS_VIRT=y
# CONFIG_ARMV7_NONSEC=y
# CONFIG_ARMV7_VIRT=y
# in .config gesetzt sein, damit der Prozessor im für Virtualisierung benötigten
# HYP Modus gestartet wird. Ist dies nicht der Fall, so müssen diese manuell
# gesetzt werden.

# Kompilieren mit der in .config angegebenen Zielplattform.
make CROSS_COMPILE=$CROSS_COMPILE $MAKEOPTS
```

```
# Erstellen einer U-Boot Konfiguration, die Xen mit dom0 startet.

mkdir boot
cd boot
echo \
'# Basierend auf:
# https://raw.githubusercontent.com/mirage/xen-arm-builder/master/boot/boot-cubieboard2.cmd

# Die Adressen sind nur für Systeme mit 1GB Hauptspeicher geeignet und müssen
# gegebenenfalls angepasst werden.

# Ende des Hauptspeichers:         0x80000000
# Xen relocate addr                0x7fe00000

setenv fdt_addr       0x7ec00000 # 2M
setenv fdt_high       0xffffffff # Lade fdt auf der Stelle anstatt zu rellokieren

# Lade Xen und Linux in die obere Hälfte des Hauptspeichers, da ansonsten ein
# Fehler beim Allozieren der 512 MB für dom0 auftritt.
setenv kernel_addr_r  0x6ee00000
setenv xen_addr_r     0x6ea00000 # 2M

# sunxi-common.h gibt diese Adressen an:
# CONFIG_SYS_TEXT_BASE  0x4a000000 - Adresse des Codes von U-Boot
# CONFIG_SYS_SDRAM_BASE 0x40000000 - Anfangsadresse des Hauptspeichers

# Lade Xen nach ${xen_addr_r}.
fatload mmc 0 ${xen_addr_r} /xen
# Setze einige Parameter für den Bootvorgang
# (serielle Konsole und zu reservierender Speicher für dom0)
setenv bootargs \
"console=dtuart␣dtuart=/soc@01c00000/serial@01c28000␣dom0_mem=512M,max:512M"

# Lade Gerätebaum nach ${fdt_addr}
fatload mmc 0 ${fdt_addr} /sun7i-a20-bananapi.dtb
fdt addr ${fdt_addr} 0x40000
fdt resize
fdt chosen
fdt set /chosen \#address-cells <1>
fdt set /chosen \#size-cells <1>

# Lade Linux-Kern nach ${kernel_addr_r}
fatload mmc 0 ${kernel_addr_r} /vmlinuz

fdt mknod /chosen module@0
fdt set /chosen/module@0 compatible "xen,linux-zimage" "xen,multiboot-module"
fdt set /chosen/module@0 reg <${kernel_addr_r} 0x${filesize} >
fdt set /chosen/module@0 bootargs \
"console=hvc0␣ro␣root=/dev/mmcblk0p2␣rootwait␣clk_ignore_unused"

# Starte Xen
bootz ${xen_addr_r} - ${fdt_addr}
' > boot.cmd

# Achtung: dom0 bekommt hier 512 MB Hauptspeicher zugewiesen, um genug Raum für
# speicherintensive Operationen zu lassen.
# Durch Setzen von autoballoon=on in /etc/xen/xl.conf, xl wird der für dom0
# reservierte Speicher dynamisch reduziert, falls möglich.

# Diese Textdatei muss nun in eine Binärdatei kompiliert werden, die von U-Boot
# verwendet werden kann.
```

```
../tools/mkimage -C none -A arm -T script -d "boot.cmd" "boot.scr"

cd ../../

##################################
# -- Linux-Kern - Betriebssystem
##################################
# Vollen Support für den Banana Pi gibt es nur in Linux-Kern Version 3.4.
# Diese unterstützt aber weder Xen noch KVM. Deswegen ist es notwendig, einen
# neueren Linux-Kern selbst zu kompilieren. Nicht alle Funktionen des Banana Pi
# werden dann unterstützt. Es ist ratsam, die aktuelle stabile Version von
# Linux zu nutzen, da die Unterstützung für den Banana Pi laufend besser wird.

git clone https://github.com/torvalds/linux
cd linux
git checkout tags/v4.6
# Allgemeine Konfigurationsdatei für ARMv7 erstellen.
make ARCH=arm CROSS_COMPILE=$CROSS_COMPILE multi_v7_defconfig

# Konfiguration manuell anpassen, es müssen folgende Optionen gesetzt werden:
# In Klammern steht die Schnelltaste, mit der zu der Option gesprungen werden
# kann.
# (S)ystem Type -> (S)upport for the Large Physical Adress Extension
# (K)ernel Features -> (X)en guest support on ARM
# N(e)tworking support -> N(e)tworking options -> 802.1(d) Ethernet Bridging
# (D)evice Drivers -> M(u)ltiple devices driver support (RAID and LVM)
#                               -> (D)evice mapper support
# (D)evice Drivers -> (B)lock devices -> (X)en block-device backend driver
# (D)evice Drivers -> N(e)twork device support -> (X)en backend network device
# (V)irtualization
# (V)irtualization -> H(o)st kernel Accelerator for virtio net
make ARCH=arm CROSS_COMPILE=$CROSS_COMPILE menuconfig

# Linux-Kern für ARM kompilieren.
make ARCH=arm CROSS_COMPILE=$CROSS_COMPILE zImage dtbs modules $MAKEOPTS

cd ..

##################################
# -- Xen - Hypervisor
##################################
git clone git://xenbits.xen.org/xen.git
cd xen
git checkout tags/RELEASE-4.6.1
# Nun muss ein Patch angewandt werden, um ein fehlerfreies Starten von Xen zu
# ermöglichen. Der Patch sorgt dafür, dass Schreibvorgänge auf gewisse Register
# ignoriert werden, da diese von Xen noch nicht unterstützt werden.
echo 'diff --git a/xen/arch/arm/vgic-v2.c b/xen/arch/arm/vgic-v2.c
index cb1fbb8..b35fa10 100644
--- a/xen/arch/arm/vgic-v2.c
+++ b/xen/arch/arm/vgic-v2.c
@@ -335,11 +335,12 @@ static int vgic_v2_distr_mmio_write(struct vcpu *v, mmio_info_t *info)
          return 0;

      case GICD_ICACTIVER ... GICD_ICACTIVERN:
-         if ( dabt.size != DABT_WORD ) goto bad_width;
+         /*if ( dabt.size != DABT_WORD ) goto bad_width;
          printk(XENLOG_G_ERR
                 "%pv: vGICD: unhandled word write %#"PRIregister" to ICACTIVER%d\n",
                 v, *r, gicd_reg - GICD_ICACTIVER);
-         return 0;
```

```
+         return 0;*/
+         goto write_ignore_32;

      case GICD_ITARGETSR ... GICD_ITARGETSR7:
          /* SGI/PPI target is read only */
diff --git a/xen/arch/arm/vgic-v3.c b/xen/arch/arm/vgic-v3.c
index f1c482d..4fd1cdf 100644
--- a/xen/arch/arm/vgic-v3.c
+++ b/xen/arch/arm/vgic-v3.c
@@ -427,11 +427,12 @@ static int __vgic_v3_distr_common_mmio_write(const char *name, struct vcpu
          return 0;

      case GICD_ICACTIVER ... GICD_ICACTIVERN:
-         if ( dabt.size != DABT_WORD ) goto bad_width;
+         /*if ( dabt.size != DABT_WORD ) goto bad_width;
          printk(XENLOG_G_ERR
                 "%pv: %s: unhandled word write %#"PRIregister" to ICACTIVER%d\n",
                 v, name, *r, reg - GICD_ICACTIVER);
-         return 0;
+         return 0;*/
+         goto write_ignore_32;

      case GICD_IPRIORITYR ... GICD_IPRIORITYRN:
          if ( dabt.size != DABT_BYTE && dabt.size != DABT_WORD ) goto bad_width;
' > vgic.patch

patch -p1 < vgic.patch

# Xen für ARM kompilieren.
make dist-xen XEN_TARGET_ARCH=arm32 CROSS_COMPILE=$CROSS_COMPILE $MAKEOPTS


# -- SD-Karte vorbereiten.

# !== VORSICHT ==!
# Die nachfolgenden Befehle setzen voraus, dass der Pfad zu der zu
# beschreibenden SD-Karte /dev/mmcblk0 ist. Es ist unbedingt zu prüfen, ob
# dieser Pfad korrekt ist, denn sonst droht der komplette Datenverlust auf
# einem anderen Datenträger.

# Partitionstabelle anlegen:
# Die erste Partition sollte erst ab Sektor 2048 (1MB) beginnen, um Platz für
# die Bootloader Das U-Boot SPL und Das U-Boot zu lassen.
# 1. 16M VFAT - für U-Boot, Linux-Kern und Xen
# 2. 4GB ext4 - für dom0
# 3. Rest als LVM für domU etc.
sudo blockdev --rereadpt /dev/mmcblk0
sudo sfdisk /dev/mmcblk0 <<EOT
1M,16M,c
,4G,L
,,8e
EOT

# Partitionen formatieren (s.o.).
sudo mkfs.vfat /dev/mmcblk0p1
sudo mkfs.ext4 /dev/mmcblk0p2

# -- U-Boot auf SD-Karte schreiben, der Allwinner A20 SoC bootet vom 8. Block
# der SD-Karte. u-boot-sunxi-with-spl.bin enthält sowohl den minimalen SPL, als
# auch das vollständige U-Boot, das automatisch vom SPL geladen wird.
sudo dd if=u-boot/u-boot-sunxi-with-spl.bin of=/dev/mmcblk0 bs=1024 seek=8
```

```
# -- U-Boot Skript, Linux und Xen auf SD-Karte schreiben.
# Aufsetzpunkt einrichten
sudo mkdir /mnt/mmc1
# VFAT Partition einhängen und benötigte Dateien kopieren.
sudo mount /dev/mmcblk0p1 /mnt/mmc1
# Skript für U-Boot kopieren.
sudo cp u-boot/boot/boot.scr /mnt/mmc1/
# Linux-Kern kopieren.
sudo cp linux/arch/arm/boot/zImage /mnt/mmc1/vmlinuz
# Gerätebaum (dtb) für Banana Pi kopieren.
sudo cp linux/arch/arm/boot/dts/sun7i-a20-bananapi.dtb /mnt/mmc1/
# Xen Binärdatei kopieren.
sudo cp xen/xen/xen /mnt/mmc1/


# -- rootfs - Minimales Ubuntu Abbild auf SD-Karte schreiben.
wget -c \
http://releases.linaro.org/ubuntu/images/developer/latest/linaro-vivid-developer-20151215-714.ta

# ext4 Partition der SD-Karte einhängen.
sudo mkdir /mnt/mmc2
sudo mount /dev/mmcblk0p2 /mnt/mmc2

# Heruntergeladenes Ubuntu Abbild entpacken und Dateien kopieren.
sudo tar zxf linaro-vivid-developer-20151215-714.tar.gz -C /mnt/mmc2 --strip 1
# Warten, bis Dateien auf SD-Karte geschrieben wurden.
sync

# Linux-Kern für domU kopieren.
sudo mkdir /mnt/mmc2/root
sudo cp /mnt/mmc1/vmlinuz /mnt/mmc2/root/zImage

# VFAT Partition aushängen.
sudo umount /mnt/mmc1

# Module für Linux-Kern installieren.
cd linux
sudo make ARCH=arm CROSS_COMPILE=$CROSS_COMPILE INSTALL_MOD_PATH='/mnt/mmc2' \
        modules_install
cd ..


# Netzbrücke in dom0 einhängen, die Kommunikation zwischen domUs ermöglicht.
echo '
auto lo
iface lo inet loopback

auto eth0
iface eth0 inet dhcp
  up ip link set eth0 up

# Netzbrücke für Xen VM Instanzen.
auto br0
iface br0 inet dhcp
  bridge_ports eth0
' | sudo tee -a /mnt/mmc2/etc/network/interfaces


# Dateisystemtabelle anlegen.
echo '/dev/mmcblk0p2 / ext4 rw,defaults 0 0
```

```
none /tmp tmpfs defaults 0 0' | sudo tee /mnt/mmc2/etc/fstab

# Machine-id beim nächsten Boot generieren lassen.
sudo touch /mnt/mmc2/etc/machine-id

# ext4 Partition aushängen.
sudo umount /mnt/mmc2
# Sicherstellen, dass alle Änderungen aus dem Puffer auf die SD-Karte
# geschrieben werden
sync

# -----------
# Zwischenstand: Mit diesen Befehlen ist ein Abbild auf der SD-Karte entstanden,
# das bereits bootbar ist. Xen wird automatisch mit dom0 gestartet. Für einen
# produktiven Einsatz reicht dies aber noch nicht, es müssen nicht nur domU
# Gäste erstellt werden, sondern auch noch einiges konfiguriert werden.
# -----------

# Nun kann über die serielle Konsole mit Xen kommuniziert werden.
#
# Unter Serial Port Setup folgende Einstellungen vornehmen:
# Serial Device: /dev/ttyUSB0   (oder anderer Pfad zum Transceiver)
# Bps/Par/Bits: 115200 8N1
# Hardware Flow Control: No
sudo minicom -s
# Nutzer zur dialout Gruppe hinzufügen, damit er auf die serielle Konsole
# zugreifen darf.
sudo usermod -a -G dialout $user
# Nutzer neu einloggen, damit Änderungen übernommen werden,
# anschließend minicom starten.
minicom -c on -R utf-8

# Nun sollte der Transceiver mit dem Banana Pi verbunden werden, siehe dazu auch
# https://linux-sunxi.org/Banana_Pi#Locating_the_UART .
# Dann kann die SD-Karte in den Banana Pi gesteckt und dieser gestartet werden.
# Jetzt sollten in Minicom Ausgaben von U-Boot, Xen und dom0 sichtbar werden.

# Nach erfolgreichem Bootvorgang kann mit der Konfiguration begonnen werden.
# Passwort for root Konto ändern.
passwd

# Hostnamen anpassen.
echo dom0s1 > /etc/hostname
# hostname bei 127.0.1.1 durch o.g. ersetzen.
vi /etc/hosts
# DNS Server in resolv.conf eintragen.
echo "37.235.1.174" > /etc/resolv.conf

# Paketquellen updaten.
apt-get update

# ssh Server und screen installieren.
apt-get install openssh-server screen
cd
mkdir .ssh
vi .ssh/authorized_keys # eigenen ssh Key eintragen

# Ab hier empfiehlt es sich, statt über die serielle Konsole per ssh auf das
# System zuzugreifen.

# -- Xen Toolstack
```

```
# Der Toolstack kann einfach aus den Paketquellen installiert werden.
apt-get install xen-utils-4.6
# Eintrag des xenfs in Dateisystemtabelle
echo 'none /proc/xen xenfs defaults 0 0' >> /etc/fstab
# Anschließend sollte das folgende Kommando eine Liste der laufenden VM ausgeben
xl list

# LVM für domU einrichten.
apt-get install lvm2
pvcreate /dev/mmcblk0p3
vgcreate vg0 /dev/mmcblk0p3

# Neue LVM Partition einrichten und formatieren.
lvcreate -L 2G vg0 --name linux-guest-1
mkfs.ext4 /dev/vg0/linux-guest-1

# -- Abbild für Gast-Betriebssystem errichten.
# Benötigte Software installieren.
apt-get install debootstrap
# Partition einhängen und Betriebssystem installieren.
mount /dev/vg0/linux-guest-1 /mnt
debootstrap --arch armhf vivid /mnt

# Mit chroot Konfiguration des Gast-Betriebssystems anpassen.
chroot /mnt

# Passwort ändern.
passwd

# Hostnamen anpassen.
echo domUs1 > /etc/hostname

# Netzzugang einrichten.
echo '
auto eth0
iface eth0 inet static
  address 192.168.1.201
  netmask 255.255.255.0' >> /etc/network/interfaces

# Dateisystemtabelle (fstab) einrichten.
echo '/dev/xvda       / ext4   rw,relatime,data=ordered       0 1
' >> /etc/fstab

# ssh Zugang einrichten.
mkdir -m 0700 /root/.ssh
vi /root/.ssh/authorized_keys # eigenen ssh Key einfügen

# chroot verlassen und Abbild aushängen.
exit
umount /mnt

# domU VM definieren:
echo 'kernel = "/root/zImage"
memory = 256
name = "domUs1"
vcpus = 2
serial="pty"
disk = [ "phy:/dev/vg0/linux-guest-1,xvda,w" ]
vif = ["bridge=br0"]
extra = "console=hvc0␣xencons=tty␣root=/dev/xvda"
' > domUs1
```

```
# Netzbrücke für VMs einrichten.
apt-get install bridge-utils
brctl addbr br0
# dhcp muss nun bei eth0 zu manual geändert werden
vi etc/network/interfaces

# VM namens guest1 starten.
xl create domUs1 -c

# Die domU Konsole kann mit Strg+] ausgeblendet werden und mit
# "xl console <domU id>" wieder aufgerufen werden.

# Die Installation von Xen ist nun abgeschlossen und die virtuellen Maschinen
# können nach Belieben verwendet werden.
```

B. Shell-Instruktionen zum Installieren von KVM/ARM auf einem Banana Pi

```
# Hier wird beschrieben, wie KVM auf einem Banana Pi M1 installiert und
# konfiguriert wird, sodass es in einem Virtualisierungscluster verwendet werden
# kann.
# Die einzelnen Schritte unterscheiden sich für einen Banana Pi R1 (Lamobo R1)
# nicht, mit der Ausnahme, dass Datei-, VM- und Hostnamen angepasst werden
# sollten und dass es notwendig ist, für den Banana Pi R1 mit den Armbian build
# tools ein eigenes Abbild zu erstellen, da sich die Hardware unterscheidet.

# Zunächst sollte auf einem PC ein bootbares Abbild von Ubuntu 16.04 erstellt
# werden. Dazu können die Armbian build tools
# (https://github.com/igorpecovnik/lib) verwendet werden. Vor dem Ausführen der
# Skripte sollte in compile.sh KERNEL_CONFIGURE auf "yes" gesetzt werden und in
# menuconfig für den Linux-Kern sollten folgende Optionen gesetzt werden:
# (Schnelltaste zum Springen zur Option in Klammern)
# (S)ystem Type -> (S)upport for the Large Physical Adress Extension
# N(e)tworking support -> N(e)tworking options -> 802.1(d) Ethernet Bridging
# (D)evice Drivers -> M(u)ltiple devices driver support (RAID and LVM)
#                       -> (D)evice mapper support
# (V)irtualization
# (V)irtualization -> Kernel-based Virtual Machine (KVM) support
# (V)irtualization -> H(o)st kernel Accelerator for virtio net

# Für die VMs muss zusätzlich ein spezieller Linux-Kern kompiliert
# werden.
# Dazu kann der von den Armbian Skripten heruntergeladene Linux-Kern-Quellcode
# verwendet werden oder der Quellcode separat heruntergeladen werden.
# Der Linux-Kern soll die Versatile Express A15 Referenzplattform von ARM
# unterstützen. Dieser Schritt ist notwendig, um eine einheitliche
# Hardwareplattform für die VMs zu bieten, da es ansonsten bei heterogener
# Hardware wie in unserem Cluster zu Inkompatibilitäten bei der Migration
# kommen könnte.

# Hier wird angenommen, dass das aktuelle Arbeitsverzeichnis der Konsole der
# Basisordner des Quellcodes des Linux-Kerns ist.

# Variable, die bei Weiterreichen an make Kreuzkompilieren für ARM aktiviert.
CROSS_COMPILE=arm-linux-gnueabihf-
# Variable, die zusätzliche Optionen für make enthält.
MAKEOPTS="-j$(nproc)"

# Basiskonfiguration für Versatile Express Plattformen laden.
make ARCH=arm CROSS_COMPILE=$CROSS_COMPILE vexpress_defconfig

# Folgende Optionen sollten zusätzlich in menuconfig aktiviert sein.
# (Schnelltaste zum Springen zur Option in Klammern)
# (G)eneral setup -> (C)onfigure standard kernel features (expert users)
# (G)eneral setup -> (o)pen by fhandle syscalls
# (E)nable the block layer -> (S)upport for large (2TB+) block devices and files
# (V)irtualization -> H(o)st kernel accelerator for virtio net
make ARCH=arm CROSS_COMPILE=$CROSS_COMPILE menuconfig
```

```
# Linux-Kern für ARM kompilieren.
make ARCH=arm CROSS_COMPILE=$CROSS_COMPILE zImage dtbs $MAKEOPTS

# Linux-Kern und Gerätebaum (dtb) für Gast-Betriebssysteme kopieren.
# Hier wird angenommen, dass das mit Armbian erstellte Abbild bereits auf die
# SD-Karte geschrieben wurde und der Pfad zur SD-Karte /dev/mmcblk0 ist.
# Vor dem Kopieren muss die SD-Karte mit dem von den Armbian Skripten erstelltem
# Abbild mindestens einmal im Banana Pi eingesetzt worden sein, denn beim ersten
# Start wird die Partitionsgröße an den gesamten Speicherplatz der SD-Karte
# angepasst.
sudo mkdir /mnt/mmc1
sudo mount /dev/mmcblk0p1 /mnt/mmc1
sudo cp arch/arm/boot/zImage /mnt/mmc1/root/vexpress-a15-zImage
sudo cp arch/arm/boot/dts/vexpress-v2p-ca15-tc1.dtb /mnt/mmc1/root
sudo umount /mnt/mmc1

# Nun kann die SD-Karte in den Banana Pi M1 gesteckt und dieser gestartet
# werden.
# Nach erfolgreichem ersten Boot von Armbian kann sofort damit begonnen werden,
# die Umgebung für Virtualisierung vorzubereiten.

# Hostnamen anpassen.
echo kvm_s1 > /etc/hostname
sed -i 's/bananapi/kvm_s1/g' /etc/hosts

# Benötigte Programme und Werkzeuge installieren.
# Dafür sollte der Banana Pi an ein Netz mit Internetzugang und DHCP Server
# gehängt werden oder die Netzkonfiguration muss manuell angepasst werden, damit
# ein Internetzugang möglich ist.
apt-get update
apt-get install kpartx debootstrap bridge-utils qemu-kvm qemu-utils \
        libvirt-bin virtinst

# Nach der Installation ist ein Neustart notwendig.
shutdown -r now

# Zeigt der folgende Befehl eine leere Liste an laufenden VMs an, so ist soweit
# alles korrekt konfiguriert.
virsh list

# -- Abbild eines Gast-Betriebssystems als Grundlage für VMs erstellen.

# Zunächst Datei erstellen, die Abbild enthalten soll.
qemu-img create -f qcow2 ubuntu_16.04-guest1.qcow 2G

# Abbild einhängen.
# Dazu Network Block Device Treiber Modul laden.
modprobe nbd
# Abbild als Network Block Device einhängen.
qemu-nbd -c /dev/nbd0 ubuntu_16.04-guest1.qcow

# Abbild partitionieren.
sfdisk --force /dev/nbd0 <<EOT
,,L
EOT

# Mapping-Device für Partition erstellen.
kpartx -a /dev/nbd0

# Partition des Abbilds als ext4 formatieren.
```

```
mkfs.ext4 /dev/mapper/nbd0p1

# Soeben formatierte Partition einhängen.
mkdir /mnt/img
mount -t ext4 /dev/mapper/nbd0p1 /mnt/img

# Mit debootstrap ein Gast-Betriebssystem installieren.
debootstrap --arch armhf xenial /mnt/img

# Mit chroot Konfiguration des Gast-Betriebssystems anpassen.
chroot /mnt/img

# Passwort einrichten.
passwd

# Grundlegende Software installieren.
apt-get install openssh-server openssh-client
# Hostnamen anpassen.
echo 'kvm_s1_1' > /etc/hostname

# ssh Zugang einrichten.
mkdir -p -m 0700 /root/.ssh
vi /root/.ssh/authorized_keys # eigenen ssh Key einfügen

# Netzzugang einrichten.
echo '
auto eth0
iface eth0 inet static
  address 192.168.1.201
  netmask 255.255.255.0
' >> /etc/network/interfaces

# chroot verlassen.
exit
umount /mnt/img
qemu-nbd -d /dev/nbd0

# Das Abbild kann nun gesichert und als Grundlage für alle weiteren VMs
# verwendet werden.
tar -czf ubuntu_16.04-guest.tar.gz ubuntu_16.04-guest1.qcow

# Im Host-Betriebssysteme eine Netzbrücke für KVM-Gäste einrichten.
# Nach diesem Schritt ist ohne manuelle Konfiguration keine Verbindung per DHCP
# mehr möglich.
ifdown eth0

echo '
auto br0
iface br0 inet static
  address 192.168.1.2
  netmask 255.255.255.0
  bridge_ports eth0' > /etc/network/interfaces

ifup br0

# Hostnamen in Host-Datei eintragen, dies wird für Live-Migration benötigt.
echo '192.168.1.1 master
192.168.1.2 kvm_s1
192.168.1.3 kvm_s2
192.168.1.101 kvm_m_1
192.168.1.201 kvm_s1_1
```

```
192.168.1.221 kvm_s2_1' >> /etc/hosts

# -- VM starten
# Vorher Lesezugriffe für libvirt-qemu Nutzer auf HOME-Verzeichnis von root
# erlauben.
setfacl -m u:libvirt-qemu:r-x /root/

# Vorbereitetes Abbild umbenennen.
mv ubuntu_16.04-guest1.qcow ubuntu_16.04-slave1_1.qcow

# VM definieren und starten.
virt-install \
        --name ubuntu_16.04-slave1_1\
        --machine vexpress-a15 --cpu host \
        --vcpus=1 --ram=256 \
        --accelerate \
        --os-type=linux \
        --boot kernel=/root/vexpress-a15-zImage,
               dtb=/root/vexpress-v2p-ca15-tc1.dtb,
               kernel_args="console=ttyAMA0␣rw␣root=/dev/vda1" \
        --disk /root/ubuntu_16.04-slave1_1.qcow,format=qcow2,bus=virtio --import \
        --nographics \
        --network bridge:br0,model=virtio

# Die Konsole einer laufenden VM kann mit Strg+] ausgeblendet werden und mit
# "virsh console <VM id>" wieder aufgerufen werden.

# Nun kann, sofern eine Verbindung zum Zentralknoten besteht, bereits das
# Abbild im laufenden Betrieb migriert werden.

# Die erste Live-Migration einer VM überträgt das komplette Abbild.
virsh migrate ubuntu_16.04-slave1_1 qemu+ssh://192.168.1.1/system \
        --live --verbose --copy-storage-all

# Ab der zweiten Migration kann ein inkrementelles Übertragen des Abbilds
# genutzt werden, um die Migrationszeit zu verkürzen.
virsh migrate ubuntu_16.04-slave1_1 qemu+ssh://192.168.1.1/system \
        --live --verbose --copy-storage-inc

# -----------
# Zwischenstand: Live-Migration ist bereits möglich, aber dazu muss das Abbild
# einer VM übertragen werden. Im Folgenden wird erläutert, wie auf dem Banana Pi
# R1, dem Zentralknoten des Virtualisierungsclusters, eine SSD über das NFS-
# Protokoll im Netz als Speicher für VM-Abbilder zur Verfügung gestellt werden
# kann.
# -----------

# Hier wird angenommen, dass der Banana Pi R1 nach den vorhergehenden
# Instruktionen eingerichtet worden ist und eine SSD an den SATA-Anschluss
# angeschlossen ist. Der Pfad zur SSD ist hier /dev/sda und der Hostname des R1
# ist "master".

# NFS Server installieren.
apt-get install nfs-kernel-server

# SSD partitionieren.
sfdisk /dev/sda <<EOT
,,L
EOT

# Partition formatieren.
```

```
mkfs.ext4 /dev/sda1

# Dateisystemtabelle anpassen.
# /nfs soll für alle ans Netz angeschlossene Geräte verfügbar sein.
# /nfs/qemu-img soll die Abbilder von VMs beinhalten.
mkdir -p /nfs/qemu-img
echo '/dev/sda1 /nfs ext4 defaults,rw 0 0' >> /etc/fstab
echo \
  'master:/nfs/qemu-img /mnt/qemu-img nfs rsize=8192,wsize=8192,timeo=14,intr' \
  >> /etc/fstab

# Zuvor erstelltes VM-Abbild nach /nfs/qemu-img entpacken.
tar xf ubuntu_16.04-guest.tar.gz -C /nfs/qemu-img/
mv /nfs/qemu-img/ubuntu_16.04-guest.qcow /nfs/qemu-img/ubuntu_16.04-master1.qcow

# /nfs Verzeichnis exportieren.
echo '
/nfs/    *(rw,sync,no_root_squash)
' >> /etc/exports
exportfs -a

# Netz des Banana Pi R1 konfigurieren.
echo '
auto lo
iface lo inet loopback

auto eth0.101
iface eth0.101 inet manual
    pre-up swconfig dev eth0 set reset 1
    pre-up swconfig dev eth0 set enable_vlan 1
    pre-up swconfig dev eth0 vlan 101 set ports '3 8t'
    pre-up swconfig dev eth0 set apply 1

auto eth0.102
iface eth0.102 inet manual
    pre-up swconfig dev eth0 vlan 102 set ports '0 1 2 4 8t'
    pre-up swconfig dev eth0 set apply 1

auto br0
iface br0 inet static
bridge_ports eth0.102 eth0.101
    address 192.168.1.1
    netmask 255.255.255.0
' > /etc/network/interfaces


# NFS-Dienst starten.
service nfs-kernel-server start


#########################################################
# Konfiguration für alle anderen Knoten (Banana Pi M1)
#########################################################

# Benötigte NFS-Software installieren.
apt-get install nfs-common

# Dateisystemtabelle anpassen, sodass /nfs/qemu-img Verzeichnis des Servers
# automatisch als /mnt/qemu-img eingehängt wird, sofern es verfügbar ist.
mkdir -p /mnt/qemu-img
echo \
```

```
  'master:/nfs/qemu-img /mnt/qemu-img nfs rsize=8192,wsize=8192,timeo=14,intr' \
  >> /etc/fstab

# Ist master:/nfs/qemu-img zur Bootzeit nicht verfügbar gewesen, so kann es
# mit folgendem Befehl nachträglich eingehängt werden.
mount master:/nfs/qemu-img

# VM mit VM-Abbild auf SSD definieren und starten.
virt-install \
        --name ubuntu_16.04-slave1_1 \
        --machine vexpress-a15 --cpu host \
        --vcpus=1 --ram=256 \
        --accelerate \
        --os-type=linux \
        --boot kernel=/mnt/qemu-img/vexpress-a15-zImage,
            dtb=/mnt/qemu-img/vexpress-v2p-ca15-tc1.dtb,
            kernel_args="console=ttyAMA0␣rw␣root=/dev/vda1" \
        --disk /mnt/qemu-img/ubuntu_16.04-slave1_1.qcow,
            format=qcow2,bus=virtio,cache=none --import \
        --nographics \
        --network bridge:br0,model=virtio

# Nun ist die Konfiguration des Virtualisierungscluster abgeschlossen. Die
# soeben definierte und dabei automatisch gestartete VM kann im laufenden
# Betrieb nach Belieben zwischen den Knoten migriert werden und wird dabei immer
# über das NFS-Protokoll auf das zentral gespeicherte Abbild der VM zugreifen.
```

Literaturverzeichnis

[AdAg 06] ADAMS, KEITH und OLE AGESEN: *A comparison of software and hardware techniques for x86 virtualization*. ACM SIGOPS Operating Systems Review, 40(5):2–13, 2006.

[ÅFNK 11] ÅSBERG, MIKAEL, NILS FORSBERG, THOMAS NOLTE und SHINPEI KATO: *Towards real-time scheduling of virtual machines without kernel modifications*. In: *Emerging Technologies & Factory Automation (ETFA), 2011 IEEE 16th Conference on*, Seiten 1–4. IEEE, 2011.

[amd 16] *AMD Opteron A-Series Processors*. `https://www.amd.com/en-us/products/server/opteron-a-series`, 2016. [Online; abgerufen am 21.7.2016].

[arm 14] *ARMv7 Reference Manual*. `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dai0425/BABCDDJG.html`, 2014.

[arm 16] *Armbian build tools*. `https://github.com/igorpecovnik/lib`, 2016. [Online; abgerufen am 21.7.2016].

[BaBu 99] BAKER, MARK und RAJKUMAR BUYYA: *Cluster computing: the commodity supercomputer*. Software-Practice and Experience, 29(6):551–76, 1999.

[BDF+ 03] BARHAM, PAUL, BORIS DRAGOVIC, KEIR FRASER, STEVEN HAND, TIM HARRIS, ALEX HO, ROLF NEUGEBAUER, IAN PRATT und ANDREW WARFIELD: *Xen and the art of virtualization*. In: *ACM SIGOPS Operating Systems Review*, Band 37, Seiten 164–177. ACM, 2003.

[BMS 13] BLEM, EMILY, JAIKRISHNAN MENON und KARTHIKEYAN SANKARALINGAM: *Power struggles: Revisiting the RISC vs. CISC debate on contemporary ARM and x86 architectures*. In: *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, Seiten 1–12. IEEE, 2013.

[bpi 14a] *Banana Pi Product Specification*. `http://www.lemaker.org/product-bananapi-specification.html`, 2014. [Online; abgerufen am 14.7.2016].

[bpi 14b] *Banana Pi R1 Product Specification*. `http://www.banana-pi.org/r1.html`, 2014. [Online; abgerufen am 14.7.2016].

[Che 00] *RISC architecture*. `https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/risccisc/`, 2000. [Online; abgerufen am 6.7.2016].

[Danc 09] DANCIU, VITALIAN A: *Host Virtualization: A Taxonomy of Management Challenges.* In: *GI Jahrestagung*, Seiten 2596–2610, 2009.

[DaNi 13] DALL, CHRISTOFFER und JASON NIEH: *KVM/ARM: Experiences building the Linux arm hypervisor*. 2013.

[DaNi 14] DALL, CHRISTOFFER und JASON NIEH: *KVM/ARM: the design and implementation of the Linux ARM hypervisor*. In: *ACM SIGPLAN Notices*, Band 49, Seiten 333–348. ACM, 2014.

[deb 16] *Debian build demon statistics*. `https://buildd.debian.org/stats/`, 2016. [Online; abgerufen am 22.7.2016].

[DgFKL 11] DANCIU, VITALIAN A, NILS GENTSCHEN FELDE, DIETER KRANZLMÜLLER und TOBIAS LINDINGER: *High-performance aspects in virtualized infrastructures*. In: *2010 4th International DMTF Academic Alliance Workshop on Systems and Virtualization Management (SVM 2010). Institute of Electrical and Electronics Engineers*, Band 4, 2011.

[DLL+ 16] DALL, CHRISTOFFER, SHIH-WEI LI, JIN TACK LIM, JASON NIEH und GEORGIOS KOLOVENTZOS: *ARM Virtualization: Performance and Architectural Implications*. 2016.

[FMMG 08] FENN, MICHAEL, MICHAEL A MURPHY, JIM MARTIN und SEBASTIEN GOASGUEN: *An evaluation of KVM for use in cloud computing*. In: *Proc. 2nd International Conference on the Virtual Computing Initiative, RTP, NC, USA*, 2008.

[Furb 00] FURBER, STEPHEN BO: *ARM system-on-chip architecture*. pearson Education, 2000.

[Gebh 10] GEBHARDT, BASTIAN: *Empirische Identifikation von Parametern mit Einfluss auf die Effizienz von Virtualisierung am Beispiel von Xen*, 2010.

[goo 16] *Meet Chromebooks*. `https://www.google.com/chromebook/`, 2016. [Online; abgerufen am 21.7.2016].

[HAN 99] HEGERING, H.-G., S. ABECK und B. NEUMAIR: *Integrated Management of Networked Systems – Concepts, Architectures and their Operational Application*. Morgan Kaufmann Publishers, ISBN 1-55860-571-1, 1999. 651 p.

[hdp 16] *hdparm man page*. `http://manpages.ubuntu.com/manpages/precise/man8/hdparm.8.html`, 2016. [Online; abgerufen am 7.8.2016].

[HSH+ 08] HWANG, JOO-YOUNG, SANG-BUM SUH, SUNG-KWAN HEO, CHAN-JU PARK, JAE-MIN RYU, SEONG-YEOL PARK und CHUL-RYUN KIM: *Xen on ARM: System virtualization using Xen hypervisor for ARM-based secure mobile phones*. In: *2008 5th IEEE Consumer Communications and Networking Conference*, Seiten 257–261. IEEE, 2008.

[IJJ 09] ISEN, CIJI, LIZY K JOHN und EUGENE JOHN: *A tale of two processors: Revisiting the RISC-CISC debate*. In: *SPEC Benchmark Workshop*, Seiten 57–76. Springer, 2009.

[Inc. 08] INC., VMWARE: *Timekeeping in VMware Virtual Machines*, 2008.

[int 16] *Integrity Multivisor Internetseite*. `http://www.ghs.com/products/rtos/integrity_virtualization.html`, 2016. [Online; abgerufen am 25.7.2016].

[jai 16] *Siemens Jailhouse Internetseite*. `https://github.com/siemens/jailhouse`, 2016. [Online; abgerufen am 25.7.2016].

[KKL+ 07] KIVITY, AVI, YANIV KAMAY, DOR LAOR, URI LUBLIN und ANTHONY LIGUORI: *kvm: the Linux virtual machine monitor*. In: *Proceedings of the Linux symposium*, Band 1, Seiten 225–230, 2007.

[kvm 13] *KVM/ARM Internetseite*. `https://systems.cs.columbia.edu/projects/kvm-arm/`, 2013. [Online; abgerufen am 25.7.2016].

[Lemb 10] LEMBERGER, GÜNTER: *Empirische Identifikation von Parametern mit Einfluss auf die Effizienz von Virtualisierung am Beispiel von VMwareESXi*, 2010.

[LiBo 08] LIKELY, GRANT und JOSH BOYER: *A symphony of flavours: Using the device tree to describe embedded hardware*. In: *Proceedings of the Linux Symposium*, Band 2, Seiten 27–37, 2008.

[lin 16] *Linux Sunxi U-Boot wiki*. `https://github.com/linux-sunxi/u-boot-sunxi/wiki`, 2016. [Online; abgerufen am 22.7.2016].

[Lind 10] LINDINGER, TOBIAS: *Optimierung des Wirkungsgrades virtueller Infrastrukturen*. Doktorarbeit, LMU, 2010.

[liv 16] *Redhat Developer Blog: Live Migrating QEMU-KVM Virtual Machines*. `http://developers.redhat.com/blog/2015/03/24/live-migrating-qemu-kvm-virtual-machines`, 2016. [Online; abgerufen am 5.8.2016].

[LuLi 07] LUBLIN, URI und ANTHONY LIGUORI: *KVM Live Migration*. In: *KVM Forum*, 2007.

[mbw 15] *mbw - Memory Bandwidth Benchmark*. `https://github.com/raas/mbw`, 2015. [Online; abgerufen am 03.8.2016].

[mem 16] *memtester website*. `http://pyropus.ca/software/memtester/`, 2016. [Online; abgerufen am 28.7.2016].

[OKKA 10] OH, SOO-CHEOL, KANGHO KIM, KWANGWON KOH und CHANG-WON AHN: *ViMo (virtualization for mobile): a virtual machine monitor supporting full virtualization for ARM mobile systems*. Proc. Advanced Cognitive Technologies and Applications, COGNITIVE, 2010.

[okl 16] *OKL 4 Microvisor Internetseite*. `https://gdmissionsystems.com/cyber/products/trusted-computing-cross-domain/microvisor-products/`, 2016. [Online; abgerufen am 25.7.2016].

[OPD+ 12] OU, ZHONGHONG, BO PANG, YANG DENG, JUKKA K NURMINEN, ANTTI YLÄ-JÄÄSKI und PAN HUI: *Energy-and cost-efficiency analysis of arm-based clusters*. In: *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*, Seiten 115–123. IEEE, 2012.

[PdOVN 12] PADOIN, EDSON L, DANIEL AG DE OLIVEIRA, PEDRO VELHO und PHILIPPE OA NAVAUX: *Evaluating performance and energy on ARM-based clusters for high performance computing*. In: *2012 41st International Conference on Parallel Processing Workshops*, Seiten 165–172. IEEE, 2012.

[PoGo 74] POPEK, GERALD J und ROBERT P GOLDBERG: *Formal requirements for virtualizable third generation architectures*. Communications of the ACM, 17(7):412–421, 1974.

[RoGa 05] ROSENBLUM, MENDEL und TAL GARFINKEL: *Virtual machine monitors: Current technology and future trends*. Computer, 38(5):39–47, 2005.

[Roma 10] ROMANYUK: *Empirische Identifikation von Parametern mit Einfluss auf die Effizienz von Virtualisierung am Beispiel von MS Hyper-V*, 2010.

[RRT+ 08] RAGHAVENDRA, RAMYA, PARTHASARATHY RANGANATHAN, VANISH TALWAR, ZHIKUI WANG und XIAOYUN ZHU: *No power struggles: Coordinated multi-level power management for the data center*. In: *ACM SIGARCH Computer Architecture News*, Band 36, Seiten 48–59. ACM, 2008.

[Russ 08] RUSSELL, RUSTY: *virtio: towards a de-facto standard for virtual I/O devices*. ACM SIGOPS Operating Systems Review, 42(5):95–103, 2008.

[Ryzh 06] RYZHYK, LEONID: *The ARM Architecture*. Chicago University, Illinois, EUA, 2006.

[SKM 08] SINGH, AAMEEK, MADHUKAR KORUPOLU und DUSHMANTA MOHAPATRA: *Server-storage virtualization: integration and load balancing in data centers*. In: *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, Seite 53. IEEE Press, 2008.

[Stab 16] STABELLINI, STEFANO ET AL.: *Xen ARM with Virtualization Extensions white paper*. `http://wiki.xen.org/wiki/Xen_ARM_with_Virtualization_Extensions_whitepaper`, 2016. [Online; abgerufen am 14.7.2016].

[StCa 12] STABELLINI, STEFANO und IAN CAMPBELL: *Xen on arm cortex a15*. Xen Summit North America, 2012, 2012.

[SVTR 11] SEMENOV, EVGENY, DANIEL VERSICK, DJAMSHID TAVANGARIAN und FORSCHUNGSGRUPPE RECHNERARCHITEKTUR: *Ansätze zur Reduktion der Live-Migrationsdauer virtueller Maschinen*. 2011.

[SWM 11] SIDDHISENA, BUDDHIKA, LAKMAL WARUSAWITHANA und MITHILA MENDIS: *Next generation multi-tenant virtualization cloud computing platform*. In: *Advanced Communication Technology (ICACT), 2011 13th International Conference on*, Seiten 405–410. IEEE, 2011.

[Tsio 10] TSIOUPROU, EVANGELIA: *Empirische Identifikation von Parametern mit Einfluss auf die Effizienz von Virtualisierung am Beispiel von OpenVZ/Virtuozzo*, 2010.

[UNR+ 05] UHLIG, RICH, GIL NEIGER, DION RODGERS, AMY L SANTONI, FERNANDO CM MARTINS, ANDREW V ANDERSON, STEVEN M BENNETT, ALAIN KAGI, FELIX H LEUNG und LARRY SMITH: *Intel virtualization technology*. Computer, 38(5):48–56, 2005.

[VaHe 11] VARANASI, PRASHANT und GERNOT HEISER: *Hardware-supported virtualization on ARM*. In: *Proceedings of the Second Asia-Pacific Workshop on Systems*, Seite 11. ACM, 2011.

[Weiß 09] WEISS, MICHAEL: *Der Xen-Hypervisor*. 2009.

[WRS 11] WILLEMS, CHRISTIAN, SEBASTIAN ROSCHKE und MAXIM SCHNJAKIN: *Virtualisierung und Cloud Computing: Konzepte, Technologiestudie, Marktübersicht*. Nummer 44. Universitätsverlag Potsdam, 2011.

[xen 13] *Xen 4.3 release notes*. `http://wiki.xenproject.org/wiki/Xen_4.3_Release_Notes`, 2013. [Online; abgerufen am 25.7.2016].

[Xen 16a] *Xen ARM with Virtualization Extensions*. `http://wiki.xen.org/wiki/Xen_ARM_with_Virtualization_Extensions`, 2016. [Online; abgerufen am 14.7.2016].

[xen 16b] *Xen Project Release Features*. `http://wiki.xenproject.org/wiki/Xen_Project_Release_Features`, 2016. [Online; abgerufen am 26.7.2016].

[Zyng 15] ZYNGIER, MARC: *Virtualization on the ARM architecture*. `https://events.linuxfoundation.org/sites/events/files/slides/xds15_0.pdf`, 2015. [Online; abgerufen am 14.7.2016].

Abkürzungsverzeichnis

ABI Application Binary Interface

AMD Advanced Micro Devices

ARM Advanced RISC Machine

ARM VE ARM Virtualization Extensions

CISC Complex Instruction Set Computing

GB Gigabyte

KB Kilobyte

KVM Kernel Virtual Machine

NFS Network File System

NTP Network Time Protocol

RISC Reduced Instruction Set Computing

SoC System-on-Chip

SPL Secondary Program Loader

ssh Secure Shell

TCP Transmission Control Protocol

VLANs Virtual Local Area Networks

VM Virtuelle Maschine