INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Diplomarbeit

Überprüfung der Echzteit-Fähigkeiten von Intel-VT-basierten Virtualisierungslösungen

Oleg Galimov

INSTITUT FÜR INFORMATIK

DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Diplomarbeit

Überprüfung der Echtzeit-Fähigkeiten von Intel-VT-basierten Virtualisierungslösungen

Oleg Galimov

Aufgabensteller:	Prof. Dr. Heinz-Gerd Hegering
Betreuer:	Dipl.-Inform. Ralf König Dr. Vitalian Danciu Dr. Harald Rölle (Siemens)
Abgabetermin:	22. Oktober 2008

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

München, den 22. Oktober 2008

. . . .
(Unterschrift des Kandidaten)

Abstract

In dieser Arbeit wird die Praxistauglichkeit von Virtualisierung von eingebetteten Systemen untersucht. Dabei wird vor allem der Aspekt der Echtzeitfähigkeit überprüft. Um dies zu erreichen werden praktische Versuche auf der Intel-VT Plattform durchgeführt. Die Versuche bilden dabei verschiedene Anwendungsszenarien für Virtualisierung ab. Im Rahmen der Versuche werden mit Hilfe von Benchmarks Latenzen gemessen. Auch die vielen Probleme, die ich bei der Versuchsdurchführung hatte, kommen zur Sprache. Am Ende der Arbeit werden mit Hilfe dieser Messergebnisse dann Schlüsse bezüglich der Eignung von Virtualisierung für Echtzeitanwendungen gezogen.

Inhaltsverzeichnis

1 Problemstellung **1**

1.1 Motivation . . . 1

1.2 Fragestellung . . . 2

1.2.1 Begriffe . . . 2

1.2.2 Schwierigkeiten bei der Virtualisierung von Echtzeitanwendungen . . . 4

1.2.3 Zentrale Fragestellung . . . 7

1.2.4 Übersicht über die Arbeit . . . 7

2 Vorüberlegungen **9**

2.1 Anwendungen für Virtualisierung bei eingebetteten Systemen . . . 9

2.1.1 Schichtenprinzip . . . 9

2.1.2 Anwendungsszenarien . . . 12

2.2 Versuchsszenarien . . . 16

3 State of the Art **19**

3.1 Überblick über Produkte zur Virtualisierung . . . 19

3.1.1 KVM . . . 20

3.1.2 Xen . . . 20

3.1.3 VMWare . . . 20

3.2 Related Work . . . 21

3.2.1 Mikrokernel-Ansätze . . . 21

3.2.2 ARM TrustZone . . . 22

4 Versuchsvorbereitungen **23**

4.1 Auswahl der Hard- und Software . . . 23

4.1.1 Hardware . . . 23

4.1.2 Virtuelle Maschine . . . 23

4.1.3 Host-Betriebssystem . . . 24

4.2 Versuchsdefinition . . . 25

4.2.1 Allgemeines . . . 25

4.2.2 Messungen . . . 25

4.2.3 Versuch 1 - Echtzeitanwendung auf dem Host mit grafischer Anwendung im Gast . . . 28

4.2.4 Versuch 2 - Echtzeitanwendung im Gast . . . 29

4.2.5 Versuch 3 - Zwei virtuelle Maschinen jeweils mit Echtzeitanwendung . 30

4.2.6 Versuch 4 - Zwei virtuelle Maschinen je eine mit Echtzeitanwendung und grafischer Anwendung . . . 30

4.3 Konfiguration . . . 31

4.3.1 Linux-Kernel . . . 31

4.3.2 Konfiguration von KVM . . . 31
4.3.3 Konfiguration von VMWare . . . 33
4.4 Echtzeitbenchmark . . . 33
4.4.1 Auswahl . . . 33
4.4.2 Anpassung . . . 34
4.4.3 Anpassung für Messung innerhalb der virtuellen Maschine . . . 35
4.5 Timehog . . . 37
4.5.1 Überblick . . . 37
4.5.2 Implementierung . . . 38

5 Versuchsdurchführung und Ergebnisse 41
5.1 Überblick über die Versuche . . . 41
5.2 Versuch 1 - Echtzeitanwendung auf dem Host mit grafischer Anwendung im Gast . . . 42
5.2.1 Beschreibung . . . 42
5.2.2 Vorbereitung . . . 42
5.2.3 Erwartungen . . . 43
5.2.4 Subjektiver Eindruck . . . 44
5.2.5 Problem bei der Durchführung . . . 44
5.2.6 Vergleich 1 - Latenzen mit und ohne parallel laufende VM . . . 45
5.2.7 Vergleich 2 - Latenzen mit und ohne RT-Patch . . . 46
5.2.8 Vergleich 3 - Verschiedene Arten von Last im Gast . . . 47
5.2.9 Langzeitmessung . . . 48
5.2.10 Fazit . . . 48
5.3 Versuch 2 - Echtzeitanwendung im Gast . . . 49
5.3.1 Beschreibung . . . 49
5.3.2 Vorbereitungen . . . 49
5.3.3 Erwartungen . . . 50
5.3.4 Probleme bei der Durchführung . . . 50
5.3.5 Vergleich 1 - Latenzen auf dem Host und in der virtuellen Maschine . 51
5.3.6 Vergleich 2 - Verschiedene Arten von Last im Gast . . . 52
5.3.7 Langzeitmessung . . . 52
5.3.8 Fazit . . . 52
5.4 Versuch 3 - Zwei virtuelle Maschinen jeweils mit Echtzeitanwendung . . . 53
5.4.1 Beschreibung . . . 53
5.4.2 Vorbereitung . . . 53
5.4.3 Probleme bei der Durchführung . . . 53
5.4.4 Ergebnisse . . . 55
5.4.5 Fazit . . . 55
5.5 Versuch 4 - Zwei virtuelle Maschinen je eine mit Echtzeitanwendung und grafischer Anwendung . . . 56
5.5.1 Beschreibung . . . 56
5.5.2 Vorbereitung . . . 56
5.5.3 Ergebnisse . . . 56
5.5.4 Fazit . . . 56
5.6 Zusammenfassung . . . 57

6 Demonstrator 59
6.1 Motivation 59
6.2 Konzept 59
6.3 Implementierung 60
6.3.1 Programme 60
6.3.2 Skripte 60
6.4 Beschreibung 61
6.4.1 Szenario 1 61
6.4.2 Szenario 2 62
6.4.3 Szenario 3 63
6.4.4 Szenario 4 63

7 Zusammenfassung und Ausblick 65
7.1 Zusammenfassung 65
7.2 Ausblick 66

A Inhalt der CD 67

Abbildungsverzeichnis 69

Literaturverzeichnis 71

1 Problemstellung

1.1 Motivation

Während Virtualisierung noch vor 10 Jahren ein Nischendasein führte und nur in Randbereichen, wie bei der Entwicklung oder in Mainframes eingesetzt wurde, ist sie heute bei verschiedensten Anwendungen standardmäßig im Einsatz. Vor allem im Serverbereich werden Virtualisierungstechnologien eingesetzt, um Hardware zu konsolidieren, vorhandene Hardware besser auszunutzen und das Management der Systeme zu erleichtern. Da im Bereich der eingebetteten Systeme andere Anforderungen als im Serverbereich gelten, kommen viele Entwicklungen dort später zum Einsatz. Doch heute liegen auch dort Faktoren vor, die den Einsatz von Virtualisierung nahe legen, z.B.:

- Hardware-Virtualisierung
- Im Regelbetrieb brachliegende Leistungs- und Speicherreserven
- Mehrkernprozessoren

Auch gibt es bei eingebetteten Systemen viele ähnliche und neue Anwendungen, die von Virtualisierung auf verschiedene Art und Weise profitieren könnten. Man denke, im Hinblick auf das Konsolidierungsszenario, nur an die fast 100 Prozessoren, die heute in einer Oberklasselimusine zu finden sind. Daher lag es für Siemens Corporate Technology nahe, den Einsatz von Virtualisierung bei eingebetteten Systemen zu überprüfen. Aus dieser Problemstellung heraus entstand diese Diplomarbeit.

1.2 Fragestellung

Dieses Kapitel soll helfen, die Fragestellung der Diplomarbeit besser zu verstehen. Dazu werden zunächst grundlegende Begriffe eingeführt. Danach werden die besonderen Schwierigkeiten der Aufgabe dargestellt. Daraus werden dann konkrete Fragen abgeleitet, die im Laufe der Arbeit beantwortet werden. Am Ende des Kapitels wird dann ein Überblick über die Arbeit gegeben.

1.2.1 Begriffe

Um die Problemstellung und die damit verbundenen Schwierigkeiten besser verstehen zu können sollen im Vorfeld wichtige Begriffe erläutert werden, die den Kern der Aufgabenstellung ausmachen.

Virtualisierung

Für den Begriff Virtualisierung gibt es in der Fachwelt keine allgemein anerkannte Definition. Für unsere Zwecke soll der Begriff eine Hardwareabstraktionsschicht bezeichnen. Bei der Systemvirtualisierung werden dann Teile eines Computers emuliert, um weitere Betriebssysteme ablaufen lassen zu können. Besser kann der Begriff eingegrenzt werden, wenn man einige verbreitete Arten der Systemvirtualisierung betrachtet:

- Bei der **Emulation** wird die Hardware in Software nachgestellt.
- Bei der **Paravirtualisierung** wird die Hardware nicht virtualisiert. Stattdessen nutzen speziell modifizierte Betriebssystemversionen eine Abstraktionsschicht in Software, um auf Ressourcen zuzugreifen.
- Bei der **Vollvirtualisierung** wird dem Gastsystem ein Teil der Hardware zur Verfügung gestellt. Um dies problemlos zu ermöglichen, muss die Hardware dies unterstützen.

Bei der Systemvirtualisierung stellt vor allem die Virtualisierung des Prozessors ein Problem dar. Dadurch, dass andere Systemkomponenten bereits durch Treiber abstrahiert werden, ist es vergleichsweise einfach, diese zu emulieren. Außerdem sind dabei die Anforderungen an die Performance nicht so hoch. Die Virtualisierung des Prozessors hat sich in der Vergangenheit als schwierig erwiesen, da der Prozessor dem Betriebssystem Funktionalitäten (z.B. Prozesswechsel oder Speicherschutz) zur Verfügung stellt, die sich nicht ohne weiteres mit Gastsystemen teilen lassen. Daher mussten diese umständlich emuliert werden.

Da die großen Prozessorhersteller das Potential von Systemvirtualisierung erkannt haben, haben sie daher spezielle Unterstützung in die Prozessoren eingebaut (Intel-VT, AMD Pacifica, ARM TrustZone) um die Lösung dieses Problems zu vereinfachen. Die Einführung dieser Technologien ermöglichte volle Virtualisierung bei, im Vergleich zur Emulation, deutlich geringeren Performanceverlusten.

Erst die Einführung dieser Virtualisierungstechnologien macht diese Diplomarbeit überhaupt möglich, denn nur mit Hilfe von Vollvirtualisierung ergibt sich die notwendige Performance,

während zugleich andere wünschenswerte Eigenschaften (z.B. Sicherheit, Unterstützung für nicht modifizierte Gastbetriebssysteme) in einem gewissen Umfang erfüllt werden.

Eingebettetes System

Ein eingebettetes System ist eine Kombination von Hard- und Software, die dazu bestimmt ist eine oder mehrere Spezielle Funktionen zu erfüllen [Net]. Nach dieser Definition ist es das Gegenteil von einem herkömmlichen Desktop oder Serversystem, denn diese sind dazu geschaffen, möglichst viele verschiedene Funktionen zu erfüllen.

Durch die Tatsache, dass die Leistungsfähigkeit der Prozessoren immer weiter zunimmt verschwimmen allerdings die Grenzen. So können heute Handys, die eigentlich ein Beispiel für ein eingebettetes System sind, eine Vielzahl von verschiedenen Programmen (Organizer, Spiel, Internetbrowser, Virus) ausführen.

Eingebettete Systeme zeichnen sich durch spezielle Anforderungen aus, die jedoch stark vom konkreten Einsatzzweck abhängen. Eine der häufigsten Anforderungen ist die Echtzeitfähigkeit, die einen zentralen Punkt bei der Frage darstellt, ob sich eine Lösung für den Einsatz in einem eingebetteten System eignet. Aus diesem Grund stellt die Frage nach der Echtzeitfähigkeit den zentralen Untersuchungspunkt dieser Arbeit da.

Echtzeit

Eine Echtzeitanforderung bedeutet, dass eine Berechnung oder eine andere Aktion in einer vorher fest definierten Zeitspanne abgeschlossen sein muss. Zu einem späteren Zeitpunkt hat das Ergebnis keinen Wert. Da reale Computer über Eingabe und Ausgabeschnittstellen gesteuert werden und der Zugriff darauf mit zusätzlichen Verzögerungen verbunden ist, zählt in der Realität der Zeitabschnitt zwischen Eingabe und der gewünschten Ausgabe darauf. Dieser wird als Latenz bezeichnet.

In Abbildung 1.1 ist ein Beispiel für eine solche Anwendung dargestellt. Der schematisch dargestellte Controller verbindet den Scanner mit einem Greifarm der die ankommenden Waren auf Ausgabebänder legt. Offensichtlich muss, damit das System funktioniert, der Greifarm einen Steuerbefehl bekommen, wenn sich das Werkstück unter ihm befindet. Die maximal zulässige Latenz wird hier also durch die Zeit bestimmt, die das Werkstück auf dem Laufband vom Scanner zum Greifarm benötigt. Das obere Beispiel stellt eine sogenannte harte Echtzeitanforderung da. Bei einer solchen ist eine Überschreitung der maximal zulässigen Latenz im Normalbetrieb nicht zulässig. Um dies zu garantieren wird oft die maximale Latenz für jede Komponente auf Hardwareebene ermittelt. Um dieses Vorgehen zu wählen muss außerdem das Betriebssystem entsprechende Garantien abgeben (in der Regel eine garantierte Reaktionszeit auf ein Interrupt).

Weitere Beispiele für harte Echtzeitanforderungen sind Airbags oder medizinische Geräte, bei denen sogar das Leben von Menschen von der Einhaltung der Latenzen abhängen könnte.

Moderne Desktopprozessoren sind für harte Echtzeitanforderungen mit kleinen maximal zulässigen Latenzen unter Umständen nicht geeignet, da der Prozessor mit Hilfe von Sy-

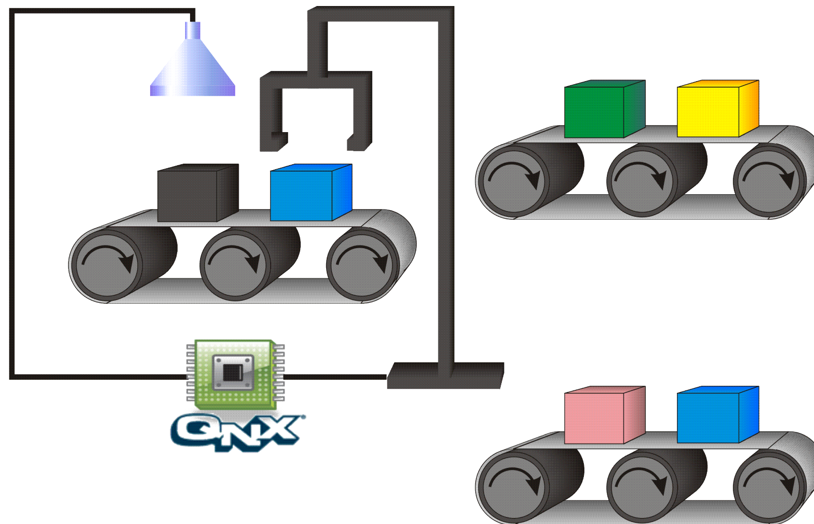


Abbildung 1.1: Beispiel für eine Anwendung mit Echtzeitanforderung

stem Management Interrupts dem Betriebssystem Rechenzeit entziehen kann. Deren Dauer ist zwar begrenzt, aber das Betriebssystem kann den Zeitpunkt nicht beeinflussen [Int08].

Auch die verbreiteten Betriebssysteme wie Windows (ohne CE), Linux oder MacOS, sind zu mindestens ohne Modifikationen nicht für die Erfüllung von harten Echtzeitanforderungen geeignet, da diese für andere Anforderungen entwickelt wurden. Vereinfacht betrachtet versucht man, bei der Entwicklung eines Echtzeitbetriebssystems die maximale Verzögerung im schlechtesten Fall zu minimieren, wohingegen bei Desktopbetriebssystemen die durchschnittliche Verzögerung möglichst klein sein soll.

Neben den harten Echtzeitanforderungen gibt es auch weiche Echtzeitanforderungen. Hier stellen auch größere Überschreitungen der maximal zulässigen Latenz kein Problem da, solange sie nur selten erfolgen. Ein Beispiel für eine solche ist z.B. Telefonie. Hier wird der Nutzer Verzögerungen sicher tolerieren können, solange diese nicht die Regel darstellen. In der Praxis ist es hier ausreichend, dass das Betriebssystem dem Programm eine hohe Priorität zuweist und hohe Verzögerungen durch das Betriebssystem selbst entsprechend vermindert werden.

Da das Garantieren von harten Echtzeitanforderungen oft mit einem hohen Aufwand verbunden ist, werden die Latenzen im praktischen Teil dieser Arbeit nur im Hinblick auf weiche Echtzeitanforderungen betrachtet.

1.2.2 Schwierigkeiten bei der Virtualisierung von Echtzeitanwendungen

Das Thema dieser Arbeit betrifft die Echtzeitfähigkeit von Virtualisierungslösungen. Da stellt sich natürlich die Frage, wieso die Echtzeitfähigkeit überhaupt ein Problem darstellen soll.

Virtualisierung kommt immer dann zum Einsatz, wenn mehrere miteinander nicht interagierende Anwendungen auf einem System ausgeführt werden sollen. Dies ist natürlich auch

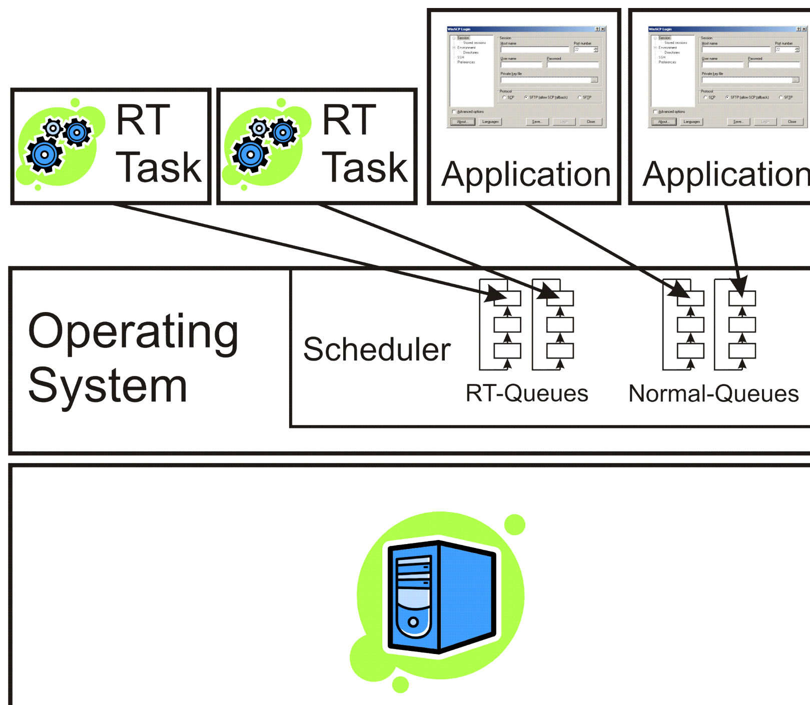


Abbildung 1.2: Illustration eines Prozess-Schedulers mit FIFO-Warteschlangen

ohne Virtualisierung möglich. Immer dann, wenn Echtzeitfähigkeiten von einigen der Anwendungen vorausgesetzt werden, erhalten die entsprechenden Prozesse eine höhere Priorität. So wird sicher gestellt, dass diese die Rechenzeit bekommen, die sie brauchen, und somit die Latenzen möglichst klein bleiben.

Bei Desktopbetriebssystemen ist es oft unerwünscht, dass Prozesse mit hoher Priorität und hohem Rechenzeitverbrauch die anderen Anwendungen und somit auch die grafische Benutzeroberfläche ausbremsen, da dies zu einem negativen Benutzererlebnis führt. Dieses Phänomen wird als „aushungern“ der entsprechenden Prozesse bezeichnet und wird dadurch verhindert, dass das Betriebssystem die Prioritäten der Prozesse zeitweilig anpassen kann, so dass jeder nach einer gewissen Zeitspanne Prozessorzeit bekommt.

Dieses Verhalten ist bei Vorhandensein von Echtzeitprozessen unerwünscht, da diese, wenn benötigt, soviel Rechenzeit wie möglich bekommen sollen um die maximale Latenz gering zu halten. Daher haben Betriebssysteme die sich für den Echtzeiteinsatz eignen spezielle Echtzeitprioritäten, die strikt befolgt werden. Ein Teil der Echtzeitprioritäten hat sogar höhere Priorität als Betriebssystemprozesse.

Die Zuteilung der Rechenzeit übernimmt in modernen Betriebssystemen ein Prozess-Scheduler. Dabei wird heute meist Präemptives Multitasking verwendet. Hier wird die Ausführung des aktuell laufenden Prozesses nach einer vorher festgelegten Zeitspanne vom Prozessor unterbrochen und der Scheduler darf entscheiden welcher Prozess als nächster Prozessorzeit zugeteilt bekommt (es kann auch der gerade unterbrochene sein). Genaueres kann man im Buch [Tan08] nachlesen.

Es gibt verschiedene Algorithmen für die Implementierung von Prozess-Schedulern, aber die

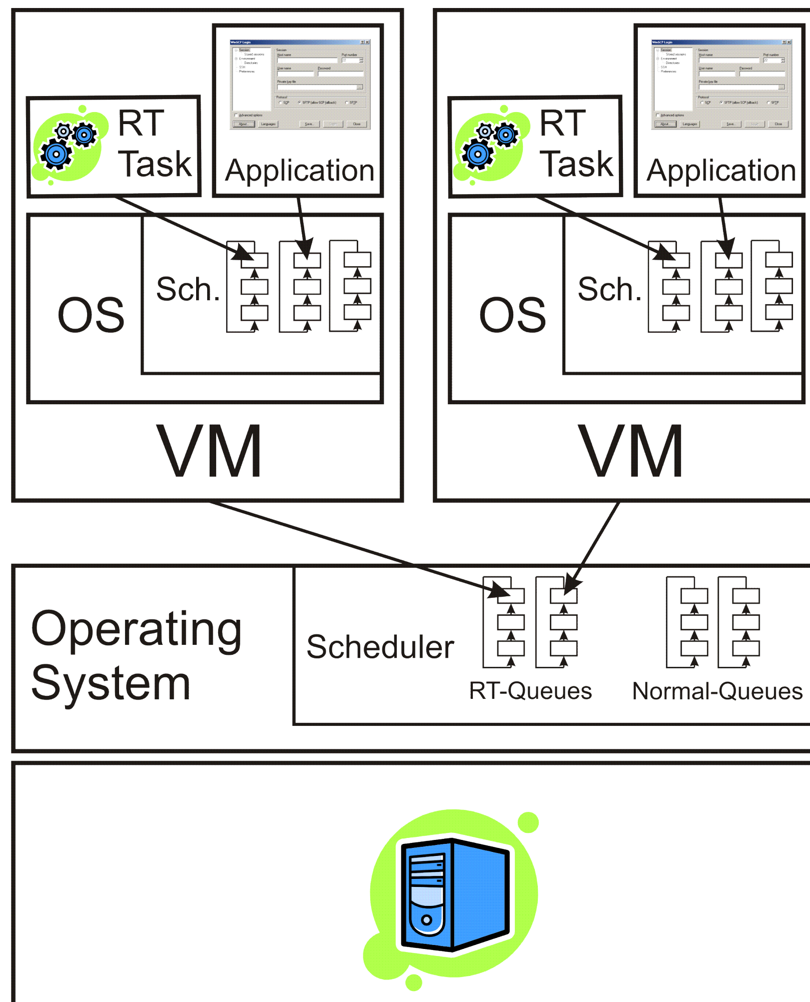


Abbildung 1.3: Beziehungen zwischen Warteschlangen bei der Verwendung von mehreren virtuellen Maschinen

meisten kann man sich logisch als eine Ansammlung von FIFO-Warteschlangen vorstellen. Für jede Priorität gibt es dabei eine eigene Warteschlange. So gibt der Scheduler demjenigen Prozess Rechenzeit, der als erstes in der Warteschlange mit der höchsten Priorität eingeordnet ist und auf Rechenzeit wartet. Bei Echtzeitprioritäten geht der Scheduler dabei strikt vor, so dass passieren kann, dass Prozesse mit niedriger Priorität nie Rechenzeit zugeteilt bekommen. Bei den anderen Prozessen merkt sich der Scheduler hingegen, wer schon oft dran war, so dass jeder Prozess mal CPU-Zeit zugeteilt bekommt. Ein Beispiel mit insgesamt vier Anwendungen, davon je zwei mit Echtzeitpriorität und mit normaler Priorität, sieht man in Abbildung 1.2.

Abbildung 1.3 illustriert nun das Grundproblem, dass entsteht, wenn man Virtualisierung im Zusammenhang mit festen Prioritäten verwenden will. Während es ohne virtuelle Maschinen einen Scheduler gibt, der die Prioritäten aller Prozesse kennt, sieht man in Abbildung 1.3 drei Scheduler die nichts von der gegenseitigen Existenz wissen. Aus diesem Grund muss man den einzelnen virtuellen Maschinen manuell Prioritäten zuweisen, was nicht zwingend

funktionieren muss, da es sich bei den Virtualisierungslösungen um Kernelerweiterungen handelt, die nicht zwingend mit dem Scheduler kooperieren müssen.

Das in Abbildung 1.3 dargestellte Beispiel zeigt aber noch ein weiteres Problem auf. Angenommen, der RT-Task in der linken virtuellen Maschine habe eine höhere Priorität als der andere. Um sicher zu stellen, dass dieser die nötige Rechenzeit auch zugeteilt bekommt, muss man auf dem Hostsystem der gesamten linken virtuellen Maschine höhere Priorität zuweisen als der rechten. Das hätte jedoch zur Folge, dass auch die Benutzeranwendung in der linken virtuellen Maschine höhere Priorität hätte, als der RT-Task in der rechten VM! So gesehen kann das Beispiel so gar nicht sinnvoll umgesetzt werden. Dieses Problem muss beim Systemdesign unbedingt beachtet werden.

1.2.3 Zentrale Fragestellung

Im Lichte der eben dargelegten Entwicklungen und Schwierigkeiten kommt man zwangsläufig zu der Frage, ob man die Vorteile der Virtualisierung schon heute in eingebetteten Systemen nutzen kann. Da der Leser dieser Arbeit vermutlich ein Systementwickler sein wird, der sich mit der Frage beschäftigt, ob er Virtualisierung benutzen kann, um ein konkretes Problem in einem Projekt zu lösen, soll deshalb von der folgenden zentralen Fragestellung ausgegangen werden:

Ist Virtualisierung in eingebetteten Systemen praxistauglich?

Dabei soll im Laufe der Diplomarbeit mit Hilfe von praktischen Versuchen dieser Frage nachgegangen werden. Angesichts des breiten Spektrums von Plattformen und Anwendungsmöglichkeiten kann diese Frage im Rahmen einer Diplomarbeit nicht einmal ansatzweise erschöpfend geklärt werden. Vielmehr sollen einige Versuche überlegt und durchgeführt werden, die den wichtigen Aspekt der Echtzeitfähigkeit auf einer Beispielplattform untersuchen. Andere Aspekte sollen wenn möglich trotzdem angesprochen werden. Um einen Überblick über andere Plattformen zu geben, werden im Kapitel "Related Work" auch Ansätze präsentiert, die dann bei den Versuchen nicht weiter verfolgt werden.

Da die zentrale Frage im Rahmen dieser Arbeit sicher nicht beantwortet werden kann, besteht das Ziel der Arbeit vielmehr darin, einen ersten Eindruck von den Möglichkeit der Virtualisierung von Anwendungen mit Echtzeitanforderungen zu geben, diesbezüglich erste Messungen zu machen, sowie anfallende Probleme zu identifizieren.

1.2.4 Übersicht über die Arbeit

Hier soll ein Überblick über die weiteren Kapitel der Arbeit gegeben werden um den verfolgten Gedankengang verständlich zu machen.

Theoretischer Teil

Der theoretische Teil der Arbeit dient dazu den praktischen Teil vorzubereiten. Zum einen wird hier eine Auswahl an Experimenten, die im praktischen Teil verwirklicht werden, getroffen und begründet, zum anderen wird der State of the Art dargestellt.

Vorüberlegungen Um überhaupt Versuche durchführen zu können, muss man sich mögliche Anwendungsmöglichkeiten analysieren. Dies wird in diesem Kapitel gemacht. Im Anschluss daran, werden daraus Versuchsszenarien abgeleitet, die die Basis für den praktischen Teil bilden.

State of The Art In diesem Kapitel werden verschiedene Virtualisierungslösungen dargestellt. Zusätzlich werden im Abschnitt Related Work Arbeiten präsentiert, die andere Ansätze verfolgen.

Praktischer Teil

Im praktischen Teil werden die Versuche nun durchgeführt.

Versuchsvorbereitung Bevor die Versuche beginnen, soll in der Versuchsvorbereitung eine sinnvolle Versuchskonfiguration gefunden werden. Diese beinhaltet eine Auswahl von Hard- und Software für die Versuche. Unter Software fallen hier auf der einen Seite die Betriebssysteme und auf der anderen Seite der verwendete Benchmark. Zudem wird die von mir verwendete Softwarekonfiguration dargestellt und begründet.

Versuche Einen großen Teil der Arbeit werden Versuche ergeben, deren Ergebnisse im Lichte der Erwartungen zu analysieren sind. Dabei entstehen qualitative Ergebnisse (was funktioniert) und quantitative Ergebnisse (wie sind die Latenzen).

Demonstrator Zusätzlich zu den Ergebnissen soll ein Demonstrator (ein PC der zu Demonstrationszwecken eingerichtet ist) entstehen, der die Echtzeitfähigkeit des Systems demonstriert, indem er die Szenarien aus dem theoretischen Teil darstellt und dabei die Latenz live anzeigt. Der Demonstrator soll einem versierten Nutzer einen schnellen Überblick über die Eignung verschiedener Szenarien für seine Anwendungen geben.

2 Vorüberlegungen

Im praktischen Teil der Arbeit soll die Frage nach der Echtzeitfähigkeit von Virtualisierungslösungen geklärt werden. Um dies zu erreichen müssen sinnvolle Versuche gewählt werden. Die Grundlage dafür sind wiederum die Anwendungsszenarien. Diese sollen in diesem Kapitel überlegt werden. Dazu werden zunächst einige Anwendungen für die Nutzung von Virtualisierung aufgezählt. Im zweiten Teil wird dann überlegt, welche Versuche benötigt werden, um eine sinnvolle Teilmenge der möglichen Anwendungsszenarien abzudecken.

2.1 Anwendungen für Virtualisierung bei eingebetteten Systemen

2.1.1 Schichtenprinzip

Die Beziehungen zwischen Computerkomponenten, die aufeinander aufbauen, können oft mit Hilfe von einem Schichtenmodell visualisiert werden. Das Grundprinzip eines Schichtenmodells besteht darin, dass höher liegenden Schichten auf die genau darunter liegende Schicht zugreifen, aber kein Wissen von den weiter unten liegenden Schichten haben. Dies wird dadurch erreicht, dass jeweils höher liegende Schichten die darunter liegende Schicht abstrahieren und ein standardisiertes Interface bieten. In Abbildung 2.1 wird das Schichtenmodell auf ein normales Computersystem angewendet. Die Abbildung stellt dabei eine starke Vereinfachung da. So bestehen Anwendungen und Betriebssystem oft selber aus einer Vielzahl von Schichten.

Im Idealfall kann der Systementwickler die einzelnen Schichten austauschen, ohne dass es Auswirkungen auf andere Schichten hat. In der realen Welt ist dies jedoch nur möglich, wenn die einzelnen Schichten stark standardisiert sind. Dies ist im Allgemeinen jedoch nicht der Fall. So kann man z.B. nur Hardware verwenden, für die es auch einen passenden Treiber

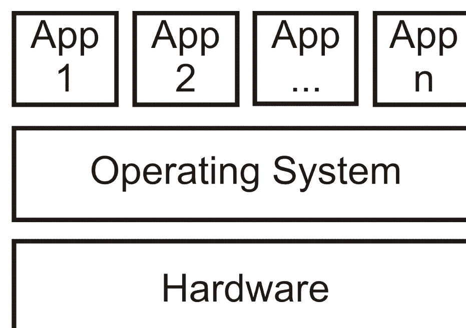


Abbildung 2.1: Schichtenprinzip

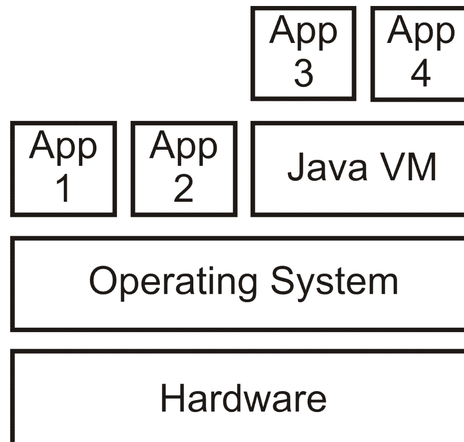


Abbildung 2.2: Schichtenprinzip mit Java

für das verwendete Betriebssystem gibt. Auch kann man nur Anwendungen verwenden, die auf dem speziellen Betriebssystem laufen. Dies erfordert selbst bei der Verwendung von betriebssystemunabhängigen Programmierschnittstellen oft einigen Aufwand.

Einer der Vorteile von virtuellen Maschinen im Allgemeinen besteht darin, dass sie dabei helfen können, dieses Problem zu umgehen. So wird bei Mobiltelefonen heute oft eine Java-Laufzeitumgebung eingesetzt, um Anwendungen auf einer Vielzahl von verschiedenen Geräten ablaufen lassen zu können. In Abbildung 2.2 wird dies illustriert. Durch die Java-Laufzeitumgebung wird eine zusätzliche Schicht eingefügt, die die Flexibilität erhöhen kann. So kann das Betriebssystem frei unter den von der virtuellen Maschine unterstützten gewählt werden und alle Anwendungen sollten in der Regel ohne Änderungen darauf laufen. Diese und andere Vorteile der virtuellen Maschine erkauft man sich mit verschiedenen Nachteilen. Diese Nachteile entstehen praktisch immer, wenn eine zusätzliche Schicht hinzugefügt wird:

- Performanceverluste
- Möglichkeit von Fehlern und Sicherheitslücken in der virtuellen Maschine
- Benötigte Funktionalität wird von der darunterliegenden Schicht zur Verfügung gestellt, aber nicht von der virtuellen Maschine abstrahiert

Ein weiterer Vorteil, der durch die Verwendung von Java entsteht und von dessen Entwickler Sun beworben wird, ist Sicherheit. Mit Sicherheit sei hier der Schutz des Gesamtsystems und sensibler Daten vor böswilligen Anwendungen gemeint. Am Schichtenmodell lässt sich der Sicherheitsgewinn dadurch erklären, dass Anwendungen nur Funktionen benutzen können, die von der darunterliegenden Schicht zur Verfügung gestellt werden. Durch das hinzufügen weiterer Schichten kann nun auf zweierlei Art ein Sicherheitsgewinn entstehen:

- Die virtuelle Maschine kann den Zugriff auf sicherheitsrelevante Funktionalitäten verhindern, einschränken oder vom Benutzer (etwa in Form von Sicherheitsabfragen) überwachen lassen.
- Die Chance auf Sicherheitslücken sinkt, da sich diese in allen relevanten Schichten befinden müssen und hier eine Schicht hinzugefügt wird.

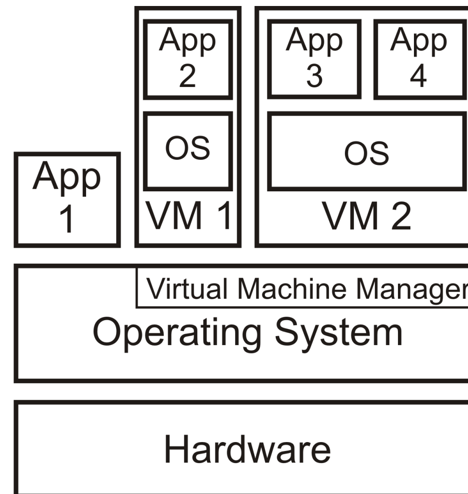


Abbildung 2.3: Schichtenprinzip mit virtuellen Maschinen

Die Flexibilität, die durch die Verwendung der Java-Laufzeitumgebung entsteht, erstreckt sich allerdings nur auf Anwendungen, die speziell dafür entwickelt wurden. Mit Hilfe der Systemvirtualisierung, die in Abbildung 2.3 dargestellt ist, lassen sich die Vorteile auch auf native Programme anwenden. Die virtuelle Maschine hat im Falle von Vollvirtualisierung auch Teile, die auf der Betriebssystemeben agieren (Kernelmodule bzw. Treiber). Daher muss der Virtual Machine Monitor (VMM), die zentrale Komponente der virtuellen Maschine, zur Betriebssystemschicht gezählt werden, auch wenn der Prozess der virtuellen Maschine selber ein normaler Benutzerprozess ist. Es gibt auch VMMs, die nicht auf einem Host-Betriebssystem laufen, sondern direkt auf der Hardware. Logisch betrachtet, sind sie jedoch zur Abbildung 2.3 äquivalent, da das Betriebssystem einfach Teil des VMMs ist.

Die Vorteile entstehen bei diesem Ansatz dadurch, dass sich zwischen Anwendung und Hardware nun zwei Betriebssysteme befinden - das Gastbetriebssystem und das Hostbetriebssystem. So kann man nun tatsächlich Schichten ohne größeren Aufwand austauschen:

- Das Host-Betriebssystem kann frei gewählt werden, solange die virtuelle Maschine darauf läuft. Dies ist interessant, wenn die benötigte Hardware nicht von allen Betriebssystemen unterstützt wird, oder wenn man spezielle Eigenschaften, wie Echtzeitfähigkeit, benötigt.
- Der VMM kann ebenfalls ausgetauscht werden, wenn man spezielle Anforderungen daran hat. Dabei muss zwar die virtuelle Maschine unter Umständen neu eingerichtet werden, Änderungen an den darin laufenden Anwendungen sind jedoch nicht notwendig.
- Die VMs unterstützen oft eine breite Auswahl an Gast-Betriebssystemen. So kann man auf einem Prozessor verschiedene Anwendungen laufen lassen, die verschiedene Betriebssysteme benötigen.

Auch die Sicherheitsvorteile lassen sich auf die Systemvirtualisierung übertragen. Anwendungen in virtuellen Maschinen haben nur Zugriff auf ihnen explizit zugewiesene Ressourcen. Dadurch sinkt die Gefahr, die von schadhaften Programmen ausgeht, da diese selbst im

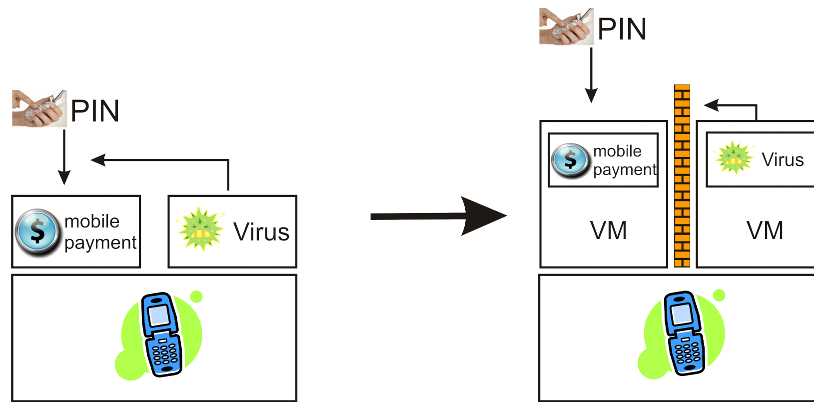


Abbildung 2.4: Mit Hilfe von einer virtuellen Maschine können Trojaner und Viren daran gehindert werden sensible Daten auszuspionieren

Falle von Sicherheitslücken im Betriebssystem keinen Zugriff auf Anwendungen und Daten außerhalb der VM haben.

2.1.2 Anwendungsszenarien

Nun sollen einige mögliche Szenarien, in denen der Einsatz von Virtualisierungslösungen angebracht ist, schemenhaft vorgstellt werden. Hierbei sollen Vorteile von Virtualisierung, die zum Teil im letzten Kapitel vorgestellt wurden, auf mögliche realistische Probleme angewendet werden.

Ein wichtiger Hinweis: Ich gehe bei der theoretischen Betrachtung der Merkmale von virtuellen Maschinen davon aus, dass diese, für die spezielle Anwendung betrachtet, ausreichend fehlerfrei programmiert sind. Das ist heute wahrscheinlich bei keinem Produkt der Fall.

Sicherhet

Bei eingebetteten Systemen im Consumer-Bereich gibt es ein neues Problem, seit diese Geräte immer öfter über einen Internetzugang verfügen: Viren und Trojaner können über das Internet auf die Geräte gelangen und so Benutzerdaten ausspionieren. Auf der anderen Seite werden laufend neue sicherheitsrelevante Technologien wie z.B. mobiles Bezahlen eingeführt. Da die Trojaner meist auf der Betriebssystemebene laufen können diese ohne Probleme etwa die Eingabe einer PIN abhören.

Wie in Abbildung 2.4 dargestellt, kann das Problem mit Hilfe von Virtualisierung gelöst werden. Da einzelne Gastsysteme nicht ohne weiteres miteinander kommunizieren können, muss zur Lösung erreicht werden, dass sicherheitsrelevante Anwendungen und Benutzeranwendungen (wozu dann unbewusst auch Viren und Trojaner gehören) jeweils in verschiedenen virtuellen Maschinen laufen.

Kompatibilität

Da die Lebenserwartung von eingebetteter Software bei manchen Anwendungen sehr hoch ist, begegnet man dem Problem, dass Hardware, die zur Ausführung der Software oder des von der Software vorausgesetzten Betriebssystems notwendig ist, nicht mehr verfügbar ist.

Zum Beispiel waren früher NE 2000 kompatible ISA-NICs verbreitet. Will man heute eine Legacy-Anwendung ausführen, die eine solche voraussetzt, hat man ein Problem, denn diese NICs sind zusammen mit dem ISA-Bus Ende der 90er ausgestorben und heutige NE 2000 kompatible Karten für den PCI-Bus sind nicht mehr mit den alten Treibern kompatibel.

Dieses Problem kann mit Hilfe einer virtuellen Maschine gelöst werden, da diese verschiedene Hardware emulieren kann. Diese Möglichkeit lässt sich am Schichtenmodell wie folgt erklären: Ohne den Einsatz von Virtualisierung lässt sich die Hardwareschicht nicht beliebig austauschen, da das (Legacy-)Betriebssystem bestimmte Hardware voraussetzt. Dank dem Einsatz einer virtuellen Maschine ist es nun möglich, die Hardware zusammen mit dem Host-Betriebssystem beliebig zu wählen, solange die virtuelle Maschine darauf läuft und die benötigte Hardware emulieren kann.

Stabilität und Wartbarkeit

Im Bereich der eingebetteten Systeme ist oftmals eine hohe Verfügbarkeit nötig. Dies ist insbesondere bei Systemen mit Echtzeitanforderungen der Fall, da die Anforderungen offensichtlich nicht eingehalten werden, wenn das gesamte System ausfällt. Lässt man mehrere Anwendungen auf einem Prozessor ablaufen, so steigert sich die Gefahr noch weiter, denn jede Anwendung kann durch einen Programmfehler die Systemstabilität gefährden.

In diesem Fall können die Gefahren durch den Einsatz von Systemvirtualisierung verringert werden. Lässt man einige Anwendungen in einer virtuellen Maschine laufen, so wirkt sich ein Absturz dieser auch nur innerhalb der virtuellen Maschine aus. Die anderen Komponenten sind davon nicht betroffen. Als positiver Nebeneffekt wird dabei die Wartung zur Laufzeit vereinfacht. So kann man die virtuelle Maschine im Falle eines Absturzes beispielsweise neu starten.

Konsolidierung

Aus dem Serverbereich bekannt ist das Konsolidierungsszenario. Dabei werden mehrere Dienste, die traditionell auf verschiedenen Computern laufen mit Hilfe von virtuellen Maschinen auf einem (meist leistungsfähigerem) Computer zusammengefasst. Das Konsolidierungsszenario ist meiner Meinung nach der wichtigste Grund um Systemvirtualisierung einzusetzen. Dennoch wird es hier als letztes beschrieben, da die anderen vorher dargestellten Vorteile, stark dazu beitragen, im Falle einer Konsolidierung virtuelle Maschinen einzusetzen.

In Abbildung 2.5 ist eine mögliche Anwendung in einem eingebetteten System dargestellt. In der Abbildung sieht man einen PC, auf dem eine grafische Oberfläche läuft, sowie eine Maschine mit einem Controller auf dem eine Steuerungssoftware läuft. Diese führt dann

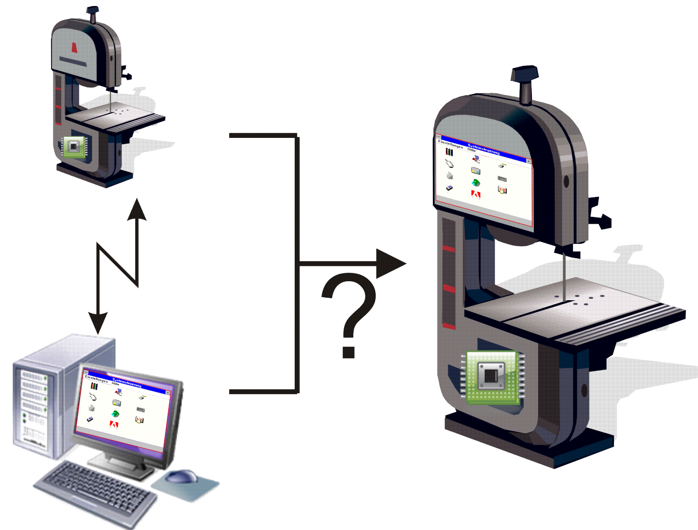


Abbildung 2.5: Mögliche Anwendung des Konsolidierungsszenarios

die Befehle die von dem PC kommen aus. Angesichts der Leistungsfähigkeit heutiger Prozessoren, stellt sich schnell die Frage, ob man nicht beide Anwendungen nicht auf einem Computer ausführen könnte. Dies ist im industriellen Bereich besonders interessant, da dort die Laufzeiten für Maschinen lang sind, und die Komponenten über Jahrzehnte gewartet werden. Eine Reduktion der Anzahl der elektronischen Teile kann so zu erheblichen finanziellen Einsparungen führen. Ohne den Einsatz von Virtualisierung könnte man die grafische Oberfläche und die Steuerungssoftware einfach auf einem PC laufen lassen. Dabei könnten folgende Probleme entstehen:

- Die Anwendungen erfordern verschiedene Betriebssysteme, so dass Anpassungen notwendig sind

- Fehler in der grafischen Oberfläche oder in sonst nicht benötigten Betriebssystemteilen könnten das System samt Steuerungssoftware zum Absturz bringen

- Fehler im Berechtigungsmanagement könnten einen unberechtigten Zugriff auf die Steuerungssoftware erlauben

Alle drei Probleme können, wie bereits beschrieben, durch den Einsatz von Systemvirtualisierung gelöst werden. Eine mögliche Implementierung wird in Abbildung 2.6 dargestellt.

In einem etwas allgemeineren Fall will der Systementwickler eventuell mehrere Anwendungen jeweils in virtuellen Maschinen ablaufen lassen. Dieses Beispiel wird in Abbildung 2.7 dargestellt. Diese Lösung ist nicht immer möglich, da die Anwendungen in virtuellen Maschinen spezielle Anforderungen haben könnten, die sich innerhalb dieser nicht realisieren lassen. Für diese Arbeit zentral sind dabei Echtzeitanforderungen und ihre Erfüllbarkeit innerhalb der VMs.

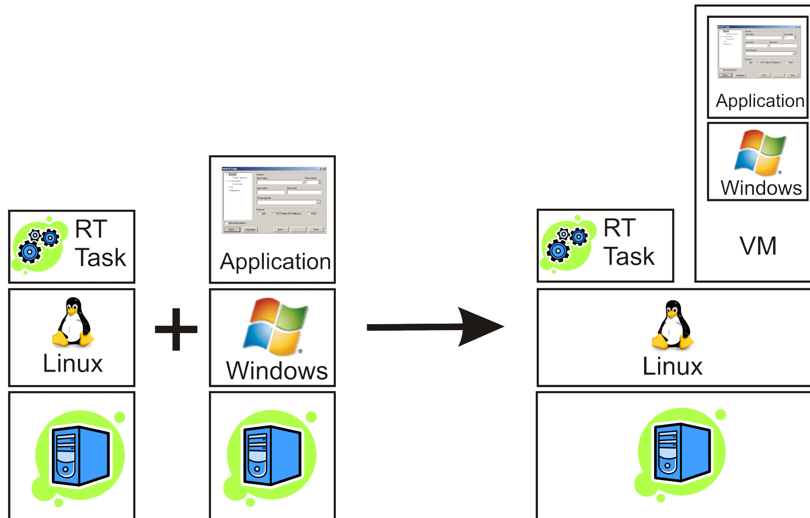


Abbildung 2.6: Konsolidieren einer GUI und einer Echtzeitanwendung

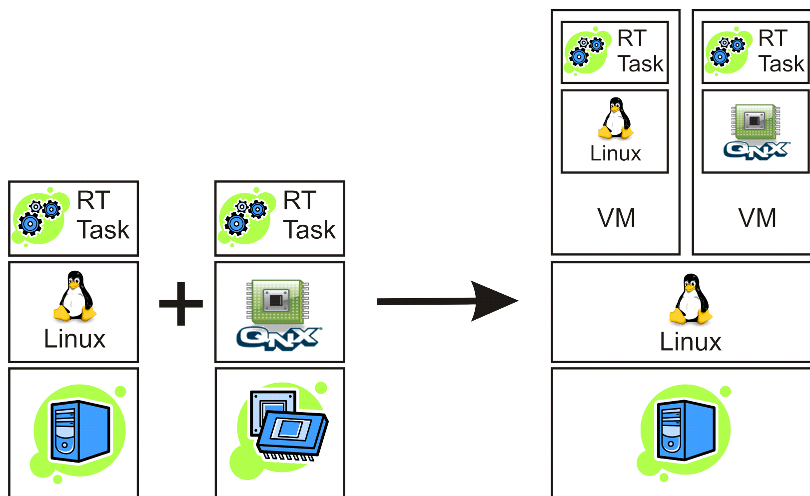


Abbildung 2.7: Konsolidieren von mehreren Echtzeitanwendungen

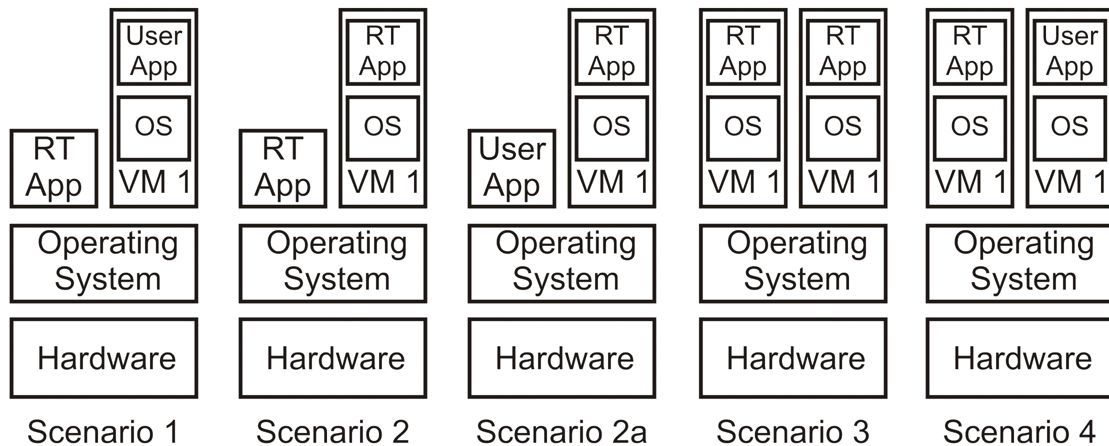


Abbildung 2.8: Konsolidieren von mehreren Echtzeitanwendungen

2.2 Versuchsszenarien

Mit dem etwas besseren Bild von den Vorteilen und Anwendungsmöglichkeiten der Systemvirtualisierung werden nun Versuchsszenarien definiert. Da dabei der Aspekt der Echtzeitfähigkeit im Vordergrund steht kommen nur Versuchsaufbauten in Frage, bei denen mindestens ein Programm Echtzeitanforderungen stellt.

Bei den bischer beschriebenen Anwendungen wurden meist mehrere Programme auf einem System ausgeführt. So werden wir uns auf Versuche beschränken, bei denen mehrere Programme simuliert werden. Anwendungen mit nur einem Prozess, der in einer virtuellen Maschine läuft, kann man dabei ohne weiteres als Spezialfall eines komplexeren Versuchs betrachten. Auf der anderen Seite dürfte der Erkenntnissgewinn durch Versuche, die sich nur durch die Anzahl der Programme voneinander unterscheiden, eher gering ausfallen. So kann man davon ausgehen, dass man durch Versuche mit genau zwei zu konsolidierenden Programmen bereits ein weites Feld abdeckt. Da mindestens ein Programm mit Echzeitanforderungen dabei sein muss ergeben sich so zwei möglich Kombinationen: Echtzeitanwendung mit Benutzeranwendung oder zwei Echtzeitanwendungen.

Die heute verbreiteten Desktop-Virtualisierungslösungen werden alle auf einem Host-Betriebssystem eingerichtet(während manche Server-Virtualisierungslösungen direkt auf der Hardware laufen). Dies erlaubt zusätzliche Möglichkeit für das Systemdesign. So kann ein Teil der Anwendungen direkt auf dem Hostsystem laufen, während der andere Teil in virtuellen Maschinen läuft. Das bedeutet für die Versuche konkret, dass jede Anwendung entweder direkt auf dem Host-Betriebssystem oder aber in einer virtuellen laufen könnte. Da nur Szenarien mit genau zwei Anwendungen untersucht werden, ergeben sich so drei Verschiedene Kombinationslösungen: beide Anwendungen laufen auf dem Hostsystem; beide Anwendungen laufen in virtuellen Maschinen; je eine Anwendung läuft in einer VM und auf dem Host-Betriebssystem. Die erste Möglichkeit spielt aber für die Versuche keine Rolle, da es um den Einfluss von Systemvirtualisierung geht, der bei dieser gar nicht vorkommt.

Aus diesen möglich Kombinationen ergeben sich durch Permutation fünf mögliche Versuchsszenarien, die in Abbildung 2.8 dargestellt sind. Dabei werden die Szenarien 2 und 2a für die

späteren Versuche als eines betrachtet, da der Fokus bei diesem Szenario auf der Anwendung im Gast liegt.

Aus diesen Szenarien sind nun konkrete Versuche zu definieren. Dies wird in Kapitel 4 geschehen, nachdem die in Frage kommenden Virtualisierungsprodukte fest stehen.

3 State of the Art

In diesem Kapitel werden verschiedene Virtualisierungsprodukte vorgestellt. Anschliessend werden im Abschnitt Related Work andere Ansätze präsentiert.

3.1 Überblick über Produkte zur Virtualisierung

Wie der Titel bereits andeutet liegt der Fokus dieser Arbeit auf Virtualisierungsprodukten, die die Intel-VT Technologie unterstützen. Eine Auswahl soll in diesem Abschnitt vorgestellt werden. In der Wikipedia findet sich unter [Wik08] eine Liste von Virtualisierungsprodukten für die Systemvirtualisierung. Grundsätzlich lassen sich Virtualisierungsprodukte hinsichtlich des vorgesehenen Anwendungsgebietes einteilen. Hier sind vier wichtige Kategorien:

- Desktopvirtualisierung
- Servervirtualisierung
- Virtualisierung für eingebettete Systeme
- Entwicklungssysteme

Da Entwicklungssysteme nicht Gegenstand dieser Arbeit sind, werden sie auch nicht weiter betrachtet.

Virtualisierung für eingebettete Systeme ist ein relativ neues Gebiet. Bis vor kurzem konzentrierten sich die kommerziellen Lösungen auf Mikrokernelansätze. Da es sich dabei nicht um echte Systemvirtualisierung handelt werden sie im Rahmen dieser Arbeit nicht betrachtet. Eine Auswahl findet sich im Abschnitt Related Work.

Mit dem RTS Hypervisor[1] gibt es ein erstes Produkt, welches auch Intel-VT unterstützt. Da es erst seit Juni 2008 Windows unterstützt konnte es in diesem Rahmen leider nicht berücksichtigt werden.

Zahlreiche Virtualisierungsprodukte gehören in die beiden verbleibenden Kategorien: Desktopvirtualisierung und Servervirtualisierung. Aus diesen musste ich eine Auswahl an Kandidaten für die Versuche treffen. Da Produkte für die Servervirtualisierung stark auf ihren Anwendungsbereich zugeschnitten und zudem sehr teuer sind, habe ich mich dabei auf Desktopvirtualisierungsprodukte konzentriert. Im Folgenden werden einige verbreitete Produkte präsentiert, die Intel-VT unterstützen.

[1] http://www.real-time-systems.com/

3.1.1 KVM

Bevor Intel-VT und AMD-V auf den Markt kamen, war QEmu eine weit verbreitete Systemvirtualisierungslösung, die Emulation als Ansatz nutzte. QEmu hat zwar teilweise eine schlechtere Performance, als kommerzielle Alternativen, steht dafür aber unter der GPL.

KVM ist eine Modifikation von QEmu, die jedoch auf Vollvirtualisierung statt Emulation setzt. Alle anderen Eigenschaften, von der emulierten Hardware über Imageformate bis hin zur Kommandozeilensyntax sind dieselben wie bei QEmu. KVM läuft zurzeit nur unter Linux, Ports sind jedoch in Arbeit. Sollte das System, auf dem KVM ausgeführt wird, keine Vollvirtualisierung unterstützen, so wird automatisch Emulation von QEmu benutzt. Um aber Vollvirtualisierung zu ermöglichen braucht man Kernelmodule, die seit 2.6.20[2] Bestandteil des offiziellen Kernels sind. KVM und alle benötigten Komponenten sind ebenfalls unter der GPL oder der LGPL veröffentlicht.

KVM ist für den Einsatz in eingebetteten Systemen gut geeignet, da es zahlreiche „exotische“ Betriebssysteme, wie ReactOS und AROS unterstützt. MS-DOS kann auf KVM hingegen nicht ausgeführt werden. Eine komplette Liste der unterstützten Betriebssysteme findet sich unter [kw].

3.1.2 Xen

Xen ist ursprünglich für andere Anwendungen entwickelt wurden. Es nutzt das Konzept der Paravirualisierung um mehrere modifizierte Versionen von Linux oder Solaris gleichzeitig auf einem PC ablaufen zu lassen. Seit der Einführung von Intel-VT und AMD-V werden diese von Xen ebenfalls unterstützt um auch unmodifierzierte Versionen von Betriebssystemen laufen zu lassen. Dabei verwendet auch Xen Bestandteile von QEmu um Hardware zu emulieren. Auch Xen wurde unter der GPL veröffentlicht.

3.1.3 VMWare

Im Unterschied zu den beiden Vorgängern ist VMWare ein kommerzielles Produkt. Es ist schon länger auf dem Markt und hat vor der Unterstützung der Vollvirtualisierung auf Emulation aufgebaut. Von VMWare gibt es verschiedene Versionen, hier wird VMWare Workstation betrachtet, die für den Desktopeinsatz konzipiert ist. Eine Besonderheit von VMWare, die der verbesserten Performance im Desktopbereich dient, sind die VMWare Tools. Dabei handelt es sich um ein Software- und Treiberpaket für das Gastsystem.

Im Unterschied zu den beiden Vorgängern läuft VMWare auch unter Windows. Dies ist zwar nur im Zusammenhang mit Szenarien ohne Echtzeitanforderungen von Interesse, aber dort könnte wegen der breiteren Hardwareunterstützung durch Windows, VMWare die einzige Option darstellen. VMWare unterstützt weniger Gastbetriebssysteme als KVM, im Unterschied zu diesem kann aber das weit verbreitete MS-DOS als Gast eingesetzt werden ([VMW08]).

[2] http://www.heise.de/english/newsticker/news/82344

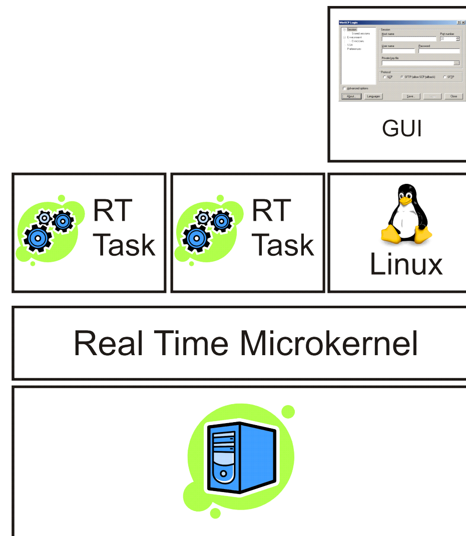


Abbildung 3.1: Paravirtualisierung mit einem Mikrokernel

3.2 Related Work

3.2.1 Mikrokernel-Ansätze

Eine Möglichkeit einige Vorteile von Virtualisierung zu erhalten, ohne Vollvirtualisierung zu nutzen, bietet ein Mikrokernel-Ansatz, wie in Abbildung 3.1 beispielhaft dargestellt. Dieser wurde speziell für eingebettete Systeme mit einem hohen Maß an Funktionalität entworfen. Kommerzielle Beispiele werden in [Kai08] und [Hei07] präsentiert. Das ursprüngliche Ziel bestand darin, die Funktionalität eines Echtzeitbetriebssystem und eines Benutzerbetriebssystems (in den Beispielen Linux) zu vereinen.

Die Idee besteht nun darin, dass man einen echtzeitfähigen Mikrokernel direkt auf der Hardware laufen lässt um auf dessen Basis die Echtzeittasks ablaufen zu lassen. Auf diesem Mikrokernel läuft nun ein modifiziertes Linux im User Mode. Das Linux ermöglicht es, dass auch Tasks, die ein umfangreiches Betriebssystem brauchen, wie z.B. graphische Benutzeroberflächen, auf demselben Gerät laufen können. Da hier keine Hardware emuliert wird, kann man diesen Ansatz als Paravirtualisierung einordnen.

Von den oben dargestellten Anforderungen werden hier Stabilität, Sicherheit, Performance und Echtzeitfähigkeit komplett adressiert. Da jedoch kein komplettes System nachgestellt wird, kann der Entwickler nur auf Betriebssysteme zugreifen, die vom Anbieter zur Verfügung gestellt werden. Zur Entwicklung von Echtzeittasks ist der Entwickler sogar vollständig auf spezifische Schnittstellen des Systems angewiesen.

Auch bei der Hardwareauswahl kann es Probleme geben, denn damit ein Echtzeittask auf eine Hardware zugreifen kann, muss der Mikrokernel auf diese zugreifen können. Somit werden die anderen Anforderungen bei diesem Ansatz nicht erfüllt.

3.2.2 ARM TrustZone

Die ARM Architektur erfreut sich eines hohen Verbreitungsgrades, wenn es um eingebettete Systeme geht. Vor allem bei Anwendungen, die hohe Performance verlangen, wie z.B. bei Mobiltelefonen ist ARM Marktführer. Dort spielt das oben dargestellte Sicherheitsszenario eine wichtige Rolle. Daher wurde die ARM-Architektur um Vollvirtualisierung erweitert, wobei ein besonderes Augenmerk auf die Sicherheit gelegt wurde. Das Ergebnis nennt sich ARM TrustZone und wird in [CB07] ausführlich beschrieben.

Im Unterschied zu anderen Virtualisierungslösungen werden hier exakt zwei virtuelle Umgebungen, genannt Welten, zur Verfügung gestellt. Eine der Welten ist dabei „sicher“ und die andere „unsicher“. Anwendungen, die in der „unsicheren“ Welt ablaufen, können auf Speicher und Ressourcen der „sicheren“ Welt nicht zugreifen.

Mit diesem Ansatz erfüllt ARM alle Anforderungen des Sicherheitsszenarios, unter der Voraussetzung, dass zwei virtuelle Umgebungen ausreichen.

4 Versuchsvorbereitungen

Bevor mit den Versuchen begonnen werden kann, müssen noch einige Vorbereitungen getroffen werden:

- Die Hardware und die Software(insbesondere Betriebssysteme und VMs) für die Versuche müssen sinnvoll gewählt werden.
- Die benötigten Benchmarks müssen ausgewählt und gegebenenfalls angepasst werden.
- Die Konfiguration der Versuche und die Messungen sowie deren Ziele müsssen definiert werden.
- Das Betriebssystem muss sinnvoll konfiguriert werden, die Konfiguration sollte mit Hilfe eined Vorversuchd überprüft werden.

4.1 Auswahl der Hard- und Software

Bislang wurden keine Angaben hinsichtlich der für die Versuche zu verwendenden Hardware und Software gemacht. Dies geschah mit dem Ziel, den bischerigen Teil der Arbeit möglichst allgemein zu halten. Dies muss nun für die Versuchsplanung nachgeholt werden.

4.1.1 Hardware

Um dem Umfang einer Diplomarbeit gerecht zu werden, mussten vor allem im Hinblick auf die praktischen Versuche Einschränkungen getroffen werden. Die praktischen Versuche werden auf einer Plattform durchgeführt. Die Entscheidung fiel auf ein Notebook mit einem Intel Core 2 Duo Prozessor. Dieser unterstützt die titelgebende Technologie Intel-VT [1], was Versuche mit Vollvirtualisierung ermöglicht. Ein Notebook mit einem neueren AMD-Prozessor, der die AMD-V Technologie unterstützt, wäre eine gleichwertige Alternative, die voraussichtlich ähnliche Ergebnisse liefen würde, da hier keine besonderen Einschränkungen gegenüber Intel-VT bekannt sind.

4.1.2 Virtuelle Maschine

Um alle Versuche auf allen Produkten gleichermassen durchführen zu können, wurden vor allem virtuelle Maschinen betrachtet, die auf einem Host-Betriebssystem aufsetzten. Des weiteren wurde der Fokus auf Produkte gesetzt, die für die Desktopvirtualisierung entwickelt

[1] http://www.intel.com/technology/virtualization/

wurden, da diese näher an dem getesteten Einsatzgebiet liegen als entsprechende Serverprodukte. Ausserdem mussten die Produkte natürlich die gewählte Hardwareplattform - IntelVT unterstützen. Damit standen die folgenden Produkte zur Auswahl: Xen, KVM, VMWare.

Da Xen und KVM einige Komponenten miteinander teilen und Xen einen geringeren Funktionsumfang bezüglich Vollvirtualisierung besitzt, wurde auf Versuche mit Xen verzichtet. Somit sind KVM und VMWare als Versuchsplattformen übrig geblieben. KVM hat zusätzlich den Vorteil, dass es sich dabei um eine Erweiterung von QEmu handelt. Damit ist es möglich, falls Vollvirtualisierung Probleme bereitet oder auf dem Host-System nicht zur Verfügung steht, dieselbe virtuelle Maschine ohne Vollvirtualisierung auf QEmu auszuführen. Dieser Vorteil wird auch im Rahmen der Versuche benutzt und ein Teil der Versuche mit KVM werden auch mit QEmu durchgeführt. Damit entsteht, ohne größeren Aufwand, ein Vergleich einer Vollvirtualisierungs- und einer Emulationslösung.

4.1.3 Host-Betriebssystem

Als Host-Betriebssystem kam nur Linux infrage, denn es ist das einzige von den virtuellen Maschinen unterstützte Betriebssystem mit Echtzeitfunktionalität. Linux wird zwar mittlerweile häufig als Betriebssystem für eingebettete Anwendungen eingesetzt, ist aber in seiner Grundversion kein klassisches Echtzeitbetriebssystem.

Um auf Linux Echtzeitanforderungen einhalten zu können, wurden in der Vergangenheit spezielle Erweiterungen wie RTLinux oder das frei verfügbare RTAI eingesetzt. Dagegen enthalten die 2.6.X Kernel bereits viele Eigenschaften, die das Betriebssystem für den Echtzeiteinsatz geeigneter erscheinen lassen. Auch unterstützen die Scheduler Echtzeitprioritäten. Trotzdem gibt es noch einige Lücken, die dem Einsatz von Linux 2.6.X als Echtzeitbetriebssystem im Wege stehen. Diese werden durch den CONFIG_PREEMPT_RT Patch Set (RT-Patch) behoben. Die Änderungen durch den RT-Patch werden in [McK05] beschrieben.

Als Gast-Betriebssystem für Echtzeitanwendungen wird dasselbe Betriebssystem wie auf dem Host eingesetzt. Für grafische Anwendungen wird dagegen Windows XP zum Einsatz kommen. Dieses ist zum einen für diesen Anwendungsbereich verbreiteter als Linux. Zum anderen hat man so eine gewisse Vielfältigkeit bei den Gast-Betriebssystemen.

Windows-Version Hier wurde Windows XP Home mit Service Pack 2 gewählt, da es die neueste Windows Version war, die zum Zeitpunkt der Versuche stabil unter KVM lief.

Linux-Version Hier habe ich mich für das damals aktuelle Ubuntu 7.10 entschieden. Zum Zeitpunkt der Versuche war gerade der 2.6.25 Kernel aktuell. Diesen habe ich dem mitgelieferten 2.6.22 Kernel vorgezogen, da dieser einen neuen Scheduler - Completely Fair Scheduler (CFS) - enthielt [Cor07b]. Ursprünglich hatte ich vor, das darin eingebaute CFS Group Scheduling [Cor07a] zu nutzen. Obwohl diese Option der wichtigste Grund für die Entscheidung zugunsten des 2.6.25 Kernels war, habe ich sie am Ende nicht wahrgenommen.

Nachdem die Entscheidung auf den 2.6.25 Kernel gefallen ist, musste ich etwas warten, denn ich konnte nur eine Kernelversion verwenden, für die der RT-Patch zur Verfügung stand. Das war erst mit der Version 2.6.25.4 der Fall.

4.2 Versuchsdefinition

4.2.1 Allgemeines

Durch die Auswahl der Hardware und der Betriebssysteme ist ein Rahmen entstanden, in dem nun aus den Versuchsszenarien, die im Abschnitt 6.2 definiert wurden, Versuche gestaltet werden können. Diese bilden die Szenarien ab, enthalten aber konkrete Angaben zu den verwendeten Betriebssystemen.

Bevor in den Versuchen der Hauptuntersuchungspunkt - die Echtzeitfähigkeit mit Hilfe von einem Benchmark überprüft werden kann, muss zunächst sichergestellt werden, dass der Versuchsaufbau insgesamt stabil funktioniert. Somit besteht jeder Versuch aus drei Teilen deren Einordnung in Abbildung 4.1 dargestellt ist:

1. Installieren - einrichten des Host und Gastsystems
2. Testen - Überprüfung ob der Aufbau startet und stabil läuft; subjektive Einschätzung ob die Performance der grafischen Oberfläche ausreicht
3. Benchmarks - Durchführung von Messungen der Latenz mit dem Ziel die Echtzeitfähigkeit des Systems zu überprüfen sowie Messergebnisse zu erhalten

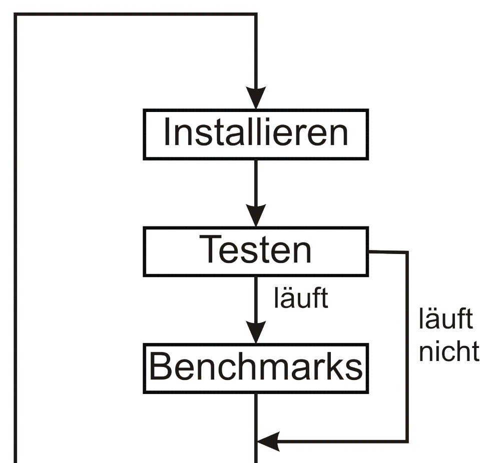
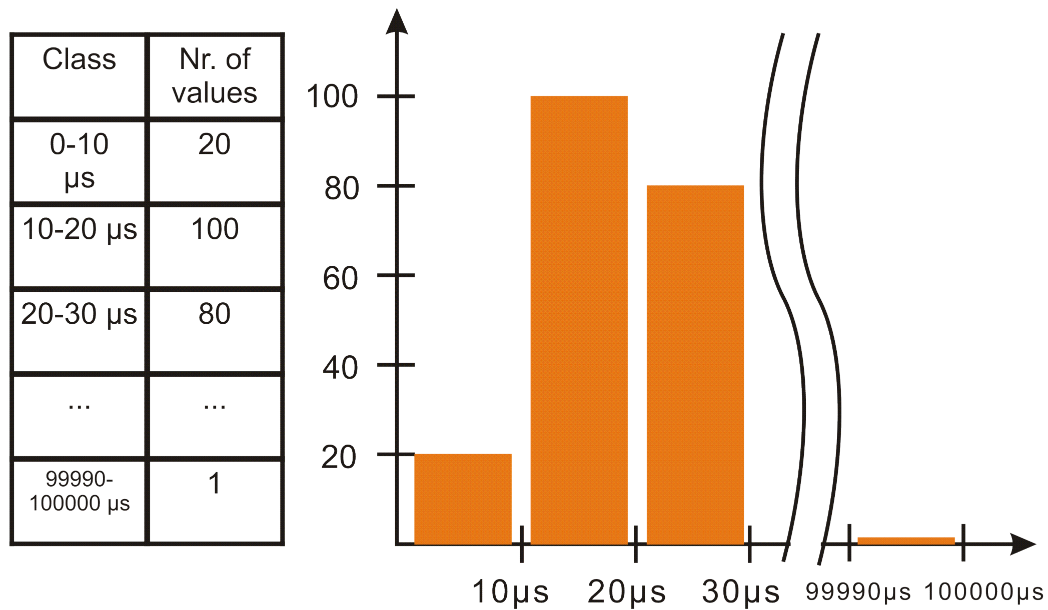


Abbildung 4.1: Versuchsablauf

4.2.2 Messungen

Die Benschmarks werden nur durchgeführt, wenn der Versuchsaufbau startet und eine ausreichende Stabilität festgestellt wurde, da andernfalls ein Einsatz in einem eingebetteten System ausgeschlossen ist. Dabei werden pro Versuch mehrere verschiedene Messungen durchgeführt. Damit sollen verschiedene Ziele erreicht werden. Zum einen will man Vergleiche anstellen für die zusätzliche Messungen notwendig sind. Zum anderen wird der Einfluss der folgenden Zusatzfaktoren auf die Latenzen unter Laborbedingungen überprüft:

- Festplattenzugriffe



Class	Nr. of values
0-10 µs	20
10-20 µs	100
20-30 µs	80
...	...
99990-100000 µs	1

Abbildung 4.2: Beispielhistogramm

- Prozess der versucht viel Rechenzeit zu verbrauchen
- Normale nutzung der grafischen Oberfläche
- Wiedergabe eines MPEG Videos

Da dies bei Vorversuchen Probleme bereitet hat, wird zusätlich zu den oben erwähnten Faktoren auch noch das Verschieben von Festern überprüft.

Als Ergebnis der Messungen wird das Benchmarkprogramm ein Latenzhistogramm abspeichern. Dieses enthält die absolute Häufigkeit des Auftretens einer Bestimmten Bandbreite von Latenzen (z.B. die Anzahl der Messergebnisse zwischen 10μs und 20μs). Diese Bereiche in denen die Messergebnisse zusammengefasst werden nennt man Klassen. Die Echtzeituhr von Linux hat eine Auflösung von 1 ns, allerdings ist es sicher nicht Sinnvoll alle Werte mit dieser Genauigkeit abzuspeichern. Durch Testmessungen habe ich festgestellt, dass die meisten Latenzen je nach Versuchsaufbau zwischen 5μs und 10000μs(=10ms) betragen.

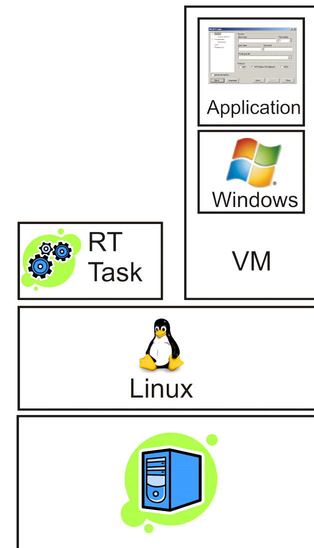
Mit diesen Erkenntnissen habe ich nun die Breite und Anzahl der Klassen festgelegt. Da der Fokus auf weichen Echtzeitanwendungen liegt, sind die genauen Messwerte oberhalb des zehnfachen des höchsten Testwertes sicher nicht interessant (wohl aber deren Anzahl, die man entweder speichern könnte oder als Differenz zwischen der Anzahl der Messungen und Anzahl der Messergebnisse im Histogramm ermitteln könnte). Als Folge werden nur Werte bis 100000μs im Histogramm abgespeichert. Bei der Klassenbreite habe ich mich für 10μs entschieden. Diese führt zu einem gewissen Genauigkeitsverlust bei einfachen Experimenten, wo die Messwerte zwischen 5μs und 200μs schwanken. Dieser Genauigkeitsverlust hat allerdings voraussichtlich keine Praxisrelevanz, da die Genauigkeit so immer noch etwa 1/20 der Maximallatenz beträgt und so innerhalb deren Fehlertoleranz liegt. Auf der anderen Seite erlauben derartig breite Klassen es dem Leser, sich eine Meinung über die Messwerte zu bilden, ohne eine (grafische) Auswertung vorzunehmen. Ein Beispielhistogramm mit den

gewählten Klassen ist in Abbildung 4.2 dargestellt.

Zusätzlich zu den zahlreichen Messungen bei denen die Latenzhistogramme als Ergebnis entstehen, wird für jeden Versuch auch eine längere Messung durchgeführt um die maximal aufgetretene Latenz zu ermitteln. Diese Messung ist notwendig, da die maximale Latenz höher als 100000μs sein kann und dann nicht in den Latenzhistogrammen zu sehen ist.

Neben dem Histogramm und der maximallen Latenz, gibt es noch eine weitere Größe, die für die Entscheidung über die Echtzeitfähigkeit eine zentrale Rolle spielt - das 99,5%-Quantil. Das X%-Quantil gibt dabei an, welcher Wert die X% besten Werte vom Rest trennt. Welches Quantil konkret von Interesse ist, hängt von der tatsächlichen Anwendung ab. Dabei sind oft vor allem höhere Quantile interessant. Diese wären jedoch bei den geplanten 10000 Werten pro Messung nicht signifikant.

4.2.3 Versuch 1 - Echtzeitanwendung auf dem Host mit grafischer Anwendung im Gast



Überblick

Im ersten Versuch geht es darum, zu überprüfen, ob man mit Hilfe von Virtualisierung eine grafische Anwendung und eine Echtzeitanwendung, die verschiedene Betriebssysteme voraussetzen, auf einer Maschine ablaufen lassen kann. Hier wird der einfachste denkbare Aufbau verwendet, bei dem die grafische Anwendung in einer virtuellen Maschine läuft. Statt der Echtzeitanwendung wird, wie in allen anderen Versuchen auch, ein Echtzeitbenchmark eingesetzt. Dieser soll die Echtzeitfähigkeit des Aufbaus bestätigen.

Da ein Zweikernprozessor zum Einsatz kommt, wird der Benchmark auf beiden Prozessorkernen ausgeführt. Um sinnvoll die Auswirkungen der virtuellen Maschine auf einen der Kerne zu überprüfen, ist es sicher sinnvoll der virtuellen Maschine einen festen Prozessorkern zuzuordnen. Dies geschieht nur im Falle von KVM und QEmu, da es im Fall von VMWare nicht möglich ist.

Messungen

In diesem Versuch werden viele verschiedene Messungen durchgeführt. Bei den Grundmessungen wird die Latenz auf dem Hostsystem gemessen, während in der jeweiligen virtuellen Maschine mit Windows XP jeweils folgendes durchgeführt wird:

- nichts
- kopieren von großen Dateien
- Wiedergabe eines MPEG Videos
- normale Nutzung von Windows-Programmen, wie Notepad und Paint

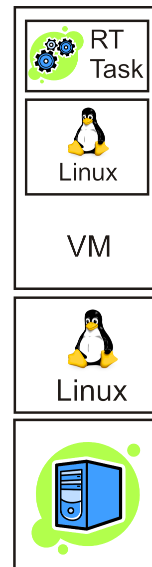
Diese Messung soll die Echtzeitfähigkeit des Versuchsaufbaus allgemein bestätigen und die Auswirkung der verschiedenen Typen von möglichen Benutzeraktivitäten auf die Latenzen zeigen.

Um die Einordnung der Ergebnisse zu erleichtern, werden zusätzlich zu diesen Grundmessungen die folgenden Vergleichsmessungen durchgeführt:

- auf dem Hostsystem wird ein Linux ohne RT-Patch ausgeführt; damit wird die Auswirkung des Patches auf die Echtzeitfähigkeit mit und ohne virtuelle Maschine überprüft
- es wird dar keine virtuelle Maschine ausgeführt; damit wird der Einfluss der virtuellen Maschine auf die Latenzen sichtbar gemacht

Als längere Messung wird die Grundmessung mit der Wiedergabe des MPEG Videos für jede virtuelle Maschine wiederholt. Am Ende wird lediglich die maximale Latenz notiert.

4.2.4 Versuch 2 - Echtzeitanwendung im Gast



Überblick

Im zweiten Versuch soll die Echtzeitfähigkeit innerhalb der virtuellen Maschine überprüft werden. Hierzu wird der Echtzeitbenchmark dieses in der virtuellen Maschine ausgeführt. Sowohl auf dem Host als auch auf dem Gast kommt ein Linux mit RT-Patch zum Einsatz, dessen Bedeutung im ersten Versuch demonstriert wurde. Im Unterschied zum ersten Versuch muss hier sicher gestellt werden, dass die virtuelle Maschine Echtzeitpriorität erhält. Andernfalls ist an eine Echtzeitfähigkeit des darin ausgeführten Benchmarks nicht zu denken.

Der Versuch ist an das zweite Szenario angelehnt. Dennoch werden keine Messungen auf dem Host durchgeführt, da dies ja schon im ersten Versuch durchgeführt wurde.

Messungen

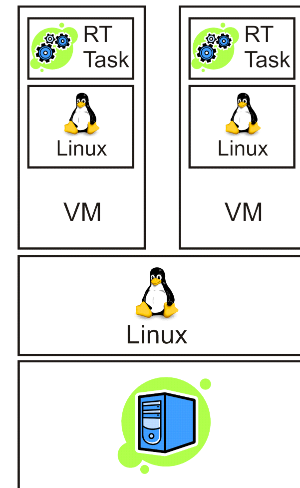
In diesem Versuch wird ein Satz Messungen durchgeführt. Wieder kommen alle virtuellen Maschinen zum Einsatz, parallel zu dem Benchmark wird dabei jeweils auf die folgende Art Last erzeugt:

- nichts
- kopieren von großen Dateien
- ausführen eines Prozesses der versucht viel Rechenzeit zu verbrauchen aber niedrigere Priorität hat

Beim Langzeittest gibt es bei diesem Versuch keine Zusatzlast, da diese auch in einem vermutlichen Szenario mit verschärften Echtzeitanforderungen wohl auf ein Minimum reduziert werden würde.

4.2.5 Versuch 3 - Zwei virtuelle Maschinen jeweils mit Echtzeitanwendung

Überblick



Beim dritten Versuch kommen nun zwei gleich konfigurierte virtuelle Maschinen zum Einsatz. Diese sind Kopien der virtuellen Maschinen aus dem zweiten Versuch. Die Latenzen werden auf beiden Maschinen gleichzeitig gemessen.

Messungen

Da die Auswirkungen der verschiedenen Lasten schon in Versuch 2 überprüft wurden, werden hier nur noch zwei verschiedene Messungen pro virtuelle Maschine durchgeführt:

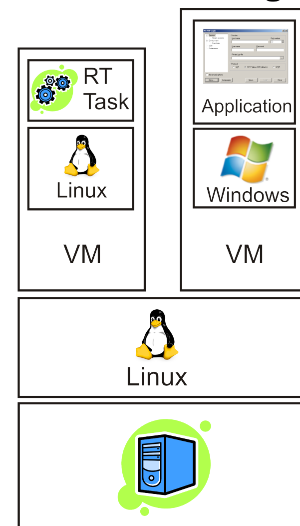
- Leerlauf
- Last mit niedrigerer Priorität

Die zweite Messung ist hier weiterhin notwendig, da die Verteilung der Prozessorlast ja gerade die Schwierigkeit dieses Aufbaus darstellt.

Der Langzeittest entspricht dem aus Versuch 2 und wird bei Leerlauf durchgeführt.

4.2.6 Versuch 4 - Zwei virtuelle Maschinen je eine mit Echtzeitanwendung und grafischer Anwendung

Überblick



Der vierte Vesuch entspricht weitgehend dem ersten. Nur läuft hier die Echtzeitanwendung bzw. der Benchmark in einer virtuellen Maschine und nicht direkt auf dem Hostsystem.

Messungen

Auch hier wird nur ein Teil der Messungen aus Versuch 1 durchgeführt. Als Last wird hier nur die Videowiedergabe verwendet, so dass nur eine Messung pro virtuelle Maschine übrig bleibt. Der Langzeittest ist identisch zu dem aus Versuch 1.

4.3 Konfiguration

In diesem Kapitel werden Konfigurationsdateien und nichttriviale Kommandozeilenbefehle, die ich für die Versuche benutzt habe, wiedergegeben. Die Konfiguration wird, soweit sinnvoll, so vorgestellt, wie ich sie mir vor den Versuchen überlegt habe. Änderungen an der Konfiguration, die im Laufe der Versuche notwendig wurden, werden bei den Versuchsergebnissen dargestellt.

4.3.1 Linux-Kernel

Für die Versuche habe ich eine normale Version des Kernels und eine mit RT-Patch übersetzt. Dabei hat mir das sehr gute How To unter [fal] sehr geholfen. Ich musste beide Kernel selber übersetzen, da ich den Kernel 2.6.25.4 verwenden wollte, für den es, zum Zeitpunkt der Versuche, keine vorkompilierte Version für Ubuntu gab.

Für den normalen Kernel habe ich die Konfigurationsdatei aus dem mitgelieferten 2.6.22 Kernel übernommen. Die Konfigurationen für die aktuell installierten Kernel können aus dem /boot Verzeichnis kopiert werden. Für den Kernel mit dem RT-Patch waren einige Änderungen notwendig. Von der offiziellen Homepage [2] konnte ich den RT-Patch für den von mir verwendeten Kernel finden und runterladen. Mit dem Befehl,

```
bzip2 -dc ../patch-2.6.25.4-rt4.bz2 | patch -p1
```

ausgeführt im Quellcodeverzeichnis, habe ich den Patch angewendet. Wieder wurde die Konfiguration vom mitgelieferten 2.6.22-rt Kernel übernommen. In diesem Fall musste ich jedoch einige Änderungen vornehmen. Hierfür habe ich `menuconfig` eingesetzt. Dabei wurden alle Power Management Optionen außer ACPI sowie der virtuelle Speicher deaktiviert, da diese Features die Messung der Echtzeitfähigkeit stören könnten. ACPI das früher in Echtzeitsystemen ebenfalls nicht verwendet wurde, darf nicht mehr deaktiviert werden, da es seit der Version 2.6.18-rt6 des Linux Kernels für High Resolution Timers (HRT) zwingend benötigt wird [FS]. HRT wiederrum erlaubt das Messen von Zeiten mit einer Genauigkeit von 1ns statt 1ms. Abbildung 4.3 und Abbildung 4.4 zeigen Screenshots von den gemachten Änderungen.

Die erzeugten Kernelpakete habe ich dann sowohl im Host- als auch im Gastsystem verwenden können.

4.3.2 Konfiguration von KVM

Um die Konfiguration zu überprüfen habe ich auf beiden Systemen KVM gestartet und ein Linux von CD gebootet. Für die Ausführung von KVM werden die Kernelpakete kvm und kvm_intel (bzw. kvm_amd für Prozessoren, die AMD-V unterstützen) benötigt, die mit dem Linux-Programm modprobe geladen werden können.

Bei der Ausführung hatte ich das Problem, dass die Maus oft in der rechten unteren Ecke „festklebte“. Dieses Problem kann behoben werden, indem man vor der Ausführung von KVM die Umgebungsvariable SDL_VIDEO_X11_DGAMOUSE auf 0 setzt.

[2] http://www.kernel.org/pub/linux/kernel/projects/rt/

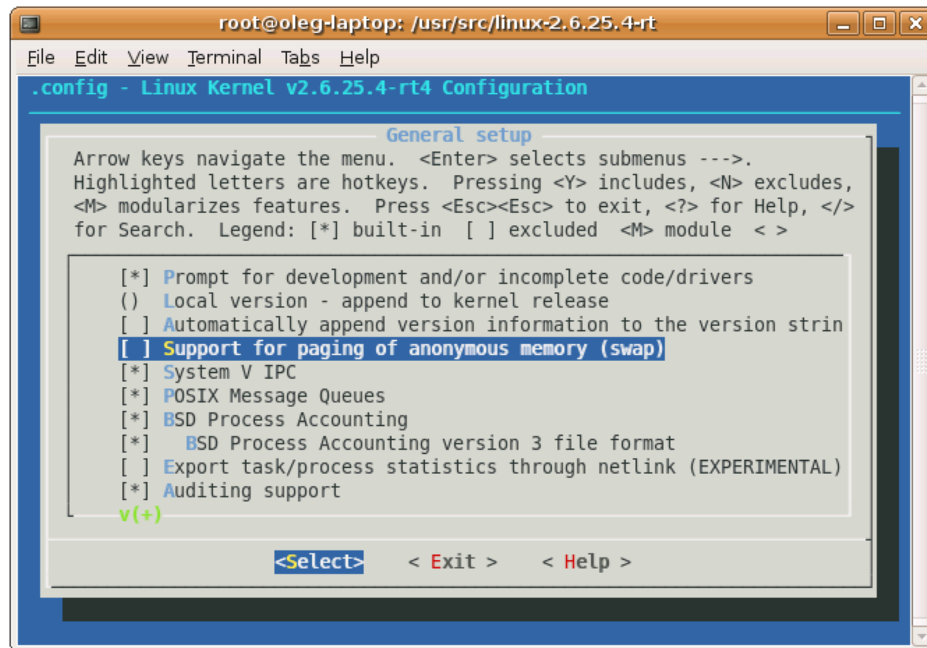


Abbildung 4.3: Deaktivieren von virtuellem Speicher

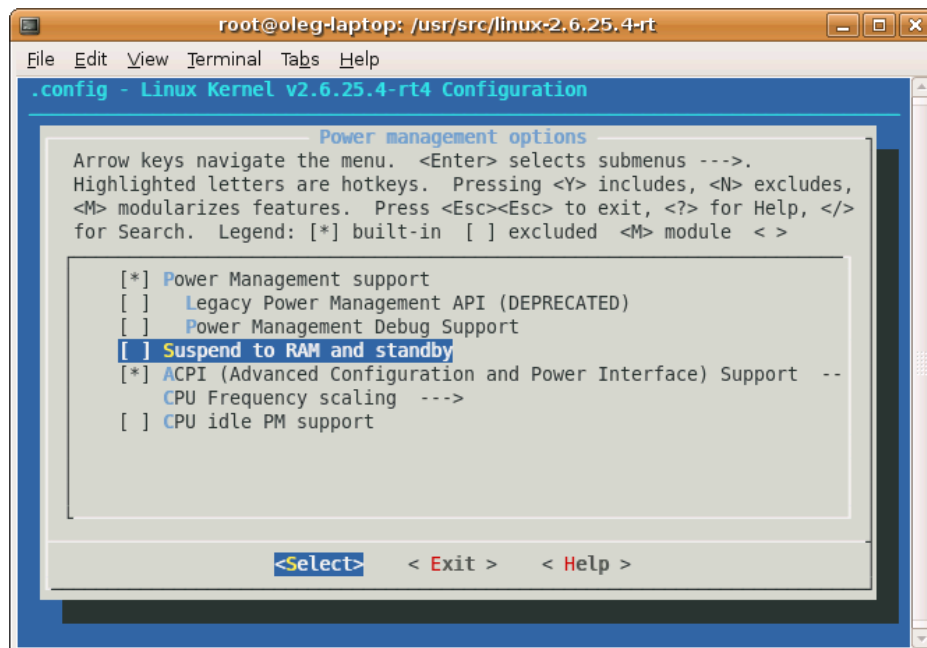


Abbildung 4.4: Deaktivieren von Power Management

Um aus dem Gast heraus auf das Netz zugreifen zu können, habe ich die Kommandozeilenoptionen `-net nic -net user` verwendet, die eine NAT (Network Address Translation) einrichten. Um andersherum vom Host auf den Gast zuzugreifen habe ich Port-Umleitung verwendet. Z.B. leitet die Kommandozeilenoption `-redir tcp:5022::22` lokale Zugriffe über den Port 5022 auf den Port 22 im Gast um. So konnte ich im Gast einen SSH-Server betreiben.

4.3.3 Konfiguration von VMWare

Da es sich bei VMWare um ein kommerzielles Produkt handelt, wurde bei der Konfiguration viel Wert auf Benutzerfreundlichkeit gelegt. Im Unterschied zu KVM gibt es bei VMWare neben Komponenten für das Hostsystem auch welche, die auf dem Client installiert werden. Diese sind nicht notwendig, verbessern jedoch die Performance auf dem Gastsystem. Zusätzlich schalten sie nützliche Features, wie Drag & Drop zwischen dem Gast- und Hostsystem frei. Für beide Installationen gibt es praktische Pearl-Skripte, die ohne Fachwissen ablaufen, solange ein C Compiler, die benötigten (Standard-)Bibliotheken und die Kernel-Header installiert sind. Praktischerweise richtet das Installationsskript für den Host auch eine Bridge ein, über die die Clients sowohl mit dem Host als auch mit dem Netz, in dem sich der Host befindet, Verbindungen aufbauen können.

4.4 Echtzeitbenchmark

4.4.1 Auswahl

Das Benchmarken von Echtzeitbetriebssystemen selbst bietet genug Stoff für eine Diplomarbeit (z.B. [Feu07]). Da diese Arbeit nur einen Überblick bieten soll, war ich auf der Suche nach einem einfachen Tool zum Messen von Latenzen. Dieses muss kostenlos und Open-Source sein, damit ich es, wenn nötig, an meine Bedürftnisse anpassen kann. Am Ende habe ich drei Kandidaten untersucht: Rt-test von IBM, Cyclictest und Interbench. Alle drei verfolgen verschiedene Ansätze:

Rt-test ist eine Sammlung von verschiedenen Benchmarks um verschiedene Aspekte der Echtzeitfähigkeit eines Betriebssystems zu untersuchen. Dabei handelte es sich vor allem um Programme, die überprüften ob das Betriebssystem ein bestimmtes Feature hat oder nicht. Diese Sammlung enthält auch ein kleines Programm zum Messen von Latenzen.

Bei **Interbench** wird ein Szenario-basierter Ansatz verwendet. Interbench misst nur Latenzen. Dabei kombiniert es die Messung mit zwei verschiedenen lasterzeugenden Simulationen. Z.B. Audiowiedergabe bei gleichzeitigem brennen einer CD oder Videowiedergabe bei gleichzeitigem schreiben einer großen Datei.

Cyclictest misst ebenfalls nur Latenzen. Dabei gibt es keine weiteren Funktionen, dafür aber eine Vielzahl von Optionen, um den Messprozess an die eigenen Bedürfnisse anzupassen. Die wichtigste Funktion erlaubt es auf allen auf dem Computer vorhandenen Prozessorkernen gleichzeitig zu messen.

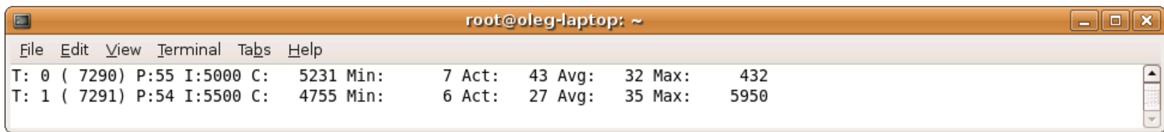


Abbildung 4.5: Ausgabe von Cyclictest

Die Entscheidung fiel am Ende auf Cyclictest. Rt-test bietet im Bereich Latenzmessung weniger Funktionen. Der Ansatz von Interbench hingegen ist sehr gelungen, passt aber nicht in das in diesem Kapitel dargestellte Messkonzept, da die Lasten zum Teil in anderen virtuellen Maschinen entstehen sollen. Cyclictest bietet zusätzlich den Vorteil, dass es den aktuellen Stand der Messung live anzeigt. Diese Funktion hat sich bei der Gestaltung des Demonstrators als nützlich erwiesen.

Eine Beispielausgabe von Cyclictest ist in Abbildung 4.5 zu sehen. Dort sieht man von links nach rechts:

- Threadnummer
- Prozess ID (PID)
- Echtzeitpriorität des Prozesses
- Messintervall (zeitlicher Abstand zwischen zwei Messungen) in μs
- Nummer der aktuellen Messung
- bisher kleinste Latenz in μs
- aktuelle Latenz in μs
- durchschnittliche Latenz in μs
- bisher höchste Latenz in μs

Das Messintervall muss dabei so eingestellt werden, dass zwei Messungen sich im Normallfall nicht überschneiden.

4.4.2 Anpassung

Keines der Benchmarks lieferte die gewünschte Ausgabe - Latenzhistogramm in Datei. Nachdem ich mich für Cyclictest entschieden habe, musste ich es entsprechend anpassen:

1. Ich habe eine neue Konstante eingeführt, die die Zahl der verschiedenen Ergebnisklassen angibt.

```
87 #define RESULTS_SIZE 10000
```

2. Ich habe die threadspezifische Datenstruktur, die für jeden Messthread angelegt wird um eine Variable erweitert, die die Ergebnisse speichert.

```
108 struct thread_stat {
...
```

```
116     long results[RESULTS_SIZE];
...
120 };
```

3. In der Initialisierungsfunktion für die Struktur, werden alle Werte auf Null gesetzt.

```
811 for (ii = 0; ii < RESULTS_SIZE; i++) {
812     stat[i].results[ii] = 0;
813 }
```

4. In der Messfunktion wird der Wert jedes mal inkrementiert, dessen Klasse die gemessene Latenz enthält. Dabei wird der Wert mit dem Index 0 freigehalten, um die Ergebnisse mit dem aus dem nächsten Abschnitt kompatibel zu halten. Dort wird dieser Wert für die Messfehler verwendet.

```
474 if (diff / 10 < RESULTS_SIZE - 1) {
475     stat->results[diff / 10 + 1]++;
476 }
```

5. Die Werte werden in der Aufräumfunktion nun durch Zeilenumbrüche getrennt in einer Datei abgelegt.

```
888 f = fopen(file_name, "w");
889 if(f) {
890     for(ii = 1; ii < RESULTS_SIZE; ii++) {
891         if(fprintf(f, "%ld\n", stat[i].results[ii])
                == -1)
892             break;
893     }
894     fclose(f);
894 }
```

4.4.3 Anpassung für Messung innerhalb der virtuellen Maschine

Überlegungen

Das so angepasste Cyclictest konnte nun für die Messungen auf dem Hostsystem eingesetzt werden. Für den Einsatz innerhalb der virtuellen Maschine ist es hingegen, wie jedes andere nicht speziell angepasste Benchmark auch, nicht geeignet. Dies liegt daran, dass die Uhr innerhalb der virtuellen Maschine, die auch von Cyclictest zur Messung der Latenz verwendet wird, langsamer läuft. Dies passiert immer dann, wenn die virtuelle Maschine nicht genug Rechenzeit bekommt. Setzt man nun ein Benchmark innerhalb der virtuellen Maschine ein, so werden jegliche Verzögerungen, die außerhalb der virtuellen Maschine auftreten, von diesem nicht bemerkt.

Um dieses Problem zu lösen muss die Messung außerhalb der virtuellen Maschine durchgeführt werden. Dazu mussten Signale mit möglichst geringer Verzögerung vom Gast auf

den Hostsystem gelangen. Idealerweise sollte die Methode sowohl für KVM/QEmu als auch für VMWare gleich sein. Zur Auswahl als Kommunikationsmethode standen so die emulierte NIC und eine emulierte serielle Schnittstelle, die auf eine Named Pipe umgeleitet wurde. Da die Emulation der seriellen Schnittstelle von KVM beim Testen aus ungeklärter Ursache Daten verlor, fiel diese Möglichkeit weg. So habe ich mich für UDP(User Datagram Protocol) als Transportweg entschieden.

Die Messung in der virtuellen Maschine funktioniert folgendermaßen: Auf dem Gastsystem wird ein nochmals modifiziertes Cyclictest ausgeführt, dass zusätzlich zur Messung ein UDP-Paket an eine, als Kommandozeilenparameter übergebene, Netzadresse schickt. Die UDP-Pakete enthalten lediglich fortlaufende Nummern um den Empfang in der richtigen Reihenfolge überprüfen zu können, der Wert -1 zeigt das Ende der Übertragung. Auf dem Hostsystem wird nun ein UDP-Server ausgeführt, der die Pakete empfängt und die Zeit zwischen dem Empfang der Pakete misst und im selben Format wie Cyclictest speichert. Pakete, die in falscher Reihenfolge ankommen, werden als Fehler abgespeichert. Messungen, die solche Fehler enthalten, werden wiederholt.

Der von mir gewählte Ansatz reduziert die Genauigkeit der Messung deutlich, da der Jitter im Netzverkehr zwischen Gast und Host als zusätzliche unkontrollierbare Variable hinzukommt, die die Ergebnisse in beiden Richtungen beeinflusst. Fast keine Rolle spielt dagegen die Netzlatenz. Bei Idealbedingungen ohne Jitter, sorgt sie dafür, dass alle Messwerte um 1-2μs größer sind, was weniger als 1‰ des Durchschnittswertes ausmacht. Trotzdem lassen sich die Ergebnisse noch verwenden, weil der Jitter ja dafür sorgt, dass die Messwerte schlechter sind als die Realität (es gibt mehr hohe Werte als bei „perfekter“ Messung). Das heisst, sind die gemessenen Werte gut genug, so sind auch die tatsächlichen Werte gut genug.

Ein hoher Jitter lässt sich während der Messung recht gut bemerken, da er zu negativen Latenzen führt, die es ohne Jitter nicht geben kann. In der Praxis hängen Jitter und Latenz zusammen, da die Netzfunktionen des Gasts mindestens genau so hohe Priorität haben, wie der Benchmark, der darin läuft.

Implementierung

Bei der Implementierung wurde zunächst Cyclictest geändert:

1. Es werden zwei neue globale Variablen hinzugefügt, um die IP-Adresse und den UDP-Port zu speichern.

```
129 static char ipaddr[18] = "";
130 static char udpport[7] = "";
```

2. Der Wert, der über den neuen Kommandozeilenparameter -u bzw. –udp übergeben wurde, wird nun interpretiert und in diesen Variablen abgelegt. (Listing 4.1)

3. Es wurde eine neue Funktion mit dem Namen `sendudp` angelegt, die einen übergebenen Parameter vom Typ `int` an die entsprechende Adresse/Port Kombination sendet.

4. Diese Funktion wird nun in der Messfunktion bei jedem Durchlauf mit dessen Nummer aufgerufen. Nachdem alle Messungen abgeschlossen sin, wird die Funktion zusätzlich mit dem Wert -1 aufgerufen, um das Ende der Übertragung zu signalisieren.

Listing 4.1: Interpretation des neuen Kommandozeilenparameters -u

```
576 static struct option long_options[] = {
...
591     {"udp", required_argument, NULL, 'u'},
...
595 };
596 int c = getopt_long (argc, argv,
            "a::b:c:d:fi:l:no:p:qrst::u:v",
597         long_options, &option_index);
...
600 switch (c) {
...
626 case 'u':
627    temp = strchr(optarg, ':');
628    if(temp && temp - optarg < 17) {
629        strncpy(ipaddr, optarg, temp - optarg);
630        strncpy(udpport, temp + 1, 7);
631    }
...
635 }
```

Nachdem Cyclictest geändert wurde, musste ich nun das Gegenstück für den Host - einen UDP-Server programmieren. Diesen habe ich in C++, meiner bevorzugten Programmiersprache, geschrieben. Um den Aufwand gering zu halten, habe ich die Bibliothek `Sockets` eingesetzt. Dieser lag auch ein Demonstrationsserver bei. Diesen habe ich als Ausgangspunkt genommen, zu dem ich nur noch die benötigte Funktionalität hinzufügen musste.

Die Implementierung selbst ist trivial, weil ich nur den entsprechenden Teil von Cyclictest nachbilden musste. Es gibt jedoch eine wichtige Änderung. Bei diesem Aufbau können, durch Jitter bedingt, negative Messwerte auftreten. Diese werden vom Programm ignoriert. Dies hat jedoch zur Folge, das man nicht mehr aus der Anzahl fehlender Messwerte auf die Anzahl von Latenzen grösser 100000μs schliessen kann. Deshalb wird diese Anzahl als letzter Wert extra gespeichert.

4.5 Timehog

4.5.1 Überblick

Für manche Messungen wird ein Programm benötigt, dass mit niedriger Priorität versucht viel Prozessorzeit zu bekommen. Zusätzlich dazu ist für den Demonstrator ein Programm, in dem man diese Prozessorbelastung variieren kann, wünschenswert. Beide Anforderungen wurden in einem extra von mir geschriebenen Programm mit dem Namen timehog umgesetzt.

Das Programm erstellt einen Prozess, der mit Echtzeitpriorität 20 versucht möglichst viel

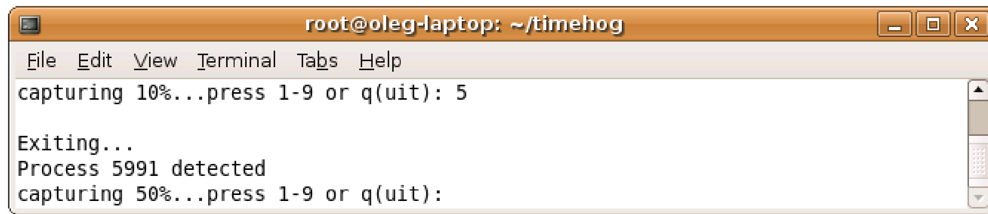


Abbildung 4.6: Oberfläche von timehog

Prozessorzeit zu bekommen. Dem Prozess wird eine feste Affinität für den ersten Prozessorkern zugewiesen. Mit Hilfe des Programms `cpulimit` [3] wird seine Prozessorauslastung auf einen Wert zwischen 10% und 90% in Zehnerschritten limitiert. Dabei kann der Benutzer den Wert durch eine einfache Kommandozeilenoberfläche, die in Abbildung 4.6 dargestellt ist, einstellen.

4.5.2 Implementierung

Das Programm wurde ebenfalls in C++ geschrieben, wobei, aufgrund der Systemnähe, nur C-Funktionen verwendet wurden:

1. Am Anfang wird mit Hilfe der Funktion `fork` ein Prozess erzeugt, der viel Rechenzeit verbraucht und später mit `cpulimit` gezügelt wird.

```
14 pid_t timehogPid = fork();
15 if (timehogPid == 0)
16     for(;;);
```

2. Dem Prozess wird mit Hilfe der Funktion `sched_setaffinity` eine feste Affinität für den ersten Prozessorkern (ID 0) zugewiesen.

```
24 cpu_set_t mask;
25 CPU_ZERO(&mask);
26 CPU_SET(0, &mask);
27 sched_setaffinity(timehogPid, sizeof(mask), &mask);
```

3. Am Anfang wird das Limit auf 10% gesetzt, später kann es vom Benutzer geändert werden

```
32 int limit = 10;
```

4. Danach wird erneut der Befehl `fork` eingesetzt. Diesmal wird ein Prozess für das Programm `cpulimit` erzeugt, während der aktuelle Prozess für die Interaktion mit dem Benutzer verantwortlich ist.

```
30 pid_t cpuLimitPid;
...
36 cpuLimitPid = fork();
```

[3] http://cpulimit.sourceforge.net/

5. Dem neu erzeugten Prozess wird die Priorität 30 zugewiesen. Diese Priorität ist höher als die vom kontrollierten Prozess. Dies ist zwingend notwendig, da sonst auf Einprozessorsystemen der `cpulimit` Prozess keine Prozessorzeit erhalten würde, um den kontrollierten Prozess zu beschränken. Dies würde dazu führen, dass dieser 100% des Prozessors erhält und das ganze System wäre nicht mehr bedienbar. Anschließend wird `cpulimit` mit dem Befehl `execl` gestartet.

```
48 schedParam.sched_priority = 30;
49 if(sched_setscheduler(getpid(), SCHED_RR,
      &schedParam) == -1)
...
55 sprintf(commandBuffer, "-l␣%d", limit);
56 sprintf(commandBuffer2, "-p␣%d", timehogPid);
57 execl("/usr/bin/cpulimit", "cpulimit", commandBuffer,
         commandBuffer2, (char*) NULL);
58 exit(0);
```

6. Analog wird dem timehog Prozess Priorität 20 zugewiesen. Davor wird jedoch mit Hilfe des Befehls `sleep` eine Pause von 1s eingelegt. Diese stellt sicher, dass in dem Moment, wo der Prozess Echtzeitpriorität erhält `cpulimit` bereits läuft. Andernfalls könnte es sich 100% der Prozessorzeit schnappen, bevor `cpulimit` die höhere Priorität erhalten hat.

```
62 sleep(1);

65 schedParam.sched_priority = 20;
66 if(sched_setscheduler(timehogPid, SCHED_RR,
      &schedParam) == -1)
```

7. Während nun `cpulimit` läuft, wird die Benutzerausgabe angezeigt. In einer Schleife wartet das Programm auf eine Eingabe seitens des Benutzers.

8. Entscheidet sich der Benutzer für eine andere Prozessorlast, so wird `cpulimit` mit dem Befehl `kill` beendet. Anschließend werden die Schritte ab Nummer 4 mit dem neuen Wert wiederholt. Hier ist es wichtig, die Priorität des timhog Prozesses wieder auf normal zurückzusetzen. Sonst hätte dieser abermals die Gelegenheit, 100% der Prozessorzeit zu kapern.

5 Versuchsdurchführung und Ergebnisse

In diesem Kapitel werden nun meine Erfahrungen, die ich während der Versuchsdurchführung hatte, präsentiert. Neben den Ergebnissen der Latenzmessung, die helfen sollen, die Entscheidung zu treffen, ob der Einsatz der getesteten Virtualisierungslösungen überhaupt in Frage kommt, werden auch die Probleme aufgezeigt, die ich während der Durchführung hatte. Dies soll zum einen helfen, die Versuche nachzustellen um auf dem gewünschten System messen zu können. Zum anderen können die Probleme so im Falle einer reallen Anwendung in die Planung mit einbezogen werden.

5.1 Überblick über die Versuche

Die Grundstruktur der Versuche wurde in Abschnitt 4.2 dargestellt. Im Laufe dieses Kapitels werden die Versuche nun präzisiert. Die einzelnen Versuche bestehen aus Messungen. Verschiedene Messungen werden, um die Ergebnisse zu interpretieren, miteinander verglichen. Somit ist ein Versuch im Allgemeinen durch den Versuchsaufbau gekennzeichnet. Durch die verschiedenen Vergleiche können im Rahmen eines Versuches zahlreiche verschiedene Ergebnisse gewonnen werden.

Die einzelnen Versuche werden mit den drei ausgewählten Virtualisierungsprodukten durchgeführt. Wie man der nachfolgenden Tabelle entnehmen kann, wurden fast alle möglichen Kombinationen überprüft. Ein rotes X kennzeichnet dabei einen Versuch, bei dessen Durchführung größere Probleme aufgetreten sind, die die Verwendbarkeit der Messwerte einschränken.

	KVM	QEmu	VMWare
Versuch 1 - Echtzeitanwendung auf dem Host mit grafischer Anwendung im Gast	X	X	X
Versuch 2 - Echtzeitanwendung im Gast	X	X	X
Versuch 3 - Zwei virtuelle Maschinen jeweils mit Echtzeitanwendung	X	X	X
Versuch 4 - Zwei virtuelle Maschinen je eine mit Echtzeitanwendung und grafischer Anwendung	X		X

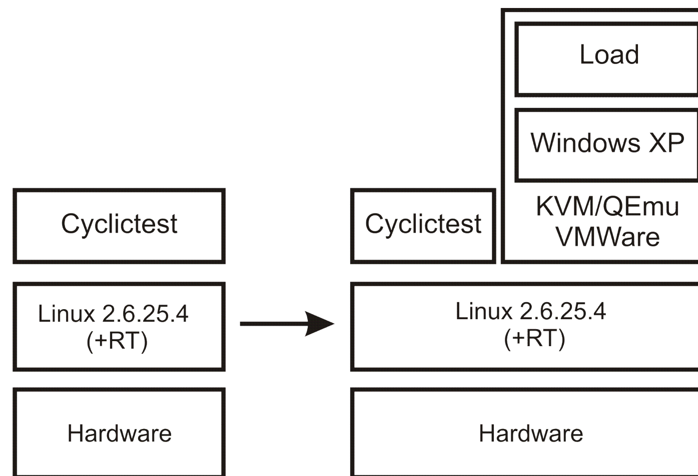


Abbildung 5.1: Aufbau von Versuch 1

5.2 Versuch 1 - Echtzeitanwendung auf dem Host mit grafischer Anwendung im Gast

5.2.1 Beschreibung

Beim ersten Versuch wird von einem Linux-System ausgegangen, auf dem bereits erfolgreich ein Echtzeittask läuft. Ferner gehen wir davon aus, dass dessen Echtzeitanforderungen mit Reserve erfüllt sind. Die Latenzen werden mit Cyclictest gemessen.

Nun wird auf dem System eine virtuelle Maschine mit Windows XP gestartet. In der virtuellen Maschine wird ein Video abgespielt um etwas Last zu erzeugen. Die auf dem Host in diesem Versuchsaufbau gemessenen Latenzen werden im **Vergleich 1** mit denen aus dem Aufbau ohne virtuelle Maschine verglichen. Analog zum ersten Vergleich werden im **Vergleich 2** wieder ein unmodifiziertes Linux und eins mit RT-Patch verglichen.

Im **dritten Vergleich** werden die Auswirkungen verschiedener Lasten innerhalb der virtuellen Maschine auf die Latenzen auf dem Host gemessen. Dabei wird für die Wiedergabe des Videos der Microsoft Media Player verwendet, die anderen Lasten werden manuell erzeugt. Bei der **Langzeitmessung** kommt der Aufbau aus Vergleich 1 zum Einsatz.

Da bei den Messungen eine virtuelle Maschine zum Einsatz kommt, müssen diese für jede der drei bei den Versuchen zum Einsatz kommenden virtuellen Maschinen einzeln gemacht werden. Der Grundlegende Versuchsaufbau wird in Abbildung 5.1 dargestellt.

5.2.2 Vorbereitung

Den größten Teil der Vorbereitung beanspruchte die Installation von Windows auf KVM und VMWare. Zum Glück sind KVM und QEmu vollständig kompatibel, so dass für QEmu kein zusätzlicher Aufwand angefallen ist. Zusätzlich wurden die Betriebssysteme jeweils auf den neuesten Stand gebraucht.

Listing 5.1: Skript zum starten von KVM mit Windows XP

```
1 modprobe kvm
2 modprobe kvm_intel
3 export SDL_VIDEO_X11_DGAMOUSE=0
4 schedtool -a 0 -e kvm [...]
```

Auf dem Host habe ich für jede virtuelle Maschine ein Skript geschrieben, um sie mit nur einem kurzen Aufruf zu starten. Das Skript für KVM ist in Listing 5.1 zu sehen. In den Zeilen 1 und 2 werden die benötigten Kernelmodule, wenn nicht bereits vorhanden, geladen. Zeile 3 ist notwendig um den Bug aus Abschnitt 4.3.2 zu umgehen. In der vierten Zeile wird dann KVM selbst gestartet. Die Parameter von kvm sind ganz gewöhnlich und wurden hier aufgrund ihrer Länge weggelassen. Von Interesse ist hingegen, dass kvm nicht direkt gestartet wird, sondern mit Hilfe des Programms `schedtool`. Es wird im Laufe der Arbeit verwendet, um den gestarteten Prozessen bestimmte Priorität zu geben oder sie einem bestimmten Prozessorkern zuzuweisen. In dieser konkreten Anwendung wird `schedtool` verwendet, um sicher zu stellen, dass die virtuelle Maschine auf dem ersten Prozessorkern läuft.

Das Skript zum starten von QEmu mit Windows XP ist ähnlich, aber es fehlen Zeilen 1 und 2, da hier keine Kernelmodule benötigt werden. Zeile 4 wird lediglich um den Kommandozeilenparameter `-no-kvm` ergänzt. Dieser sorgt dafür, dass QEmu statt KVM zum Einsatz kommt.

Das Skript zum Starten von VMWare ist trivial, denn es enthält nur den Aufruf von VMWare mit der entsprechenden virtuellen Maschine. VMWare überprüft beim Start selbständig, ob die notwendigen Kernelmodule geladen sind und lädt sie gegebenenfalls nach. `Schedtool` kann man mit VMWare leider nicht verwenden um die Affinität einzustellen. Es hat nicht die gewünschte Wirkung und macht das System instabil.

5.2.3 Erwartungen

Beim zweiten Vergleich, wo die Auswirkungen der RT-Patches überprüft werden, habe ich ein deutliches Ergebnis erwartet. Damit verbundene Eindrücke und Probleme, werden auf der Mailingliste linux-rt-users[1] diskutiert. Der Anteil an Latenzen, die ein vielfaches der durchschnittlichen darstellen, sollte ohne RT-Patch deutliches höher ausfallen, denn dies ist ja gerade der Grund ein echtzeitfähiges Betriebssystem einzusetzen. Der negative Effekt dabei ist, dass die durchschnittliche Latenz mit RT-Patch etwas höher ausfallen könnte als ohne.

Bei den andere Vergleichen hatte ich die Hoffnung, dass trotz virtueller Maschine die Echtzeitfähigkeit erhalten bleibt. Dies ist nicht sicher, denn die Kernelmodule von KVM und VMWare sind von den Anpassungen durch den RT-Patch nicht betroffen und an sich nicht kooperativ geschrieben. Das heißt, sie können nicht zu einem Task-Switch gezwungen werden, wie es beim präemptiven Multitasking normalerweise der Fall ist. So lag es im Bereich

[1] http://marc.info/?l=linux-rt-users

des Möglichen, dass die Latenzen durch die parallel ausgeführte virtuelle Maschine ansteigen. Da sie sich aber im Regelfall kooperativ verhalten, war zu hoffen, dass nicht zu viele Latenzen hoch ausfallen und die Echtzeitfähigkeit für Anwendungen mit weichen Echtzeitanforderungen gewahrt bleibt.

5.2.4 Subjektiver Eindruck

Bei diesem Szenario ist neben der Echtzeitfähigkeit auf dem Host natürlich auch die Praxistauglichkeit für den Benutzer interessant. Dieser soll ja in der Regel nicht vom Echtzeitteil erfahren und seine Erfahrung wird sich auf die Bedienung eines Programms auf dem Gastsystem beschränken. Diese ist leider bei keiner Virtualisierungslösung so gut wie ohne Virtualisierung. Man spürt ab und zu Ruckler und die Grafik erscheint einem schon bei einfachen Operationen, wie dem Öffnen und Verschieben von Fenstern, etwas langsamer. Trotzdem lassen sich die Systeme sehr gut bedienen und man kommt nach etwas Eingewöhnung gut zurecht. Dabei machte VMWare den besten Eindruck. Nach der Installation der VMWare Tools liefen viele Operationen sehr flüssig ab. Auch mit KVM war das System noch sehr gut bedienbar. Mit QEmu hingegen, welches ja als einziges keine Vollvirtualisierungslösung darstellt, fühlte sich das Gesamtsystem sehr träge an. Für komplexere Benutzerinteraktionen ist QEmu nur bedingt geeignet. Der Systemstart dauerte bei allen drei Produkten mit mehreren Minuten sehr lange, was aber bei Industrieanwendungen nicht ins Gewicht fallen sollte.

Zusätzlich ist schnell die Inkompatibilität von KVM und VMWare aufgefallen. Wenn die jeweilige virtuelle Maschine auf dem Betriebssystem schon mal ausgeführt wurde, dann startet die andere nicht mehr. So liegt zwischen einer Messung mit KVM und einer Messung mit VMWare stets ein Neustart des PC.

5.2.5 Problem bei der Durchführung

Schon bei der Durchführung der ersten Messungen ist mir ein Problem aufgefallen, welches zu einer Änderung der Konfiguration geführt hat. Die Messergebnisse der Latenz waren unter Last besser als ohne Last. Dieses Problem hat vermutlich folgende Ursache: Steht das Gesamtsystem nicht unter Last, so kommt der Leerlaufprozess öfter mal zum Zuge. Dieser benutzt den Prozessor-Befehl HALT. Bei modernen Prozessoren sorgt dieser Befehl nun dafür, dass der Prozessor in den Stromsparmodus wechselt [Int04]. Da das Aufwachen aus dem Stromsparmodus einige Zeit in Anspruch nimmt, steigt die durchschnittliche Latenz.

Unter Linux existiert eine Möglichkeit dieses Problem zu umgehen. Der Parameter `idle=poll`, der beim Systemstart an den Kernel übergeben werden muss, sorgt dafür, dass ein Leerlaufprozess zum Einsatz kommt, der nicht auf die HLT-Anweisung zurückgreift.

Um die Auswirkungen zu visualisieren, habe ich eine zusätzliche Messung gemacht, die die Auswirkungen des `idle=poll` Parameters auf die Messergebnisse deutlich macht. Das daraus entstandene Diagramm ist in Abbildung 5.2 zu sehen. Man erkennt deutlich, dass mit `idle=poll`, fast alle Messergebnisse unter 20 μs liegen, während es ohne diese Einstellung eine deutliche Streuung gibt.

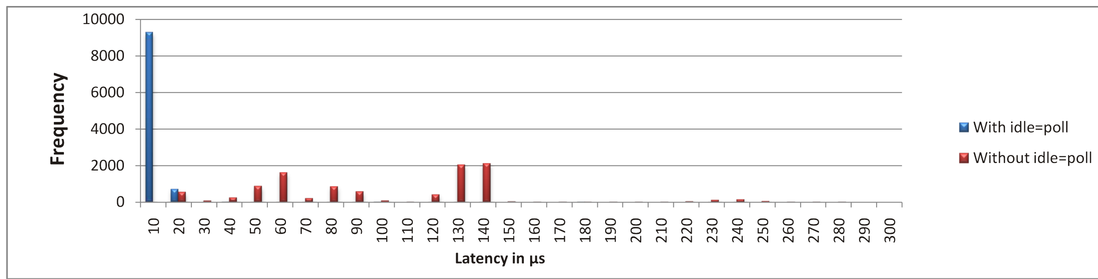


Abbildung 5.2: Auswirkungen von `idle=poll`

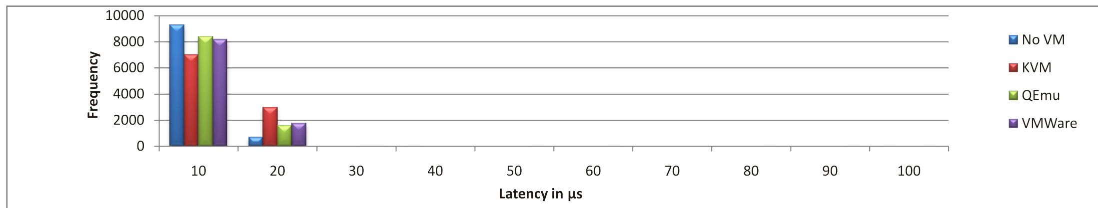


Abbildung 5.3: Latenzen von Cyclictest auf dem Host mit Windows XP im Gast - Thread 0

Als Folge dieser Erkenntniss wurden alle weiteren Messungen dieses Versuchs mit der Einstellung `idle=poll` durchgeführt.

5.2.6 Vergleich 1 - Latenzen mit und ohne parallel laufende VM

Beim ersten Vergleich werden die mit Cyclictest ermittelten Latenzen mit und ohne parallel laufende virtuelle Maschine verglichen. Es gab insgesamt vier Messungen. Bei der ersten Messung läuft keine virtuelle Maschine. Bei den anderen drei Messungen kommen die drei verschiedenen Virtualisierungslösungen zum Einsatz. Abbildung 5.3 und Abbildung 5.4 zeigen die Latenzverteilung des ersten bzw. zweiten Messthreads in Abhängigkeit von der eingesetzten virtuellen Maschine. Abbildung 5.5 zeigt die dabei maximal aufgetretenen Latenzen. Das 99,5%-Quantil liegt in allen Messungen bei unter $20\mu s$.

Aus diesen Daten lassen sich nun folgende Schlüsse ableiten:

- KVM und QEmu haben praktisch betrachtet keine Auswirkungen auf die für Echtzeitfähigkeit relevanten Parameter.

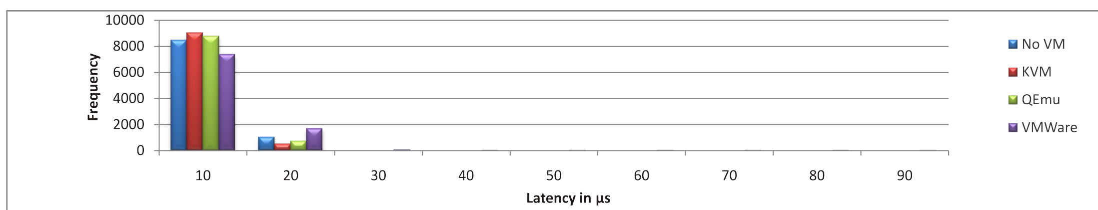


Abbildung 5.4: Latenzen von Cyclictest auf dem Host mit Windows XP im Gast - Thread 1

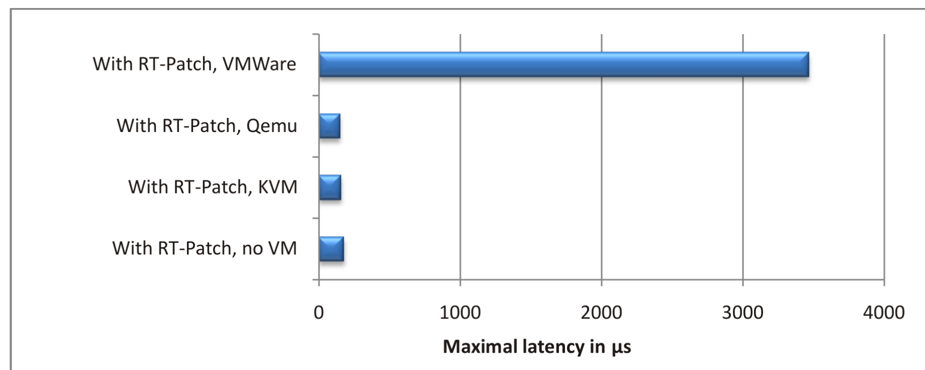


Abbildung 5.5: Maximale Latenzen von Cyclictest auf dem Host mit Windows XP im Gast - beide Threads

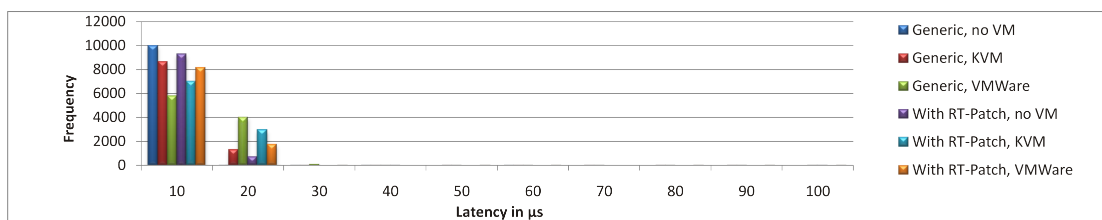


Abbildung 5.6: Latenzen von Cyclictest auf einem Host ohne RT-Patch mit Windows XP im Gast - Thread 0

- VMWare hat keine Auswirkung auf die Verteilung von Latenzen oder auf das 99,5% Quantil, und ist somit für Anwendungen mit weichen Echtzeitanforderungen ohne weiteres geeignet.
- Unter VMWare steigt die maximale Latenz auf über 3,5ms deutlich an. Dieser Aufbau ist somit nicht für Anwendungen geeignet, wo solch hohe Maximallatenzen nicht zulässig sind. Bei der generellen Betrachtung muss man allerdings beachten, dass 3,5ms ein vergleichsweise kleiner Wert ist.

5.2.7 Vergleich 2 - Latenzen mit und ohne RT-Patch

In diesem Abschnitt wird die Bedeutung des RT-Patches für Linux untersucht. Dafür mussten, zusätzlich zu den bereits gemachten Messungen, gleiche Messungen auf einem Host ohne den RT-Patch durchgeführt werden. Dabei wurde auf QEmu als Vergleich verzichtet, da es keinen Eingriff in den Linux-Kernel benötigt. Die Ergebnisse des ersten Threads werden in Abbildung 5.6 dargestellt, der zweite Thread hat sehr ähnliche Werte. Die maximalen Latenzen sind in Abbildung 5.7 zu finden. Das 99,5%-Quantil liegt wieder bei unter 20μs.

Hierbei sieht man folgendes:

- Linux ist auch ohne RT-Patch für Anwendungen mit weichen Echtzeitanforderungen sehr gut geeignet.

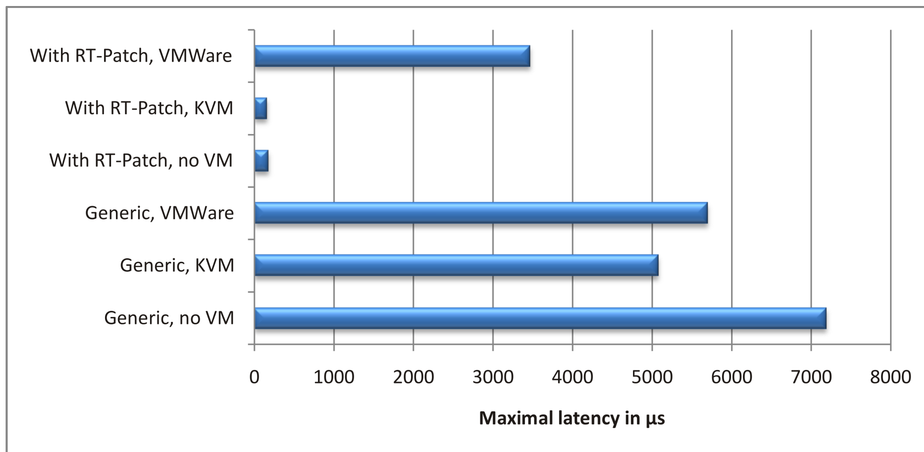


Abbildung 5.7: Maximale Latenzen von Cyclictest auf einem Host ohne RT-Patch mit Windows XP im Gast - beide Threads

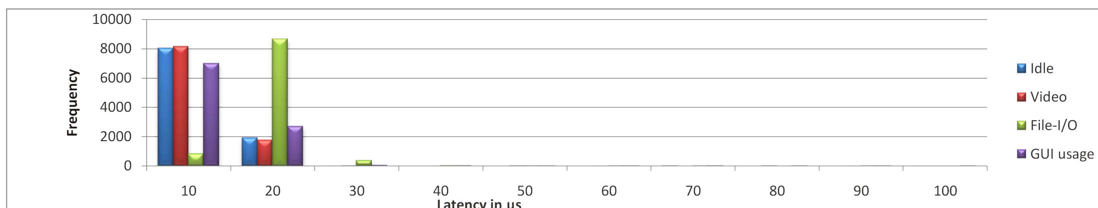


Abbildung 5.8: Latenzen von Cyclictest auf einem Host mit Windows XP im Gast, in Abhängigkeit vom Typ der im Gast erzeugten Last

- Der RT-Patch funktioniert wie erwartet, indem er die Durchschnittslatenz minimal erhöht und dafür die maximale Latenz stark reduziert. Dies funktioniert, wie man es nach Vergleich 1 auch erwarten würde, nicht im Falle von VMWare.

Daraus habe ich den Schluss gezogen, dass die weiteren Messungen mit RT-Patch durchgeführt werden sollten.

5.2.8 Vergleich 3 - Verschiedene Arten von Last im Gast

Beim dritten Vergleich werden die Auswirkungen von verschiedenen Arten von Lasten innerhalb der virtuellen Maschinen auf den Benchmark im Host untersucht. Bei KVM und QEmu gab es keine nennenswerten Auswirkungen, so dass ich hier auf Diagramme verzichte. Bei VMWare hingegen gab es eine klare Auswirkung, die man in Abbildung 5.8 sieht. Durch den Festplattenzugriff haben sich die Werte deutlich um mindestens 10μs verschoben. Auch die 99,5% Quantile liegen weit höher - bei der GUI-Benutzung bei 500μs und beim Festplattenzugriff sogar bei 1000μs.

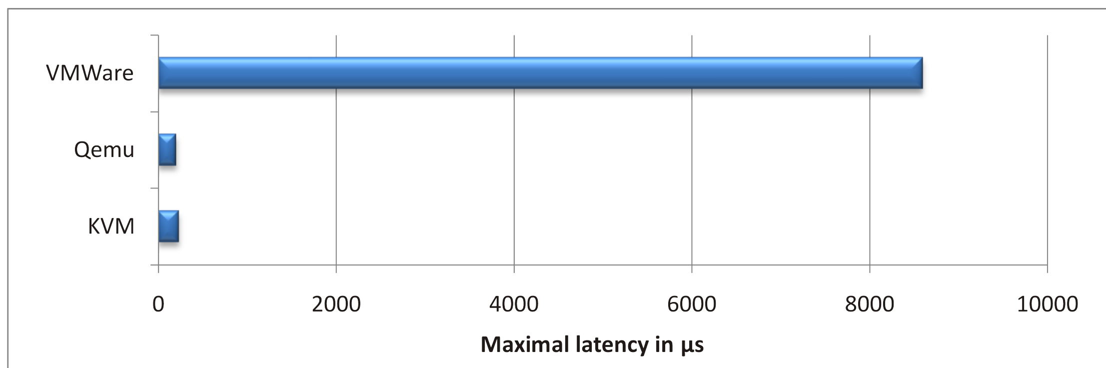


Abbildung 5.9: Latenzen von Cyclictest auf einem Host mit Windows XP im Gast, in Abhängigkeit vom Typ der im Gast erzeugten Last

5.2.9 Langzeitmessung

Während bei allen anderen Messungen lediglich 10 000 Durchläufe gemacht wurden, wurden für diese Messung 200 000 Durchläufe durchgeführt. Damit soll ein Eindruck von der höchsten zu erwartenden Latenz vermittelt werden. Das Ergebnis ist in Abbildung 5.9 zu sehen. Die maximalen Latenzen bei KVM (231μs) und QEmu (203μs) haben sich gegenüber der kurzen Messung fast nicht verändert, während die hier ermittelte maximale Latenz für VMWare 8595μs beträgt.

5.2.10 Fazit

Diese Resultate führen zu dem Schluss, dass VMWare in diesem Aufbau nur in Frage kommt, wenn häufiger auftretende Latenzen von bis zu 1000μs kein Problem darstellen. Schließlich kommen längere Festplattenzugriffe bei Gästen mit Windows XP, z.B. im Zusammenhang mit virtuellem Arbeitsspeicher, häufig und unerwartet vor. Auch müssen maximale Latenzen von bis zu 10ms akzeptabel sein. Die beiden anderen Virtualisierungslösungen können hingegen praktisch ohne Einschränkungen eingesetzt werden.

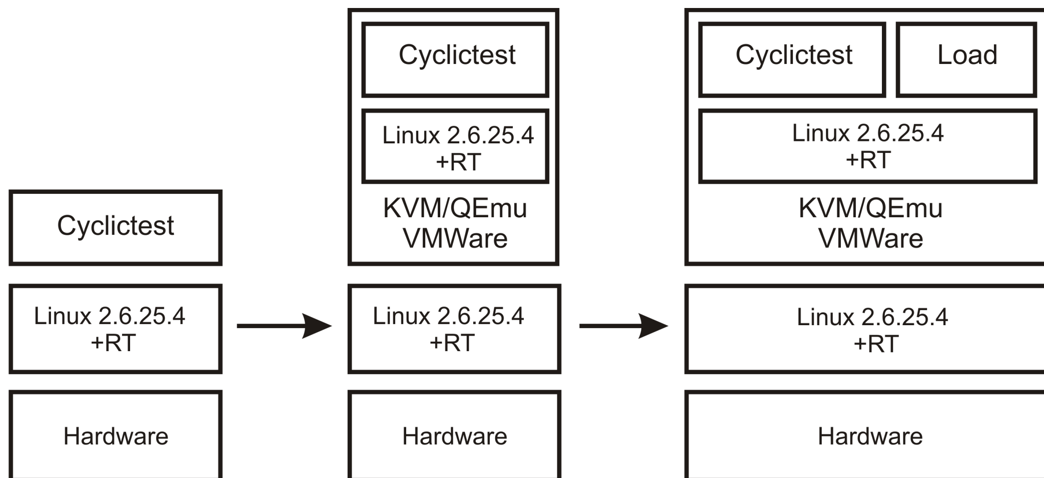


Abbildung 5.10: Aufbau von Versuch 2

5.3 Versuch 2 - Echtzeitanwendung im Gast

5.3.1 Beschreibung

Beim zweiten Versuch wird, ähnlich wie beim ersten, von einem Linux-System ausgegangen, auf dem ein Echtzeittask läuft. Nun soll durch den Versuch untersucht werden, ob man diesen auch in die virtuelle Maschine verlegen kann.

Dies soll im Rahmen von zwei Vergleichen geschehen. Dazu werden Latenzen innerhalb der virtuellen Maschine gemessen. Beim **ersten Vergleich** werden diese Latenzen mit solchen aus dem ersten Versuch verglichen. Anschließend werden innerhalb der virtuellen Maschine verschiedene Lasten erzeugt. Die dabei anfallenden Messwerte werden verwendet, um im Rahmen des **zweiten Vergleichs** die Auswirkungen dieser Lasten auf die Echtzeitfähigkeit zu überprüfen. Die beiden Vergleiche werden in Abbildung 5.10 dargestellt.

Am Ende wird wieder je eine Langzeitmessung pro Virtualissierungslösung durchgeführt. Diese findet ohne Last in der virtuellen Maschine statt.

5.3.2 Vorbereitungen

Die Vorbereitungen auf dem Gast waren wieder relativ einfach. Zunächst musste darauf Ubuntu 7.10 installiert werden, was auf VMWare ohne Probleme klappte. KVM hingegen kommt mit der gfxboot Erweiterung, die für graphische Bootmenüs zuständig ist, nicht zurecht. Dies ist ein bekanntes Problem, welches auf der offiziellen Homepage erwähnt wird [kw]. Durch das Drücken der Taste Shift beim Systemstart, wird gfxboot deaktiviert und das Betriebssystem kann nun normal installiert werden. Am Ende muss das grafische Bootmenü für das installierte System deaktiviert werden.

Auf dem Host wurden Skripte zum Starten der virtuellen Maschinen eingerichtet, die denen aus Versuch 1 sehr ähnlich sind. Der größte Unterschied zu diesen besteht darin, dass die

virtuellen Maschinen selbst nun Echtzeitpriorität benötigen. Die virtuellen Maschinen haben in diesem Versuch die Priorität 40 (Begründung weiter unten im Abschnitt 5.4.3). Für die Festlegung der Priorität wird, genau so wie für die Prozessorkernzuweisung in Versuch 1, das Programm `schedtool` verwendet. Der folgende Aufruf startet beispielsweise VMWare mit Priorität 40:

```
schedtool -F -p 40 -e vmware [...]
```

5.3.3 Erwartungen

Schon vor der Durchführung der ersten der Messung war klar, dass bei diesem Versuch Latenzen unter 20μs, wie sie beim ersten Versuch den Regelfall darstellten, hier nicht zu erwarten sind. Stattdessen habe ich mit Latenzen, die um ein vielfaches höher liegen, gerechnet. Für Echtzeitanwendungen wichtig sind aber nicht nur die durchschnittlichen Latenzen sondern auch die Abweichungen nach oben. Hier habe ich erwartet, dass es eine Konstellation gibt, wo fast alle Latenzen unter einer vertretbaren Grenze liegen.

5.3.4 Probleme bei der Durchführung

Instabilität bei KVM

Im Rahmen der Arbeit habe ich sehr lange mit KVM gearbeitet. Während dieser Zeit ist es insgesamt zweimal aus unbekannten Gründen abgestürzt (bekannte Bugs mit existierendem Workaround zählen nicht). Bei einem der Abstürze ist ein unbeteiligter Prozess ebenfalls abgestürzt. Dies wird an dieser Stelle erwähnt, da zu dem Zeitpunkt Linux als Gast-Betriebssystem zum Einsatz kam. Während dies eine sehr gute Bilanz für ein Desktopprogramm darstellen würde, mindert dies die Eignung für eingebettete Systeme. Bei QEmu und VMWare habe ich während der Versuche keinen nicht erklärbaren Absturz gehabt.

Dies alles hat natürlich keine Beweiskraft. Insbesondere gibt es im Internet Berichte über unerklärbare Abstürze bei allen drei Virtualisierungsprodukten. Ein Systementwickler, der Virtualisierung einsetzen will, kann zum heutigen Zeitpunkt eine sehr hohe, aber keine hundertprozentige Zuverlässigkeit erwarten.

Freeze von KVM/QEmu beim Verschieben von Fenstern

Während der Messungen für den zweiten Versuch ist mir aufgefallen, dass die Ausführung des Gastbetriebssystems bei KVM/QEmu unterbrochen wird, während das Fenster mit der virtuellen Maschine verschoben wird. Das gleiche passiert, wenn ein anderes Fenster über dem Fester mit der virtuellen Maschine bewegt wird. Dabei entstehen Latenzen, die genau die Zeitspanne angeben, in der man das Fenster ununterbrochen bewegt hat. Bei den weiteren Messungen habe ich das Problem umgangen, indem ich auf das Verschieben von Fenstern außerhalb der virtuellen Maschine während der Messung verzichtet habe.

Anschließend habe ich einen Workaround gesucht, denn man kann im Feld natürlich niemandem das Verschieben von Fenstern verbieten. Der Workaround besteht darin, dass man

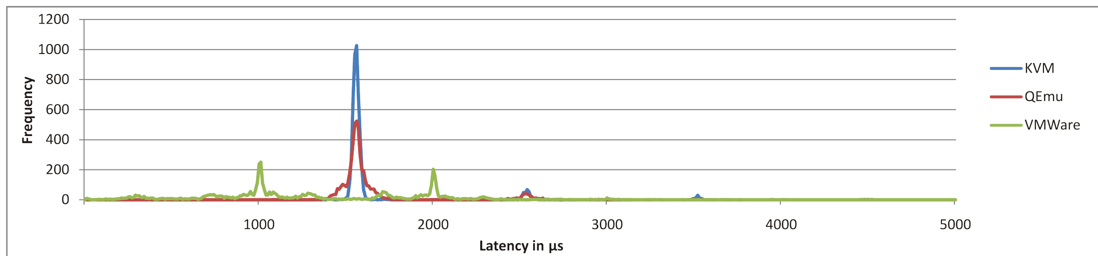


Abbildung 5.11: Latenzen von Cyclictest im Gast ohne Last

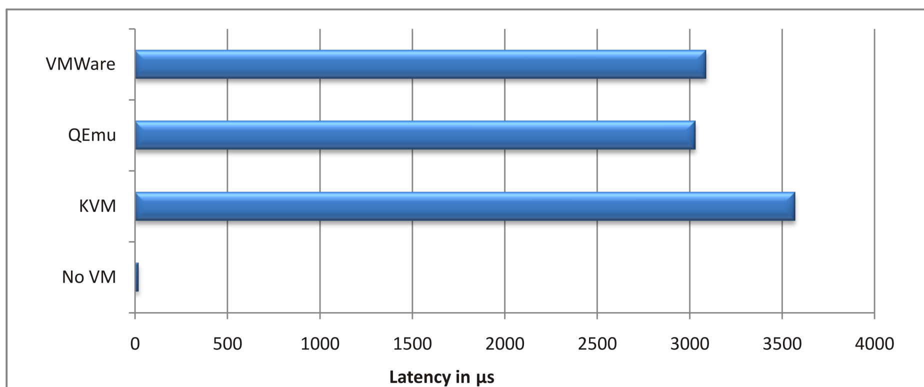


Abbildung 5.12: 99,5%-Quantil im Gast ohne Last

die virtuelle Maschine nach der Installation ohne Grafikkarte ausführt. Dazu wird KVM mit der Kommandozeilenoption `-nographic` ausgeführt. Das fehlen der Grafikkarte ist für die meisten Echtzeitanwendungen kein Problem, da sie nicht über eine grafische Oberfläche verfügen. Um trotzdem auf das Gastsystem zugreifen zu können, habe ich auf diesem einen SSH-Server installiert. Falls Zugriffe übers Netzwerk unerwünscht sind, kann man stattdessen auch über ein emuliertes serielles Terminal auf den Gast zugreifen.

Mit dem so präparierten Gast habe ich nun die Messung wiederholt und mich davon überzeugt, dass das Problem mit dem Verschieben von Fenstern nicht mehr auftauchte.

5.3.5 Vergleich 1 - Latenzen auf dem Host und in der virtuellen Maschine

Der erste Vergleich ist sehr spannend, denn sollte hier ein negatives Ergebnis herauskommen, wären alle weiteren Messungen überflüssig. Glücklicherweise war es nicht der Fall, wie man Abbildungen 5.11 und 5.12 entnehmen kann.

Wie erwartet, sind die gemessenen Latenzen nun deutlich gestreut und im Schnitt weit höher als auf dem Host. Trotzdem ist eine Eignung für Echtzeitanwendungen, die mit Latenzen im Millisekundenbereich zurechtkommen, gegeben. Im Unterschied zum ersten Versuch ist hier das 99,5%-Quantil unabhängig von der Virtualisierungslösung bei ca. 3ms.

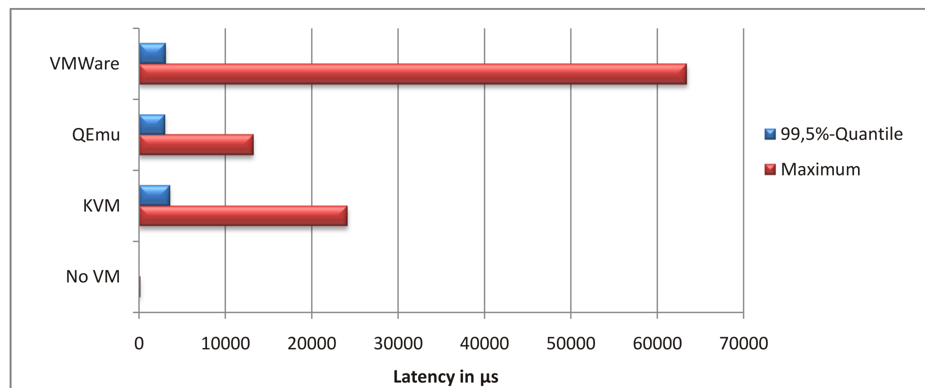


Abbildung 5.13: Maximale Latenz und 99,5%-Quantil im Gast ohne Last

5.3.6 Vergleich 2 - Verschiedene Arten von Last im Gast

Der zweite Vergleich ist deutlich weniger spannend, denn eigentlich sollte eine Last innerhalb derselben virtuellen Maschine keine Auswirkungen auf deren Echtzeitfähigkeit haben. Schließlich kennt der Scheduler in diesem Fall die Prioritäten. Das Ergebnis ist genau wie erwartet: Die Last führt nicht zu einer signifikanten Verschlechterung der durchschnittlichen Latenz oder des 99,5%-Quantils. Leider gibt es wieder eine negative Beobachtung im Zusammenhang mit VMWare und Festplattenzugriffen. Es gab in den 5000 Messwerten gleich 3, die über der Messgrenze von 100ms lagen. Dies engt den möglichen Einsatzbereich von VMWare in Szenarien 2-4 stark ein.

5.3.7 Langzeitmessung

Bei der Langzeitmessung wurde wieder ohne Last gemessen. Die Werte zusammen mit den 99,5% Quantilen aus Vergleich 1 sind in Abbildung 5.13 gemeinsam zu sehen. Wie man sieht müssen Echtzeitanwendungen, die in KVM oder QEmu ausgeführt werden, tolerant gegenüber Latenzen im zweistelligen Millisekundenbereich sein. Der Wert für VMWare ist angesichts der Erkentnis aus dem letzten Vergleich nur bedingt von Interesse.

5.3.8 Fazit

Das Gesamtergebnis kann man auf jeden Fall als positiv für KVM und QEmu bezeichnen. In diesen beiden virtuellen Maschinen können Anwendungen mit entsprechend niedrigen Echtzeitanforderungen ohne weiteres eingesetzt werden. Dabei muss man beachten, dass KVM und QEmu zwar ähnliche Ergebnisse bezüglich Latenzen liefern, aber KVM eine deutlich bessere Performance liefert. Aus diesem Grund wird KVM, wenn eine gewisse Prozessorleistung benötigt wird, den Vorzug erhalten.

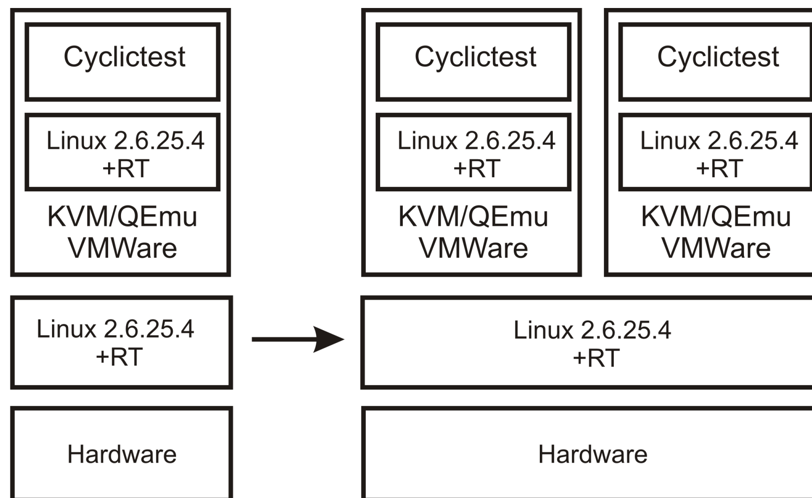


Abbildung 5.14: Aufbau von Versuch 3

5.4 Versuch 3 - Zwei virtuelle Maschinen jeweils mit Echtzeitanwendung

5.4.1 Beschreibung

Der dritte Versuch ähnelt dem zweiten, nur dass hier zwei virtuelle Maschinen gleichzeitig zum Einsatz kommen. Entsprechend werden die Ergebnisse mit denen aus dem letzten Verusch verglichen. Der Aufbau ist in Abbildung 5.14 zu sehen.

5.4.2 Vorbereitung

Die Vorbereitung bestand diesmal nur darin, die virtuellen Maschinen aus Versuch 2 jeweils zu kopieren. Hierfür gibt es in VMWare eine extra Funktion. Bei KVM genügt es alle Dateien zu kopieren. Anschließend musste ich noch die Portnummern in den Benchmarks auf der zweiten Maschine anpassen, damit sie einen anderen Port zum Messen verwendet als die erste. Bei QEmu und KVM mussten die Startskripte so angepasst werden, dass die erste virtuelle Maschine auf dem ersten Prozessorkern ausgeführt wird, und die zweite virtuelle Maschine auf dem zweiten Kern.

5.4.3 Probleme bei der Durchführung

Zu hohe Priorität der virtuellen Maschinen

Bei der Vorbereitung zu Versuch 2 habe ich der virtuellen Maschine Priorität 40 vergeben. Ursprünglich habe ich jedoch Priorität 60 vergeben. Damit konnte ich die Messungen von Versuch 2 ohne Probleme durchführen, auch wenn ich da andere Ergebnisse hatte. Bei diesem

Versuch bin ich jedoch mit dieser Priorität auf sehr große Probleme gestoßen. Die Messwerte waren extrem hoch und das ganze System wurde instabil.

Die Begründung für dieses Problem besteht in der Tatsache, dass wichtige Systemprozesse unter Linux mit Priorität 50 laufen. Die Tatsache, dass ich den virtuellen Maschinen nun eine höhere Priorität zugewiesen habe, führte bei zwei gleichzeitig laufenden virtuellen Maschinen nun dazu, dass Systemprozesse aushungerten. Außerdem waren die Messergebnisse falsch, denn die Netzkomponenten auf dem Host hatte ja auch Priorität 50.

Die Lösung bestand offensichtlich darin, für die virtuellen Maschinen eine Priorität kleiner 50 zu wählen.

Zwei QEmu-VMs gleichzeitig

Da QEmu keine Vollvirtualisierungslösung ist, verbraucht es deutlich mehr Prozessorzeit auf dem Host als KVM/VMWare. Dies führte zu großen Problemen als ich auf zwei QEmu-VMs gleichzeitig gemessen habe. Als beide im Leerlauf waren konnte ich die Messung ohne Schwierigkeiten durchführen. Aber sobald ich auf beiden 90%-Last mit Hilfe von Timehog hinzugefügt habe, wurde das Gesamtsystem plötzlich unbedienbar. Das lag daran, dass die beiden VMs mit Echtzeitpriorität jeweils 100% der Prozessorzeit des ihnen zugewiesenen Kerns verbrauchten. Aus diesem Grund wurde der entsprechende Versuch mit QEmu nicht durchgeführt.

Das gleiche Problem hat man natürlich auch mit VMWare/KVM, wenn man dort einen Prozess startet, der unkontrolliert 100% der Prozessorzeit verbraucht. Das Problem ist hier allerdings nicht neu, da es genau so auftritt wenn man einen Echtzeitprozess mit 100% Rechenzeitverbrauch direkt auf dem Host startet. Bei der Verwendung von QEmu genügen hingegen weit weniger als 100% Auslastung um das Problem zu erzeugen.

Zwei VMs gleichzeitig

Beim Starten von zwei virtuellen Maschinen gleichzeitig hatte ich oft das Problem, dass sich dabei das gesamte System aufgehängt hat. Dieses Problem hatte ich sowohl bei VMWare als auch bei KVM. Das Problem habe ich umgangen, indem ich eine Pause von einer Minute eingehalten habe.

Hoher Netz-Jitter bei VMWare

Nach der Messung waren die Ergebnisse mit VMWare bei Last in keinster Weise mit denen aus Versuch 2 vergleichbar. Das lag daran, dass der Netz-Jitter der von VMWare emulierter Netzwerkkarte bei diesem Versuchsaufbau oft hohe Werte erreichte. Während der durchschnittliche Messwert im selben Bereich lag wie im zweiten Versuch, da er vom Jitter ja nicht betroffen ist, war die Varianz um das hundertfache höher. Somit sind die gemessenen Latenzen fast wertlos und werden nicht ausgewertet. Dieses Problem stellt insofern ein Ergebnis da, als das Echtzeitanwendungen, die auf Netzkommunikation angewiesen sind, auf diese Art und Weise sicher nicht virtualisiert werden können.

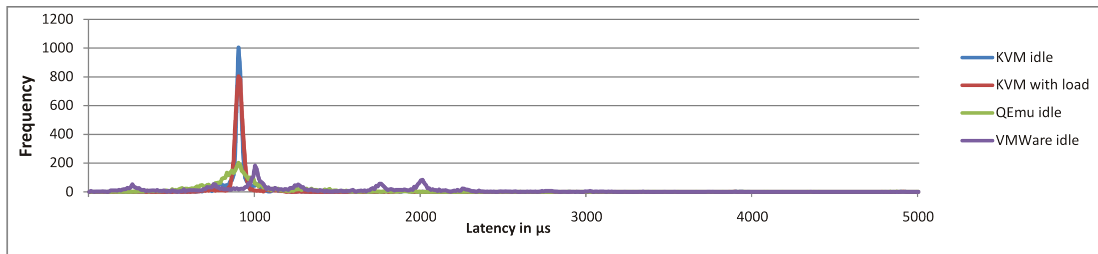


Abbildung 5.15: Latenzen im Gast bei zwei gleichen VMs

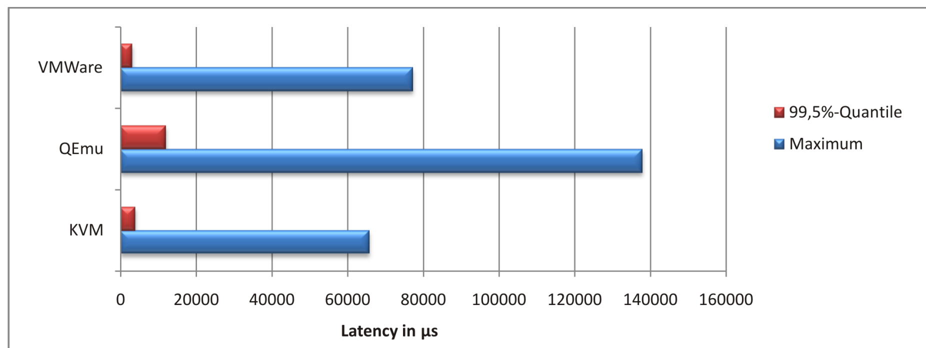


Abbildung 5.16: Maximum und 99,5%-Quantil im Gast bei zwei gleichen VMs

5.4.4 Ergebnisse

Die Ergebnisse der Messungen sind in den Abbildungen 5.15 und 5.16 dargestellt. Obwohl es natürlich jeweils zwei Ergebnismengen pro Messung gibt, da ja zwei virtuelle Maschinen gibt, habe ich nur eine im Diagramm dargestellt, da die beiden austauschbar sind. Während es bei der Latenzverteilung nichts Negatives zu bemerken ist, sind die Werte des 99,5%-Quantils etwas schlechter als in Versuch 2. Die maximale Latenz ist sogar sehr deutlich schlechter. Echtzeitanwendungen, die gemäß diesem Versuchsaufbau virtualisiert werden, müssen somit mit maximalen Latenzen in der Größenordnung von 100ms zurechtkommen.

5.4.5 Fazit

Die Ergebnisse dieses Versuchs lassen die Vermutung zu, dass dieser Aufbau für Anwendungen mit Echtzeitanforderungen nicht geeignet ist. Die Latenzen nehmen bei Last zu und die Maxima sind im Vergleich zum Durchschnitt hoch. Somit verhält sich das Gesamtsystem überhaupt nicht so, wie man es von einem echtzeitfähigen Aufbau erwarten würde.

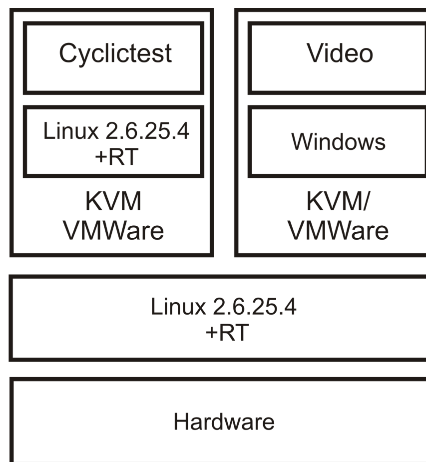


Abbildung 5.17: Aufbau von Versuch 4

5.5 Versuch 4 - Zwei virtuelle Maschinen je eine mit Echtzeitanwendung und grafischer Anwendung

5.5.1 Beschreibung

Wie beim letzten Versuch kommen auch hier zwei virtuelle Maschinen zur Anwendung. Im Unterschied dazu, läuft jedoch bei diesem Versuch auf einer der virtuellen Maschinen Windows, so dass nur eine virtuelle Maschine echtzeitfähig sein muss (Abbildung 5.17). Im Unterschied zu den anderen Versuchen, wurde hier auf eine Messung mit QEmu verzichtet, weil die sowieso GUI-Performance von QEmu für solche aufbauten im Normallfall nicht ausreichen würde, so dass man sowieso KVM benutzen würde.

5.5.2 Vorbereitung

Die Vorbereitung zu diesem Versuch war trivial, da die virtuellen Maschinen aus den vorherigen Versuchen benutzt werden konnten.

5.5.3 Ergebnisse

Die Ergebnisse sind bei diesem Aufbau noch schlechter als in Versuch 3. Während das Latenzhistogramm ähnlich aussieht, sind die maximale Latenz und das 99,5%-Quantil noch höher, wie man Abbildung 5.18 entnehmen kann.

5.5.4 Fazit

Da die Latenzen hier noch höher sind als in Versuch 3, ist dieser Aufbau ebenfalls in der Regel nicht für Anwendungen mit Echtzeitanforderungen geeignet.

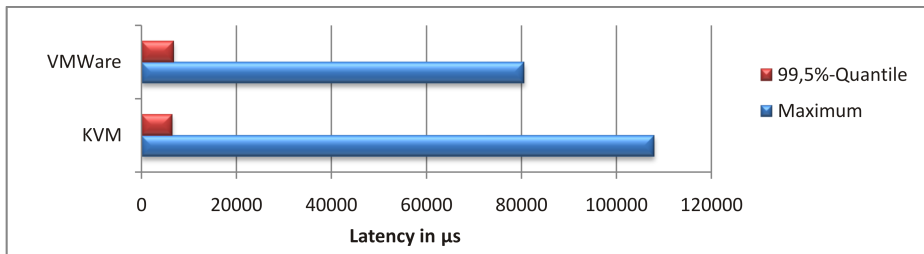


Abbildung 5.18: Maximum und 99,5%-Quantil im Gast bei einer VM mit Linux und einer mit Windows

5.6 Zusammenfassung

Im Laufe der Versuche wurden zahlreiche Messungen vorgenommen. Eine genaue Interpretation der Messwerte konnte mangels einer vorgegebenen Anwendung nicht stattfinden, dennoch konnten einige allgemeine Aussagen gemacht werden. Zusammenfassend lässt sich sagen, dass nur im ersten Versuch, der dem ersten Szenario entspricht, die Echtzeitfähigkeit erhalten bleibt. Im zweiten Szenario, dass im zweiten Versuch überprüft wurde, ist bereits mit größeren Einschränkungen zu rechnen. Bei den letzten beiden Szenarien kann von Echtzeitfähigkeit keine Rede mehr sein.

Trotz dieser Ergebnisse sind Anwendungen denkbar, die dennoch virtualisiert werden können. Nur müssen deren Anforderungen bezüglich Latenz entsprechend gering sein. Dies kann jeder Entwickler, dem konkrete Anforderungen vorliegen, mit Hilfe meiner Messwerte überschlagen. Sollten die Darstellungen in diesem Kapitel hierfür nicht ausrechen, so lassen sich alle Messwerte auf der beigelegten CD finden.

6 Demonstrator

Der praktische Teil der Arbeit wird durch die Einrichtung eines Notebooks zu Demonstrationszwecken abgerundet. Dieser soll in Zukunft Demonstrator genannt werden. Der Demonstrator soll auf eine interaktive Art und Weise die Möglichkeiten der Virtualisierung im Zusammenhang mit Echtzeitanforderungen darstellen.

6.1 Motivation

In den Versuchen im letzten Kapitel wurden verschiedene Messergebnisse ermittelt und interpretiert. Dabei war eine Einordnung der Ergebnisse oft schwierig, da die Anforderungen stark von der angestrebten Anwendung abhängen. Dabei sollte eine Präsentation von verschiedenen Diagrammen mit den genauen Versuchsergebnissen dem Leser eine eigene Einordnung ermöglichen. Eine bessere Gelegenheit, sich selbst eine Meinung über die Echtzeitfähigkeit von verschiedenen Aufbauten bilden zu können, wird der Demonstrator geben. Er bietet dem Nutzer die Möglichkeit die Messergebnisse mit eigenen Augen zu sehen. Zusätzlich entsteht die Möglichkeit „rumzuspielen“, um verschiedene Anwendungsszenarien nachzubilden und dabei die Auswirkungen sofort zu sehen.

6.2 Konzept

Genaue Anforderungen für den Demonstrator seitens des Auftraggebers lagen nicht vor, so dass ich bei dessen Entwicklung relativ freie Hand hatte. Fest stand jedoch der angestrebte Nutzerkreis. Der Demonstrator sollte einem technisch versierten Benutzer die Entscheidung erleichtern, ob Virtualisierung in seinem konkreten Anwendungsfall für den Systemaufbau in Frage kommt.

Als Grobkonzept für den Demonstrator habe ich mich an die Versuchsszenarien aus dem Abschnitt gehalten. Da diese den Ausgangspunkt für die Versuche bildeten, werden sich diese im Demonstrator wiedererkennen lassen. Dies erlaubt es die in Kapitel 5 gewonnenen Versuchsergebnisse mit denen aus dem Demonstrator zu vergleichen.

Die vier Versuchsszenarien und die drei verwendeten Virtualisierungsprodukte ergeben zwölf Kombinationsmöglichkeiten. Von diesen haben sich einige im letzten Kapitel als untauglich erwiesen. Eine Kombinationsmöglichkeit wurde noch nicht mal überprüft. Da der Demonstrator es dem Nutzer erlauben soll, sich selbst eine Meinung zu bilden, sind alle zwölf Kombinationsmöglichkeit im Demonstrator enthalten. Diese bilden die zwölf Versuche die der Benutzer des Demonstrators direkt ausführen kann.

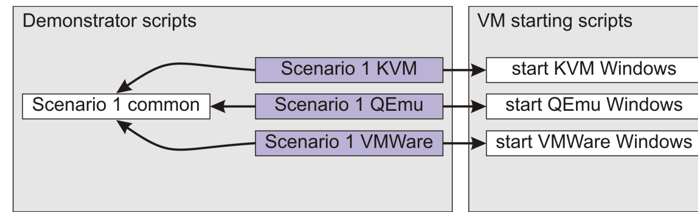


Abbildung 6.1: Aufbau der Demonstrationsskripten für das erste Szenario

Die Aufbauten werden etwas anders sein als in den Versuchen. Der Benutzer wird zusätzliche Möglichkeiten erhalten was zu beeinflussen. Außerdem sollen ja auch bestimmte Konzepte illustriert werden.

6.3 Implementierung

Die einzelnen Versuche werden durch bash-Skripte abgebildet. Links zum Starten dieser Skripte sind auf dem Desktop abgelegt. Zu jedem Szenario gibt es zusätzlich eine HTML-Seite, die als Beschreibung und Bedienungsanleitung für die jeweiligen Versuche fungiert.

6.3.1 Programme

Bei den Versuchen werden dieselben Programme wie bei den Messungen benutzt. Für die interaktive Darstellung der Latenzen wird genauso Cyclictest benutzt. Dieses eignet sich dafür, denn es zeigt die minimale, maximale, durchschnittliche und aktuelle Latenz laufend an. In verschiedenen Aufbauten soll der Benutzer die Möglichkeit erhalten die Prozessorlast zu variieren. Dafür wird das Programm Timehog eingesetzt. Es wurde ja mit dieser Anwendung im Kopf entwickelt. Damit der Benutzer die Konzepte zur Rechenzeitverteilung sieht, ist es hilfreich diese während der Versuche anzuzeigen. Hierfür wird das unter Linux hierfür verbreitete Programm `top` eingesetzt.

6.3.2 Skripte

Die Skripte sind sehr einfach aufgebaut und kurz. Im Folgenden wird die Implementierung vom ersten Szenario, die auch in Abbildung 6.1 dargestellt ist, kurz skizziert:

1. Die Startskripte zum starten der virtuellen Maschinen aus dem letzten Kapitel können ohne Änderung weiterverwendet werden. Listing 5.1 auf Seite 43 stellt ein Beispiel da.
2. Ein Skript startet alle Teile, die nicht für eine virtuelle Maschine spezifisch sind. Im ersten Kapitel sind es die bereits beschriebenen Programme `top`, Cyclictest und Timehog. Dabei wird für jedes der Programme ein neues Terminal geöffnet und mit Hilfe des `--geometry` Parameters an einer bestimmten Stelle auf dem Desktop positioniert. Die genaue Implementierung sieht man in Listing 6.1.

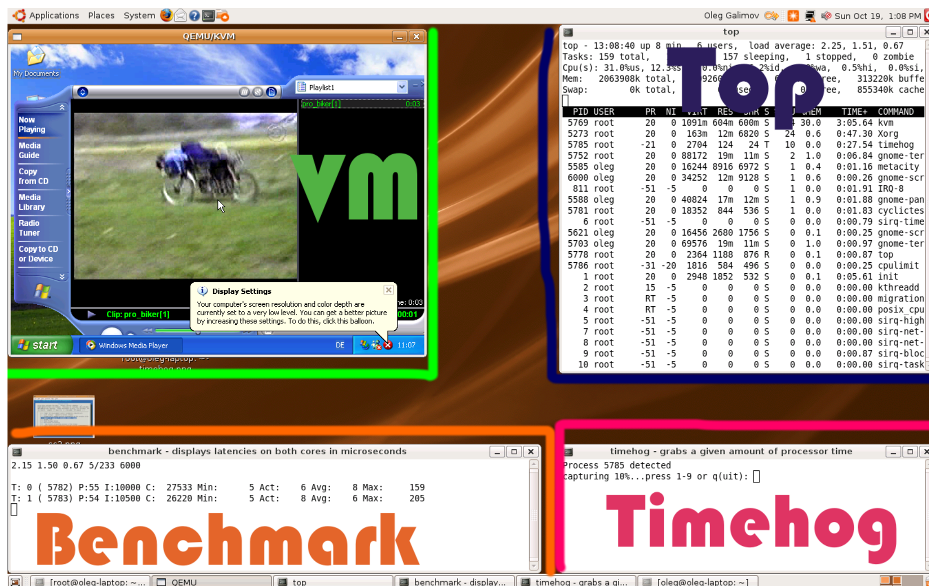


Abbildung 6.2: Screenshot vom ersten Szenario

3. Nun wird für jedes Virtualisierungsprodukt ein eigenes Skript erstellt, welches die virtuelle Maschine und das Skript aus Nummer 2 startet.
4. Für jedes Skript wird eine Verknüpfung auf dem Desktop angelegt. Mithilfe des Programms **gksu** werden dem Skript root-Rechte gegeben, die vom Timehog vorausgesetzt werden.

Listing 6.1: Skript zum starten von sonstigen Komponenten im ersten Szenario

```
gnome-terminal -e ~/docyclic-long --hide-menubar
     --geometry=100x10-900+900 --title="benchmark
␣␣␣␣␣-␣displays␣latencies␣on␣both␣cores␣in␣microseconds"&
gnome-terminal -e ~/timehog/timehog --hide-menubar
     --geometry=70x10-0+900 --title=
     "timehog␣-␣grabs␣a␣given␣amount␣of␣processor␣time"&
gnome-terminal -e top --hide-menubar
    --geometry=70x30-0+10 --title="top"&
```

6.4 Beschreibung

In diesem Abschnitt werden die entstehenden Aufbauten nun aus Benutzersicht dargestellt.

6.4.1 Szenario 1

Ein Screenshot vom ersten Szenario sieht man in Abbildung 6.2. Der Benutzer sieht hier eine virtuelle Maschine, auf dem Windows XP als Betriebssystem installiert ist. Auf dem

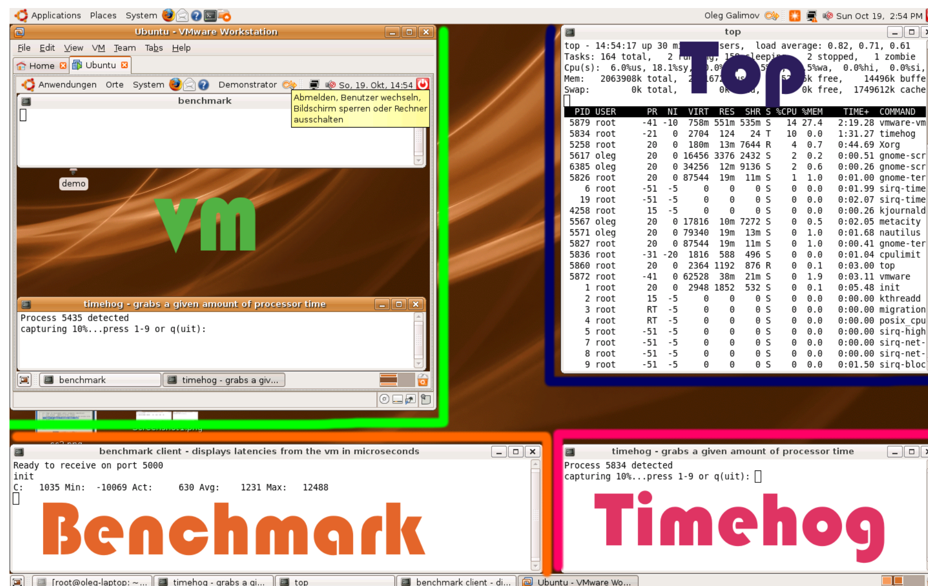


Abbildung 6.3: Screenshot vom zweiten Szenario

Windows XP läuft der Windows Media Player in dem ein Video abgespielt wird. Unten sieht der Benutzer ein Benchmark (Cyclictest), welches die Latenzen auf dem Hostsystem misst. Rechts oben befindet sich `top` welches die CPU-Zuteilung an einzelne Prozesse anzeigt. Rechts unten wird Timehog gestartet.

Der Benutzer hat nun zwei vorgesehene Interaktionsmöglichkeiten, deren Auswirkungen auf die Latenzen er überprüfen kann:

1. Die Last im Timehog kann verändert werden. Dabei wird das Video in der virtuellen Maschine eventuell langsamer.
2. Der Benutzer kann Programme im Gast und auf dem Host starten und benutzen.

6.4.2 Szenario 2

Szenario 2, welches in Abbildung 6.3 zu sehen ist, ist vom Aufbau her dem ersten Szenario sehr ähnlich. Die Fenster mit `top` und Timehog bleiben unverändert. Das Benchmarkfenster zeigt jedoch nicht mehr Cyclictest sondern einen Client, der die Versuchsergebnisse aus der virtuellen Maschine anzeigt. Die virtuelle Maschine selbst ist nicht mehr mit Windows XP sondern mit Linux eingerichtet. Nach dem Start der virtuellen Maschine muss der Benutzer die Messung manuell starten, indem er das Symbol „demo“ auf dem Desktop der VM anklickt. Dies ist notwendig, da das Demonstrationsskript in der virtuellen Maschine root-Rechte benötigt um Timehog auszuführen. Hierfür wird wieder `gksu` benutzt. Nach dem ausführen, sieht der Nutzer nun zwei Fenster in der VM. Eines davon zeigt Cyclictest, dessen Ergebnisse allerdings außerhalb der virtuellen Maschine angezeigt werden. Das andere Fenster zeigt Timehog.

Gegenüber dem ersten Szenario ist hier eine weitere Interaktionsmöglichkeit hinzugekommen.

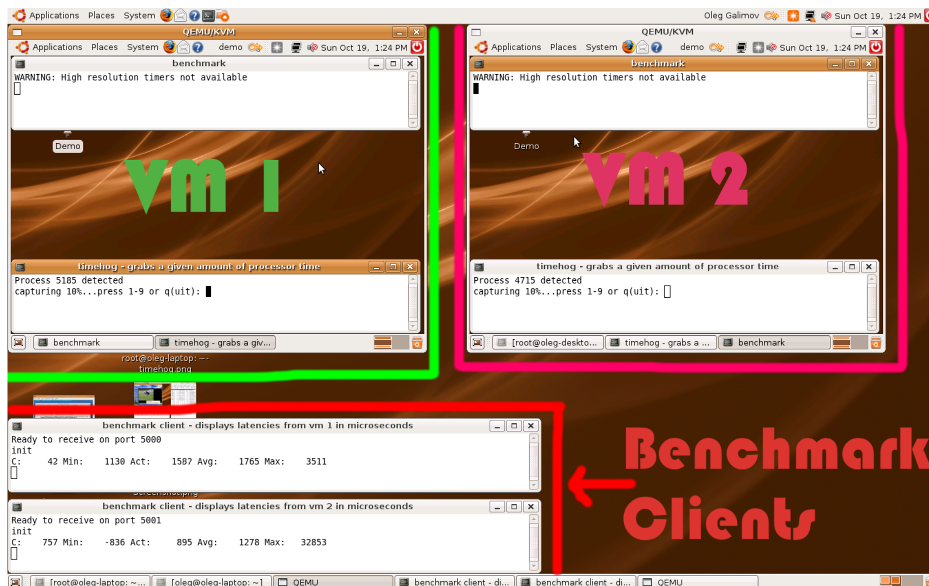


Abbildung 6.4: Screenshot vom dritten Szenario

Der Benutzer kann nun die Prozessorlast in der virtuellen Maschine, mit Hilfe des darin laufenden Timehog, beeinflussen.

6.4.3 Szenario 3

Der Aufbau des dritten Szenarios, in Abbildung 6.4 zu sehen, unterscheidet sich etwas vom Aufbau der anderen. Die Fenster mit `top` und Timehog sind nicht zu sehen, da es in diesem Szenario möglichst wenig Prozessorlast auf dem Host geben soll. Der Anwender sieht lediglich zwei virtuelle Maschinen, die denen aus Szenario 2 gleichen. Zusätzlich sieht er zwei Clients, die die Latenzen aus den beiden virtuellen Maschinen anzeigen.

Der Nutzer kann nun, mit Hilfe der Timehog-Programme die in den VMs laufen, die Prozessorlast variieren und deren Auswirkung auf die Latenzen beobachten.

6.4.4 Szenario 4

In Abbildung 6.5 ist ein Screenshot des vierten Szenarios zu sehen. Hier sind wieder zwei virtuelle Maschinen zu sehen. Die eine entspricht der aus dem ersten Szenario, die andere der aus den Szenarios 2 und 3. Bei KVM und QEmu laufen die beiden virtuellen Maschinen auf demselben Kern. Bei VMWare ist eine solche Festlegung nicht möglich. Zusätzlich laufen auf dem Host der Client für den Benchmark in der virtuellen Maschine sowie `top`.

Das Szenario demonstriert eine funktionierende Lastenverteilung zwischen den virtuellen Maschinen. Diese Problematik wurde in Abschnitt 1.2.2 erläutert. Diese Beobachtung lässt sich leider nur mit KVM und QEmu machen, da bei VMWare die beiden virtuellen Maschinen auf verschiedenen Kernen laufen. Verändert der Benutzer nun die Prozessorauslastung in der virtuellen Maschine mit Linux, so beginnt das Video in der virtuellen Maschine mit

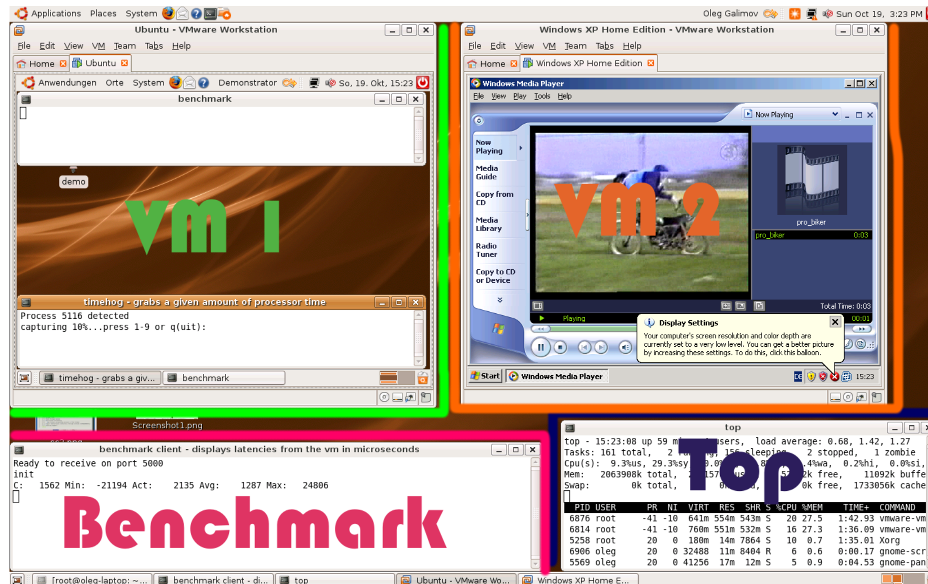


Abbildung 6.5: Screenshot vom vierten Szenario

Windows zu ruckeln. Dies kann funktionieren, weil die virtuelle Maschine mit Linux eine höhere Priorität als die andere besitzt. Somit bekommt sie soviel CPU-Zeit wie angefordert. Die VM mit Windows bekommt dann den Rest. Dies kann man auch im `top` beobachten.

7 Zusammenfassung und Ausblick

7.1 Zusammenfassung

Die Basis für die Arbeit bildete die zentrale Frage aus dem ersten Kapitel:

Ist Virtualisierung in eingebetteten Systemen praxistauglich?

Aufgrund der Tatsache, dass nicht alle Aspekte dieser Frage untersucht wurden und die Versuche auch nur auf einer Plattform durchgeführt wurden, kann diese Frage im Rahmen dieser Arbeit natürlich nicht gänzlich beantwortet werden. Trotzdem wurden durch die Versuche in Kapitel 5 viele Erkenntnisse gewonnen, die bei der Beantwortung der Frage im Bezug auf die untersuchte Intel-VT Plattform helfen. Eine weitere Einschränkung entstand durch die Auswahl der virtuellen Maschinen.

Wie am Anfang der Arbeit festgestellt wurde, haben Anwendungen in eingebetteten Systemen oft Echtzeitanforderungen. Durch die Versuche wurde nun Messdaten gewonnen, mit deren Hilfe überprüft werden kann, ob die Echtzeitanforderungen von konkreten Anwendungen in den getesteten Szenarien erfüllt werden.

Eine generelle Aussage über die Echtzeitfähigkeit eines Systems kann hingegen nur schwer getroffen werden, da die Anforderungen der möglichen Anwendungen sehr unterschiedlich sind. Dennoch können folgende Aussagen im Bezug auf Echtzeitfähigkeit gemacht werden:

- Wenn auf einem System eine Echtzeitanwendung läuft, dann wird sie durch eine zusätzlich laufende virtuelle Maschine mit KVM oder QEmu nicht gestört. Wird hingegen VMWare eingesetzt, so muss die Anwendung mit zusätzlichen Latenzen im Bereich von 10ms auskommen.

- Echtzeitanwendungen innerhalb einer virtuellen Maschine erwarten um den Faktor 1000 höhere Latenzen. Trotzdem ist bei KVM und QEmu eine gewisse Echtzeitfähigkeit gegeben.

- Werden mehrere virtuelle Maschinen auf einem System ausgeführt, so ist keinerlei Echtzeitfähigkeit innerhalb der virtuellen Maschinen gegeben. Sowohl die durchschnittlichen Latenzen als auch die Varianzen sind sehr hoch.

Aus diesen Aussagen lässt sich der Schluss ziehen, dass Virtualisierung von Echtzeitanwendungen mit KVM/QEmu/VMware auf der Intel-VT Plattform zur Zeit im Allgemeinen nicht möglich ist.

7.2 Ausblick

Bei der Beantwortung der Frage nach der Praxistauglichkeit der Virtualisierung in eingebetteten Systemen gibt es zwei verschiedene Dimensionen zu beachten. Zum einen stellt sich die Frage welche Plattformen untersucht werden. Dabei hat man sowohl bei der Hardware, als auch bei den Betriebssystemen und virtuellen Maschinen eine große Auswahl. Zum anderen ergeben verschiedene Aspekte zusammen die Praxistauglichkeit.

Im Rahmen dieser Arbeit wurde nun der Aspekt Echtzeitfähigkeit auf einer bestimmten Plattform überprüft. Daraus ergeben sich nun direkt weitere Themen, die untersucht werden können:

- Es könnten ähnliche Untersuchungen mit anderen Hardware-Plattformen (z.B. ARM, AMD-V), auf anderen Betriebssystemen (z.B. RTAI, Xenomai) oder mit anderen Virtualisierungsprodukten durchgeführt werden.
- Andere Aspekte wie Sicherheit, Stabilität (man denke an die Abstürze), Kompatibilität usw. könnten detailiert untersucht werden.

Nichtdestotrotz wurde das Ziel dieser Diplomarbeit, einen ersten Eindruck von den Möglichkeiten und Problemen der Virtualisierung von Echtzeitanwendungen zu geben, erreicht.

A Inhalt der CD

Auf der beigelegten CD befinden sich folgende Dateien:

- Im Hauptverzeichnis befindet sich die Datei gali08.pdf - es ist diese Arbeit als PDF
- Im Verzeichnis logs finden sich die Messergebnisse
- Im Verzeichnis demonstrator finden sich die Demonstratorbestandteile:
  - Im Unterverzeichnis scripts finden sich die Skripten. Um den Demonstrator nachzubauen, müssen die Pfade darin angepasst werden.
  - Im Unterverzeichnis bin befinden sich ausführbare Versionen der benöätigten Programme
  - Im Unterverzeichnis doc befindet sich die englische Dokumentation, die der Benutzer des Demonstrator sieht.
- Im Verzeichnis kernel befinden sich die benutzten Kernel als Debian-Pakete
- Im Verzeichnis src befinden sich die C/C++ Sourcen für die drei verwendeten Programme: Cyclictest, Timehog, Extbench
- Im Verzeichnis text befindet sich der Latex-Sourcecode für diese Arbeit

Abbildungsverzeichnis

1.1 Beispiel für eine Anwendung mit Echtzeitanforderung 4
1.2 Illustration eines Prozess-Schedulers mit FIFO-Warteschlangen 5
1.3 Beziehungen zwischen Warteschlangen bei der Verwendung von mehreren virtuellen Maschinen 6

2.1 Schichtenprinzip 9
2.2 Schichtenprinzip mit Java 10
2.3 Schichtenprinzip mit virtuellen Maschinen 11
2.4 Mit Hilfe von einer virtuellen Maschine können Trojaner und Viren daran gehindert werden sensible Daten auszuspionieren 12
2.5 Mögliche Anwendung des Konsolidierungsszenarios 14
2.6 Konsolidieren einer GUI und einer Echtzeitanwendung 15
2.7 Konsolidieren von mehreren Echtzeitanwendungen 15
2.8 Konsolidieren von mehreren Echtzeitanwendungen 16

3.1 Paravirtualisierung mit einem Mikrokernel 21

4.1 Versuchsablauf 25
4.2 Beispielhistogramm 26
4.3 Deaktivieren von virtuellem Speicher 32
4.4 Deaktivieren von Power Management 32
4.5 Ausgabe von Cyclictest 34
4.6 Oberfläche von timehog 38

5.1 Aufbau von Versuch 1 42
5.2 Auswirkungen von `idle=poll` 45
5.3 Latenzen von Cyclictest auf dem Host mit Windows XP im Gast - Thread 0 . 45
5.4 Latenzen von Cyclictest auf dem Host mit Windows XP im Gast - Thread 1 . 45
5.5 Maximale Latenzen von Cyclictest auf dem Host mit Windows XP im Gast - beide Threads 46
5.6 Latenzen von Cyclictest auf einem Host ohne RT-Patch mit Windows XP im Gast - Thread 0 46
5.7 Maximale Latenzen von Cyclictest auf einem Host ohne RT-Patch mit Windows XP im Gast - beide Threads 47
5.8 Latenzen von Cyclictest auf einem Host mit Windows XP im Gast, in Abhängigkeit vom Typ der im Gast erzeugten Last 47
5.9 Latenzen von Cyclictest auf einem Host mit Windows XP im Gast, in Abhängigkeit vom Typ der im Gast erzeugten Last 48
5.10 Aufbau von Versuch 2 49
5.11 Latenzen von Cyclictest im Gast ohne Last 51

5.12 99,5%-Quantil im Gast ohne Last 51
5.13 Maximale Latenz und 99,5%-Quantil im Gast ohne Last 52
5.14 Aufbau von Versuch 3 53
5.15 Latenzen im Gast bei zwei gleichen VMs 55
5.16 Maximum und 99,5%-Quantil im Gast bei zwei gleichen VMs 55
5.17 Aufbau von Versuch 4 56
5.18 Maximum und 99,5%-Quantil im Gast bei einer VM mit Linux und einer mit Windows 57

6.1 Aufbau der Demonstrationsskripten für das erste Szenario 60
6.2 Screenshot vom ersten Szenario 61
6.3 Screenshot vom zweiten Szenario 62
6.4 Screenshot vom dritten Szenario 63
6.5 Screenshot vom vierten Szenario 64

Literaturverzeichnis

[CB07] CEREIA, MARCO und IVAN CIBRARIO BERTOLOTTI: *Virtual machines for distributed real-time systems*, September 2007. http://dx.doi.org/10.1016/j.csi.2007.10.010.

[Cor07a] CORBET, JONATHAN: *CFS group scheduling*, Juli 2007. http://lwn.net/Articles/240474/; Abgerufen am 01. Oktober 2008.

[Cor07b] CORBET, JONATHAN: *Schedulers: the plot thickens*, April 2007. http://lwn.net/Articles/230574/; Abgerufen am 01. Oktober 2008.

[fal] FALKO: *How To Compile A Kernel - The Ubuntu Way.* http://www.howtoforge.com/kernel_compilation_ubuntu; Abgerufen am 17. August 2008.

[Feu07] FEUERER, PETER: *Benchmark and comparison of real-time solutions based on embedded Linux.* Diplomarbeit, Hochschule Ulm, Juli 2007. http://piie.net/diploma_thesis/diploma_thesis.pdf; Abgerufen am 15. Oktober 2008.

[FS] FU, LUOTAO und ROBERT SCHWEBEL: *RT PREEMPT HOWTO.* http://rt.wiki.kernel.org/index.php/RT_PREEMPT_HOWTO; Abgerufen am 17. August 2008.

[Hei07] HEISER, GERNOT: *Virtualization for Embedded Systems*, November 2007. http://www.ok-labs.com/_assets/download_library/OK_Virtualization_WP.pdf.

[Int04] INTEL: *Enhanced Intel® SpeedStep® Technology for the Intel® Pentium® M Processor*, März 2004. ftp://download.intel.com/design/network/papers/30117401.pdf; Abgerufen am 14. Oktober 2008.

[Int08] INTEL: *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3B*, Oktober 2008. http://www.intel.com/design/processor/manuals/253669.pdf; Abgerufen am 14. Oktober 2008.

[Kai08] KAISER, ROBERT: *Applying Virtualisation to Real-Time Embedded Systems*, Februar 2008.

[kw] WIKI KVM: *Guest_Support_Status.* http://kvm.qumranet.com/kvmwiki/Guest_Support_Status; Abgerufen am 17. August 2008.

[McK05] MCKENNEY, PAUL: *A realtime preemption overview*, August 2005. http://lwn.net/Articles/146861/; Abgerufen am 01. Oktober 2008.

[Net] NETRINO: *Embedded Systems Glossary.* http://www.netrino.com/Embedded-Systems/Glossary-E; Abgerufen am 17. August 2008.

[Tan08] TANENBAUM, ANDREW S.: *Modern Operating Systems*. Prentice Hall, 2008.

[VMW08] VMWARE: *Embedded Systems Glossary*, August 2008. `http://www.vmware.com/pdf/GuestOS_guide.pdf`; Abgerufen am 17. August 2008.

[Wik08] WIKIPEDIA: *Comparison of virtual machines — Wikipedia, The Free Encyclopedia*, 2008. `http://en.wikipedia.org/w/index.php?title=Comparison_of_virtual_machines&oldid=246426608`; Abgerufen am 21. Oktober 2008.