Diplomarbeit

Sichere Virtualisierung auf dem *Fiasco*-Mikrokern

Henning Schild

31. Juli 2008

Technische Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur
Professur Betriebssysteme

Betreuender Hochschullehrer: Prof. Dr. rer. nat. Hermann Härtig
Betreuender Mitarbeiter: Dipl.-Inf. Michael Peter

Technische Universität Dresden
Fakultät Informatik

AUFGABENSTELLUNG FÜR DIE DIPLOMARBEIT

Name des Studenten: **Henning Schild**

Studiengang: Informationssystemtechnik
Immatrikulationsnummer: 2932479

Thema:

Sichere Virtualisierung auf dem Fiasco-Mikrokern

Zielstellung:

Zwei Schlüsseltechnologien zur Erreichung der Sicherheits-, Funktionalitäts- und Leistungsziele in kritischen Anwendungsfeldern sind die Betriebssystemkonstruktion nach dem Mikrokernprinzip und die Nutzung von Virtualisierung zur kontrollierten Weiternutzung vorhandener Betriebssysteme und Applikationen.

Virtuelle Maschinen erlauben es, ein weites Spektrum an vorhandener Software ohne aufwendige Modifikationen weiterzuverwenden. Die Ausführung in einer virtuellen Maschine bietet eine geeignete Ausführungsumgebung und schränkt die Interaktionsmöglichkeiten mit der Plattform ein. Bei geeigneter Konfiguration kann sie zur Kapselung verwendet werden. Es existieren mehrere Ansätze der Realisierung, die einen weiten Bereich mit Hinblick auf Funktionalität, Leistungsfähigkeit und Sicherheit abdecken. Bisherige Implementierungen sind entweder sicher oder leistungsfähig, haben aber in dem jeweilig anderen Punkt erhebliche Defizite. Ziel der Arbeit ist es, eine sichere und möglichst schnelle Umgebung für Virtualisierung zu entwerfen, zu implementieren und zu bewerten.

Im ersten Teil der Arbeit soll untersucht werden, welche Ansätze möglich sind. Diese sind hinsichtlich ihre Leistungsfähigkeit und ihres Realisierungsaufwandes zu bewerten. Als relevante Arbeitslast ist dabei die oftmals für Vergleichszwecke herangezogene Übersetzung eines Linux-Kerns anzunehmen.

Die am besten beurteilte Variante soll im zweiten Teil implementiert und anhand von Testszenarien bewertet werden. Neben der angenommenen Hauptarbeitslast sind dabei sowohl weitere Betriebssysteme wie auch charakterisierende Mikrobenchmarks zu berücksichtigen. Außer den Sicherheitsaspekten sind auch Vergleiche zu anderen Virtualisierungslösungen von Interesse.

verantwortlicher Hochschullehrer: Prof. Dr. Hermann Härtig

Betreuer: Dipl.-Inf. Michael Peter
Institut: Systemarchitektur
Professur: Betriebssysteme
Beginn: 01. 02. 2008
Einzureichen: 31. 07. 2008

Dresden, 15. 01. 2008

Unterschrift

Prof. Dr. Hermann Härtig

Erklärung

Hiermit erkläre ich, dass ich diese Arbeit selbstständig erstellt und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Dresden, den 31. Juli 2008

Henning Schild

Danksagung

Ich möchte mich an dieser Stelle bei Michael Peter bedanken. Er hat mich in allen Phasen meiner Arbeit sehr engagiert betreut. Durch seine fachliche Kompetenz war er immer der richtig Ansprechpartner für technische Fragen. Seine Anregungen und Kritik haben einen entscheidenden Beitrag zu dieser Arbeit geleistet. Bei Adam Lackorzynski und Alexander Warg möchte ich mich auch bedanken. Sie haben mir ebenfalls in vielen technischen Fragen zur Seite gestanden.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Rechner werden immer leistungsfähiger und es ist zunehmend zu beobachten, dass sie von einzelnen Anwendungen nicht vollständig ausgelastet werden. Daraus ergibt sich, dass Virtualisierung, die schon seit den 1970er Jahren eingesetzt wird, nicht mehr nur für Großrechner interessant ist. Durch die kontrollierte Wiederverwendung von vorhandenen Betriebssystemen und Applikationen und die Isolation, die Virtualisierung bietet, kann die Funktionalität vorhandener Systeme gesteigert, und erhöhten Sicherheitsanforderungen entsprochen werden.

In den letzten Jahren hat Virtualisierung wieder an Bedeutung gewonnen. Hohe Leistungsfähigkeit bei geringen Preisen lassen aktuelle Systeme zu attraktiven Alternativen zu Großrechner–Systemen werden. So, wie Virtualisierung heute in Rechenzentren weit verbreitet ist, wird sie in den kommenden Jahren auch im Bereich des *Personal Computings* zu einer Standardtechnologie werden.

Computersysteme verarbeiten vertrauliche Daten von hohem Wert und sind ein wichtiger Teil unseres Alltags. Sie müssen gegen Angriffe auf ihre Verfügbarkeit, Integrität und Vertraulichkeit abgesichert werden. Eine Schlüsseltechnologie zur Konstruktion sicherer Systeme ist die Verwendung von Mikrokernen. Diese minimalen Betriebssystemkerne implementieren nur die grundlegenden Mechanismen für die Betriebssystemkonstruktion. Sie bilden eine zuverlässige Basis, auf der komplexe Systeme konstruiert werden können. Die *Trusted Computing Base* (TCB) von Systemen beinhaltet alle Teile, denen im Bezug auf Sicherheitsziele von Anwendungen vertraut werden muss. Der Betriebssystemkern ist Teil der *Trusted Computing Base*. Die Größe von Mikrokernen ist wesentlich geringer als die von monolithischen Betriebssystemkernen. Betriebssystemdienste, die nicht von allen Applikationen benötigt wird, ist bei mikrokernbasierten Betriebssystemen aus dem Kern ausgelagert. Dazu zählen zum Beispiel Gerätetreiber oder Dateisysteme. Sie sind nur Teil der TCB einer Applikation, wenn sie von ihr genutzt werden. Im Vergleich dazu ist bei Anwendungen auf Betriebssystemen mit monolithischen Kernen oftmals viel Funktionalität in der TCB enthalten, die von einzelnen Anwendungen nicht benötigt wird.

Im Gegensatz zu den ersten Mikrokernen, sind aktuelle Mikrokerne der zweiten Generation sehr leistungsfähig. Ein Beispiel für einen solchen Kern, ist der an der Technischen Universität Dresden entwickelte *Fiasco*–Mikrokern. Der Kern bietet Garantien bezüglich des zeitlichen Ausführungsverhaltens, die ihn für Echtzeitanwendungen interessant machen. Mit *L⁴Linux* steht eine Version vom Linux–Kern zur Verfügung, die auf dem *Fiasco*–Mikrokern eingesetzt werden kann. Dieser angepasste Linux–Kern erlaubt es, unmodifizierte Linux–Anwendungen neben sicherheits– oder zeitkritischen Applikationen zu nutzen. Damit können allerdings nur Linux–Applikationen und Linux–Kernfunktionen wiederverwendet werden.

Besser wäre es, wenn man unmodifizierte Betriebssysteme weiterverwenden könnte.

Dadurch fällt der Aufwand, der durch die Kernanpassungen entsteht weg. Es können zudem Betriebssysteme genutzt werden, die zum Beispiel aus rechtlichen Gründen nicht verändert werden können. Mit Hilfe von Emulation kann man unmodifizierte Betriebssysteme auf dem Mikrokern bereits einsetzen. Die virtuellen Maschinen, die damit zur Verfügung stehen, sind allerdings nicht leistungsfähig. Existierende Virtualisierungslösungen, die es erlauben unmodifizierte Betriebssysteme effizient in einer virtuellen Umgebung auszuführen, basieren auf monolithischen Betriebssystemkernen. Sie haben damit eine große *Trusted Computing Base*.

1.1 Ziel der Arbeit

Ziel dieser Arbeit ist es, eine effiziente Virtualisierungslösungen für den *Fiasco*–Mikrokern zu entwickeln. Die Lösung soll unmodifizierte Gast–Betriebssysteme zusammen mit ihren Anwendungen unterstützen. Vergleichsbasis zur Leistungsbewertung sind vorhandene Virtualisierungslösungen, die den gestellten Sicherheitsanforderungen entsprechen. Diese basierten zum Zeitpunkt dieser Arbeit auf kompletter Emulation. Die Leistungen dieser vorhandenen Lösungen soll übertroffen werden. Durch die Verwendung des Mikrokerns kann eine kleine *Trusted Computing Base* für vollständige Virtualisierung erreicht werden. Dadurch ist es möglich, sicherheitskritische Anwendungen neben kompletten virtuellen Maschinen zu verwenden.

2 Grundlagen und Stand der Technik

2.1 Grundlagen von Virtualisierung

2.1.1 Grundbegriffe

Virtualisierung

Der Begriff der Virtualisierung bezeichnet die Abstraktion der Schnittstellen von Ressourcen. Diese Abstraktion wird verwendet, um Ressourcen aufzuteilen oder zusammenzufassen. Je nach Abstraktionsgrad und Anwendungsfall wird dabei die Schnittstelle der physischen Ressourcen nachgebildet oder eine andere Schnittstelle angeboten.

Virtuelle Maschinen

Dei der Aufteilung von Ressourcen einer realen Maschine kann mit Hilfe von Virtualisierung so weit abstrahiert werden, dass mehrere virtuelle Maschinen zur Verfügung stehen. Dadurch können nicht nur mehrere Anwendungsprogramme, sondern auch mehrere Betriebssysteme gleichzeitig auf einer physischen Maschine ausgeführt werden. In dieser Arbeit wird der Begriff der Virtualisierung primär im Bezug auf virtuelle Maschinen, und somit das Aufteilen von physischen Ressourcen, verwendet.

Bei virtuellen Maschinen unterscheidet man den Gast und den Host. Der Begriff des Gastes wird im Bezug auf virtuelle Maschinen verwendet, wobei Host im Bezug auf die zugrundeliegende reale Maschine verwendet wird.

Emulation

Emulation ist eine Form von Virtualisierung, bei der Schnittstellen komplett in Software nachgebildet werden. Sie abstrahiert damit vollständig von zugrundeliegender Hardware. Dadurch ist es möglich, virtuelle Ressourcen anzubieten, die physische nicht vorhanden sind.

2.1.2 Formale Anforderungen

Die folgenden drei formalen Anforderungen an Virtualisierung sind in [PG74] im Bezug auf virtuelle Maschinen aufgeführt. Sie können allgemein auf Virtualisierung angewendet werden.

- Äquivalenz
- Isolation
- Effizienz

Äquivalenz bedeutet, dass das Verhalten von Anwendungen durch Virtualisierung nicht verändert werden darf. Dafür müssen die Schnittstellen der physischen Ressourcen komplett nachgebildet werden. Äquivalenz kann auch erreicht werden, indem Applikationen auf die Schnittstellen virtueller Ressourcen angepasst werden. Im Fall von virtuellen Maschinen bedeutet das, dass ein Gast keine Zustände beobachten darf, die auf einer realen Maschine nicht zu erwarten wären.

Bei der Aufteilung von Ressourcen muss sichergestellt sein, dass Anwendungen nur die Teile dieser Ressourcen verwenden, die ihnen zugewiesen wurden. Der gemeinsame Zugriff auf Teile des Gesamtsystems darf nur erfolgen, wenn er explizit erlaubt ist. Isolation ist nötig, um Äquivalenz sicherzustellen. Für virtuelle Maschinen ergibt sich daraus, dass Gäste nicht auf Ressourcen des Hosts oder anderer Gäste zugreifen dürfen. Sie dürfen das Verhalten der Anderen nicht beeinflussen.

Die dritte Anforderung ist Effizienz. Die Leistungseinbußen durch die Verwendung von Virtualisierung sollen möglichst minimal sein. Die Leistungen, die Anwendungen in virtuellen Umgebungen zeigen, sollen also mit den Leistungen in der jeweils nachgebildeten Umgebung vergleichbar sein.

2.1.3 Gründe Virtualisierung einzusetzen

Auslastung

Virtualisierung kann nur dann sinnvoll eingesetzt werden, wenn Ressourcen durch die Anwendungen, die sie verwenden, nicht komplett benötigt werden. Ist das der Fall, können die übrigen Ressourcen, mit Hilfe von Virtualisierung, für andere Anwendungen zur Verfügung gestellt werden. Mit Virtualisierung lassen sich somit vorhandene Ressourcen besser ausnutzen, die Auslastung von Hardware kann verbessert werden.

Wiederverwendung

Durch Abstraktion von Schnittstellen, ermöglicht Virtualisierung die Wiederverwendung von Applikationen in Umgebungen, in denen diese Anwendungen ohne Virtualisierung nicht eingesetzt werden könnten. Virtualisierung ist für Wiederverwendung vor allem dann interessant, wenn vorhandene Anwendungen nicht auf alternative Schnittstellen angepasst werden können. Der Aufwand solche Schnittstellen mit Hilfe von Virtualisierung nachzubilden ist in der Regel wesentlich geringer, als alle Applikationen, die die Schnittstelle verwenden, anzupassen.

Isolation

Isolation ist eine formale Anforderung an Virtualisierung und wird von ihr damit auch angeboten. Zusätzliche Abstraktionen von vorhandenen Schnittstellen ermöglichen es, einzelne Anwendungen zusätzlich vom Rest des Systems zu isolieren.

2.2 Anwendungen für Virtualisierung

Virtuelle Maschinen erlauben es, Anwendungen, die für unterschiedliche Betriebssysteme entwickelt wurden, auf derselben Maschine zu verwenden. So ist es beispielsweise möglich, auf einer Linux–Installation mit Hilfe von Virtualisierung auch Windows zusammen mit Windows–Anwendungen auszuführen. Eine weitere Möglichkeit ist der Einsatz mehrere Instanzen desselben Betriebssystems in verschiedenen Versionen. Zum Beispiel weil installierte Anwendungen unverträglich sind und somit nicht auf einer Instanz lauffähig wären oder weil ältere Programme strikte Abhängigkeiten haben und nur auf entsprechend alten Versionen des jeweiligen Betriebssystems verwendet werden können. Diese Möglichkeit ist zum Beispiel für Webentwickler interessant, die ihre Webanwendungen mit vielen Web–Browsern in verschiedenen Versionen testen wollen. Für *Internet–Explorer* in den Versionen 5, 6 und 7 sind drei Installationen von Windows notwendig. Für Linux– oder MacOS–Browser werden ebenfalls Installationen dieser Betriebssysteme benötigt, eventuell ebenfalls mehr als eine Instanz pro System.

Virtualisierung kommt in Rechenzentren zur Server–Konsolidierung zum Einsatz. Dabei werden mehrere virtuelle Server auf derselben physischen Maschine installiert. Aus Administrationsgründen ist es oft sinnvoll, Serverdienste auf einzelne Maschinen zu verteilen. Diese Methode zur Isolation führt allerdings dazu, dass für zusätzliche Dienste auch zusätzliche Maschinen benötigt werden. Dazu kommt, dass diese Maschinen je nach Art und Nutzung des Dienstes schlecht ausgelastet sein können. Mit virtuellen Maschinen kann man Isolation erreichen, ohne für jeden Dienst eine physische Maschine zu betreiben. Das steigert die Auslastung der vorhanden Ressourcen und senkt gleichzeitig die Betriebskosten. Die laufenden Kosten werden zum Beispiel durch die Einsparung von Energie und Platz, und durch geringere Wartungskosten gesenkt.

Die Steigerung der Auslastung einzelner physischer Maschinen ist besonders für Systeme mit mehr als einem Prozessor interessant [JVL07]. Viele, vor allem ältere, Anwendungen nutzen für die Erfüllung ihrer Aufgaben nur eine CPU. Um aktuelle Rechnerarchitekturen mit mehreren CPU–Kernen ausnutzen zu können, müssten solche Anwendungen an die neue Umgebung angepasst werden. Eine solche Portierung von Anwendungen ist aufwendig und nicht für alle Arten von Anwendungen sinnvoll. Aus diesem Grund wird in der Praxis oft Virtualisierung eingesetzt, um mehrere solcher Applikationen in virtuellen Maschinen auf derselben physischen Maschine zu nutzen.

Als Beispiel für Server–Konsolidierung seien hier ein Web– und ein Mailserver genannt. Sie sollen aus Sicherheitsgründen nicht auf derselben Maschine laufen, lasten zwei separate Maschinen allerdings nicht aus. Ein weiteres Beispiel ist die Vermietung von Servern. Die Kunden wollen einen vollwertigen Zugang zum gemieteten System. Mit Hilfe von Virtualisierung kann der Betreiber des Rechenzentrums flexibel neue virtuelle Server für seine Kunden bereitstellen. Für sie ist es transparent, dass sie sich eine physische Maschine mit anderen Kunden des Rechenzentrums teilen.

Weitere Anwendungsgebiete sind die Entwicklung und Tests von Software. Betriebssysteme können in virtuellen Maschinen sehr gut getestet werden. Dabei braucht ein Entwickler nur eine physische Maschine für Entwicklung und Tests und muss diese Maschine zum Testen nicht neu starten. Außerdem bieten virtuelle Maschinen verschiedene Möglichkeiten den Maschinenzustand auszulesen und damit Betriebssysteme zu analysieren.

Das kann zum Beispiel bei der Fehlersuche während der Entwicklung vorteilhaft sein.

Virtuelle Maschinen können einfach von einer physischen Maschine auf eine andere übertragen werden. Einige Virtualisierungslösungen ermöglichen eine solche Migration sogar zur Laufzeit. Dadurch werden Wartungsarbeiten in Rechenzentren einfacher, Hardware kann zum Beispiel ohne Ausfallzeiten ausgetauscht oder aufgerüstet werden.

Software kann fertig installiert in einer virtuelle Maschine ausgeliefert werden. Aufwendige oder komplizierte Installationen müssen nur einmal durchgeführt werden und können anschließend einfach vervielfältigt und weitergegeben werden. Solche fertigen Installationen, in Form von virtuellen Maschinen, werden als *Virtual Appliances* [SL03] bezeichnet.

2.3 Anwendungs–Virtualisierung

Bei Anwendungs–Virtualisierung wird auf der Maschine nur ein Betriebssystemkern ausgeführt. Sie konzentriert sich auf die Wiederverwendung von Applikationen in einer bestehenden Umgebung und auf die Isolation von Anwendungen voneinander.

2.3.1 Virtualisierung auf Betriebssystem–Ebene

Wie in 2.2 beschrieben gibt es Fälle, in denen vorhandene Software in einer isolierten Umgebung ausgeführt werden soll. Virtuelle Adressräume und Prozesse sind eine Abstraktion, die einen gewissen Grad an Isolation von Anwendungen erlaubt. Ausführungszeit und Speicher sind allerdings nicht die einzigen Ressourcen, die Betriebssysteme ihren Anwendungen anbieten.

Viele Betriebssysteme bieten Container–Mechanismen, um für einzelne Anwendungen den Zugriff auf Betriebssystemdienste einzuschränken. Mit dem `chroot`–Mechanismus ist es unter Unix–Systemen zum Beispiel möglich, einzelnen Prozessen nur einen Teil des Dateibaums zur Verfügung zu stellen. Diese Prozesse können dann nur auf Dateien in einem Verzeichnis und dessen Unterverzeichnissen zugreifen.

Weitere Beispiele für diese Art der Virtualisierung sind: FreeBSD Jails [KW00] ,Solaris Zones [PT04] und Linux VServer [lin]. Diese Beispiele bieten neben der Isolation auf Dateiebene noch mehr Möglichkeiten zur Isolation. Ich werde allerdings nicht im Detail auf die Unterschiede eingehen. [SPF+07] geht näher auf Containern–basierte Virtualisierung ein und untersucht ihre Eignung für Serverkonsolidierung.

Alle genannten Lösungen haben gemein, dass die Anwendungen die jeweiligen Schnittstellen des Betriebssystem–Kerns verwenden. Sie werden vom Kern stellenweise anders behandelt, um sie vom Rest des Systems zu isolieren. Es sind aber Anwendungen, die für das jeweilige Betriebssystem und dessen Schnittstellen entwickelt wurden.

2.3.2 Nachbildung von Betriebssystem–Schnittstellen

Ein weiterer Anwendungsfall von Virtualisierung ist die Wiederverwendung von Applikationen auf Betriebssystemen, für die diese Anwendungen nicht entwickelt wurden. Um das zu ermöglichen, muss die Schnittstelle (API) des Betriebssystems, für die die

jeweilige Anwendung entwickelt wurde, auf dem vorhandenen Betriebssystem nachgebildet werden. Diese Schnittstellen beinhalten neben den eigentlich Kernschnittstellen ebenfalls Systembibliotheken, die nachgebildet werden müssen.

Ein Beispiele für diese Technik ist *Wine* [AJ]. Es bietet die Möglichkeit, Windows–Applikationen auf Unix–Systemen wie Linux, FreeBSD oder Solaris auszuführen. Ein weiteres Beispiel für diese Art der Virtualisierung ist der FreeBSD–Kern. Er bietet die API des Linux–Kerns an. Dadurch ist es möglich, Linux–Anwendungen auch unter FreeBSD zu verwenden.

Bei dieser Art von Virtualisierung wird für die entsprechenden Anwendungen die Schnittstelle eines anderen Betriebssystems zur Verfügung gestellt. Der Ansatz ist allerdings sehr aufwendig. Neben dem möglicherweise großen Umfang solcher Schnittstellen, können fehlende Dokumentation und sich ändernde Schnittstellen Gründe für den Aufwand sein. Die beiden vorgestellten Beispiele implementieren nur Teile der jeweiligen Schnittstellen. Aus diesem Grund stellen sie keinen vollwertigen Ersatz für die nachgebildeten Betriebssysteme dar. Der Ansatz konzentriert sich mehr auf die Wiederverwendung von vorhandener Software, als auf Isolation.

2.3.3 Emulation einzelner Anwendungen

Weiterhin gibt es den Fall, dass Anwendungen nicht nur eine alternative API erwarten, sondern auch im Binärformat für eine andere Hardware–Architektur vorliegen. Der Code solcher Anwendungen kann mit Hilfe von Emulation ausgeführt werden. Typischerweise muss die Emulation einzelner Anwendungen mit der zuvor erwähnten Imitation der Betriebssystem–API kombiniert werden. Es ist allerdings auch vorstellbar, dass die Kernschnittstelle übereinstimmt.

Beispiele für Emulation einzelner Anwendungen sind *Rosetta* [ros] und *QEMU User space emulation*. Auch bei diesem Ansatz geht es primär um die Wiederverwendung vorhandener Software.

2.4 Plattform–Virtualisierung

Mit Plattform–Virtualisierung werden komplette virtuelle Maschinen zur Verfügung gestellt. Plattform–Virtualisierung bietet die Möglichkeiten von Anwendungs–Virtualisierung und zusätzlich die Möglichkeit mehrere Betriebssysteme gleichzeitig zu verwenden. Wie in 2.3.2 erwähnt, können mit der Nachbildung von Betriebssystem–Schnittstellen in der Regel nicht alle Anwendungen mit vertretbarem Aufwand unterstützt werden. In diesem Fällen kann auf Plattform–Virtualisierung, und damit die tatsächlichen Ziel–Betriebssystemen zurückgegriffen werden. Außerdem kann die Verwendung von mehr als einem Betriebssystem zum Beispiel für zusätzliche Isolation sinnvoll sein. Auf diese Weise können zum Beispiel Sicherheitsprobleme von Gast–Betriebssystemen auf die jeweiligen Gäste beschränkt werden. Die Wiederverwendung von Betriebssystemen kann auch dafür eingesetzt werden, Teile ihrer Funktionen für das Gesamtsystem bereitzustellen. In diesem Zusammenhang steht zum Beispiel die Wiederverwendung von Gerätetreibern, wie sie in [LUSG04] beschrieben ist.

2.4.1 Paravirtualisierung

Paravirtualisierung ist eine Technik, bei der Gast–Betriebssysteme angepasst werden, um den Aufwand für ihre Ausführung in virtuellen Maschinen zu verringern. Die Virtualisierungssoftware bietet dabei eine Schnittstelle an, die sich von der der zugrundeliegenden Hardware unterscheidet. Die Anwendungen der Gast–Systeme müssen typischerweise nicht angepasst werden, sie nutzen dieselbe Schnittstelle, die ihnen auf der herkömmlichen Version des jeweiligen Gast–Betriebssystems auch angeboten wird.

Mit Paravirtualisierung lassen sich virtuelle Maschinen implementieren, deren Leistungsverluste im Vergleich zu realen Maschinen sehr gering sind. Allerdings ist Paravirtualisierung mit dem Aufwand der Portierung von Gast–Systemen verbunden. Betriebssysteme müssen für Paravirtualisierung ganz oder teilweise quelloffen sein.

Beispiele für Paravirtualisierung sind *User Mode Linux* [Dik00] und *L⁴Linux* [HHL+97]. Die Virtualisierungslösung *XEN* [BDF+03] bietet ebenfalls eine Schnittstelle für Paravirtualisierung an. Für *XEN* gibt es zum Beispiel Version von Linux, FreeBSD und Solaris.

2.4.2 Vollständige Virtualisierung

Der Begriff der vollständigen Virtualisierung wird für virtuelle Maschinen verwendet, die dieselben Schnittstellen wie reale Maschinen anbieten. Auf solchen virtuellen Maschinen können unveränderte Gast–Betriebssysteme und deren Anwendungen eingesetzt werden. Verglichen mit den bisher vorgestellten Möglichkeiten, Virtualisierung einzusetzen, bieten sie den höchsten Abstraktionsgrad von den Schnittstellen, die das Host–Betriebssystem auf der physischen Maschine anbietet.

Vollständige Virtualisierung durch Emulation

Vollständige Virtualisierung wird zum Beispiel von den Maschinen–Emulatoren *Bochs* [Law] und *QEMU* [Bel05] angeboten. *Bochs* ist ein Emulator für x86–Maschinen. Mit ihm es zum Beispiel möglich verschiedene Versionen von Windows auf Unix–Systemen einzusetzen. Der Maschinen–Emulator *QEMU* bietet neben der Emulation von x86–Maschinen Unterstützung für weitere Architekturen wie ARM, Sparc oder PowerPC.

Emulation ermöglicht es, virtuelle Hardware anzubieten, die einem real nicht zur Verfügung steht. So können sich die Host– und die Gast–Architektur bei der Verwendung von Emulatoren unterscheiden.

Für die Implementierung von vollständigen virtuellen Maschinen reicht es nicht, nur virtuelle Plattform–Hardware wie eine CPU und virtuellen Arbeitsspeicher zur Verfügung zu stellen. Um Gast–Betriebssysteme in virtuellen Maschinen wie gewohnt ausführen zu können, benötigt man auch virtuelle Peripheriegeräte wie Netzwerkkarten, Grafikkarten und Festplatten. Solche virtuellen Geräte können mit Emulation ebenfalls implementiert werden.

Vollständige Virtualisierung mit *Native Execution*

Im Gegensatz zur reinen Emulation besteht die Möglichkeit, Gästen kontrolliert direkten Zugriff auf physische Ressourcen zu geben. Bei Virtualisierung mit *Native Execution* wird dem Gast direkter Zugriff auf die CPU und den Arbeitsspeicher der physischen Maschine gegeben. Mit dieser Technik können virtuelle Maschinen implementiert werden, die in ihrer Leistungsfähigkeit mit realen Maschinen vergleichbar sind. *Native Execution* erlaubt es, der Anforderung nach Effizienz gerecht zu werden, wobei Isolation und Äquivalenz auch eingehalten werden.

Beispiele für vollständige Virtualisierungslösungen, die mit *Native Execution* arbeiten, sind: *KVM* [Hab08], *VMware* [vmw], *VirtualBox* [SM08] und *QEMU* mit *kqemu* [Bel].

Durch die Verwendung von *Native Execution* besteht bei diesen virtuellen Maschinen die Einschränkung, dass die Host– und Gast–Architektur übereinstimmen müssen. Ressourcen wie die CPU und Speicher werden direkt vom Gast genutzt, deshalb können ihre Schnittstellen nur begrenzt abstrahiert werden.

Diese Art von virtuellen Maschinen wird in der Regel mit Emulation kombiniert. Gast–Code, der nicht nativ ausgeführt werden kann, wird an einen Emulator übergeben. Außerdem wird Emulation verwendet, um in dieser Art von virtuellen Maschinen Peripheriegeräte bereitzustellen.

2.5 Vollständige x86–Plattform–Virtualisierung

Virtualisierung hat für x86–Maschinen in den letzten Jahren an Bedeutung gewonnen. Prozessoren werden immer leistungsfähiger, so dass Ressourcen für zusätzliche Anwendungen, wie virtuelle Maschinen zur Verfügung stehen.

Effiziente virtuelle Maschinen lassen sich nur mit Hilfe von *Native Execution* implementieren. Die grundlegende Methode für diese native Ausführung nennt sich *Trap–and–Emulate*. Diese Methode geht davon aus, dass die reale Maschine mehrere Privilegierungsstufen anbietet. Gast–Code wird generell mit eingeschränkten Privilegien ausgeführt, um Isolation zu gewährleisten. Alle privilegierten Instruktionen des Gast–Codes lösen dann Fehler (*Exceptions*) aus. Der *Virtual Machine Monitor* (VMM) [Gol74] behandelt diese *Exceptions* indem er alle Effekte der fehlgeschlagenen Befehle nachbildet, zum Beispiel durch Emulation.

In der x86–Architektur gibt es vier Privilegierungsstufen, diese werden Ringe genannt. Ring 0 ist die Stufe mit den meisten Privilegien (Systemmodus) während Code, der in Ring 3 (Nutzermodus) ausgeführt wird, die wenigsten Privilegien hat. Die beiden übrigen Ringe bilden Abstufungen zwischen dem System– und dem Nutzermodus. Die meisten x86–Betriebssysteme verwenden nur die Ringe 3 und 0 für die Trennung von Anwendungen vom Kern.

Um Isolation zu garantieren, darf *Native Execution* nur im Nutzermodus des Hosts passieren. Daraus ergeben sich zwei verschiedene Möglichkeiten, virtuelle Maschinen mit nativer Ausführung von Gast–Code zu implementieren. Die erste Möglichkeit besteht darin, dass Gast–Code aller vier Ringe auf dem Host im Nutzermodus ausgeführt wird. Bei dieser Ringkompression wird also auch Gast–Code, der im Normalfall im Systemmodus der CPU laufen würde, im Nutzermodus auf der Host–CPU ausgeführt. Die

Alternative ist Gast–Code nur dann nativ auszuführen, wenn das Gast–System im Nutzermodus läuft. In dem Fall wird kein Gast–Code in einem Ring ausgeführt, für den er nicht vorgesehen ist. Es muss allerdings für Kern–Code immer auf Emulation zurückgegriffen werden.

Die x86–Architektur lässt sich nicht einfach virtualisieren. Der Befehlssatz der Architektur enthält Befehle, die nicht als privilegiert eingestuft sind, es für Virtualisierung aber sein müssten. Diese 17 kritischen Instruktionen [RI00] erschweren es, die x86–Schnittstellen entsprechend ihrer Spezifikation äquivalent nachzubilden. Sie verletzen die Forderung nach Äquivalenz entweder weil sie Host–Informationen offenlegen oder sich im Nutzermodus anders verhalten als im Systemmodus. Die zweite Problematik ist nur bei der Verwendung von Ringkompression relevant, während die Offenlegung von Informationen des Hosts auch ohne Ringkompression auftreten kann. Die Motivation Ringkompression trotzdem einzusetzen ist, dass damit weniger Gast–Code im Emulator ausgeführt werden muss. Mit Ringkompression kann die Anforderung nach Effizienz demnach besser erfüllt werden.

[RI00] zeigt nicht nur alle kritischen Operationen auf, es wird auch detailliert beschrieben, warum sie problematisch sind. In der Folge beschreibe ich kurz einige der kritischen Operationen. Die PUSH–Instruktion kann zum Beispiel verwendet werden, um durch das Lesen von bestimmten Registern die aktuelle Privilegierungsstufe der CPU (CPL) abzufragen. Ihr Ergebnis ist also also in jedem der vier Ringe unterschiedlich. Bei der Verwendung von Ringkompression kann Gast–Code, der mit dem PUSH–Befehl CPL abfragt sein Verhalten möglicherweise ändern. Instruktionen wie SIDT, SGDT und SLDT legen den Zustand der Hosts für den Gast–Code offen. Dieser kann sich von dem, vom Gast–Code erwarteten Zustand unterscheiden. Damit sind diese Instruktion auch kritisch, wenn keine Ringkompression angewendet wird.

2.5.1 Schließen von Virtualisierungslücken in Software

Die oben genannten Virtualisierungslücken können mit verschiedenen Techniken teilweise in Software geschlossen werden. Komplette Äquivalenz zur Schnittstelle der x86–Architektur kann mit ihnen allerdings nicht erreicht werden.

Gast–Code kann vor der Ausführung durchsucht und angepasst werden. Dabei wird Code, der kritische Operationen enthält, durch anderen Code ersetzt. Dafür gibt es die Möglichkeit, den Code so anzupassen, dass er *Exceptions* auslöst. Durch diese *Exceptions* wird die Kontrolle an den VMM übergeben, wo der Code mit Emulation behandelt werden kann. Es ist allerdings auch möglich, kritische Operationen durch Code zu ersetzen, der keine kritischen Operationen enthält und dabei die Effekte des ursprünglichen Codes simuliert.

Das Durchsuchen des Gast–Codes muss im besten Fall nur einmal erfolgen. Wenn der VMM Gast–Systeme unterscheiden und erkennen kann, ist es somit möglich, vorgefertigte Listen von Anpassungen mit der Virtualisierungssoftware auszuliefern.

Anpassen von kritischem Gast–Code ist aufwendig. Dabei müssen eine Reihe von Problemen gelöst werden. Eines dieser Probleme ist unterschiedliche Codelänge vom angepassten zum ursprünglichen Code. [AA06] beschreibt diese Probleme und mögliche Lösungen. *VirtualBox* [SM08] und *VMware* sind Produkte die solche Techniken einsetzen

um die Virtualisierungslücken der x86–Architektur zu stopfen.

2.5.2 Virtualisierung mit Hardware–Unterstützung

AMD und Intel haben die Erweiterungen *AMD Virtualization* (AMD–V) beziehungsweise *Intel Virtualization Technology* (IVT) für ihre x86–Prozessoren entwickelt. Diese Erweiterungen sind in den meisten aktuellen Prozessoren der beiden Hersteller verfügbar.

AMD–V und IVT schließen die Virtualisierungslücken herkömmlicher x86–Prozessoren und erlauben es, einfache *Trap–and–Emulate* VMMs zu verwenden. Viele Virtualisierungslösungen nutzen diese Hardware–Unterstützung bereits aus. Beispiele sind *VMware*, *XEN* und *KVM*.

Die erste Generation von Hardware–Unterstützung hat sich primär auf das Schließen der Virtualisierungslücken konzentriert, um die Komplexität von VMMs zu senken. Ein signifikanter Leistungsgewinn gegenüber ausgereiften VMMs wie *VMware* ist durch diese Art der Hardware–Unterstützung nicht zu erzielen. Neuere Generationen beinhalten mit *Nested Paging* erweiterte Virtualisierungs–Unterstützung für die MMU. Mit *Nested Paging* lösen *Page Faults* im Gast–Kontext keine Kontextwechsel zum Host–Kontext mehr aus, sie können ohne die MMU–Emulation des VMMs aufgelöst werden. Dadurch kann die Leistung von virtuellen Maschinen verbessert werden [AA06].

2.5.3 Peripherie

Für virtuelle Maschinen werden neben CPU und Speicher auch noch andere virtuelle Geräte benötigt. Diese Peripheriegeräte können komplett in Software nachgebildet, also emuliert werden. Alternativ ist es mögliche, dass Gäste kontrolliert direkten Zugriff auf physische Geräte bekommen.

Emulation

Emulation von Peripheriegeräten ist eine weit verbreitete Möglichkeit, Geräte für virtuelle Maschinen bereitzustellen. Geräte, auf die direkter Zugriff nicht möglich ist, oder die physisch nicht vorhanden sind, können Gast–Systemen mit Hilfe von Emulation zur Verfügung gestellt werden. Im Rahmen des *QEMU*–Projekts sind freie Geräte–Modelle für Standardgeräte entstanden, für die Treiber bei vielen Betriebssystemen schon im Lieferumfang enthalten sind. Darunter fallen zum Beispiel *ne2000*–Netzwerk– oder *cirrus*–Grafikkarten. Die *QEMU*–Geräte–Modelle werden auch von anderen Projekten wie *XEN* oder *VirtualBox* verwendet.

Die Leistung solcher emulierten Geräte sind teilweise wesentlich schlechter als die Leistungen realer Geräte.

Direkter Zugriff

Auch bei Peripheriegeräten müssen die formalen Anforderung erfüllt bleiben. Geräte wie Netzwerk– oder Grafikkarten können mittels *Direct Memory Access* (DMA) direkt auf physischen Speicher des Hosts zugreifen. Die Kontrollmechanismen der CPU dabei

umgangen und der Inhalt des Hauptspeichers kann gelesen oder verändert werden. Damit könnten Gerätetreiber des Gastes, Speicher vom Host–System oder von anderen Gästen lesen oder manipulieren. Somit wäre die Forderung nach Isolation verletzt. Außerdem beziehen sich DMA–Operationen auf den physischen Speicher des Hosts. Der Speicher, den der Gast als seinen physischen Speicher benutzt, ist auf dem Host wahrscheinlich auf anderen Adressen abgelegt und kann auch aus dem physischen Speicher des Hosts ausgelagert sein. Das heißt, auch die Forderung nach Äquivalenz kann bei der Verwendung von DMA nicht erfüllt werden.

Abhilfe schaffen Paravirtualisierung oder Hardware, die Virtualisierung explizit unterstützt. Gerätetreiber verwenden für die Kommunikation mit dem jeweiligen Gerät privilegierte Operationen. Dadurch entstehen Kontetextwechseln in den VMM. In [SVL] zeigen die Autoren am Beispiel einer Netzwerkkarte, dass Gerätetreiber viele Kontextwechsel auslösen und dadurch Leistungseinbußen entstehen. Der von ihnen untersuchte Netzwerktreiber benutzt elf privilegiert Instruktionen für das Versenden eines einzelnen Paktes. Außerdem demonstrieren sie, wie die Leistung zum Beispiel mit paravirtualisierten Gerätetreibern gesteigert werden kann. Die Grundidee besteht darin, die Anzahl der Unterbrechungen durch Gerätetreiber zu reduzieren. Mit Paravirtualisierung lassen sich Netzwerktreiber entwickeln, bei denen lediglich eine Unterbrechung pro Paket notwendig ist oder sogar mehrere Pakete pro Kontextwechsel verschickt werden können. Paravirtualisierte Gerätetreiber werden zum Beispiel von *XEN* [FHN+04] und *VMware* eingesetzt.

Aktuelle Versionen der x86–Architekturerweiterungen von Intel und AMD enthalten auch Anpassungen am *Input/Output* (I/O) Subsystem. Beide Hersteller unterstützen sogenannte I/O–MMUs. Eine I/O–MMU kann die Adressumsetzung von physischen Adressen, die der Gast verwendet, auf tatsächliche physische Adressen des Host–Systems übersetzen. Bei entsprechender Konfiguration lässt sich damit Isolation durchsetzen, auch wenn der Gast Zugriff auf Geräte hat, die DMA unterstützen. DMA–Operationen werden auf den physischen Speicher des Gastes beschränkt, wodurch der Speicher anderer Gäste und des Hosts geschützt ist. Mit dieser Erweiterung können Peripheriegeräte unter Einhaltung von Isolation und Äquivalenz direkt von Gast–Systemen genutzt werden.

Wenn solche Geräte von mehreren Gästen gleichzeitig verwendet werden sollen, ist außerdem Virtualisierungs–Unterstützung auf Seite der Geräte nötig. Der Zustand von Geräten, die Virtualisierung unterstützen, kann gesichert und entsprechend rückgesichert werden, oder sie unterstützen von sich aus mehrere Zustände zwischen denen umgeschaltet werden kann. Damit besteht die Möglichkeit, Geräte in verschiedenen Kontexten zu betreiben. Sie können, genau wie die CPU, im Zeitmultiplexverfahren zwischen mehreren Gast–Kontexten und dem Host–Kontext umgeschaltet werden. Geräte ohne Virtualisierungs–Unterstützung können trotz I/O–MMU in der Regel nur einem Gast zur Verfügung gestellt werden.

2.6 Sicherheit

Existierende Virtualisierungslösungen bestehen aus Anwendungen, die unter anderem Emulation für kritischen Code oder Peripheriegeräte durchführen und Schnittstellen zur Bedienung der virtuellen Maschinen anbieten. Diese Anwendungen laufen unter Betriebssystemen mit monolithischen Kernen wie Linux, Windows, Solaris oder BSD. Die jeweiligen Betriebssystem–Kerne bieten in der Regel keine ausreichenden Mechanismen, um virtuelle Maschinen zu unterstützen. Aus diesem Grund wird die Unterstützung für Virtualisierung durch dynamisch nachgeladene Erweiterungen zur Verfügung gestellt. *VMware*, *VirtualBox* und *kqemu* funktionieren nach diesem Prinzip.

Die *Trusted Computing Base* (TCB) einer typischen Installation solcher Virtualisierungssoftware ist sehr groß. Sie ist im Wesentlichen identisch mit der TCB der verwendeten Host–Betriebssysteme und besteht aus einigen hunderttausend bis zu mehreren Millionen Zeilen Quellcode. Komplexität und Größe der TCB stehen im direkten Zusammenhang mit der Sicherheit von Systemen. Hohe Komplexität von Software geht mit Fehlern und damit Verwundbarkeit einher. Solche Fehler können bei Code aus der TCB nicht nur die jeweilige Funktion behindern, sondern die Sicherheit des ganzen Systems gefährden. Für sicherheitskritische Anwendungen ist es wünschenswert, den Umfang der TCB möglichst gering zu halten [SPHH06].

Die erwähnten Kernerweiterungen werden zusammen mit der jeweiligen Virtualisierungssoftware ausgeliefert. Sie kommen in der Regel nicht direkt vom Hersteller des Host–Betriebssystems. Dynamisch nachladbare Kernerweiterungen können ein Sicherheitsproblem darstellen [SESS96]. Wenn sie von Drittanbietern geliefert werden und nicht quelloffen sind, kann ihnen eventuell nicht vertraut werden.

Der *Fiasco*–Mikrokern bietet eine kleine TCB. Außerdem besteht mit *L⁴Linux* die Möglichkeit, vorhandene Linux–Applikationen neben sicherheitskritischen Anwendungen auf dem Mikrokern zu verwenden. Diese Lösung basiert auf Paravirtualisierung und bietet nur Linux als Gast–Betriebssystem an. [MMH08] stellt eine angepasste Version von *XEN* mit reduzierter TCB vor. Die TCB, die mit diesem Ansatz erreicht werden konnte ist trotz der Reduktion noch sehr groß.

Vollständige Virtualisierung mit einer kleinen TCB ist wünschenswert, um nicht vertrauenswürdige Betriebssysteme und ihre Anwendungen neben sicherheitskritischen Anwendungen auf derselben Maschine zu nutzen.

3 Analyse

3.1 Wahl der Software

Für die Umsetzung der gesteckten Ziele bot sich *QEMU* an. Dieser Emulator erfüllt wichtige Kriterien um für eine Implementierung in Frage zu kommen:

- läuft auf Linux und damit auch auf *L⁴Linux*
- es gibt bereits eine Kernerweiterung für *Native Execution*
- Quellen sind frei verfügbar

Mit *L⁴Linux* hat man die Möglichkeit, vorhandene Linux–Software auf dem Mikrokern *Fiasco* zu verwenden. Da es für Linux als Host–System schon diverse Virtualisierungslösungen gibt, war es naheliegend, eine der vorhandenen zu nutzen.

Für *QEMU* gibt es *kqemu*, eine Kernerweiterung zur Virtualisierungs–Beschleunigung mittels *Native Execution*. Diese Kernerweiterung ist für Windows, Linux und FreeBSD verfügbar. Die Linux–Version kann allerdings nicht für *L⁴Linux* verwendet werden da sie davon ausgeht, dass Linux in Ring 0 läuft, was bei *L⁴Linux* nicht der Fall ist.

Eine Weiterverwendung einer vorhandenen Virtualisierungs–Software ist nur dann sinnvoll möglich wenn deren Quellen verfügbar sind und auch angepasst werden dürfen.

Ich habe mich für mich die Portierung von *kqemu*, der Kernerweiterung für den Emulator *QEMU*, auf *L⁴Linux* entschieden.

3.2 Architektur

kqemu besteht aus zwei Teilen. Dem Teil, der im Kontext des Host–Kerns läuft, und dem *monitor*. Der *monitor* stellt eine Ausführungsumgebung für Gast–Code zur Verfügung und verwaltet diese.

Abbildung 3.1 zeigt die beiden Komponenten zusammen mit dem Kern und Anwendungsprogrammen des Hosts. Es ist zu sehen, dass der *monitor* neben dem Host–Kern ebenfalls in Ring 0 läuft. Der Gast–Code hingegen läuft im Nutzermodus, also in Ring 3.

Die folgende Beschreibung der Funktionsweise und Schnittstellen von *kqemu* bezieht sich auf die Version 1.3.0*pre*11, die ich im Rahmen meiner Arbeit verwendet habe.

3.3 Kernteil

Der Kernteil ist der Code von *kqemu*, der tatsächlich im Kontext des Hosts, also in dessen Betriebssystemkern ausgeführt wird. Er stellt eine Schnittstelle zwischen *QEMU*

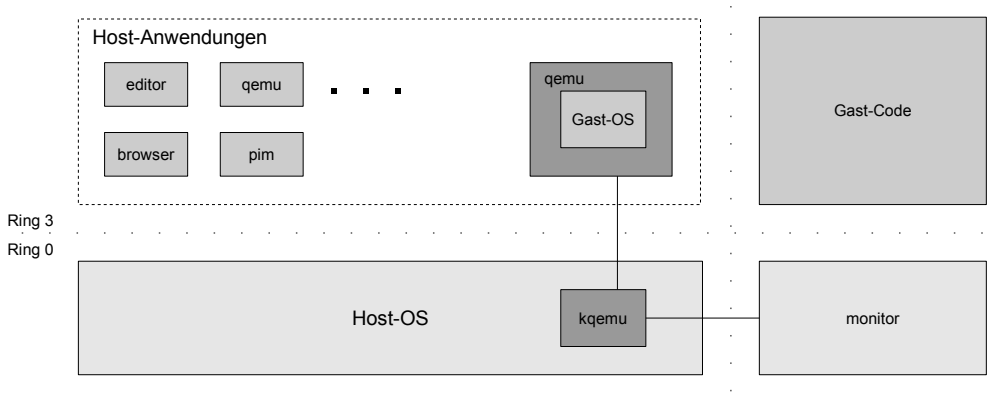

Abbildung 3.1: Architektur von *kqemu*

und dem *monitor* zur Verfügung. Dieser Teil von *kqemu* ist vom Host–Betriebssystem abhängig. Seine Implementierung unterscheidet sich für die verschiedenen unterstützten Host–Kerne.

Der Kernteil implementiert die Schnittstelle für *QEMU*. Über diese Schnittstelle kann eine Anwendung auf dem Host, also der *QEMU*–Prozess, in den Kernteil eintreten. Die folgenden Operationen werden von dieser Schnittstelle angeboten:

- `KQEMU_GET_VERSION` — die Version der Schnittstelle erfragen
- `KQEMU_INIT` — einen neuen Ausführungskontext initialisieren
- `KQEMU_EXEC` — Kontextwechseln in den *monitor*, um Gast–Code auszuführen
- `KQEMU_MODIFY_RAM_PAGES` — Änderungen am physischen Speicher des Gastes signalisieren

In der von mir betrachteten Version von *kqemu* werden nur die ersten drei dieser Operationen tatsächlich verwendet. Die Funktion `KQEMU_MODIFY_RAM_PAGES` ist nicht implementiert. Veränderungen am Gast–Speicher werden durch gemeinsam genutzte Datenstrukturen zwischen dem *QEMU*–Prozess und dem *monitor* signalisiert.

Für die eigentliche Ausführung von Gast–Code mit *kqemu* wird die Funktion `KQEMU_EXEC` verwendet. Mit ihr übergibt der *QEMU*–Prozess zunächst die Kontrolle an den Kernteil. Die Kommunikation zwischen dem Kernteil und *QEMU* findet dabei über den Rückgabewert dieser Funktion und gemeinsame Datenstrukturen statt. Die wichtigsten Rückgabewerte sind dabei:

- `KQEMU_RET_INT` — Unterbrechung reflektieren
- `KQEMU_RET_EXCEPTION` — Fehler reflektieren
- `KQEMU_RET_SOFTMMU` — mit Emulation fortfahren

Die ersten beiden Rückgabewerte signalisieren Fehler oder Unterbrechungen, die nicht vom *monitor* behandelt werden können. Der jeweilige Fehler wird daraufhin von *QEMU* an das Gast–Betriebssystem weitergeleitet. Reflexion wird immer dann benutzt, wenn der Gast–Code Fehler ausgelöst hat, die er auf einer realen Maschine ebenfalls ausgelöst hätte. Der Rückgabewert `KQEMU_RET_SOFTMMU` signalisiert *QEMU*, dass der folgende Gast–Code im Emulator ausgeführt werden soll. Er wird immer dann verwendet, wenn *kqemu* den Gast–Code aus irgendeinem Grund nicht nativ ausführen kann.

Neben der Schnittstelle für *QEMU* implementiert der Kernteil auch eine Schnittstelle für den *monitor*. Über diese Schnittstelle unterstützt der Kern den *monitor* bei der Verwaltung des Adressraums für den Gast–Code. Die wesentlichen Funktionen sind:

- `MON_REQ_ALLOC_PAGE` — Speicher allozieren
- `MON_REQ_LOCK_USER_PAGE` — Speicher *Pinning* und Adressumsetzung
- `MON_REQ_UNLOCK_USER_PAGE` — Speicher *Un–Pinning*

Der *monitor* benötigt für verschiedene Datenstrukturen Speicher, der bei der Initialisierung noch nicht reserviert wird. Dieser Speicher wird nach Bedarf beim Host–Kern angefordert.

Da das Host–Betriebssystem die Seitentabellen des *monitor*s nicht verwaltet, ist in dessen Speicherverwaltung auch nicht bekannt, dass die Rahmen noch in einem Adressraum referenziert werden. Um zu verhindern, dass das Host–Betriebssystem diesen Speicher auslagert und den Inhalt der jeweiligen Rahmen somit ändert, muss der Speicher durch *Pinning* festgesetzt werden. Das *Pinning* wird dabei für alle Rahmen verwendet, die zum jeweiligen Zeitpunkt im virtuellen Adressraum des *monitor*–Kontextes eingeblendet sind.

Für *Pinning* gibt es auch die Umkehroperation. Beim *Un–Pinning* wird der Speicher wieder zur Auslagerung freigegeben. Die Kernerweiterung verwendet *Pinning* höchstens für die Hälfte des physischen Speichers, um das Host–Betriebssystem bei der Speicherverwaltung nicht zu stark einzuschränken. Ist diese Grenze erreicht, werden einzelne Rahmen wieder frei gegeben. Diese Freigabe kann nur erteilt werden, wenn die Rahmen nicht mehr in einem *monitor*–Adressraum eingeblendet sind. Beim *Un–Pinning* werden Rahmen bevorzugt, die gerade nicht referenziert werden. Sind keine freien Rahmen mehr verfügbar, müssen referenzierte Rahmen freigegeben werden, wobei die entsprechenden Referenzen aus den Seitentabellen entfernt werden. Die Funktion `MON_REQ_UNLOCK_USER_PAGE` wird außerdem für das kontrollierte Beenden von *kqemu* benötigt. Wenn keine Gäste aktiv sind oder das Modul aus dem Kern entfernt wird, werden alle gesperrten Rahmen wieder zur Auslagerung freigegeben.

Um Speicher aus dem *QEMU*–Prozess in den Adressraum des *monitor*s einblenden zu können, muss die Rahmennummer, also die Adresse im physischen Speicher des Hosts, bekannt sein. In *QEMU* sind allerdings nur die virtuellen Adressen der jeweiligen Datenstrukturen bekannt. Diese virtuellen Adressen werden bei der Interaktion mit dem Kernteil an diesen übergeben. Der Kernteil kann dann mit Mechanismen des Host–Kerns die Übersetzung von virtuellen Adressen zu physischen Rahmen vornehmen. Die Adressumsetzung und das *Pinning* sind in derselben Funktion implementiert.

Neben den Funktionen zur Speicherverwaltung sind in der Schnittstelle zwischen dem Kernteil und dem *monitor* noch weitere Funktionen implementiert.

- `MON_REQ_EXIT` — zu *QEMU* zurück schalten
- `MON_REQ_ABORT` — Abbruch bei schweren Fehlern
- `MON_REQ_IRQ` — Unterbrechung signalisieren
- `MON_REQ_EXCEPTION` — Fehler signalisieren
- `MON_REQ_LOG` — Protokollierung, Ausgabe von Warnungen oder Fehlermeldungen

Diese Funktionen dienen dem Kontrollfluß und sind weniger interessant, um die Funktionsweise von *kqemu* zu verstehen. Interessant ist lediglich die Signalisierung von *Interrupts/Exceptions*.

Im *monitor*–Kontext werden alle *Interrupts* zunächst vom *monitor*, und nicht, wie üblich, vom Host–Kern behandelt. Alle Unterbrechungen, die der Gast–Code auslöst, können nur vom VMM behandelt werden und müssen diesem dafür zugestellt werden. Allerdings können auch externe Unterbrechungen auftreten, die nicht vom Gast–Code, sondern von Geräten oder von einem *Timer* ausgelöst wurden. Diese Fälle werden vom Host–Kern behandelt und aus diesem Grund vom *monitor* an den Kernteil weitergeleitet. *Timer–Interrupts* werden zum Beispiel an den Host–Kern weitergeleitet und Seitenfehler vom *monitor* behandelt. Wieder andere Ereignisse werden an *QEMU* weitergeleitet. Das betrifft im normalen Modus ohne Ringkompression zum Beispiel alle Kerneintritte im Gast–Code.

3.4 *monitor*

Der *monitor* stellt für den Gast–Code eine Ausführungsumgebung zur Verfügung. Er läuft in einem eigenen Kontext, der vom Host–Betriebssystem weitestgehend unabhängig ist. Diese Unabhängigkeit trifft auch auf seine Implementierung zu. Der *monitor*–Code ist für alle unterstützten Host–Betriebssysteme gleich.

Die Umgebung, in der der Gast–Code ausgeführt wird, bietet einen fast leeren virtuellen Adressraum. In dem Adressraum sind 32 Megabyte zusammenhängender Speicher im oberen Gigabyte für den *monitor* reserviert. Der restliche Speicher kann vom Gast verwendet werden. Kollisionen mit dem reservierten *monitor*–Speicher werden erkannt, weil der entsprechende Speicher für Nutzer–Code schreib– und lesegeschützt eingeblendet ist. Im Fall solcher Kollisionen wird *QEMU* zur Emulation verwendet. Viele Betriebssysteme, wie zum Beispiel Linux, Windows oder FreeBSD, nutzen den obereren Teil des virtuellen Adressraums ausschließlich für den Kern. Aus diesem Grund sind Kollisionen mit diesem reservierten Speicher, für den normalen Betriebsmodus ohne Ringkompression, in der Praxis nur selten ein Problem.

Die 32 Megabyte sind virtuelle zusammenhängend und können nicht dynamisch verschoben werden. Ein solches Verschieben ist sinnvoll, wenn die Ausführung durch häufige Kollisionen und die damit verbundene Emulation verlangsamt wird. Der *monitor* könnte dann an eine Position im virtuellen Adressraum verschoben werden, an der beim

jeweiligen Gast keine oder weniger Speicherzugriffe Kollisionen auslösen. Laut der Dokumentation von *kqemu* ist diese Funktion geplant, sie ist aber in der aktuellen Version noch nicht implementiert.

Der *monitor* läuft mit eigenen System–Datenstrukturen. Darunter zählen neben den Seitentabellen auch die Tabellen für Speicher–Segmentbeschreibungen (GDT und LDT). Außerdem läuft der *monitor* mit einer eigenen *Interrupt Descriptor Table* (IDT). Diese Tabelle weist jeder *Exception* beziehungsweise jedem *Interrupt* einen Zeiger auf den Code zu, der zur Behandlung angesprungen werden soll. Der *monitor* verwendet eine alternative IDT, um zunächst alle *Interrupts* abfangen zu können. Wie schon erwähnt, werden einige *Interrupts* direkt vom *monitor* behandelt, während andere an *QEMU* oder den Host–Kern reflektiert werden.

3.4.1 Segmentierung

Mit Segmentierung bietet die x86–Architektur neben virtuellem Speicher (Paging) noch einen weiteren Kontrollmechanismus für Speicher. Der virtuelle Adressraum kann zusätzlich in Segmente aufgeteilt werden. Segmentierung bietet die Möglichkeit, Teilbereiche des virtuellen Adressraums mit zusätzlichen Schutzmechanismen zu versehen. Speichersegmente werden durch ihre Basisadresse im virtuellen Adressraum, ihre Größe und einer Menge von Attributen beschrieben. Die Segementbeschreibungen werden in zwei Tabellen, der GDT (*Global Descriptor Table*) und der LDT(*Local Descriptor Table*) abgelegt.

Jegliche Speicherzugriffe auf virtuellen Speicher sind mit der Angabe des zu verwendenden Segments verbunden.

Die x86–Architektur bietet sechs Segmentregister, deren Inhalt zusammen mit der GDT und der LDT die Segmentierung steuert. Jedes dieser Register enthält einen Selektor, welcher seinerseits aus einer einem Tabellenindex, einem Tabellenindikator und einer Privilegierungsstufe besteht. Welcher Selektor für einen Speicherzugriff verwendet wird, hängt von der Art des Zugriffs ab. Für Code wird implizit der Selektor CS verwendet, während bei Datenzugriffen der Selektor DS genutzt wird. Neben den impliziten Vorgaben können zum Beispiel die Selektoren ES, FS und GS durch Instruktions–Prefixe explizit für einzelne Instruktionen angegeben werden.

Viele x86–Betriebssysteme verwenden Segmentierung nicht oder nur sehr eingeschränkt. Sie schützen den Speicher mit den Möglichkeiten von Paging und initialisieren die Segment–Beschreibungen und –Selektoren so, dass sie keine zusätzliche Schutzfunktion implementieren. Diese Betriebssysteme nutzen Segmente, die den kompletten virtuellen Adressraum abdecken und keine Zugriffsschutz–Attribute definieren.

Um die Forderung nach Äquivalenz für den Gast–Code vollständig zu erfüllen, müssen die komplette GDT, LDT und die Segmentselektoren, die das Gast–Betriebssystem vorgibt, in die Ausführungsumgebung des Gast–Codes übernommen werden. Damit der Gast–Code überhaupt korrekt ausgeführt werden kann, müssen die Segmentselektoren auf Segmente zeigen, die denen des Gastes entsprechen. Insbesondere muss die Basisadresse mit der Vorgabe übereinstimmen, damit die Adressierung von Speicher korrekt funktioniert.

Segmentselektoren können aus Ring 3 gelesen und verändert werden. Aus diesem

Grund müssen die Segmentbeschreibungen in der durch den Index vorgegebenen Tabelle an der entsprechenden Stelle abgelegt sein. Wäre das nicht der Fall, könnte der Gast–Code einen unerwarteten Selektor auslesen und sein Verhalten ändern. Außerdem ist es denkbar, dass Gast–Code Segmentselektoren verändern soll. Stimmen die GDT und LDT nicht mit der erwarteten überein, wäre das nicht möglich.

Bei *kqemu* werden die LDT und die Segmentselektoren entsprechend den Vorgaben in die *monitor*–Umgebung übernommen. Die GDT wird bis auf wenige Einträge, die für den *monitor* benötigt werden vom Gast übernommen.

3.5 Speicherverwaltung

Abbildung 3.2 zeigt die drei Sätze von Seitentabellen, die an der Code–Ausführung mit *kqemu* beteiligt sind. Im oberen Teil der Abbildung ist das Gast–System mit seinen Seitentabellen dargestellt. Der virtuelle Adressraum von Gast–Anwendungen wird in der virtuellen Maschine ganz regulär von deren Betriebssystem konstruiert und verwaltet. Die Abbildung auf physischen Speicher des Gastes erfolgt über die Gast–Seitentabellen in der virtuellen Maschine.

Der physische Speicher des Gastes ist auf dem Host zusammenhängender virtueller Speicher des *QEMU*–Prozesses. Der Begriff des physischen Speichers bezieht sich im Zusammenhang mit dem Gast also nicht auf real existierende Hardware. Er bezieht sich auf virtuellen Speicher, den das Gast–Betriebssystem als Arbeitsspeicher der virtuellen Maschine nutzt.

Für diesen virtuellen Speicher gibt es wiederum einen Satz Seitentabellen, der zur Abbildung auf den physischen Speicher des Hosts dient. Diese Host–Seitentabellen bilden die Adressen also schließlich auf real existierende Hardware ab, sie sind im unteren Teil der Abbildung dargestellt.

Um Gast–Code nativ ausführen zu können muss der virtuelle Adressraum des Gastes auf dem physischen Speicher des Hosts aufgebaut werden. Diese Adressumsetzung wird mit der *Shadow Page Table* vorgenommen, welche im rechten Teil von Abbildung 3.2 zu sehen ist. Diese *Shadow Page Table* wird zur Laufzeit nach Bedarf aufgebaut. Am Anfang enthält der virtuelle Adressraum, der mit diesen Tabellen beschrieben wird, lediglich den reservierten Bereich vom *monitor*. Sobald Gast–Code zur Ausführung kommen soll, löst dieser Code in dem leeren Adressraum Seitenfehler aus. So führt zum Beispiel der erste Eintritt zu einem sofortigen Seitenfehler am Befehlszeiger (IP).

Durch die veränderte IDT werden diese Fehler dem *monitor* zugestellt und von *monitor*–Code behandelt. Für die Behandlung von Seitenfehlern emuliert der *monitor* eine *Memory Management Unit* (MMU). Das heißt, die Seitentabellen des Gastes werden vom *monitor* durchlaufen, um die virtuellen Adressen des Gastes auf Rahmennummern des Gastes zu übersetzen. Dabei werden, genau wie bei einer realen MMU, die Attribute *Accessed* und *Dirty* entsprechend angepasst. Wenn dieser Vorgang ein *Mapping* in den Seitentabellen des Gastes findet, dann wird ein entsprechender Eintrag in die *Shadow Page Table* übernommen. Wird kein Eintrag gefunden, reflektiert der *monitor* den Seitenfehler. Der Kernteil schaltet mit dem Rückgabewert `KQEMU_RET_EXCEPTION` zurück zu *QEMU*. Über den Emulator wird der Seitenfehler dem Gast–Betriebssystem

zugestellt, denn in diesem Fall hätte der Gast–Code auch auf einer realen Maschine den Fehler ausgelöst.

Für das Übertragen von *Mappings* wird die Unterstützung vom *kqemu*–Kernteil benötigt. Die Übersetzung von Gast–Rahmennummer zu virtuellen Adresse im *QEMU*–Adressraum ist trivial. Da der physische Speicher des Gastes zusammenhängend eingeblendet ist, besteht diese Umsetzung lediglich aus der Addition mit der Basisadresse dieses Bereichs. Die Übersetzung in eine Host–Rahmennummer und das *Pinning* werden dann mit Hilfe des Kernteils erledigt. Der *monitor* verwendet dafür die Funktion `MON_REQ_LOCK_USER_PAGE`. Anschließend wird das *Mapping* mit einem Eintrag in die *Shadow Page Table* etabliert und die Ausführung des Gast–Codes fortgesetzt.

Da der *monitor* die Gast–Seitentabellen für diesen Vorgang lesen und verändert muss, müssen sie ebenfalls in seinen Adressraum eingeblendet sein. Sie werden nach Bedarf vom Kernteil angefordert, mit *Pinning* vor Auslagerung geschützt und anschließend in dem reservierten Bereich sichtbar gemacht. Für die Konstruktion der *Shadow Page Table* verwendet der *monitor* die `MON_REQ_ALLOC_PAGE`–Funktion um nach Bedarf Seiten für diese Datenstruktur zu allozieren.

Der *QEMU*–Prozess und der *monitor* haben gemeinsamen Speicher. In diesem Speicher sind verschiedene Datenstrukturen abgelegt, die zur Kommunikation zwischen den beiden verwendet werden. Über diese Datenstrukturen werden Schreibzugriffe in einem Kontext an den jeweils anderen signalisiert. Diese Informationen werden zum Beispiel benötigt, damit *QEMU* bereits übersetzten Gast–Code verwirft und nochmals übersetzt wenn der jeweilige Speicher im *monitor*–Kontext beschrieben wurde. Außerdem werden sie verwendet, um *Dirty–Flags*, also Attribute von Seitentabellen–Einträgen zu synchronisieren. Der gemeinsame Speicher ist im *monitor*–Adressraum in den reservierten 32 Megabyte eingeblendet.

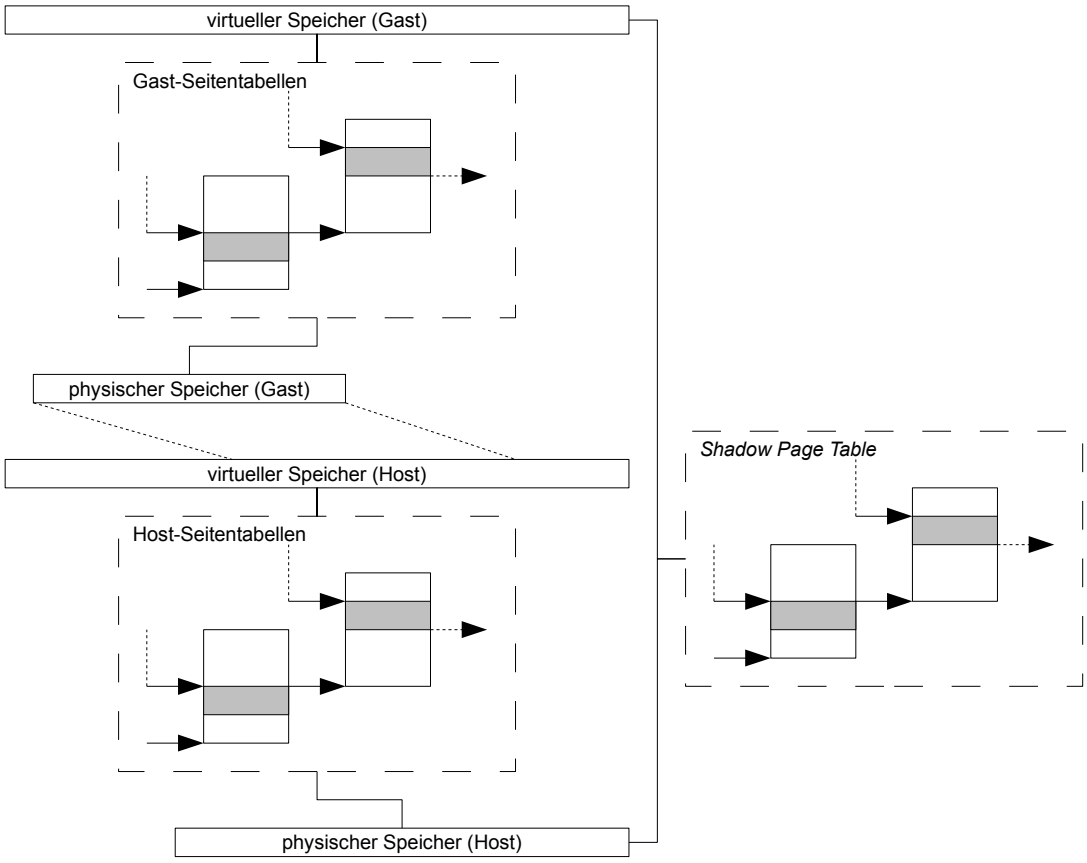

Abbildung 3.2: Adressübersetzung mit drei verschiedenen Sätzen von Seitentabellen

4 Entwurf

Die hier vorgestellten Entwurfsvarianten beruhen alle auf der grundlegenden Entwurfsentscheidung, *kqemu* für die Implementierung einer Virtualisierungslösung für *Fiasco* zu verwenden. Diese Entscheidung wurden getroffen, um den Realisierungsaufwand möglichst gering zu halten und damit den gegebenen Zeitrahmen einer Diplomarbeit einzuhalten.

Die Weiterverwendung einer bestehenden Virtualisierungslösung für Linux ist naheliegend, da Linux–Anwendungen mit Hilfe von *L⁴Linux* auf *Fiasco* eingesetzt werden können. Mit *VirtualBox* und *QEMU* mit *kqemu* stehen als Ausgangspunkt zwei freie Lösungen für Linux zur Verfügung. *QEMU* mit *kqemu* wendet im Gegensatz zu *VirtualBox* im normalen Betriebsmodus keine Ringkompression an. Das heißt, es wird nur Ring 3 Gast–Code nativ, und der restliche Code im Emulator ausgeführt. In diesem Modus sind die Virtualisierungslücken der x86–Architektur nicht relevant, was die Komplexität des VMMs drastisch reduziert. Damit ist *kqemu* eine gute Basis für die Implementierung eines zunächst einfachen VMM für *Fiasco*. Da *kqemu* ebenfalls Ringkompression unterstützt, stellt diese Wahl auch keine Einschränkung für Weiterentwicklungen dar.

Für die Portierung von *kqemu* auf den *Fiasco*–Mikrokern gibt es grundsätzlich zwei Möglichkeiten. Die erste Möglich besteht darin, eine Kernerweiterung für den Mikrokern zu entwickeln. Die Alternative dazu ist die Verwendung von Mechanismen, die der Mikrokern bereits anbietet. *Fiasco* unterstützt mit *Alien Threads* bereits Virtualisierung. Dieser Mechanismus wird zum Beispiel von *L⁴Linux* verwendet und wurde auch im Hinblick auf *L⁴Linux* entwickelt.

4.1 *Fiasco*–Kernerweiterung

Abbildung 4.1 zeigt die Anwendung der ursprünglichen *kqemu*–Architektur, die in 3.2 beschrieben ist, auf den Mikrokern *Fiasco*. Neben dem Kern gibt es den *monitor*, der ebenfalls mit allen Privilegien, also in Ring 0, läuft.

Der Mikrokern enthält bei diesem Entwurf eine Erweiterung. Diese Erweiterung muss dieselben Aufgaben erfüllen wie der Kernteil vom herkömmlichen *kqemu*.

Im *L⁴Linux*–Kern ist ebenfalls eine Erweiterung zu sehen. Dieses Kernmodul dient als Abstraktionsschicht zwischen *QEMU* und der Erweiterung in *Fiasco*. Mit dieser Erweiterung für *L⁴Linux* muss *QEMU* nicht angepasst werden. Die Übersetzung der *kqemu*–Schnittstelle auf die entsprechende *Fiasco*–Erweiterung würde in diesem Teil stattfinden. Eine Anpassung von *QEMU* auf eine neue Schnittstelle ist nicht sinnvoll. Mit einer Nachbildung der *kqemu*–Schnittstelle erreicht man mit geringem Aufwand Kompatibilität mit der Standardversion von *QEMU*.

Der beschriebene Entwurf hat den Vorteil, dass der *monitor* nicht verändert werden muss. Wenn die *Fiasco*–Erweiterung die in 3.3 beschriebene Schnittstelle für den *monitor*

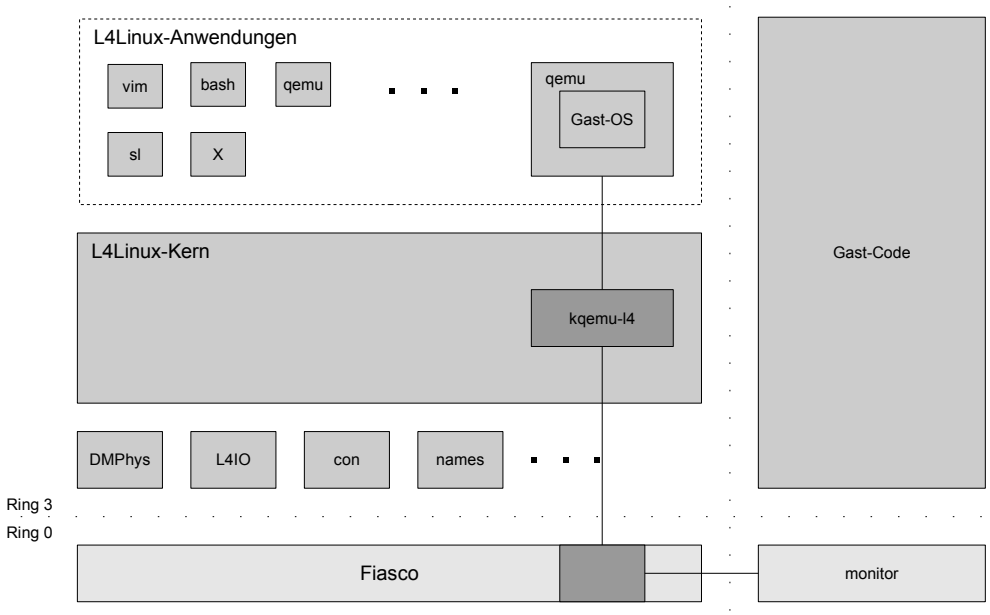

Abbildung 4.1: Entwurf mit Kernerweiterung

anbietet, kann er unverändert eingesetzt werden. Der *monitor* selbst braucht nur wenig Platz im virtuellen Adressraum und der GDT. Es gelten die gleichen Einschränkungen wie beim herkömmlichen *kqemu*. Dadurch muss bei diesem Entwurf nicht mehr emuliert werden, als bei der Verwendung von *kqemu* auf Linux oder Windows.

Im Gegensatz zur Implementierung mit Hilfe von *Alien Threads* bedeutet dieser Ansatz aber wesentlich mehr Aufwand. Es müssen Kernerweiterungen für L^4Linux und *Fiasco* implementiert werden. Die Kernerweiterung von L^4Linux ist nicht kompliziert, dafür können Teile des Programmcodes von *kqemu* verwendet werden. Die komplexen Aspekte, die Aufgaben den Kernteils von *kqemu* betreffen, müssen in die *Fiasco*–Erweiterung eingebaut werden.

4.2 *Alien Threads*

Alien Threads sind ein Mechanismus von *Fiasco*, der zur Virtualisierung verwendet werden kann. Im Gegensatz zu regulären *Fiasco–Threads* werden bei *Alien Threads* jegliche *Exceptions*, die vom Code eines solchen *Threads* ausgelöst werden, nicht vom Kern behandelt. Stattdessen werden sie vom Kern an den *Pager* der entsprechenden *Task* reflektiert. Mit *Alien Threads* stellt *Fiasco* wichtige Grundlagen für Virtualisierung mittels *Trap–and–Emulate* zur Verfügung.

In Abbildung 4.2 auf der nächsten Seite ist ein Entwurf unter der Verwendung von *Alien Threads* dargestellt.

Bei diesem Entwurf ist lediglich eine Erweiterung für L^4Linux vorgesehen. Diese Erwei-

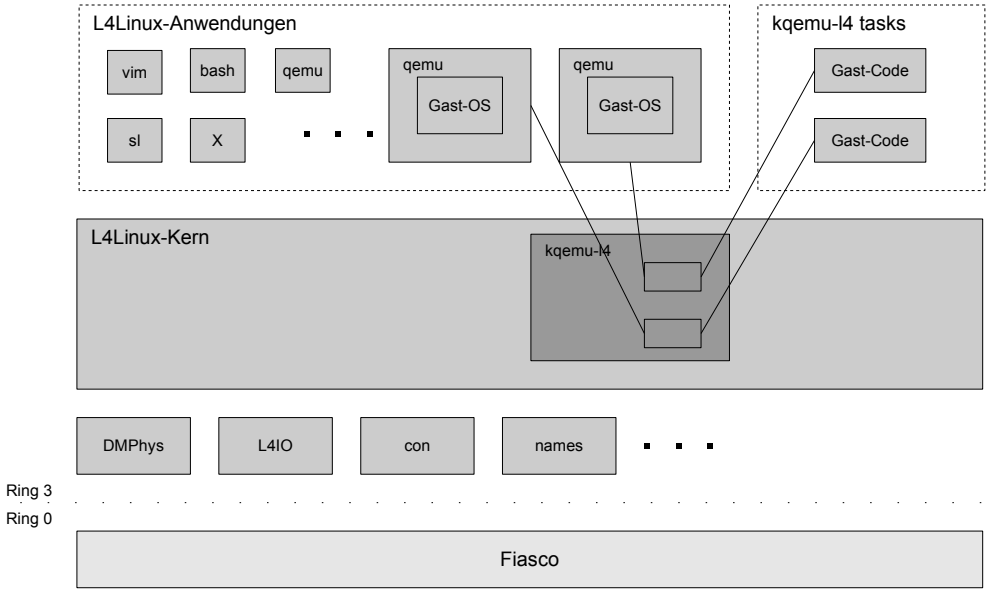

Abbildung 4.2: Entwurf mit *Alien Threads*

terung nutzt *Alien Threads* als Ausführungsumgebung für den Gast–Code. Dabei wird für jede Instanz von *QEMU*, die *kqemu* verwenden soll, eine *Task* mit einem *Alien Thread* bereitgestellt.

Dieser Entwurf ist aus vielen Gründen einfacher umzusetzen als eine Erweiterung des Mikrokerns *Fiasco*. Bei ihm muss nur eine Erweiterung für *L⁴Linux* implementiert werden. Wenn diese Erweiterung *QEMU* die gewohnte Schnittstelle anbietet, kann *QEMU* unverändert auf *L⁴Linux* eingesetzt werden. In diesem Entwurf sind auch keine Änderungen am *Fiasco*–Mikrokern nötig, weil nur bestehende Mechanismen verwendet werden.

Außerdem wird bei dieser Variante ein Großteil vom herkömmlichen *kqemu*–Code nicht mehr benötigt, seine Funktionalität wird von *Fiasco* erbracht. Der *monitor* ist bei *kqemu* unter anderem für die Verwaltung der Seitentabellen der Ausführungsumgebung verantwortlich. Desweiteren stellt er eine Umgebung bereit, in der alle Unterbrechungen abgefangen werden können, um von *kqemu* behandelt zu werden. Diese Aufgaben erfüllt bei diesem Entwurf *Fiasco*. Die Speicherverwaltung, also das *Mapping* von Speicher, kann mit den Mechanismen von *Fiasco* erfolgen. Damit ist bei diesen Entwurf der *monitor* nicht mehr nötig.

Die Verwendung von *Alien Threads* bringt allerdings auch einige Einschränkungen mit sich. Bei *Alien Threads* können nur die unteren 3 Gigabyte des virtuellen Adressraums für Nutzercode verwendet werden. Das obere Gigabyte ist vom Mikrokern reserviert. Außerdem können mit der Verwendung von *Alien Threads* nicht alle Aspekte von segmentiertem Speicher abgedeckt werden. Der Grund dafür ist, dass *Fiasco* die Beschreibungstabellen für Speichersegmente (GDT und LDT) und die Segmentselekto-

ren kontrolliert. Diese Systemdatenstrukturen und Register können für *Alien Threads* in einem gewissen Rahmen vom *Pager* kontrolliert werden. Die Kontrolle ist in der aktuellen Implementierung allerdings weniger flexibel als bei der Weiterverwendung des *monitor*s.

4.3 Bewertung

Die beiden Möglichkeiten sind im Hinblick auf Realisierunsaufwand und Funktionalität sehr unterschiedlich.

Die Implementierung von *kqemu–l4* als Erweiterung des Mikrokerns *Fiasco* ist wesentlich aufwendiger als eine Umsetzung mit Hilfe von *Alien Threads*. Auch wenn große Teile des vorhandenen Codes übernommen werden können, bleibt die höhere Komplexität und der damit verbundene hohe Implementierungsaufwand. Ziel der Arbeit ist es, eine sichere Virtualisierungslösung für den *Fiasco*–Mikrokern zu entwickeln. Der Mikrokern *Fiasco* ist die sicherheitskritische Komponente beider Entwürfe. Der Kern ist Teil der TCB. Änderungen an derart kritischen Komponenten können unerwünschte Nebeneffekte haben und die Sicherheit des Gesamtsystems gefährden. Um solche Nebeneffekte nicht zu übersehen, erfordern Anpassungen an der TCB einen erhöhten Einarbeitungsaufwand.

Bei der Umsetzung mit *Alien Threads* kann nicht so viel Funktionalität und damit Code von *kqemu* wiederverwendet werden. Aber in dem Fall ist es auch nicht nötig. *Fiasco* stellt einen Großen Teil der benötigten Funktionen bereits zur Verfügung. Hier ist keine Anpassung am Mikrokern nötig und es läuft im Gegensatz zum Entwurf mit Kernerweiterung auch kein zusätzlicher Code in Ring 0. Damit bleibt die TCB einer herkömmlichen *L⁴Linux*–Installation erhalten.

Die Verwendung von *Alien Threads* bringt zusätzliche Einschränkungen mit sich. Der für den Gast–Code verfügbare virtuelle Adressraum ist eingeschränkt. Auch die eingeschränkte Kontrolle über Systemdatenstrukturen, wie GDT und Segmentselektoren, kann bei diesem Entwurf einen schlechten Einfluss auf die Leistung der virtuellen Maschine haben. Die Einschränkungen führen dazu, dass potenziell mehr Gast–Code in *QEMU* ausgeführt werden muss.

Wie problematisch diese Einschränken tatsächlich sind, ist schwer vorauszusagen. Auch wenn der Entwurf zunächst keine Anpassungen an *Fiasco* vorsieht, sind sie dennoch möglich. Mit entsprechenden Anpassungen am Mikrokern lassen sich besonders kritische Einschränken aufheben oder lockern.

Ich habe mich für die Implementierung mit Hilfe von *Alien Threads* entschieden. Diese Entscheidung war vor allem durch den geringeren Implementierungsaufwand motiviert.

5 Implementierung

Im Laufe der Arbeit ist eine *L⁴Linux*–Kernerweiterung nach dem beschriebenen Entwurf entstanden. Mit dem Modul ist es möglich, eine Reihe von Gast–Betriebssystemen mit Virtualisierungs–Beschleunigung durch *Native Execution* auf *L⁴Linux* und dem Mikrokern *Fiasco* zu verwenden.

Ein Hauptziel der Implementierung war es, die Umgebung so wenig wie möglich anzupassen. Im Idealfall hieße das, nur ein Modul für *L⁴Linux* zu entwickeln und alle anderen Komponenten nicht zu verändern. Das verspricht einen geringen Implementierungs– und Wartungs–Aufwand.

Dieses Ziel konnte nicht komplett erreicht werden. Insbesondere *Fiasco* musste an einigen Stellen angepasst werden, um *kqemu–l4* zu unterstützen. An *L⁴Linux* und *QEMU* mussten ebenfalls einige wenige Veränderungen vorgenommen werden.

5.1 Vorgehensweise

Das *L⁴Linux*–Kernmodul *kqemu–l4* habe ich komplett neu entwickelt. Ich habe also zunächst ein Code–Gerüst für ein Linux Kernmodul geschrieben und dieses Schritt für Schritt erweitert. Die Alternative wäre gewesen, das bestehende Modul für Linux als Ausgangspunkt für die Entwicklung zu verwenden.

Bei dieser Erweiterung habe ich schließlich auch wieder Quellcode aus *kqemu* übernommen. Das betrifft vor allem:

- die Implementierung der Schnittstelle zu *QEMU*
- Code zur Initialisierung des Moduls
- die Verwaltung von Gast–Kontexten
- einige Datenstrukturen und Algorithmen

Außerdem orientiert sich die Struktur von *kqemu–l4* sehr stark an der von *kqemu*. Das betrifft vor allem Funktionsnamen und –parameter und den Kontrollfluss zwischen den einzelnen Funktionen. Die Implementierungen der einzelnen Funktionen weichen allerdings teilweise stark vom Original ab, was auf die grundlegend andere Architektur zurückzuführen ist.

Für Code, der Mechanismen des *Fiasco*–Mikrokerns verwendet, habe ich stets versucht die entsprechenden Funktionen von *L⁴Linux* zu verwenden. An Stellen wo das nicht möglich war habe ich mich an den entsprechenden Funktionen im *L⁴Linux*–Kern orientiert.

Für die Erzeugung und das Zerstören von *Tasks* greift *kqemu–l4* teilweise direkt auf Funktionen des *L⁴Linux*–Kerns zurück. Die Ausführungsschleife von *kqemu–l4* ist der von *L⁴Linux* nachempfunden.

5.2 Werkzeuge

`kqemutest`

Das Quellarchiv von *kqemu* enthält ein Testwerkzeug (`kqemutest`). Diese Anwendung verwendet die Schnittstelle, die im Normalfall vom *QEMU*–Prozess verwendet wird, um Test–Code mit Hilfe von *kqemu* nativ auf dem Host auszuführen. Sie stellt alle von *kqemu* benötigten Systemdatenstrukturen und einen Satz Seitentabellen zur Verfügung.

Für den Test–Code gibt es eine sehr eingeschränkte C–Bibliothek. Mit ihr lässt sich virtueller Speicher allozieren. Außerdem bietet sie einige wenige Systemaufrufe, über die der Test–Code mit dem Testwerkzeug interagieren kann.

Diese Testumgebung habe ich nicht nur für die anfängliche Analyse der Funktionsweise von *kqemu* verwendet, das Testwerkzeug war auch für die Entwicklung von *kqemu–l4* sehr hilfreich. Die Testumgebung bot bereits fertige Funktionen zum Testen der Speicherverwaltung. Sie beinhaltet Tests, die virtuellen Speicher zunächst allozieren, ihn lesen und später beschreiben. Auch Funktionen wie das mehrfache Einblenden von Rahmen im virtuellen Adressraum, konnte ich mit dem mitgelieferten Code testen.

Gerade in der ersten Phase der Implementierung habe ich ausschliesslich dieses Werkzeug für meine Tests verwendet. Dafür habe ich es stellenweise angepasst und dabei zum Beispiel die Schnittstelle für Systemaufrufe erweitert. Damit ist es jetzt möglich mit Test–Code gezielt in den Kern–*Debugger* vom *Fiasco*–Mikrokern zu schalten. Bei der Fehlersuche während der Entwicklung habe ich `kqemutest` ebenfalls regelmäßig eingesetzt und teilweise angepasst.

QEMU

Nachdem die Speicherverwaltung und einige andere grundlegende Mechanismen fertig implementiert waren, habe ich angefangen erste Tests mit *QEMU* durchzuführen. Dafür habe ich eine Version von *QEMU* genutzt, mit der es mir möglich war, sehr viele Informationen über den aktuellen Zustand des Gast–Systems zu untersuchen. Diese Erweiterung von *QEMU* erlaubt es zum Beispiel, einzelne Speicherzugriffe, Zustandsübergänge der virtuellen CPU oder ausgewählte Instruktionen des Gast–Codes in Log–Dateien zu protokollieren. Diese angepasste Version von *QEMU* wurde von meinem Betreuer Michael Peter an der Technischen Universität Dresden entwickelt. Zusammen mit einigen wenigen Anpassung meinerseits, war diese spezielle Version von *QEMU* ebenfalls ein sehr nützliches Werkzeug. Das Programm hat mir dabei geholfen die Schnittstelle zwischen *QEMU* und *kqemu* zu analysieren. Mit dieser Version von *QEMU* habe ich die ersten Tests von *kqemu–l4* mit tatsächlichen Gast–Betriebssystemen durchgeführt.

Nachdem *Fiasco* als erstes Gast–Betriebssystem zusammen mit *kqemu–l4* funktioniert hat, waren die Möglichkeiten zur Protokollierung für die weitere Entwicklung nicht mehr

nötig. In der Folge habe ich die jeweils aktuelle *QEMU*–Version aus der Versionskontrolle des Projektes verwendet.

5.3 Anpassungen an *QEMU*

Während der Entwicklung von *kqemu–l4* sind eine Reihe kleinerer Probleme aufgetreten, die ich mit Anpassungen an der Code–Basis von *QEMU* gelöst habe.

kqemu–l4 ist weniger generell als *kqemu*. Es gibt Situationen in denen *kqemu–l4* auf Emulation durch *QEMU* zurückgreifen muss, wo *kqemu Native Execution* anwenden kann. Das ergibt sich aus Einschränkungen, die in der Architektur von *kqemu–l4* begründet sind. Welche Unterschiede es gibt und wie sie die Leistung von *kqemu–l4* im Vergleich zu *kqemu* beeinflussen, ist in Kapitel 5.6 näher beschrieben.

An *QEMU* habe ich zwei Änderungen vorgenommen. Beide Anpassungen behandeln das Problem, dass *kqemu–l4* nicht immer *Native Execution* nutzen kann wenn *kqemu* es könnte. In *QEMU* werden mit der Funktion `kqemu_is_ok` eine Reihe von Tests durchgeführt, um zu überprüfen, ob *kqemu* für die Ausführung verwendet werden kann. Ich habe in *kqemu–l4* eine Reihe zusätzlicher Tests eingebaut, um regelmässig zu prüfen ob *Native Execution* möglich ist oder, ob die Kontrolle wieder an *QEMU* übergeben werden sollte. Im Vergleich zu *kqemu* gibt es in *kqemu–l4* mehr Situationen, in denen keine native Ausführung des Gast–Codes möglich ist. Die Machbarkeitstest in *QEMU* sind allerdings auf *kqemu* zugeschnitten. Deshalb kann es in einigen Fällen zu einem ständigen Hin– und Herschalten zwischen *QEMU* und *kqemu–l4* kommen ohne das dabei Gast–Code in *kqemu–l4* ausgeführt wird. Dieser Pingpong–Effekt wirkt sich negativ auf die Leistung der virtuellen Maschine aus, anstatt eine Beschleunigung zu erreichen wird die Ausführung verlangsamt.

Für die Lösung dieses Problems habe ich *QEMU* so angepasst, dass man die Nutzung von *kqemu* zur Laufzeit durch die Zustellung eines Signals ein– oder ausschalten kann. Desweiteren habe ich einen Mechanismus implementiert, mit dem *kqemu–l4* signalisieren kann, dass *Native Execution* zum jeweiligen Zeitpunkt nicht möglich ist. Dabei wird in einem ungenutzen Feld einer gemeinsamen Datenstruktur eine Zahl übergeben. Mit dieser Zahl wird *QEMU* signalisiert, wie oft das Umschalten zu *kqemu–l4* übersprungen werden soll. Bei jeder Überprüfung, ob die Ausführung mit *kqemu* möglich ist, wird diese Zahl dekrementiert bis sie wieder gleich 0 ist. Damit habe ich den Machbarkeitstest in *QEMU* nur minimal verändern müssen. Gleichzeitig habe ich die Möglichkeit, die Tests aus *kqemu–l4* weiter zu verwenden, ohne sie in *QEMU* nochmals implementieren zu müssen. Dieser Mechanismus war vor allem während der anfänglichen Entwicklung wertvoll. Anfangs befanden sich die Tests in *kqemu–l4* noch in der Entwicklung und wurden häufig angepasst.

Beide Anpassungen sind als Entwicklungsversionen zu sehen. Für den tatsächlichen Einsatz von *kqemu–l4* sollten an ihrer Stelle zusätzliche Tests in *QEMU* eingebaut werden. Mit Anpassungen an der Funktion `kqemu_is_ok` kann der Pingpong–Effekt verhindert werden.

Aber gerade der erste Mechanismus hat sich für die Entwicklungsarbeit bewährt. Während der Entwicklung war noch unklar, welche Tests in *kqemu–l4* nötig wären. Sie

wurden Stück für Stück implementiert und der Code hat sich mehrfach verändert. Diese Tests ebenfalls in *QEMU* einzupflegen, hätte mehr Entwicklungsaufwand bedeutet.

Auch die zweite Anpassung ist nur für Entwicklungszwecke interessant. Es hat sich gezeigt, dass es schwer ist die Zahl richtig zu bestimmen. Sie hängt stark vom verwendeten Gast–Betriebssystem und dem jeweils fehlgeschlagenen Test ab. Überspringt man zu viele Wechsel zu *kqemu–l4*, läuft die virtuelle Maschine langsamer als sie könnte. Ist die Zahl zu klein, tritt der Pingpong–Effekt weiterhin auf. Allerdings kann man die Frequenz, der Kontextwechsel ohne Fortschritt, reduzieren.

kqemu–l4 kann auch mit einer unangepassten Version von *QEMU* verwendet werden. Die Schnittstelle zwischen *kqemu* und *QEMU* wird immer noch erfüllt. Für die Verwendung von Linux oder Windows auf der virtuellen Maschine ist es zum Beispiel nicht relevant, ob man eine angepasste Version von *QEMU* verwendet. Wenn man zum Beispiel FreeBSD als Gast–System verwenden will, sind die Anpassungen sinnvoll. Der Bootloader von FreeBSD provoziert ein solches Hin– und Herschalten und sollte übersprungen werden.

5.4 Anpassungen an *L⁴Linux*

Für die Implementierung von *kqemu–l4* musste auch *L⁴Linux* geringfügig angepasst werden. Die Anpassungen beschränken sich auf den Export von Symbolen, damit diese von externen Modulen verwendet werden können. Funktional wurde *L⁴Linux* nicht verändert.

Das Modul selbst zähle ich nicht als Anpassung an *L⁴Linux*, hier geht es nur um Anpassungen der Code–Basis vom *L⁴Linux*–Kern. *kqemu–l4* ist ein eigenständiges Projekt.

5.5 *Fiasco*

Fiasco bietet mit *Alien Threads* schon einen sehr guten Mechanismus um Gast–Code nativ auszuführen. Der Mechanismus ist aber für *L⁴Linux* entwickelt worden und damit in der jetzigen Version von *Fiasco* nicht generell, sondern auf die Bedürfnisse von Linux–Anwendungen zugeschnitten.

Bei der Implementierung von *kqemu–l4* sind Probleme aufgetreten, für deren Lösung der Mikrokern *Fiasco* angepasst werden musste. In der Folge werde ich näher auf diese Problem und meine Lösungen eingehen.

5.5.1 Artefakte im Adressraum

Adressräume in *Fiasco* sind aus Nutzersicht nicht komplett leer, auch wenn vom *Pager* noch keine *Mappings* angelegt wurden. Das heißt, es gibt Bereiche, die aus Ring 3 ohne *Page Fault* zugreifbar sind. Für den normalen Betrieb von *Fiasco*–Applikationen sind diese Bereiche auch notwendig. Virtualisierung ist mit diesen Artefakten im Adressraum allerdings nicht zuverlässig möglich. Der virtuelle Adressraum des Gastes muss komplett

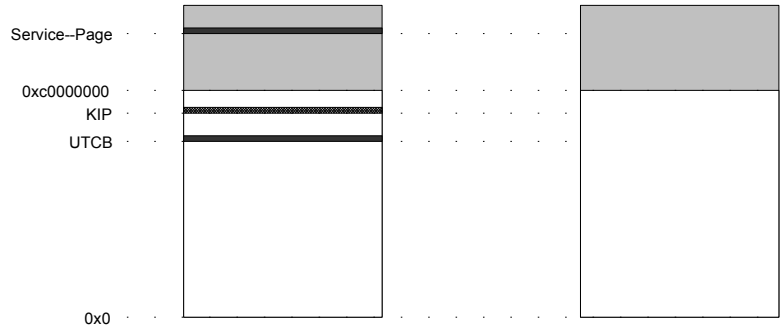

Abbildung 5.1: Adressraum: links mit, rechts ohne Artefakte

unter der Kontrolle von *kqemu–l4* sein. Das heißt es dürfen dort keine Seiten zugreifbar sein, die nicht von *kqemu–l4* dort eingeblendet wurden. In diesem Fall würde der Gast an den betroffenen Stellen den Speicher vom Host vorfinden und würde nicht wie erwartet funktionieren. Das würde eine Verletzung der Forderungen nach Isolation und Äquivalenz bedeuten.

Adressräume in *Fiasco* können drei solche Artefakte haben. Zwei davon, der *UTCB* und die *Service–Page*, sind standardmäßig in jedem Adressraum eingeblendet und aus Ring 3 zugreifbar. Das Einblenden der *Kernel–Info–Page* (KIP), dem dritten Artefakt, wird durch eine spezielle Instruktionsfolge ausgelöst. Wenn Gast–Code diese Instruktionsfolge enthält, zum Beispiel weil man *Fiasco* als Gast–Betriebssystem verwendet, würde die KIP des Hosts als Artefakt in dem Adressraum sichtbar werden.

Ich habe das Problem dieser Artefakte gelöst, indem ich den *Fiasco*–Kernobjekten *Thread*, *Task* und *Memspace* ein neues Attribut hinzugefügt habe. Dieses Attribut wird bei der Initialisierung der *kqemu–l4–Task* gesetzt. Der angepasste *Fiasco*–Kern sorgt dafür, dass keine Artefakte mehr zugreifbar sind und die KIP nicht eingeblendet wird.

Abbildung 5.1 zeigt die Änderung. Auf der linken Seite sind die drei Artefakte als dunkelgraue Balken im virtuellen Adressraum dargestellt.

5.5.2 SYSENTER

Der Maschinen–Befehl SYSENTER von x86–Prozessoren kann zum Kerneintritt verwendet werden. Die Instruktion ist eine schnellere Alternative zur Verwendung von *Software–Interrupts* für Kerneintritte. Für Virtualisierung stellt der Befehl ein Problem dar, weil *Fiasco* SYSENTER intern auf einen anderen Befehl, einen *Software–Interrupt*, abbildet. Im *Pager* eines *Alien Threads* kommt zwar eine Nachricht (*exception IPC*) an, diese lässt sich aber nicht mehr zweifelsfrei der SYSENTER–Instruktion zuordnen.

Alien Threads wurden bisher in erster Linie für *L⁴Linux*–Anwendungen eingesetzt. Linux–Applikationen enthalten keinen Code für Kerneintritte, sie verwenden dafür eine Bibliothek. Diese verwendet Code, der vom Linux–Kern beim Systemstart vorbereitet wird. Ob dieser Code SYSENTER oder *Software–Interrupts* nutzt, hängt von den Fähigkeiten der jeweiligen CPU ab. Da nicht alle x86–Prozessoren den SYSENTER–

Befehl unterstützen, ist diese Unterscheidung nötig. *L⁴Linux* trifft diese Unterscheidung allerdings nicht. Der Code, den *L⁴Linux* für Kerneintritte bereitstellt, enthält keine SYSENTER–Instruktionen. Aus diesem Grund tritt dieser Befehl in Verbindung mit *L⁴Linux* in der Praxis nicht auf.

In *kqemu–l4* können wir nicht mehr davon ausgehen, dass der SYSENTER–Befehl nicht im Gast–Code enthalten ist. Viele Betriebssysteme verwenden diesen Befehl, um schnell vom Nutzer– zum Kern–Modus umzuschalten. Da die Gast–Systeme unmodifiziert genutzt werden sollen, muss SYSENTER behandelt werden und kann nicht mehr ignoriert werden.

Die implementierte Lösung besteht darin, dafür zu sorgen, dass SYSENTER in *Alien Threads* immer fehlschlägt. Das wird erreicht, indem das Kontrollregister `IA32_SYSENTER_CS` immer den Wert Null enthält, wenn *kqemu–l4–Threads* laufen. Immer wenn in *Fiasco* ein *Thread*–Wechsel stattfindet, wird der Inhalt des Registers entsprechend angepasst. Beim Wechsel zurück zu einem herkömmlichen *Fiasco–Thread* wird der ursprüngliche Wert des Registers wieder hergestellt. Mit dieser Anpassung am Mikrokern kann die SYSENTER–Instruktion mit dem klassischen *Trap–and–Emulate* sicher erkannt und behandelt werden. Der Versuch SYSENTER auszuführen löst einen *General Protection Fault* aus. Dieser Fehler wird vom *Fiasco*–Mikrokern mittels *exception IPC* an *kqemu–l4* übermittelt, wo die SYSENTER–Instruktion als Ursache des Fehlers ermittelt werden kann.

Eine andere Möglichkeit, das SYSENTER–Problem von *exception IPC* zu umgehen, wäre es, SYSENTER in *Fiasco* nicht zu verwenden und den Registerinhalt somit nicht bei jedem *Thread*–Wechsel anpassen zu müssen. Das Kontrollregister könnte dauerhaft den Wert Null tragen und SYSENTER somit dauerhaft abgeschaltet bleiben. Das häufige Beschreiben eines Kontrollregisters hat potentiell einen schlechten Einfluss auf die Dauer von Kontextwechseln und damit die Leistung des Gesamtsystems. Messungen haben gezeigt (Kapitel 6.1.3), dass das häufige Umschalten des Zustandsregisters einen negativen Einfluss auf die Leistung von *kqemu–l4* hat.

Eine Erweiterung vom *exception IPC*–Mechanismus um die Signalisierung von SYSENTER könnte das Problem lösen, anstatt es zu umgehen. Der SYSENTER–Befehl sollte nicht auf einen *Software–Interrupt* abgebildet werden, sondern als Sonderfall behandeln werden. Die implementierte Umgehung des eigentlichen Problems, nämlich dass der SYSENTER–Sonderfall nicht zuverlässig erkannt werden kann, war allerdings einfacher zu implementieren als eine Lösung dieses Problems. Deshalb habe ich mich zunächst dafür entschieden, das Problem durch das Abschalten von SYSENTER im Gast–Kontext zu umgehen.

5.5.3 Segmentierung/GDT

Mit der Segmentierung von virtuellem Speicher sind einige Probleme für *kqemu–l4* verbunden. Die Systemdatenstrukturen, also die GDT und LDT, und die Segmentselektoren, werden vom *Fiasco*–Mikrokern verwaltet. Es gibt aber Fälle, in denen die Selektoren und Tabellen für die native Ausführung des Gast–Codes angepasst werden müssen.

Pager können in *Fiasco* zwei der sechs Segmentselektoren, Teile der GDT und die LDT ihrer *Tasks* manipulieren. Diese Funktionalität wurde in *Fiasco* integriert, um *L⁴Linux*

zu unterstützen. Der Linux–Kern verwendet Segmentierung nur sehr eingeschränkt. Die Segmente für Code und Daten überspannen den kompletten virtuellen Adressraum und implementieren keine zusätzlichen Schutzmechanismen. Seit der Version 2.6 unterstützt der Linux–Kern *Thread Local Storage* (TLS). Dafür wird die Segmentierung von virtuellem Speicher verwendet.

Die Möglichkeit, die beiden Segmentselektoren FS und GS und drei ausgewählte Einträge der GDT zu kontrollieren, ist nötig um *L⁴Linux*–2.6 zu unterstützen. Die Kontrolle der LDT wurde ebenfalls für spezielle Linux–Anwendungen implementiert.

Mit *kqemu–l4* kamen allerdings neue Anforderungen. Es reichte nicht mehr aus, nur drei GDT–Einträge kontrollieren zu können. Für Gast–Systeme wie *Fiasco* und Linux–2.4, die Segmentierung kaum verwenden, stellt sie kein Problem dar. Aktuelle Versionen von Linux (Linux–2.6) und andere Betriebssysteme wie FreeBSD verwenden Segmentierung in größerem Umfang. Das heißt bei solchen Gast–Betriebssystemen muss damit gerechnet werden, dass Gast–Code Segmentselektoren liest und verändert. Damit sich Gast–Code dieser Betriebssysteme erwartungsgemäss verhält, war es nötig, *Fiasco* so anzupassen, dass mehr GDT–Einträge vom *Pager* kontrolliert werden können.

Die letztendlich entstandene Lösung stellt die ersten 63 Einträge der GDT für individuelle Nutzung zur Verfügung. Alle vom Mikrokern selbst benötigten Segment–Deskriptoren sind entsprechend weiter oben in der GDT abgelegt. Abbildung 5.2 stellt die Veränderungen an der GDT von *Fiasco* dar. Während der Entwicklung von *kqemu–l4* gab es mehrere Varianten dieser Anpassung an *Fiasco*. In einer der ersten Versionen habe ich zunächst nur die drei kontrollierbaren Deskriptoren so verschoben, dass Linux–2.6 als Gast–System verwendet werden konnte. Die letztendlich entstandene Lösung basiert darauf, dass keines der Gast–Systeme, die ich getestet habe, GDT–Einträge oberhalb von 64 benötigt hat. Die internen Deskriptoren von *Fiasco* wurden verschoben, um Kollisionen mit Gast–Deskriptoren zu vermeiden.

5.6 Vergleich zu *kqemu*

5.6.1 Vereinfachungen

Im Gegensatz zu *kqemu* ist *kqemu–l4* weniger kompliziert. Durch die Portierung auf den Mikrokern *Fiasco* sind komplexe Teile des Codes überflüssig geworden, ihre Aufgaben werden in *kqemu–l4* von *Fiasco* erledigt. Vereinfachungen haben sich vor allem bei der Speicherverwaltung ergeben. Beim Auftreten eines *Page Fault* müssen zunächst die Seitentabellen des Gast–Systems durchlaufen werden. Anhand der Einträge in diesen Tabellen kann festgestellt werden, ob eine Seite in den Gast–Adressraum eingeblendet werden soll oder ob der Fehler an das Gast–System reflektiert werden muss.

In *kqemu* werden *Page Faults* vom *monitor* behandelt. Damit ist es in *kqemu* unter bestimmten Umständen möglich, diese häufig auftretende Fehlerbehandlung ohne Kontextwechsel durchzuführen. Neben der Behandlung von Seitenfehlern hat der *monitor* von *kqemu* noch andere Aufgaben, für die er auf Datenstrukturen von *QEMU* zugreifen muss. Dieser Zugriff ist aus dem *monitor* aber immer nur dann möglich wenn die betreffenden Seiten schon in den Adressraum eingeblendet sind. Für alle anderen Zugriffe in den Adressraum von *QEMU* müssen die betreffenden Seiten entweder in den *monitor*–

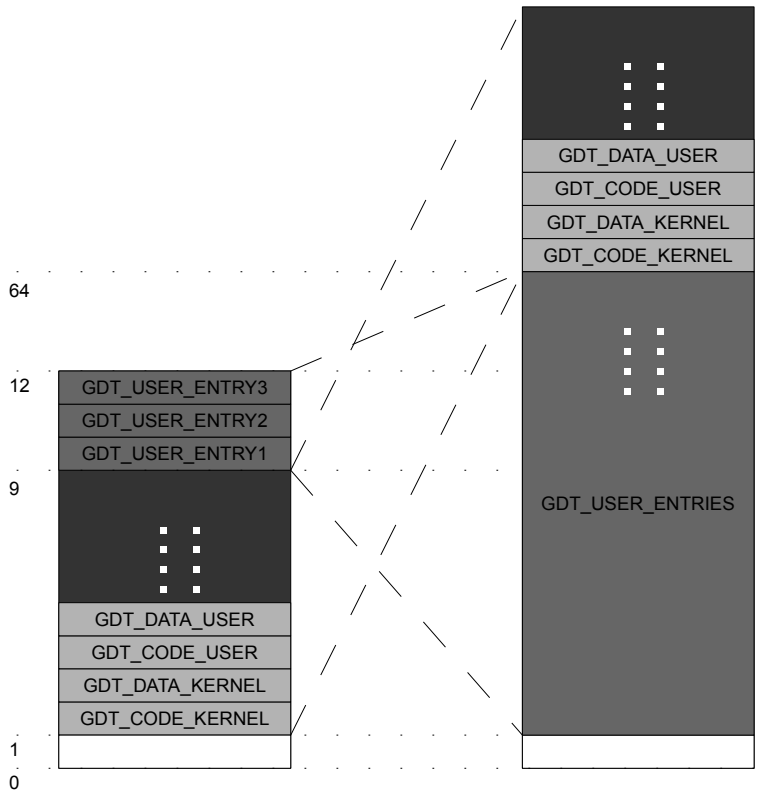

Abbildung 5.2: GDT–Belegung: links original *Fiasco*, rechts angepasster Mikrokern

Adressraum eingeblendet werden oder die Zugriffe aus dem Host–Kern heraus erfolgen. In jedem Fall sind aber Kontextwechsel zwischen *monitor* und Host–Kern nötig.

In *kqemu–l4* gibt es keinen *monitor* mehr. Alle seine Aufgaben wurden in *L⁴Linux* verlagert. Daraus ergibt sich, dass der Zugriff auf gemeinsame Datenstrukturen von *QEMU* und *kqemu–l4* immer mit den Linux–Funktionen `copy_to_user` und `copy_from_user` erfolgen kann. Für das Modul ist es nicht relevant, an welcher Stelle die Daten im physischen Speicher des Hosts liegen, da die Seiten nicht mehr in einen anderen anderen Adressraum eingeblendet werden müssen. Ein Großteil der Adressübersetzungen ist damit nicht mehr notwendig oder wird von *L⁴Linux* vorgenommen.

Die Verwaltung der *Shadow Page Table* wird von *kqemu* direkt erledigt. Der *monitor* hat Zugriff auf die Seitentabellen und manipuliert sie entsprechend. Es werden also keine Funktionen des Host–Kerns weiterverwendet. Diese Aufgabe ist komplex und mit erheblichem Aufwand verbunden. Bei *kqemu–l4* hingegen wird die *Shadow Page Table* mit Host–Mechanismen konstruiert. *Mappings* werden mit *Flexpage* IPC etabliert und können mit dem Systemaufruf `l4_fpage_unmap` entfernt werden. Für das Einblenden von Seiten in den Gast–Adressraum muss in *kqemu–l4* keine Übersetzung in Rahmennummern des Hosts vorgenommen werden. Die für die *Mappings* nötigen Adressübersetzungen werden zum Großteil von *Fiasco* durchgeführt. Dadurch ist *kqemu–l4* wesentlich weniger komplex und aufgrund der Abstraktion auch weniger architekturabhängig.

Das originale *kqemu*–Modul übernimmt immer für die Dauer der Ausführung die Kontrolle über die komplette Maschine. Das bedeutet, dass es alle nötigen Systemdatenstrukturen für x86–Prozessoren und damit eine Umgebung ähnlich einem Betriebssystem zur Verfügung stellen muss. Außerdem enthält es Code, um zwischen dem Gast– und Host–Kontext umzuschalten. Im Gegensatz dazu nutzt *kqemu–l4* Mechanismen von *Fiasco*. Das Erzeugen einer Ausführungsumgebung für Gast–Code entspricht der Erzeugung einer *Task* mit einem *Alien Thread*. Das Umschalten zwischen den Kontexten und die Auslieferung aller *Exceptions* an den VMM sind ebenfalls von *Fiasco* schon angebotene Funktionen, die in *kqemu–l4* nicht implementiert werden mussten. Auch an dieser Stelle konnte in *kqemu–l4* Funktionalität, Code und damit auch Komplexität eingespart werden.

5.6.2 Limitierungen

Durch die Umgebung, in der *kqemu–l4* läuft, ergeben sich eine Reihe von Einschränkungen und damit auch Nachteile gegenüber dem herkömmlichen *kqemu*.

Auf dem Mikrokern *Fiasco* werden *Page Faults* an den *Pager* der jeweiligen *Task* weitergeleitet. Das bedeutet, dass bei jedem *Page Fault* ein Adressraumwechsel, von der jeweiligen Anwendung zu ihrem zugeordneten *Pager*, stattfindet. In *kqemu–l4* haben alle *Exceptions* einen Adressraumwechsel zum *L⁴Linux*–Kern zur Folge. Bei *kqemu* hingegen können einige Außnahmen direkt im *monitor*–Kontext behandelt und damit Adressraumumschaltungen eingespart werden.

Durch die Verwendung von *Alien Threads* stehen nur die unteren drei Gigabyte des virtuellen Adressraums für Gast–Code und Daten zur Verfügung. Das obere Gigabyte wird vom *Fiasco*–Mikrokern benutzt. Im Gegensatz dazu benötigt der *monitor* nur 32 Megabyte, wodurch beim herkömmlichen *kqemu* erheblich mehr virtueller Adressraum

für den Gast nutzbar ist. Die reservierten Bereiche sind aus Ring 3 nicht zugreifbar. *Page Faults* in einem reservierten Bereich führen dazu, dass Code nicht nativ ausgeführt, sondern in *QEMU* emuliert wird. Generell kann eine Einschränkung des verfügbaren Adressraumes also zu mehr Emulation und weniger nativer Ausführung führen. In der Praxis hängt es aber sehr stark vom verwendeten Gast–System ab, wie groß die Leistungsverluste durch die Reservierung einzelner Bereiche des virtuellen Adressraums tatsächlich sind.

Weit verbreitete Betriebssysteme wie Linux und Windows verwenden den oberen Teil des virtuellen Adressraums für die Ausführung ihrer Kerne. Da bei *Alien Threads* das obere Gigabyte nicht zur Verfügung steht, ist die Implementierung der `-kernel-kqemu`–Funktion für *kqemu–l4* nicht sinnvoll und wurde daher auch nicht implementiert. Im Gegensatz zu *kqemu* bietet *kqemu–l4* also keine Ringkompression zur nativen Ausführung von privilegiertem Gast–Code an.

Sowohl *kqemu* als auch *kqemu–l4* verwenden einen Adressraum als Ausführungsumgebung für alle Anwendungen des Gast–Systems. Alle Adressräume, die in der virtuellen Maschine existieren, werden per Zeitmultiplex auf einen virtuellen Adressraum im Host abgebildet. Adressraumwechsel im Gast führen dazu, dass dieser geteilte virtuelle Adressraum komplett geleert wird. Diese *Flushes* gehen im herkömmlichen *kqemu* mit geringen Aufwand, da dort lediglich Seitentabellen des *monitor*s invalidiert werden müssen. Das Entfernen von *Mappings* in *kqemu–l4* erfolgt mit dem *Fiasco*–Systemaufruf `14 _fpage_unmap`. Durch das Konzept der hierarchischen *Pager* muss der Mikrokern bei diesem Aufruf nicht nur den jeweiligen Eintrag in der Seitentabelle löschen. Außerdem müssen Einträge in der *Mapping*–Datenbank des Kerns angepasst werden und der Speicher eventuell auch anderen *Tasks* entzogen werden.

Mit der implementierten Anpassung der GDT erreicht *kqemu–l4* keine Äquivalenz im Bezug auf Segmentierung. Wie schon erwähnt, müsste *kqemu–l4* für vollständige Äquivalenz die GDT, LDT und alle sechs Segmentselektoren kontrollieren können. Da aber die meisten x86–Betriebssysteme Segmentierung nur in einem geringen Maß verwenden, muss sie für deren Unterstützung nur teilweise behandelt werden. In *kqemu–l4* bestehen die folgenden Einschränkungen für die Virtualisierung von segmentiertem Speicher:

- es können nur die zwei Selektoren GS und FS kontrolliert werden
- Code– und Datensegment überspannen immer den kompletten virtuellen Adressraum
- es kann nur ein kleiner Teil der GDT–Einträge angepasst werden
- die LDT ist auf eine Seite virtuellen Speicher begrenzt

Im Gegensatz dazu unterscheidet sich bei *kqemu* nur die GDT von der, die der Gast–Code auf einer realen Maschine benutzen würde.

5.7 Umfang

Der Umfang in Codezeilen wird oft herangezogen, um die Komplexität von Software zu messen. In Tabelle 5.1 ist dargestellt wie viele Codezeilen für die Implementierung von

Komponente	**LOC**
kqemu–l4	~ 2500
Fiasco	~ 50
L^4Linux	~ 10
QEMU	~ 10

Tabelle 5.1: für *kqemu–l4* entstandener Code

kqemu–l4 nötig waren. Außerdem ist dargestellt, wie umfangreich die Anpassungen an den anderen Komponenten im System sind.

Festzuhalten ist hier vor allem, dass die Änderungen an *Fiasco* mit rund 50 Zeilen Code sehr überschaubar sind. Das ist auch der einzige Code, der tatsächlich zur *Trusted Computing Base* gezählt werden muss, da der restliche Code ohne Privilegien auf der unsicheren Seite ausgeführt wird.

Die in 5.6 beschriebenen Vereinfachungen durch den Wegfall des *monitor*s und die Verwendung von *Fiasco*–Mechanismen an seiner Stelle, wird auch im Codeumfang deutlich. Das *kqemu*–Modul basiert auf knapp 19000 Zeilen Quellcode, ist also ca. 7.5 mal so umfangreich wie die Implementierung von *kqemu–l4*.

6 Auswertung

6.1 Leistung

6.1.1 Messungen

Im Folgenden stelle ich einige Messungen vor. Während der Entwicklung habe ich mich zunächst weniger auf Leistungsfähigkeit, als auf Funktionalität konzentriert. Das heißt, ich habe *kqemu–l4* weiterentwickelt, um mehr Gast–Systeme zu unterstützen und dabei keinen besonderen Wert auf Leistungsfähigkeit gelegt. Im Laufe dieser Arbeit sind die meisten der folgenden Messungen entstanden. Am Ende habe ich einige Optimierungen in *Fiasco* und *kqemu–l4* implementiert und deren Auswirkung auf einzelne Testfälle gemessen.

Fiasco–Benchmarks

Auf *Fiasco* habe ich eine Anwendung zur Berechnung von Mandelbrot–Fraktalen für die Benchmarks verwendet. Dabei habe ich einmal die reine Berechnung (`fractal`) und die Berechnung zusammen mit der grafischen Darstellung des Fraktals (`fractal+con`) gemessen.

Abbildung 6.1 zeigt die Ergebnisse dieser Messungen im Vergleich zu anderen Virtualisierungslösungen. Außerdem zeigt die Abbildung die Ergebnisse der jeweiligen Messungen auf einer realen Maschine (`native`) als Vergleichswerte. Der Messergebnisse von *QEMU* ohne Beschleunigung sind ebenfalls dargestellt. In diesem Diagramm ist auf der Y–Achse die Ausführungszeit abgetragen. Je kleiner der jeweilige Balken desto besser ist die Leistung.

Dieses Benchmark ist eine reine Berechnung von Gleitkommazahlen und damit ein CPU–Benchmark. Die Messungen zeigen, dass Virtualisierung bei dieser Art von Benchmark sehr nah an die Leistung einer realen Maschine herankommt. *QEMU* als reiner Emulator ohne *Native Execution* ist dagegen wesentlich langsamer in der Berechnung der Fraktale. *kqemu–l4* ist bei der Berechnung ohne grafische Darstellung sehr nah an der Leistungsfähigkeit der anderen Virtualisierungslösungen.

Die Messungen mit grafischer Darstellung der Fraktale zeigen dagegen deutliche Leistungsunterschiede zwischen den Virtualisierungslösungen mit *Native Execution*. Neben der CPU–Virtualisierung wird hier auch I/O–/Geräte–Virtualisierung für die Grafikausgabe benötigt. *kqemu–l4* ist in diesem Benchmark deutlich langsamer als die reale Maschine oder *KVM*.

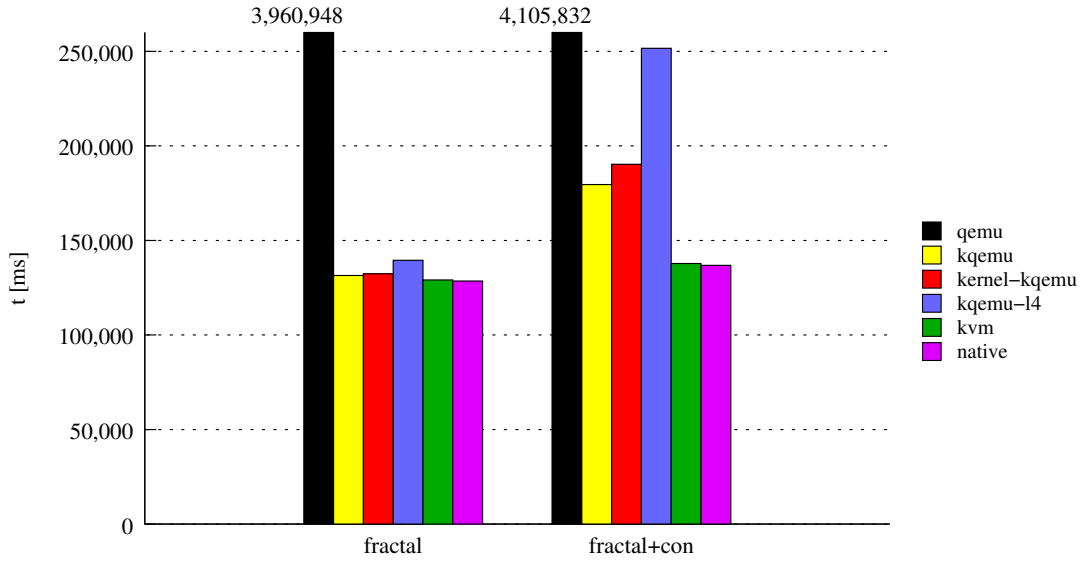

Abbildung 6.1: *Fiasco* Fraktal Benchmarks

Windows–Benchmarks

Unter Windows habe ich zwei verschiedene Benchmark–Anwendungen verwendet. Hier sind ebenfalls Vergleichswerte zur reinen Emulation mit *QEMU*, anderen Virtualisierungslösungen und der Ausführung auf der realen Maschine angegeben.

Abbildung 6.2 zeigt die Messergebnisse mit der Anwendung *StraightMark 2005*. Im Diagramm sind Benchmark–Punkte auf der Y–Achse dargestellt. Hohe Balken stehen für hohe Leistung.

Diese Anwendung ist ein reiner CPU–Benchmark. Wie schon in 6.1.1 festgestellt, zeigen auch diese Messwerte, dass die CPU–Virtualisierung mit *kqemu–l4* sehr leistungsfähig ist. Die Leistungsverluste gegenüber einer realen Maschine und anderen Virtualisierungslösungen mit *Native Execution* sind relativ gering. Der Leistungsgewinn gegenüber *QEMU* ohne Beschleunigung ist dementsprechend hoch.

Die Diagramme in den Abbildungen 6.3 und 6.4 zeigen die Messwerte der Benchmark–Anwendung *CPUBENCH*.

In Abbildung 6.3 sind die Einzelbenchmarks dieser Anwendung dargestellt, bei denen Punkte vergeben werden. Das Diagramm zeigt die Ergebnisse aller Testläufe relativ zur Leistung der realen Maschine. Hohe Balken stehen in diesem Diagramm für hohe Leistung.

Abbildung 6.4 zeigt die Einzelbenchmarks von *CPUBENCH*, bei denen die Ausführungszeiten verglichen werden. In diesem Diagramm stehen also hohe Werte für geringe Leistung.

Diese Messungen zeigen zum Großteil ähnliche Ergebnisse, wie die bisher vorgestellten Messungen. Besonders auffällig sind jedoch die Benchmarks KPPS aus Abbildung 6.3 und Queens, B5 aus Abbildung 6.4. Bei den anderen Messungen liefert die reale Maschine stets die besten und der Emulator stets die schlechtesten Messwerte. Virtuali-

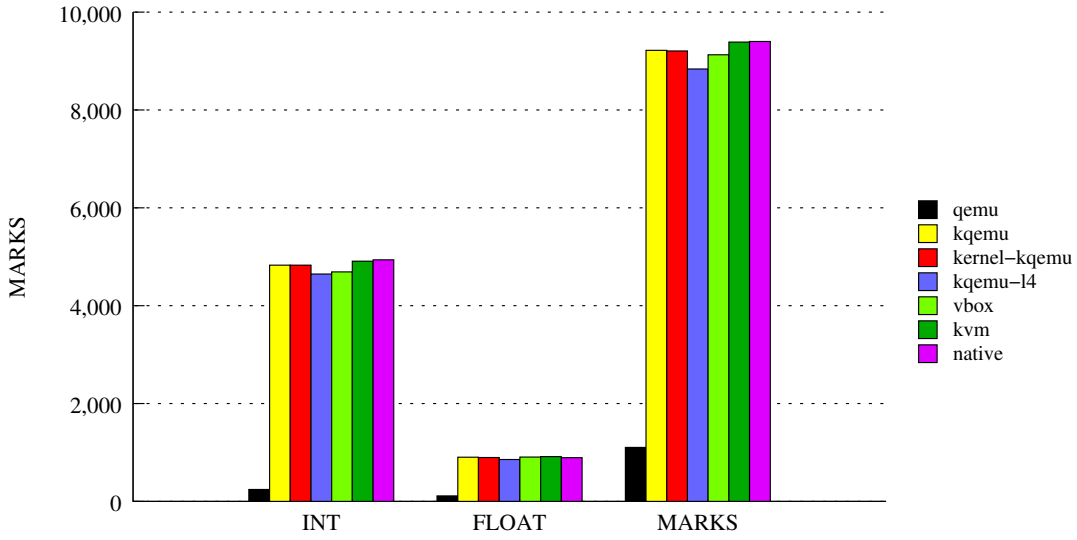

Abbildung 6.2: Windows XP Benchmarks *StraightMark 2005 1.2.1*

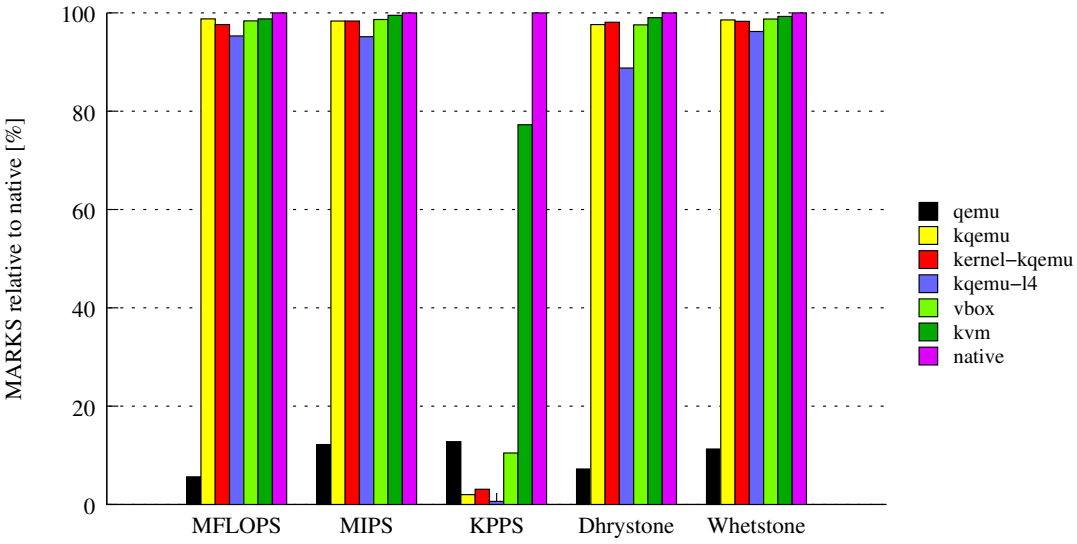

Abbildung 6.3: Windows XP Benchmarks *CPUBENCH v4.0.0.6*

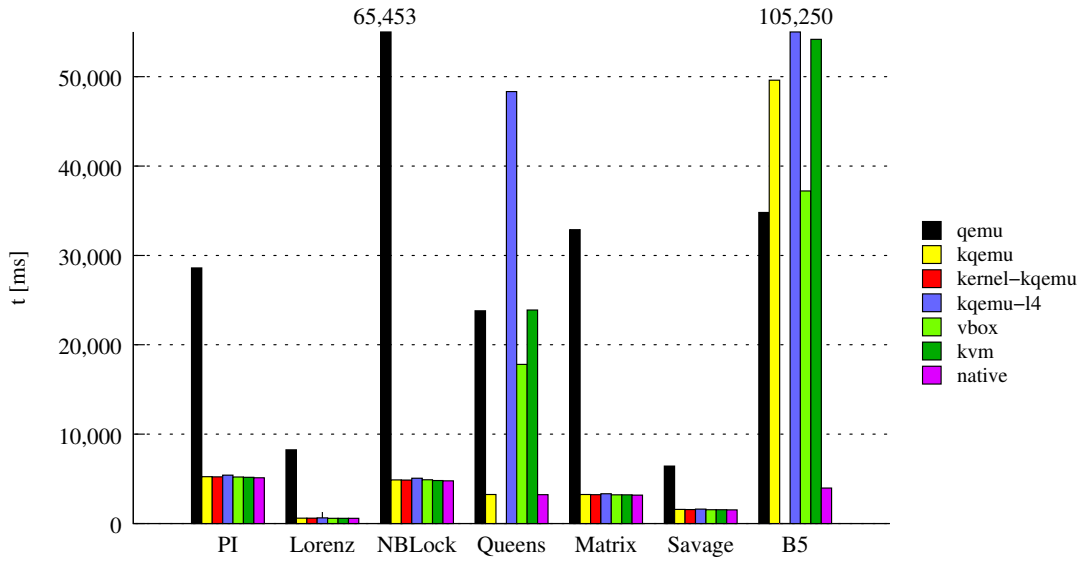

Abbildung 6.4: Windows XP Benchmarks *CPUBENCH v4.0.0.6*

sierung mit *Native Execution* ist immer etwas langsamer als Ausführung auf der realen Maschine, aber wesentlich besser als reine Emulation. Die drei genannten Einzelbenchmarks weichen deutlich von diesem Schema ab.

Eine genaue Erklärung für diese Abweichungen habe ich nicht gesucht. In Anbetracht der Tatsache, dass sowohl die Benchmarks als auch das Betriebssystem nicht quelloffen sind, wäre die Suche nach den Ursachen zu aufwendig. Im Wesentlichen zeigen die Messungen von *CPUBENCH* unter Windows XP ähnliche Leistungsmerkmale von *kqemu–l4* gegenüber den Vergleichswerten. Die drei Einzelbenchmarks weichen in allen untersuchten virtuellen Maschinen vom Schema ab. Aus diesem Grund schließe ich ein spezielles Problem in *kqemu–l4* aus.

Linux–Benchmarks

Mit Linux als Gast–System habe ich drei verschiedene Anwendungen gemessen.

- `bunzip2` — Dekomprimieren/Entpacken eines Linux Kernarchivs
- `bzip2` — Komprimieren/Packen eines Linux Kernarchivs
- `make` — Übersetzen ein Linux–Kerns

Abbildung 6.5 zeigt die Ergebnisse dieser Messungen. In diesem Diagramm ist die Ausführungszeit auf der Y–Achse dargestellt. Je niedriger ein Balken, desto höher ist die Leistung.

Das Ein– und Auspacken des Kernarchivs sind wieder Benchmarks, die vorrangig die CPU–Virtualisierung messen. Die Ergebnis der jeweiligen Berechnungen wurden nicht auf eine Festplatte geschrieben, sondern sofort verworfen. Außerdem habe ich jede der Messungen zwei mal durchgeführt und jeweils den Wert des zweiten Durchlaufs für das

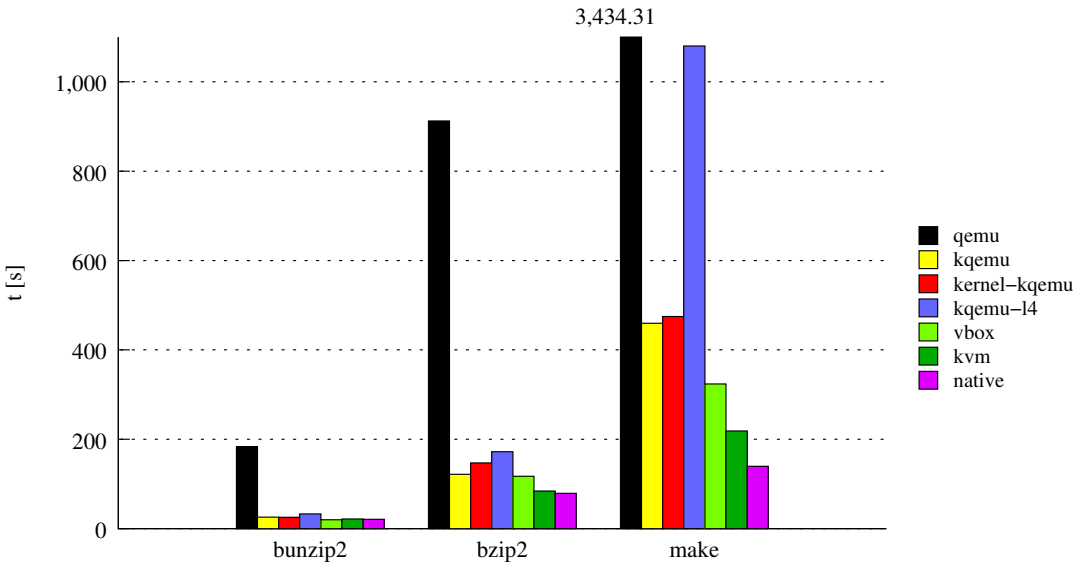

Abbildung 6.5: Linux Benchmarks

Diagramm verwendet. Damit wollte ich den Einfluss der I/O–Emulation für die virtuelle Festplatte möglichst gering halten. Beide Benchmarks zeigen ähnliche Ergebnisse wie die bisher vorgestellten Messungen mit anderen Gast–Systemen.

Die Übersetzung des Linux–Kerns hingegen zeigt eine deutliche Abweichung in *kqemu–l4*. Das Modul für *L⁴Linux* hat in den bisherigen Messungen stets 75 bis 95% der Leistung von *kqemu* erreicht. Bei der Übersetzung eines Linux–Kerns hat *kqemu–l4* hingegen nur knapp über 40% der Leistung von *kqemu* erzielt. Damit ist *kqemu–l4* immer noch ca. um den Faktor 3 schneller als *QEMU* auf *L⁴Linux*. Die Beschleunigung ist allerdings nicht so deutlich wie in den bisherigen Messungen.

Die von *kqemu–l4* bereitgestellte Ausführungsumgebung besteht im Wesentlichen aus den zwei Ressourcen CPU und Speicher. Die bisherigen Messungen haben gezeigt, dass die CPU–Virtualisierung von *kqemu–l4* leistungsfähig ist. Das entspricht auch dem, was man erwarten würde, da der Gast–Code nativ auf der Host–CPU ausgeführt wird. Die Leistungsverluste gegenüber einer realen Maschine sind in der Verwaltung der Ausführungsumgebung und den Kontextwechseln zwischen Host und Gast zu suchen. Bis auf die Speicherverwaltung sind diese Verwaltungsaufgaben jedoch relativ statisch und nicht vom Gast–Code abhängig. Aus diesem Grund habe ich die Speicherverwaltung von *kqemu–l4* als mögliche Ursache der schlechteren Leistung bei der Übersetzung des Linux–Kerns, näher untersucht.

6.1.2 Kontextwechsel und Speicherverwaltung in *kqemu–l4*

Die bisher betrachteten Benchmarks sind sehr rechenintensiv, brauchen für ihre Aufgabe aber nur relativ wenig Speicher. Bei der Übersetzung des Linux–Kerns kommen zu den eigentlichen Berechnungen noch viele Zugriffe auf virtuellen Speicher hinzu. In *kqemu–l4* wird ein Adressraum per Zeitmultiplex zwischen allen Anwendungen eines Gast–

Systems aufgeteilt. Immer wenn im Gast ein Adressraumwechsel durchgeführt wird, wird der Adressraum der *kqemu–l4–Task* komplett geleert (*Flush*). Anschließend wird er seitenweise wieder befüllt.

Messungen haben ergeben, dass bei den CPU–Benchmarks nur wenig virtueller Speicher zugegriffen wird. Dazu habe ich gemessen, wie viele Seiten im Durchschnitt bei einem Adressraum–*Flush* ausgeblendet werden. Für die CPU–Benchmarks werden im Schnitt 5 bis 10 Seiten pro *Flush*–Operation aus dem Adressraum entfernt. Bei der Übersetzung des Linux–Kerns hingegen habe ich Durchschnittswerte von 80 bis 100 Seiten gemessen.

Die richtige Deutung dieser Zahlen ist schwer. Der erste Faktor ist das Zugriffsverhalten einzelner Anwendungen auf virtuellen Speicher. Darunter fällt nicht nur der generelle Speicherbedarf, sondern auch Zugriffs–"Muster und –Intervalle auf bereits allozierten virtuellen Speicher. Der Umfang der Arbeitsmenge von Seiten wird von der jeweiligen Applikation bestimmt. Er ist aber auch davon abhängig, wie lange die Anwendung ausgeführt wurde, bevor eine *Flush*–Operation durchgeführt werden soll. Die Größe der Arbeitsmenge ist damit ebenfalls abhängig davon, wie viele Anwendungen im Gast–System ausführungsbereit sind und wie viele Adressraumwechsel daraufhin vom Gast–Betriebssystem durchführt werden.

In *kqemu* werden Seitenfehler vom *monitor* behandelt. Dabei ist es möglich, dass die Behandlung von Seitenfehlern ohne Kontextwechsel stattfinden kann. Das ist dann möglich, wenn die Seitentabellen für die Behandlung im *monitor*–Adressraum eingeblendet sind und der physische Speicher von vorherigen Zugriffen noch vor Auslagerung geschützt ist, das *Pinning* also noch aktiv ist. Aufgrund der Architektur von *kqemu–l4* tritt bei jedem Seitenfehler ein Adressraumwechsel zum VMM, also zum *Pager* der *kqemu–l4–Task*, auf.

Genau wie in *kqemu* wird in *kqemu–l4* bei einem Adressraum–*Flush* Seite für Seite aus dem Gast–Adressraum ausgeblendet. In *kqemu* invalidiert der *monitor* die entsprechenden Einträge in der *Shadow Page Table*. In *kqemu–l4* wird jede einzelne Seite mit dem `unmap`–Systemaufruf ausgeblendet. Das heißt, für jede Seite der Arbeitsmenge wird ein Kerneintritt in den *Fiasco*–Mikrokern durchgeführt. Jeder Kerneintritt ist potentiell mit Leistungsverlusten verbunden.

Im *Fiasco*–Kern wird der Zustand aller Adressräume in der *Mapping*–Datenbank gehalten. Die Operationen `map` und `unmap` ändern nicht nur die entsprechenden Einträge in den Seitentabellen, sie aktualisieren auch den Zustand dieser Datenstruktur. Die Pflege der *Mapping*–Datenbank kommt in *kqemu*, im Gegensatz zu *kqemu–l4*, nicht vor.

Diese Betrachtungen der Unterschiede zwischen *kqemu* und *kqemu–l4* zeigen mögliche Gründe für die schlechtere Leistung von *kqemu–l4* bei der Übersetzung des Linux–Kerns auf. Um die Vermutungen zu bestätigen, habe ich einige Optimierungen vorgenommen, um die Speicherverwaltung von *kqemu–l4* zu beschleunigen und die Beschleunigung anhand von Messergebnissen zu zeigen.

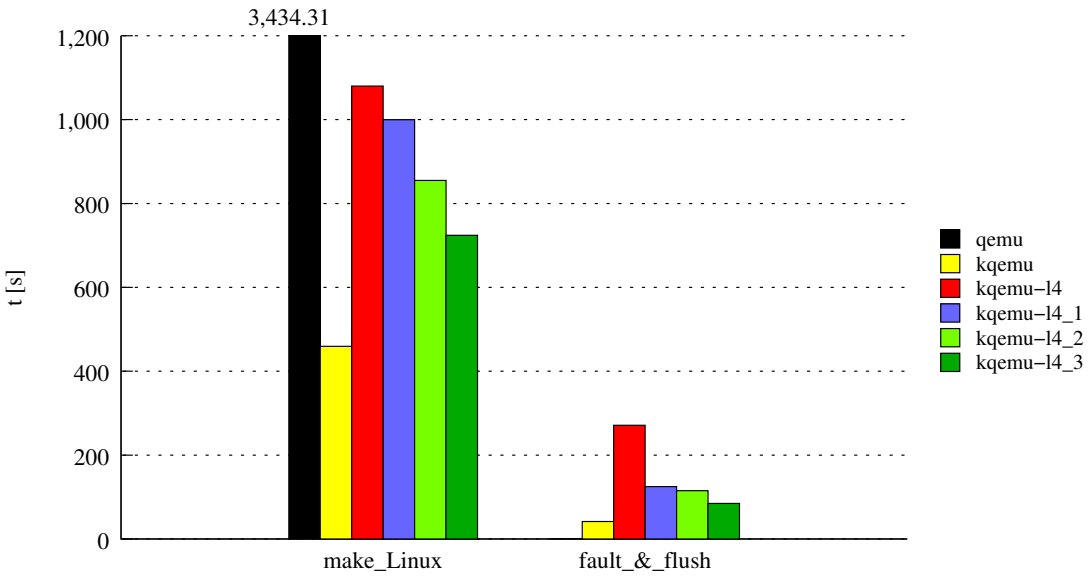

Abbildung 6.6: *kqemu–l4* Optimierungen

6.1.3 Optimierungen von *kqemu–l4*

Zunächst habe ich ein Benchmark entwickelt, um die Speicherverwaltung von *kqemu* und *kqemu–l4* direkt miteinander vergleichen zu können. Dieses Benchmark alloziert 100 Megabyte virtuellen Speicher und beschreibt jeweils das erste Byte jeder Seite. Es löst also viele Seitenfehler aus, ohne dabei komplizierte Berechnungen mit dem verwendeten Speicher durchzuführen. Sobald die 100 Megabyte seitenweise zugegriffen wurden, wird der Adressraum durch eine *Flush*–Operation geleert. Der gesamte Vorgang wird 1000 Mal in einer Schleife wiederholt. Damit konzentriert sich dieses Benchmark auf die Speichervirtualisierung, Berechnungen werden kaum durchgeführt.

Für die Implementierung dieses Benchmarks habe ich die Testumgebung erweitert, die mit *kqemu* ausgeliefert wird. Dieses spezielle Benchmark ist somit nur für Messungen von *kqemu* und *kqemu–l4* geeignet.

Wie erwartet zeigen die Messungen bei *kqemu–l4* eine wesentlich schlechtere Leistung bei der Speichervirtualisierung. Der beschriebene Test dauerte unter *kqemu–l4* etwa 6.5 Mal so lange, wie unter *kqemu*.

In Abbildung 6.6 sind die Ergebnisse dieser Messung im Diagramm dargestellt (`fault_&_flush`). Der rote Balken (`kqemu-l4`) zeigt jeweils die nicht optimierte Variante von *kqemu–l4*. Die anderen Varianten von *kqemu–l4* zeigen Messwerte, die mit verschiedenen optimierten Varianten aufgenommen wurden. Die Balken, die mit `make_Linux` beschriftet sind, zeigen jeweils die Zeit, die für die Übersetzung des Linux–Kerns benötigt wurde. In diesem Diagramm stehen kleine Balken für gute Leistung.

Basierend auf den Vermutungen, welche Unterschiede in der Speicherverwaltung den Leistungsverlust gegenüber *kqemu* auslösen, habe ich drei Optimierungen implementiert.

- eine Umgehung der *Mapping*–Datenbank für *kqemu–l4–Tasks*

- einen Systemaufruf, um Adressräume mit nur einem Kerneintritt zu leeren
- Verzicht auf die Deaktivierung der `SYSENTER`–Instruktion beim Kontextwechsel

Umgehung der *Mapping*–Datenbank

Die *Mapping*–Datenbank in *Fiasco* wird benötigt, damit *Tasks* ihren virtuellen Speicher an andere *Tasks* im System weitergeben können. Außerdem wird sie genutzt um *Mappings* zu finden, wenn Speicher wieder entzogen werden soll. Die *kqemu–l4–Tasks* sind in der Baumstruktur, die sich durch die hierarchischen *Pager* im Mikrokern ergibt, Blätter. Sie können keinen Speicher ihres Adressraums an andere *Tasks* weitergeben. Solange ihrem *Pager*, also dem *L⁴Linux*–Kern, kein Speicher entzogen wird, sind für diese *Tasks* keine Einträge in der *Mapping*–Datenbank nötig.

Die erste Optimierung besteht darin, dass für die *Tasks*, die *kqemu–l4* als Ausführungsumgebung für Gast–Code verwendet, keine Einträge in der *Mapping*–Datenbank angelegt werden. Dementsprechend besteht die `map`–Operation nur noch aus dem Anlegen eines Eintrages in den Seitentabellen des jeweiligen Adressraums. Die `unmap`–Implementierung iteriert in *Fiasco* über alle Kind–Knoten des Speicherbaums und entfernt die entsprechenden Einträge aus der Datenbank. Mit der Optimierung, die ich eingeführt habe, ist keine Iteration und damit Suche in der *Mapping*–Datenbank nötig. Anstatt beim `unmap` die virtuelle Adresse des *Pager*s anzugeben und anschließend *Mappings* zu suchen, wird bei der Optimierung lediglich ein konkretes *Mapping* angegeben und vom Kern entfernt. Das *Mapping* wird durch eine *TaskID* und eine virtuelle Adresse aus dem Adressraum der entsprechenden *Task* spezifiziert. Der Kern löscht gezielt den Eintrag aus der jeweiligen Seitentabelle und umgeht dabei die *Mapping*–Datenbank.

Diese Optimierung ist in Abbildung 6.6 in allen drei verbesserten Varianten von *kqemu–l4* enthalten (`kqemu-l4_1-3`). Die Variante `kqemu-l4_1` zeigt die alleinige Anwendung dieser Optimierung. Im Mikrobenchmark ist der Leistungsgewinn durch die Umgehung der *Mapping*–Datenbank deutlich zu sehen. Die Leistung hat sich im Gegensatz zu der nicht optimierten Variante verdoppelt. Der Einfluss der *Mapping*–Datenbank auf die Übersetzungsdauer des Linux–Kerns ist dagegen mit rund 8% relativ gering.

Für diese Optimierung musste ich *kqemu–l4* und den *Fiasco*–Mikrokern anpassen. Wie schon erwähnt, funktioniert diese Optimierung nur unter der Annahme, dass dem *L⁴Linux*–Kern kein Speicher entzogen wird. Sollte ihm dennoch Speicher entzogen werden, so kann ohne Einträge in der *Mapping*–Datenbank nicht festgestellt werden, ob der Speicher an die *kqemu–l4–Tasks* weitergegeben wurde und ebenfalls entzogen werden muss. Um die Semantik von `unmap` einzuhalten könnte beim Auftreten dieser Operation allen Kind–Adressräumen, die die Attribute der *kqemu–l4*–Adressräume tragen, generell jeglichen Speicher entziehen. Diese Lösung habe ich allerdings nicht implementiert, da der *L⁴Linux*–Kern in der Praxis keinen Speicher entzogen bekommt.

Adressraum–*Flush* mit einem speziellen Systemaufruf

Wie schon erwähnt, sind *Flushes* in der nicht optimierten Version von *kqemu–l4* durch einzelne `unmap`–Systemaufrufe implementiert. Dabei wird der Adressraum seitenweise geleert. Je nach Umfang der jeweiligen Arbeitsmenge kann das eine erhebliche Menge

an Systemaufrufen zur Folge haben. Dadurch werden viele Kontextwechsel zwischen dem Mikrokern und dem *L⁴Linux*–Kern, in dem *kqemu–l4* läuft, ausgelöst.

Um die Iteration über die Arbeitsmenge an eingeblendeten Seiten, und die damit verbundenen Kontextwechsel einzusparen, habe ich den `unmap`–Systemaufruf in *Fiasco* modifiziert. Mit dieser Veränderung reicht ein Aufruf von `unmap` mit definierten Parametern, um den kompletten Adressraum einer *kqemu–l4–Task* zu leeren.

Diese Variante von `unmap` implementiert im Prinzip einen neuen *Flush*–Systemaufruf. Damit ist der Aufwand für das leeren eines Gast–Adressraums konstant und nicht mehr vom Zustand der *Shadow Page Table* abhängig. Die Größe der aktuellen Arbeitsmenge hat in der modifizierten Version von *kqemu–l4* also keinen Einfluss mehr auf die Anzahl der Systemaufrufe oder Kontextwechsel für *Flushes*.

Die Messergebnisse zu dieser Optimierung sind ebenfalls in Abbildung 6.6 enthalten. Die Version, die in dem Diagramm mit `kqemu-l4_2` bezeichnet ist, enthält die beiden bisher angesprochenen Verbesserungen. Die Einführung und Verwendung des neuen Systemaufrufs hat auf die Ergebnisse im spezielle entwickelten Speicher–Benchmark nur eine geringe Auswirkung. Ich konnte eine leichte Leistungssteigerung gegenüber der vorherigen Version von *kqemu–l4* messen. Die Übersetzung des Linux–Kerns hingegen, hat von dieser Verbesserung ungleich stärker profitiert. Dort hat die Optimierung eine Leistungssteigerung um 15% bewirkt.

Für die Implementierung dieser Optimierung mussten ebenfalls der *Fiasco*–Mikrokern und *kqemu–l4* angepasst werden. Bei dieser Optimierung wird nur über die erste Stufe der Seitentabellen (*Page Directory*) iteriert. Werden im *Page Directory* gültige Einträge gefunden, werden sie invalidiert. Es werden also *Mappings* gelöscht, ohne die *Mapping*–Datenbank entsprechend anzupassen. Damit kann diese Optimierung ohne die Umgehung der *Mapping*–Datenbank nicht implementiert werden.

SYSENTER–Instruktion nicht deaktivieren

In 5.5.2 habe ich erläutert, dass der SYSENTER–Befehl in *kqemu–l4* eine *Exception* auslösen muss, um korrekt behandelt werden zu können. Um sicherzustellen, dass diese Instruktion fehlschlägt habe ich den *Fiasco*–Mikrokern modifiziert. Die Änderung besteht darin, ein Zustandsregister der CPU umzuschalten.

Änderungen am Prozessorzustand können in der Hardware dazu führen, dass Caches und die Befehls–Pipeline geleert werden. Das regelmäßige Umschalten dieses Zustandsregisters ist damit eine mögliche Quelle für Leistungsverluste bei den Kontextwechseln.

Um den Einfluss auf die Leistung von *kqemu–l4* zu messen, habe ich die SYSENTER–Anpassung des *Fiasco*–Mikrokerns für einige Messungen wieder ausgebaut. Eine erfolgreiche Messung ist mit dieser Version allerdings nur möglich, wenn der Gast–Code den SYSENTER–Befehl nicht enthält. Diese Bedingung ist für des Speicher–Benchmark gegeben. Das Linux Live–System, welches ich für die Übersetzung des Linux–Kerns verwendet habe, benutzt allerdings SYSENTER.

Um SYSENTER im Gast–System zu deaktivieren, habe ich in *QEMU* die Attribute der emulierten CPU so verändert, dass diese die SYSENTER–Instruktion nicht unterstützt. Linux fragt beim Startvorgang die Fähigkeiten des Prozessors ab. Je nachdem, ob die benutzte CPU SYSENTER unterstützt, wird unterschiedlicher Code für den Ker-

neintritt von Linux–Anwendungen an einer definierten Stelle im virtuellen Adressraum abgelegt. Dieser Code wird während des Betriebs von allen Linux–Anwendungen für den Kerneintritt verwendet. Die Unterscheidung wird nur ein Mal getroffen und die Fähigkeiten des Prozessors werden aus Ring 0 ausgelesen.

Diese Anpassung an *QEMU* ist allerdings keine generelle Lösung zur Vermeidung von SYSENTER in Gast–Code. Wenn die Abfrage der CPU–Fähigkeiten mittels CPUID in Ring 3 Code erfolgen würde, dann würde der Gast–Code die Fähigkeiten des realen Prozessors auslesen. In diesem Fall ist eine Maskierung einzelner Attribute nicht möglich. Diese Optimierung ist nicht universell einsetzbar und wurde nur mit Linux als Gast–System getestet.

Die Messwerte, die in Abbildung 6.6 mit `kqemu-l4_3` bezeichnet sind, sind mit einer Version von *kqemu–l4* aufgenommen, die alle drei, von mir implementierten Optimierungen enthält. Wie im Diagramm zu sehen ist, hat diese dritte Optimierung nochmal eine deutliche Leistungssteigerung bei beiden Benchmarks bewirkt.

Diese Anpassung verringert die Leistungsverluste, die durch die Kontextwechsel zwischen *L⁴Linux* und dem Gast–Code entstehen. Mit ihr sollten also auch Verbesserungen bei den CPU–Benchmarks messbar sein. Im Zusammenhang mit dem Speicherbenchmark ist die Leistungssteigerung darauf zurückzuführen, dass jeder Seitenfehler in *kqemu–l4* automatisch mit Kontextwechseln verbunden ist

6.2 Gast–Betriebssysteme

Ich habe eine Reihe von Gast–Betriebssystemen unter *kqemu–l4* getestet, die Ergebnisse sind in Tabelle 6.1 dargestellt. Einige Betriebssysteme werden sehr gut unterstützt. Bei ihnen kann durch die Verwendung von *kqemu–l4* eine erhebliche Beschleunigung im Gegensatz zu *QEMU* erreicht werden. Sie können außerdem teilweise mit einer unmodifizierten Version von *QEMU* verwendet werden. Für FreeBSD und DragonFlyBSD sollte eine angepasste Version von *QEMU* verwendet werden. Ihre Bootloader lösen den Pingpong–Effekt aus und sollten übersprungen werden.

Für andere Systeme gelten Einschränkungen. Diese Einschränkungen sind auf die Beschränkungen von *kqemu–l4* im Bezug auf Segmentierung zurückzuführen. *Fiasco* gibt die Segmentselektoren DS und CS vor, sie können zur Laufzeit nicht von *kqemu–l4* für den entsprechenden Gast angepasst werden. Die getesteten Windows–Versionen stellen Erwartungen an CS und DS. Sie können nicht als Gast–Betriebssysteme verwendet werden, wenn diese Erwartungen von *kqemu–l4* nicht erfüllt werden. Die Lösung dieses Problems besteht darin, *Fiasco* so anzupassen, dass CS und DS genau dieselben Werte enthalten, die von Windows–Gast–Code erwartet werden.

Die dritte Gruppe von Gast–Betriebssystemen kann zwar prinzipiell unter *kqemu–l4* verwendet werden, aber bei ihnen bringt *kqemu–l4* keine Beschleunigung. In der derzeitigen Version von *kqemu–l4* sind diese Systemen im Gegensatz zu *QEMU* sogar langsamer. Der Grund dafür ist der Pingpong–Effekt. Er wird bei einigen Betriebssystemen durch die Verwendung von virtuellem Speicher oberhalb der 3 Gigabyte–Grenze ausgelöst, bei anderen ist Segmentierung das Problem. Wie schon beschrieben kann das Problem des Pingpong–Effekts durch eine Anpassung der `kqemu_is_ok`–Funktion von *QEMU* gelöst

Status	**Gast–Betriebs-system**	**Kommentar**
funktioniert	Linux 2.4/2.6	
	Fiasco	
	ReactOS	
	DragonFlyBSD	Bootloader sollte übersprungen werden
	FreeBSD	Bootloader sollte übersprungen werden, Pingpong–Effekt beim Start
mit Einschränkungen	Windows 2000 Windows XP	nur mit spezieller GDT (CS und DS müssen übereinstimmen)
	NetBSD	einige Anwendungen führen zum Absturz (Segmentierung)
keine Beschleunigung	OpenBSD	Zugriffe auf reservierten Speicher oberhalb von 3 Gigabyte
	Plan 9	
	Haiku	
	Thix	Segmente nicht kompatibel (CS und DS)

Tabelle 6.1: Unterstützte Gast–Systeme

werden. Das würde bei diesen Systemen aber bedeuten, dass *kqemu–l4* vom angepassten *QEMU* nicht zur nativen Ausführung verwendet würde. Für sie würde die Anpassung von *QEMU* nur bedeuten, dass *kqemu–l4* keine Verlangsamung mehr auslöst, Beschleunigung kann nicht erreicht werden.

Neben den in Tabelle 6.1 genannten Gast–Betriebssystemen habe ich auch Betriebssysteme gefunden, die mit *kqemu–l4* nicht verwendet werden können. Beispiele sind *GNU Hurd* und OpenSolaris. Für beide Beispiele ist die fehlende Äquivalenz im Bezug auf Segmentierung der Grund warum sie nicht eingesetzt werden können.

7 Zusammenfassung und Ausblick

7.1 Ausblicke

Die Messungen haben ganz klar ergeben, dass das *Flushing* des virtuellen Adressraumes viel Zeit kostet und wesentlich langsamer ist als unter *kqemu*. Es ist theoretisch möglich, Gast–Anwendungen anhand ihrer Adressraumkonfiguration in *kqemu–l4* zu unterscheiden. In dem Fall kann man mehrere virtuelle Adressräume in *Fiasco* verwenden und spart potentiell viele *Flush*–Operationen und Seitenfehler.

Diese Idee implementiert im Prinzip einen Cache für *Shadow Page Tables*. Diesen Zwischenspeicher über Kontextwechsel zwischen *QEMU* und *kqemu–l4* kohärent zu halten ist nicht trivial. Der problematische Fall ist, dass der Gast–Kern Seitentabellen–Einträge ändert, die gerade in einer *Shadow Page Table* aktiv sind. Diese Einträge müssen dann im Cache angepasst oder invalidiert werden. *QEMU* übergibt *kqemu–l4* beim Eintritt eine Liste von Rahmennummern. Diese Liste beschreibt die Teile des physischen Speichers, die während der Aktivität von *QEMU* beschrieben wurden. Mit dieser Information ist es theoretisch möglich, die *Shadow Page Tables* kohärent zu halten. Diese Weiterentwicklung scheint sehr vielversprechend und könnte eventuell sogar leistungsfähiger sein als *kqemu*. Genau wie *kqemu–l4*, verwendet *kqemu* nur einen virtuellen Adressraum für Nutzer–Code. Dieser wird im Zeitmultiplex von allen Gast–Anwendungen geteilt.

Das häufige Beschreiben von `IA32_SYSENTER_CS` bei den *Thread*–Wechseln in *Fiasco*, hat einen deutlich messbaren negativen Einfluss auf die Leistung des Gesamtsystems. Wie schon in 5.5.2 erläutert, gibt es noch die Möglichkeit SYSENTER auch in *Fiasco* nicht zu nutzen und somit dauerhaft deaktiviert zu lassen. Man kann untersuchen, ob die Nachteile, die durch das häufige Umschalten entstehen, die Vorteile, die aus der Verwendung von SYSENTER in *Fiasco* resultieren, überwiegen. Anstatt das eigentliche Problem im *exception IPC*–Mechanismus zu umgehen, wäre die in 5.5.2 beschriebene Erweiterung von *exception IPC* wünschenswert.

kqemu–l4 benutzt für komplexe Aufgaben wie die Speicherverwaltung Mechanismen von *Fiasco*. Der Mikrokern bietet damit eine Abstraktionsschicht zwischen der tatsächlichen Hardwareplattform und dem *kqemu–l4*–Code. Dadurch ist *kqemu–l4* von der Plattform unabhängiger als *kqemu*. Für die Zukunft ist es denkbar, *kqemu–l4* zum Beispiel auf die ARM–Plattform zu portieren.

kqemu–l4 ist ein Forschungsprojekt, das im Rahmen meiner Diplomarbeit entstanden ist. Für den Produktiveinsatz müsste es noch eine Reihe von kleineren Änderungen erfahren. Eine der wichtigsten Anpassungen wären Änderungen an *QEMU*. *kqemu–l4* unterliegt anderen Rahmenbedingungen als *kqemu*. Das führt dazu, dass *kqemu–l4* in einigen Situationen nicht zur *Native Execution* von Gast–Code verwendet werden kann und *QEMU* trotzdem einen Eintritt in *kqemu–l4* veranlasst. Die Funktion `kqemu_is_ok` von *QEMU* müsste dementsprechend angepasst werden, um den in 5.3 beschrieben

Pingpong–Effekt zu vermeiden. Die Vermeidung dieses Effekts würde das Problem lösen, dass *kqemu–l4* in einigen Fällen nicht für Beschleunigung sondern Verlangsamung der virtuellen Maschine sorgt.

Ich habe *kqemu–l4* auf einer 32 Bit Maschine entwickelt und getestet. Für den Einsatz auf 64 Bit x86–Maschinen muss *kqemu–l4* an einigen Stellen angepasst werden. Der Umfang der nötigen Anpassungen ist gering, aber sie müssten trotzdem vorgenommen und getestet werden.

7.2 Zusammenfassung

Mit dieser Arbeit konnte ich zeigen, dass es möglich ist, eine leistungsfähige virtuelle Maschine für vollständige Virtualisierung auf einer kleinen *Trusted Computing Base* zu implementieren. Der *Fiasco*–Mikrokern wurde zwar angepasst, aber diese Anpassungen fallen bei der TCB kaum ins Gewicht. Im Prinzip hat eine herkömmliche *L⁴Linux*–Installation vom Umfang her dieselbe TCB wie eine Installation, die *kqemu–l4* unterstützt. Dadurch kann man auch ein ähnlich hohes Maß an Sicherheit von einem solchen System erwarten. Die *Trusted Computing Base* wurde gegenüber *kqemu* auf Linux, Windows oder FreeBSD erheblich verkleinert. Während die TCB bei den bisher unterstützen monolithischen Betriebssystem–Kernen im Bereich mehrere Millionen Zeilen Quellcode liegt, konnte sie in dieser Arbeit auf wenige zehntausend Zeilen reduziert werden.

Meine Messungen zeigen für viele Anwendungen nur geringe Leistungsverluste gegenüber *kqemu* auf Linux. Außerdem konnte ich für verschiedene Gast–Systeme und Anwendungen eine erhebliche Leistungssteigerung, verglichen mit *QEMU* auf *L⁴Linux* messen. Die Übersetzung eines Linux–Kerns, die in der Aufgabenstellung als relevante Arbeitslast vorgegeben ist, konnte mit *kqemu–l4* um den Faktor 4.7 beschleunigt werden. Messungen, die sehr CPU–lastig sind, zeigten, dass die Verluste der reinen CPU–Virtualisierung relativ gering sind. Ich habe Messungen vorgenommen, in denen die virtuelle Maschine lediglich 5% Leistungsverlust gegenüber der physischen Maschine gezeigt hat.

Was die Leistungsfähigkeit von *kqemu–l4* angeht, hat sich allerdings auch gezeigt, dass es in einigen Fällen wesentlich langsamer ist als *kqemu*. Außerdem gibt es Fälle in denen *kqemu–l4* weniger zur Beschleunigung, als vielmehr zur Verlangsamung der virtuellen Maschine beiträgt. Ich habe diese Fälle untersucht und mögliche Erklärungen für das unterschiedliche Leistungsverhalten geliefert. Mit der Implementierung von verschiedenen Optimierungen und detailierten Messungen habe ich einige dieser Erklärungen untermauert.

Die Leistungsverluste von *kqemu–l4* gegenüber *kqemu* sind auf die sehr unterschiedliche Implementierung zurückzuführen. Ich habe mich mich anfangs dazu entschieden den *Fiasco*–Mikrokern nicht zu erweitern, sondern den vorhandenen Mechanismus der *Alien Threads* für die Implementierung von *kqemu–l4* zu verwenden. Diese Entscheidung hat die Implementierung vergleichsweise einfach gemacht. Sie hat allerdings auch Einschränkungen mit sich gebracht, die sich negativ auf die Leistung auswirken.

Die meisten dieser Einschränkungen waren vorher bekannt. Meine Arbeit hat jedoch dabei geholfen, ihre Auswirkungen besser zu verstehen und messbar zu machen. Daraus konnten wertvolle Erkenntnisse zur Eignung von *Alien Threads* für Virtualisierung

getroffen werden, die für der zukünftige Entwicklung von *Fiasco* von Interesse sind.

Neben Linux und der obligatorischen Übersetzung des Linux–Kerns, habe ich noch eine Reihe weiterer Gast–Betriebssysteme erfolgreich unter *kqemu–l4* verwendet, darunter Microsoft Windows 2000 und XP, ReactOS, FreeBSD und NetBSD. Andere Systeme, wie zum Beispiel OpenBSD, plan9, haiku und Solaris, laufen dagegen nicht oder nicht zufriedenstellend.

Literaturverzeichnis

[AA06] Adams, Keith ; Agesen, Ole: A comparison of software and hardware techniques for x86 virtualization. In: *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*. New York, NY, USA : ACM, 2006. – ISBN 1–59593–451–0, S. 2–13

[AJ] Amstadt, Bob ; Johnson, Michael K.: Wine. In: *Linux J.*. – ISSN 1075–3583

[BDF+03] Barham, Paul ; Dragovic, Boris ; Fraser, Keir ; Hand, Steven ; Harris, Tim ; Ho, Alex ; Neugebauer, Rolf ; Pratt, Ian ; Warfield, Andrew: Xen and the art of virtualization. In: *SIGOPS Oper. Syst. Rev.* 37 (2003), Nr. 5, S. 164–177. `http://dx.doi.org/http://doi.acm.org/10.1145/1165389.945462`. – DOI http://doi.acm.org/10.1145/1165389.945462. – ISSN 0163–5980

[Bel] Bellard, Fabrice: *QEMU Website*. `http://bellard.org/qemu/`, Abruf: 26. Juli. 2008

[Bel05] Bellard, Fabrice: QEMU, a fast and portable dynamic translator. In: *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*. Berkeley, CA, USA : USENIX Association, 2005, S. 41–41

[Dik00] Dike, Jeff: A user-mode port of the linux kernel. In: *ALS'00: Proceedings of the 4th conference on 4th Annual Linux Showcase & Conference, Atlanta*. Berkeley, CA, USA : USENIX Association, 2000, S. 7–7

[FHN+04] Fraser, Keir ; Hand, Steven ; Neugebauer, Rolf ; Pratt, Ian ; Warfield, Andrew ; Williamson, Mark: Reconstructing I/O. 2004. – Forschungsbericht. – Xen's next generation I/O architecture

[Gol74] Goldberg, Robert P.: Survey of Virtual Machine Research. In: *IEEE Computer Magazine* (1974), June, S. 34–43

[Hab08] Habib, Irfan: Virtualization with KVM. In: *Linux J.* 2008 (2008), Nr. 166, S. 8. – ISSN 1075–3583

[HHL+97] Härtig, H. ; Hohmuth, M. ; Liedtke, J. ; Schönberg, S. ; Wolter, J.: The Performance of μ-Kernel-based Systems. In: *Proceedings of the 16th ACM Symposium on Operating System Principles (SOSP)*. Saint-Malo, France, Oktober 1997, S. 66–77

[JVL07] Jerger, N.E. ; Vantrease, D. ; Lipasti, M.: An Evaluation of Server Consolidation Workloads for Multi-Core Designs. In: *Workload Characterization, 2007. IISWC 2007. IEEE 10th International Symposium on* (2007), Sept., S. 47–56. http://dx.doi.org/10.1109/IISWC.2007.4362180. – DOI 10.1109/IISWC.2007.4362180

[KW00] Kamp, Poul-Henning ; Watson, Robert N. M.: Jails: Confining the omnipotent root. (2000)

[Law] Lawton, Kevin P.: Bochs: A Portable PC Emulator for Unix/X. In: *Linux J.*. – ISSN 1075–3583

[lin] *The Linux–VServer Project.* http://linux-vserver.org/, Abruf: 26. Juli. 2008

[LUSG04] LeVasseur, Joshua ; Uhlig, Volkmar ; Stoess, Jan ; Götz, Stefan: Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines. In: *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*. San Francisco, CA, Dezember 2004

[MMH08] Murray, Derek G. ; Milos, Grzegorz ; Hand, Steven: Improving Xen security through disaggregation. In: *VEE '08: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*. New York, NY, USA : ACM, 2008. – ISBN 978–1–59593–796–4, S. 151–160

[PG74] Popek, Gerald J. ; Goldberg, Robert P.: Formal requirements for virtualizable third generation architectures. In: *Commun. ACM* 17 (1974), Nr. 7, S. 412–421. http://dx.doi.org/http://doi.acm.org/10.1145/361011.361073. – DOI http://doi.acm.org/10.1145/361011.361073. – ISSN 0001–0782

[PT04] Price, Daniel ; Tucker, Andrew: Solaris Zones: Operating System Support for Consolidating Commercial Workloads. In: *LISA '04: Proceedings of the 18th USENIX conference on System administration*. Berkeley, CA, USA : USENIX Association, 2004, S. 241–254

[RI00] Robin, John S. ; Irvine, Cynthia E.: Analysis of the Intel Pentium's ability to support a secure virtual machine monitor. In: *SSYM'00: Proceedings of the 9th conference on USENIX Security Symposium*. Berkeley, CA, USA : USENIX Association, 2000, S. 10–10

[ros] *Apple — Rosetta Website.* http://www.apple.com/de/rosetta/, Abruf: 26. Juli. 2008

[SESS96] Seltzer, Margo I. ; Endo, Yasuhiro ; Small, Christopher ; Smith, Keith A.: Dealing with disaster: surviving misbehaved kernel extensions. In: *OSDI '96: Proceedings of the second USENIX symposium on Operating*

systems design and implementation. New York, NY, USA : ACM, 1996. – ISBN 1–880446–82–0, S. 213–227

[SL03] Sapuntzakis, Constantine ; Lam, Monica S.: Virtual appliances in the collective: a road to hassle-free computing. In: *HOTOS'03: Proceedings of the 9th conference on Hot Topics in Operating Systems*. Berkeley, CA, USA : USENIX Association, 2003, S. 10–10

[SM08] Sun Microsystems, Inc. (Hrsg.): *The VirtualBox architecture*. Version: 2008. `http://virtualbox.org/wiki/VirtualBox_architecture`, Abruf: 26. Juli. 2008

[SPF+07] Soltesz, Stephen ; Pötzl, Herbert ; Fiuczynski, Marc E. ; Bavier, Andy ; Peterson, Larry: Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In: *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. New York, NY, USA : ACM, 2007. – ISBN 978–1–59593–636–3, S. 275–287

[SPHH06] Singaravelu, Lenin ; Pu, Calton ; Härtig, Hermann ; Helmuth, Christian: Reducing TCB complexity for security-sensitive applications: three case studies. In: *EuroSys '06: Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*. New York, NY, USA : ACM, 2006. – ISBN 1–59593–322–0, S. 161–174

[SVL] Sugerman, J. ; Venkitachalam, G. ; Lim, B.: *Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor*. `citeseer.ist.psu.edu/venkitachalam01virtualizing.html`

[vmw] *VMware Website*. `http://www.vmware.com/`, Abruf: 26. Juli. 2008