Kapselung von Standard-Betriebssystemen

Dissertation

zur Erlangung des akademischen Grades Doktoringenieur (Dr.-Ing.)

vorgelegt an der
Technischen Universität Dresden
Fakultät Informatik

eingereicht von

Diplom-Informatiker Frank Mehnert
geboren am 18. Juli 1971 in Altdöbern

Gutachter: Prof. Dr. rer. nat. Hermann Härtig
Technische Universität Dresden

Prof. Dr. rer. nat. Wolfgang Nagel
Technische Universität Dresden

Prof. Dr.-Ing. habil. Winfried Kühnhauser
Technische Universität Ilmenau

Tag der Verteidigung: 14. Juli 2005

Dresden im Juli 2005

Danksagung

An erster Stelle möchte ich mich bei meinem Betreuer, Herrn Prof. Dr. Hermann Härtig, für seine große Unterstützung und seinen unerschütterlichen Glauben in meine Fähigkeiten bedanken. Er versteht es hervorragend, seine Mitarbeiter zu motivieren und zu fordern.

Die meisten Mitglieder der Gruppe Betriebssysteme der TU Dresden waren mittelbar am Entstehen dieser Arbeit beteiligt. Ich verdanke ihnen viele fruchtbare Diskussionen und ein entspanntes Arbeitsumfeld. Hervorheben möchte ich an dieser Stelle Dr.-Ing. Michael Hohmuth, Jean Wolter und Jork Löser. Adam Lackorzynski hatte immer ein offenes Ohr für meine Probleme und ermöglichte mir in seiner Aufgabe als Administrator stets eine professionelle Arbeitsumgebung. Explizit danken möchte ich hiermit auch unserer Sekretärin, Frau Spehr.

Besonderer Dank gebührt meinen Gutachtern, Herrn Prof. Dr.-Ing. Winfried Kühnhauser und Herrn Prof. Dr. Wolfgang E. Nagel, für ihre schnelle Reaktion und ihre Kommentare, die mir sehr bei der Verbesserung dieser Arbeit geholfen haben.

Mein ganz spezieller Dank gilt meiner Eike, die lange auf den Abschluss dieser Arbeit warten musste und mich oft von meinen häuslichen Pflichten entband, die aber immer an mich glaubt und mir mit ihrer Liebe die Unterstützung gibt, die man sich nur wünschen kann.

Bedanken möchte ich mich auch ganz herzlich bei meinen Eltern, die mir immer vertrauen und mit Liebe und Rat zur Seite stehen.

Inhaltsverzeichnis

1 Einleitung **1**

2 Grundlagen und Stand der Technik **7**

2.1 Begriffsbestimmungen 7
2.2 Wiederverwendung von Standard-Betriebssystemen 8
2.2.1 Einsatzgebiete 8
2.2.2 Gründe für die Kapselung von Standard-Betriebssystemen 9
2.2.3 Grundlegende Arten der Kapselung 10
2.2.4 Verbesserte Kapselung durch Hardware-Erweiterungen 14
2.2.5 Nicht vertrauenswürdige Dienste in vertrauenswürdiger Umgebung 15
2.3 L^{4}Linux / Nizza 15
2.3.1 Die Nizza-Architektur 16
2.3.2 Der Fiasco-Mikrokern 17
2.3.3 Verwaltung von Ressourcen 17
2.3.4 L^{4}Linux 19
2.4 Spezielle Aspekte der Virtualisierung 22
2.4.1 Cache als Ressource und Cache-Coloring 22
2.4.2 Optimierung der Adressraumwechsel 23
2.4.3 Einschränkung der IO-Rechte 25
2.4.4 Mehrprozessorsysteme 27
2.5 IO-Architekturen 28
2.5.1 Programmierte IO (PIO) 28
2.5.2 Direkter Speicher-Zugriff (DMA) 28
2.5.3 Busmaster-DMA über PCI-Busse 29
2.5.4 Einschränkung von Busmaster-DMA mittels spezieller Hardware 30

3 Entwurf **33**

3.1 Synchronisation ohne Sperren der Interrupts 33
3.1.1 Zeitkritischer Code 33
3.1.2 Gegenseitiger Ausschluss 34
3.1.3 Vorgehensweise 35
3.2 Sichere Ein-/Ausgabe mittels Busmaster-DMA 35
3.2.1 Verbot von Busmaster-DMA 36
3.2.2 Verwaltung von IO-Adressräumen 37
3.2.3 IOMMU-Implementierung in Hardware 40
3.2.4 IOMMU-Implementierung in Software 40

4 Implementierung **47**
4.1 Linux mit eingeschränkten IO-Rechten . . . 47
4.1.1 Globales `cli`-Lock . . . 47
4.1.2 Atomares Aufwecken . . . 49
4.1.3 Programmierung des Interrupt-Controllers . . . 50
4.1.4 Programmierung der CMOS-Uhr . . . 52
4.1.5 Übertragung auf andere Systeme . . . 52
4.2 Performance-Betrachtungen auf Standard-Hardware . . . 53
4.2.1 Schneller Kernein- und -austritt . . . 53
4.2.2 Linux im kleinen Adressraum mit eingeschränkten IO-Rechten . . . 53
4.3 Software-IOMMU . . . 56
4.3.1 Digital DS2114x Tulip Fast-Ethernet-Adapter . . . 56
4.3.2 Intel PRO/1000 Gigabit-Ethernet-Adapter . . . 57
4.3.3 IDE-Controller . . . 58
4.3.4 Wiederverwendung von Mediator und Emulator . . . 58
4.3.5 Codegrößen . . . 59

5 Leistungsbewertung **61**
5.1 DMA-Virtualisierung bei IDE-Controllern . . . 63
5.1.1 Anwendungsbenchmark . . . 63
5.1.2 Synthetischer Anwendungsbenchmark . . . 65
5.2 DMA-Virtualisierung bei Netzwerkadaptern . . . 66
5.2.1 Fast-Ethernet . . . 66
5.2.2 Gigabit-Ethernet . . . 68
5.3 Kosten der Ausführung von Linux auf Fiasco . . . 70
5.4 Linux mit eingeschränkten IO-Rechten . . . 73

6 Schlussfolgerungen und Ausblick **75**
6.1 Beiträge dieser Arbeit . . . 75
6.2 Vorschläge für zukünftige Arbeiten . . . 76

A Testumgebung **77**

B Kenngrößen ausgewählter Prozessoren **79**
B.1 Kosten für wichtige Prozessorinstruktionen . . . 79
B.2 Cache-Größen . . . 80
B.3 TLB-Größen und -Zugriffszeiten . . . 80

C IPC-Messungen unter Fiasco **83**
C.1 IPC innerhalb eines Adressraumes . . . 84
C.2 IPC zwischen zwei Adressräumen . . . 84
C.3 Vergleich von Fiasco mit dem L4-Kern von Liedtke . . . 85

Glossar **87**

Literaturverzeichnis **91**

Kapitel 1

Einleitung

Populäre Betriebssysteme sind in heutiger Zeit meist monolithisch aufgebaut und bieten eine Fülle von Kern-Funktionalität. Der aktuelle Linux-Kern in der Version 2.6.8.1 unterstützt 283 zum Teil sehr komplexe Systemaufrufe. Die Entwicklung verläuft sehr dynamisch, was sich in einer Vielzahl von Protokollen, Dateisystemen und zu unterstützender Hardware offenbart. Zeitliche Zusagen oder Garantien bestimmter sicherheitsrelevanter Eigenschaften können so genannte *Legacy*-Systeme wie Linux und Microsoft Windows XP aufgrund ihrer Komplexität nicht bieten. Der Entwicklungsfokus liegt hier vielmehr in einer Vielzahl von unterstützter Hardware und Protokolle sowie in einer guten *Gesamt-Performance*.

Die immer häufiger auftretenden Meldungen über Sicherheitslücken in heutigen Betriebssystemen [92] und Browser-Anwendungen [15, 16, 24] zeigen, dass diese Systeme für hochzuverlässige Lösungen prinzipiell nicht geeignet sind. Weiterhin sind Standard-Betriebssysteme in heutiger Zeit immer noch nicht in der Lage, zeitliche Anforderungen von Multimedia-Anwendungen unabhängig von der aktuellen Systemlast einzuhalten. Das Abspielen eines Video unter Linux oder Windows XP kann durch gleichzeitige Benutzung von Festplatte oder CPU durch andere Anwendungen kurzzeitig unterbrochen werden.

Neuentwicklungen von Betriebssystemen, die im Hinblick auf Zusagenfähigkeit und Sicherheit keine Kompromisse eingehen, haben oft nur ein begrenztes Angebot an Anwendungen und können in Bezug auf Unterstützung aktueller Hardware und Protokolle nicht mit Standard-Betriebssystemen mithalten. Als mögliche Lösung für dieses Problem könnten viele Komponenten aus Standard-Betriebssystemen *wiederverwendet* werden. Da diesen nicht vertraut werden kann, müssen sie von vertrauenswürdigen Anwendungen *isoliert* werden, um Vertraulichkeit und Integrität sensibler Daten sowie die Verfügbarkeit der anderen Dienste nicht zu beeinträchtigen.

Mit der heute verfügbaren Leistung von Standardprozessoren werden große Server-Farmen mit jeweils tausenden von voreinander geschützten Laufzeitumgebungen wirtschaftlich (Server-Konsolidierung). Auf einem Server wird nicht *eine* Umgebung mit *einem* Betriebssystem für viele Nutzer ausgeführt. Vielmehr kann jeder Nutzer *sein* Betriebssystem für *seine* Umgebung wählen, wobei aufgrund der oben dargestellten Gründe auch hier jede Umgebung isoliert von den anderen ausgeführt werden muss.

Ein Trend geht daher in Richtung Kapselung durch Virtualisierung. Anstatt ein Betriebssystem direkt auf der Hardware auszuführen, wird es als so genannter Gast innerhalb einer virtuellen Maschine (VM) gekapselt. Mehrere Gäste können parallel in einer jeweils eigenen VM ausgeführt werden. Ein *VM-Monitor* stellt die Trennung der Gäste sicher und ist für die Kommunikation des Gastes mit der Außenwelt sowie ggf. für die Kommunikation der Gäste untereinander verantwortlich. Der VM-Monitor läuft in einem privilegierten Prozessormodus, die Gäste haben nur eingeschränkte Rechte und laufen im Nutzermodus.

Virtualisierung ist nicht neu [81]: Bereits Mitte der sechziger Jahre des zwanzigsten Jahrhunderts forschte IBM Watson an der M44/44X-Architektur, die virtuelle Maschinen als Abbild eines IBM-7044-Systems bot. Das IBM-System VM/370 bildete schließlich den Ausgangspunkt für die Entwicklung einer bis heute anerkannten und robusten Architektur von Großrechnern.

Eine vollständige Virtualisierung ist aber beispielsweise auf der heute weit verbreiteten x86-Architektur nicht möglich, da bestimmte Voraussetzungen der Hardware dies verhindern. Eine Erweiterung der x86-Architektur im Hinblick auf bessere Virtualisierung zeichnet sich ab, ist jedoch noch nicht in aktueller Hardware zu finden.

Eine abgeschwächte Form der Virtualisierung, die Para-Virtualisierung, erlaubt Veränderungen am Gast-System, so dass dieses besser mit der virtuellen Maschine zusammenarbeiten kann. Aktuelle Systeme wie L^4Linux [25] und Xen/XenoLinux [5] zeigen, dass mit dieser Technik eine gute Performance erreicht werden kann. L^4Linux stellt eine spezielle Form der Para-Virtualisierung dar, wobei der architekturabhängige Teil des Linux-Kerns durch eine Emulationsschicht ersetzt wurde, die bestimmte direkte Hardware-Zugriffe durch Systemaufrufe eines Mikrokerns ersetzen.

Der Aufwand für die Virtualisierung kann beträchtlich steigen, wenn einem bzw. mehreren Gästen erlaubt wird, direkt auf die Hardware zuzugreifen. Dies kann einerseits aus Performance-Gründen notwendig sein, andererseits kann der Gast als Treiber-Lieferant gebraucht werden. Direkter Zugriff auf DMA-fähige Geräte durch nicht vertrauenswürdige Treiber erfordert eine Lösung für das DMA-Problem: Der Zugriff auf Speicher wird üblicherweise durch die *Memory Management Unit* (MMU) virtualisiert, die virtuelle Adressräume zur Verfügung stellt und somit Speicherschutz zwischen den Gästen sowie zwischen Gast und VM-Monitor bietet. Dieser Schutz kann allerdings durch die DMA-Einheit von Geräten überwunden werden, da die Adressräume der MMU für die CPU, nicht aber für IO-Busse gelten.

In aktueller Standard-Hardware sind so genannte IOMMUs zu finden, die *einen* eigenen Adressraum für *alle* IO-Geräte eines IO-Busses erzeugen und damit zumindest verhindern, dass DMA-fähige Geräte auf beliebigen Speicher zugreifen können. Allerdings schränken heutige IOMMUs nicht die Kommunikation zwischen Geräten ein – allen Geräten eines Busses ist der *gleiche* IO-Adressraum zugeordnet.

Isolation kann auch anders erreicht werden als durch Kapselung mit Hardwareunterstützung. Durch interpretative Abarbeitung wird bei Java-Programmen verhindert, dass diese aus einer Sandbox mit definierten Zugriffsrechten auf Ressourcen außerhalb der Laufzeitumgebung ausbrechen können. Der Nachteil der geringeren Abarbeitungsgeschwindigkeit kann durch *on-the-fly* Compilationstechniken[1] abgeschwächt werden. Allerdings eignet sich nicht jede Sprache für die interpretative Abarbeitung. Gerätetreiber sind fast ausschließlich in C oder C++ geschrieben. Interpretation ist schon aufgrund der fehlenden Typsicherheit dieser Sprachen sehr schwierig. Ferner sind Implementierungen solch geschützter Laufzeitumgebungen meist sehr umfangreich, weshalb man diesen wiederum nicht vertrauen kann.

Ein weiteres Mittel zur Isolation nicht vertrauenswürdigen Codes ist die statische Programmanalyse, bei der alle Pfade *off-line*, d. h. vor der Ausführung, ermittelt und auf mögliche Zugriffsverletzungen hin untersucht werden. Wie bei der interpretativen Ausführung werden auch hier spezielle Anforderungen an die Programmiersprache gestellt. Im Gegensatz zur Interpretation steigt allerdings der Verifikationsaufwand exponentiell mit zunehmender Codegröße. Diese Kosten müssen zudem bei jeder neuen Version des Programms erneut bezahlt werden.

[1] oft auch als just in time compilation (JIT) bezeichnet

Ziele dieser Arbeit

Für die isolierte Ausführung umfangreicher Komponenten erscheinen Techniken zur Kapselung unter Einbeziehung von Hardwareschutzmechanismen als geeignetes Mittel. Die Para-Virtualisierung stellt im Vergleich mit einer vollständigen Virtualisierung geringere Anforderungen an die Hardware, erfordert aber eine Modifikation des Gastes, der daher (zumindest teilweise) im Quellcode vorliegen muss.

Zusammenfassend werden bei der Anwendung der Para-Virtualisierung folgende Anforderungen gestellt:

Starke Kapselung Es sollen Standard-Betriebssysteme wiederverwendet werden, die als nicht vertrauenswürdig und bösartig angesehen werden (siehe Abschnitt 2.2.3). Dies setzt eine starke Kapselung des Codes voraus.

Benutzung von Standard-Hardware Populäre Betriebssysteme sind vor allem auf der x86-Architektur zu finden, die nur ungenügende Unterstützung für vollständige Virtualisierung und starke Kapselung bietet (siehe Abschnitt 2.2.3). Daraus ergeben sich besondere Herausforderungen.

Geringe Zusatzkosten Der Aufwand für die Kapselung impliziert zusätzliche Kosten zur Laufzeit durch stärkere Nutzung von Ressourcen (CPU, Cache, Speicher). Dieser Overhead ist zu minimieren.

Keine/geringe Änderungen am Quellcode Je weniger Änderungen am Quellcode des zu kapselnden Objektes notwendig sind, desto einfacher ist die Portierung auf andere Architekturen bzw. Betriebssysteme.

Kleine Trusted Computing Base (TCB) Der Umfang an vertrauenswürdigem Code soll so gering wie möglich sein, um die Wahrscheinlichkeit von potenziellen Fehlern zu minimieren.

Ausführung in einer Echtzeitumgebung Der gekapselte Code soll in einer Echtzeitumgebung ausführbar sein und Prozesse mit zeitlichen Anforderungen nicht beeinflussen. Diese Anforderung überschneidet sich mit dem ersten Punkt.

Als Grundlage für meine Arbeit wählte ich L^{4}Linux, eine Portierung des Linux-Kerns auf den Mikrokern L4 [25]. Die wichtigste Eigenschaft von L^{4}Linux ist die Ausführung des Linux-Kerns als Prozess im Nutzermodus. Dies reduziert drastisch die Menge an Code, der im privilegierten Modus ausgeführt werden muss. Allerdings muss die bisherige Implementierung von L^{4}Linux noch immer vertrauenswürdig sein:

- Sowohl Xen als auch L^{4}Linux nutzen bisher keine Technik, um das DMA–Sicherheitsproblem zu lösen.
- Viele derartige Systeme wurden bisher ausschließlich auf Performance optimiert: Der Xen-VM-Monitor ist nicht echtzeitfähig, und L^{4}Linux wurde bisher auf nicht echtzeitfähigen L4-Mikrokernen ausgeführt [56, 90].

- Der Linux-Kern und Linux-Anwendungen werden zwar mit Nutzerprivilegien, aber ohne Einschränkung der IO-Rechte ausgeführt und können daher das System kompromittieren, z. B. durch Abschalten der Prozessor-Interrupts oder durch unkontrollierte Ein-/Ausgabe über IO-Ports mit externen Geräten. Nicht davon betroffen ist die Ein-/Ausgabe über eingeblendeten Speicher (*memory-mapped* IO), da diese über die MMU kontrolliert werden kann.

- Die in [25] verwendete Version von L^{4}Linux hat direkten Zugriff auf den Interrupt-Controller und ist daher in der Lage, den Zeitgeber-Interrupt des Kerns zu sperren. Das preemptive Scheduling des Kerns ist auf den Zeitgeber-Interrupt angewiesen.

Das Ziel dieser Arbeit besteht darin, diese Probleme zu lösen und damit zu ermöglichen, dass L^{4}Linux als nicht vertrauenswürdige Anwendung ausgeführt werden kann.

Daneben wird untersucht, welche Performance auf Standard-Hardware mit dieser Art von „gezähmtem“ Linux im Vergleich zu Standard-Linux erreicht werden kann. Für L^{4}Linux wurde in der ursprünglichen Implementierung von 1997 auf der damaligen Hardware ein Performance-Overhead von weniger als 5 % bei Standardanwendungen gemessen. Seitdem hat sich die Hardware um mehrere Generationen weiterentwickelt. Im Unterschied zum damals benutzten L4-Kern von Liedtke [56], der explizit auf Performance optimiert war, wird für die vorliegende Arbeit der echtzeitfähige Fiasco-Mikrokern [35] verwendet.

Ergebnisse

In dieser Arbeit wird eine Lösung des DMA-Problems und eine mögliche Implementierung in Hardware vorgestellt. Dabei werden IO-Geräten jeweils separate IO-Adressräume zugewiesen.

Eine derartige Implementierung existiert bisher nicht in verfügbarer Standard-Hardware. Mittels Teil-Virtualisierung von Geräten können IO-Adressräume allerdings in Software emuliert werden. Dabei werden Zugriffe von Gerätetreibern auf bestimmte Register von DMA-fähigen Geräten mit herkömmlichen Techniken kontrolliert und bei Bedarf abgebrochen. Anhand von beispielhaften Implementierungen für Netzwerkadapter und IDE-Controller wird gezeigt, welche zusätzlichen Kosten dabei entstehen. Mit typischen Anwendungen steigt bei IDE-Controllern die CPU-Last um weniger als 3 %. Bei Netzwerkadaptern ist die Steigerung der CPU-Last stark abhängig von der Interrupt-Last. Bei Fast-Ethernet wurde bei TCP eine Erhöhung der Auslastung von etwa 10 % gemessen, bei Gigabit-Ethernet etwas mehr (ein direkter Vergleich ist schwierig, da gleichzeitig die nutzbare Datenrate sinkt).

Weiterhin wird gezeigt, wie L^{4}Linux mit eingeschränkten IO-Rechten ausgeführt werden kann. Viele der dabei verwendeten Techniken lassen sich auf andere Systeme übertragen.

Schließlich ist eine umfangreiche Performance-Analyse von L^{4}Linux auf der Nizza-Architektur Bestandteil dieser Arbeit.

Struktur dieser Arbeit

Diese Arbeit ist wie folgt aufgebaut: In Kapitel 2 werden aktuelle Entwicklungen auf dem Gebiet der Kapselung von Gerätetreibern und Betriebssystemen dargestellt. An dieser Stelle wird insbesondere die Nizza-Architektur mit Fiasco als Basis und L^{4}Linux als Ausführungsumgebung für

Standard-Linux-Anwendungen vorgestellt, und es werden Probleme in der bisherigen Implementierung von L^{4}Linux benannt.

In Abschnitt 3.1 wird gezeigt, wie alle Stellen, an denen L^{4}Linux bisher noch die Interrupts sperrt, durch geeignete Algorithmen ersetzt werden können. Dies ist notwendig, damit L^{4}Linux mit eingeschränkten IO-Rechten ausgeführt werden kann. Abschnitt 3.2 widmet sich der Lösung des DMA-Problems, sowohl mit Hilfe von Hardware-Erweiterungen als auch als rein softwarebasierte Implementierung.

In Kapitel 4 werden die implementierten Änderungen an L^{4}Linux und Fiasco dargestellt. Für die Kapselung von Linux bzgl. Busmaster-DMA werden in Abschnitt 4.3 Beispielimplementierungen für IDE-Controller und Netzwerkadapter vorgestellt.

Kapitel 5 enthält Ergebnisse von Performance-Messungen. Hierzu werden die Kosten für softwarebasierte IO-Adressräume ermittelt (Abschnitte 5.1 und 5.2) sowie die Zusatzkosten für ein Lock zur Synchronisation innerhalb des Linux-Kerns, das ohne Sperren der Interrupts auskommt (Abschnitt 5.4). In Abschnitt 5.3 wird der Einfluss verschiedener Systemkonfigurationen auf die Performance des gekapselten Linux-Systems untersucht.

In Kapitel 6 erfolgt eine Zusammenfassung und ein Ausblick auf zukünftige Arbeiten.

Kapitel 2

Grundlagen und Stand der Technik

Dieses Kapitel ist wie folgt aufgebaut: Nachdem im folgenden Abschnitt grundlegende sicherheitstechnische Begriffe erläutert werden, wird im Abschnitt 2.2 genauer dargestellt, dass Standard-Betriebssysteme auch auf Systemen mit speziellen Anforderungen wiederverwendet werden können und wie diese gekapselt werden müssen. Abschnitt 2.3 konzentriert sich auf die Nizza-Architektur, in deren Kontext diese Arbeit entstanden ist. Im darauf folgenden Abschnitt 2.4 werden performancerelevante Aspekte der Virtualisierung beleuchtet. Schließlich behandelt Abschnitt 2.5 verschiedene Techniken bei der Ein-/Ausgabe von Daten mit Hilfe von Betriebssystemen und die dabei auftretenden Probleme.

2.1 Begriffsbestimmungen

Ein Standard-Betriebssystem ist Teil eines informationstechnischen (IT-) Systems, also eines Systems zur Verarbeitung von *Informationen*. Diese entstehen durch Interpretation von *Daten* mit Hilfe einer Beschreibung [98]. Die Unterscheidung zwischen Daten und Informationen ist abhängig vom Betrachtunspunkt: Das E-Mail-Programm extrahiert die Information einer verschlüsselten E-Mail, den Daten, durch Anwendung des passenden Schlüssels, der Beschreibung. Die extrahierten Informationen stellen auch für das E-Mail-Programm nur Daten dar, die für ein Präsentationsprogramm zu darstellbaren Informationen werden, indem die Daten einem Grafikformat folgend interpretiert werden. Aus Sicht des Nutzers arbeiten Programme als Teil von IT-Systemen allerdings immer mit Daten, deren Information sich erst mittels der im menschlichen Gehirn gespeicherten Beschreibung erschließt.

Informationen sind im allgemeinen Sinne schützenswert, und zwar in Bezug auf unbefugten Zugriff, unbefugte Modifikation und unbefugte Beeinträchtigung der Funktionalität. Daraus lassen sich folgende Schutzziele für IT-Systeme ableiten [37]:

Vertraulichkeit (*Confidentiality*) Nur autorisierte Nutzer (Entitäten, Principals etc.) haben Zugriff auf bestimmte Informationen. Die Einhaltung dieses Schutzziels gegenüber unbefugten Nutzern kann entweder durch Kontrolle des Zugriffs auf die Interpretationsvorschrift der Daten, die nicht interpretierten Daten oder die interpretierten Daten durchgesetzt werden.

Integrität (*Integrity*) Entweder ist die Information aktuell, korrekt und vollständig, oder es ist für den Nutzer möglich, das Nichtvorhandensein einer dieser Eigenschaften zu erkennen. In der Literatur [22] wird dieser Begriff häufig so definiert, dass eine Veränderung bzw. Zerstörung der Daten (und damit der Information) ohne Autorisierung nicht möglich ist. Dies überschneidet sich jedoch mit dem Begriff *Verfügbarkeit*. Integrität im Sinne dieser Definition wird üblicherweise durch Signierung sichergestellt.

Verfügbarkeit (*Availability*) Informationen sind verfügbar, wann und wo ein autorisierter Nutzer diese benötigt. Dies kann beispielsweise mittels Redundanz sichergestellt werden.

In den folgenden Abschnitten wird der Begriff „Daten“ allgemein im Sinne von „durch IT-Systeme zu verarbeitende Informationen“ gebraucht.

2.2 Wiederverwendung von Standard-Betriebssystemen

2.2.1 Einsatzgebiete

In der Einleitung wurden bereits Gründe für die Wiederverwendung von Standard–Betriebssystemen dargestellt. Folgende Einsatzgebiete lassen sich dabei unterscheiden:

- Das Betriebssystem wird als *Service-OS* benutzt, das vertrauenswürdigen Komponenten bestimmte Dienste zur Verfügung stellt. Das Service-OS kann ein bestimmtes Gerät ansteuern oder den Dienst eines kompletten Protokoll-Stacks anbieten. Vertraulichkeit gegenüber anderen Prozessen außerhalb des Service-OS wird hier durch starke Kapselung und Verschlüsselung der Daten erreicht, Integrität durch Signieren der Daten und Verfügbarkeit durch Redundanz.

- Das komplette Betriebssystem kann innerhalb einer Domain eines bestimmten Sicherheitsniveaus als Teil einer *Multilevel-Security*-Architektur (MLS) [82] ausgeführt werden. MLS-Systeme beschränken den Datenfluss zwischen Domains gemäß eines bestimmten Modells, beispielsweise nach dem Bell-LaPadula-Modell [6]: Eine Domain darf nicht lesend auf Daten einer Domain mit höherem Sicherheitsniveau (*no read up*) zugreifen, ebenso nicht schreibend auf Daten einer Domain mit niedrigerem Sicherheitsniveau (*no write down*)[1]. Diese Anwendung unterscheidet sich vom Szenario des Service-OS insofern, dass hier genau festgelegt ist, auf welche Daten eine Domain in welcher Weise Zugriff bekommt. Eine Verschlüsselung der Daten ist bei diesem Anwendungsszenario normalerweise nicht notwendig.

- Ein zu MLS erweiterter Ansatz wird mit der µSINA-Architektur [29, 31] verfolgt: Zwei nicht vertrauenswürdige Instanzen eines Netzwerk-Stacks werden isoliert voreinander auf einem Mikrokern ausgeführt. Dabei ist ein Stack per Netzwerkadapter mit der Außenwelt (der „unsicheren“ Seite) verbunden und verarbeitet nur verschlüsselte Daten. Der andere Stack auf der „sicheren“ Seite kommuniziert per Netzwerkadapter nur mit einem internen Firmen-Netz und verarbeitet unverschlüsselte Daten. Der Datenaustausch zwischen beiden Seiten erfolgt ausschließlich verschlüsselt durch ein „Viadukt“, eine kleine vertrauenswürdige Anwendung. Als Grundlage für beide Netzwerk-Stacks inklusive der Netzwerktreiber dient Linux.

- Server-Konsolidierung: Auf einem Server werden viele Instanzen des gleichen Betriebssystems ausgeführt. Zwischen den Instanzen gibt es entweder keine Kommunikation, oder die Kommunikation erfolgt über genau definierte Kanäle, z. B. über ein virtuelles Netzwerk. Die Instanzen sind durch starke Kapselung voreinander geschützt, so dass die Fehlfunktion bzw. der Absturz einer Instanz die anderen Instanzen nicht beeinflusst.

[1] Tatsächlich wird im Modell noch zwischen *Überschreiben* und *Anfügen* von Daten unterschieden. Überschreiben ist nur für Daten auf gleichem Sicherheitsniveau erlaubt.

Alle dargestellten Anwendungsszenarien setzen voraus, dass die nicht vertrauenswürdigen Komponenten innerhalb einer starken Kapsel ausgeführt werden. Gründe dafür werden im folgenden Abschnitt genannt.

2.2.2 Gründe für die Kapselung von Standard-Betriebssystemen

Monolithische Betriebssysteme sind nicht vertrauenswürdig Sicherheitsarchitekturen bestehen üblicherweise aus wenigen vertrauenswürdigen Komponenten mit geringem Codeumfang [27]. Geringe Codegröße und -komplexität verringert die Wahrscheinlichkeit von Fehlern und bildet die Grundlage für Programm-Evaluation bis hin zur Programm-Verifikation [36].

Heutige Standard-Betriebssystem-Kerne sind sehr umfangreich: So besitzt der Linux-Kern 2.6.8.1 mehr als 3 Millionen Code-Zeilen (*Source Lines of Code, SLOC*). Davon entfallen mehr als 2 Millionen SLOC auf Gerätetreiber.[2] Mit steigender Codegröße erhöht sich der Anteil an Programmierfehlern und potenziellen Sicherheitslöchern. Gerätetreiber haben eine nachweislich schlechtere Qualität als andere Teile von Kernen [12].

Die Wahrscheinlichkeit von Programmierfehlern in Standard-Betriebssystemen ist auch deshalb hoch, weil die Kerne meist in den Programmiersprachen C bzw. C++ implementiert sind, die aus Sicht sicherer Systeme viele Nachteile besitzen [50]. Im Gegensatz dazu würden spezielle Sprachen für Gerätetreiber wie DEVIL [69] die automatische Überprüfung von sicherheitskritischen Eigenschaften des Codes zur Compilationszeit ermöglichen. Sichere Sprachen wie Modula oder Lisp erlauben es nicht, auf Objekte ohne deren Namen zuzugreifen, genau dies ist aber unter C/C++ möglich.

Die Entwicklung des Linux-Kerns verläuft sehr dynamisch, und es werden immer wieder Sicherheitslöcher im Kern entdeckt. Es ist sogar denkbar, dass ein Angreifer unbemerkt bösartigen Code im Kern versteckt, dessen Existenz aufgrund der Codegröße nicht unbedingt auffallen muss – entsprechende Versuche gab es bereits [3]. Mit steigender Codegröße steigt die Wahrscheinlichkeit, dass solche Angriffe nicht entdeckt werden. Eventuell möchte man auch dem Hersteller eines Betriebssystems nicht vertrauen, da dieser z. B. unter (teilweiser) Kontrolle bestimmter politischen Interessengruppen stehen könnte.

Die rapide Entwicklung auf dem Gebiet der Hardware zwingt zudem zu ständigen Updates der Gerätetreiber im Kern. Ein Anwender/Administrator kann sich nicht immer sicher sein, dass ein Treiber-Update aus einer vertrauenswürdigen Quelle stammt. Gleichzeitig kann oder möchte der Nutzer nicht auf die Verwendung aktueller Hardware und Anwendungen verzichten und ist deshalb auf die neuesten und damit vergleichsweise fehleranfälligen Versionen der monolithischen Betriebssysteme angewiesen.

Standardanwendungen sollen die Vorhersagbarkeit des Gesamtsystems nicht beeinflussen Bösartige oder fehlerhafte Anwendungen können, mit ausreichenden Rechten ausgestattet, die Zustellung von Interrupts unterbinden und damit preemptives Scheduling einschränken.

Eine Anwendung kann vertrauenswürdig sein und trotzdem das System so kompromittieren, dass zeitliche Zusagen nicht mehr eingehalten werden können, wenn die äußere Umgebung keine ausreichende Kontrolle von Ressourcen ermöglicht. Eine Linux- oder Windows-XP-Anwendung

[2] Diese Zahlen wurden mit `sloccount` 2.26 [95] für die relevanten Quelldateien der x86-Architektur ermittelt.

kann beispielsweise niemals harte Echtzeitzusagen garantieren, weil der Kern selbst keine Garantie für die Ausführung von Systemaufrufen in vorbestimmter Zeit geben kann. Die Ausführung eines Systemaufrufes kann andere Aktivitäten beeinflussen.

Programme mit zeitlichen Anforderungen müssen deshalb isoliert von Standardanwendungen ausgeführt werden, und deren Ressourcen müssen beschränkt werden.

2.2.3 Grundlegende Arten der Kapselung

Der stärkste Anspruch wird an eine Kapselung gestellt, wenn das zu kapselnde Objekt als potenziell bösartig betrachtet wird. Bösartiger Code kann z. B. versuchen, das System oder einzelne Anwendungen gezielt zu kompromittieren, Verfügbarkeit von Ressourcen einzuschränken oder Informationen auszuspähen (Verletzung der Vertraulichkeit). Nicht immer ist es erforderlich, diesen schlimmsten Fall zu berücksichtigen.

Hier betrachtete Standard-Betriebssysteme sind nicht ohne weiteres bösartig, sondern bilden aufgrund des monolithischen Aufbaus potenzielle Ziele für Angriffe derart, dass das System vollständig penetriert wird: Durch Ausnutzung einer Lücke im Kern erhält ein Angreifer alle Rechte, mit denen der Kern ausgeführt wird, und damit meist direkten Zugriff auf die Hardware und sensible Daten.

In den folgenden Abschnitten werden verschiedene Techniken der Kapselung vor allem unter den in Abschnitt 1 dargestellten Zielen dieser Arbeit betrachtet und qualitativ verglichen. Unter anderem soll jeweils die Stärke der Kapselung untersucht werden. In Abhängigkeit davon, ob die Technik Schutz gegen bösartigen Code bietet oder nicht, wird die Stärke der Kapselung als „stark“ oder „schwach“ bezeichnet.

Separate Schutz-Domänen

Nooks sind Subsysteme für die Kapselung von Gerätetreibern in Betriebssystem-Kernen [86]. Treiber laufen in so genannten *leichtgewichtigen Schutz-Domänen* in separaten Adressräumen, nutzen aber teilweise gemeinsame Datenstrukturen mit dem Kern und sind mit den Rechten des Kerns ausgestattet. Auf Kern-Datenstrukturen können Treiber nur lesend zugreifen. Die Kommunikation mit dem Kern erfolgt über spezielle Wrapper-Funktionen, die die Parameter überprüfen und übersetzen sowie den Adressraum umschalten. Die Implementierung wurde für den Linux-Kern vorgenommen, die Idee lässt sich aber auch auf andere Systeme übertragen.

Das Nooks-Subsystem ist primär für den Schutz vor Programmierfehlern in Treibern entworfen worden. Ein Treiber, der auf ungültige Adressen zugreift, wird neu gestartet. Die Nooks-Technik bietet keinen Schutz vor bösartigem Code und nicht autorisierten Datentransaktionen mittels Busmaster-DMA (siehe Abschnitt 2.5.3). Da die Nooks-Schutz-Domänen mit Kern-Rechten ausgestattet sind, können sie durch Aufsetzen eines eigenen Seitenverzeichnisses die Adressraumseparierung überwinden. Ferner müssen die Treiber dem monolithischen Linux-Kern vertrauen.

Die Nooks-Technik ist daher als schwache Kapselung einzuordnen.

Spezielle Virtuelle Maschinen

Der *Denali-Isolation-Kernel* [96] stellt Gast-Betriebssystemen eigene virtuelle Maschinen in eigenen Adressräumen zur Verfügung, die von einem VM-Monitor (*Virtual Machine Monitor*) kontrolliert werden. Im Unterschied zur vollständigen Virtualisierung werden hier aber viele Eigenschaften

der Hardware nicht der virtuellen Maschine offeriert bzw. zusätzliche Eigenschaften hinzugefügt (z. B. virtuelle Register). Probleme bei der Virtualisierung der x86-Architektur (siehe Abschnitt 2.2.3) werden ignoriert bzw. vereinfacht in der virtuellen Maschine emuliert.

Jeder Gast verfügt über nur *einen* Adressraum. Denali emuliert einige Standard-Geräte (virtueller Netzwerkadapter, persistente virtuelle Festplatten, Ein-/Ausgabegeräte). Viele Geräte werden auf realer Hardware mittels einer Abfolge von IO-Instruktionen programmiert, die bei einer vollständigen Virtualisierung wie unter VMware einen großen Overhead erzeugen, da jeder Zugriff auf IO-Ports emuliert werden muss und viele Zugriffe davon einen Wechsel in den VM-Monitor zur Folge haben [84]. Aus diesem Grund bietet Denali für derartige Geräte (z. B. den Interrupt-Controller) eine vereinfachte Schnittstelle.

Das Einsatzgebiet von Denali liegt in der gleichzeitigen Ausführung einer Vielzahl von nicht vertrauenswürdigen Internet-Diensten und nicht in der Unterstützung von Standard-Betriebssystemen. Mit dem Denali-Isolation-Kernel kann starke Kapselung erreicht werden.

Vollständige Virtualisierung

Bei der vollständigen Virtualisierung ist das gekapselte System Gast in einer separaten virtuellen Maschine (VM). Die VMs werden von einem VM-Monitor überwacht, der entweder als Anwendung auf einem Host-Betriebssystem ausgeführt wird oder selbst ein minimales Betriebssystem enthält und direkt auf der Hardware läuft (z. B. VMware ESX [94]).

Bei der erstgenannten Technik wird der VM-Monitor entweder auf einem herkömmlichen monolithischen Betriebssystem ausgeführt (z. B. VMware GSX [93] auf Linux oder Windows) oder auf einem *Hypervisor* (z. B. IBMs sHype [79]), der, ähnlich wie ein Mikrokern, eine schlanke Abstraktion der Hardware anbietet und nur die notwendigsten Mittel zur Trennung von Gästen implementiert. Der Zugriff auf physische Geräte erfolgt in diesem Fall üblicherweise mittels einer privilegierten virtuellen Maschine, die direkten Zugriff auf die Hardware erhält.

Der VM-Monitor stellt für die Gäste virtuelle Geräte zur Verfügung, die gebräuchlichen Standard-Geräten nachgebildet sind. Der Gast besitzt dafür mit hoher Wahrscheinlichkeit Treiber. Bei der Kommunikation mit der realen Hardware entstehen zusätzliche Kosten, weil der VM-Monitor eine Abbildung von der virtuellen auf die tatsächliche Hardware vornehmen muss. Ferner lassen sich aufgrund der notwendigen Abstraktion nicht alle Eigenschaften der physischen Hardware mit vertretbarem Aufwand in den virtuellen Geräten nachbilden.

Die Kommunikation zwischen virtuellen Maschinen erfolgt über virtuelle Geräte. Dies ist aus Sicht des Gastes eine elegante Lösung, weil der Gast über Standard-Geräte (z. B. Netzwerkadapter) mit anderen virtuellen Maschinen und mit der Außenwelt über Standardprotokolle, z. B. TCP/IP kommunizieren kann. Sobald der VM-Monitor Treiber für ein bestimmtes physisches Gerät besitzt, kann auch der Gast dieses Gerät benutzen. Allerdings bildet ein virtuelles Gerät eine sehr umfangreiche Schnittstelle, was diesen Ansatz aus sicherheitstechnischer Sicht kontraproduktiv erscheinen lässt.

Virtualisierung von Betriebssystemen wird schon lange erfolgreich eingesetzt. IBMs VM [14, 66] als eines der ältesten Beispiele ist auf die spezielle IBM-360-Architektur angewiesen. Neuere Arbeiten beschäftigen sich mit der Virtualisierung auf Standard-Hardware, z. B. VMware [84], Disco [9] und Terra [21].

Limitationen Herkömmliche x86-Systeme lassen sich aufgrund bestimmter Eigenschaften der Hardware nur mit großem Aufwand vollständig virtualisieren [76]:

- Der Prozessor löst beim Ausführen des `popf`-Befehles, der durch Löschen des Interrupt-Flags die Interrupts sperren kann, keine Exception aus, sondern die Änderung des Interrupt-Flags wird einfach ignoriert [46]. Der VM-Monitor erkennt somit kritische Abschnitte nicht, in denen der Gast nicht unterbrochen werden möchte. Analog erkennt der VM-Monitor nicht das Ende eines kritischen Abschnitts, wenn durch Ausführung des `popf`-Befehls die Zustellung von Interrupts wieder erlaubt werden sollte.

- Bestimmte Befehle zum Auslesen von Statusinformationen des Prozessors liefern im nicht privilegierten Modus Informationen über die tatsächliche Privilegstufe. Ein Gast kann daran erkennen, dass er nicht im privilegierten Modus ausgeführt wird. Als Konsequenz könnte sich der Gast anders verhalten oder seine Arbeit einstellen.

- Der TLB auf x86-Systemen wird automatisch durch die Hardware gefüllt. Dies erschwert die Virtualisierung gegenüber einer Implementierung in Software (zu finden z. B. bei Alpha, MIPS und SPARC), da bei jeder Änderung an der Seitentabelle durch den Gast diese vollständig validiert werden muss.

Eigenheiten dieser Art bedingen zum Teil umfangreiches Scannen und Patchen des Gast-Codes vor der Ausführung und vergrößern zwangsläufig die Basis an vertrauenswürdigem Code.

Eine vollständige Virtualisierung führt zu einer starken Kapselung des Gastes, da der Gast vollständig unter Kontrolle des VM-Monitors und ohne direkten Zugriff auf physische Geräte ausgeführt wird. Der Nachteil der vollständigen Virtualisierung sind der große Umfang an vertrauenswürdigem Code (der VM-Monitor und das darunter liegende Betriebssystem inklusive dessen Gerätetreiber) sowie die hohen Anforderungen an die Hardware.

Para-Virtualisierung

Vollständige Virtualisierung ist nicht immer möglich oder sinnvoll. Besonders für schwer zu virtualisierene Hardware kann es vernünftiger sein, das Gast-Betriebssystem so zu verändern, dass es besser mit dem VM-Monitor zusammenarbeiten kann.

Aktuelle Beispiele für diese Technik sind L^4Linux [25] und Xen [5]. Anstatt, wie im vorherigen Abschnitt beschrieben, den Gast-Code nach Auftreten von `popf`-Befehlen zu scannen, wird dieser Befehl im *Quellcode* durch Aufruf einer entsprechenden Funktion des VM-Monitors ersetzt. Zugriffe auf geschützten Speicher und Spezialregister erzeugen im nicht privilegierten Prozessormodus eine Schutzverletzung, die normalerweise vom VM-Monitor emuliert werden muss. Das Ersetzen des Codes durch direkte Aufrufe des VM-Monitors kann beträchtlichen Emulationsaufwand ersparen.

Ein VM-Monitor eines vollständig virtualisierten Systems besitzt meist einen wesentlich größeren Umfang an Code als der VM-Monitor eines para-virtualisierten Systems. Ersterer bietet dem Gast die komplette Hardware (bzw. eine spezielle Ausprägung davon) als Schnittstelle an, während bei der Para-Virtualisierung oft eine viel abstraktere und schlankere Schnittstelle an den Gast exportiert wird. Von dieser Schnittstelle hängt ab, wie stark die Änderungen des Gastes ausfallen und wie leicht sich diese auf neue Versionen des Gast-Kerns übertragen lassen.

Grundlage von L^{4}Linux sind Mikrokerne mit L4-Schnittstelle. Der Codeumfang liegt etwa zwischen 10 KB und 100 KB (siehe Abschnitt 2.3). Der Xen-VM-Monitor bietet eine etwas breitere Schnittstelle als L4-Mikrokerne und hat eine Größe im Bereich von etwa 300-500 KB (aktuelle Entwicklerversion 3.0). Der Linux-Kern 2.6.10 hat bereits in der Standard-Konfiguration für die x86-Architektur einen Umfang von mehreren Megabyte.

Auch Exokernel [18] können die Grundlage für Para-Virtualisierung bilden. In der Komplexität liegen sie zwischen Mikrokernen und VM-Monitoren für vollständige Virtualisierung. Exokernel exportieren ein relativ genaues Abbild der Hardware in die virtuelle Maschine, überlassen die Verwaltung von Ressourcen aber einer nicht vertrauenswürdigen Emulationsschicht im Nutzermodus.

Mit Para-Virtualisierung lässt sich eine vollständige und starke Kapselung des Gastes erreichen. Im Vergleich mit der vollständigen Virtualisierung ist der Umfang an vertrauenswürdigem Code geringer, weil der Emulationsaufwand des VM-Monitors sinkt. Para-Virtualisierung setzt jedoch immer das Vorhandensein des Quelltextes voraus.

Real-Time Linux

Die bisher aufgezählten Techniken zur Kapselung von Standard-Betriebssystemen haben vor allem zum Ziel, nicht vertrauenswürdigen Code so zu kapseln, dass vertrauenswürdige Anwendungen in Bezug auf Integrität und Vertraulichkeit nicht beeinträchtigt werden.

RTLinux [100, 101] implementiert eine Erweiterung des Linux-Kerns, die es ermöglicht, Programme mit zeitlichen Anforderungen gleichzeitig mit normalen nicht echtzeitfähigen Linux-Programmen auszuführen. Um harte zeitliche Zusagen garantieren zu können, wurden bestimmte Komponenten des Linux-Kerns durch eine *Real-Time-Executive* bestehend aus Zeitbasis, Real-Time-Scheduler und Kommunikationsmitteln ersetzt. Der Linux-Scheduler wird dem Real-Time-Scheduler untergeordnet, so dass Linux als Echtzeitprozess mit niedrigster Priorität ausgeführt wird.

Der Linux-Kern ist, wie unter Standard-Linux, mit Systemprivilegien ausgestattet. Echtzeitanwendungen bestehen aus zwei Teilen, von denen der zeitkritische Teil als Kernmodul an den Linux-Kern gelinkt und mit dessen Rechten ausgeführt wird. Der andere Teil der Anwendung ist als normales Linux-Programm implementiert, das mit dem Kernmodul asynchron über Shared Memory oder über nicht blockierende FIFO-Puffer kommuniziert.

Prozessor-Interrupts sind entweder an Echtzeitanwendungen oder an den Linux-Kern gebunden. Wenn der Linux-Kern die Interrupts für Synchronisationszwecke sperren möchte, erfolgt dies nicht am Interrupt-Controller, sondern die Zustellung der Interrupts an Linux wird von der RT-Executive unterbrochen. Treten während der Sperrung für den Linux-Kern relevante Interrupts auf, werden diese Ereignisse gespeichert und später zugestellt. Durch diese Implementierung wird vermieden, dass Linux Interrupts sperren kann, die Echtzeit-Tasks oder dem Echtzeit-Scheduler zugewiesen sind.

RTLinux zeichnet sich durch eine sehr gute Vorhersagbarkeit für die Reaktion auf Ereignisse (*Predictability*) aus. Echtzeitanwendungen und Linux-Kern müssen einander aber gegenseitig vertrauen, da der Zugriffsschutz zwischen Linux-Kern und Echtzeitprozessen nur auf Software-Ebene implementiert ist und Echtzeitprogramme praktisch als Teil des Kerns ausgeführt werden. Der Linux-Kern wird nur in Bezug auf die Ressource CPU gekapselt, alle anderen Ressourcen (Speicher, Zugriff auf Geräte) werden kooperativ zwischen Linux-Kern und Echtzeitprozessen aufgeteilt.

Zusammenfassung

In Tabelle 2.1 sind die gezeigten Arten der Kapselung anhand ihrer Eigenschaften aufgezählt. Die Para-Virtualisierung wird in dieser Arbeit aufgrund der überwiegenden Vorteile genauer untersucht – konkret anhand der L^{4}Linux-Implementation.

	Stärke der Kapselung	Hardware-Anforderungen	Kosten	Quellcode-Änderungen	Umfang TCB
Separate Schutz-Domänen	schwach	gering	mittel	wenig	groß
Spezielle Virtuelle Maschinen	stark	gering	mittel	groß[a]	klein
Vollständige Virtualisierung	stark	hoch	hoch	keine	groß
Para-Virtualisierung	stark	gering	klein bis mittel	wenig	klein
Real-Time Linux	keine	keine	gering	wenig	groß

[a] Spezielle Anwendung für die virtuelle Maschine

Tabelle 2.1: *Vergleich von Kapselungstechniken für nicht vertrauenswürdigen Code anhand der Merkmale aus Kapitel 1.* Der Einsatz in einer Echtzeitumgebung ist prinzipiell mit allen Techniken möglich.

2.2.4 Verbesserte Kapselung durch Hardware-Erweiterungen

Für die separierte Ausführung von Prozessen sind mindestens zwei Privilegstufen notwendig: Im *Systemmodus* wird Code mit speziellen Rechten ausgeführt, der im *Nutzermodus* ausgeführte Aktivitäten mit eingeschränkten Rechten verwaltet und die Kommunikation zwischen diesen ermöglicht. Vor allem aus Gründen der Performance führen aktuelle Standard-Betriebssysteme einen relativ umfangreichen (monolithischen) Kern im privilegierten Modus aus, der außer für die Verwaltung von Adressräumen und Aktivitäten auch für die Ansteuerung der Hardware verantwortlich ist und eine Vielzahl von Protokollen implementiert. In Abschnitt 2.2.2 wurde bereits dargestellt, dass monolithische Betriebssysteme nicht vertrauenswürdig sein können. Neue Sicherheitsarchitekturen sollen diesem Fakt Rechnung tragen.

Microsofts *Next Generation Secure Computing Base* (NGSCB) [11, 13] und Intels *LaGrande Technology* (LT) [44] führen jeweils einen neuen Modus mit neuen Privilegstufen (Microsoft: *Right Hand Side* (RHS) oder *Nexus Mode*, Intel: *Protected Partition* bzw. *Root Partition*) ein, der gegenüber dem normalen Systemmodus mit speziellen Rechten ausgestattet sind. Innerhalb dieser neuen Privilegstufen wird ein spezieller Kern (Microsoft: *NGSCB Nexus*, Intel: *Protected OS Kernel* oder *Domain Manager*) ausgeführt. Nachfolgend wird dieser spezielle privilegierte Modus als Root Mode und der mit diesen Rechten ausgestattete Kern als Domain Manager bezeichnet.

Beide Ansätze haben zum Ziel, heutige Standard-Betriebssysteme mit möglichst wenigen Änderungen auszuführen und durch Auslagerung bestimmter sicherheitsrelevanter Funktionen in den

Domain Manager die Systemsicherheit zu erhöhen. Der Domain Manager erhält exklusiven Zugriff auf bestimmten Speicher und soll vor allem den Zugriff auf sicherheitsrelevante Informationen kontrollieren. Die Einführung des Domain Managers ermöglicht auch, mehrere Standard-Betriebssysteme isoliert im normalen Systemmodus auszuführen. Als Unterstützung implementiert Intel in zukünftiger Hardware mit der *Vanderpool Technology* (VT) [49] Erweiterungen, um die in Abschnitt 2.2.3 auf Seite 11 beschriebenen Anomalien bei der Virtualisierung auf der heutigen x86-Architektur zu beseitigen.

Die ARM TrustZone-Architektur [2] definiert ähnliche Erweiterungen wie NGSCB und LT, bezieht sich aber vorrangig auf eingebettete Systeme. ARM führt ein so genanntes *Security Bit* in alle wichtigen Komponenten der Hardware (CPU, Speicher, bestimmte Geräte) ein, um eine Partitionierung in *gesicherte* und *nicht gesicherte* Daten zu ermöglichen.

Bisher war es auf Standard-Hardware nur mit großem Aufwand möglich, Gäste vollständig zu virtualisieren (siehe die in Abschnitt 2.2.3 gezeigten Limitationen). Mit den neuen Eigenschaften zukünftiger Hardware wird dies wesentlich vereinfacht. Ob damit vollständige Virtualisierung ohne Einschränkung und mit vertretbaren Kosten möglich ist, muss in zukünftigen Arbeiten untersucht werden. Mit der Einführung der neuen Hardware insbesondere für die x86-Architektur ist erst in einiger Zeit zu rechnen. Ferner sind genaue Spezifikationen erst in Teilen verfügbar (Stand Januar 2005). Für die vorliegende Arbeit konnten diese Erweiterungen der Hardware deshalb nicht berücksichtigt werden.

2.2.5 Nicht vertrauenswürdige Dienste in vertrauenswürdiger Umgebung

Die Frage nach Integrität und Verfügbarkeit bei nicht vertrauenswürdigen Diensten wird von Li et. al. im Projekt SUNDR [61] genauer behandelt. Dort wird gezeigt, wie Daten auf einem entfernten Datei-Server gespeichert werden können, ohne dass Klienten diesem Server vertrauen müssen.

Ein Client kann durch Vergleich von Hash-Summen überprüfen, ob die gesuchte Datei noch aktuell ist und ob unautorisierte Veränderungen vorgenommen wurden. Diese Lösung arbeitet mit zwei unabhängigen Servern: Der Block-Server speichert Dateien, die durch Hash-Werte indiziert werden. Der Konsistenz-Server stellt sicher, dass jeder Client immer die zuletzt geschriebene Version sieht. Beide Server müssen nicht vertrauenswürdig sein.

Ähnliche Techniken lassen sich verwenden, um Integrität bei der Verarbeitung von Daten durch ein Service-OS sicherzustellen. Besitzt das nicht vertrauenswürdige Service-OS freien Zugang auf nicht vertrauenswürdige Kanäle (z. B. die „unsichere Seite“ in µSINA, siehe Abschnitt 2.2.1), müssen Daten, die dieser Dienst verarbeitet, generell verschlüsselt werden (Sicherstellung der Vertraulichkeit). Durch Überprüfung der Daten auf Konsistenz und die eventuell notwendige Verschlüsselung entstehen zusätzliche Kosten, die bei der Bewertung derartiger Systeme zu berücksichtigen sind.

In dieser Arbeit wurden diese Kosten nicht untersucht, da der Fokus vor allem auf der Untersuchung von Kapselungstechniken lag.

2.3 L4Linux / Nizza

Die vorliegende Arbeit ist im Kontext der Nizza-Architektur [27] entstanden, deshalb soll diese Architektur im folgenden näher beschrieben werden. Nizza basiert auf dem L4-kompatiblen [56] Mikrokern *Fiasco* [35].

2.3.1 Die Nizza-Architektur

Nizza besteht aus einer kleinen Anzahl vertrauenswürdiger *Server* – Prozesse in separaten Adressräumen, die bestimmte Dienste für andere Prozesse über eine definierte Schnittstelle zur Verfügung stellen (Abbildung 2.1).

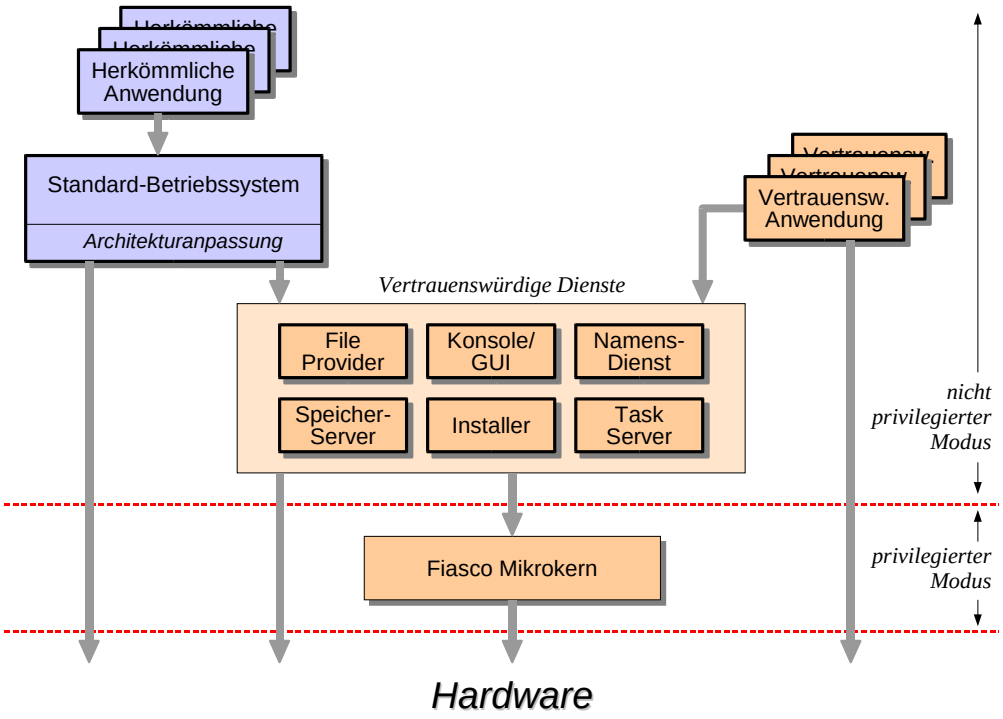


Abbildung 2.1: *Die Nizza-Architektur.* Ein Standard-Betriebssystem als Server für herkömmliche Anwendungen (*Legacy Applications*) läuft gekapselt im eigenen Adressraum im nicht privilegierten Modus. Adressraumgrenzen sind durch dicke Umrandungen gekennzeichnet. Vertrauenswürdige Komponenten sind orange, nicht vertrauenswürdige Komponenten sind blau dargestellt. Auch nur bedingt vertrauenswürdige Komponenten können direkten Zugriff auf *bestimmte* Geräte erhalten.

Durch die klare Trennung der Aufgaben auf verschiedene Server kann die Implementierung eines Servers leichter ausgetauscht werden, als dies bei einem monolithischen System möglich wäre. Es ist sogar denkbar, einen Server zur Laufzeit neu zu starten, wenn dessen Funktionsweise beeinträchtigt ist (Programmierfehler, Vireninfekt o.ä.). Darüber hinaus implizieren übersichtliche Schnittstellen eine bessere Software-Qualität, weil sich der Entwickler schon während des Entwurfsprozesses Gedanken über die Aufteilung der Funktionalität machen muss.

Nizza ermöglicht die gleichzeitige Ausführung von *vertrauenswürdigen* und *nicht vertrauenswürdigen* Anwendungen sowie von *Echtzeitanwendungen* neben herkömmlichen Anwendungen ohne zeitliche Anforderungen (*Legacy Applications*).

2.3.2 Der Fiasco-Mikrokern

Fiasco [35] implementiert nur die Mechanismen eines Betriebssystem-Kerns, die zwingend im privilegierten Prozessormodus ausgeführt werden müssen: Adressräume, Aktivitäten (Threads) und Kommunikation zwischen Adressräumen (Inter-Prozess-Kommunikation, IPC). IPC zwischen Threads findet immer synchron statt, d. h. der Sender einer IPC blockiert so lange, bis der Empfänger die übertragenen Daten empfangen hat. IPC besitzt folgende Ausprägungen:

- Für die Übertragung von kleinen Informationsmengen, die in den CPU-Registern Platz finden, kann eine optimierte Version der IPC, die so genannte *Short IPC* verwendet werden. Wenn bestimmte Seitenbedingungen erfüllt sind, wird eine Short IPC im *Fast Path* ausgeführt, der in Assembler implementiert ist [75]. Anderenfalls erfolgt die Ausführung im *Slow Path*. Die Implementierung der Short IPC in Assembler ist besonders schnell, allerdings nicht portabel und schwieriger zu pflegen als C++-Code. Die Unterscheidung in einen optimierten und einen nicht optimierten Pfad bedingt zusätzliche Kosten.

- Mittels *Long IPC* können virtuelle Seiten über Adressraumgrenzen hinweg eingeblendet oder größere Datenblöcke kopiert werden. Eine einzelne Seite lässt sich auch mittels Short IPC übertragen.

- Interrupts und Seitenfehler werden mittels IPC zugestellt.

Fiasco wurde vor allem im Hinblick auf gute *Echtzeiteigenschaften* entwickelt. Anhand der *L4RTL*-Implementierung konnten wir zeigen, dass Fiasco sehr kurze Reaktionszeiten auf externe Ereignisse ermöglicht [67]. L4RTL implementiert eine rudimentäre Emulationsschicht, die es ermöglicht, einfache RTLinux-Programme nach Neucompilierung auf Nizza auszuführen. Als Vergleich zwischen beiden Systemen wurde die maximale Interrupt-Latenz gemessen, die unter hoher Systembelastung erreicht werden kann.

Im Vergleich mit RTLinux lagen die maximal erreichbaren Verzögerungen von L4RTL auf Standard-Hardware in der gleichen Größenordnung: Auf einem Pentium 4 mit 1.6 GHz wurden unter RTLinux maximal 24 µs gemessen, auf Nizza 33 µs. Im Unterschied zu RTLinux werden unter Nizza Prozesse grundsätzlich in eigenen Adressräumen mit eingeschränkten Rechten im Nutzermodus ausgeführt (siehe Abschnitt 2.2.3).

Der Fiasco-Mikrokern unterstützt die L4-Schnittstellen Version 2 (*V.2*) [59] und Version *X.0* [60]. Die experimentelle Schnittstelle Version *X.2* [91] wird von Fiasco bisher nur teilweise implementiert. Grundlage dieser Arbeit bildet Fiasco mit der stabilen Schnittstelle V.2.

2.3.3 Verwaltung von Ressourcen

Mikrokerne mit L4-Schnittstelle stellen die Trennung von Ressourcen sicher, implementieren aber keinerlei Regeln für deren Verwaltung – diese Aufgabe übernehmen Server im Nutzermodus.

Speicherverwaltung

Unter L4 werden Adressräume hierarchisch durch *Pager* verwaltet [56]. Der initiale Adressraum σ_0 besitzt exklusiven Zugriff auf den gesamten physischen Adressraum der CPU (also auf den

Hauptspeicher und eingeblendeten Speicher von Geräten, siehe Abschnitt 2.5.3) und kann ihm zugeordnete Seiten – Abbildungen auf Kacheln des physischen Adressraumes – mittels IPC in andere Adressräume einblenden. Diese können ihrerseits Seiten an andere Adressräume weitergeben.

Jeder L4-Thread besitzt einen Pager, der Seitenfehler behandelt. Generell muss der Empfänger beim Empfang von Seiten kooperieren, indem er den Empfang der entsprechenden IPC zulässt. Da der Empfänger nicht überprüfen kann, auf welche Kacheln die Seiten abgebildet werden, muss er dem Sender in dieser Hinsicht vertrauen.

Seiten können mittels der *unmap*-Operation anderen Adressräumen entzogen werden. Die *Mapping-Datenbank* im Mikrokern stellt sicher, dass diese Operation transitiv über mehrere Adressräume ausgeführt werden kann.

Für die Vereinfachung der Speicherverwaltung wird in Nizza das Modell der *Dataspaces* [4] implementiert. Ein Dataspace ist ein abstrakter Container für unstrukturierte Daten und wird durch seine Nummer sowie durch den implementierenden Server, den *Dataspace-Manager*, eindeutig identifiziert. Der Dataspace-Manager implementiert Rechte auf einem Dataspace und legt damit unter anderem fest, welcher Prozess wie (lesend, schreibend) auf den Dataspace zugreifen darf. Der Eigentümer eines Dataspaces kann Rechte (inklusive der Eigentümerschaft) an andere Entitäten weitergeben.

Ein Dataspace kann an eine Region eines virtuellen Adressraumes angeschlossen werden. Der Zugriff darauf löst Seitenfehler aus, die vom Dataspace-Manager behandelt werden müssen. An einen Adressraum werden üblicherweise Dataspaces von unterschiedlichen Dataspace-Managern angeschlossen. Da immer nur ein Pager für einen Thread verantwortlich ist, muss ein Seitenfehler in Abhängigkeit von der virtuellen Adresse an den jeweils verantwortlichen Dataspace-Manager weitergeleitet werden. Der *Region-Mapper* ermittelt aus dem Seitenfehler die ID des Dataspaces sowie den Offset im Dataspace und sendet diese Information als Aufforderung zum Mappen der entsprechenden Seite an den zugehörigen Dataspace-Manager.

Vertrauensbeziehungen zwischen Pager und Client

Ein Pager kann einem Client jederzeit Seiten entziehen. Der Pager ist auch dafür verantwortlich, dass er Clients in separaten Adressräumen Abbildungen auf disjunkte Mengen von Kacheln zur Verfügung stellt, um zu verhindern, dass nicht vertrauenswürdiger Code im Kontext einer vertrauenswürdigen Anwendung ausgeführt wird bzw. eine nicht vertrauenswürdige Anwendung Zugriff auf Daten einer vertrauenswürdigen Anwendung erlangt (es sei denn, dies ist explizit erwünscht). Ein Client muss daher immer seinen Pagern vertrauen, die Seitenfehler auf vertrauenswürdige Daten (z. B. Sektionen für Code und statische Daten) behandeln. Ein Pager vertraut jedoch üblicherweise nicht seinen Clients.

Geräte führen DMA-Operationen auf dem physischen Adressraum aus (Abschnitt 2.5.2). Informationen über die physische Entsprechung einer Region im virtuellen Adressraum kann nur der Pager dieser Region, d. h. der Manager des entsprechenden Dataspaces liefern. Ist dieser nicht vertrauenswürdig, muss diese Information mit Hilfe *seiner Pager* validiert werden. Von einer nicht vertrauenswürdigen Anwendung besteht keine für andere Prozesse sichtbare vertrauenswürdige Rückbeziehung auf ihren Pager. Daher muss die Validierung beim vertrauenswürdigen Root-Pager σ_0 beginnen und aufwärts ausgeführt werden.

In Abbildung 2.2 ist ein Szenario mit einem vertrauenswürdigem Gerätetreiber dargestellt, der Dataspaces von verschiedenen Dataspace-Managern an seinen Adressraum angeschlossen hat.

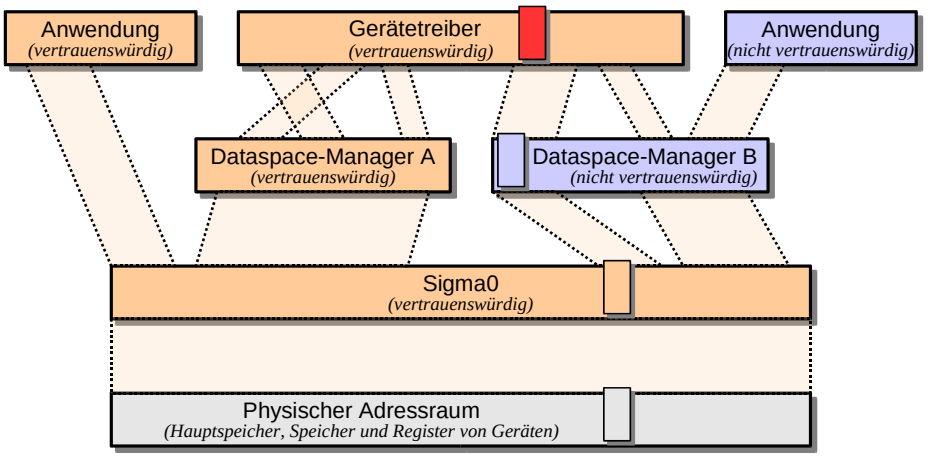


Abbildung 2.2: *Szenario mit hierarchichischen Pagern.* Für eine DMA-Operation soll die physische Adresse des rot markierten Bereiches ermittelt werden.

Der rot markierte Bereich im Treiber soll als Puffer für DMA-Operationen verwendet werden. Der Region-Mapper ermittelt den verantwortlichen Dataspace-Manager B. An ihn wird die Anfrage nach der physischen Adresse des Puffers gesendet. Da Dataspace-Manager B nicht vertrauenswürdig ist, muss die von ihm gelieferte Adresse validiert werden. Ausgehend vom Root-Pager σ_0 muss überprüft werden, welche Seiten an den jeweils nächsten Dataspace-Manager weitergegeben wurden. Im Beispiel ist σ_0 unmittelbarer Pager des Dataspace-Managers B.

Diese Validierung kann sehr aufwändig sein, da ausgehend vom Root-Pager potenziell viele Schritte durchlaufen werden müssen. In [51] setzen sich Kågström und Molin ausgiebig mit diesem Thema auseinander und beschreiben, wie die Kommunikation zwischen Dataspace-Managern mittels gemeinsamem Speicher und Software-TLBs optimiert werden kann.

Die Ermittlung von physischen Adressen ist nicht nur für DMA-Operationen notwendig, sondern beispielsweise auch für die Implementierung von Cache-Coloring (siehe Abschnitt 2.4.1).

Geräteregister, Interrupts und Task-Erzeugungsrechte

Der Zugriff auf Geräteregister erfolgt mittels programmierter IO über *memory mapped IO* oder über *IO-Ports* (siehe Abschnitt 2.5.1). Das Recht für den Zugriff auf beide Registerarten wird analog zu den Kacheln des Hauptspeichers durch Pager verwaltet.

Hardware-Interrupts und Task-Erzeugungsrechte sind immer an bestimmte Threads gebunden.

2.3.4 L4Linux

L4Linux [25] dient als Server für Standardanwendungen, die *neben* Anwendungen mit Echtzeit- bzw. Sicherheitsanforderungen ausgeführt werden. Der Code von L4Linux ist bis auf die Architekturanpassung mit Standard-Linux identisch. So manipuliert L4Linux beispielsweise Seitentabellen nicht direkt, sondern nutzt dazu Primitive des Mikrokerns. Linux-Anwendungsprogramme können ohne Änderung auf L4Linux ausgeführt werden. Die für L4Linux zur Verfügung stehenden Ressourcen werden von externen Servern verwaltet (siehe Abschnitt 2.3.3).

In dieser Arbeit wurde L^{4}Linux auf Basis von Linux 2.2.26 verwendet. In den folgenden Abschnitten werden daher für den Kontext dieser Arbeit wichtige Grundprinzipien von L^{4}Linux erläutert.

Ausführung im Nutzermodus

Die Durchsetzung eingeschränkter Rechte für Anwendungen wird üblicherweise durch die Wahl geeigneter Privilegstufen für Anwendungen und Kern erreicht. Der Betriebssystem-Kern mit maximalen Rechten läuft auf der höchsten Privilegstufe. Die Privilegstufe wird auch als *Current Privilege Level* (CPL) bezeichnet. Bei x86-Systemen sind insgesamt vier Privilegstufen verfügbar: CPL 0 mit den meisten Rechten bis CPL 3 mit den wenigsten Rechten (auch benannt als Ring 0 bis Ring 3). Der Kern läuft auf Ring 0 (Systemmodus), die Anwendungen im Nutzermodus auf Ring 3. Andere Architekturen verfügen meist nur über zwei Privilegstufen.

Bestimmte Befehle zeigen ein spezielles Verhalten in Abhängigkeit davon, auf welcher Privilegstufe sie ausgeführt werden. Die Instruktion zum Setzen des Seitenverzeichnisses benötigt Systemrechte, weil Seitentabellen die Voraussetzung für den Schutz des Kerns vor den Anwendungen und für die Trennung der Anwendungen untereinander bilden. Versucht eine Anwendung mit CPL$>$0 die Seitentabelle zu setzen, wird ein Ausnahmefehler (*Exception*) ausgelöst, der vom Kern behandelt werden muss.

Um ein Standard-Betriebssystem zu kapseln, liegt es nahe, den Kern komplett als Anwendung mit eingeschränkten Rechten auszuführen. Damit L^{4}Linux als Anwendung auf einem Mikrokern ausgeführt werden kann, mussten privilegierte Befehle des Linux-Kerns durch geeignete Mikrokern-Primitive bzw. Aufrufe von vertrauenswürdigen Servern ersetzt werden.

Die ursprüngliche Version von L^{4}Linux [25] wurde später erweitert, um es hinsichtlich der Anforderung an Ressourcen einzuschränken (Tamed L^{4}Linux [28]).

Systemruf-Emulation

Unter Standard-Linux wird der Linux-Kern zur Ausführung von Systemaufrufen üblicherweise durch Ausführung einer speziellen Instruktion betreten (auf x86-Systemen durch `int 0x80` oder mittels `sysenter`). Da der Linux-Kern in alle Adressräume eingeblendet ist, ist beim Übergang zwischen Nutzerprozess und Kern kein Adressraumwechsel notwendig.

Unter L^{4}Linux werden Linux-Nutzerprozesse direkt auf L4-Prozesse abgebildet. Der Linux-Kern wird als eigenständiger L4-Prozess im Nutzermodus in einem separaten Adressraum ausgeführt (siehe vorheriger Abschnitt) und dient als Server für die Linux-Prozesse. Führt ein solcher die privilegierte Instruktion zum Betreten des Kerns aus, wird eine Exception ausgelöst. Diese muss unter L4 V.2 vom auslösenden Prozess behandelt werden. Die Behandlungsroutine ruft mittels IPC den Linux-Server auf, der den Systemaufruf behandelt und das Ergebnis wiederum per IPC an den Linux-Prozess zurück liefert.

Diese Systemruf-Emulation kostet drei Kernein- und -austritte sowie zwei Adressraumwechsel (Linux: ein Kernein- und -austritt, kein Adressraumwechsel). Systemaufrufe werden unter Linux von der C-Library gekapselt. Durch Ersetzen der Instruktion `int 0x80` in dieser Bibliothek durch Trampoline-Code, der den Linux-Server direkt ohne Umweg über die Exception aufruft, würden sich die Emulationskosten um einen Kernein- und -austritt reduzieren.

L4 X.2 [91] bietet eine Möglichkeit, Exceptions auch über Adressraumgrenzen hinweg zuzustellen. Somit könnte die durch `int 0x80` hervorgerufene Exception direkt vom Linux-Server behandelt werden und die Emulation wäre genauso teuer, wie bei Verwendung von Trampoline-Code.

Auslagerung von Gerätetreibern

Sollen Anwendungen mit verschiedenen zeitlichen und sicherheitsrelevanten Anforderungen auf das gleiche Gerät zugreifen, müssen die Zugriffe synchronisiert und priorisiert werden. In diesem Fall ist eine Auslagerung des Gerätetreibers aus dem Betriebssystem-Kern in separate Adressräume notwendig, wobei die Anbindung an den Treiber über einen virtuellen Gerätetreiber (Stub) ermöglicht wird. Der Kern greift dabei als Client auf Dienste eines externen Servers zu und muss sich Aufträgen von Echtzeitanwendungen, die ebenfalls Dienste der Geräteserver nutzen, unterordnen [28, 26].

Die Isolation von Treibern kann auch zu einer verbesserten Systemsicherheit führen, wenn dieser nicht von mehreren Clients benutzt wird, da Treiber häufiger von Programmierfehlern betroffen sind als andere Komponenten [12]. Beispiele für die Auslagerung von Treibern aus L4Linux sind in [23, 30, 62, 68] zu finden. In einer aktuellen Arbeit [53] stellen LeVasseur et. al. eine allgemeine Umgebung für Gerätetreiber vor, die ebenfalls auf einem L4-Mikrokern [90] basiert.

Die Schnittstelle zum Datenaustausch zwischen Client und separiertem Gerätetreiber hat besonderen Einfluss auf die erreichbare Performance. Daten, die von Geräten mit geringer Bandbreite verarbeitet werden (z. B. Tastatur, Maus, Analog-Modem) können zwischen Adressräumen kopiert werden. Bandbreiten-intensive Geräte (z. B. Festplatten-Controller, Netzwerkadapter) übertragen Daten in den Hauptspeicher via Busmaster-DMA (Abschnitt 2.5.3). Die Übertragung von Daten zwischen Adressräumen findet dabei mittels Deskriptorlisten statt, die Zeiger auf physische Kacheln enthalten. Soll die Kommunikation zwischen Client und Server (Gerätetreiber) asynchron stattfinden, sind weitere Probleme zu lösen (Entzug von Speicher, zeitliche Synchronisation). Das *DROPS Streaming Interface* (DSI) [64] wird in Nizza z. B. für die Übertragung von Netzpaketen zwischen Linux-Server und ausgelagertem Netzwerktreiber verwendet [62].

Einen ähnlichen Ansatz wie L4Linux zum Auslagern von Gerätetreibern verfolgt das Flux OSKit [19]. Hier werden Treiber ohne Veränderungen wiederverwendet, indem universeller *Glue-Code* eine einheitliche Abstraktion für Ressourcen zur Verfügung stellt. So gekapselte Treiber werden entweder in eigenen Adressräumen oder in einem gemeinsamen Adressraum mit einer Anwendung ausgeführt.

Die Auslagerung von Treibern aus Standard-Betriebssystemen ist aufgrund der monolithischen Architektur vieler Betriebssystem-Kerne mit großem Aufwand verbunden. Oft sind die internen Schnittstellen nicht klar definiert und dokumentiert.

In Xen [5] wurde eine Schnittstelle für Geräte integriert, die viele der auftretenden Probleme in ähnlicher Art und Weise wie die auf einem Mikrokern basierende Nizza-Architektur löst [20]. Separierung der Adressräume wird durch den Xen VM-Monitor durchgesetzt. Ein dedizierter Gast hat privilegierten Zugriff auf die Hardware und stellt anderen Gästen eine geeignete Abstraktion dieser Dienste zur Verfügung. Die Kommunikation zwischen Treiber und Client erfolgt mit L4-ähnlichen Mechanismen mittels Deskriptorlisten auf gemeinsamem Speicher. Ereignisse werden über bidirektionale *inter-domain device channels* ausgetauscht.

2.4 Spezielle Aspekte der Virtualisierung

2.4.1 Cache als Ressource und Cache-Coloring

Caches stellen in heutigen Systemen eine wichtige Ressource dar. Aufgrund der wachsenden Differenz zwischen dem Datendurchsatz der CPU und den Zugriffszeiten des Hauptspeichers werden Caches immer größer und bilden einen maßgeblichen Anteil bei der Leistungssteigerung moderner Prozessoren. Caches sind üblicherweise in Zeilen gleicher Größe (*Cache-Lines*) aufgeteilt. Bei einem Zugriff auf ein Datenwort, für das der Cache keine Kopie des Hauptspeichers bieten kann (*Cache-Miss*), wird immer eine komplette Cache-Line aus dem Speicher geladen. Heutige Systeme enthalten meist mehrstufige Caches, wobei die folgenden Aussagen für alle Cache-Stufen gelten.

Durch geeignete Zuordnung von Code bzw. Daten auf verschiedene Bereiche des Caches können unterschiedliche Ziele verfolgt werden:

- Physisch indizierter Cache kann in Bereiche (Farben) unterteilt werden, die jeweils bestimmten Anwendungen zugeordnet sind. Cache-Coloring kann genutzt werden, um Echtzeitanwendungen und Anwendungen ohne zeitliche Anforderungen verschiedene Farben zuzuordnen. Pro Anwendung steht somit nicht mehr der volle Cache zur Verfügung. Weiterhin muss der Hauptspeicher im Verhältnis der Farben aufgeteilt werden. Ist beispielsweise der Einsatz von 16 Farben möglich, so wird der Hauptspeicher in 16 gleich große Bereiche unterteilt. Einer Anwendung, die eine der Farben verwendet, ist genau 1/16 des Hauptspeichers zugeordnet.

  Damit steht pro Anwendung durchschnittlich weniger Speicher und Cache zur Verfügung. Nicht-Echtzeitanwendungen können aber auch keine Cache-Misses bei Echtzeitanwendungen provozieren [58]. Dadurch verbessert sich die Vorhersagbarkeit der Echtzeitanwendungen, deren *Worst-Case Execution Time* (WCET) dann näher an der bestmöglichen Ausführungszeit liegt.

  Die Anzahl möglicher Farben liegt bei heutigen Systemen im Bereich von 8 bis 32, siehe dazu Anhang B.2.

- Die Performance *innerhalb eines Prozesses* kann gesteigert werden, indem die Arbeitsmenge des Prozesses so vollständig wie möglich im Cache gehalten wird. Je mehr Speicherzugriffe aus dem Cache befriedigt werden können, desto weniger muss auf den langsamen Hauptspeicher zugegriffen werden. Insbesondere sind Ping-Pong-Effekte zu vermeiden, bei denen Cache-Lines durch ungünstige Verteilung von Daten ständig neu geladen werden müssen. Optimierungen dieser Art werden normalerweise innerhalb von Anwendungen bzw. innerhalb des Betriebssystem-Kerns vorgenommen [99].

- Caches stellen als begrenzte Ressource eine potenzielle Grundlage für verdeckte Kanäle dar. Durch Zuordnung von verschiedenen Cache-Bereichen an verschiedene Anwendungen kann die Nutzung des Caches als verdeckter Kanal verhindert werden.

Die Verwaltung der Ressource Cache ist also sowohl aus Sicht von Anwendungen mit zeitlichen Anforderungen als auch aus Sicht von sicherheitskritischen Anwendungen relevant.

Die Verwaltung des Caches über Adressraumgrenzen hinweg ist nur dann möglich, wenn der Cache physisch indiziert wird, wie beispielsweise der L2-Cache auf x86-Systemen. Für die Durchsetzung von Cache-Coloring ist die Kontrolle der Abbildung von virtuellen Seiten auf physische Kacheln für *alle* Adressräume notwendig. Bei monolithischen Systemen muss Cache-Coloring daher im Kern implementiert werden. Auf L4-Systemen wird die Zuordnung von Kacheln an Anwendungen von Pagern im Nutzermodus kontrolliert (siehe Abschnitt 2.3.3). Cache-Coloring lässt sich damit hierarchisch im Nutzermodus implementieren.

Soll ein ganzer Betriebssystem-Kern in einer eigenen Cache-Partition ausgeführt werden, treten zusätzliche Probleme auf: In einigen Gerätetreibern von Linux werden Annahmen getroffen, dass der Kern auf zusammenhängendem physischem Speicher ausgeführt wird. Ein Beispiel ist der IDE-DMA-Treiber in Linux 2.6.8.1. Die Annahme ist korrekt, wenn Linux als Kern die volle Kontrolle über die Speicherverwaltung ausübt. Wird der Kern allerdings als Nutzeranwendung in einer eigenen Cache-Partition ausgeführt, so ergibt sich dagegen eine Abbildung des linearen virtuellen Kernspeichers auf viele nicht zusammenhängende physische Kacheln.

Treiber dieser Art müssen ersetzt werden, z. B. durch externe Treiber, die als virtuelle Geräte (Stubs) an den Kern angebunden werden (siehe Abschnitt 2.3.4).

2.4.2 Optimierung der Adressraumwechsel

Auf der x86-Architektur ist der Wechsel zwischen Adressräumen im Vergleich zu anderen Architekturen relativ teuer. Das Fehlen von Adressraum-IDs bei TLBs macht es erforderlich, diese komplett zu leeren, wenn das Seitenverzeichnis umgeschaltet wird. Das Setzen des Seitenverzeichnisses impliziert daher auf der x86-Architektur einen automatischen TLB-Flush. Ausgenommen davon sind TLB-Einträge für Seiten, die in alle Adressräume eingeblendet sind (üblicherweise für den Betriebssystem-Kern). Die entsprechenden Seitentabelleneinträge enthalten ein *Global*-Bit, welches das automatische Invalidieren des TLBs für diese Seiten verhindert [47].

Aufgrund des notwendigen TLB-Flushes müssen TLB-Einträge *nach* dem Adressraumwechsel wieder neu geladen werden. Diese *indirekten* Kosten können auf der x86-Architektur um ein Vielfaches höher ausfallen als die direkten Kosten (Kerneintritt, Code im Kern, Kernaustritt). Das Ersetzen eines TLB-Eintrags für Daten kostet auf einem Pentium 4 etwa 48 Taktzyklen, für Code-TLBs etwa 31 Taktzyklen [88]. Im Worst Case ist ein Neuladen aller TLB-Einträge notwendig, womit sich die indirekten Kosten auf etwa 3000 Takte für das Neuladen des Daten-TLBs (64 Einträge) plus 4000 Takte für das Neuladen des Code-TLBs (128 Einträge) belaufen.

Wird ein Betriebssystem-Kern, der normalerweise in allen Adressräumen eingeblendet ist, in einem eigenen Adressraum ausgeführt, sind mit jedem Systemaufruf plötzlich zwei zusätzliche Adressraumwechsel verbunden. Die dadurch entstehenden Kosten müssen unbedingt minimiert werden.

Xen [5] blendet deshalb den Betriebssystem-Kern des Gastes (z. B. XenoLinux) in alle Adressräume ein. Das automatische Invalidieren der betreffenden TLB-Einträge bei Adressraumwechseln wird durch des Setzen des Global-Bits verhindert. Der Schutz des Gast-Kerns vor den Gast-Anwendungen wird durch Ausnutzung der Privilegstufen erreicht, von denen die x86-Architektur vier besitzt (siehe Abschnitt 2.3.4). Der VM-Monitor von Xen läuft auf Ring 0, der Gast-Kern auf Ring 1, und die Gast-Anwendungen laufen auf Ring 3. Die meisten Architekturen besitzen nur zwei Privilegstufen, weshalb das Gast-Betriebssystem unter Xen auf diesen Architekturen nur im eigenen Adressraum laufen könnte.

L4Linux verfolgt einen anderen Weg: Der Linux-Server wird in einem separaten Adressraum auf der niedrigsten Privilegstufe (auf x86 Ring 3) ausgeführt. Um die Kosten der Adressraumwechsel zu minimieren, wird die Technik der „kleinen Adressräume“ angewendet, die Liedtke beschreibt [55]. Der Linux-Server wird in einem kleinen Adressraum ausgeführt, der in alle „großen“ Adressräume eingeblendet ist. Der Schutz des Gast-Kerns vor der Gast-Anwendung wird bei L4Linux im Unterschied zu Xen nicht durch Ausführung auf einer anderen Privilegstufe erreicht, sondern durch Zuweisung von verschiedenen Speicher-*Segmenten* an Gast-Kern und Gast-Anwendung (siehe Abbildung 2.3). Aus Sicht der Anwendung geschieht dies völlig transparent.

Unter L4Linux findet pro Systemruf des Gastes jeweils mindestens ein zusätzlicher Kernein- und -austritt statt, da IPC zwischen der Anwendung und dem Linux-Server generell durch den Mikrokern ausgeführt wird. Unter Xen löst ein Systemruf des Gastes analog zu L4Linux eine Exception aus. Diese wird aber direkt dem Gast-Kern zugestellt. Auf Architekturen mit nur zwei Privilegstufen fallen auch bei Xen pro Systemruf mindestens zwei Kernein- und -austritte an.

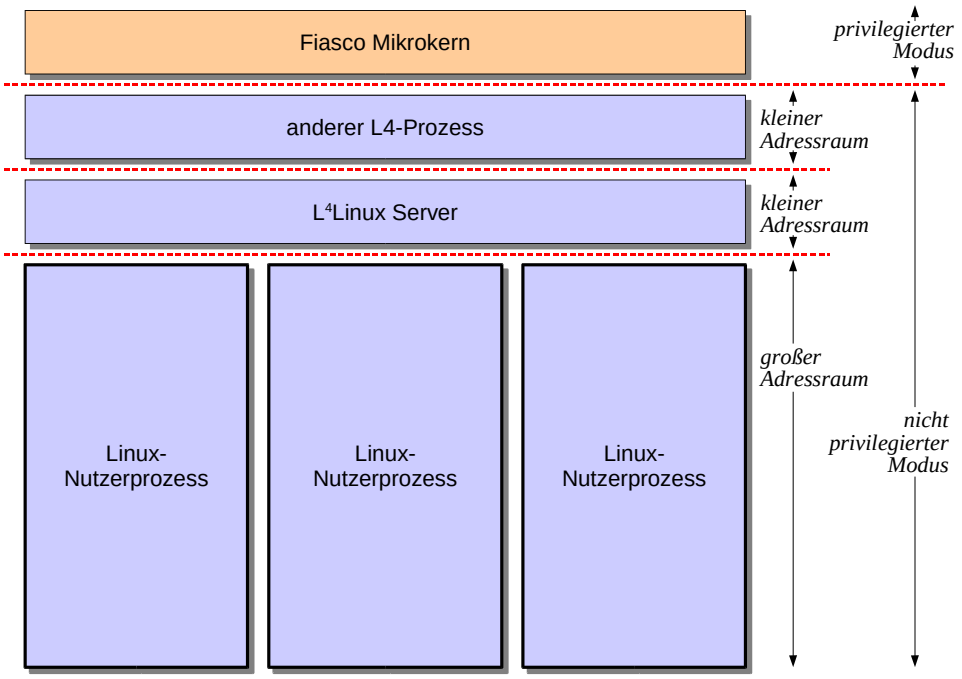


Abbildung 2.3: *L4Linux im kleinen Adressraum.* Der Mikrokern wird im privilegierten Modus ausgeführt und ist deshalb vor den Nutzeranwendungen geschützt. Der Linux-Server läuft in einem kleinen Adressraum, der durch Segmentierung von den Linux-Anwendungen in den großen Adressräumen separiert wird. Neben L4Linux können noch weitere L4-Anwendungen in kleinen Adressräumen ausgeführt werden.

Die Implementierung von kleinen Adressräumen mittels Segmentierung impliziert zusätzliche Kosten durch mehr Codekomplexität und durch zusätzliches Umladen der Segmente beim Wechsel von kleinen auf große Adressräume sowie bei den meisten Kernein- und -austritten. In ungünstigen Fällen können diese Zusatzkosten den Nutzen von kleinen Adressräumen neutralisieren. Werden z. B. mehrere aktive Prozesse nach *Round Robin* abgearbeitet, so sollte die TLB-Arbeitsmenge

aller Prozesse zzgl. der für den Kern als global gekennzeichneten TLB-Einträge die Anzahl an TLB-Einträgen in der CPU nicht wesentlich überschreiten.

Der mit Ring-0-Rechten ausgeführte Kern wird bei modernen Betriebssystemen meist in alle Adressräume eingeblendet, um Adressraumwechsel bei Systemaufrufen zu vermeiden. Da der Kern dedizierten Speicher für Code und Daten benötigt, ist der für Anwendungen verfügbare virtuelle Adressraum kleiner als der virtuelle Adressraum der CPU. Unter Linux für x86-Systeme können Anwendungen 3 GB virtuellen Speicher nutzen, während für den Kern 1 GB reserviert ist. Für volle Kompatibilität mit Linux müssen einer Linux-Anwendung auf L^{4}Linux bzw. XenoLinux ebenfalls 3 GB an virtuellem Speicher zur Verfügung stehen. Deshalb müssen sich der Xen-VM-Monitor bzw. der Fiasco-Mikrokern einen Bereich von 1 GB mit dem Linux-Kern teilen, wenn dieser aus Performance-Gründen in alle Adressräume eingeblendet wird. Die aktuelle Implementierung von Fiasco bietet aufgrund dieser Einschränkung für kleine Adressräume nur Platz für maximal 192 MB. Da die Größe eines kleinen Adressraumes eine Zweierpotenz sein muss (siehe L4-Manual [59]), stehen L^{4}Linux maximal 128 MB zur Verfügung.

Fiasco bot bereits zu Beginn dieser Arbeit Unterstützung für kleine Adressräume [34]. Die gemeinsame Nutzung von kleinen Adressräumen und eingeschränkten IO-Rechten (siehe folgender Abschnitt) war allerdings nicht möglich, eine entsprechende Implementierung wurde daher im Rahmen dieser Arbeit vorgenommen (vgl. Abschnitt 4.2.2).

2.4.3 Einschränkung der IO-Rechte

Einschränkung der IOPL Auf x86-Systemen definiert der *IO Privilege Level* (IOPL) die IO-Rechte eines Kontextes. Der IOPL wird in den CPU-Flags als Teil des Registersatzes gespeichert und kann nur im privilegierten Modus (CPL 0) geändert werden. Gilt IOPL $\geq$ CPL, so können einerseits Befehle zum Zugriff auf IO-Ports ohne weitere Restriktionen ausgeführt werden. Weiterhin kann das Interrupt-Flag mit den Anweisungen `cli` und `sti` gelöscht bzw. gesetzt werden [45]. Dieses Flag steuert die Zustellung von Hardware-Interrupts an die CPU.

Eine Nutzeranwendung läuft üblicherweise auf CPL 3 und kann daher mit IOPL 3 das Zustellen von Interrupts verhindern. Das vorübergehende Sperren der Interrupts ist ein beliebtes Mittel, um kritische Abschnitte auf Ein-Prozessor-Systemen zu schützen. Während die Interrupts gesperrt sind, ist eine Zwangsunterbrechung (*Preemption*) des aktuellen Prozesses durch den Scheduler oder andere Interrupt-Handler ausgeschlossen.[3] Bei der Synchronisation mehrerer CPUs ist diese Methode nicht anwendbar, da sich das Sperren der Interrupts nur auf eine CPU auswirkt.

Ein nicht vertrauenswürdiges Programm mit dem Recht, die Prozessor-Interrupts sperren zu dürfen, stellt ein Sicherheitsrisiko dar, weil es die Zwangsunterbrechung durch das Betriebssystem sowie die Zustellung von Geräte-Interrupts an Echtzeitprozesse verhindern kann. Dadurch werden Verfügbarkeit und Zusagenfähigkeit des Systems beeinträchtigt.

Aus diesen Gründen ist es notwendig, Nutzerprozesse mit *eingeschränkten* IO-Rechten auszuführen. Das Sperren von Interrupts muss dann durch andere Synchronisationsmittel ersetzt werden, z. B. durch eine geeignete Lock-Implementierung. Die Semantik von `cli` und `sti` muss dabei insbesondere im Hinblick auf Prioritäten eingehalten werden:

[3] Weiterhin muss sichergestellt sein, dass der kritische Abschnitt keine direkten (Syscalls) bzw. indirekten (z. B. Seitenfehler) Kerneintritte ausführt.

`cli` Nur ein Thread ist jeweils Eigentümer *E* des Locks. Der Thread, der sich zuerst um das `cli`-Lock bewirbt, wird Eigentümer des Locks. Spätere Bewerber um das Lock, die auch eine höhere Priorität als *E* haben können, blockieren und geben die CPU ab. Damit wird *Busy Waiting* vermieden.

`sti` Sobald *E* das `cli`-Lock freigibt, muss überprüft werden, ob es einen wartenden Thread gibt, der eine höhere Priorität als *E* besitzt. In diesem Fall muss der höchstpriore Thread *W* in der Warteschlange aufgeweckt werden, und dieser muss sofort aktiv werden. Im anderen Fall bleibt *E* aktiv.

Tamed-L^{4}Linux implementiert ein derartiges Lock als Ersatz für `cli` und `sti`, allerdings sperrt die bisherige Implementierung selbst die Interrupts für kurze Zeit und ist deshalb auf IOPL 3 angewiesen.

Solange ein Programm an mindestens einer Stelle im Code die Interrupts sperrt, muss es mit IOPL 3 ausgeführt werden. Dieses Programm kann damit auch ohne weitere Restriktionen auf alle IO-Ports zugreifen. Als Alternative wäre auch eine Emulation der `cli`- und `sti`-Anweisungen durch den Kern möglich, da diese Instruktionen Traps auslösen, wenn sie mit geringerer IOPL ausgeführt werden. Diese Lösung ist aber meist nicht praktikabel, da Kerneintritte auf vielen Architekturen teuer sind. Bei der x86-Architektur kommt hinzu, dass nicht alle Anweisungen, die für die Steuerung der Interrupt-Zustellung relevant sind, Traps auslösen (siehe Abschnitt 2.2.3).

In Tamed-L^{4}Linux wurden die folgenden Abschnitte identifiziert, die Interrupts für Synchronisationszwecke sperren:

- Das Testen und Setzen des neuen Lock-Eigentümers in der Implementierung des `cli`-Locks muss atomar erfolgen.
- Die Abfrage der CMOS-Uhr muss synchronisiert werden, weil die Geräteregister nur während eines bestimmten Zustandes ausgelesen werden dürfen.
- Wenn der L^{4}Linux-Server auf neue Aufträge wartet, muss er atomar von einer Top-Half-Interrupt-Routine aufgeweckt werden.
- Tamed-L^{4}Linux bestätigt eingehende Interrupts am Interrupt-Controller. Die Programmierung des Interrupt-Controllers muss synchronisiert mit anderen Anwendungen erfolgen.

Im Rahmen dieser Arbeit wurden alle diese Stellen durch Implementierungen ersetzt, die mit eingeschränkten IO-Rechten auskommen (vgl. Abschnitt 4.1).

IO-Bitmap und kleine Adressräume Wird ein Programm mit einer IOPL $<$ CPL ausgeführt, so überprüft der Prozessor bei Ausführung eines Befehls zur Ein-/Ausgabe über einen IO-Port die *IO-Bitmap*. Diese enthält ein Bit pro IO-Port. Ist dieses gesetzt, darf der aktuelle Prozess weder auf den entsprechenden Port schreiben, noch davon lesen. Die IO-Bitmap ist Teil des *Task State Segments* (TSS) und hat eine Größe von 8 KB (entspricht 65536 IO-Ports). Sie wird unter Fiasco bei jedem Adressraumwechsel automatisch mit dem Wechsel des Seitenverzeichnisses umgeschaltet, da jeder Prozess seine eigene IO-Bitmap in das TSS einblendet.

Soll zu einem kleinen Adressraum umgeschaltet werden, so muss die Umschaltung der IO-Bitmap separat erfolgen, weil dann das Seitenverzeichnis nicht verändert wird – es werden nur

neue Segmente geladen. Falls im vorherigen Adressraum eine IO-Bitmap eingeblendet war, müssen ferner die der IO-Bitmap zugehörigen TLB-Einträge geleert werden. Das explizite Leeren eines TLBs kann sehr teuer sein (siehe Abschnitt 4.2).

Linux unterstützte bis Version 2.4 nur 1024 IO-Ports. Ein Prozess, der auf die Ports 1025-65535 zugreifen möchte, wird daher mit IOPL 3 ausgeführt. Dies ist gefährlich, weil dadurch z. B. der gesamte X-Server mit uneingeschränkten IO-Rechten ausgeführt werden muss. Ab Linux Version 2.6.8 kann jeder Prozess eine IO-Bitmap für alle 65536 Ports erhalten. Im Unterschied zu Fiasco wird die IO-Bitmap beim Adressraumwechsel nicht durch Austausch der entsprechenden Seitentabellen-Einträge umgeschaltet, sondern durch Kopieren. Auf einem Pentium M 1.6 GHz dauert das Kopieren eines Bytes ohne Verwendung von Prefetch-Befehlen etwa 4 Zyklen. Das Kopieren von 8 KB kann daher mehr als 30000 Takte in Anspruch nehmen. Kopiert wird allerdings nicht bei jedem Adressraumwechsel, sondern erst dann, wenn der Prozess auf einen IO-Port zugreift.

2.4.4 Mehrprozessorsysteme

Sollen Anwendungen auf Mehrprozessorsystemen ausgeführt werden, sind im Vergleich zu Einprozessorsystemen zusätzliche Maßnahmen für die Synchronisation kritischer Abschnitte notwendig. Ein Standard-Betriebssystem, das als Nutzeranwendung auf einer virtuellen Maschine oder auf einem Mikrokern ausgeführt wird, kann mehrere Prozessoren nach folgenden Modellen unterstützen:

1-zu-1 Auf jedem Prozessor wird eine eigene Instanz des Betriebssystems mit jeweils eigenen Ressourcen ausgeführt, die jeweils als Server für dedizierte Anwendungen auf diesen Prozessoren dienen. Dieses Modell kann auf den Einprozessor-Fall abgebildet werden, weil jedes Betriebssystem nur einen Prozessor „sieht“. Aus Sicht der Anwendung ist dieses Modell unflexibel, da Anwendungen statisch an einen Prozessor gebunden sind und Ressourcen zwischen Nutzeranwendungen auf verschiedenen Prozessoren nur eingeschränkt dynamisch zur Laufzeit verteilt werden können.

1-zu-n Es wird nur eine Instanz des Betriebssystems ausgeführt, die statisch an einen Prozessor gebunden ist. Dieses Modell ähnelt der ersten Variante, wobei hier Nutzeranwendungen zwischen den Prozessoren migrieren können und ein Server die Ressourcen für alle Anwendungen verwaltet. Allerdings stellt der Prozessor, der das Betriebssystem ausführt, den Flaschenhals für das System dar.

n-zu-n Eine Instanz des Betriebssystems dient als Server für die Anwendungen auf verschiedenen Prozessoren und wird auf allen Prozessoren ausgeführt. Dieses Modell ist am flexibelsten, bedingt aber auch den größten Aufwand an Synchronisation. Weiterhin stellt dieses Modell besondere Anforderungen an die Schnittstelle zum Mikrokern.

Ferner werden Single-Server und Multi-Server unterschieden:

- L^{4}Linux ist als klassischer Single-Server implementiert. Systemaufrufe von Anwendungen werden mittels synchroner IPC im Server serialisiert, so dass zu einem bestimmten Zeitpunkt jeweils nur der Systemaufruf bearbeitet wird. Diese Implementierung ist mit allen Prozessormodellen von oben vereinbar.

- Linux als Multi-Server würde es ermöglichen, Systemaufrufe von verschiedenen Linux–Anwendungen gleichzeitig im Kern zu behandeln. Damit ergäbe sich aber auch zusätzlicher Synchronisationsaufwand innerhalb des Servers.

In dieser Arbeit wird L^{4}Linux als Single-Server auf einem Einprozessorsystem ausgeführt, da bisher nur eine *experimentelle* Fiasco-Implementierung für Mehrprozessorsysteme existiert.

2.5 IO-Architekturen

Eine wesentliche Aufgabe von Betriebssystemen ist die Abstraktion der Hardware. Gerätetreiber bilden dabei die untersten Einheiten. Ein wichtiger Bestandteil eines Gerätetreibers ist der Austausch von Daten mit dem Gerät. Genauso wie der Datenaustausch zwischen Anwendungen nur über spezielle Kanäle erfolgen darf, müssen zusätzliche Sicherheitsaspekte bei Treibern beachtet werden, wenn dem Betriebssystem nicht vertraut werden soll.

In vielen Architekturen ist es üblich, dass entweder die CPU aktiv an der Ein-/Ausgabe von Daten beteiligt ist, oder dass Geräte selbstständig Datentransaktionen vornehmen können. In der x86-Architektur werden Daten üblicherweise mittels *programmierter IO* oder mittels *direktem Speicherzugriff* (DMA) bewegt.

2.5.1 Programmierte IO (PIO)

In dieser Betriebsart greift die CPU lesend oder schreibend *direkt* auf Adressen externer Geräte zu. Das kann entweder über einen speziellen Adressraum, die IO-Ports (bei x86-Systemen) erfolgen, oder das Gerät blendet Register in den physischen Adressraum der CPU ein (*memory-mapped IO*). Während die CPU im ersten Fall spezielle Ein-/Ausgabe-Befehle ausführt, erfolgt der Zugriff bei memory-mapped IO mit normalen Befehlen zum Lesen/Schreiben von Speicher.

Bei direktem Zugriff ist die CPU unmittelbar am Datentransfer beteiligt und steht deshalb im Laufe der IO-Operation nicht für andere Aufgaben zur Verfügung. Bei CPUs mit mehreren Registersätzen (z. B. Hyper-Threading bei Intel Pentium 4) könnte ein dedizierter Thread für IO-Zugriffe vorgesehen werden. Weil dieser spezielle IO-Thread nur bestimmte Recheneinheiten belegt, dürften andere Threads mit hohem Rechenanteil nur unmerklich durch Ein-/Ausgabe behindert werden. Diese Lösung führt allerdings zu einer geringeren Auslastung der CPU, falls mehrere Threads im System rechenbereit sind.

Programmierte IO ist für schnelle Kommunikation mit externen Geräten nicht nur aufgrund der hohen CPU-Last ungeeignet, sondern auch aufgrund eingeschränkter Performance: Heutige PC-Hardware fügt aus Gründen der Abwärtskompatibilität Wartezyklen bei Zugriffen auf IO-Ports ein. Deshalb lassen sich mit dieser IO-Betriebsart meist nur verhältnismäßig geringe Datenraten erzielen.

Der Datenfluss bei programmierter IO ist durch Einschränkung der IO-Rechte kontrollierbar (siehe Abschnitt 2.4.3).

2.5.2 Direkter Speicher-Zugriff (DMA)

Für schnelle Kommunikation mit IO-Geräten ist der direkte Speicherzugriff (*Direct Memory Access, DMA*) vorgesehen, bei dem die aktive Rolle bei der Datenübertragung eine eigene Einheit, der

DMA-Controller, übernimmt. Für einen Datentransfer benötigt der DMA-Controller Informationen über Größe und Anfangsadresse von Blöcken im *physischen* Adressraum. Der Transfer erfolgt zwischen Gerät und Hauptspeicher ohne weiteres Mitwirken der CPU. Die Arbeit der CPU wird nur soweit ausgebremst, wie sich DMA-Einheit und CPU die Bus- und Hauptspeicher-Bandbreite teilen müssen. Nach Abschluss des Datentransfers löst die DMA-Einheit ggf. einen Interrupt aus.

Um die Caches innerhalb der CPU auch während des DMA-Transfers konsistent mit dem Hauptspeicher zu halten, „lauscht“ die CPU auf x86-Systemen am Adressbus und invalidiert ggf. ungültig gewordene Cache-Lines (*Snooping*). Auf anderen Architekturen müssen die Caches betreffender Seiten explizit invalidiert werden.

Die ursprüngliche PC-Architektur sieht keine speziellen Schutzmechanismen gegenüber DMA vor. Da der DMA-Controller selbst ein Gerät ist, das mittels programmierter IO programmiert wird, kann der Zugriff auf die DMA-Geräteregister eingeschränkt werden. Der Original-IBM-PC hatte nur einen DMA-Controller, der zentral von einer vertrauenswürdigen Komponente verwaltet werden kann. Seit der Einführung des PCI-Busses besitzen aber auch externe Geräte jeweils eigene DMA-Einheiten, die gerätespezifisch programmiert werden müssen (siehe folgender Abschnitt). Eine zentrale Verwaltung aller DMA-Einheiten ist deshalb nicht möglich.

2.5.3 Busmaster-DMA über PCI-Busse

Der PCI-Bus [74] ist ein weit verbreiteter paralleler IO-Bus für die Verbindung schneller IO-Geräte. Er ist üblicherweise über eine *PCI-Bridge* mit dem Systembus verbunden, an dem auch Hauptspeicher und CPU angeschlossen sind (siehe Abbildung 2.4). Der PCI-Bus unterstützt mehrere Adressierungsarten für die Interaktion zwischen Geräten. Der *Memory Space* aller PCI-Geräte und der Hauptspeicher bilden zusammen einen großen physischen Adressraum [80]. Auf den *I/O-Space* kann nur unter Verwendung spezieller IO-Zyklen über IO-Ports zugegriffen werden.

PCI-Geräte können temporär die Kontrolle über den Bus übernehmen (sie werden *Busmaster*) und während dieser Zeit aktiv Daten von einem Gerät am Bus (dem *Target*) empfangen bzw. an dieses Gerät senden. Die Auswahl des Ziel-Gerätes für eine Datentransaktion erfolgt implizit durch Spezifikation einer entsprechenden physischen IO-Adresse am Beginn der Transaktion. Der PCI-Bus unterstützt nicht mehrere parallele Datenströme: Ein zentraler Arbiter entscheidet, wer die Kontrolle über den Bus für eine bestimmte Zeiteinheit übernimmt. Der Arbiter ist normalerweise Bestandteil der PCI-Schnittstelle im Chipsatz (z. B. bei Intel 810 [48]).

Werden Daten zur PCI-Bridge am Systembus übertragen, so werden diese unter Umgehung der MMU direkt an die jeweiligen physischen Adressen im Hauptspeicher geschrieben. Ein Festplatten-Controller kann so beispielsweise aktiv und ohne Mitwirken der CPU Daten vom Controller-Cache in den Hauptspeicher der CPU übertragen. Diese Übertragungsart wird im folgenden als *Busmaster-DMA* bezeichnet. Der Hauptspeicher selbst kann nie Transaktionen initiieren.

PCI-Busmaster sind aufgrund des direkten Zugriffs auf den Hauptspeicher in der Lage, Adressraumgrenzen und Privilegstufen zu umgehen. Treiber für PCI-Geräte bedeuten daher ein Sicherheitsrisiko für vertrauenswürdige Anwendungen und den Betriebssystem-Kern.

Dass DMA ein Sicherheitsrisiko darstellt, wurde bereits 1981 von Rushby festgestellt [78]. Konventionelle Kerne müssten deshalb alle IO-Operationen generell kontrollieren, und dies verkompliziert das Design nach seiner Ansicht wesentlich. Das von ihm vorgestellte *Secure User Environment* wählt deshalb den radikalen Ansatz und verbietet DMA generell.

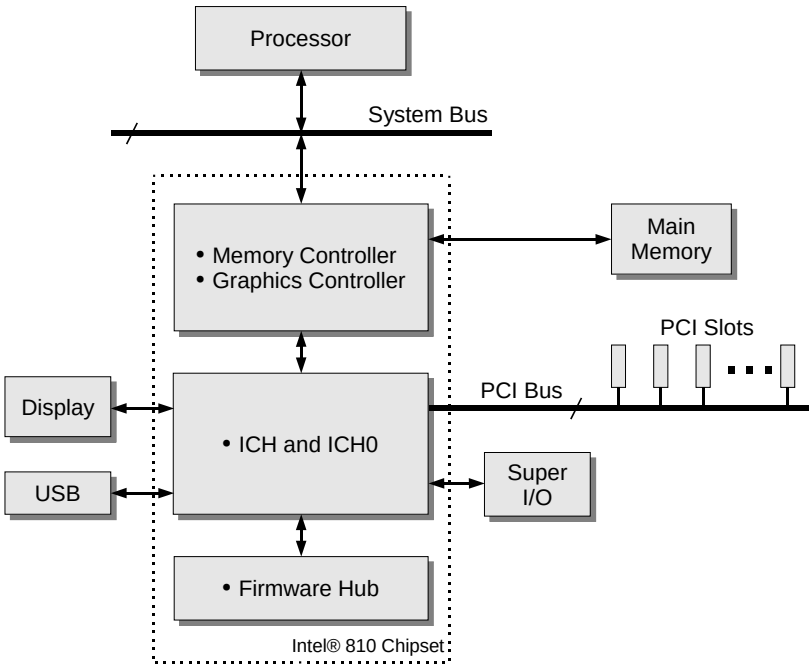


Abbildung 2.4: *Anbindung des PCI-Busses am Beispiel des Chipsatzes Intel 810 (nach [48]).*

PCI-Express [8] ist der aktuelle Nachfolger des PCI-Busses. Aus Programmierersicht verhält sich diese Architektur ähnlich wie ein PCI-Bus, aus Hardware-Sicht stellt er jedoch eine sternförmige Struktur mit einem Switch als Vermittler zwischen Endpunkten dar. Neben der größeren Datenrate unterstützt PCI-Express im Gegensatz zum PCI-Bus mehrere parallele Datenströme.

2.5.4 Einschränkung von Busmaster-DMA mittels spezieller Hardware

Herkömmliche IOMMUs

Eine IOMMU kann sicherstellen, dass IO-Geräte nicht auf beliebige physische Adressen zugreifen können. Das Funktionsprinzip ist ähnlich einer MMU: Es werden nur die Zugriffe auf den physischen Adressraum zugelassen, für die ein Eintrag in einer ggf. mehrstufigen Tabelle existiert. Der Hauptspeicher und Geräte an den IO-Bussen bilden den physischen Adressraum. IOTLBs speichern gültige Abbildungen, damit nicht bei jedem IO-Zugriff über die Tabelle traversiert werden muss.

Auf der x86-Architektur wurde die Funktionalität von IOMMUs zuerst innerhalb von AGP-Bridges implementiert. Der AGP-Bus dient zur schnellen Anbindung der Grafikkarte an den Hauptspeicher, damit große Datenmengen, die nicht in den Speicher der Grafikkarte passen (z. B. umfangreiche Texturen, Geometrieinformationen und Kommandosequenzen), effizient aus dem Hauptspeicher gelesen werden können. Aus Sicht der Grafikkarte ist ein linearer Zugriff auf den Hauptspeicher erwünscht, allerdings ist es für Betriebssysteme nicht praktikabel, einen großen statischen zusammenhängenden Bereich des physischen Hauptspeichers für teilweise temporäre Daten der Grafikkarte zu reservieren. Aus diesem Grund enthält jede AGP-Bridge ab AGP Rev. 2.0 eine IOMMU, die in der Lage ist, aus verteilten Kacheln des Hauptspeichers eine lineare virtuelle Region

(die *AGP Graphics Aperture*) für die Grafikkarte zusammenzufügen [41]. Zugriffe auf Bereiche außerhalb der Graphics Aperture werden ohne Übersetzung an den Hauptspeicher weitergeleitet.

In aktuellen Rechnerarchitekturen sind IOMMUs vor allem auf 64-Bit-Systemen zu finden, wo sie die Umsetzung zwischen dem physischen Adressraum der CPU und dem 32-Bit PCI-Bus übernehmen. Beispiele dafür sind im AMD Opteron [1], in Chipsätzen für Intel Itanium [73] sowie in der microSPARC-Architektur [85] zu finden.

Leslie und Heiser zeigen, wie mittels einer herkömmlichen IOMMU der Zugriff von Treibern auf beliebige physische Adressen verhindert werden kann [52]. Während der gesamten Datenübertragung darf der in die Übertragung involvierte physische Speicher keiner anderen Anwendung zugeordnet werden. Insbesondere muss das Auslagern einer solchen Kachel während dieser Zeit ausgeschlossen werden (*Pinning*). Vor Beginn der Übertragung muss der Treiber gültige Abbildungen physischen Speichers für den Zugriff durch das Gerät in der IOMMU etablieren. Ein Gerät kann nur auf Daten zugreifen, für die in der IOMMU eine Abbildung existiert. Allein die Komponente zum Programmieren der IOMMU muss vertrauenswürdig sein. Sie hat sicherzustellen, dass ein Client nur physische Bereiche, die *ihm* zugeordnet sind, für externe Geräte zugreifbar machen kann.

Da die beschriebene IOMMU generell Abbildungen für den gesamten PCI-Bus vornimmt, existiert nur ein Adressraum pro IO-Bus bzw. für alle IO-Busse. Geräte können damit ohne Vermittlung der IOMMU untereinander kommunizieren. Wenn zwei Geräte gleichzeitig per Busmaster-DMA auf den Hauptspeicher zugreifen, kann ein Gerät am Datenstrom des anderen Gerätes mitlauschen, wenn es die aktuelle Abbildung in der IOMMU erraten hat.

Wird ein zu kapselnder Gerätetreiber als bösartig angenommen, reicht dieses Modell einer IOMMU nicht aus, um unkontrollierten Datenaustausch zu vermeiden. In Abschnitt 3.2 wird ein diesbezüglich allgemeineres Modell einer IOMMU vorgestellt.

NGSCB und LaGrande Technology

Der direkte Zugriff von Busmaster-DMA-fähigen Geräten auf Speicher kann mit NGSCB (siehe Abschnitt 2.2.4) kontrolliert werden, indem privilegierte Programme im Root Mode eine Maske definieren, die ausgewählte Speicherbereiche mit Seitengranularität von DMA ausschließt [70]. Ähnlich wird dieses Problem mit LaGrande Technology gelöst. Beide Techniken erweitern den Schutz im Gegensatz zu herkömmlichen IOMMUs auch auf den Chipsatz, so dass nicht nur der Hauptspeicher vor nicht autorisierter DMA geschützt werden kann, sondern alle Geräte.

Über beide Techniken ist allerdings bisher nicht bekannt, ob sie nur *eine systemweite Maske* für gültige DMA-Regionen definieren oder *eine Maske pro Gerät*. Bei nur einer systemweiten Maske kann nicht verhindert werden, dass zwei gleichzeitig aktive Geräte über eine DMA-Region kommunizieren. Dies soll am folgenden Beispiel verdeutlicht werden:

In Abschnitt 2.2.1 wurde dargestellt, wie in der µSINA-Architektur (MLS) zwei nicht vertrauenswürdige Netzwerk-Stacks Daten nur über eine vertrauenswürdige Anwendung (den Viadukt) austauschen können. Beide Stacks würden mit eingeschränkten Rechten, also *nicht* im Root Mode, ausgeführt. Für Netzwerk-Kommunikation mittels Busmaster-DMA müssten beide Treiber gültige DMA-Regionen für ihren zugehörigen Netzwerkadapter vom Domain Manager beantragen. Sind beide Geräte gleichzeitig aktiv, so kann ein Adapter die DMA-Region des anderen benutzen, um Daten mit dem anderen Stack unter Umgehung des Viaduktes auszutauschen.

Die S/390 Channel-Architektur

Die S/390 Channel-Architektur [83] unterscheidet sich erheblich vom IO-System der PC-Architektur. Sie enthält ein IO-Subsystem, das eine einheitliche Sicht auf IO-Geräte bietet.

Datenübertragungen werden von so genannten *Integrated Offload Processors* (IOPs) durchgeführt, die kleine Programme (*Channel Programs*) ausführen. Die IOPs sind unabhängig von den Geräten und werden einheitlich programmiert. Ein Channel Program enthält Befehle für Datentransfer vom bzw. zum Hauptspeicher und Regeln für die Fehlerbehandlung. Der Datenverkehr wird immer vom *Host* initiiert, niemals vom *Gerät*.

Diese Architektur erlaubt die parallele Ausführung von IO-Operationen unabhängig von der CPU. Im Gegensatz zur Channel-Architektur muss ein PCI-Gerät auf Standard-Hardware entsprechend programmiert werden, damit es eine Datenübertragung per Busmaster-DMA durchführen kann, und die Programmierschnittstelle ist gerätespezifisch. Eine Kontrolle der DMA-Einheit erfordert daher den zusätzlichen Aufwand von IOMMUs.

Kapitel 3

Entwurf

Mit L^{4}Linux stand bereits zu Beginn dieser Arbeit ein para-virtualisiertes System zur Verfügung, bei dem das Standard-Betriebssystem Linux durch Ausführung auf einem Mikrokern sowie wenigen vertrauenswürdigen Servern gekapselt wird. Linux hat nur noch eingeschränkte Rechte und kann damit bestimmte privilegierte Befehle nicht mehr ausführen (z. B. Setzen der Seitentabelle).

Allerdings müssen Anwendungen, die neben L^{4}Linux ausgeführt werden, diesem immer noch vertrauen, da dieses nicht mit eingeschränkten IO-Rechten ausgeführt wird und damit Interrupts sperren und beliebig auf Geräte zugreifen kann, die über IO-Ports programmiert werden. In Abschnitt 3.1 wird gezeigt, wie kritische Abschnitte anders als durch Sperrung der Interrupts geschützt werden können.

Das verbleibende Problem stellen Geräte dar, die Daten per Busmaster-DMA übertragen können. Da diese Übertragungsart auf aktueller Hardware nicht ausreichend kontrolliert werden kann, sind Treiber für diese Geräte in der Lage, mit anderen Geräten direkt zu kommunizieren und beliebige Daten im Hauptspeicher zu lesen bzw. zu schreiben. In Abschnitt 3.2 werden daher Möglichkeiten erörtert, wie sich das System vor Kompromittierung durch nicht vertrauenswürdige Treiber schützen lässt, und es wird eine allgemeine Lösung für das DMA-Problem dargestellt.

3.1 Synchronisation ohne Sperren der Interrupts

Wie in Abschnitt 2.4.3 beschrieben, verwenden Betriebssysteme häufig das Sperren von Interrupts zur Synchronisation kritischer Abschnitte. In einem zusagenfähigen System mit eingeschränkten Rechten müssen kritische Abschnitte mit anderen Hilfsmitteln synchronisiert werden, weil das Sperren von Interrupts meist spezielle Privilegien voraussetzt und die Vorhersagbarkeit beeinflusst. Die Verwendung von Interrupt-Sperren erfolgt in Betriebssystemen üblicherweise aus zwei Gründen, die in den folgenden Abschnitten dargestellt sind.

3.1.1 Zeitkritischer Code

Ein Abschnitt enthält eine Folge von Anweisungen, die innerhalb einer bestimmten maximalen Zeit ausgeführt werden müssen. Dieser Fall tritt vor allem bei Gerätetreibern auf. Beispielsweise ist der Zugriff auf den Zähler der CMOS-Uhr eines PCs nur während eines bestimmten Zeitintervalls erlaubt [72]. Außerhalb dieses Intervalls ist das Ergebnis der Lese- oder Schreiboperation nicht bestimmt. Der Anfang des gültigen Zeitfensters wird durch Pollen eines Statusregisters auf ein bestimmtes Bit ermittelt. Durch Sperren der Interrupts wird sichergestellt, dass das Programm von der Erkennung des Bits bis nach dem Zugriff auf das Geräteregister atomar ausgeführt wird.

Die korrekte Ausführung zeitkritischer Abschnitte kann wie folgt ohne Sperrung der Interrupts erreicht werden:

- Handelt es sich um einen sehr kurzen Abschnitt, kann eine Technik namens *Delayed Preemption* [89] verwendet werden. Dabei wird der Betriebssystem-Kern durch geeignete Mittel (z. B. Systemruf, gemeinsamer Datenbereich) informiert, dass der Prozess für eine kurze Zeit (im Bereich von wenigen Mikrosekunden) nicht unterbrochen werden möchte. Nach Ausführung des atomaren Codes gibt der Prozess freiwillig die Kontrolle an den Scheduler ab, falls dieser im kritischen Abschnitt unterbrochen werden sollte. Die Information über eine versuchte Unterbrechung muss vom Kern in geeigneter Form exportiert werden. Liegt ein Versuch zur Unterbrechung des kritischen Abschnitts vor und lässt der Prozess die maximale Zeitspanne ohne freiwillige Preemption verstreichen, erfolgt eine Zwangsunterbrechung durch den Scheduler. Diese Methode impliziert minimale Kosten für den Fall, dass die reservierte Zeit nicht überschritten wird.

- Eine Alternative zu Delayed Preemption ist die Auslagerung des kritischen Abschnitts in einen vertrauenswürdigen Server, der das Recht hat, die Interrupts zu sperren bzw. der mit einer höheren statischen Priorität ausgestattet ist. Im Vergleich zu Delayed Preemption fallen zusätzliche Kosten für Adressraumwechsel an. Die Zwangsunterbrechung des Schedulings für die Zeit des kritischen Abschnitts kann negative Auswirkungen auf die Vorhersagbarkeit des Systems haben. Die Synchronisation durch Prioritäten erfordert eine Sonderbehandlung im Multiprozessor-Fall.

- Bei zeitkritischen Abschnitten, die ein *Rollback* erlauben, kann der kritische Abschnitt ohne Synchronisation ausgeführt werden, ähnlich wie in [7] beschrieben. Für den Fall, dass während der Ausführung ein Kontextwechsel stattgefunden hat, wird der zeitkritische Abschnitt noch einmal wiederholt. Ob die zeitlichen Bedingungen eingehalten wurden, kann z. B. mittels *Time Stamp Counter* [47] überprüft werden. Für das Beispiel der CMOS-Uhr des PCs wird das Auslesen einfach wiederholt, falls während des Lesens eine Unterbrechung stattfand.

- Bestimmte kritische Abschnitte können nach Unterbrechung nicht neu gestartet werden. Wird ein kritischer Abschnitt unterbrochen, könnte dieser aber vor Ausführung der Unterbrechungsroutine erst zu Ende ausgeführt werden (*Rollforward* [71]). Diese Technik erfordert jedoch, dass alle möglichen Unterbrechungsroutinen mit diesem Fall umgehen können. Dies ist insbesondere nicht möglich, wenn kritische Abschnitte in Nutzerprogrammen durch Unterbrechungsroutinen des Kerns (z. B. für Zeitgeber-Interrupt) unterbrochen werden, da aus Gründen der Sicherheit im Kontext des Kerns kein Nutzercode ausgeführt werden sollte.

3.1.2 Gegenseitiger Ausschluss

Der klassische Einsatzfall für das Sperren von Interrupts ist der gegenseitige Ausschluss paralleler Aktivitäten bei kritischen Abschnitten. Hier gibt es üblicherweise keine zeitlichen Anforderungen, sondern es soll meist eine bestimmte *Reihenfolge* der Zugriffe auf Variablen oder Geräteregister durch nebenläufige Aktivitäten sichergestellt werden. Als Alternative zur Sperrung von Interrupts kann hier ein Lock benutzt werden, das die Semantik von `cli` und `sti` nachbildet und auf das alle

beteiligten Threads synchronisieren müssen (siehe Abschnitt 2.4.3). Eine geeignete Implementierung für Nizza ist in Abschnitt 4.1.1 angegeben.

Locks bzw. Semaphore basieren oft auf Prozessorinstruktionen, die atomar Werte im Speicher verändern und testen können (z. B. Dekrementieren kombiniert mit Test auf Null). Auf Architekturen ohne derartige Instruktionen müssen diese geeignet nachgebildet werden, beispielsweise mittels Systemrufen oder mittels Delayed Preemption (siehe vorheriger Abschnitt).

3.1.3 Vorgehensweise

Die Auswahl, welche der gezeigten Techniken anstelle der Interrupt-Sperrung verwendet werden kann, erfolgt durch Analyse des Codes. Den Normalfall stellt unter Linux der gegenseitige Ausschluss bei kritischen Abschnitten dar. Daher wurden die in Linux definierten Funktionen `cli` und `sti` durch ein Lock ersetzt. Besonders untersucht werden müssen Code-Abschnitte im architekturabhängigen Teil. Dort mussten einige wenige kritische Abschnitte durch andere Maßnahmen geschützt werden (siehe Abschnitt 4.1.1).

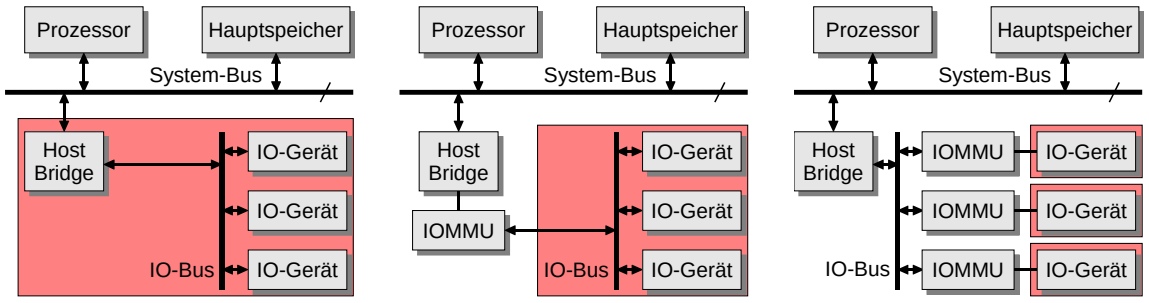
Im Rahmen dieser Arbeit konnten alle `cli`- und `sti`-Instruktionen in Linux eliminiert werden. Es ist nun möglich, den Linux-Server mit eingeschränkten IO-Rechten (IOPL 0) auszuführen.

Mit Hilfe der in den Abschnitten 3.1.1 und 3.1.2 dargestellten Methoden sollte es generell vermeidbar sein, Interrupts in Nutzerprozessen zum Zwecke der Synchronisation zu sperren.

3.2 Sichere Ein-/Ausgabe mittels Busmaster-DMA

Externe Geräte können mittels Busmaster-DMA selbstständig und ohne Kontrolle Daten im Hauptspeicher referenzieren und untereinander kommunizieren. Dies stellt ein Problem dar, da in sicheren Systemen jeder Austausch von Informationen autorisiert sein muss. Als Angriffsmodell wird hier die vollständige Penetration des Gerätetreibers angenommen. Treiber und Gerät sind damit potenziell bösartig und müssen geeignet gekapselt werden. Als Lösung für dieses Problem bieten sich zwei Möglichkeiten an: Entweder wird es nicht vertrauenswürdigen Treibern nicht gestattet, DMA-Operationen auszuführen, oder DMA-Operationen werden *kontrolliert* ausgeführt.

Generell lassen sich drei Hardware-Architekturen unterscheiden, die jeweils unterschiedlichen Schutz gegen Busmaster-DMA bieten, siehe Abbildung 3.1:



(a) Ohne IOMMU.

(b) Ein IO-Adressraum pro IO-Bus.

(c) Ein IO-Adressraum pro IO-Gerät.

Abbildung 3.1: *Busstrukturen.* Schutz-Domänen sind farblich markiert.

(a) DMA-Zugriffe können *ohne IOMMU* nicht kontrolliert werden. Um Sicherheit garantieren zu können, müssen die Treiber für alle DMA-fähigen Geräte sowie die Geräte selbst vertrauenswürdig sein.

(b) *Ein IO-Adressraum pro IO-Bus* ermöglicht es, Kern und Anwendungen vor direkter Manipulation mittels Busmaster-DMA zu schützen, indem einer Gruppe von Geräten nur der Zugriff auf Bereiche des physischen Adressraumes gestattet wird, für die gültige Abbildungen von diesem IO-Adressraum definiert sind (Abschnitt 2.5.4). Die Zuteilung eines IO-Adressraumes pro Bus verhindert aber nicht, dass ein IO-Gerät eine Abbildung in der IOMMU benutzt, die nicht für dieses eingerichtet wurde. Wenn zwei Geräte A und B der selben Gruppe parallel für Busmaster-DMA programmiert wurden, dann könnte Gerät B eine Abbildung, die für Gerät A eingerichtet wurde, erraten und damit den Datenstrom an bzw. von Gerät A mitlesen oder verändern. Dies führt im weiteren Sinne zu einer Kommunikation der beiden Geräte ohne Vermittlung der IOMMU.

(c) Für einen vollständigen Schutz ist es deshalb notwendig, jedes Gerät einzeln zu kapseln. Dies erfordert *einen IO-Adressraum pro Gerät*, implementierbar durch *eine logische IOMMU pro Gerät*. Der Adressraum eines IO-Gerätes besteht aus Abbildungen zwischen den Adressen des IO-Gerätes und den Adressen des physischen Adressraumes. Hier wird der Begriff „logische“ IOMMU verwendet, weil anstelle einer physischen IOMMU pro Gerät (wie im Bild dargestellt) auch eine einzige IOMMU für alle Geräte eines IO-Busses in der System-Bridge implementiert sein könnte, die für den gerade aktiven Busmaster die Kontrolle übernimmt (siehe Abschnitt 3.2.3).

Architektur (c) erreicht die vollständige Kontrolle der Geräte-DMA durch räumliche Kapselung. Leider ist diese Architektur auf heutiger Hardware noch nicht zu finden.

Es gibt allerdings den Ansatz, herkömmliche IOMMUs nach Modell (b) zeitlich so zu multiplexen, dass immer nur ein IO-Gerät pro IOMMU einen Datentransfer ausführen kann [53]. Damit entscheidet nicht mehr der Bus-Arbiter, welches Gerät Busmaster wird, sondern eine vertrauenswürdige Instanz innerhalb des Betriebssystems, der IO-Scheduler. Jedes Gerät besitzt einen eigenen Kontext in der IOMMU, der aktiviert wird, wenn der IO-Scheduler das Gerät auswählt. Dieser Ansatz impliziert zusätzliche Kosten für das Umschalten der Kontext-Informationen in der IOMMU. Außerdem können bei per Software erzwungenem Scheduling nur eingeschränkt Informationen über den aktuellen Gerätezustand einbezogen werden.

Nachfolgend werden die einzelnen Varianten einer IOMMU dargestellt. Für über den Trivialfall (Verbot von Busmaster-DMA) hinausgehende Lösungen wird im übernächsten Abschnitt der Begriff des DMA-Managers eingeführt.

3.2.1 Verbot von Busmaster-DMA

Durch Verbieten des Busmaster-DMA-Modus kann verhindert werden, dass eine nicht vertrauenswürdige Anwendung mittels DMA das System kompromittiert. Die Abschaltung kann z. B. über den Konfigurationsbereich von PCI-Geräten erfolgen. Wenn Busmaster-DMA-fähige Geräte nur durch vertrauenswürdige Treiber gesteuert werden und anderen Geräten der DMA-Modus nicht erlaubt wird, dann ist eine Integration einer IOMMU im System nicht notwendig.

Viele Geräte beherrschen neben dem DMA-Modus auch die Möglichkeit, Daten per programmierter IO (PIO) auszutauschen. Aus den im Abschnitt 2.5.1 aufgeführten Gründen kann allerdings nicht immer auf die Möglichkeit, Daten per DMA zu transportieren, verzichtet werden. Beispielsweise können Netzwerk-Controller, die Deskriptorlisten im Hauptspeicher halten, nicht mit programmierter IO betrieben werden.

3.2.2 Verwaltung von IO-Adressräumen

Für die Durchsetzung von virtuellen Adressräumen werden klassischerweise *Memory Management Units* (MMUs) eingesetzt. Die MMU wird meist durch den Kern eines Betriebssystems als vertrauenswürdige Instanz programmiert. Der Kern exportiert eine geeignete Schnittstelle, die es Nutzeranwendungen ermöglicht, virtuelle Adressräume aus Abbildungen auf den physischen Adressraum aufzubauen. Dieser umfasst neben dem Hauptspeicher auch *memory mapped* Register und eingeblendeten Speicher von IO-Geräten. In Nizza sind Pager für den Aufbau der Adressräume verantwortlich (vgl. Abschnitt 2.3.3).

Durch IOMMUs werden – analog zu MMUs – Adressräume für IO-Geräte geschaffen, je nach Implementierung entweder ein Adressraum pro IO-Bus oder ein Adressraum pro IO-Gerät. IO-Adressräume enthalten ebenfalls Abbildungen auf den physischen Adressraum der CPU. Diese werden in geeigneter Form gespeichert. Nachfolgend wird für diese Struktur der Begriff *IO-Tabelle* verwendet – analog den Seitentabellen für virtuelle Adressräume der CPU.

Wer programmiert IOMMUs?

Prinzipiell fügen sich IO-Adressräume nahtlos in das Konzept der virtuellen Adressräume ein und könnten unter Nizza mittels Pager verwaltet werden: Ein Gerät benötigt für eine DMA-Operation eine Abbildung auf den physischen Adressraum. Diese könnte der Gerätetreiber einrichten (lassen), der damit gleichzeitig die Rolle eines Pagers für das Gerät übernimmt. Der Treiber hat selbst nur Zugriff auf Abbildungen, die ihm aus anderen Adressräumen übereignet wurden.

Bei näherer Betrachtung offenbart sich allerdings, dass die Seitentabellen der MMU und die IO-Tabellen der IOMMUs konsistent gehalten werden müssen, d. h. Abbildungen dürfen in einen IO-Adressraum nur aufgenommen werden, wenn sie von einem virtuellen Adressraum oder einem IO-Adressraum weitergegeben wurden. Bei Eintragung einer Abbildung in eine IO-Tabelle durch einen Treiber muss daher folgendes bekannt sein:

- Auf welche Kachel soll die Abbildung eingerichtet werden?
- Besitzt der Treiber das Recht, eine solche Abbildung einzurichten, d. h. hat er selbst eine Abbildung auf diese Kachel mit den geforderten Rechten (nur lesbar oder les- und schreibbar)?

Unter Nizza werden Seitentabellen ausschließlich vom Kern gelesen und geschrieben. Um sicherzustellen, dass nur rechtmäßig weitergegebene Abbildungen in IO-Tabellen eingetragen werden, gibt es folgende Möglichkeiten:

- IOMMUs werden ausschließlich vom Kern programmiert. Dieser ist vertrauenswürdig und besitzt direkten Zugriff auf die Seitentabellen der virtuellen Adressräume der Nutzerprozesse.

- Die Programmierung der IOMMUs wird von vertrauenswürdigen Nutzerprozessen übernommen. Zur Kontrolle der Rechtmäßigkeit von Abbildungen exportiert der Kern die Seitentabellen der Nutzerprozesse lesend. Alternativ dazu müssen die Informationen über zugehörige Kachelnummer und Rechte der Abbildung auf andere Art und Weise ermittelt werden. Der zum Root-Pager σ_0 gehörige Adressraum enthält bei Start des Systems exklusive Abbildungen auf alle physischen Kacheln. Diese können selektiv an andere Adressräume weitergegeben werden. Ausgehend von σ_0 können die benötigten Informationen ausschließlich auf Ebene der Nutzerprozesse ermittelt werden.

Die Programmierung der IOMMUs durch den Kern erfordert entsprechende *Treiber* als integralen Bestandteil des Kerns. Geeignete IOMMUs wurden bisher noch nicht in Hardware implementiert. Auch wenn dies für kommende Generationen zu erwarten ist, ist die Schnittstelle zur Programmierung von IOMMUs höchstwahrscheinlich umfangreicher als die Schnittstelle zur Programmierung von MMUs (Seitentabellen): MMUs sind *interner* Bestandteil der meisten modernen CPUs, während IOMMUs als externe Geräte implementiert werden. Mikrokerne mit L4-Schnittstelle sollten jedoch nur kleine und unbedingt notwendige Komponenten enthalten. Aus diesem Grund wird in dieser Arbeit der zweite Ansatz verfolgt.

Der DMA-Manager als vertrauenswürdige Instanz

Im folgenden wird die vertrauenswürdige Instanz zur Programmierung einer IOMMU als DMA-Manager bezeichnet. Aus praktischen Gründen erscheint es sinnvoll, die Programmierung *aller* IOMMUs durch *einen* DMA-Manager zu kontrollieren. Es wäre auch möglich, mehrere DMA-Manager einzusetzen, die jeweils bestimmte Gruppen von IOMMUs (z. B. eine Gruppe pro IO-Bus) kontrollieren. Dies vereinfacht das Modell aber nicht, da letztlich jeder DMA-Manager vertrauenswürdig sein muss.

Es existiert noch ein weiterer Grund, die Anzahl an DMA-Managern gering zu halten: Wenn ein Prozess sicherstellen möchte, dass auf einer Seite seines Adressraumes kein anderer Prozess und kein IO-Gerät zugreifen kann, führt der Prozess eine *unmap*-Operation auf diese Seite aus (vgl. Abschnitt 2.3.3). Die Mapping-Datenbank stellt sicher, dass nach erfolgreicher Ausführung dieser Operation keine Abbildungen in anderen virtuellen Adressräumen existieren. Um Abbildungen aus allen IO-Adressräumen zu entfernen, müssen nun ebenfalls alle DMA-Manager aufgerufen werden.

Wie überprüft der DMA-Manager, ob der Auftraggeber eines Eintrags in der IO-Tabelle die notwendigen Rechte an der entsprechenden Kachel besitzt? In Abschnitt 2.3.3 wurden Dataspaces als Container für abstrakte Daten eingeführt. Ein Client kann durch Angabe einer Dataspace-ID (und ggf. eines Offsets und einer Länge) einem Treiber Puffer für DMA-Operationen zur Verfügung stellen. Für die Einrichtung einer Abbildung vom Adressraum des Gerätes auf den Dataspace wenden sich entweder Client oder Treiber unter Angabe der Dataspace-ID an den DMA-Manager. Dieser erfragt vom Dataspace-Manager die physischen Adressen der Kacheln des Dataspaces. Kann der DMA-Manager dem Dataspace-Manager nicht vertrauen, dann muss diese Information ausgehend vom Root-Pager validiert werden (siehe oben). Der Dataspace-Manager überprüft seinerseits, ob der DMA-Manager berechtigt ist, auf den entsprechenden Dataspace zuzugreifen. Dieses Recht muss dem DMA-Manager vom Client oder Treiber erteilt worden sein.

Alternativ zum Durchlaufen der Vertrauenskette ausgehend von σ_0 könnte der Kern auch lesenden Zugriff auf alle Seitentabellen exportieren. Dieser Ansatz wurde hier nicht weiter verfolgt, da

dies dem Prinzip von L4 widerspricht, dem Kern nur Eigenschaften hinzuzufügen, die sich nicht oder nur mit großem Aufwand im Nutzermodus implementieren lassen.

Pinning

Indem der DMA-Manager die physische Adresse der Region validiert hat, kann ausgeschlossen werden, dass ein nicht vertrauenswürdiger Client beliebigen physischen Speicher für DMA-Operationen zugreifbar machen kann. Zusätzlich muss noch sichergestellt werden, dass der Puffer während der DMA-Operation nicht freigegeben oder ausgelagert wird. Dies wird durch *Pinning* des Dataspaces beim entsprechenden Dataspace-Manager erreicht. Bevor das Pinning eines Dataspaces aufgehoben werden kann, muss der entsprechende Dataspace-Manager mit Hilfe des DMA-Managers sicherstellen, dass in keinem IO-Adressraum Abbildungen auf Kacheln dieses Dataspaces vorliegen.

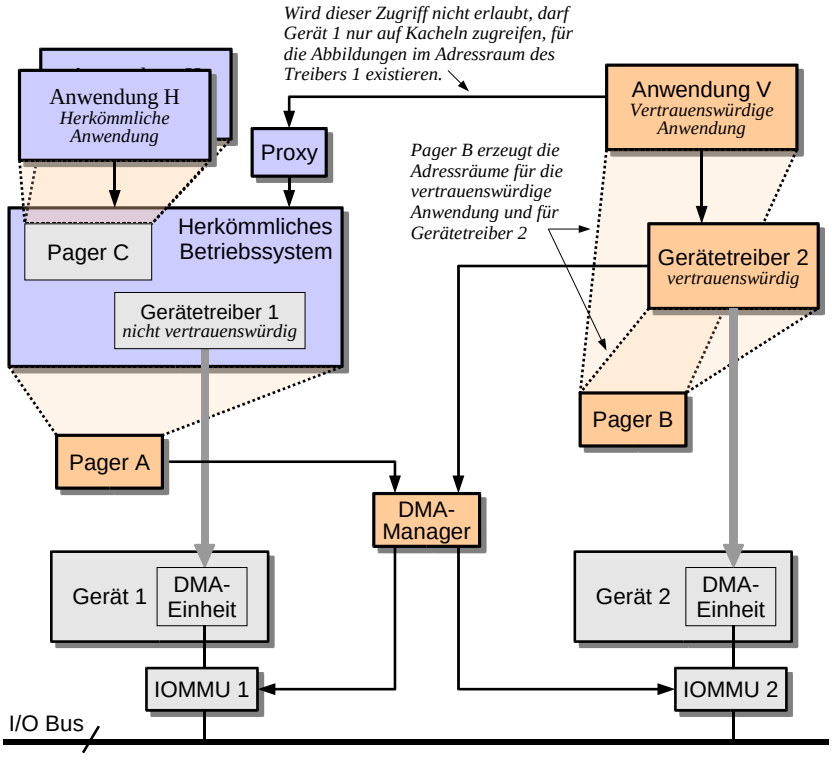


Abbildung 3.2: *Treiber-Szenarios mit IOMMUs und DMA-Manager.* Der DMA-Manager verwaltet für jedes Gerät Fenster für zulässige DMA-Operationen und programmiert die IOMMUs. Im Beispiel gehört Gerätetreiber 2 zur TCB.

In Abbildung 3.2 ist ein Beispielszenario mit zwei Gerätetreibern dargestellt. Gerätetreiber 1 ist Teil eines Standard-Betriebssystems und deshalb nicht vertrauenswürdig. Sofern keine Zugriffe von anderen Prozessen auf diesen Gerätetreiber ausgeführt werden, müssen Puffer für DMA-Transaktionen auf Kacheln des Pagers A begrenzt werden, da dieser den Adressraum des herkömmlichen Betriebssystems erzeugt. Pager A ist auch für das Pinnen der entsprechenden Kacheln verantwortlich.

Da Client und Treiber Bestandteil des gleichen Prozesses sind, kann von außen nicht entschieden werden, wann ein Auftrag mit potenziellen DMA-Transaktionen beginnt oder endet. Somit muss der *komplette* Speicher des herkömmlichen Betriebssystems permanent gepinnt und für Gerät 1 zugreifbar bleiben. Dies wird durch entsprechende Einträge in der IO-Tabelle von IOMMU 1 erreicht.

Gerätetreiber 1 kann allerdings auch von *außen* über einen Proxy angesprochen werden, der für den Export der Schnittstelle des Treibers verantwortlich ist. Greift Anwendung V auf Gerätetreiber 1 zu, muss entweder die Anwendung V oder der Proxy eine gültige Abbildung in IOMMU 1 beim DMA-Manager beantragen.

Da Gerätetreiber 2 vertrauenswürdig ist, kann entweder dieser Treiber oder Anwendung V das Eintragen von gültigen Abbildungen für Gerät 2 beim DMA-Manager erfragen. Das Pinning der Puffer muss über Pager B sichergestellt werden.

3.2.3 IOMMU-Implementierung in Hardware

Ein IO-Adressraum pro PCI-Bus Wie im Abschnitt 2.5.4 beschrieben, gibt es aktuelle IOMMU-Implementierungen, die Umsetzungen zwischen 32-Bit-Adressen des PCI-Busses und physischen 64-Bit-Adressen des Hauptspeichers vornehmen. Diese IOMMUs sind in der PCI-Bridge implementiert und stellen sicher, dass Geräte nur auf solche physischen Hauptspeicher-Adressen zugreifen können, für die eine gültige Abbildung in der IOMMU vorliegt.

In Abschnitt 2.5.4 wurde bereits erläutert, dass mit dieser gebräuchlichen Variante bereits eine erhebliche Verbesserung der Systemsicherheit erreicht werden kann, da Geräte nicht mehr auf beliebige physische Adressen zugreifen können. Es bleibt immer noch der Nachteil, dass zwei Busmaster-fähige Geräte auf den selben IO-Bereich zugreifen können, wenn sie zur gleichen Zeit aktiv sind.

Ein IO-Adressraum pro PCI-Gerät Für die vollständige Separierung von PCI-Geräten muss eine IOMMU logisch zwischen jedes Gerät und den PCI-Bus eingefügt werden.

Technisch realisierbar wäre allerdings auch *eine physische* IOMMU als Kontrollinstanz auf dem Bus, die dem Datenverkehr lauscht und illegale Transaktionen abbricht, wie in Abbildung 3.3 dargestellt. Dies resultiert aus der Überlegung, dass während eines bestimmten Zeitabschnitts immer nur ein Gerät Busmaster sein kann.

Dem IO-Adressraum eines Busmasters ist die IO-Tabelle zugeordnet (siehe Abschnitt 3.2.2), die angibt, welche Adressen der Busmaster lesen oder schreiben darf. Um zwischen mehreren IO-Adressräumen zu unterscheiden, benötigt eine IOMMU ebenfalls Informationen über den unmittelbar aktiven Busmaster. Diese Information wird vom Bus-Arbiter bereitgestellt, der Teil des PCI-Chipsatzes ist (vgl. Abschnitt 2.5.3).

Eine Integration einer solchen IOMMU in die PCI-Bridge erscheint möglich und sinnvoll. Die IO-Tabellen der Geräte könnten entweder in dediziertem Speicher in der PCI-Bridge oder im Hauptspeicher abgelegt werden.

3.2.4 IOMMU-Implementierung in Software

Steht keine IOMMU-Implementierung in Hardware zur Verfügung, müssen Treiber für Geräte mit Busmaster-DMA-Modus anders gekapselt werden.

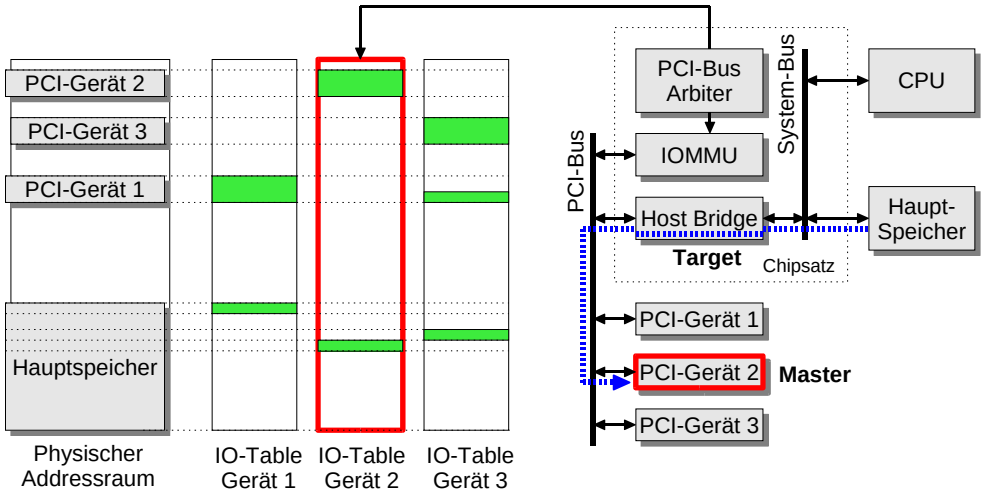


Abbildung 3.3: *Eine logische IOMMU pro PCI-Gerät wird durch Multiplexen einer physischen IOMMU im PCI-Chipsatz implementiert.* Die IOMMU bekommt vom Bus-Arbiter die Information, welches Gerät gerade Master ist und wählt dessen IO-Tabelle aus. Zugriffe, für die in der IO-Tabelle keine gültigen Einträge vorhanden sind, werden abgebrochen. In diesem Beispiel greift Gerät 2 (Master) lesend über die PCI-System-Bridge auf den Hauptspeicher (Target) zu. Gültige Einträge in den IO-Tabellen sind grün dargestellt.

DMA-Zugriffe können auch durch Virtualisierung kontrolliert werden. Virtualisierung bedeutet hier, dass die Zugriffe auf die DMA-Einheit des Gerätes abgefangen und an eine vertrauenswürdige Kontrollinstanz in einem separaten Adressraum, den *Mediator*, weitergeleitet werden (siehe Abbildung 3.4). Den folgenden Algorithmus haben wir erstmals in [26] veröffentlicht:

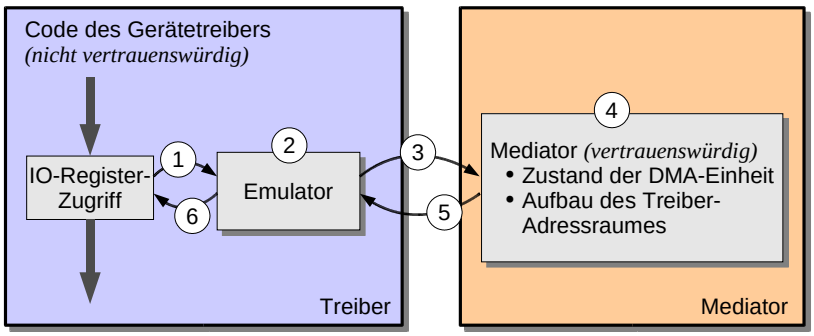


Abbildung 3.4: *DMA-Zugriffe werden durch Virtualisierung der DMA-Einheit des Gerätes kontrolliert.* Der Mediator ist vor dem Gerätetreiber durch Adressraumgrenzen geschützt.

Zugriffe auf DMA-Register lassen sich durch Sicherheitsmechanismen der Hardware kontrollieren. Auf x86-Systemen sind das entweder in der MMU implementierte Seitentabellen für *memory-mapped IO* oder IO-Bitmasken für *Port IO*. Ein Zugriff auf einen IO-Port im überwachten Bereich löst eine Exception aus (1). Ein *Emulator* interpretiert die unterbrochene Assembler-Anweisung. Die Exception liefert sowohl die Adresse der auslösenden Anweisung als auch die referenzierte Adresse. Es fehlt aber noch die Information über die Datenbreite und das zu schreibende Daten-

wort. Der Emulator muss diese Werte durch Code-Analyse der die exception-auslösende Anweisung ermitteln (2).

Die ermittelten Werte werden an einen Mediator weitergeleitet (3). Dieser überprüft, ob der Zugriff erlaubt ist und führt ggf. die IO-Operation aus (4). Für diese Aufgabe muss der Mediator wissen, welche Aktion bei einem Gerät durch das Beschreiben bzw. Lesen eines bestimmten Datenwortes an einer bestimmten Adresse zu einem bestimmten Zeitpunkt ausgelöst wird. Der Mediator muss dafür einen Teil des inneren Zustandes des Gerätes inklusive der DMA-Einheit mitführen und weiterhin Informationen über die dem Adressraum des Gerätetreibers zugeordneten Kacheln besitzen (Abbildung 3.5).

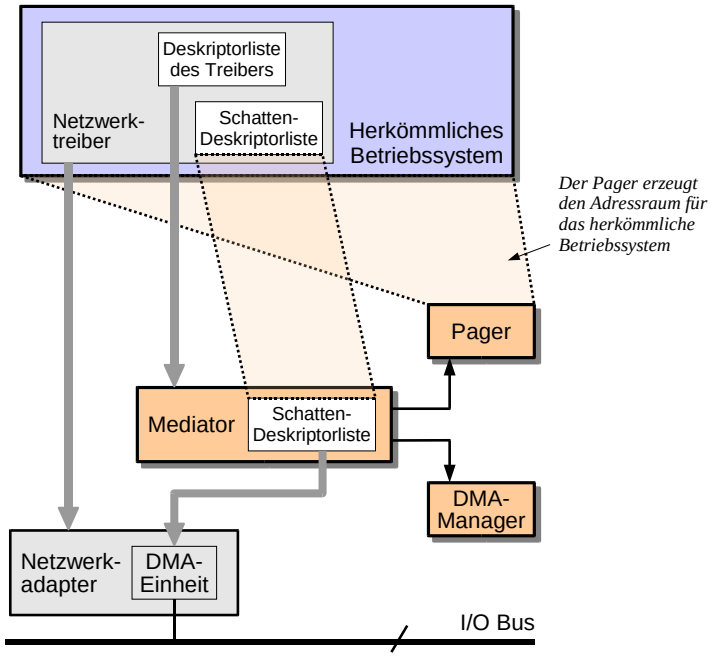


Abbildung 3.5: *Architektur mit DMA-Mediator.* Der bzw. die Pager sind für den Aufbau des Adressraumes des Legacy-OS zuständig. Der Mediator arbeitet mit den Pagern zusammen und kennt somit die dem Legacy-OS zugeordneten physischen Kacheln. Der Mediator sendet anstelle der Original-Deskriptorliste eine *Kopie*, die Schatten-Deskriptorliste, an die DMA-Einheit des Gerätes.

Nachdem der Mediator die IO-Operation ausgeführt hat, überspringt der Emulator die unterbrochene Assembler-Anweisung durch Setzen des Programmzählers auf die nächstfolgende Anweisung (6).

Bei Vorliegen des Treiber-Quellcodes ist es möglich, den Treiber durch Makros o. ä. zu patchen, um bei IO-Operationen direkt zum Emulator zu verzweigen und somit die Exception zu vermeiden.

Sowohl Emulator als auch Mediator sind *geräteabhängige* Implementierungen und müssen jeweils an neue Geräte bzw. Klassen von Geräten angepasst werden. In Abschnitt 4.3.5 wird gezeigt, dass die ausschließliche Virtualisierung der DMA-Zugriffe oft wesentlich leichter zu implementieren und damit leichter zu verifizieren ist, als den kompletten Treiber neu zu implementieren. Nur ein Teil der Spezifikation des Gerätes muss berücksichtigt werden, nämlich der Teil, der beschreibt, wie die DMA-Einheit programmiert wird.

Der Emulator im Adressraum des Treibers braucht nicht vertrauenswürdig zu sein. Nur der Hardware-Mechanismus zum Kontrollieren der Zugriffe sowie der Mediator müssen vertrauenswürdig sein. Der Mediator läuft aus diesem Grund in seinem eigenen Adressraum. Falls der Emulator fehlerhaft ist oder durch den Treiber kompromittiert wurde, kann im schlimmsten Fall die Verfügbarkeit des Dienstes beeinträchtigt werden.

Der Gerätezustand

Wie oben dargestellt, benötigt der Mediator Informationen, welche Aktionen des Gerätes mit dem Zugriff auf ein Geräteregister verbunden sind. Für die Übertragung von Daten per Busmaster-DMA weist der Treiber dem Gerät einen bestimmten Bereich im physischen Adressraum zu.

Im einfachsten Fall handelt es sich um genau *einen DMA-Bereich* pro Aktion des Gerätes, der diesem durch Schreiben auf ein Geräteregister übergeben werden soll. Der Mediator verifiziert diesen Bereich und schreibt dessen Parameter in das Geräteregister. Nachdem das Gerät eine Aktion ausgeführt hat, erhält es einen neuen Bereich, der wieder verifiziert wird. Der alte Bereich ist ab diesem Zeitpunkt nicht länger in Benutzung.

Viele Geräte können allerdings gleichzeitig mit mehreren DMA-Bereichen arbeiten und erwarten eine entsprechende Datenstruktur, die viele Regionen enthalten kann. Der Mediator muss den Aufbau der Struktur kennen und verifizieren können. Um zu vermeiden, dass ein Treiber die Struktur nachträglich verändert, muss der Mediator eine Kopie der Struktur an das Gerät senden. Je komplizierter die Struktur aufgebaut ist, desto mehr Aufwand entsteht im Mediator.

Weiterhin muss der Mediator wissen, wie lange DMA-Bereiche für das Gerät *gültig* sind, d. h. bis zu welchem Zeitpunkt das Gerät mit diesen Adressen arbeitet. Ein DMA-Bereich kann für ein Gerät so lange gültig sein, bis er durch einen anderen ersetzt wurde oder für die Zeit, während der er durch Zustandsbits (als Teil der Datenstruktur) als aktiv markiert ist. Die Benachrichtigung über eine Zustandsänderung des Gerätes erfolgt meist mittels Interrupt.

Pinning

Das Aufheben des Pinning der Seiten der DMA-Region muss bei der softwaregestützten Lösung im Einvernehmen mit dem Mediator stattfinden. Der Mediator ist die einzige Komponente, die mit Sicherheit weiß, ob eine Adresse, die an das Gerät geliefert wurde, noch aktiv ist oder nicht.

Atomarer Zugriff auf Geräteregister

Bei Zugriffen auf Geräteregister sind folgende Besonderheiten zu beachten:

- Die *zeitliche Reihenfolge* der Zugriffe ist entscheidend, sowohl lesende und schreibende Zugriffe auf *ein* Register als auch Zugriffe auf verschiedene Register.

- Manchmal ist der *zeitliche Abstand* zwischen zwei Zugriffen entscheidend, siehe z. B. die Programmierung der PC-CMOS-Uhr (Abschnitt 3.1).

Die zeitliche Reihenfolge von Zugriffen auf Register von Geräten, deren DMA-Zugriffe virtualisiert werden, wird dadurch sichergestellt, dass der DMA-Mediator mittels synchroner IPC in

den Zugriff mit einbezogen wird. Der Programmfluss wird an der Stelle des Registerzugriffs unterbrochen und nach Aufruf des Mediators sowie Empfang von dessen Antwort an dieser Stelle fortgesetzt. Dies gilt auch für Mehrprozessorsysteme.

Durch die zusätzlichen Kosten der IPC können sich zeitliche Abweichungen bei Zugriffen auf Geräteregister ergeben. Dies stellt aber auch schon dann ein Problem dar, wenn der Gerätetreiber in einer preemptiven Mehr-Prozess-Umgebung als Nutzerprozess ausgeführt wird.

Nebenläufigkeiten *innerhalb* des Treibers treten üblicherweise nur zwischen der Interrupt-Behandlungsroutine und den Worker-Threads, die für den Start neuer Aufträge verantwortlich sind, auf. Diese Aktivitäten müssen unabhängig vom Mediator synchronisiert werden.

Einschränkungen der DMA-Virtualisierung

Einige Klassen von Geräten sind für die Virtualisierung von Busmaster-DMA nicht geeignet.

Viele moderne Geräte wie SCSI-Hostadapter, Gigabit-Netzwerkadapter oder WLAN-Adapter sind programmierbar: Der Treiber lädt während der Initialisierung eine Firmware direkt in das Gerät. Operationen des Gerätes werden durch Kommunikation mit der Firmware durch Nutzung eines proprietären Protokolls initiiert.

Die Firmware-Aufrufe enthalten physische Adressen für die Benutzung mit DMA-Transaktionen und können zwar kontrolliert werden, jedoch gibt es folgende Einschränkungen:

1. Meist ist das Protokoll zwischen Treiber und Firmware nicht dokumentiert. Treiber von verschiedenen Herstellern nutzen potenziell verschiedene Firmware und unterschiedliche Protokolle. Meine Implementierung beruht aber auf genauer Kenntnis der kritischen Geräteeigenschaften.

2. Die Korrektheit der Firmware kann nicht garantiert werden. Wir können somit dem Gerät, zu dem die Firmware gehört, nicht trauen. Ein Beispiel: Der Linux-Treiber für die SCSI-Hostcontroller-Familie NCR53c8xx enthält Firmware mit etwa 2000 Anweisungen (entspricht einer Größe des Binärcodes von etwa 8 KB). Die Firmware dieses Treibers liegt als Quellcode vor und könnte somit zumindest untersucht werden. Allerdings gibt es für aktuelle Hardware immer mehr Beispiele von Treibern, die um Binärcode nur Wrapper legen (z. B. für Modems, WLAN-Adapter und Sound-Karten).

Die Kontrolle der Schnittstelle zwischen Firmware auf dem Gerät und dem Gerät selbst ist nicht möglich, weil normalerweise kein genereller Kontrollmechanismus im Gerät implementiert ist.

Beispiele für DMA-Virtualisierung

Im folgenden werden einige Geräteklassen beschrieben, für die eine Virtualisierung der DMA-Einheit mit überschaubarem Aufwand möglich ist.

Netzwerkadapter Netzwerkgeräte nutzen üblicherweise Deskriptorlisten für die Kommunikation mit dem Treiber. Vereinfacht dargestellt enthält ein Deskriptor die physische Adresse und die Größe eines Pakets, das empfangen oder gesendet werden soll, und ein Eigentums-Bit. Ist dieses Bit gesetzt, darf das Gerät auf das Paket zugreifen. Der Netzwerkadapter darf jedoch niemals auf ein Paket mit gelöschtem Eigentums-Bit zugreifen.

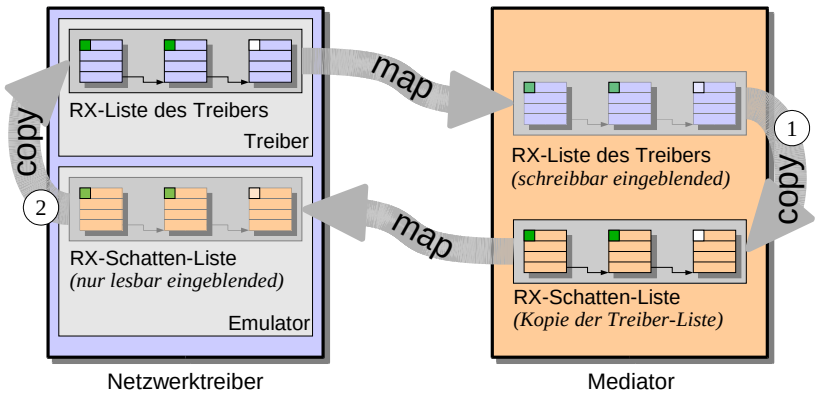


Abbildung 3.6: *Virtualisierung der DMA bei Netzwerkadaptern für die Empfangsrichtung.* Die Liste der Empfangs-Deskriptoren des Netzwerktreibers ist zusätzlich in den Adressraum des Mediators eingeblendet. Bei einem Schreibzugriff auf das entsprechende Register der DMA-Einheit des Gerätes überprüft der Mediator die Liste vollständig und legt eine Kopie an, mit der das Gerät dann arbeitet (1). Nach Abarbeitung des Auftrages muss die Original-Liste mit der Kopie abgeglichen werden, damit der Treiber die Änderung der Status-Werte (grün markiert) „sieht“ (2). Aus Sicherheitsgründen ist die Liste, die an das Gerät gesendet wird, nur lesbar in den Adressraum des Treibers eingeblendet.

Um Pakete vom Netzwerk zu empfangen, stellt der Treiber eine Deskriptorliste zur Verfügung. Nachdem der Treiber die Adresse der Liste in ein Geräteregister geschrieben hat, beginnt der Netzwerkadapter die Puffer mit Paketen zu füllen, bis der erste Deskriptor mit gelöschtem Eigentums-Bit erreicht ist. Nachdem ein oder mehrere Pakete empfangen wurden, generiert das Gerät einen Interrupt.

Um Pakete zu senden, schreibt der Treiber die Adresse einer Liste mit Deskriptoren zu sendender Pakete in ein Geräteregister. Der Netzwerkadapter sendet die Pakete und löscht jeweils das Eigentums-Bit. Es werden solange Pakete gesendet, bis das erste gelöschte Eigentums-Bit gefunden wird. Nach dem Senden wird ggf. ein Interrupt generiert, sofern dies im Deskriptor so angegeben wurde.

Die DMA-Operationen für Sende- und Empfangsrichtung können bei Netzwerkadaptern wie folgt überwacht werden (vgl. Abbildung 3.6 für die Empfangsrichtung):

Der Mediator muss Schreibzugriffe auf die Register des Netzwerkadapters abfangen, in die die Adressen der Deskriptorlisten geschrieben werden. Eine validierte *Kopie* der Deskriptorliste, die Schatten-Deskriptorliste, wird an das Gerät geliefert. Die Einträge dieser Liste enthalten die physischen Adressen der vom Treiber bereitgestellten Puffer. Weil das Gerät mit einer vom Mediator erstellten Kopie arbeitet, kann der Treiber die Puffer-Adressen nicht mehr verändern, nachdem sie an das Gerät weitergegeben wurden. Der Mediator stellt sicher, dass die Adressen der Puffer auf Speicherbereiche verweisen, die dem Treiber zugeordnet sind.

Am Ende eines Auftrages zum Empfangen oder Senden von Paketen wird ein Interrupt generiert. Bevor die Interrupt-Behandlungsroutine die Deskriptorliste inspizieren kann, muss der Status der Deskriptoren zurück auf die Liste des Treibers kopiert werden, damit der Treiber die Ergebnisse der IO-Operation sehen kann.

Diese Methode funktioniert gut mit Netzwerkadaptern, die mit Deskriptorlisten arbeiten, die von den IO-Puffern getrennt sind (z. B. Digital DS2114x Tulip [40]). Liegen die Puffer hardwarebedingt unmittelbar hinter den Deskriptoren, so ist die Virtualisierung aufwändiger. Die Netzwerkadapter Intel 8255x [43] sehen dieses Speicherlayout für Empfangs-Deskriptoren vor. Die Puffer können damit nicht von den Deskriptoren getrennt werden und müssten ebenfalls kopiert werden.

Die Virtualisierung der Sende- und Empfangs-Deskriptoren funktioniert nur, wenn der Netzwerkadapter nach erfolgreichem Senden bzw. Empfangen von Paketen Interrupts generiert. Anderenfalls kann keine ereignisgesteuerte Synchronisation zwischen der Deskriptorliste des Treibers und der Deskriptorliste des Mediators erfolgen. Periodisches Abfragen (Polling) wäre an dieser Stelle zwar denkbar, aber bei Netzwerkadaptern aus Performance-Gründen nicht praktikabel.

Bei den beschriebenen Netzwerkadaptern müssen nur die *schreibenden* Zugriffe auf die Geräteregister für die Sende- und Empfangs-Deskriptorlisten sowie für den Start/Stopp des Sende- und Empfangsprozesses virtualisiert werden. *Lesende* Zugriffe brauchen nicht überprüft werden, da sie den Zustand der DMA-Einheit nicht verändern.

IDE-Controller Wesentlich einfacher ist die Virtualisierung von Zugriffen auf Geräteregister für die Steuerung von Busmaster-DMA bei IDE-Controllern. Diese nehmen eine Liste von Blöcken mit physischen Adressen und Größen von Puffern entgegen, die auf Massenspeicher-Medien geschrieben oder von dort gelesen werden sollen. Der Mediator muss nur Schreibzugriffe auf das entsprechende Geräteregister kontrollieren und die Liste auf gültige Puffer-Adressen überprüfen. Da eine Liste mit Blöcken meist viele Blöcke enthält, ist der Overhead des Mediators pro Byte viel geringer als bei Netzwerkadaptern.

Bei IDE-Controllern muss nur der *schreibende* Zugriff auf das Register für die Liste mit den Blöcken virtualisiert werden. Die DMA-Einheit von IDE-Controllern wird über einen generischen Teil des Geräte-Interfaces programmiert, der sich nur selten ändert – siehe z. B. [48].

Kapitel 4

Implementierung

Die verschiedenen Techniken zur Kapselung von Standard-Betriebssystemen wurden exemplarisch in L^{4}Linux 2.2.26 implementiert. Eine Übertragung auf aktuelle L^{4}Linux-Versionen und auf andere Standard-Betriebssysteme ist möglich (siehe Abschnitt 4.1.5). Insbesondere die Technik der DMA-Virtualisierung lässt sich, im Rahmen der in Abschnitt 3.2.4 gezeigten Einschränkungen, auf beliebige Gerätetreiber übertragen.

4.1 Linux mit eingeschränkten IO-Rechten

In Abschnitt 2.4.3 wurde gezeigt, dass eine nicht vertrauenswürdige Anwendung auf der x86-Architektur nicht mit IOPL-3-Rechten ausgestattet sein darf, weil sie sonst ohne Limitation auf alle IO-Ports zugreifen sowie die Prozessor-Interrupts sperren kann. Eine Zugriffskontrolle über die IO-Bitmap erfolgt durch den Prozessor nur, wenn der aktuelle Kontext im Nutzermodus mit IOPL-0-Rechten ausgeführt wird. In den folgenden Abschnitten wird dargestellt, wie die relevanten Stellen von Linux geändert wurden, damit Linux mit eingeschränkten IO-Rechten ausführbar ist.

4.1.1 Globales `cli`-Lock

Für die in Abschnitt 2.4.3 dargestellte Semantik von `cli`/`sti` wurde eine entsprechende Implementierung für den Linux-Server gewählt. Als Synchronisierungsprimitiv werden neben der vom Mikrokern Fiasco bereitgestellten synchronen IPC atomare Operationen (Dekrement, Inkrement) auf Speicher verwendet.

Linux 2.2 implementiert drei verschiedene Arten von Aktivitäten [77]:

- *Top-Halves* behandeln Hardware-Interrupts. Es ist sichergestellt, dass ein Top-Half nur von anderen Top-Halves unterbrochen werden kann.
- *Bottom-Halves* sind durch Top-Halves unterbrechbar und führen umfangreicheren Code aus. Die Trennung in Top-Half und Bottom Half wird vorgenommen, um sicherzustellen, dass Interrupts so wenig wie möglich am Interrupt-Controller blockiert werden.
- Die *Hauptaktivität* des Kerns behandelt Kerneintritte der Prozesse (Systemaufrufe, Exceptions, Seitenfehler).

Unter L^{4}Linux werden diese Aktivitäten auf Threads mit unterschiedlichen Prioritäten abgebildet [25]: Der Service-Thread wird mit der geringsten Priorität ausgeführt. Die Priorität des Bottom-Half-Threads ist größer als die des Service-Threads, und die Top-Halves werden von Threads mit der höchsten Priorität behandelt.

Bei der Implementierung des `cli`-Locks ist zu beachten, dass IPC unter Fiasco immer mit impliziter Prioritäts- und Zeitscheibenvererbung stattfindet, sofern dies nicht explizit ausgeschlossen wird. Ferner sollte das Lock ohne *Busy Waiting* auskommen.

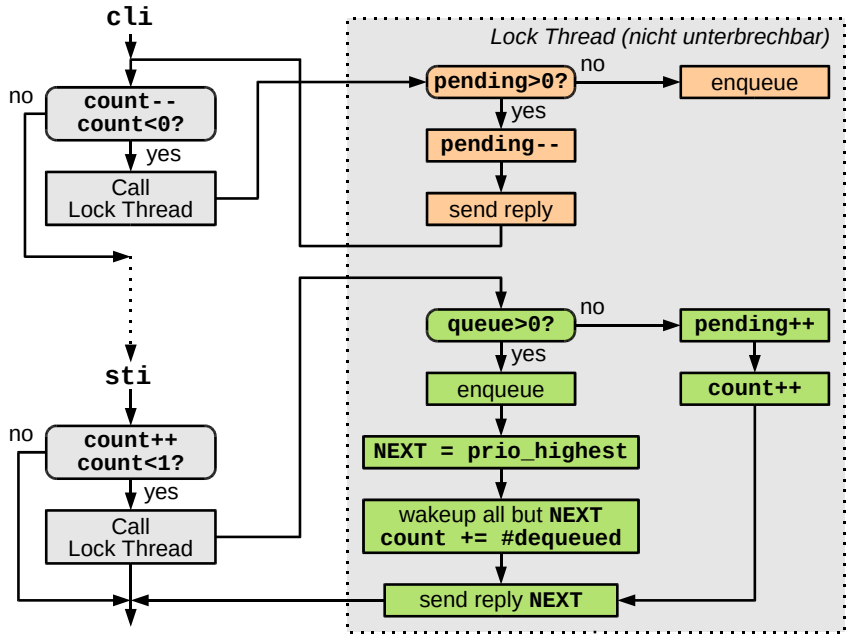


Abbildung 4.1: *Implementierung des `cli`-Locks.* Der Lock-Thread besitzt die höchste Priorität des L^{4}Linux-Servers und ist damit nicht durch lokale Aktivitäten von Linux preemptierbar. Im Unterschied zu einer üblichen Semaphor-Implementierung erhält ein blockierter Thread nach Aufwecken durch einen Thread, der `cli` ausführt, nicht automatisch das Lock, sondern muss sich erneut bewerben.

Kernelement des Locks ist ein dedizierter Lock-Thread mit der höchsten Priorität des Servers, der deshalb durch keinen lokalen Thread unterbrochen werden kann (Abbildung 4.1). Dies ist zwar nur für Einprozessorsysteme garantiert, schränkt die Anwendung hier aber nicht ein, da sich `cli`/`sti` auf Mehrprozessorsystemen ebenfalls nur auf die lokale CPU auswirken.

Threads, die einen kritischen Abschnitt betreten (also `cli` ausführen) wollen, synchronisieren sich zunächst über einen atomaren Zähler (in der Abbildung als `counter` bezeichnet). Falls das Lock schon belegt ist, besitzt der Zähler nach dem Dekrementieren einen negativen Wert. In diesem Fall sendet der unterlegene Thread eine IPC an den Lock-Thread und blockiert.

Wenn das Lock vom Eigentümer freigegeben werden soll (dies entspricht der Ausführung der `sti`-Anweisung), sendet dieser eine Freigabe-IPC an den Lock-Thread. Der alte Eigentümer blockiert nun ebenfalls. Der Lock-Thread wählt nun den Thread mit der höchsten Priorität aus (in der Abbildung als `NEXT` bezeichnet). Allen anderen Threads sendet er eine Antwort und aktiviert sie damit, ohne zu ihnen umzuschalten. Ferner wird der Zähler auf Null gesetzt (dies bedeutet, dass genau ein Thread das Lock belegt).

Der weiterhin aktive Lock-Thread sendet nun dem höchstprioren Thread eine Antwort. Handelt es sich dabei um den alten Lock-Eigentümer, fährt dieser mit der Bearbeitung des Codes hinter der `sti`-Anweisung fort. Im anderen Fall bewirbt sich ein höherpriorer Thread um das `cli`-Lock und wird neuer Eigentümer.

Alle anderen wartenden Threads sind in dieser Phase bereit. Sobald sie aktiv werden, bewirbt sich ein jeder Thread erneut um das Lock (dekrementieren des Zählers und IPC an den Lock-Thread).

Vor dem Aufruf einer Interrupt-Service-Routine muss sich der Interrupt-Handler um das `cli`-Lock bewerben. Dieser Algorithmus garantiert die Beibehaltung des beschriebenen Prioritäts-Schemas zwischen den Threads des L^{4}Linux-Servers.

Kosten Die Ausführung der Instruktion `cli` gemeinsam mit `sti` kann je nach Prozessor zwischen 10 und 86 Prozessorzyklen verbrauchen (siehe Anhang B.1). In L^{4}Linux wurden diese Instruktionen durch Funktionsaufrufe zum Betreten und Verlassen des `cli`-Locks ersetzt. Sofern sich nicht mehrere Threads gleichzeitig um das Lock bewerben (*Lock Contention*), fallen keine IPCs zum Lock-Thread an, weil dann nur der Zähler des Locks atomar dekrementiert (`cli`) bzw. inkrementiert (`sti`) wird. In diesem Fall liegen die Kosten in der gleichen Größenordnung wie beim Ausführen von `cli` und `sti`.

Falls Lock Contention vorliegt, entstehen zusätzliche Kosten für mindestens vier IPC-Operationen:

1. Thread *B* stellt fest, dass das `cli`-Lock bereits belegt ist und ruft deshalb den Lock-Thread *L* auf.
2. Der Thread *A*, der das Lock belegt hat, merkt, dass ein anderer Thread warten musste und sendet deshalb bei Freigabe des Locks eine entsprechende IPC an den Lock-Thread *L*.
3. *L* sendet jeweils eine Antwort an *A* und *B*.

Eine Roundtrip-IPC zwischen zwei Threads innerhalb eines Adressraumes kostet auf einem Pentium 4 unter Fiasco etwa 1450 Takte bzw. 3000 Takte, falls nicht der optimierte Pfad genommen werden kann (siehe Anhang C.1).

4.1.2 Atomares Aufwecken

Systemaufrufe von Linux-Programmen werden unter L^{4}Linux auf synchrone IPC abgebildet. Der Linux-Server blockiert im Normalfall in einer IPC-Operation, in der er auf neue Aufträge von Linux-Programmen wartet. Tritt in dieser Zeit ein Hardware-Interrupt auf, muss die entsprechende Top-Half-Routine in der Lage sein, den Server aufzuwecken.

Das Aufwecken eines in einer IPC blockierten Threads kann unter Fiasco auf zwei verschiedene Arten erfolgen: Entweder wird dem Thread eine IPC gesendet, die der aufzuweckende Thread empfängt und damit die IPC beendet, oder die IPC wird mit dem Systemruf `lthread_ex_regs()` abgebrochen. Alternativ könnte der Thread durch Angabe eines entsprechenden Timeouts auch nach Ablauf einer Zeitspanne ohne IPC automatisch aufgeweckt werden. Allerdings würde dies bei zu kurzem Timeout zu Busy-Waiting des Service-Threads führen und bei zu langem Timeout zu einer ungewollten Verzögerung der Interrupt-Behandlung.

Die Schwierigkeit besteht darin, atomar ein Flag `sleeping` als Information zu setzen, dass der Service-Thread des Linux-Servers schläft und dann den IPC-Systemaufruf zu starten. Wird der Thread zwischen Setzen von `sleeping` Flags und Aufsetzen der IPC unterbrochen, muss darauf gesondert reagiert werden. Ferner ist es möglich, dass der Thread bereits eine IPC eines Linux-Prozesses empfangen hat aber trotzdem aufgeweckt werden soll, weil `sleeping` noch gesetzt ist.

Bei der Implementierung habe ich mich für die Benutzung der L4-Systemfunktion `lthread_ex_regs()` entschlossen. Mit diesem Systemruf kann der Zustand eines Threads, unter anderem der Programmzähler, verändert werden. Befindet sich der Thread in einer IPC, wird diese abgebrochen (Abbildung 4.2). Befindet sich der Thread noch nicht oder nicht mehr in der IPC, wird der Programmzähler trotzdem umgesetzt, sofern `sleeping` gesetzt war. Der Code nach dem IPC-Systemruf muss unterscheiden können, ob IPC ausgeführt wurde oder nicht.

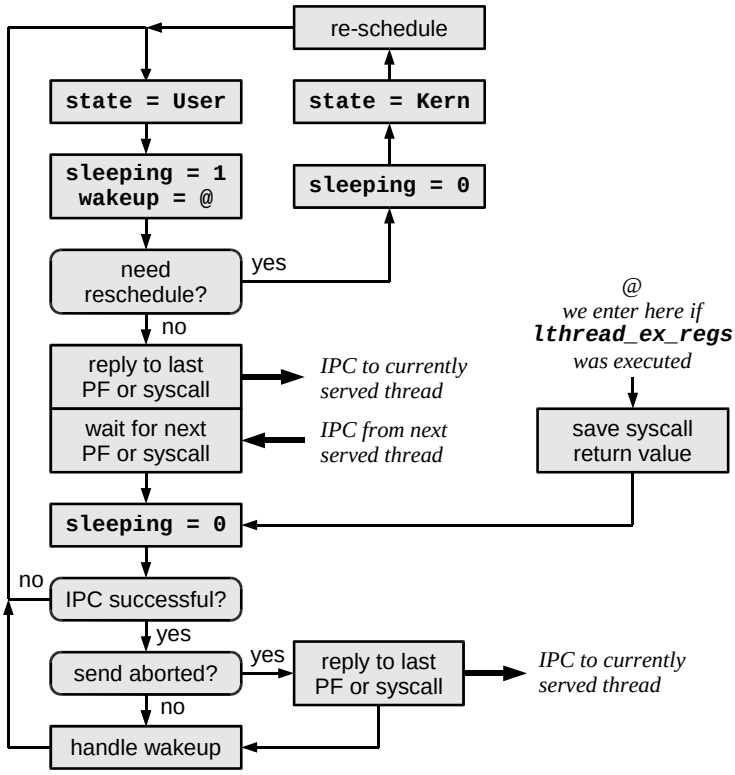


Abbildung 4.2: *Aufwecken des Linux-Service-Threads.* Dieser wartet auf die nächste eingehende IPC von einer Linux-Anwendung (Systemruf oder Seitenfehler). Falls in dieser Zeit ein Interrupt auftritt, muss der Service-Thread ggf. aufgeweckt werden, um eine neue Scheduling-Entscheidung zu treffen.

4.1.3 Programmierung des Interrupt-Controllers

Der Mikrokern ist auf einen Zeitgeber-Interrupt als Trigger für preemptive Scheduler-Aufrufe angewiesen. Eine mögliche Zeitquelle für den Kern ist der *Programmable Interval Timer* (PIT) [39], da der Ausgang von Kanal 0 des PIT auf PCs üblicherweise mit dem *Programmable Interrupt Controller* (PIC) [38] verbunden ist, siehe z. B. [48].

Der Interrupt-Controller wird aus zwei Gründen nicht von L^{4}Linux, sondern von einem vertrauenswürdigen externen Server namens *Omega0* [63] programmiert:

1. Usermode-Programme dürfen nicht in der Lage sein, die Zeitquelle für den Scheduler abzuschalten (z. B. durch Sperrung des Zeitgeber-Interrupts), da sonst Prozesse am Ende ihrer

Zeitscheibe nicht mehr unterbrochen werden und mehr Zeit verbrauchen können, als ihnen zusteht.

2. Der Kern ist für die Zuordnung von Interrupts an Threads verantwortlich. *Vor* der Zuordnung eines bestimmten Interrupts an einen Thread wird die Interrupt-Leitung zuerst durch den Kern gesperrt, um zu verhindern, dass Interrupts ausgelöst werden, bevor der Thread diese behandeln kann. Nach Aufhebung einer Zuordnung wird der entsprechende Interrupt ebenfalls vom Kern gesperrt. Der Kern greift also einerseits selbst auf Geräteregister des PIC zu, andererseits sind L4-Usermode-Programme dafür verantwortlich, Interrupts nach Empfangen am PIC zu bestätigen. Die dafür notwendigen Zugriffe auf den PIC müssen mit dem Kern synchronisiert werden. Dies ist unter Fiasco nur durch Abschalten der Prozessor-Interrupts mittels `cli` möglich. Dafür sind *IOPL-3-Rechte* notwendig.

Tritt ein Hardware-Interrupt auf, wird dieser vom Kern dem Omega0-Server zugestellt. Dieser maskiert und bestätigt den Interrupt zuerst am PIC und sendet dann eine IPC an den Thread, der sich für den betreffenden Interrupt registriert hat. Nachdem der Thread aus der Interrupt-Behandlungsroutine zurückgekehrt ist und auf die nächste IPC wartet, demaskiert Omega0 den Interrupt wieder und wartet auf den nächsten Interrupt vom Kern. Diese Behandlung stellt einerseits sicher, dass der nächste Interrupt erst dann auftritt, wenn der vorherige Interrupt am Gerät behandelt wurde.

Die Einfügung einer Indirektionsstufe in den Interrupt-Pfad bedingt zusätzliche Kosten, die vermieden werden könnten, wenn der Kern selbst die Bestätigung der Interrupts vornähme. Die L4-API-Spezifikation X.2 [91] definiert ein zu Omega0 ähnliches Protokoll für die Behandlung von Hardware-Interrupts, womit ein dedizierter Interrupt-Server nicht mehr notwendig wäre. Fiasco basiert in der aktuellen stabilen Implementierung allerdings auf der V.2-Spezifikation [57], die keine Vorgehensweise für die Interrupt-Behandlung definiert. In einer speziellen Betriebsart[1] bestätigt der Fiasco-Kern Interrupts am Interrupt-Controller bereits *vor* der Zustellung an den Nutzerprozess. In dieser Betriebsart können allerdings nur flankengetriggerte Interrupts behandelt werden. PCI-Geräte generieren jedoch meist pegelgetriggerte Interrupts. Ein solcher liegt nach Bestätigung sofort wieder am PIC an, falls die Ursache des Interrupts am entsprechenden Gerät noch nicht behoben wurde.

Im Vergleich mit Linux fallen im Interrupt-Pfad folgende zusätzliche Kosten an:

- Standard-Linux kann den Wert des Interrupt-Masken-Registers (IMR) cachen, weil der Linux-Kern die einzige Instanz im System ist, die auf den PIC zugreift. Bei Verwendung von Omega0 kann der Wert des IMR nicht gecacht werden, weil Fiasco dieses Register ebenfalls potenziell verändert. Dadurch fallen zwei zusätzliche lesende Zugriff auf das IMR an (einmal vor und einmal nach der Interrupt-Behandlung).

- Fiasco benutzt im Unterschied zu Linux den *Special Fully Nested Mode* des PIC [38]. In dieser Betriebsart werden im Unterschied zum üblichem PIC-Modus Interrupts des Slave-PIC-Controllers auch dann an die CPU weitergeleitet, wenn bereits ein solcher Interrupt aussteht. In dieser Betriebsart fällt während der Interrupt-Bestätigung ein weiterer lesender Zugriff auf ein IO-Register des PIC an.

[1] Diese Betriebsart kann durch den Parameter `-always_irqack` aktiviert werden.

Bei der Implementierung mit Omega0 werden im Vergleich mit Linux auf einem Pentium III pro Interrupt-Zustellung zusätzliche Kosten in Höhe von etwa 6000 Takten erwartet. Diese setzen sich zusammen aus den Kosten für zusätzliche Zugriffe auf den PIC mittels Port-IO (siehe Tabelle 4.1) sowie zusätzliche direkte und indirekte Kosten durch IPC. Für den Celeron 4 werden etwa 8000 zusätzliche Taktzyklen erwartet.

	MHz	IMR Lesen	IMR Schreiben
Pentium III (Coppermine)	800	1649 Zyklen (2,1 µs)	1649 Zyklen (2,1 µs)
Celeron 4 (Northwood)	2000	1746 Zyklen (0,9 µs)	1687 Zyklen (0,8 µs)

Tabelle 4.1: *Kosten für eine Lese- bzw. Schreiboperation des PIC-Interrupt-Masken-Registers (IMR).* Die Werte hängen auch vom Chipsatz ab. In Anhang A sind beide Rechner genauer beschrieben.

4.1.4 Programmierung der CMOS-Uhr

Wie in Abschnitt 3.1 gezeigt, ist das Auslesen und das Setzen der CMOS-Uhr zeitkritisch. Linux liest den Zähler der CMOS-Uhr während der Boot-Phase und stellt die Systemzeit darauf ein.

Unter L^{4}Linux wird die CMOS-Uhr nicht mehr direkt ausgelesen, sondern diese Aufgabe übernimmt ein vertrauenswürdiger externer Server, der beim Booten des Betriebssystems die Uhr einmalig ausliest und dabei die Interrupts sperrt. Der Server berechnet dann einen Umrechnungsfaktor relativ zum *Time Stamp Counter* (TSC) der CPU und kann darauf basierende Zeitanfragen von Clients beantworten, ohne auf die CMOS-Uhr zugreifen zu müssen. An dieser Stelle wird die Drift zwischen CMOS-Uhr und TSC vernachlässigt.

Neben dem Kern greifen auch Anwendungen direkt auf die CMOS-Uhr zu, die sich dafür mit IOPL-3-Rechten ausstatten (Linux-Syscall `iopl`). Ein Beispiel ist das Programm `hwclock`. Da L^{4}Linux im Nutzermodus läuft und nur IOPL-0-Rechte besitzt, kann und darf es der Anwendung keine IOPL-3-Rechte übertragen. Der Code für den Zugriff der CMOS-Uhr von `hwclock` muss deshalb ersetzt werden. Zum Einsatz kommt der in Abschnitt 3.1 beschriebene Try-and-Retry-Algorithmus.

4.1.5 Übertragung auf andere Systeme

In den Abschnitten 4.1.1 bis 4.1.4 wurden Techniken vorgestellt, die es ermöglichen, einen Gerätetreiber bzw. einen Betriebssystem-Kern mit eingeschränkten IO-Rechten auszuführen. Wie in Abschnitt 3.1.3 dargestellt, muss prinzipiell für jeden Einzellfall entschieden werden, wie das Sperren von Interrupts vermieden werden kann. In der Praxis kann fast immer das `cli`-Lock gewählt werden, da der zeitkritische Zugriff auf Geräteregister nur äußerst selten mit `cli` geschützt wird.

Vorraussetzung für das direkte Ersetzen der `cli`/`sti`-Anweisungen ist allerdings die Vorlage des Quelltextes. Linux-Kernmodule in Binärform, bei denen diese Anweisungen fest eincompiliert sind, müssen ggf. unter größerem Aufwand gepatcht werden. Zwar lösen `cli` und `sti` unter eingeschränkten IO-Rechten entsprechende Exceptions aus, dies gilt aber nicht für den `popf`-Befehl, der durch Setzen bzw. Löschen des entsprechenden Flags ebenfalls das Zustellen von Interrupts erlauben bzw. verhindern kann. Damit lässt sich die Semantik von `popf` auf aktuellen x86-Systemen nicht virtualisieren (siehe Abschnitt 2.2.3). Viele Hersteller von Treibern in Binärform kapseln

jedoch die entsprechenden Funktionen in externen Quelldateien und erlauben damit deren Überschreiben.

Die in Abschnitt 4.1.1 vorgestellte Implementierung des `cli`-Locks benutzt atomare Operationen, um einen Wert im Speicher ohne Unterbrechung zu dekrementieren bzw. zu inkrementieren und zu testen, ob der Wert nach dem Dekrementieren kleiner als Null bzw. nach dem Inkrementieren gleich Null ist. Stellt die Architektur Operationen dieser Art nicht zur Verfügung, muss das Lock entweder anders implementiert werden oder die atomaren Befehle müssen mit Kern-Hilfe nachgebildet werden.

Für die zentralisierte Programmierung des Interrupt-Controllers (Abschnitt 4.1.3) muss das zu kapselnde Betriebssystem generell geändert werden oder der Interrupt-Controller muss virtualisiert werden.

4.2 Performance-Betrachtungen auf Standard-Hardware

Durch die gekapselte Ausführung eines Betriebssystems als Anwendung auf einem Mikrokern entstehen zusätzliche Kosten. Die folgenden Abschnitte erläutern Implementierungsdetails, um diesen Overhead möglichst gering zu halten.

4.2.1 Schneller Kernein- und -austritt

Für Mikrokerne sind schnelle Kernein- und -austritte essentiell wichtig, weil Privilegwechsel häufiger auftreten als bei monolithischen Betriebssystemen. In Abschnitt 2.3.4 wurde erläutert, dass unter L^{4}Linux im Vergleich mit Standard-Linux pro Systemruf mindestens ein zusätzlichen Kernein- und -austritt notwendig ist.

Aus Performance-Gründen ist es daher vor allem bei neueren CPUs notwendig, den Kern mit speziellen Befehlen zu betreten bzw. zu verlassen. Auf älteren x86-CPUs wird der Kern üblicherweise per `int`-Befehl betreten und mit `iret` verlassen. Die Kosten für beide Befehle betragen bei Pentium-4-Prozessoren in Summe fast 1000 Zyklen (siehe Anhang B.1).

Wird `sysenter` zum Betreten des Kerns und `sysexit` zum Verlassen des Kerns verwendet, sinken die Kosten beim Pentium 4 auf etwa 150 Takte, dies ergibt eine Einsparung von mehr als 80 %. Beim Pentium III beträgt die Einsparung immer noch 75 %.

4.2.2 Linux im kleinen Adressraum mit eingeschränkten IO-Rechten

Ist an einem Adressraumwechsel mindestens *ein* kleiner Adressraum beteiligt, so können die indirekten Kosten für den Wechsel sinken, da der TLB nicht invalidiert wird und somit weniger TLB-Einträge neu geladen werden müssen (siehe Abschnitt 2.4.2). Anhang B.3 zeigt die Anzahl der TLB-Einträge heutiger x86-Prozessoren sowie die Kosten zum Neuladen der TLB-Einträge für Daten und Code.

Zu erkennen ist der Trend zu immer mehr TLB-Einträgen in der CPU. Damit werden kleine Adressräume immer wichtiger: Je mehr TLB-Einträge vorhanden sind und je länger das Laden eines Eintrags dauert, desto mehr Einträge müssen im *Worst Case* nach einem kompletten Flush neu geladen werden.

Die bisherige Implementierung von Fiasco erlaubte es nicht, L[4]Linux mit eingeschränkten IO-Rechten in einem kleinen Adressraum auszuführen. Im Kontext dieser Arbeit wurde Fiasco entsprechend erweitert.

Durch die Unterstützung kleiner Adressräume und eingeschränkter IO-Rechte im Kern entstehen zusätzliche Kosten:

Behandlung der TLB-Einträge für die IO-Bitmap Wird der Adressraum umgeschaltet, ohne dass das Seitenverzeichnis gewechselt wird, müssen die Einträge für die IO-Bitmap neu geladen und ggf. invalidiert werden. Die direkten Kosten für das Invalidieren von zwei[2] TLB-Einträgen betragen zwischen 92 Takten (Pentium) und 1040 Takten (Pentium 4), siehe Anhang B.1. Weiterhin entstehen durch das Laden der TLBs mit den Abbildungen des neuen Adressraumes indirekte Kosten, die sich allerdings nicht von den Kosten bei Wechsel des Seitenverzeichnisses unterscheiden.

Die Kosten für das Invalidieren der TLBs der IO-Bitmap könnten eingespart werden, wenn die Anzahl der kleinen Adressräume im System auf vier beschränkt würde, da im TSS Platz für maximal fünf auf Seitengrenzen ausgerichtete IO-Bitmaps ist (siehe Abbildung 4.3). Bei Umschalten auf einen kleinen Adressraum müsste dann nur der IO-Bitmap-Offset im TSS geändert werden. Die Lage der IO-Bitmap ist dann nicht mehr in allen Adressräumen gleich und der IPC-Pfad wird geringfügig komplizierter.

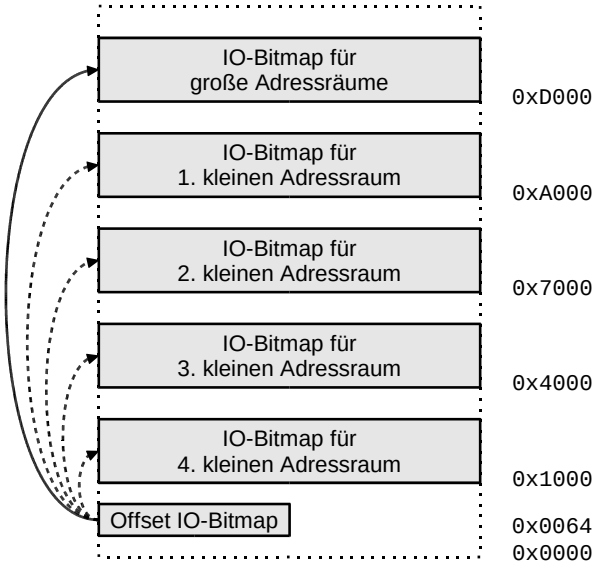


Abbildung 4.3: *Aufbau des TSS und mögliche Lage der IO-Bitmaps für kleine Adressräume.* Die IO-Bitmap muss auf Seitengrenze ausgerichtet sein. Sie beansprucht 8 KB und zusätzlich ein Byte als Abschluss. Durch generelle Sperrung der ersten acht Ports könnten die Bitmaps dichter gepackt werden, wodurch bis zu sieben IO-Bitmaps in das TSS passen würden. Im Unterschied zu dieser Darstellung implementiert Fiasco nur *einen* Bereich, in den die IO-Bitmap jedes Adressraumes eingeblendet wird. Bei Umschalten des Adressraumes ohne Wechsel des Seitenverzeichnisses muss daher die IO-Bitmap explizit ausgetauscht werden.

[2] Die IO-Bitmap hat eine Größe von 8 KB, das entspricht 65536 IO-Ports bzw. zwei 4 KB Seiten.

Sicherung der IOPL bei `sysenter` Beim Kerneintritt über `sysenter` muss explizit der IOPL des aktuellen Threads ermittelt und gespeichert werden, weil der Thread, in dessen Kontext der Kern verlassen wird, mit einer anderen IOPL ausgestattet sein könnte. Vor dem Verlassen des Kerns über `sysexit` ist die IOPL explizit wieder herzustellen.

Kein `sysexit` Verlassen des Kerns mit `sysexit` ist nur möglich, falls keine kleinen Adressräume aktiv sind. Der Grund liegt in der Implementierung von Fiasco. Die Anweisung `sysexit` setzt implizit flache Segmente. Um den Schutz zwischen kleinen und großen Adressräumen weiterhin zu gewährleisten, müssen nach dem Verlassen des Kerns über `sysexit` die aktuell gültigen Segmente gesetzt werden.

Dies kann nur im Nutzermodus durch eine in den *aktuell* gültigen Nutzeradressraum eingeblendete schreibgeschützte Trampoline-Seite geschehen. Diese Seite darf nicht mit den eingeblendeten Seiten der Anwendung in Berührung kommen – sie muss deshalb verschiebbar sein. Im Fiasco-Kern existiert bislang kein Mechanismus für eine derartige verschiebbare Seite.

Die Seite muss innerhalb der Segmentgrenzen des aktuellen Nutzer-Code-Segments eingeblendet sein, weil anderenfalls folgendes Problem auftreten könnte: Wird während der Ausführung von Code auf der Trampoline-Seite ein Interrupt (z. B. Zeitgeber-Interrupt) getriggert, so versucht die entsprechende Routine nach Behandlung an die ursprüngliche Stelle im Code zurückzuspringen. Liegt die Seite außerhalb des aktuell gültigen Nutzer-Code-Segments, so wird eine *General Protection Exception* ausgelöst. Dieser Ausnahmefall müsste gesondert behandelt werden.

Seitenfehler Wird aufgrund eines Seitenfehlers eine Seite in einen kleinen Adressraum eingeblendet, muss das Master-Seitenverzeichnis aktualisiert werden. Das Master-Seitenverzeichnis stellt das Original dar, nach dessen Muster Seiten kleiner Adressräume in alle großen Adressräume eingeblendet werden [34].

In Anhang C.2 sind für ausgewählte x86-CPUs gemessene IPC-Roundtrip-Zeiten zwischen verschiedenen Adressräumen dargestellt. Dabei wurden insbesondere Messungen für IPCs durchgeführt, die zwischen dem Linux-Server und Linux-Anwendungen auftreten (keine Long-IPC). Einer der beiden Threads liegt in einem kleinen Adressraum und entspricht damit dem Linux-Server. Anhand der Messwerte lassen sich folgende Schlussfolgerungen ziehen:

- Durch Verwendung von kleinen Adressräumen lassen sich die *direkten* IPC-Kosten senken. Je nach Prozessor können damit mehr als 10 % der Zyklen gespart werden. Hinzu kommen die gesparten indirekten Kosten, weil TLB-Einträge nach dem Adressraumwechsel nicht neu gefüllt werden müssen.

- Der große Unterschied zwischen *Fast-IPC* und *Slow-IPC* zeigt den bedeutenden Overhead des C++-Codes gegenüber dem Assembler-Pfad. IPC sollte möglichst immer den kurzen Pfad nehmen. Dies ist durch geeignete IPC-Parameter zu erreichen.

4.3 Software-IOMMU

Die in Abschnitt 3.2.4 vorgestellte Virtualisierung der DMA-Einheit von Netzwerkadaptern und IDE-Controllern wurde für L^{4}Linux prototypisch implementiert.

Der Emulator wird im Adressraum von L^{4}Linux ausgeführt. Die Aktivierung erfolgt entweder durch Exceptions, die bei Zugriffen auf durch den Mediator überwachte IO-Ports des Gerätes ausgelöst werden, oder direkt, wenn der Treiber entsprechend modifiziert werden kann: Der Zugriff auf Geräteregister erfolgt unter Linux durch Verwendung spezieller Makros[3], die z. B. im Treiber durch Aufrufe des Emulators ersetzt werden können.

Der Mediator wird als L4-Task in einem eigenem Adressraum ausgeführt. Diese Task ist auch gleichzeitig Pager für L^{4}Linux, um die Zugriffe auf IO-Ports und eingeblendete Geräteregister beschränken zu können. Der Mediator besitzt ferner Informationen über den Linux-Adressraum und kennt die Zuordnung der Seiten zu den physischen Kacheln. In einem hier nicht implementierten erweiterten Ansatz könnten Pager und Mediator in separaten Adressräumen ausgeführt werden und Informationen, die für die Überprüfung der Adressen benötigt werden, über ein geeignetes Interface austauschen.

Für die effiziente Kommunikation zwischen Emulator und Mediator/Pager wurde ein geeignetes Protokoll entworfen, das sich in das Protokoll zur Behandlung von Seitenfehlern einbetten lässt. Insbesondere erlaubt es dieses Protokoll, die Information über Adresse des Geräteregisters, Zugriffsbreite und zu schreibenden Wert in 64 Bit unterzubringen – die Voraussetzung, um *Fast-IPC* (siehe Abschnitt 2.3.2 und Anhang C.2) nutzen zu können.

Je nach Gerät und verwendetem IO-Modus kann es dazu kommen, dass Zugriffe von Geräteregistern, die *nicht* zur DMA-Einheit des Gerätes gehören, trotzdem vom Mediator behandelt werden müssen:

- Geräteregister, die über Port-IO angesprochen werden, besitzen eine durch die IO-Bitmap (Abschnitt 2.4.3) vorgegebene Granularität von 1, d. h. pro 1-Byte-Register kann entschieden werden, ob der Treiber darauf direkt zugreifen darf, oder ob der Zugriff eine Exception auslöst. Unmittelbar benachbarte, nicht zur DMA-Einheit gehörende Register können eigene Zugriffsrechte erhalten, und ein Zugriff auf diese führt nicht zum Aufruf des Mediators. Die IO-Bitmap unterscheidet jedoch nicht nach Lese- und Schreibzugriffen.

- Zugriffsrechte für eingeblendete (memory-mapped) Geräteregister können nur mit einer erheblich größeren Granularität vergeben werden. Auf allen von Linux unterstützten Architekturen beträgt die minimale Seitengröße 4 KB. Alle Geräteregister, die in der selben Seite liegen wie Register der DMA-Einheit, lösen somit Aufrufe des Mediators aus und erzeugen zusätzliche CPU-Last.

4.3.1 Digital DS2114x Tulip Fast-Ethernet-Adapter

Aus Performance-Gründen werden die Deskriptorlisten für das Senden und Empfangen von Paketen lesend in den Adressraum von L^{4}Linux eingeblendet. Der aktuelle Status jedes Deskriptors wird unmittelbar vor Aufruf des Interrupt-Handlers im Treiber von der eingeblendeten Liste des

[3] `writeb()`, `writew()`, `writel()` und `readb()`, `readw()`, `readl()` für memory-mapped IO sowie `inb()`, `inw()`, `inl()` und `outb()`, `outw()`, `outl()` für Port-IO

Mediators in die Liste von L^{4}Linux kopiert, damit der Treiber den aktuellen Zustand des Adapters kennt. Unmittelbar nach Rückkehr aus dem L^{4}Linux-Interrupt-Handler wird der Mediator aufgerufen, der neue Pakete, die der Netzwerktreiber in die Deskriptorlisten eingefügt hat, nach Prüfung in die Deskriptorliste des Gerätes übernimmt.

Weiterhin wird ein Bitfeld zwischen L^{4}Linux und dem Mediator gemeinsam genutzt, dessen Einträge kennzeichnen, ob ein Paket noch an den Netzwerkadapter übergeben werden muss oder ob das Paket bereits wieder Eigentum des Treibers ist. In beiden Fällen ist das Eigentümer-Bit des Paketes gelöscht, so dass die zusätzliche Unterscheidung notwendig ist, um doppeltes Überschreiben von Statusinformationen zu vermeiden.

Im ungünstigsten Fall sind die in Tabelle 4.2 dargestellten zusätzlichen Kosten pro Netzpaket zu erwarten. Bei einem Pentium-III-System mit 800 MHz entspricht dies einer zusätzlichen CPU-Last von etwa 8,4 % sowie etwa 5,0 % bei einem Celeron-4-System mit 2 GHz (Fast Ethernet, MTU-Größe von 1500 Byte).

Vorgang	CPU-Zyklen	
	P3	P4
Auslösung Exception bei Zugriff auf Geräteregister	800	1200
Scannen des Codes durch den Emulator	200	200
IPC an Mediator und Antwort sowie damit verbundene Adressraumwechsel	800	2400
Kopieren der Deskriptorliste im Mediator	200	200
Aktualisierung der Deskriptorliste des Treibers bei Eintreffen des Interrupts	300	300
Summe	2300	4300
Zusätzliche Kosten durch Omega0	6000	8000

Tabelle 4.2: *Erwartete Zusatzkosten pro Netzpaket bei Virtualisierung der DMA bei Netzwerkadaptern.* Der Gesamt-Overhead ist abhängig von der Anzahl der Pakete pro Interrupt.

Im praktischen Einsatz fallen die Kosten je Paket geringer aus, da meist mehrere Pakete mit einem Interrupt behandelt werden.

Da der in Linux 2.2 integrierte Treiber auf die Geräteregister des Netzwerkadapters über Port-IO zugreift, erfolgen nur wenige Aufrufe des Mediators für Register, die nicht zur DMA-Einheit gehören. Auf die Register dieses Gerätes kann aber prinzipiell auch per memory-mapped IO zugegriffen werden. Da in diesem Modus alle Register auf der gleichen 4 KB-Seite eingeblendet sind, erfolgt der Aufruf des Mediators für *jeden* Schreibzugriff auf ein Geräteregister (Lesezugriffe sind nicht gefährlich, siehe Abschnitt 3.2.4).

4.3.2 Intel PRO/1000 Gigabit-Ethernet-Adapter

Ein Gerätetreiber für Intel PRO/1000 Netzwerkadapter wurde erst in Linux 2.4 integriert. Für Linux 2.2 bietet Intel ein ladbares Kernmodul, das in dieser Arbeit mit der Version 4.3.15 verwendet wurde [42].

Der PRO/1000 Gigabit-Ethernet-Adapter wird ähnlich virtualisiert wie der Tulip-Adapter. Der Zugriff auf die Geräteregister erfolgt generell per memory-mapped IO. Die Register sind vom Hersteller aber so gruppiert, dass die für das Senden und Empfangen von Pakten wichtigen Register auf

jeweils eigenen 4 KB-Seiten eingeblendet werden, so dass der Mediator nicht für andere Register aufgerufen wird.

In der Standard-Einstellung arbeitet Linux mit der Ethernet-Paket-Größe von 1500 Byte. Eine Vergrößerung der Pakete auf bis zu 16128 Byte ist möglich (so genannte *Jumbo-Frames*), aber in heutiger Netzwerkumgebung nicht üblich, weil insbesondere Support von den Netzwerk-Switches vorhanden sein muss.

Um bei gleicher Paket-Größe und einer verzehnfachten Datenrate die CPU-Last zu begrenzen, verfügen Gigabit-Ethernet-Adapter über die Möglichkeit, mehrere Interrupts zusammenzufassen (*Interrupt Coalescing*). Dabei wartet der Controller nach Eintreffen des ersten Paketes für eine festgelegte Zeit, bis er einen Interrupt auslöst. Für die praktischen Messungen wurde diese Zeit auf 1024 µs für eingehende und ausgehende Pakete eingestellt. Die in Tabelle 4.2 dargestellten Kosten verteilen sich demnach auf mehrere Netzpakete.

Die Implementierung der DMA-Virtualisierung für den Intel Gigabit Ethernet-Adapter PRO/1000 habe ich mangels Dokumentation ausschließlich durch Analyse des Linux-Treibers vorgenommen.

4.3.3 IDE-Controller

Die Virtualisierung des IDE-Controllers ist verhältnismäßig einfach. Es gibt nur eine Richtung des Datenaustausches: Der Treiber erstellt eine Liste mit Blöcken, die gelesen bzw. geschrieben werden sollen und schreibt die Adresse dieser Liste in ein Geräteregister. Der Mediator verifiziert diese Liste und kann damit fehlerhafte DMA-Operationen verhindern. Das Gerät liest die Liste und führt die Aktion aus. Der Status über das Ergebnis dieser Aktion wird durch andere Geräteregister übergeben.

Aufgrund des günstigen Verhältnisses zwischen der Größe des Puffer-Deskriptors und dem Puffer ist bei der Virtualisierung von DMA bei IDE-Controllern ein geringerer Overhead zu erwarten.

Pro Blockliste fallen voraussichtlich die in Tabelle 4.3 angegebenen zusätzlichen Kosten an.

Vorgang	CPU-Zyklen	
	P3	P4
Auslösung Exception bei Zugriff auf Geräteregister	800	1200
Scannen des Codes durch den Emulator	200	200
IPC an Mediator und Antwort sowie damit verbundene Adressraumwechsel	800	2400
Kopieren der Blockliste im Mediator	200	200
Summe	2000	4000

Tabelle 4.3: *Erwartete Zusatzkosten pro Blockliste bei Virtualisierung des DMA-Modus bei IDE-Controllern.*

4.3.4 Wiederverwendung von Mediator und Emulator

In Linux 2.2.26 gibt es drei verschiedene Treiber-Implementierungen für Tulip-Netzwerkadapter, wovon zwei mit dem Adapter im Testrechner arbeiten konnten. Die Kombination aus Emulator und Mediator wurde für einen Treiber (`CONFIG_DE4X5`) entwickelt und konnte ohne Änderungen für den anderen Treiber (`CONFIG_DEC_ELCP`) übernommen werden.

Dieses Beispiel illustriert den verminderten Aufwand bei der Treiber-Entwicklung: Eine Implementierung der DMA-Virtualisierung arbeitet für eine *Klasse* von Treibern. Mediator und Emulator müssen bei Treiber-Updates nicht verändert werden, sofern sich die Hardware-Eigenschaften nicht ändern.

4.3.5 Codegrößen

In Tabelle 4.4 ist der Codeumfang für die Virtualisierung der DMA-Einheiten dargestellt. Für den IDE-Controller ist kein spezieller Code im Emulator erforderlich. Die Zahlen belegen, dass die Erweiterung vorhandener Gerätetreiber wesentlich weniger Aufwand bedeutet als die vollständige Implementierung eines Gerätetreibers.

	Digital DS2114x Tulip Fast Ethernet	Intel PRO/1000 Gigabit Ethernet	IDE-Controller
Treiber (Linux 2.2.26)	4800	8000	11000
Emulator (universell)	400	400	400
Emulator (pro Gerät)	180	200	0
Mediator (universell)	600	600	600
Mediator (pro Gerät)	350	350	130

Tabelle 4.4: *Codegrößen in SLOC, gemessen mit sloccount [95].*

Kapitel 5

Leistungsbewertung

Das Ziel der vorgestellten Performance-Messungen ist es, eine Übersicht über die Kosten zu geben, die bei einer starken Kapselung von Linux auf heutiger Standard-Hardware auftreten. Viele Messungen wurden bereits 1997 mit L^{4}Linux basierend auf Linux 2.0 durchgeführt. Diese basierten aber auf der damaligen Hardware und dem damals in Assembler implementierten L4-Mikrokern von Liedtke [56]. Fiasco [35], Grundlage der folgenden Messungen, ist primär auf gute Vorhersagbarkeit optimiert.

Die Kommunikation von Festplatten-Controller und Netzwerkadapter mittels DMA wird mit Hilfe der in Abschnitt 3.2.4 dargestellten Technik kontrolliert. Soweit nicht anders angegeben, wird der Emulator nicht über den Exception-Mechanismus angesprungen, sondern durch Veränderung eines Makros in den Linux-Quellen direkt aufgerufen. Damit wird pro Zugriff ein Kernein- und -austritt vermieden. Die Umgehung des Exception-Mechanismus stellt kein Sicherheitsproblem dar, weil Linux keinen direkten Zugriff auf die gefährlichen Geräteregister hat. Sollte der Treiber unter Umgehung des veränderten Makros auf die Geräteregister zugreifen, würde eine Exception ausgelöst, die der Emulator behandelt.

Für die Messungen unter Linux und L^{4}Linux wurde eine identische Umgebung geschaffen. Der L^{4}Linux-Server wird mit dem in Abschnitt 2.3.4 dargestellten Exception-Mechanismus (also ohne veränderte C-Bibliothek) betreten.

L^{4}Linux kann nicht den gesamten physischen Hauptspeicher nutzen, da der Mikrokern und andere Server im Nutzermodus ebenfalls Speicher benötigen. In der aktuellen Implementierung werden etwa 4 MB Speicher für Kern und die Ressourcen-Manager sowie 32 MB Speicher für den Kern-Heap (insbesondere TCBs und Mapping-Datenbank) reserviert.[1]

Anwendungsszenario Für die folgenden Messungen wurde das in Abbildung 5.1 dargestellte Anwendungsszenario verwendet. Je nach Messung wurde L^{4}Linux im kleinen oder im großen Adressraum ausgeführt. Die verwendete Hardware ist im Anhang A beschriebenen.

[1] Der Kern-Heap wird von Fiasco statisch allokiert und hängt unter anderem von der Anzahl Tasks und Threads im System sowie von der Anzahl und der Tiefe der Mapping-Beziehungen zwischen den Adressräumen ab. Bei den vorliegenden Messungen wurde ein Bedarf von etwa 10 MB nicht überschritten.

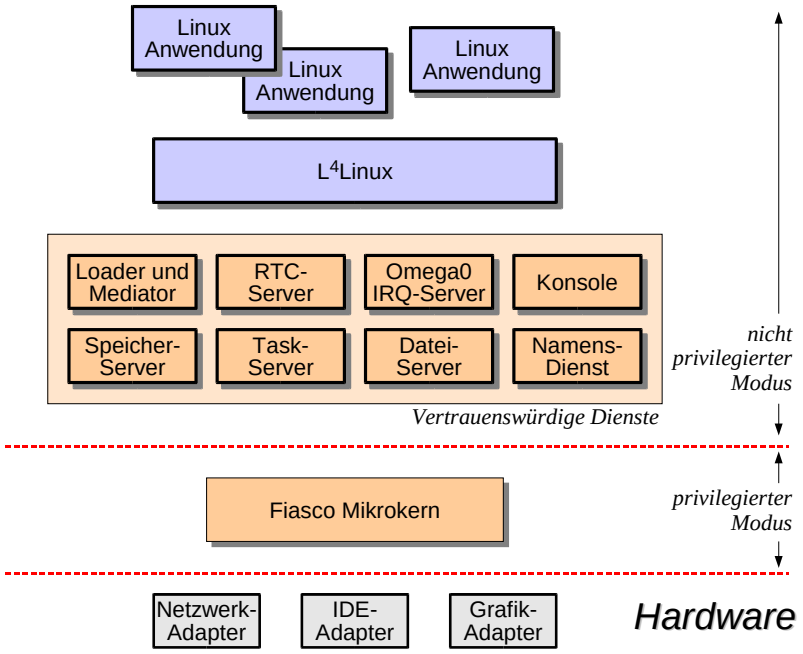


Abbildung 5.1: *Anwendungsszenario für die Messungen dieses Kapitels.* Nicht vertrauenswürdige Komponenten sind blau eingefärbt.

L4Linux-Konfigurationen In einigen der folgenden Messungen wurde der Einfluss verschiedener Konfigurationen von L4Linux auf die CPU-Last und die Performance untersucht:

4K/4M Der Adressraum von L4Linux für die x86-Architektur kann entweder aus n 4 MB-Seiten aufgebaut werden (Voreinstellung) oder aus $1024 \times n$ 4 KB-Seiten. Viele kleine Seiten bedeuten mehr TLB-Einträge bei gleicher Größe des virtuellen Speichers und damit eine größere TLB-Arbeitsmenge. Damit steigen potenziell die indirekten Kosten nach einem TLB-Flush (z. B. nach Wechsel des Seitenverzeichnisses). Die Wahl der Seitengröße von L4Linux ist durch dessen Pager steuerbar.

Large/Small *small* bedeutet Ausführung von L4Linux im kleinen Adressraum. Der Linux-Kern ist dabei in alle anderen großen Adressräume eingeblendet. Der Schutz zwischen kleinen und großen Adressräumen wird durch Segmentierung erreicht. Soweit nicht anders angegeben, wird L4Linux bei diesen Messungen im großen Adressraum ausgeführt.

Mediator Bei Verwendung des Mediators werden DMA-Zugriffe des entsprechenden Gerätes an den Mediator weitergeleitet, der diese verifiziert und dann selbst ausführt. Dadurch werden fehlerhafte DMA-Zugriffe durch fehlerhafte oder bösartige Treiber vermieden. Dies bedingt aber auch einen größeren Kommunikationsaufwand und damit zusätzliche CPU-Last.

Linux wurde für die Messungen generell im *Single User Mode* gestartet, um Beeinflussungen durch andere Prozesse zu minimieren.

5.1 DMA-Virtualisierung bei IDE-Controllern

5.1.1 Anwendungsbenchmark

Für die Messung der zusätzlichen Kosten, die aus der Ausführung von Linux als Server im Nutzermodus sowie aus der Virtualisierung der DMA-Einheit des IDE-Controllers resultieren, wurde ein geeigneter Anwendungsbenchmark entwickelt. Der Benchmark besteht aus CPU- und speicherintensiven Operationen (Compilieren des Linux-Kerns) sowie aus überwiegend Festplatten-intensiven Operationen (Kopieren der Linux-Quellen, Suchen in den Quelldateien). Der Benchmark besteht aus folgenden Teilen:

1. *extract:* Entpacken des Linux-Kerns
   (`tar -xjf linux-2.6.7.tar.bz2`)
2. *config:* Konfiguration des Kerns mit der Standard-Konfiguration
   (`make oldconfig`)
3. *build:* Erzeugen des Kerns
   (`make bzImage`)
4. *copy+grep:* Duplizieren des Verzeichnisses und rekursives Suchen
   (`cp -r linux-2.6.7 linux-2; grep -r` <*random search pattern*> `.`)
5. *cleanup:* Aufräumen
   (`make clean; rm -rf .`)

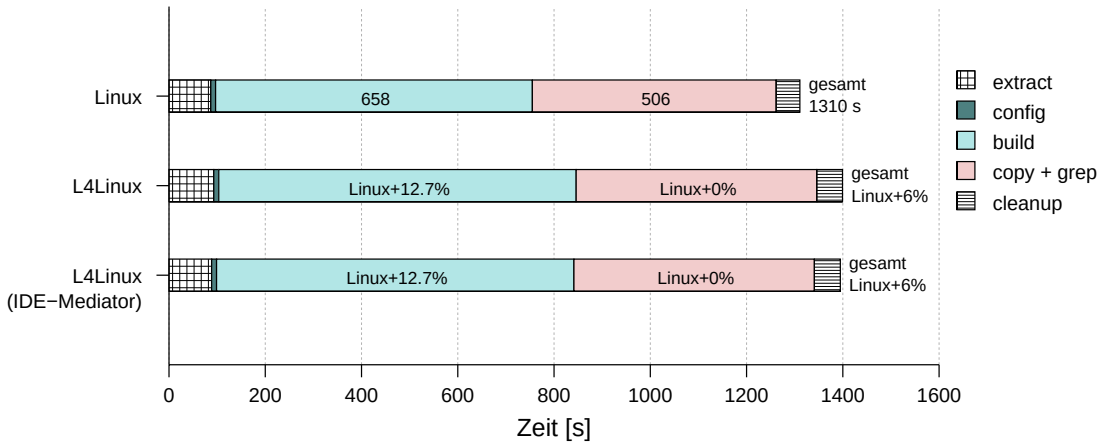
Die Schritte *extract*, *config* und *build* beanspruchen viel CPU-Zeit und vergleichsweise wenig IO-Bandbreite. Die Zeit für *build* sollte vor allem von der Leistung der CPU, vom verfügbaren Hauptspeicher und vom Speicherdurchsatz abhängen.

In diesen Messungen sollte jedoch vor allem der Einfluss der DMA-Virtualisierung gezeigt werden. Deshalb wurde sichergestellt, dass Linux und L^{4}Linux jeweils gleich viel Speicher (ca. 208 MB auf dem Pentium-III-System bzw. 438 MB auf dem Celeron-4-System) zur Verfügung hatten.

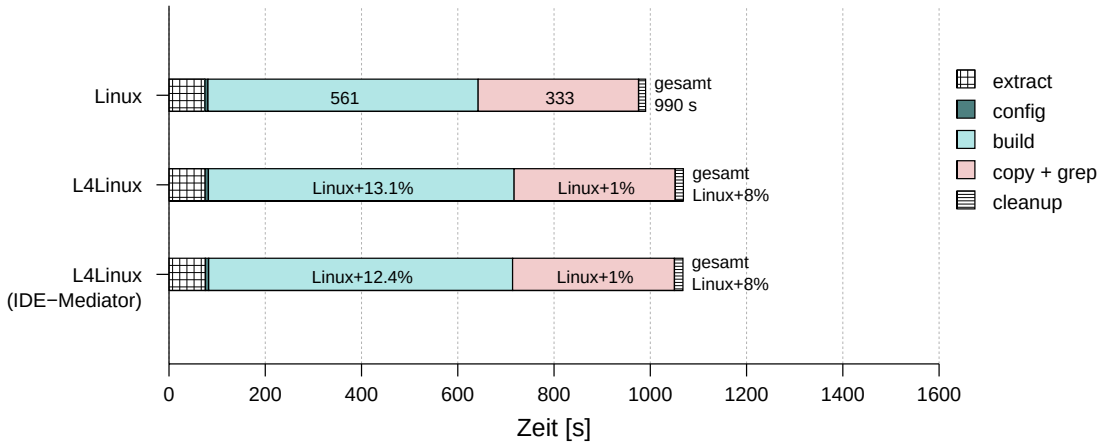
In Abbildung 5.2 sind die während der einzelnen Teilschritte gemessenen Laufzeiten und CPU-Lasten jeweils für beide Testrechner dargestellt.

Interpretation der Ergebnisse Auch bei Operationen mit hohem Anteil an Plattenbandbreite (Schritte *copy+grep* und *cleanup*) sind bei beiden Systemen keine merklichen Mehrkosten für Gesamtzeit und CPU-Last bei der Verifizierung der Adressen der DMA-Einheit von IDE-Blockgeräten erkennbar (Vergleich von L^{4}Linux ohne und mit IDE-Mediator). Damit bestätigen sich die in Abschnitt 3.2.4 getroffenen Aussagen.

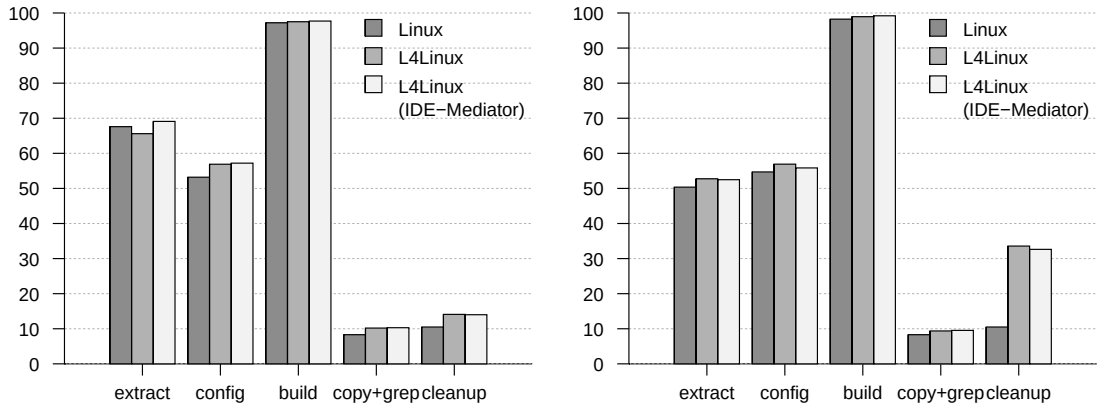
Auf den ersten Blick scheint es überraschend, dass L^{4}Linux gegenüber Standard-Linux etwa 13 % mehr Zeit zum Compilieren der Linux-Quellen benötigt. Bei diesem Test beträgt die CPU-Last jeweils mehr als 97 %. Deshalb haben alle zusätzlichen Kosten für Kerneintritte und Adressraumwechsel (Cache- und TLB-Misses) direkten Einfluss auf die Gesamtzeit des Benchmarks. Eine genauere Analyse der Mehrkosten beim Schritt *compile* ist in Abschnitt 5.3 zu finden.



(a) Pentium-III-System (800 MHz)



(b) Celeron-4-System (2 GHz)



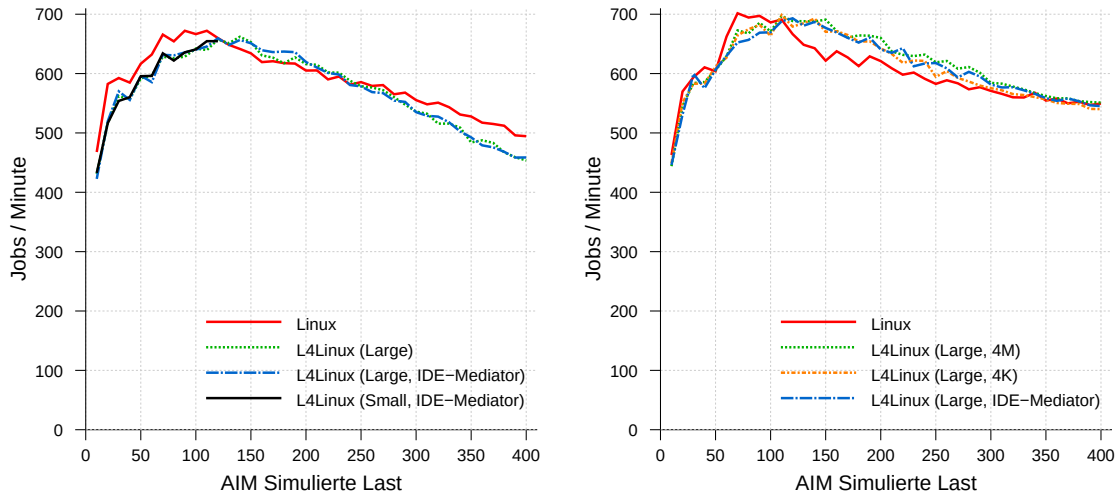
(c) CPU-Last: Pentium-III-System (800 MHz)

(d) CPU-Last: Celeron-4-System (2 GHz)

Abbildung 5.2: *Benchmark zur Ermittlung der Kosten des IDE-Mediators unter realen Anwendungen.* Compilieren, Kopieren und Suchen auf den Linux-Kern-Quellen.

5.1.2 Synthetischer Anwendungsbenchmark

Abbildung 5.3 zeigt den Einfluss der DMA-Virtualisierung bei IDE-Controllern für den AIM-Multiuser-Benchmark [10]. Dieser Benchmark wurde bereits für Messungen am ursprünglichen L^{4}Linux-System verwendet [25]. Als Workload für den Benchmark wurde der *Shared System Mix* gewählt, der aus vielen sehr kleinen Jobs besteht und eine ausgewogene Beanspruchung mit Berechnungen, Speicher- und IO-Last garantiert.



(a) Pentium-III-System (800 MHz) (b) Celeron-4-System (2 GHz)

Abbildung 5.3: *AIM-Multiuser-Benchmark Suite 7 [10].* Als Workload wurde der *Shared System Mix* mit einer großen Anzahl kleiner Tests, inklusive Festplattenlast, ausgewählt.

Die zusätzlichen Kosten der IDE-DMA-Virtualisierung sind auch hier vernachlässigbar. Zu beachten ist die niedrigere Performance von Standard-Linux auf dem Celeron-4-System. Dies liegt mit hoher Wahrscheinlichkeit an der Festplatte, deren Parameter den Benchmark entscheidend beeinflussen. Dies ist auch daran zu erkennen, dass auf beiden Systemen sehr ähnliche Resultate erreicht werden, obwohl das Celeron-4-System merklich bessere CPU- und Speicher-Performance aufweist (siehe Anhang A).

L^{4}Linux wurde bei diesen Messungen im großen Adressraum ausgeführt. Eine Messung auf dem Pentium-III-System wurde mit L^{4}Linux im kleinen Adressraum ausgeführt, dabei wurde der Speicher auf 128 MB begrenzt (siehe Abschnitt 2.4.2). Aus diesem Grund brach der Benchmark bei einer Last von 130 wegen Speichermangel ab. Bis zu dieser Last ist kein wesentlicher Unterschied zu den anderen Messungen mit L^{4}Linux zu erkennen.

Auf dem Celeron-4-System wurde auch untersucht, welchen Einfluss das Verwenden von großen TLB-Einträgen für Linux auf die Performance hat (Kurven *L4Linux (Large, 4M)* und *L4Linux (Large, 4K)*). Bei höherer Last ist ein geringer Unterschied zu erkennen, der sich aus der größeren Anzahl an TLB-Misses erklärt.

5.2 DMA-Virtualisierung bei Netzwerkadaptern

5.2.1 Fast-Ethernet

In Abschnitt 3.2.4 wurde gezeigt, wie die DMA von Netzwerkadaptern durch Kopieren der Deskriptorlisten virtualisiert werden kann, wenn bestimmte Voraussetzungen der Hardware erfüllt sind. Aufgrund der relativ hohen Interrupt-Last bei Netzwerkadaptern – im schlimmsten Fall generiert der Netzwerkadapter pro Ethernet-Paket einen Interrupt – ist eine beträchtliche Zusatzbelastung der CPU zu erwarten.

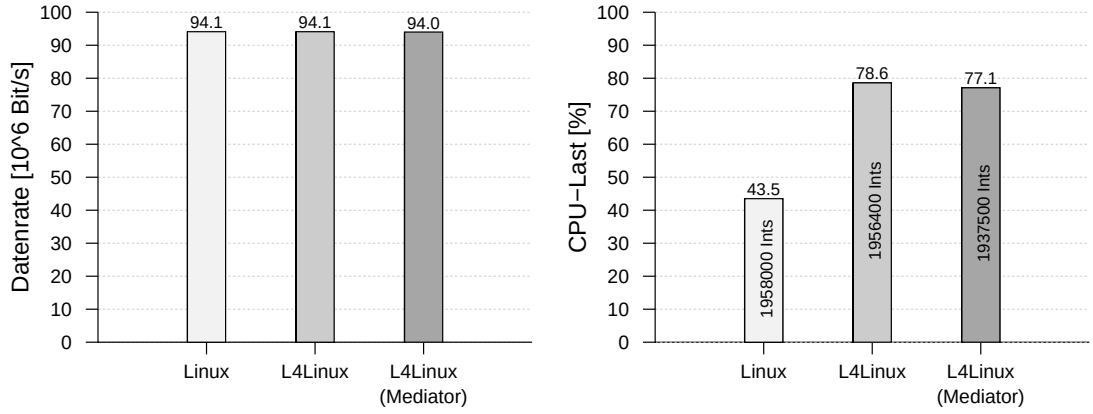
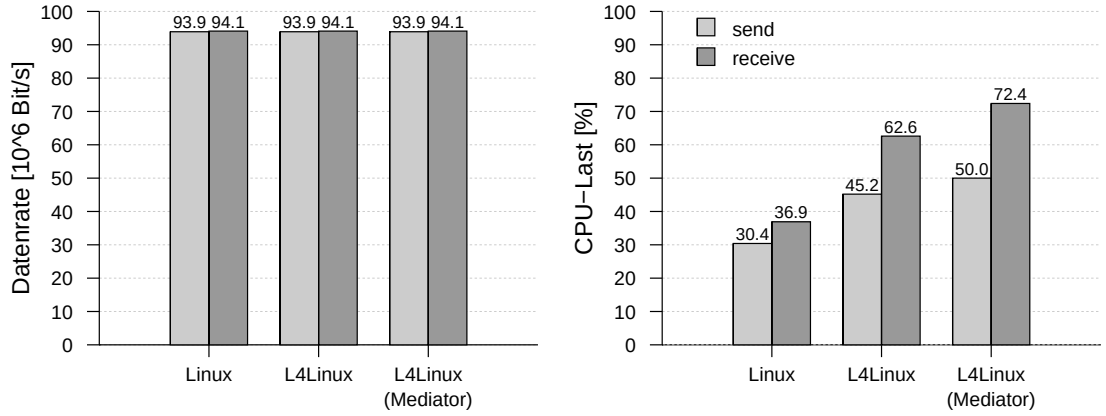


Abbildung 5.4: *`wget`-Test auf dem Pentium-III-System (800 MHz) mit Fast-Ethernet-Adapter (Digital DS21143 Tulip rev 65).* Lesen von 1800 MB per HTTP-Protokoll von einem Webserver. In der rechten Abbildung sind die CPU-Last sowie die Anzahl der vom Netzwerkadapter während der Messung ausgelösten Interrupts gezeigt.

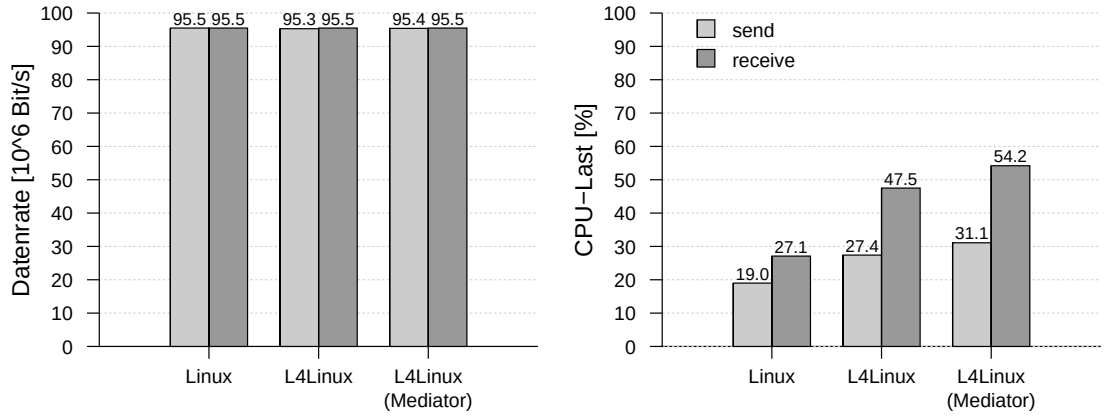
Abbildung 5.4 zeigt den Einfluss der DMA–Virtualisierung des Fast-Ethernet-Adapters auf Netto-Datenraten der Anwendungen und die CPU-Last. Hierbei liest das Linux-Programm `wget` eine große Datei von einem Server per HTTP-Protokoll und schreibt deren Inhalt nach `/dev/null`. Aus Sicht von L^{4}Linux empfängt `wget` Daten per `recv`-Systemaufruf von einem Socket und schreibt diese über den `write`-Systemaufruf auf ein virtuelles Gerät. `wget` verwendet als Blockgröße 16 KB, so dass bei Fast Ethernet mindestens 760 Blöcke pro Sekunde vom Socket gelesen werden und in das virtuelle Gerät geschrieben werden.

Zu beachten ist die etwas geringere CPU-Auslastung bei Verwendung des Mediators für den Tulip-Netzwerkadapter bei gleicher Datenrate. Dieser Effekt ist dadurch zu erklären, dass die Interrupt-Behandlung mit Mediator etwas länger dauert und dadurch bereits weitere Pakete eintreffen, die mit dem nächsten Interrupt behandelt werden können. Im Bild wird jeweils auch die Anzahl der Interrupts dargestellt.

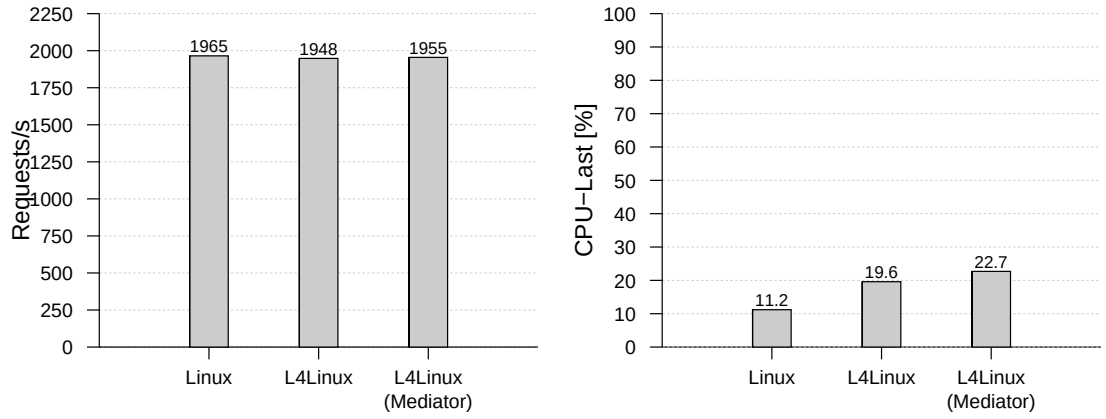
Abbildung 5.5 zeigt Ergebnisse des Benchmarks *netperf* [32] für Fast-Ethernet. Auch hier lässt sich erkennen, dass die DMA-Virtualisierung zusätzliche Kosten verursacht, die allerdings nicht groß genug sind, um Datenraten zu verringern.



(a) `TCP_STREAM`.



(b) `UDP_STREAM`.



(c) `UDP_RR`.

Abbildung 5.5: *netperf-Benchmark auf dem Pentium-III-System mit Fast-Ethernet-Adapter.* Links ist jeweils die erzielte Datenrate bzw. die Anzahl an Aufträgen dargestellt, rechts die CPU-Last.

5.2.2 Gigabit-Ethernet

Abbildung 5.6 zeigt die Ergebnisse des `wget`-Benchmarks auf dem Celeron-4-System mit Gigabit-Ethernet-Adapter (PRO/1000). Die verzehnfachte Bruttodatenrate und die damit verbundene höhere Interrupt-Last bedeutet eine wesentlich größere Beanspruchung der CPU als bei Fast-Ethernet. Selbst auf dem schnelleren Celeron-4-System kann auf L^{4}Linux ohne Mediator die Datenrate von Linux nicht erreicht werden. Bei dieser Messung wurde ebenfalls der Einfluss von kleinen Adressräumen und die Nutzung von großen TLB-Einträgen untersucht.

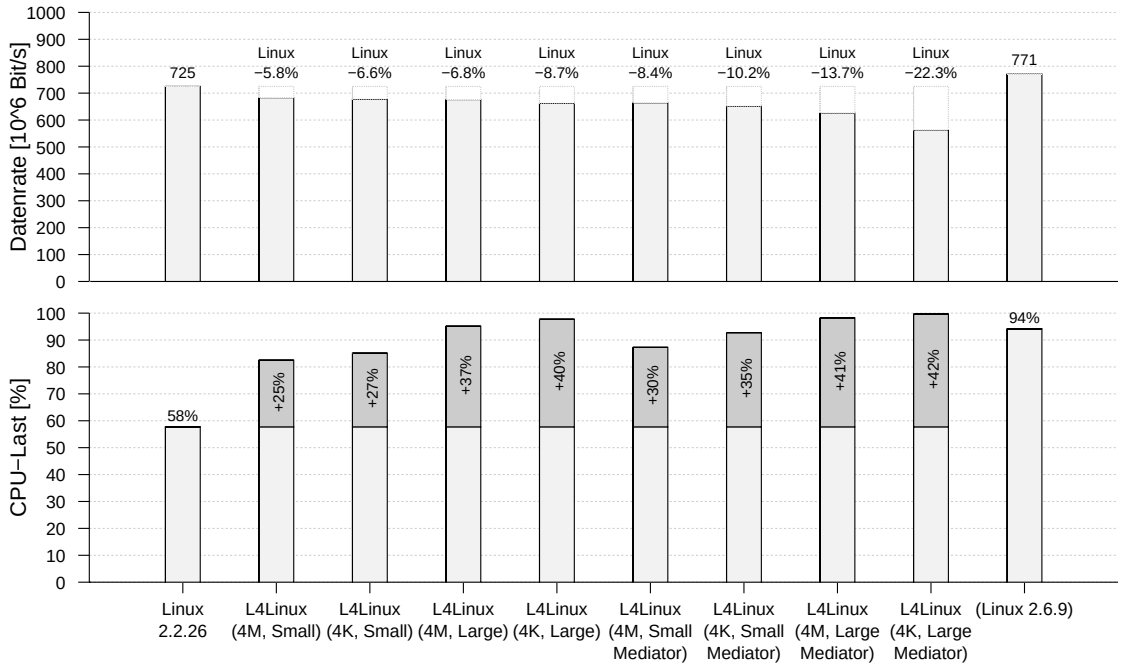
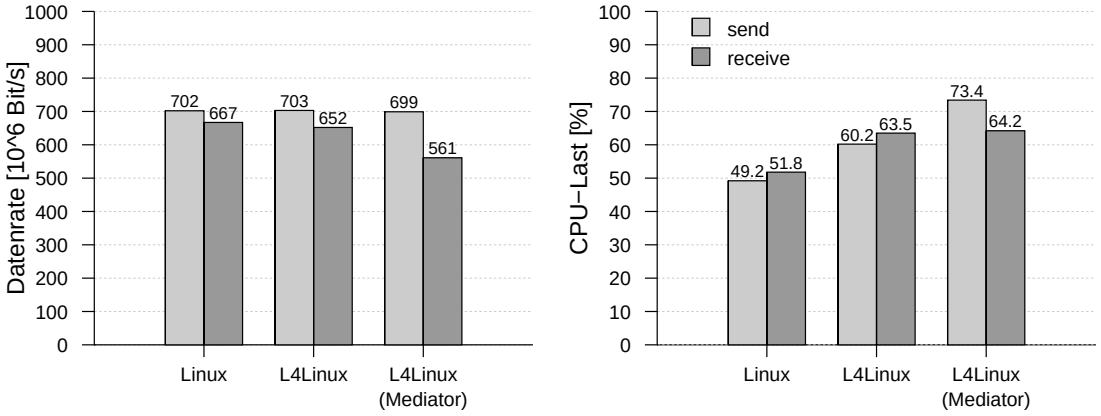


Abbildung 5.6: *`wget`-Benchmark auf dem Celeron-4-System mit Gigabit-Ethernet-Adapter (Intel PRO/1000).* Interrupt-Verzögerung ca. 1024 µs. Lesen von 2000 MB per HTTP-Protokoll von einem Webserver, die MTU-Größe beträgt 1500 Byte. Für Bedeutung der Abkürzungen siehe Beginn dieses Kapitels.

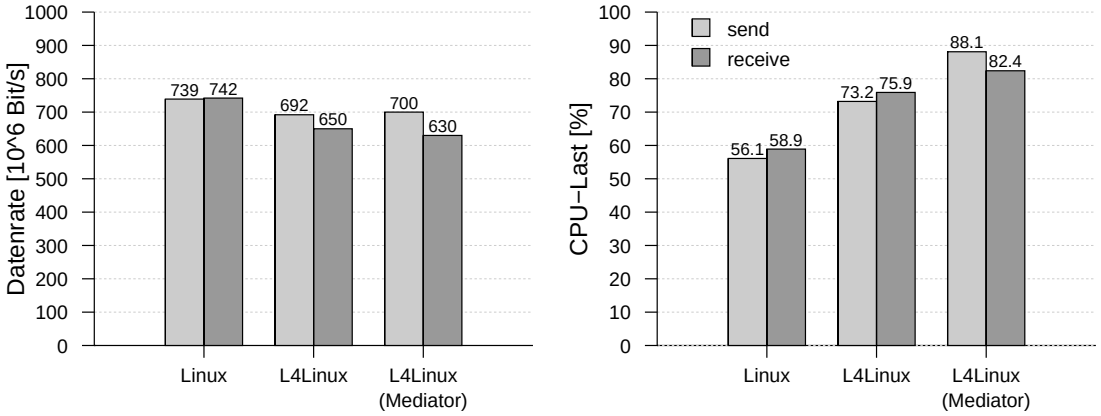
Die Ausführung von Linux als L4-Server im kleinen Adressraum kostet auf diesem System etwa 25 % CPU-Leistung. Weitere 5 % der CPU-Leistung sind erforderlich, um die DMA-Zugriffe auf den Netzwerkadapter zu virtualisieren. Sehr gut ist der Einfluss kleiner Adressräume zu erkennen: Mit Linux im kleinen Adressraum (*4M, Small*) ergibt sich eine Verminderung der CPU-Last um 12 % gegenüber Linux im großen Adressraum (*4M, Large*).

In Abbildung 5.7 sind die Ergebnisse des netperf-Benchmarks dargestellt. Bei `TCP_STREAM` wurde der gleiche Benchmark mit unterschiedlichen Verzögerungszeiten für Interrupts ausgeführt. Eine Erhöhung dieses Wertes führt dazu, dass pro Netzpaket durchschnittlich weniger Interrupts ausgeführt werden und damit die CPU-Last sinkt. Gleichzeitig erhöht sich aber die Latenz. Die Abhängigkeit der CPU-Last von der Verzögerungszeit für Interrupts ist gut zu erkennen. Standard-Linux erreicht mit einer geringeren Verzögerung eine noch größere Datenrate.

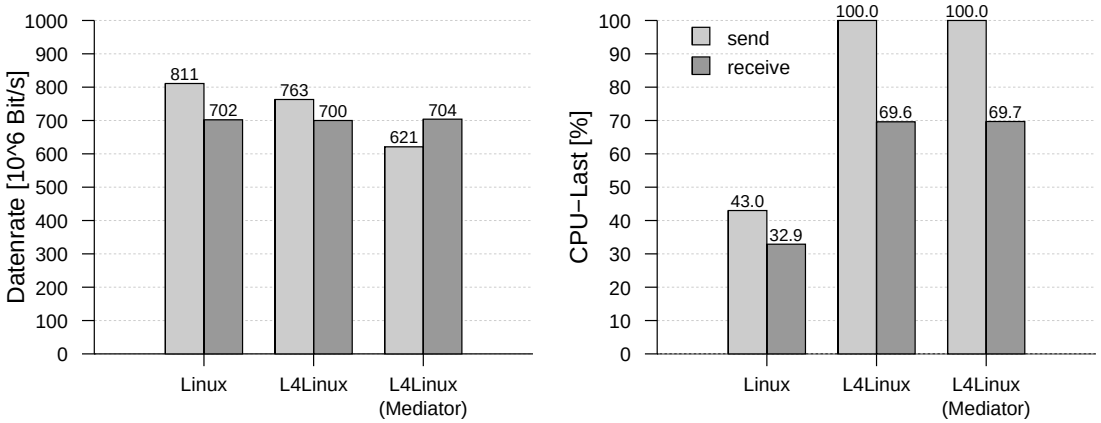
Bei `UDP_STREAM` wird beim Senden unter L^{4}Linux eine CPU-Last von 100 % erreicht.



(a) `TCP_STREAM`. Interrupt-Verzögerung ca. 1024 µs.



(b) `TCP_STREAM`. Interrupt-Verzögerung ca. 260 µs.



(c) `UDP_STREAM`. Interrupt-Verzögerung ca. 1024 µs.

Abbildung 5.7: *netperf-Benchmark auf dem Celeron-4-System mit Gigabit-Ethernet-Adapter.* Auf den linken Diagrammen sind die erzielten Datenraten dargestellt, auf den rechten die CPU-Last.

5.3 Kosten der Ausführung von Linux auf Fiasco

In Abschnitt 5.1.1 wurde festgestellt, dass L^{4}Linux auf Fiasco beim Compilieren von Code mindestens 12 % langsamer als Linux ist. Um mögliche Ursachen für die zusätzlichen Kosten zu ermitteln, wurden erweiterte Messungen mit verschiedenen Konfigurationen von L^{4}Linux und Fiasco durchgeführt.

Durch Monitoring von Kern-Ereignissen wurden folgende Werte ermittelt: Auf dem Pentium-III-System finden während des Compilierens etwa 19 Millionen Kontextwechsel[2] statt, das entspricht etwa 25300 Kontextwechsel pro Sekunde. Darin enthalten sind 18,7 Millionen Adressraumwechsel (entspricht 24900 Adressraumwechseln pro Sekunde). Ferner wurden ca. 16350 Interrupts durch den Festplatten-Controller ausgelöst (entspricht 22 Interrupts pro Sekunde) und 16300 Tasks erzeugt.

Die hohe Anzahl an Adressraumwechseln führt dazu, dass sich selbst kleine Verzögerungen in diesem kritischen Kernpfad bemerkbar machen. 80 % der Adressraumwechsel finden bei diesem Benchmark zwischen dem Linux-Server und Linux-Nutzerprozessen statt. Es liegt nahe, die Kosten für diese Adressraumwechsel durch die Verschiebung des Linux-Servers in einen *kleinen Adressraum* zu verringern.

Der Mikrokern Fiasco bietet kleine Adressräume mit einer Gesamtgröße bis 128 MB an (siehe Abschnitt 2.4.2). In der folgenden Messung (Abbildung 5.8) wurde deshalb der Linux zur Verfügung stehende Speicher auf 128 MB begrenzt. Die Beschränkung ist notwendig, da Linux 2.2 den gesamten zur Verfügung stehenden physischen Speicher einblendet. Der Compilationstest wurde noch einmal für Linux und L^{4}Linux durchgeführt, für L^{4}Linux als großer bzw. kleiner Adressraum. Ferner wurde die Seitengröße von L^{4}Linux variiert, um den Einfluss von großen TLBs zu messen.

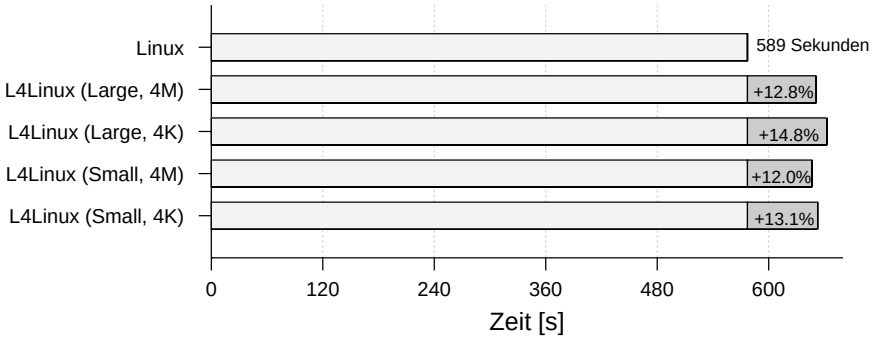


Abbildung 5.8: *Einfluss verschiedener Konfigurationen von L^{4}Linux auf das Compilieren auf dem Celeron-4-System (2 GHz).*

An den Ergebnissen ist zu erkennen, dass sich mit kleinen Adressräumen Verbesserungen erreichen lassen, ebenso hat die Verwendung von 4 MB-Seiten einen positiven Einfluss auf die Kosten. Allerdings erreicht L^{4}Linux auf Fiasco auch durch Verwendung von 4 MB-Seiten im kleinen Adressraum nicht die Zeit von Standard-Linux. Die Einsparungen betragen maximal etwa 3 %.

Als weitere Einflussgröße wurde der Einfluss der IPC-Geschwindigkeit von Fiasco auf den Benchmark untersucht. Für Abbildung 5.9 wurden in den IPC-Pfad von Fiasco schrittweise Verzögerungen bis zu 10 µs eingefügt und die Resultate verglichen. Anhand der Messungen ist zu

[2] Umschaltung des Thread-Kontextes innerhalb eines Adressraumes bzw. zwischen Adressräumen

erkennen, dass die Verlangsamung der IPC um 2000 Takte (entspricht 1 µs) zu einer Zunahme der Compilationszeit um 3,4 % führt.

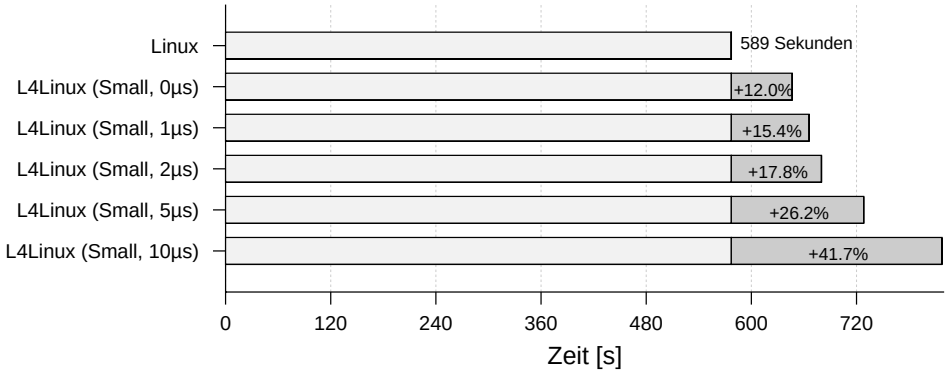
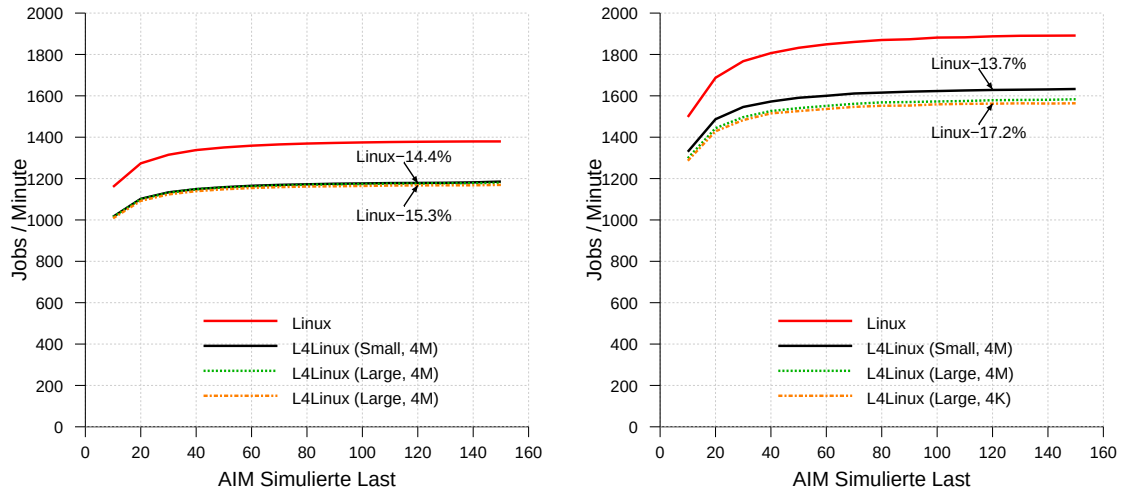


Abbildung 5.9: *Einfluss der IPC-Performance auf das Compilieren auf dem Celeron-4-System (2 GHz).* Im Vergleich zu *L4Linux* wurde der IPC-Pfad in Fiasco schrittweise um bis zu 10 µs (entsprechend 20000 Takte auf diesem System) verlangsamt.

In Abschnitt 5.1.2 wurde der synthetische AIM-Benchmark verwendet, um den Einfluss des IDE-Mediators auf die Performance bei hohen Lasten unterschiedlicher Art zu untersuchen. Im folgenden wurde dieser Benchmark *ohne* Festplattenlast wiederholt, um Einflüsse durch externe Geräte zu minimieren (siehe Abbildung 5.10).



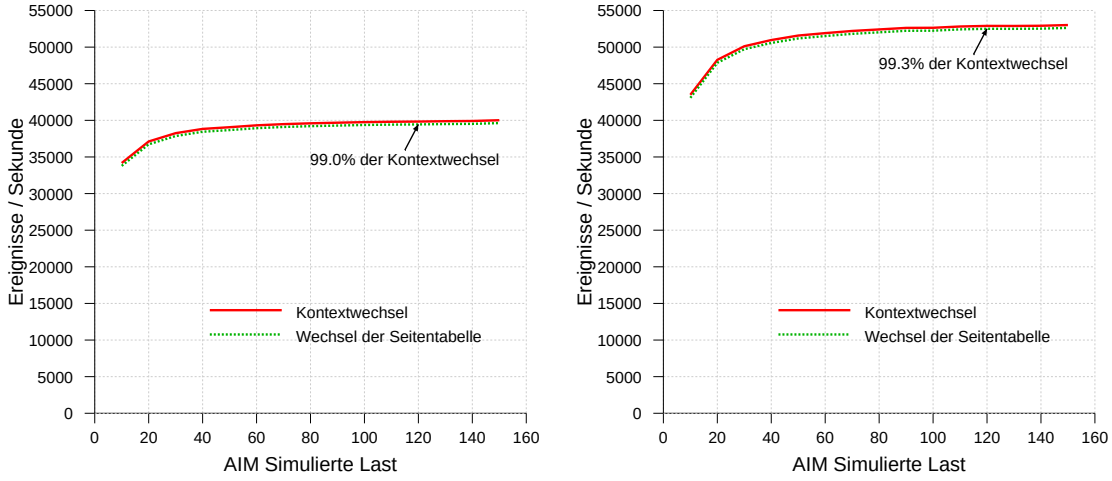
(a) Pentium-III-System (800 MHz)

(b) Celeron-4-System (2 GHz)

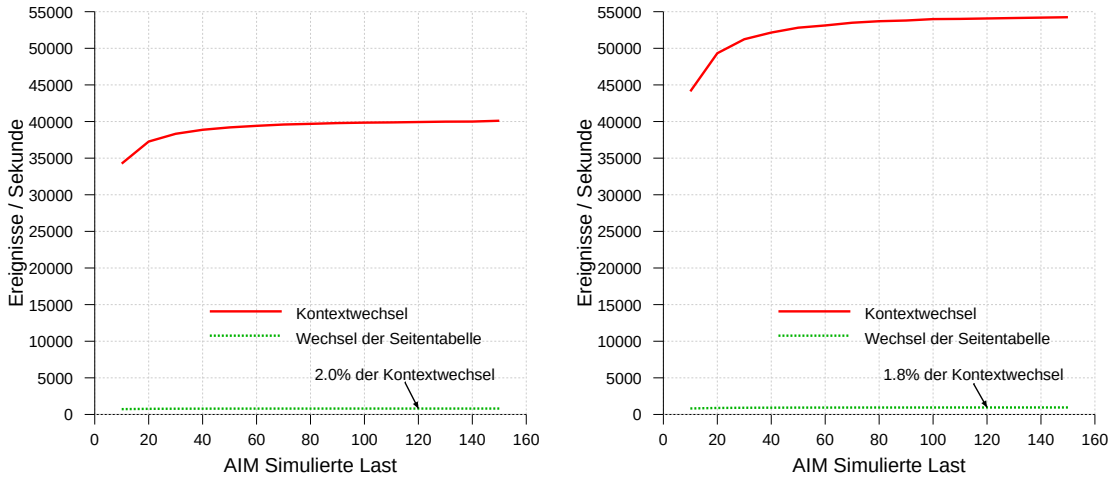
Abbildung 5.10: *AIM-Multiuser-Benchmark Suite 7 [10] ohne Plattenzugriffe.* Als Workload wurde der *Shared System Mix ohne* Festplattenlast verwendet. Der für Linux verfügbare Hauptspeicher wurde bei beiden Systemen auf 120 MB begrenzt.

Die Messungen auf beiden Systemen zeigen einen Abstand von Standard-Linux gegenüber L^{4}Linux von etwa 14 %. Im Gegensatz zum Pentium-III-System wirkt sich auf dem Celeron-4-System die Ausführung von L^{4}Linux im kleinen Adressraum spürbar aus. Für eine genauere In-

terpretation dieser Ergebnisse wurde gemessen, wie viele Kontextwechsel bzw. Wechsel der Seitentabellen während dieses Benchmarks stattfanden (vgl. Abbildung 5.11). Auf beiden Systemen reduzieren sich die Wechsel der Seitentabelle von etwa 99 % auf etwa 2 %. Es finden also sehr viele Kontextwechsel zwischen Linux und den Linux-Anwendungen statt. Warum das Celeron-4-System von kleinen Adressräumen mehr profitiert als das Pentium-III-System, konnte nicht abschließend ermittelt werden.



(a) L4Linux im **großen** Adressraum.



(b) L4Linux im **kleinen** Adressraum.

Abbildung 5.11: *Anzahl der Kern-Ereignisse während der Ausführung der AIM-Multiuser-Benchmark Suite 7 wie in Abbildung 5.10.* Die linken Diagramme zeigen die Ergebnisse für das Pentium-III-System, die rechten Diagramme für das Celeron-4-System.

5.4 Linux mit eingeschränkten IO-Rechten

In diesem Abschnitt sollen die Kosten ermittelt werden, die dadurch entstehen, dass L^{4}Linux mit eingeschränkten IO-Rechten ausgeführt wird.

In Tabelle 5.1 ist dargestellt, wie oft die `cli`-Anweisung innerhalb des in Abschnitt 5.1.1 eingeführten Benchmarks auftritt (Spalten *`cli`/s*) und wie oft eine Lock-Contention auftritt, d. h. wie oft ein Thread eine IPC zum Lock-Thread ausführen musste, weil ein anderer Thread bereits Eigentümer des `cli`-Locks war (Spalten *IPC*).

	ohne Parallellast		IDE-Mediator		Netz 100 MBit		Netz 1000 MBit	
	`cli`/s	IPC	`cli`/s	IPC	`cli`/s	IPC	`cli`/s	IPC
extract	20623	0.0028 %	20826	0.0032 %	208102	0.14 %	209528	0.16 %
config	20029	0.0007 %	20061	0.0022 %	200596	0.12 %	201170	0.12 %
build	25483	0.0023 %	25578	0.0026 %	173750	0.09 %	182764	0.09 %
copy+grep	41469	0.0027 %	40955	0.0021 %	260080	0.19 %	261503	0.23 %
cleanup	52584	0.0014 %	52490	0.0023 %	237178	0.15 %	231215	0.13 %

Tabelle 5.1: *Kosten beim Ersetzen von `cli` und `sti` durch ein Lock auf dem Celeron-4-System.* Das Feld *IPC* zeigt jeweils den relativen Anteil an `cli`-Aufrufen, die eine IPC zum `cli`-Lock zur Folge hatten, weil das Lock belegt war (Lock Contention).

Ermittelt wurden die Kosten für das in Anhang A beschriebene Celeron-4-System auf L^{4}Linux. Da sich die Konflikte um das Lock vermehren, je mehr Threads sich um das `cli`-Lock gleichzeitig bewerben, wurden die Messungen mit unterschiedlichen Interrupt-Lasten ausgeführt. In der ersten Messung wurde der Benchmark aus Abschnitt 5.1.1 ohne IDE-Mediator und ohne parallele Lasten wiederholt. Dabei treten bei den IO-intensiven Teilen des Benchmarks bis zu 52 `cli`-Operationen pro Millisekunde auf. Da es wenig Konflikte gibt, werden entsprechend wenige IPCs zum Lock-Thread ausgeführt (etwa eine IPC pro Sekunde).

Die Situation ändert sich nur wenig, wenn der IDE-Mediator aktiviert wird: Durch die Kontrolle der DMA-Adressen verlängert sich etwas die Ausführungszeit für die Linux-Top-Halves. Dadurch steigt die Chance, dass sich während dieser Zeit ein weiterer Interrupt auftritt. Da Top-Halves durch das `cli`-Lock abgesichert sind, treten damit ein wenig mehr Konflikte um das Lock auf.

Wesentlich größeren Einfluss hat der parallele Betrieb eines weiteren Gerätes. Damit steigt die Interrupt-Last stärker an, und die Konflikte zwischen Threads mehren sich. Bei *Netzlast 100 MBit* wurde parallel zum Anwendungsbenchmark eine große Datei immer wieder von einem externen Server gelesen (ähnlich dem `wget`-Benchmark in Abschnitt 5.2.1). Dabei wurde der Fast-Ethernet-Adapter Digital DS21143 Tulip rev 65 verwendet. Mit dieser Parallellast steigt die Anzahl der `cli`-Instruktionen bereits deutlich an, und es treten 50- bis 170-mal so viele Konflikte auf. Mit dem Fast-Ethernet-Adapter wird auf dem Celeron-4-System eine Grundlast von etwa 65 % erzeugt.

Bei Verwendung von Gigabit-Ethernet (Intel PRO/1000) werden noch mehr Interrupts pro Zeiteinheit generiert, aufgrund des Interrupt-Coalescing allerdings nicht 10-mal so viel – trotz 10-facher Bruttodatenrate. Die CPU-Last beträgt nun mehr als 90 % (vgl. Abschnitt 5.2.2). Die Anzahl der `cli`-Operationen pro Sekunde ist etwa gleichbedeutend mit der Belastung durch Fast-Ethernet, allerdings treten etwas mehr Konflikte am `cli`-Lock auf. Dies führt zu etwa 600 IPCs zum Lock-Thread pro Sekunde. Gegenüber dem System ohne Parallellast treten 55-mal mehr IPC Operationen

auf. Unter der Voraussetzung, dass ein Konflikt auf einem Pentium 4 etwa 6000 Takte kostet (Abschnitt 4.1.1), entspricht dies einer zusätzlichen Prozessorlast von 0,18 % bei 2 GHz CPU-Takt.

Kapitel 6

Schlussfolgerungen und Ausblick

6.1 Beiträge dieser Arbeit

Mit dieser Arbeit wurden folgende Ergebnisse erzielt:

1. Ausgehend von der bisher vorhandenen L^{4}Linux-Implementierung wurde gezeigt, wie der Linux-Kern mit eingeschränkten IO-Rechten ausgeführt werden kann (Abschnitt 3.1).

2. Für die Lösung des DMA-Problems wurde ein allgemeines Modell einer IOMMU entwickelt (Abschnitt 3.2) und gezeigt, wie sich dieses in zukünftiger Hardware umsetzen lässt (Abschnitte 3.2.2 bis 3.2.3).

3. Im Hinblick auf aktuelle Standard-Hardware ohne spezielle Erweiterungen wurde eine Methode entwickelt, wie Busmaster-DMA mittels Teil-Virtualisierung von Geräten kontrolliert werden kann (Abschnitt 3.2.4). Dabei wurde gezeigt, welchen Einschränkungen diese Technik in Bezug auf spezielle Hardware (Firmware) unterliegt.

4. In Abschnitt 4.3 wurden Beispielimplementierungen für die Virtualisierung von zwei Netzwerkadaptern und eine Klasse von IDE-Controllern gezeigt. Dabei wurde eine erhebliche Verminderung des Umfangs an vertrauenswürdigem Code erreicht (Abschnitt 4.3.5).

5. Es wurde gezeigt, dass die Kosten für die DMA-Virtualisierung und von langsameren Geräten im vertretbaren Rahmen liegen (Abschnitte 5.1 und 5.2). Bei Geräten mit hoher Interrupt-Last (Gigabit-Ethernet) sind die Kosten entsprechend höher. Die Kosten für die Einschränkung der IO-Rechte sind gering (Abschnitt 5.4).

6. Es wurden allgemeine Performance-Vergleiche zwischen Linux und L^{4}Linux durchgeführt. Insbesondere wurde der Einfluss von kleinen Adressräumen auf aktueller Hardware untersucht. Dabei konnten je nach System und Anwendungsfall Einsparungen in der CPU-Last von bis zu 12 % nachgewiesen werden (Abschnitt 5.3 und 5.2.2).

Folgende Nebenergebnisse wurden erzielt:

1. Für den Mikrokern Fiasco wurde die gleichzeitige Nutzung von kleinen Adressräumen und der Einschränkung von IO-Rechten für Nutzerprozesse implementiert.

2. Die IPC-Performance von Fiasco wurde ausführlich auf verschiedenen x86-Systemen vermessen (Abschnitte C.1, C.2 und C.3).

6.2 Vorschläge für zukünftige Arbeiten

Die im Rahmen dieser Arbeit vorgenommenen Implementierungen wurden anhand von L^{4}Linux Version 2.2 vorgenommen. Alle hier beschriebenen Techniken sollten sich auch auf aktuellere Versionen von L^{4}Linux und auf andere Betriebssysteme anwenden lassen. Zunächst erscheint es sinnvoll, die aktuelle Portierung von Linux 2.6 auf Nizza zu untersuchen. Bei dieser Version wurde versucht, die Änderungen am Linux-Kern durch Hinzufügen bestimmter Erweiterungen im Mikrokern zu verringern.

Sobald Unterstützung für IO-Adressräume in Hardware erhältlich ist, sollte Nizza daran angepasst werden. Intels Vanderpool Technology [49] wird die Virtualisierung von herkömmlichen Betriebssystemen auf x86-Hardware vereinfachen. Dabei muss aber untersucht werden, welche zusätzlichen Kosten entstehen, z. B. wie teuer Wechsel zwischen Root Mode und Gast sind und wie oft diese ausgeführt werden müssen.

Der Fiasco-Mikrokern sollte einer genaueren Performance-Analyse unterzogen werden. Für die Unterscheidung, ob eine IPC im Fast Path oder Slow Path ausgeführt werden kann, müssen etwa 15 Bedingungen überprüft werden. Der IPC-Pfad ist sehr komplex und vollständig unterbrechbar. In einer aktuellen Arbeit wird daher untersucht, inwieweit sich der IPC-Pfad von Fiasco vereinfachen lässt.

Betriebssystem-Kerne werden für die direkte Ausführung auf der Hardware entworfen und nutzen daher meist Besonderheiten der Hardware zur Steigerung der Performance. Während dem Kern bei der Para-Virtualisierung nicht alle diese Besonderheiten offeriert werden, erwarten allerdings die Anwendungen die für eine Hardware-Architektur bestimmte Schnittstelle, egal ob der Kern direkt auf die Hardware zugreift oder in einer virtuellen Maschine ausgeführt wird. Beispielsweise können Anwendungen ab Linux 2.6 ein Segmentregister für den schnellen Zugriff auf threadlokalen Speicher nutzen (*Thread Local Storage*, *TLS* [17]). Der VM-Monitor muss die spezielle Semantik dieses Registers beachten und das entsprechende Segment bei Kontextwechseln geeignet umschalten. Spezielle Anforderungen dieser Art von Anwendungen an das ABI des Kerns führen zu zusätzlichem Aufwand im Mikrokern, teilweise in oft benutzten („heißen“) Pfaden. Alle bekannten Implementierungen von Mikrokernen mit L4-Schnittstellen haben Lücken bei der Unterstützung solcher Besonderheiten. Es ist erforderlich, den erforderlichen zusätzlichen Aufwand zu untersuchen.

Die verwendete L4-Schnittstelle V.2 bietet keine ausreichende Kontrolle der Kommunikation zwischen Aktivitäten. So werden zwar nicht vertrauenswürdige Prozesse voreinander durch Adressräume separiert, sie können aber mittels IPC direkt miteinander kommunizieren. Ursprünglich vorgesehene Erweiterungen (z. B. Clans & Chiefs [54]) erwiesen sich als in der Praxis zu unflexibel und wurden in aktuellen L4-Kernen nicht implementiert. Es müssen daher andere Mechanismen zur Kommunikationskontrolle gefunden werden.

Weitere Arbeiten sollten den Einfluss von Cache-Coloring auf die Performance von L^{4}Linux und daneben ausgeführten Anwendungen zeigen.

Anhang A

Testumgebung

Grundlage für die Messungen in dieser Arbeit (insbesondere Kapitel 4 und 5) bilden zwei PCs mit folgender Ausstattung:

CPU	**Intel Pentium III (Coppermine) / 800 MHz**
Caches	32 KB L1-Cache und 256 KB L2-Cache
Code-TLBs	32 Einträge für 4 KB-Seiten, 2 Einträge für 4 MB-Seiten
Daten-TLBs	64 Einträge für 4 KB-Seiten, 8 Einträge für 4 MB-Seiten
RAM	256 MB SDRAM
IDE-Controller	VIA VT82C586 Chipsatz
Netzwerkadapter	Digital DS21143 Tulip rev 65 (100 MBit Fast Ethernet)
	Intel PRO/1000 rev 02 (1000 MBit Gigabit Ethernet)
Festplatte	IBM-DTLA-305020 (20 GB) mit 380 KB Cache
Speicherbandbreite	388 MB/s (2.06 CPU-Zyklen / Byte)

CPU	**Intel Celeron 4 (Northwood) / 2 GHz**
Caches	8 KB L1-Cache, 12 K µ-ops und 128 KB L2-Cache
Code-TLBs	128 Einträge für 4 KB- bzw. 4 MB-Seiten
Daten-TLBs	64 Einträge für 4 KB- bzw. 4 MB-Seiten
RAM	512 MB DDR333 RAM
IDE-Controller	Intel 82801EB/ER (ICH5/ICH5R) Chipsatz
Netzwerkadapter	Digital DS21143 Tulip rev 65 (100 MBit Fast Ethernet)
	Intel PRO/1000 rev 02 (1000 MBit Gigabit Ethernet)
Festplatte	IBM-DTLA-305020 (20 GB) mit 380 KB Cache
Speicherbandbreite	826 MB/s (2,41 CPU-Zyklen / Byte)

Die Messungen wurden auf beiden Rechnern mit der gleichen Festplatte und den gleichen Netzwerkadaptern vorgenommen. Der Hauptunterschied zwischen beiden Rechnern liegt in der CPU: Der Celeron 4 braucht mehr Taktzyklen beim Wechsel zwischen Nutzer- und Kernmodus (siehe Anhang B.1). Weiterhin muss bei dieser CPU der Cache für Code bei Wechsel des Seitenverzeichnisses geleert werden, da dieser mit virtuellen Tags versehen ist [88].

Bei beiden PCs wurde der im Chipsatz integrierte IDE-Controller verwendet. Diese sind aus Performance-Gründen nicht physisch an den PCI-Bus angeschlossen, sondern direkt an den internen IO-Bus des Chipsatzes. Aus Sicht der CPU verhalten sie sich wie logische PCI-Geräte an einem separaten PCI-Bus.

Als Partner-Rechner für die Netzwerk-Messungen mit *netperf* wurde ein Notebook IBM Thinkpad T40p mit folgender Ausstattung gewählt:

CPU	**Intel Pentium M (Banias) / 1.6 GHz**
Caches	64 KB L1-Cache und 1024 KB L2-Cache
Code-TLBs	128 Einträge für 4 KB-Seiten, 2 Einträge für 4 MB-Seiten
Daten-TLBs	128 Einträge für 4 KB-Seiten, 8 Einträge für 4 MB-Seiten
RAM	512 MB DDR-RAM
Netzwerkadapter	Intel PRO/1000 rev 03 (1000 MBit Gigabit Ethernet)
Festplatte	HTS726060M9AT00 (60 GB) mit 204 8 KB Cache
Speicherbandbreite	904 MB/s (1.76 CPU-Zyklen / Byte)

Alle Messungen wurden unter Linux 2.2.26 bzw. L^{4}Linux 2.2.26 auf der Linux-Distribution Debian 3.0 ausgeführt. Die Linux-Kerne wurden jeweils mit dem Compiler gcc mit der Version 2.95.4 compiliert, für Fiasco wurde gcc Version 3.4.4 verwendet. Für den Kernein- und -austritt bei IPC wurden, wenn auf dem System vorhanden, die optimierten Instruktionen `sysenter` und `sysexit` verwendet.

Für die Netzwerk-Messungen über TCP (Abschnitt 5.2) wurde die maximale Größe des TCP-Fensters von Linux und L^{4}Linux auf 512 KB gesetzt (Anpassung der Konstante `MAX_WINDOW` in der Datei `include/net/tcp.h`). Weiterhin wurden die Default- und Maximalgrößen der Sende- und Empfangspuffer auf 256 KB gesetzt, wie in [87] vorgeschlagen.

Für CPU-Lastmessungen wurden auf den PCs Performance-Counter verwendet, die die Taktzyklen zählen, während der sich der Prozessor nicht im `HLT`-Zustand befindet (Event-Nummer 0x79 auf Pentium III und Event-Nummer 0x13 auf dem Celeron 4). Sowohl Linux als auch Fiasco führen den Befehl `hlt` in der Idle-Loop aus, die die niedrigste Priorität im System besitzt. In Verbindung mit dem *Time Stamp Counter* [47] können damit präzise Zeit- und CPU-Lastmessungen vorgenommen werden.

In den Anhängen B und C wurden die ermittelten Werte für das Pentium-III-System sowie für das Celeron-4-System farblich hervorgehoben.

Anhang B

Kenngrößen ausgewählter Prozessoren

B.1 Kosten für wichtige Prozessorinstruktionen

Für die Erfassung der Kosten von privilegierten Befehlen auf aktueller Hardware wurde ein selbst entwickeltes Testprogramm für den privilegierten Modus verwendet, das die Befehle in einer Schleife mit 200000 Durchläufen ausführt und die gemessene Zeit durch die Anzahl der Durchläufe dividiert. In der folgenden Tabelle sind die Ergebnisse in Taktzyklen der CPU dargestellt.

	MHz	`int` + `iret`	`sysenter` + `sysexit`	modify + load `ES`	`cli` + `sti`	flush TLB	switch Ptab
Pentium (P54)	90	186	–	15	18	46	39
K6-II (Chomper)	350	118	–	12	29	341	189
Pentium III (Coppermine)	800	214	51	19	20	116	83
Athlon (Thunderbird)	800	202	67	18	10	92	102
Pentium M (Banias)	1600	257	66	19	19	137	94
Celeron 4 (Northwood)	2000	932	146	52	86	520	292

Das Modifizieren und Laden eines Segmentregisters ist eine Operation, die beim Umschalten auf einen kleinen Adressraum benötigt wird. Die Werte für *flush TLB* zeigen die benötigte Zeit für das Invalidieren eines bestimmten TLB-Eintrags. *Switch ptab* enthält die Kosten für das Umschalten des Seitenverzeichnisses inklusive Invalidieren des TLBs. In der Spalte *MHz* ist die Taktfrequenz des verwendeten Prozessors dargestellt.

Zu erkennen ist die Tendenz, dass privilegierte Befehle auf neueren Prozessoren mehr Taktzyklen benötigen. Der Kernein- und -austritt mittels `sysenter` und `sysexit` benötigt auf jeder CPU wesentlich weniger Zeit als die herkömmliche Variante mittels `int` und `iret`.

Die hier dargestellten Kosten für Kernein- und -austritt beziehen sich auf die Ausführung der Befehle in einer Schleife. In der Praxis werden diese Werte insbesondere bei `sysenter` und `sysexit` überschritten, da im Zusammenwirken mit dem jeweiligen Code-Kontext zusätzliche Kosten entstehen. Weiterhin sind bei beiden Befehlen im Gegensatz zu `int` und `iret` zwei Register vorbelegt, die deshalb nicht zur Parameterübergabe verwendet werden können.

B.2 Cache-Größen

In der folgenden Tabelle sind die Cache-Größen ausgewählter x86-Prozessoren dargestellt. Die Werte in den Klammern zeigen die Cache-Assoziativität. Die Spalte *Farben* zeigt die maximale Anzahl möglicher Farben des L2-Caches für Cache-Coloring.

	L1-Cache		L2-Cache		
	Code	Daten	Code	Daten	Farben
Pentium (P54)	8 KB (2)	8 KB (2)	extern 256 KB		
K6-II (Chomper)	32 KB (2)	32 KB (2)	extern 256 KB		
Pentium III (Coppermine)	16 KB (4)	16 KB (4)	256 KB (8)		8
Athlon (Thunderbird)	64 KB (2)	64 KB (2)	256 KB (16)		4
Pentium M (Banias)	32 KB (8)	32 KB (8)	1024 KB (8)		32
Celeron 4 (Northwood)	12 K µ-ops (8)	8 KB (4)	128 KB (2)		16
Pentium 4 (Prescott)	12 K µ-ops (8)	16 KB (8)	1024 KB (8)		32
Athlon 64 (Winchester)	64 KB (2)	64 KB (2)	1024 KB (16)		16

Die Werte wurden, wenn möglich, mit der Instruktion `cpuid` ermittelt. Fehlende Werte wurden aus [65] ergänzt. Die Anzahl der Farben für Cache-Coloring kann wie folgt ermittelt werden:

$$\text{Anzahl Farben} = \frac{\text{Cache-Größe}}{\text{Seitengröße} \times \text{Cache-Assoziativität}}$$

Die Seitengröße beträgt auf x86-Prozessoren üblicherweise 4 KB.

B.3 TLB-Größen und -Zugriffszeiten

In der folgenden Tabelle ist die Anzahl an TLB-Einträgen ausgewählter x86-Prozessoren und in Klammern deren Assoziativität angegeben. Die Werte wurden mit dem Befehl `cpuid` ermittelt und ggf. aus [65] ergänzt.

	Code		Daten	
	4 KB	4 MB	4 KB	4 MB
Pentium (P54)	gemeinsam 32 (4)		64 (4)	8 (4)
K6-II (Chomper)	64 (1)	1 (n)	128 (2)	2 (n)
Pentium III (Coppermine)	32 (4)	2 (n)	64 (4)	8 (n)
Athlon (Thunderbird)	16 (n) + 256 (4)	4 (n)	24 (n) + 256 (4)	4 (n)
Pentium M (Banias)	128 (4)	2 (n)	128 (4)	8 (4)
Celeron 4 (Northwood)	gemeinsam 128 (n)		gemeinsam 64 (n)	
Pentium 4 (Prescott)	gemeinsam 128 (n)		gemeinsam 64 (n)	
Athlon 64 (Winchester)	32 (n) + 512 (4)	4 (n)	32 (n) + 512 (4)	4 (n)

In der Tabelle ist zu erkennen, dass neuere Prozessoren tendenziell mehr TLB-Einträge besitzen. Je mehr TLB-Einträge vorhanden sind und je mehr das Neuladen eines Eintrags kostet, desto mehr

indirekte Kosten fallen potenziell nach dem Umschalten auf einen Adressraum durch Laden von TLBs an.

Die folgende Tabelle zeigt die Kosten für das Laden eines 4 KB-TLB-Eintrags für Code bzw. Daten. Dabei wurde mit warmem Cache gemessen, d. h. die TLB-Einträge konnten direkt aus dem Cache geladen werden. Da die Caches auf x86-Prozessoren mit physischen Tags versehen sind, müssen sie bei Adressraumwechseln nicht geleert werden.

	MHz	Code	Daten	Reload Einträge 4 KB	
		4 KB	4 KB	CPU-Zyklen	Zeit
Pentium (P54)	90	42	17	ca. 2500	28 µs
K6-II (Chomper)	350	38	37	ca. 7200	21 µs
Pentium III (Coppermine)	800	24	8	ca. 1300	1,6 µs
Athlon (Thunderbird)	800	54	76	ca. 33000	41 µs
Pentium M (Banias)	1600	31	8	ca. 5000	3,1 µs
Celeron 4 (Northwood)	2000	36	48	ca. 7700	3,9 µs

Diese hier ermittelten Werte helfen bei der Einschätzung der Kosten von Adressraumwechseln, bei denen die Seitentabelle umgeschaltet wird. Im Worst Case führen mehr als 7000 Takte beim Celeron 4 bei einer Gesamtdauer der IPC von 2000 bzw. 3800 Takten (siehe Abschnitt C.2) zu erheblichen indirekten Kosten.

Für einen Pentium 4 (1,4 GHz) wurden in [88] etwas abweichende Angaben gemessen (31 Zyklen für einen Code-TLB-Eintrag, 48 Zyklen für einen Daten-TLB-Eintrag).

Anhang C

IPC-Messungen unter Fiasco

Für die Performance eines Mikrokern-basierten Systems hat die IPC-Performance des zugrunde liegenden Mikrokerns eine wichtige Bedeutung, da aufgrund der Auslagerung jeglicher Ressourcenverwaltung in Nutzerprozesse und der verstärkten Nutzung von Adressräumen Kommunikation zwischen Prozessen häufiger stattfindet als bei herkömmlichen monolithischen Betriebssystemen [56].

In diesem Kapitel werden einige ausgewählte IPC-Messungen mit Fiasco präsentiert. Dabei wurden mit Hilfe des selbst entwickelten L4-Programms `pingpong` die direkten Kosten zwischen zwei Threads gemessen, die sich je nach Messung in gleichen oder verschiedenen Adressräumen befinden.

Ein Thread agiert als Client, ein anderer als Server. Da der Server üblicherweise dem Client nicht vertraut, sendet der Server die Antwort mit einem Timeout von Null, d. h. der Client muss die Antwort sofort abnehmen. Anderenfalls müsste der Server auf den Client warten und wäre damit blockiert, da IPC unter L4 immer synchron ausgeführt wird.

Für die Messungen wurden die IPC-Operationen in einer Schleife ausgeführt, die 100.000-mal durchlaufen wird, um Ungenauigkeiten durch Cache-Effekte und Interrupts auszuschließen.

Neben den direkten Kosten, die durch die Ausführung von Instruktionen im Kern-IPC-Pfad entstehen, müssen auch folgende indirekten Kosten beachtet werden:

- Findet während der IPC ein Adressraumwechsel statt, muss auf x86-Systemen der gesamte TLB invalidiert werden, da die Einträge keine Adressraum-IDs enthalten. Indirekte Kosten entstehen dadurch, dass im neuen Adressraum TLB-Einträge wieder geladen werden müssen.

- Der Kern greift auf Code und Daten zu und führt bedingte Sprünge aus. Die dabei verdrängten TLB-Einträge, Cache-Lines und Einträge im *Branch Target Buffer* werden bei der Ausführung des Nutzerprogramms wieder neu geladen. Dadurch entstehen weitere indirekte Kosten.

Die indirekten Kosten einer IPC können im Worst Case wesentlich größer sein als die direkten Kosten.

Alle Ergebnisse stellen *Roundtrip-Zeiten* dar, also die Gesamtzeit für die Hin- und für die Rück-IPC. *Fast-IPC* nimmt einen optimierten Kernpfad für Short-IPC, der in Assembler implementiert wurde. *Slow* zeigt Kosten für den normalen C++-Pfad einer Short-IPC. Dieser langsamere Pfad wird im Kern genommen, falls bestimmte Bedingungen nicht erfüllt sind, z. B. wenn der Empfänger einer IPC nicht bereit ist oder wenn einer der Partner einen IPC-Timeout ungleich Null oder Unendlich angegeben hat [75]. *PF* zeigt Kosten, die bei der Behandlung eines Seitenfehlers entstehen.

Fiasco wurde für diese Messungen mit Unterstützung für eingeschränkte IO-Rechte und für kleine Adressräume compiliert. Die Unterstützung kleiner Adressräume im Kern impliziert zusätzliche Kosten (siehe Abschnitt 4.2.2).

C.1 IPC innerhalb eines Adressraumes

In der folgenden Tabelle sind die direkten Kosten für Hin- und Rück-IPC (Roundtrip) zwischen zwei Threads innerhalb des selben Adressraumes dargestellt. IPC innerhalb eines Adressraumes wird meist für Synchronisationszwecke genutzt. Beispiele in L^{4}Linux sind das `cli`-Lock und das Aufwecken des Bottom-Half-Threads durch einen Top-Half-Thread.

	MHz	**Fast**	Slow
Pentium (P54)	90	**481**	2183
K6-II (Chomper)	350	**560**	2137
Pentium III (Coppermine)	800	**508**	1823
Athlon (Thunderbird)	800	**361**	1292
Pentium M (Banias)	1600	**578**	1826
Celeron 4 (Northwood)	2000	**1456**	2991

C.2 IPC zwischen zwei Adressräumen

In der folgenden Tabelle sind die direkten Kosten für Hin- und Rück-IPC (Roundtrip) zwischen zwei Threads verschiedener Adressräume dargestellt, wobei sich der Server (analog zum L^{4}Linux-Server) in einem kleinen Adressraum befindet (nur Spalte *Small*). Falls der Kern kleine Adressräume unterstützt, wird bei dieser IPC die Seitentabelle nicht gewechselt, somit muss der TLB nicht invalidiert werden. Die Messwerte unter *Large* wurden für Threads in zwei großen Adressräumen ausgeführt. Werte unter *IO* beziehen sich auf eingeschränkte IO-Rechte.

	MHz	Large			Large, IO			Small, IO		
		Fast	Slow	PF	**Fast**	Slow	PF	**Fast**	Slow	PF
Pentium (P54)	90	**943**	3224	5625	**943**	3409	5149	**724**	2762	5217
K6-II (Chomper)	350	**938**	2491	3782	**938**	2447	3636	**638**	2196	3886
Pentium III (Coppermine)	800	**654**	1846	2991	**705**	1902	3086	**612**	1880	3062
Athlon (Thunderbird)	800	**678**	1572	2805	**729**	1603	2940	**461**	1348	2675
Pentium M (Banias)	1600	**633**	1768	2827	**680**	1839	2921	**679**	1877	2884
Celeron 4 (Northwood)	2000	**1986**	4379	8370	**2193**	4373	8643	**1982**	3772	6848

Bei diesen Messungen wurde unter anderem untersucht, welche zusätzlichen Kosten durch Implementierung der Unterstützung für eingeschränkte IO-Rechte entstehen. Auf den verwendeten Systemen treten Zusatzkosten bis zu 10 % der Zyklen für den IPC-Pfad auf. Der Schritt von *Large, IO* nach *Small, IO* zeigt geringere Kosten für kleine Adressräume, wenn die Seitentabelle nicht umgeschaltet wird. Auf CPUs, auf denen diese Operation sehr teuer ist (K6-II, Celeron 4) unterscheiden sich beide Messreihen am deutlichsten.

Dass kleine Adressräume auf einem Pentium-M praktisch keine Ersparnis bei den direkten IPC-Kosten liefern, könnte an der TLB-Struktur dieses Prozessors liegen. Er besitzt nur zwei TLB-Einträge für 4 MB-Code-Seiten (vgl. Anhang B.3), wovon ein Eintrag für eine Seite im IPC-Pfad von Fiasco benötigt wird.

In der folgenden Tabelle ist die Anzahl an TLB-Einträgen dargestellt, auf die Fiasco zugreift, solange nur Short IPC ausgeführt wird bzw. Exceptions behandelt werden (der Normalfall unter L^{4}Linux).

	Code		Daten	
	4 KB	4 MB	4 KB	4 MB
TCB Quelle + Ziel (inklusive Stack)			2	
Kerncode und statische Daten		1		1
Geräteregister für Timer (Local APIC)			1	
Idt (für Kerneintritt über `int`)			1	
Trampoline-Seite für kleine Adressräume	1			
IO-Bitmap			2	
GDT, TSS			1	
Summe	1	1	7	1

C.3 Vergleich von Fiasco mit dem L4-Kern von Liedtke

In der folgenden Tabelle werden die IPC-Kosten von Fiasco mit dem L4-Kern von Liedtke [56] verglichen, der in einer früheren Veröffentlichung für Performance-Messungen mit L^{4}Linux [25] zur Verfügung stand.

	MHz	**Fast-IPC**		Slow-IPC		Pagefault	
		L4	Fiasco	L4	Fiasco	L4	Fiasco
Pentium (P54)	90	**329**	**720**	369	2725	858	4741
K6-II (Chomper)	350	**463**	**631**	503	2320	1139	3556
Pentium III (Coppermine)	800	(**586**)	(**751**)	(639)	(2095)	3128	3128
Athlon (Thunderbird)	800	(**504**)	(**608**)	(524)	(1490)	1539	2662
Pentium M (Banias)	1600	(**682**)	(**822**)	(721)	(1947)	1542	2807
Celeron 4 (Northwood)	2000	(**2499**)	(**3149**)	(2470)	(4580)	6346	6659

Der L4-Kern von Liedtke ist in Assembler implementiert und speziell für gute Performance auf Pentium-Prozessoren optimiert. Der Kern besitzt im Unterschied zu Fiasco keine Unterstützung für die neuen Befehle zum Betreten und Verlassen des Kerns mittels `sysenter` und `sysexit`. Weiterhin unterstützt dieser Kern auch keine eingeschränkten IO-Rechte. Um einen Vergleich zu ermöglichen, wurde Fiasco ohne diese Eigenschaften compiliert. Beide Kerne implementieren die L4-V.2-Schnittstelle.

Fiasco wurde primär im Hinblick auf gute Vorhersagbarkeit entwickelt, allerdings wurde der Kern an einigen Stellen optimiert, z. B. durch Einführung des schnellen IPC-Pfades.

Glossar

Branch Target Buffer (BTB) Moderne Prozessoren unterteilen Maschineninstruktionen in viele kleine Teilschritte, die in einer Pipeline ausgeführt werden. Der Effekt der Performance-Steigerung entsteht dadurch, dass zu einem bestimmten Zeitpunkt pro Pipeline-Stufe ein Teilschritt einer *anderen* Instruktion ausgeführt wird, so dass immer mehrere Instruktionen gleichzeitig ausgeführt werden können. Für die Ausführung eines bedingten Sprunges muss das Sprungziel frühzeitig bekannt sein, damit die Pipeline mit den Instruktionen am Sprungziel weiter gefüllt werden kann.

Aus diesem Grund besitzen moderne Prozessoren einen Branch Target Buffer (BTB), dessen Einträge jeweils das Ziel eines bedingten Sprunges an einer bestimmten virtuellen Adresse enthalten. Das Fehlen eines BTB-Eintrags bedingt zusätzliche Kosten durch das Anhalten der Pipeline bis das Ziel des Sprunges ermittelt wurde. Der BTB eines Pentium 4 besitzt 4096 Einträge [33].

Cache In dieser Arbeit bezeichnet Cache den L1-, L2- oder L3-Cache zwischen CPU und Hauptspeicher. Es handelt sich dabei um schnelle Zwischenspeicher für Daten des Hauptspeichers, wobei diese Daten von der CPU als Instruktionen oder als Daten im Sinne von Variablen interpretiert werden können. Der Cache ist in gleichgroße Abschnitte (Cache-Lines) unterteilt, die jeweils mit einem Tag versehen sind [97].

Die Art der *Indizierung* eines Caches entscheidet über die Zuordnung von Speicher zu Cache-Lines: Bei virtuell indiziertem Cache (auch als linear indizierter Cache bezeichnet) wird die Position eines Datums im Cache durch seine virtuelle Adresse festgelegt, bei physisch indiziertem Cache durch seine physische Adresse.

Für *Cache-Coloring* (siehe Abschnitt 2.4.1) muss der Cache physisch indiziert sein, um eine für die Anwendung transparente Zuordnung zu bestimmten Cache-Partitionen zu erreichen. Um mit virtuell indiziertem Cache Cache-Coloring zu erreichen, müssten Anwendungen, die unterschiedliche Cache-Partitionen belegen sollen, an unterschiedliche Adressen gelinkt werden. Abgesehen davon muss aber virtuell indizierter Cache bei einem Adressraumwechsel geflusht werden, weshalb er nicht von mehreren Adressräumen gleichzeitig verwendet werden kann. Auf der x86-Hardware ist der L2-Cache physisch indiziert und damit für Cache-Coloring geeignet. Die Anzahl möglicher Farben ergibt sich aus

$$\text{Anzahl Farben} = \frac{\text{Cache-Größe}}{\text{Seitengröße} \times \text{Cache-Assoziativität}}$$

Damit ist Cache-Coloring meist nur auf 4 KB-Seiten einsetzbar.

Im *Tag* einer Cache-Line ist die genaue virtuelle (bei Cache mit virtuellen Tags) bzw. physische Adresse des entsprechenden Abschnitts im Hauptspeicher abgelegt, dessen Datum in der Cache-Line gehalten wird. Ändert sich die Zuordnung der gecachten virtuellen Adresse zur physischen Adresse, so muss die entsprechende Cache-Line geleert werden. Da auf der x86-Hardware alle TLBs bei Umschalten des Seitenverzeichnisses geleert werden müssen (siehe **TLB**), müssen in diesem Fall auch Caches mit virtuellen Tags geleert werden. Der *Trace-Cache* des Intel Pentium 4 besitzt virtuelle Tags.

CPL (*Current Privilege Level*) Der CPL bzw. der *Ring* bezeichnet auf x86-Systemen die Privilegstufe (siehe Abschnitt 2.3.4) eines Kontextes. Ring 0 entspricht dem Kernmodus mit Systemrechten. Nur auf dieser Privilegstufe ist beispielsweise das Umschalten der Seitentabelle erlaubt. Ring 3 entspricht dem Usermode mit eingeschränkten Rechten. Ring 1 und Ring 2 werden von Standard-Betriebssystemen normalerweise nicht benutzt. Ausnahme ist OS/2, wo auf Ring 1 bestimmte Gerätetreiber ausgeführt werden.

Physischer Adressraum Im physischen Adressraum sind Hauptspeicher sowie Register und eingeblendete Speicher der Geräte an den IO-Bussen zusammengefasst (siehe Abschnitt 2.5.3). Die Auswahl eines Gerätes erfolgt durch Zuordnung der physischen IO-Adresse zum jeweiligen Geräteregister. Register verschiedener Geräte sind deshalb an disjunkte Bereiche des physischen Adressraumes eingeblendet. Der Hauptspeicher ist durch Vermittlung der Host-Bridge Bestandteil des physischen Adressraumes und kann von IO-Geräten durch Auswahl der entsprechenden physische Adresse gelesen und beschrieben werden.

Der physische Adressraum ist in Kacheln unterteilt. Virtuelle Adressräume bestehen aus Seiten, die auf Kacheln des physischen Adressraumes abgebildet werden.

Unabhängig vom physischen Adressraum existiert auf x86-Systemen der *IO-Space*, auf den nur mittels spezieller IO-Zyklen zugegriffen werden kann.

IO-Bitmap Die IO-Bitmap kontrolliert bei x86-Systemen den Zugriff eines Prozesses mit eingeschränkten IO-Rechten auf IO-Ports (siehe Abschnitt 2.4.3).

IOMMU (*Input/Output Memory Management Unit*) Eine IOMMU verwaltet, ähnlich einer MMU, Abbildungen von Adressen der IO-Adressräume auf physische Adressen (siehe Abschnitt 2.5.4).

IOPL (*Input/Output Privilege Level*) Die IOPL bezeichnet auf x86-Systemen die Privilegstufe eines Kontextes für IO-Operationen (siehe Abschnitt 2.4.3).

Kleiner Adressraum In jedem virtuellen Adressraum wird ein Bereich reserviert, in den ein bzw. mehrere kleine Adressräume eingeblendet werden. Ein kleiner Adressraum kann nur einen Teil des virtuellen Adressraumes nutzen. Schutz zwischen kleinen und (normalen) großen Adressräumen wird durch Segmentierung erreicht (siehe Abschnitt 2.4.2 und [55]). Sobald ein Seitenfehler oberhalb der Adressraumgrenze auftritt[1], wird der kleine Adressraum transparent für den Nutzerprozess in einen großen Adressraum umgewandelt.

[1] Auf x86-Systemen wird bei Überschreitung der Segmentgrenze eine General Protection Exception ausgelöst.

MMU (*Memory Management Unit*) Die MMU stellt eine Indirektion zwischen CPU und Speicher dar. Der Programmcode arbeitet mit virtuellen Adressen, die auf dem Weg zum Hauptspeicher in physische Adressen umgewandelt werden. Die MMU ermöglicht mit Hilfe dieser Indirektion verschiedene Sichten auf den physischen Speicher, die allgemein als virtuelle Adressräume bezeichnet werden.

PCI-Bus (*Peripheral Component Interconnect Bus*) Ein IO-Bus für den Anschluss von IO-Geräten an den System-Bus mit einer Breite für Daten und Adressen von 32 oder 64 Bit, wobei 32 Bit auf Standard-Hardware üblicher sind (siehe Abschnitt 2.5.3). Für die Abbildung von 64-Bit-Adressen auf den 32-Bit PCI-Bus wird auf manchen Systemen eine **IOMMU** verwendet (siehe Abschnitt 2.5.4).

Pinning Mit Pinning wird erreicht, dass eine physische Kachel während einer bestimmten Zeit nicht ausgelagert wird. Dies ist z. B. während der Datenübertragung per Busmaster-DMA notwendig, weil in dieser Betriebsart Daten an der *MMU* vorbei übertragen werden und deshalb keine Seitenfehler generiert werden (siehe Abschnitt 2.5.4).

Prefetching Die Latenz bei Zugriffen auf den Hauptspeicher kann sehr hoch sein. Mit Prefetching kann die CPU mit dem Einlesen von Daten beginnen, bevor eine Anweisung explizit auf die Daten zugreift. Prefetching wird explizit durch spezielle Befehle oder implizit durch spekulative Ausführung erzielt.

TLB (*Translation Lookaside Buffer*) Ein schneller assoziativer Cache für Abbildungen virtueller Adressen auf Adressen des physischen Adressraumes. Auf der x86-Architektur enthalten die TLB-Einträge keine Adressraum-IDs, so dass der komplette TLB beim Umschalten des Seitenverzeichnisses invalidiert werden muss. Ein TLB-Eintrag kann als *global* gekennzeichnet sein – er ist damit für alle Adressräume gültig und wird nicht automatisch invalidiert.

Dadurch, dass nicht-globale TLB-Einträge nach einem vollständigem Flush potenziell erst wieder neu geladen werden müssen, entstehen indirekte Kosten für TLB-Misses, die umso höher ausfallen, je mehr TLB-Einträge der Prozessor besitzt. Diese Kosten können durch Adressraum-IDs (*Address Space Identifer*, ASID), wie sie auf manchen Architekturen zu finden sind, vermieden werden.

Eine Technik, mit der das Invalidieren der TLB-Einträge bei Adressraumwechseln vermieden werden kann, sind *kleine Adressräume* (siehe Abschnitt 2.4.2).

TSS (*Task State Segment*) Das TSS ist spezifisch für die x86-Architektur und enthält wichtige Datenstrukturen für die CPU, unter anderem die IO-Bitmap.

System-Bus Der System-Bus stellt eine Verbindung zwischen CPU, Hauptspeicher und Chipsatz her (siehe auch Abschnitt 2.5.3).

Literaturverzeichnis

[1] ADVANCED MICRO DEVICES, INC.: *Porting to AMD64. Frequently Asked Questions*, May 2003.

[2] ALVES, TIAGO and DON FELTON: *TrustZone: Integrated Hardware and Software Security. Enabling Trusted Computing in Embedded Systems. White Paper.* ARM Limited, July 2004.

[3] ANDREWS, JEREMY: *Linux: Kernel "Back Door" Attempt*. KernelTrap.org, November 2003. Available from: `http://kerneltrap.org/node/1584`.

[4] ARON, M., L. DELLER, K. ELPHINSTONE, T. JAEGER, J. LIEDTKE, and Y. PARK: *The SawMill Framework for Virtual Memory Diversity*. In *Proceedings of the Sixth Australasian Computer Systems Architecture Conference (ACSAC2001)*, pages 3–10, Gold Coast, Australia, January 2001.

[5] BARHAM, PAUL, BORIS DRAGOVIC, KEIR FRASER, STEVEN HAND, TIM HARRIS, ALEX HO, ROLF NEUGEBAUER, IAN PRATT, and ANDREW WARFIELD: *Xen and the Art of Virtualization*. In *Proceedings of the 19th ACM Symposium on Operating System Principles (SOSP)*, pages 164–177, Bolton Landing, NY, October 2003.

[6] BELL, D. E. and L. J. LA PADULA: *Secure Computer System: Unified Exposition and Multics Interpretation*. Technical Report ESD-TR-75-306, United States Air Force, Hanscom Air Force Base, Bedford, Massachusetts, March 1976.

[7] BERSHAD, BRIAN N., DAVID D. REDELL, and JOHN R. ELLIS: *Fast Mutual Exclusion for Uniprocessors*. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 223–233, Boston, MA, October 1992.

[8] BHATT, AJAY V.: *Creating a Third Generation I/O Interconnect*, 2004. Available from: `http://www.intel.com/technology/pciexpress/devnet/docs/WhatisPCIExpress.pdf`.

[9] BUGNION, EDOUARD, SCOTT DEVINE, KINSHUK GOVIL, and MENDEL ROSENBLUM: *Disco: Running commodity operating systems on scalable multiprocessors*. ACM Transactions on Computer Systems, 15(4):412–447, 1997.

[10] CALDERA INTERNATIONAL, INC.: *AIM Multiuser Benchmark, Suite VII*, 1996. Available from: `http://sourceforge.net/projects/aimbench/`.

[11] CARROLL, AMY, MARIO JUAREZ, JULIA POLK, and TONY LEININGER: *Microsoft "Palladium": A Business Overview*, August 2002. Available from: `http://www.microsoft.com/PressPass/features/2002/jul02/0724palladiumwp.asp`.

[12] CHOU, ANDY, JUNFENG YANG, BENJAMIN CHELF, SETH HALLEM, and DAWSON ENGLER: *An empirical study of operating systems errors*. In *Proceedings of the 18th ACM Symposium on Operating System Principles (SOSP)*, pages 73–88, Banff, Alberta, Canada, 2001. ACM Press.

[13] CRAM, ELLEN: *Migrating Applications to NGSCB. Presented at Intel Developer Forum*. Microsoft Corporation, September 2003. Available from: `http://www.intel.com/idf/us/fall2003/presentations/F03USSCMS22_OS.pdf`.

[14] CREASY, R. J.: *The Origin of the VM/370 Time-Sharing System*, volume 25(5), page 483. IBM Journal of Research and Development, 1981.

[15] *c't Browsercheck. Demos für den Internet Explorer*, Dezember 2004. Available from: `http://www.heise.de/security/dienste/browsercheck/demos/ie/`.

[16] *c't Browsercheck. Netscape Demos*, Dezember 2004. Available from: `http://www.heise.de/security/dienste/browsercheck/demos/nc/`.

[17] DREPPER, ULRICH: *ELF Handling for Thread-Local Storage. Version 0.20*. Red Hat Inc., February 2003.

[18] ENGLER, D., M. F. KAASHOEK, and J O'TOOLE: *Exokernel, An Operating System Architecture for Application-Level Resource Management*. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP)*, pages 251–266, Copper Mountain Resort, CO, December 1995.

[19] FORD, BRYAN, GODMAR BACK, GREG BENSON, JAY LEPREAU, ALBERT LIN, and OLIN SHIVERS: *The Flux OSKit: A Substrate for Kernel and Language Research*. In *Proceedings of the 16th ACM Symposium on Operating System Principles (SOSP)*, pages 38–51, Saint-Malo, France, October 1997.

[20] FRASER, KEIR, STEVEN HAND, IAN PRATT, and ANDREW WARFIELD: *Safe Hardware Access with the Xen Virtual Machine Monitor*. In *Proceedings of the 1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS 2004)*, Boston, MA, October 2004.

[21] GARFINKEL, TAL, BEN PFAFF, JIM CHOW, MENDEL ROSENBLUM, and DAN BONEH: *Terra: A Virtual Machine-Based Platform for Trusted Computing*. In *Proceedings of the 19th ACM Symposium on Operating System Principles (SOSP)*, pages 207–222, Bolton Landing, NY, October 2003.

[22] GASSER, MORRIE: *Building a Secure Computer System*. Van Nostrand Reinhold Co., 1988.

[23] GRIESSBACH, GERD: *USB for DROPS*. Master's thesis, Technische Universität Dresden, Institute for System Architecture, March 2003. Available from:

`http://os.inf.tu-dresden.de/project/finished/finished.xml.en#griessbach-diplom`.

[24] GUNINSKI, GEORGI: *Internet Explorer security – Georgi Guninski Security Research*, December 2004. Available from: `http://www.guninski.com/browsers.html`.

[25] HÄRTIG, H., M. HOHMUTH, J. LIEDTKE, S. SCHÖNBERG, and J. WOLTER: *The performance of µ-kernel-based systems*. In *Proceedings of the 16th ACM Symposium on Operating System Principles (SOSP)*, pages 66–77, Saint-Malo, France, October 1997.

[26] HÄRTIG, H., J. LÖSER, F. MEHNERT, L. REUTHER, M. POHLACK, and A. WARG: *An I/O Architecture for Microkernel-Based Operating Systems*. Technical Report TUD-FI03-08-Juli-2003, Dresden University of Technology, Dresden, Germany, July 2003.

[27] HÄRTIG, HERMANN: *Security Architectures Revisited*. In *Proceedings of the Tenth ACM SIGOPS European Workshop*, Saint-Emilion, France, September 2002.

[28] HÄRTIG, HERMANN, MICHAEL HOHMUTH, and JEAN WOLTER: *Taming Linux*. In *Proceedings of the 5th Annual Australasian Conference on Parallel And Real-Time Systems (PART '98)*, Adelaide, Australia, September 1998.

[29] HELMUTH, C., A. WESTFELD, and M. SOBIREY: *µSINA - Eine mikrokernbasierte Systemarchitektur für sichere Systemkomponenten*. In *Deutscher IT-Sicherheitskongress des BSI*, volume 8 of *IT-Sicherheit im verteilten Chaos*, pages 439–453. Secumedia-Verlag Ingelsheim, May 2003.

[30] HELMUTH, CHRISTIAN: *Generische Portierung von Linux-Gerätetreibern auf die DROPS-Architektur*. Diplomarbeit, TU Dresden, 2001. Available from: `http://os.inf.tu-dresden.de/project/finished/finished.xml.de#helmuth-diplom`.

[31] HELMUTH, CHRISTIAN, ALEXANDER WARG, and NORMAN FESKE: *Mikro-SINA—Hands-on Experiences with the Nizza Security Architecture*. In *Proceedings of the D.A.CH Security 2005*, Darmstadt, Germany, March 2005.

[32] HEWLETT PACKARD LTD. INFORMATION NETWORKS DIVISION — NETWORKING PERFORMANCE TEAM: *Netperf: Network Performance Benchmark*, Revision 2.1 edition, 1996. Available from: `http://www.netperf.org`.

[33] HINTON, GLENN, DAVE SAGER, MIKE UPTON, DARELL BOGGS, DOUG CARMEAN, ALAN KYKER, and PATRICE ROUSSEL: *The Microarchitecture of the Pentium 4 Processor*. Intel Technology Journal, Q1, 2001.

[34] HOFFMANN, SARAH: *Kleine Adressräume für Fiasco*. Technische Universität Dresden, Institut für Systemarchitektur, August 2002. Großer Beleg.

[35] HOHMUTH, M. and H. HÄRTIG: *Pragmatic nonblocking synchronization for real-time systems*. In *USENIX Annual Technical Conference*, Boston, MA, June 2001.

[36] HOHMUTH, M., H. TEWS, and S. G. STEPHENS: *Applying source-code verification to a microkernel — the VFiasco project (extended abstract)*. In *Proceedings of the Tenth ACM SIGOPS European Workshop*, September 2002.

[37] HOHMUTH, MICHAEL, MICHAEL PETER, HERMANN HÄRTIG, and JONATHAN S. SHAPIRO: *Reducing TCB size by using untrusted components — small kernels versus virtual-machine monitors*. In *Proceedings of the Eleventh ACM SIGOPS European Workshop*, Leuven, Belgium, September 2004.

[38] INTEL CORPORATION: *8259A Programmable Interrupt Controller (8259A/8259A-2)*, December 1988. Order Number: 231468-003.

[39] INTEL CORPORATION: *8254 Programmable Interval Timer*, September 1993. Order Number: 231164-005.

[40] INTEL CORPORATION: *21143 PCI/CardBus 10/100Mb/s Ethernet LAN Controller. Hardware Reference Manual — Revision 1.0*, October 1998. Order Number: 278074-001.

[41] INTEL CORPORATION: *AGP V3.0 Interface Specification – Revision 1.0*, September 2002.

[42] INTEL CORPORATION: *Linux Gigabit Adapter Base Driver*, August 2002.

[43] INTEL CORPORATION: *Intel 8255x 10/100 Mbps Ethernet Controller Family. Open Source Software Developer Manual – Revision 1.0*, January 2003.

[44] INTEL CORPORATION: *LaGrande Technology Architectural Overview*, September 2003. Available from: `http://www.intel.com/technology/security/downloads/LT_Arch_Overview.htm`.

[45] INTEL CORPORATION, P.O. Box 5937, Denver, CO 80217-9808: *IA-32 Intel Architecture Software Developer's Manal, Volume 1: Basic Architecture*, 2004. Order Number: 25366515.

[46] INTEL CORPORATION, P.O. Box 5937, Denver, CO 80217-9808: *IA-32 Intel Architecture Software Developer's Manal, Volume 2B: Instruction Set Reference, N-Z*, 2004. Order Number: 25366715.

[47] INTEL CORPORATION, P.O. Box 5937, Denver, CO 80217-9808: *IA-32 Intel Architecture Software Developer's Manal, Volume 3: System Programming Guide*, 2004. Order Number: 25366815.

[48] INTEL CORPORATION, P.O. Box 5937, Denver, CO 80217-9808: *Intel 82801AA (ICH) and Intel 82801AB (ICH0) I/O Controller Hub*, 2004. Order Number: 290655-003.

[49] INTEL CORPORATION: *Intel Vanderpool Technology for IA-32 Processors (VT-x). Preliminary Specification*, January 2005. Order Number: C97063-001.

[50] JIM, TREVOR, GREG MORRISETT, DAN GROSSMAN, MICHAEL HICKS, JAMES CHENEY, and YANLING WANG: *Cyclone: A Safe Dialect of C*. In *Proceedings of the USENIX Annual Technical Conference*, pages 275–288, Monterey, CA, USA, June 2002.

[51] KÅGSTRÖM, SIMON and RICKARD MOLIN: *A device driver framework for multiserver operating systems with untrusted memory servers*. Master's thesis, University of Karlsruhe, Dept. of Computer Science, System Architecture Group, August 2002.

[52] LESLIE, BEN and GERNOT HEISER: *Towards untrusted device drivers*. Technical Report UNSW-CSE-TR-0303, The University of New South Wales, 2003.

[53] LEVASSEUR, J., V. UHLIG, J. STOESS, and S. GÖTZ: *Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines*. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 17–30, San Francisco, CA, December 2004.

[54] LIEDTKE, J.: *Clans & chiefs, a new kernel level concept for operating systems*. Arbeitspapiere der GMD No. 579, GMD — German National Research Center for Information Technology, Sankt Augustin, 1991.

[55] LIEDTKE, J.: *Improved address-space switching on Pentium processors by transparently multiplexing user address spaces*. Arbeitspapiere der GMD No. 933, GMD — German National Research Center for Information Technology, Sankt Augustin, September 1995.

[56] LIEDTKE, J.: *On µ-kernel construction.* In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP)*, pages 237–250, Copper Mountain Resort, CO, December 1995.

[57] LIEDTKE, J.: *L4 reference manual (486, Pentium, PPro)*. Arbeitspapiere der GMD No. 1021, GMD — German National Research Center for Information Technology, Sankt Augustin, September 1996. Also Research Report RC 20549, IBM T. J. Watson Research Center, Yorktown Heights, NY, September 1996.

[58] LIEDTKE, J., H. HÄRTIG, and M. HOHMUTH: *OS-controlled cache predictability for real-time systems*. In *Third IEEE Real-time Technology and Applications Symposium (RTAS)*, pages 213–223, Montreal, Canada, June 1997.

[59] LIEDTKE, JOCHEN: *Lava Nucleus (LN) Reference Manual. Version 2.2.* IBM T. J. Watson Research Center, March 1998.

[60] LIEDTKE, JOCHEN: *L4 Nucleus Version X Reference Manual*. University of Karlsruhe, Dept. of Computer Science, System Architecture Group, September 1999.

[61] LI, JINYUAN, MAXWELL KROHN, DAVID MAZIÈRES, and DENNIS SHASHA: *Secure Untrusted Data Repository (SUNDR)*. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 121–136, San Francisco, CA, December 2004.

[62] LOESER, JORK and HERMANN HÄRTIG: *Low-latency Hard Real-Time Communication over Switched Ethernet*. In *Proceedings of the 16th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 13–22, Catania, Italy, June 2004.

[63] LÖSER, J. and M. HOHMUTH: *Omega0 – a portable interface to interrupt hardware for L4 systems*. In *Proceedings of the First Workshop on Common Microkernel System Platforms*, Kiawah Island, SC, USA, December 1999.

[64] LÖSER, JORK, LARS REUTHER, and HERMANN HÄRTIG: *A streaming interface for real-time interprocess communication*. Technical Report TUD-FI01-09-August-2001, TU Dresden, August 2001. Available from: `http://os.inf.tu-dresden.de/~jork/dsi_tech_200108.ps`.

[65] LUDLOFF, CHRISTIAN: *Sandpile.org*, December 2004. Available from: `http://www.sandpile.org/`.

[66] MAYER, R. A. and L. H. SEAWRIGHT: *A virtual machine time sharing system*. IBM Systems Journal, 9(3):199–218, 1970.

[67] MEHNERT, F., M. HOHMUTH, and H. HÄRTIG: *Cost and benefit of separate address spaces in real-time operating systems*. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS)*, pages 124–133, Austin, Texas, USA, December 2002.

[68] MENZER, MAREK: *Entwicklung eines Blockgeräte-Frameworks für DROPS*. Diplomarbeit, Technische Universität Dresden, Institut für Systemarchitektur, September 2004. In German. Available from: `http://os.inf.tu-dresden.de/project/finished/finished.xml.en#menzer-diplom`.

[69] MÉRILLON, FABRICE, LAURENT RÉVEILLÈRE, CHARLES CONSEL, RENAUD MARLET und GILLES MULLER: *Devil: An IDL for Hardware Programming*. In: *Proceedings of the 4th USENIX Symposium on Operating System Design and Implementation (OSDI)*, Seiten 17–30, San Diego, CA, Oktober 2000.

[70] MICROSOFT CORPORATION: *Hardware Platform for the Next-Generation Secure Computing Base*, 2003. Available from: `http://www.microsoft.com/resources/ngscb/documents/NGSCBhardware.doc`.

[71] MOSBERGER, DAVID, PETER DRUSCHEL, and LARRY L. PETERSON: *Implementing Atomic Sequences on Uniprocessors Using Rollforward*. Software—Practice and Experience, 26(1):1–23, 1996.

[72] MOTOROLA SEMICONDUCTOR: *Real-Time Clock Plus RAM (RTC) – Advance Information*, 1988. Order Number: MC146818/D.

[73] PACKARD, HEWLETT: *IDF 2002: HP zx1 chipset*. Available from: `http://www.hp.com/products1/itanium/idf/chipset/index.html`.

[74] PCI SPECICAL INTEREST GROUP, 5440 SW Westgate Drive, Suite 217, Portland, Oregon, 97221: *PCI Local Bus Specification – Revision 2.3*, March 2002.

[75] PETER, MICHAEL: *Leistungs-Analyse und -Optimierung des L^4Linux-Systems*. Diplomarbeit, Technische Universität Dresden, Institut für Systemarchitektur, Juli 2002. In German. Available from: `http://os.inf.tu-dresden.de/project/finished/finished.xml.en#peter-diplom`.

[76] ROBIN, JOHN SCOTT und CYNTHIA E. IRVINE: *Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor*. In: *Proceedings of the 9th USENIX Security Symposium*, Seiten 129–144, Denver, Colorado, USA, August 2000.

[77] RUBINI, ALESSANDRO and JONATHAN CORBET: *Linux Device Drivers, 2nd Edition*. O'Reilly & Associates, 2nd edition, June 2001.

[78] RUSHBY, JOHN: *Design and Verification of Secure Systems*. In *Proceedings of the 8th ACM Symposium on Operating System Principles (SOSP)*, pages 12–21, Pacific Grove, CA, December 1981.

[79] SAILER, REINER, ENRIQUILLO VALDEZ, TRENT JAEGER, RONALD PEREZ, LEENDERT V. DOORN, JOHN L. GRIFFIN, and STEFAN BERGER: *sHype: Secure Hypervisor Approach to Trusted Virtualized Systems*. Research report rc, IBM T. J. Watson Research Center, Yorktown Heights, NY, February 2005.

[80] SCHÖNBERG, S.: *Using PCI-Bus Systems in Real-Time Environments*. PhD thesis, TU Dresden, Fakultät Informatik, September 2002.

[81] SINGH, AMIT: *An Introduction to Virtualization*. kernelthread.com, January 2005. Available from: `http://www.kernelthread.com/publications/virtualization/`.

[82] SMITH, RICHARD E.: *Mandatory Protection for Internet Server Software*. In *Proceedings of the 12th Annual Computer Security Applications Conference*, San Diego, California, USA, December 1996.

[83] SPRUTH, WILHELM G.: *OS/390 Internet Services*, December 2004. Available from: `http://www-ti.informatik.uni-tuebingen.de/os390/`.

[84] SUGERMAN, JEREMY, GANESH VENKITACHALAM, and BENG-HONG LIM: *Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor*. In *Proceedings of the 2001 USENIX Annual Technical Conference, General Track*, pages 1–14, Boston, Massachusetts, USA, June 2001.

[85] SUN MICROSYSTEMS, 2550 Garcia Avenue, Mountain View, CA U.S.A. 94043, 1-800-681-8845: *microSPARC-IIep – User's Manual*, April 1997.

[86] SWIFT, MICHAEL M., BRIAN N. BERSHARD, and HENRY M. LEVY: *Improving the Reliability of Commodity Operating Systems*. In *Proceedings of the 19th ACM Symposium on Operating System Principles (SOSP)*, pages 207–222, Bolton Landing, NY, October 2003.

[87] TIERNEY, BRIAN L.: *Tcp tuning guide for distributed application on wide area networks*. ;login, pages 33–39, February 2001.

[88] UHLIG, V., U. DANNOWSKI, E. SKOGLUND, A. HAEBERLEN, and G. HEISER: *Performance of Address-Space Multiplexing on the Pentium*. Technical Report 2001-1, University of Karlsruhe, Dept. of Computer Science, System Architecture Group, 2001.

[89] UHLIG, VOLKMAR, JOSHUA LEVASSEUR, ESPEN SKOGLUND, and UWE DANNOWSKI: *Towards Scalable Multiprocessor Virtual Machines*. In *Proceedings of the 3rd Virtual Machine Research and Technology Symposium*, pages 43–56, San Jose, CA, USA, May 2004.

[90] UNIVERSITÄT KARLSRUHE, DEPT. OF COMPUTER SCIENCE, SYSTEM ARCHITECTURE GROUP: *The L4Ka::Pistachio Microkernel*, May 2003. White Paper.

[91] UNIVERSITÄT KARLSRUHE, DEPT. OF COMPUTER SCIENCE, SYSTEM ARCHITECTURE GROUP: *L4 eXperimental Kernel Reference Manual Version X.2*, December 2004.

[92] U.S. DEPARTMENT OF ENERGY: *CIAC Bulletins and Advisories*, January 2005. Available from: `http://ciac.llnl.gov/cgi-bin/index/bulletins`.

[93] VMWARE INC.: *Website*, 2005. Available from: `http://www.vmware.org`.

[94] WALDSPURGER, CARL A.: *Memory Resource Management in VMware ESX Server*. In *Proceedings of the 5th USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 181–194, Boston, MA, December 2002.

[95] WHEELER, DAVID A.: *More Than a Gigabuck: Estimating GNU/Linux's Size*, June 2002. Available from: `http://www.dwheeler.com/sloc/redhat71-v1/redhat71sloc.html`.

[96] WHITAKER, ANDREW, MARIANNE SHAW, and STEVEN D. GRIBBLE: *Scale and performance in the Denali isolation kernel*. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 195–209. USENIX Association, 2002.

[97] *CPU cache. From Wikipedia, the free encyclopedia*, December 2004. Available from: `http://en.wikipedia.org/wiki/CPU_cache`.

[98] *Data. From Wikipedia, the free encyclopedia*, December 2004. Available from: `http://en.wikipedia.org/wiki/Data`.

[99] YAMAMURA, SHUJI, AKIRA HIRAI, MITSURU SATO, MASO YAMAMOTO, AKIRA NARUSE, and KOUICHI KUMON: *Speeding Up Kernel Scheduler by Reducing Cache Misses — Effects of cache coloring for a task structure*. In *Proceedings of the FREENIX Track: USEXNIX Annual Technical Conference*, pages 275–285, Monterey, California, USA, June 2002.

[100] YODAIKEN, VICTOR: *The RTLinux Manifesto*. In *Proceedings of the 5th Linux Expo, Raleigh, NC*, March 1999.

[101] YODAIKEN, VICTOR and MICHAEL BARABANOV: *A Real-Time Linux*. Available from: `http://www.cse.ucsc.edu/~sbrandt/courses/Winter00/290S/rtlinux.pdf`.