Bachelorarbeit

Schritte zur Portierung des Fiasco Mikrokerns auf PowerPC

Matthias Lange
matthias.lange@matze-lange.de

17. November 2006

Technische Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur
Professur Betriebssysteme

Erklärung

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

Dresden, den 17. November 2006

Matthias Lange

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Mikrokernbetriebssysteme der zweiten Generation erleichtern die flexible Konstruktion von Betriebssystemen. Ein Hauptanwendungsfeld dafür sind sichere Systeme, denn mit Hilfe von Mikrokernen lassen sich sichere Anwendungen vollständig vor Standardbetriebssystemen und ihren Anwendungen schützen.

Zunehmend werden auch eingebettete Systeme als offene und erweiterbare Plattformen eingesetzt. In diesen Systemen ist der PowerPC ein weit verbreiteter Prozessor (zum Beispiel im Automobilbereich).

Fiasco [Hoh98] ist ein Mikrokern der zweiten Generation. Er wurde im Rahmen des DROPS-Projekts (**D**resden **R**ealtime **O**perating **S**ystem) an der Technischen Universität Dresden entwickelt. Er existiert bisher jedoch nur in Versionen für IA-32, IA-64, ARM, AMD64 und für die Unix-Systemrufschnittstelle.

Diese Arbeit beschäftigt sich mit der Portierung von Fiasco auf die PowerPC-Architektur und untersucht dabei die Auswirkungen auf die Fiasco-Implementierung.

1.2 Über diese Arbeit

Das folgende Kapitel beschäftigt sich zunächst mit den Grundlagen, die für das bessere Verstehen dieser Arbeit nötig sind. Das dritte Kapitel behandelt und erläutert die Aspekte und Schritte, die für die Portierung von Fiasco auf den PowerPC-Prozessor nötig sind. Schwerpunkt der Arbeit stellt das Kapitel 4 über die architekturneutrale Seitentabellenschnittstelle für Fiasco dar. Das Kapitel 5 geht auf die Implementierung der Schnittstelle für IA-32 ein und gibt einen Ausblick auf die Schritte, die für die Implementierung für den PowerPC notwendig sind. Den Abschluss bildet das Kapitel 7 mit Ausblicken auf zukünftige Arbeiten sowie mit einer kurzen Zusammenfassung.

1.3 Bezeichnungen

Für manche Begriffe lassen sich keine vernünftigen Entsprechungen im Deutschen finden. Diese Begriffe werden von mir in kursiver Schrift hervorgehoben, zum Beispiel *hashed*-Seitentabellen. Befehle und Quellcode werden in Schreibmaschinenschrift geschrieben, zum Beispiel `__main`.

1.3.1 Byte-Ordnung

Die PowerPC-Architektur ist eine *big-endian*-Architektur. In Abbildungen wird in Übereinstimmung mit [IBM03] das Byte 0 ganz links, gefolgt von Byte 1, ..., Byte n, dargestellt. Das höchstwertige Bit (=Bit 0) steht immer ganz links. Auf den ersten Blick ist es verwirrend, dass das höchstwertige Bit als Bit 0 bezeichnet wird. Um Konfusionen zu vermeiden, habe ich mich entschieden, die von IBM gewählte Bezeichnungsweise in dieser Arbeit zu übernehmen.

2 Grundlagen

Dieses Kapitel beschäftigt sich mit grundsätzlichen Konzepten für Betriebssysteme und mit den Grundlagen der PowerPC-Architektur.

2.1 Betriebssysteme

Die heute gebräuchlichen Betriebssysteme wie zum Beispiel Windows oder Linux basieren auf einem monolithischen Kern. Unter monolithisch versteht man, dass ein großer Teil der Funktionen eines Betriebssystems, zum Beispiel Gerätetreiber, in den Kern integriert ist. Diese Architektur ist gegenüber Fehlfunktionen einzelner Komponenten anfällig, da zum Beispiel ein Gerätetreiber wichtige Kern-Daten überschreiben und damit das ganze System zum Absturz bringen kann. Ein weiteres Problem stellt die Komplexität monolithischer Kerne dar. Durch ihre Größe sind sie anfälliger für Programmierfehler und außerdem schwieriger zu testen.

Im Gegensatz dazu gibt es Mikrokerne. Sie reduzieren die Funktionen des Kerns auf ein Mindestmaß, indem zum Beispiel Gerätetreiber aus dem Kern entfernt werden und als unprivilegierte Programme ausgeführt werden.

Ein weiterer Ansatz sind sog. Nanokerne. Diese eignen sich im Allgemeinen nicht für die Konstruktion von Betriebssystemen, sondern kommen eher bei der Virtualisierung von Hardware zum Einsatz, um die parallele Ausführung mehrerer Betriebssysteme zu ermöglichen. Die Firma Apple Computer setzte zum Beispiel beim Übergang von 68k- zu PowerPC-Prozessoren zunächst einen Nanokern ein. Er übersetzte die Interrupts des PowerPCs in ein Format, welches vom klassischen Mac OS verarbeitet werden konnte [Nan].

2.2 Mikrokerne

Ein Mikrokern ist ein minimaler Betriebssystemkern. Er stellt grundlegende Primitive (Adressräume, *Threads* und Inter-Prozesskommunikation (im folgenden IPC)) zur Verfügung, um wichtige Betriebssystemdienste zu konstruieren.

In der Vergangenheit gab es bereits einige Ansätze, diese Idee in die Praxis umzusetzen. Der bekannstete dürfte hier das Mach-Projekt der Carnegie Mellon University [BBB$^+$90] sein. Solche Projekte führten zunächst zu keinen Erfolgen, da die Ausführungsgeschwindigkeit dieser Systeme im Vergleich zu klassischen monolithischen Systemen zu gering war. Hauptgrund war die mangelnde IPC-Geschwindigkeit. Das resultierte zum Beispiel bei Mach aus einem sehr komplexen IPC-Modell, welches auch über ein umfangreiches Sicherheitsmodell verfügte.

Jochen Liedtke entwickelte mit L4 [Lie96] einen Mikrokern der zweiten Generation. Durch grundlegende Vereinfachung der IPC-Kommunikation wurde versucht, die Geschwindigkeitsprobleme zu lösen. Solche Kerne bieten daher einen sehr effizienten Weg für die IPC. Zusätzlich verfügen sie nur über zwei Abstraktionen (*Threads* und Adressräume) und versuchen, sämtliche Regeln (zum Beispiel Seitenersetzungsalgorithmus) aus dem Kern zu entfernen.

Der Fiasco-Mikrokern wurde in C++ geschrieben, und es gibt mittlerweile Varianten für IA-32, IA-64, AMD64 und ARM.

Weitere L4-Implementierungen sind:

L4/Pistachio Universität Karlsruhe (IA-32, IA-64, Alpha, AMD64, ARM, MIPS 64-bit, PowerPC 32-bit, PowerPC 64-bit) (siehe [Pis]);

L4/MIPS University of New South Wales (MIPS) (siehe [Hei01]).

2.3 Die PowerPC-Architektur

Der PowerPC (**P**erformance **o**ptimization **w**ith **e**nhanced **R**ISC **P**erformance **C**hip) ist eine 1991 durch ein Konsortium der Firmen Apple Computer, IBM und Motorola spezifizierte Prozessor-Architektur. Diese Prozessoren sollten den von Apple bisher verwendeten 68k-Prozessor von Motorola ersetzen. So flossen in die Entwicklung des PowerPC u.a. wesentliche Teile von IBMs POWER-Architektur und Motorolas eigentlich als Nachfolger der 68k-Linie vorgesehenen 88k-Prozessoren (hier insbesondere die Busschnittstelle) ein. 1994 kamen die ersten Rechner mit dem neuen Prozessor auf den Markt. Ursprünglich sollte er nur in Arbeitsplatzrechnern Verwendung finden. Inzwischen ist der PowerPC auch bei eingebetteten Systemen (zum Beispiel in den Digitalreceivern „d-box 2“) oder bei Supercomputern (Virginia Tech Xserve G5 Cluster) erfolgreich.

Die PowerPC-Architektur ist als 64-bit-Architektur spezifiziert, die über eine 32-bit-Untermenge verfügt. Daher muss man zwischen drei verschiedenen Arbeitsmodi unterscheiden:

- 64-bit-Implementierung und 64-bit-Modus - hierbei verfügt der Prozessor über 64-bit-Speicheradressierung und 64-bit-Ganzzahltypen;
- 32-bit-Implementierung - hier stehen nur 32 Bits für die Speicheradressierung und Ganzzahltypen zur Verfügung;
- 64-bit-Implementierung und 32-bit-Modus - dieser Arbeitsmodus ist ein Kompatibilitätsmodus für die 32-bit-Implementierung. Ein 64-bit-Prozessor kann durch das Setzen eines bestimmten Bits im *Machine State Register* (MSR) auf Null in den 32-bit-Modus umschalten[1].

[1] Dieser Modus ist für unpreviligierte Programme gedacht. Bei einem Wechsel in den Supervisor-Modus wird automatisch in den 64-bit-Modus geschaltet. Der Betriebssystemkern könnte jetzt wieder in den 32-bit-Modus zurückschalten, jedoch bedeutet es in der Praxis meist mehr Aufwand den Kern im 32-bit-Modus auszuführen, als den Kern für den 64-bit-Modus anzupassen.

In der vorliegenden Arbeit beziehe ich mich nur auf die 32-bit-Implementierung, da es zum Beispiel im Bereich der Adressumsetzung größere Unterschiede zwischen 64-bit- und 32-bit-Varianten gibt.

Der PowerPC basiert auf einer RISC-Architektur (**R**educed **I**nstruction **S**et **C**omputing) und verfügt daher über die typischen Eigenschaften, wie:

Load/Store - Nur die Befehle `load` und `store` greifen auf den Hauptspeicher zu.

Großer Registersatz - Um häufige (langsame) Speicherzugriffe und damit zusätzliche `load/store` Befehle zu vermeiden, werden Zwischenergebnisse zur Effizienzsteigerung in einem großen Registersatz (viele allgemeine Register) vorgehalten.

Regelmäßiger Befehlssatz - Die Befehle haben alle die gleiche Breite (32 Bit) und es gibt nur wenige Befehlsformate.

Erwähnenswert ist noch die Möglichkeit, den PowerPC-Prozessor im *big-endian*- oder im *little-endian*-Modus zu betreiben. Die Umschaltung zwischen den beiden Modi ist auch während des Programmablaufs möglich. IBMs letzte Generation von PowerPCs (PPC970 alias G5), verfügt nicht mehr über diese Möglichkeit und arbeitet nur noch im *big-endian*-Modus.

Die folgenden Abschnitte enthalten eine Zusammenfassung der für die Portierung relevanten Eigenschaften des PowerPCs. Eine vollständige Dokumentation zum PowerPC kann unter [IBM03] gefunden werden.

2.3.1 Speicherverwaltung

Die Hauptfunktion der *Memory Management Unit* (MMU) des PowerPC-Prozessors ist die Umsetzung von logischen in physische Adressen. Zwei Arten von Speicherzugriffen (in Anlehnung an die Harvard-Architektur) erfordern eine Adressumsetzung:

- Zugriff auf Programminstruktionen,
- Zugriff auf Daten durch `load/store` Instruktionen.

Dabei stützt sich die Adressumsetzung auf Mechanismen wie Segmentdeskriptoren und Seitentabellen.

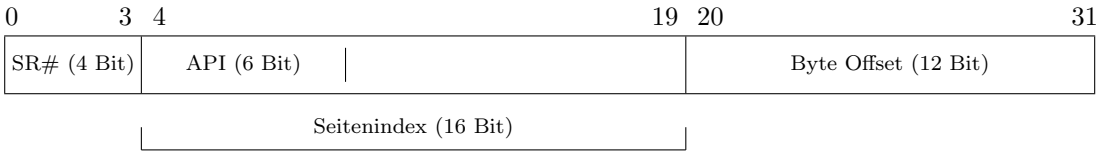
Es gibt noch eine sogenannte Block-Adressumsetzung, die hier jedoch nicht näher erläutert werden soll, da sie für die Portierung des Kerns keine Rolle spielt[2].

In Abbildung 2.1 ist das Layout einer logischen Adresse zu sehen. Die ersten vier Bit wählen einen von 16 Segmentdeskriptoren (siehe Abschnitt 2.3.1.1) aus. Das Seitenindexfeld[3] wird später bei der Berechnung der *Hash*-Funktion verwendet. Das *Byte-Offset* gibt den Versatz innerhalb einer 4-KB-Seite an. Die einzelnen Komponenten, die für die Adressumsetzung notwendig sind, werden in den folgenden Abschnitten beschrieben.

[2] Eine Ausnahme gibt es beim Kernstart. Hier werden, bevor die Seitentabellen initialisiert sind, die Code- und Daten-Sektionen zunächst über den BAT-Mechanismus gemappt.

[3] API steht für „Abbreviated Page Index“

Abbildung 2.1: Layout der logischen Adresse bei 32-bit-PowerPCs



2.3.1.1 Segmente

Der 4 Gigabyte große Adressraum des PowerPC ist in 16 Segemente zu je 256 Megabyte eingeteilt. Jedes dieser Segmente wird durch einen eigenen Segmentdeskriptor beschrieben, welcher zum Beispiel die Zugriffsrechte und eine Identifikationsnummer (im folgenden VSID) enthält. Bei 32-bit-PowerPCs sind für die Segementdeskriptoren auf dem Prozessor 16 Segmentregister vorhanden. Abbildung 2.2 zeigt das Layout eines solchen Registers und Tabelle 2.1 erläutert die verschiedenen Felder. Hervorzuheben ist hier die Möglichkeit, ein No-excute-Bit zu setzen, welches verhindert, dass Daten aus diesem Segment als Befehle interpretiert und dann ausgeführt werden.

Abbildung 2.2: Layout des Segmentregisters bei 32-bit-PowerPCs

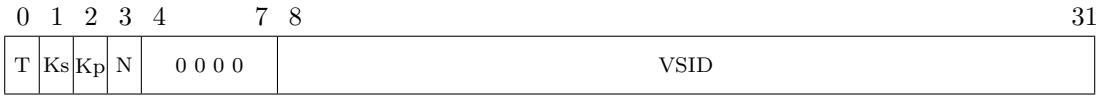


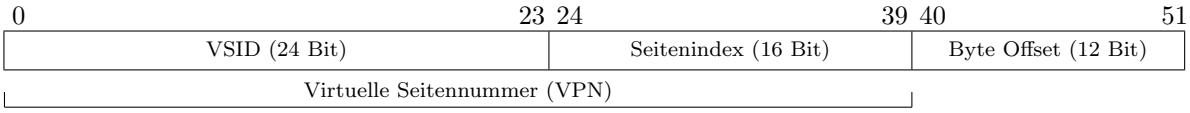
Tabelle 2.1: Bit-Einstellungen des Segmentregisters

Bits	Name	Beschreibung
0	T	T = 0 wählt dieses Registerlayout
1	Ks	Supervisor
2	Kp	User
3	N	no execute
4-7	-	reserviert
8-31	VSID	virtuelle Segment-ID

2.3.1.2 Virtuelle Adresse

Als virtuelle Adresse wird beim PowerPC eine Interims-Adresse bezeichnet, die 52 Bit lang ist. Diese wird aus der VSID (24 Bit), dem Seitenindex (16 Bit) und dem Byte-Offset (12 Bit) (beide der logischen Adresse entnommen) gebildet. VSID und Seitenindex formen zusammen die virtuelle Seitennummer (VPN). Das *Byte-Offset* gibt wie bei der logischen Adresse den Versatz innerhalb einer 4-KB-Seite an.

Abbildung 2.3: Layout der virtuellen Adresse

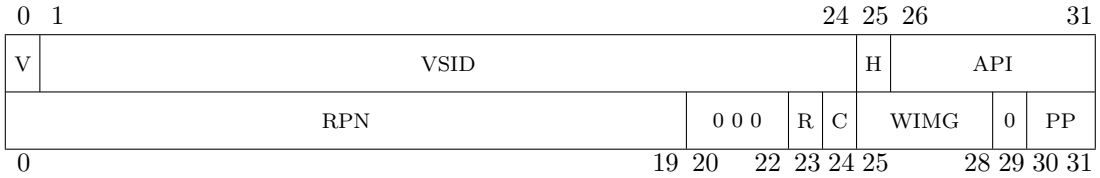


2.3.1.3 Seitentabellen

Der PowerPC verwendet sogenannte invertierte oder *hashed*-Seitentabellen. Im Gegensatz zu mehrstufigen Seitentabellen muss dabei die Größe nur ausreichen, um für alle physischen Kacheln ein *Mapping* zur Verfügung zu stellen (und nicht für den gesamten virtuellen Adressraum). Es wird also nur noch ein Eintrag je physischer Kachel angelegt. Der Nachteil der Suche in der Seitentabelle nach der richtigen Adressumsetzung wird durch das Vorschalten einer *Hash*-Funktion, die eine Gruppe von Einträgen auswählt, kompensiert.

Die Größe der Seitentabelle muss eine Zweierpotenz sein, und die Startadresse muss ein Vielfaches der Größe sein. Die Seitentabelle besteht aus einer Anzahl von Gruppen (PTEG - page table entry group) von Seitentabelleneinträgen (PTE - page table entry). Aus acht Einträgen wird eine PTEG geformt. Jeder Eintrag ist 8 Byte groß. In Abbildung 2.4 ist das Format eines PTEs dargestellt. Tabelle 2.2 erläutert die Bedeutung der einzelnen Bitfelder.

Abbildung 2.4: 8 Byte großer PTE



Jeder Seitentabelleneintrag kann sich potenziell in einer von zwei Gruppen befinden. Die erste wird primäre und die zweite sekundäre Gruppe genannt. Ein Seitentabelleneintrag befindet sich daher an einer von 16 möglichen Stellen. Zu beachten ist, dass die primäre Gruppe einer logischen Adresse die sekundäre einer anderen sein kann. Damit soll Kollisionen beim Einfügen von Einträgen in die Seitentabelle entgegengewirkt werden.

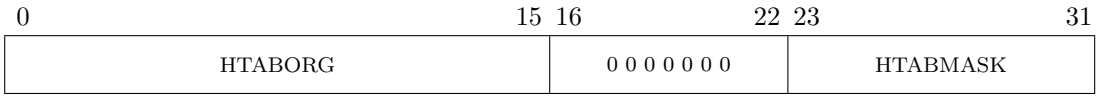
2.3.1.4 SDR1-Register

Das SDR1-Register enthält Informationen über die Seitentabelle. Zum einen werden darin die 16 höchstwertigen Bits der physischen Basisadresse der Seitentabelle sowie die Größe derselben gespeichert. Abbildung 2.5 zeigt das Format des SDR1-Registers. Das

Tabelle 2.2: Bedeutung der Bitfelder eines PTEs

Wort	Bit	Name	Beschreibung
0	0	V	Eintrag gültig (V = 1) oder ungültig (V = 0)
	1-24	VSID	virtuelle Segment-ID
	25	H	Bezeichner für die Hashfunktion (primär oder sekundär)
	26-31	API	Abbreviated Page Index
1	0-19	RPN	physische Seitennummer
	20-22	-	reserviert
	23	R	Verweis-Bit
	24	C	Geändert-Bit
	25-28	WIMG	Speicher/Cache Kontrollbits
	29	-	reserviert
	30-31	PP	Seitenschutzbits

Abbildung 2.5: Layout des SDR1-Registers bei 32-bit PowerPCs



Feld HTABORG enthält die 16 höchstwertigen Bits der physischen Adresse der Seitentabelle. Daher liegt der Anfang jeder Seitentabelle auf einer 64-KB-Grenze (2^{16} Byte). HTABMASK legt fest, wieviele zusätzliche Bits der *Hash*-Funktion, die im nächsten Abschnitt beschrieben wird, für die Berechnung der Adresse der PTEG in der Seitentabelle benutzt werden.

2.3.1.5 Hashfunktion

Die MMU benutzt zwei unterschiedliche *Hash*-Funktionen, um die physischen Adressen für die Seitentabellensuche zu generieren.

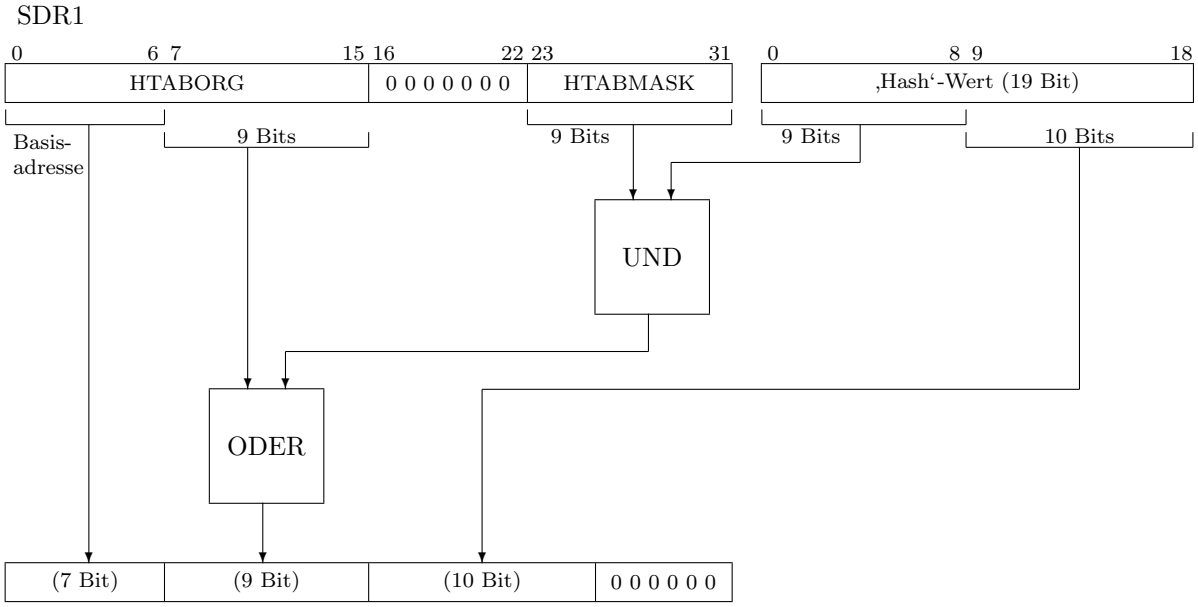
Für die Berechnung der primären *Hash*-Funktion wird zunächst das Seitenindexfeld der logischen Adresse mit 3 höchstwertigen Nullbits aufgefüllt. Dieser Wert wird dann mit den 19 niedrigstwertigen Bits des VSID-Felds des passenden Segmentregisters durch `XOR` verknüpft. Der Wert der sekundären *Hash*-Funktion ergibt sich als Einerkomplement des Wertes der primären Funktion.

2.3.1.6 Adressübersetzung

Um für eine logische Adresse den passenden Seitentabelleneintrag zu finden, werden die 10 niedrigstwertigen Bits der *Hash*-Funktion mit den 16 höchstwertigen Bits des HTABORG Felds des SDR1-Registers verbunden. Dabei legt HTABMASK fest, wieviele

niedrigstwertige Bits von HTABORG durch die nächst höchstwertigen Bits des *Hash*-Werts ersetzt werden. Die sechs niedrigstwertigen Bits werden mit Nullen gefüllt (um beim ersten Seitentabelleneintrag der Gruppe zu beginnen, 2^6 Byte = 64 Byte = Größe einer PTEG). Damit ergibt sich die physische Adresse der primären PTEG und evtl. auch der sekundären.

Abbildung 2.6: Berechnung der physischen Adresse einer PTEG



Jetzt wird jeder Eintrag der Gruppe mit der virtuellen Adresse verglichen. Für ein erfolgreiches *Matching* müssen die folgenden Bedingungen erfüllt sein:

- PTE[H] = 0 für primäre PTEG; PTE[H] = 1 für sekundäre PTEG
- PTE[V] = 1 (gültiger Eintrag)
- PTE[VSID] = VA[0-23]
- PTE[API] = VA[24-29]

Wird dabei kein Eintrag gefunden, wird ein Seitenfehler ausgelöst, der vom Betriebssystem behandelt werden muss. Werden mehrere Einträge gefunden, so werden weitere Felder verglichen, bis der korrekte (oder kein) Eintrag gefunden wird.

2.3.2 Ausnahmen

An dieser Stelle fasse ich der Einfachheit halber Software- und Hardware-Interrupts unter dem Begriff „Ausnahme“ zusammen. Ausnahmen sind Ereignisse, die das Übertragen der Kontrolle über den Prozessor an eine Ausnahmebehandlungsroutine auslösen. Außerdem

bewirken sie das Umschalten des Berechtigungsstufe des Prozessors in den Supervisor-Modus.

Damit nach Behandlung der Ausnahme die Ausführung normal weiterlaufen kann, muss der aktuelle Zustand des Prozessors vor dem Auftreten der Ausnahme gesichert werden. Wurde die Ausnahme behandelt, wird bei der Rückkehr der alte Zustand wiederhergestellt.

Ausnahmen lassen sich beim PowerPC in vier Klassen einteilen. In Tabelle 2.3 sind sie aufgelistet. Asynchrone Ausnahmen treten ohne Zusammenhang mit der aktuell aus-

Tabelle 2.3: PowerPC Ausnahmeklassen

Typ	**Ausnahme**
asynchron/nicht maskierbar	Machine Check, System Reset
asynchron/maskierbar	externe Ausnahmen, Dekrementierer (=Timer)
synchron/präzise	befehlsbedingte Ausnahme
synchron/unpräzise	befehlsbedingte unpräzise Ausnahme (Fließkommaausnahme)

geführten Instruktion auf (zum Beispiel Timer-Interrupt), während synchrone Ausnahmen (zum Beispiel Seitenfehler) durch spezielle Instruktionen (zum Beispiel `load/store`) ausgelöst werden. Maskierbar bedeutet in diesem Fall, dass sich diese Klasse von Ausnahmen durch das Setzen eines Bits im MSR bis zum Wiedereinschalten unterdrücken lässt. In dieser Zeit auftretende Ausnahmen werden dabei nicht verworfen, sondern nur zurückgehalten.

2.3.2.1 Ausnahmebehandlung

Die Ausnahmebehandlungsroutinen werden beim PowerPC beginnend vom Offset 0x100 bis 0x2FF abgelegt. Eine Übersicht der Ausnahmen und ihrer Offsets findet sich in [IBM03] im Kapitel 6.1. Dabei stehen für jede Routine 256 Byte (=64 Instruktionen) zur Verfügung.

Tritt eine Ausnahme auf, wird vom Prozessor, nachdem festgestellt wurde ob die Ausnahme zugelassen ist, folgendes ausgeführt:

1. In das *Status Save/Restore Register 0* (SRR0) wird die zur Ausnahme gehörende Instruktion der Ausnahmebehandlungsroutine geladen.

2. In das *Status Save/Restore Register 1* (SRR1) wird der Wert des MSR kopiert.

3. Das MSR wird entsprechend der Ausnahme geladen. Dabei wird zum Beispiel die Adressumsetzung abgeschaltet. Instruktionen die jetzt ausgeführt werden, müssen physische Adressen verwenden.

4. Im MSR wird das *Recoverable Interrupt Bit* auf 0 gesetzt. Das bedeutet, dass die Ausnahmebehandlungsroutine im „verletzbaren Bereich“ arbeitet und nicht mehr

zurückkehren kann, falls eine neue Ausnahme auftritt. An dieser Stelle können jedoch nur Ausnahmen vom Typ „System Reset“ und „Machine Check“ auftreten. Diese treten bei schweren Systemfehlern auf.

5. Die Ausführung setzt an der Offsetadresse fort, die durch den Ausnahmetyp bestimmt ist. Zunächst muss jetzt von der Software der Zustand von SRR0 und SRR1 gesichert werden und der *Stackpointer* aktualisiert werden. Dann kann das *Recoverable Interrupt Bit* wieder auf 1 gesetzt werden.

6. Wenn die Behandlung der Ausnahme abgeschlossen wurde, dann muss das SRR1 wieder mit dem alten Wert geladen werden. Die `rfi`-Instruktion kopiert die Bits aus SRR1 zurück in das MSR. Anschließend kehrt die Steuerung wieder an die Stelle zurück, an der sie vor der Ausnahme war.

2.4 Plattform

Computerplattformen werden durch den verwendeten Prozessor, den internen Datenbus und weitere Komponenten, die eng mit dem Prozessor zusammenarbeiten, charakterisiert. Für die Entwicklung eines Mikrokerns spielt neben dem Prozessor nur noch der Interrupt-Controller eine Rolle, da sonst keine weiteren Gerätetreiber in den Kern integriert werden sollen.

Jedoch sind ein paar Aspekte der verwendeten Plattform von Interesse, wenn es zum Beispiel um die Startsequenz und den zu verwendenden Bootloader geht. Der nächste Abschnitt greift zwei Teile der verwendeten Entwicklungsplattform auf und erläutert diese näher.

2.4.1 Open Firmware

Die *Open Firmware* ist eine hardwareunabhängige Firmware, die bereits 1988 von der Firma Sun Microsystems unter dem Namen „Open Boot“ für ihre SPARC Workstations entwickelt wurde. Später setzte u.a. die Firma Apple *Open Firmware* in ihren Macintosh-Rechnern ein. Möglich wurde das durch den IEEE-1275-Standard, in welchem die *Open Firmware* spezifiziert wurde. Der Standard wurde mittlerweile von der IEEE zurückgezogen, da er von der Open Firmware Working Group nicht erneut bestätigt wurde. Daher sind in IEEE-Dokumenten keine Informationen mehr zur *Open Firmware* erhältlich. Unter [SUN] sind jedoch einige notwendige Informationen zu finden.

Auf die *Open Firmware* kann über eine Forth-basierte Kommandozeilenumgebung zugegriffen werden. Sie verfügt über einen leistungsstarken Forth-Interpreter, der es zum Beispiel ermöglicht, das Problem der Türme von Hanoi (siehe [Sin]) mit Forth zu formulieren und zu lösen. Die *Open Firmware* bietet ein *Client-Interface* (CI) an, über das das Betriebssystem Dienste aufrufen und nutzen kann. Mit Hilfe des CI habe ich die Boot-Konsole im Rahmen dieser Arbeit implementiert.

Eine wichtige Funktion der *Open Firmware* ist die, dass sie ELF-Binärdateien direkt laden und starten kann. Jedoch kann sie nur jeweils eine Datei bzw. ein Image

laden und starten. Außerdem erstellt die *Open Firmware* beim Starten einen kompletten Gerätebaum. Er kann durchlaufen werden, um Informationen über die Hardware abzufragen (zum Beispiel Prozessor- und Busgeschwindigkeit).

2.4.2 PowerPC-Emulator

Bei der Kern-Entwicklung spart ein Simulator/Emulator viel Arbeit. Der Kern kann direkt vom Emulator ausgeführt und getestet werden und muss nicht jedes Mal aufwändig in eine Testumgebung integriert und dort gestartet werden. Außerdem ist die Anbindung an einen externen Debugger leichter möglich.

Für den PowerPC existieren drei bekannte Emulatoren: Qemu, PearPC und PSIM. Sie unterscheiden sich in ihrer Ausführungsgeschwindigkeit und im Umfang der implementierten Computerplattform. Abschnitt 5.1.2 geht näher auf die Auswahl des geeigneten Emulators ein.

3 Schritte zur Portierung

Diese Arbeit untersucht die Auswirkungen der Portierung von Fiasco (auf den PowerPC-Prozessor) auf die Struktur und die Implementierung des Kerns. Der PowerPC-Prozessor arbeitet im *big-endian*-Modus. Das kann zu Problemen mit der Definition von bestimmten Datenstrukturen führen, da IA-32 eine *little-endian*-Architektur ist.

Das folgende Kapitel erläutert die Schritte, die für eine erfolgreiche Portierung notwendig sind.

3.1 Allgemeine Datentypen

Bei der Definition von Datentypen gibt es keine Probleme. Die Portierung setzt auf der L4/V4-Schnittstelle auf. Eine Eigenschaft dieser Schnittstelle ist die prozessorunabhängige Definition von Datentypen (siehe [Tea05]).

Jedoch kann es bei der Definition von Konstanten durch `#define`-Anweisungen zu *Endian*-Problemen kommen. Das ist zum Beispiel bei der Definition der *Kernel-Info-Page-magic number* zu sehen. Die wurde bisher durch die folgende Definition festgelegt:

```
#define L4_KERNEL_INFO_MAGIC (0x4BE6344CL)
```

Das höchstwertige Byte (0x4C, entspricht dem Buchstaben L im ASCII-Code) steht hier rechts. Da aber bei einer *big-endian*-Architektur das höchstwertige Byte links stehen muss, würde die Interpretation des Werts beim PowerPC nicht zu der Zeichenkette „$L4\mu K$“, sondern zu der Zeichenkette „$K\mu 4L$“ führen.

3.2 Speicher

3.2.1 Virtuelle Segment-ID

Die im Abschnitt 2.3.1.1 besprochenen Segmente werden in den Segmentregistern des Prozessors über die VSID beschrieben. In [IBM03] ist nicht näher spezifiziert, wie diese ID aussehen muss. Sie muss lediglich für jeden Adressraum eindeutig sein.

In der PowerPC-Variante des Mach-Kerns wird für die Berechnung der VSID die Adressraum-ID verwendet, welche innerhalb des Systems eindeutig festgelegt ist. Die VSID wird nach folgender Formel berechnet

$$VSID = (currSID * incrVSID)\&(maxAdrSp - 1) \tag{3.1}$$

currSID - Nummer des Segements, für das die VSID berechnet werden soll.

incrVSID - Wird für jeden neuen Adressraum inkrementiert.

maxAdrSp - Konstante, die die maximale Anzahl an Adressräumen angibt. Sie ist bei Mach mit 16384 definiert.

Wegen der Freiheit bei der Wahl der VSID sollte man natürlich eine für die Implementierung günstige Festlegung treffen. Wie das für die PowerPC-Variante von Fiasco aussieht, erläutere ich in in Abschnitt 5.2.2.2, da hier noch Überlegungen aus dem Kapitel über die architekturneutrale Seitentabellenschnittstelle einfließen.

3.2.2 Adressraumlayout

Die ersten drei Gigabyte des virtuellen Adressraums werden für Benutzerprogramme reserviert. Das oberste Gigabyte ist für den Kern reserviert, wobei Benutzerprogramme zum Beispiel durch Systemrufe darauf zugreifen können.

Die ersten 12 Segmentregister zeigen auf einen Benutzeradressraum, während die verbleibenden 4 Segmentregister auf den Kernadressraum zeigen. Das hat den Vorteil, dass bei einem Kontextwechsel sofort der Kernadressraum zur Verfügung steht.

Teilt man das Gigabyte für den Kern in seine 4 Segmente auf, so ist es sinnvoll, ein Segment für die IPC-Fenster zu verwenden. Dieser Bereich kann dann einfach durch Änderung des Segmentregisters in einen anderen Adressraum eingeblendet werden.

3.2.3 Mapping-Datenbank

Fiasco unterstützt die Möglichkeit, Adressräume rekursiv aufzubauen (siehe [Lie96]). Der Kern muss sich dazu in einer Datenbank die hierarchischen Abhängigkeiten von eingeblendeten Seiten merken. Dies ist nötig, damit eine Seite später aus allen Adressräumen, in denen sie eingeblendet ist, wieder entzogen werden kann.

Die Restriktion der *Mapping*-Datenbank von Fiasco auf eine Größe von 4 GB für virtuellen Speicher stellt kein Problem dar, da die Portierung für einen 32-bit-Prozessor vorgenommen wird.

3.3 Kerneintritte/-austritte

Um den Kern vor Zugriffen von Programmen (und Programme vor gegenseitigem Zugriff) zu schützen, wird der Kern in einem privilegierten Modus (Supervisor-Modus) des Prozessors ausgeführt. Normale Programme laufen dagegen in einem eingeschränkten Modus (Benutzer-Modus). So lässt sich der privilegierte Kerncode vor bösartigen oder fehlerhaften Programmen schützen.

Umschaltungen vom Benutzer- in den Supervisor-Modus geschehen durch einen Systemruf eines Programms oder durch eine auftretende Ausnahme.

3.3.1 Ausnahmen

Tritt eine Ausnahme auf, schaltet der Prozessor automatisch in den Supervisor-Modus um. Im Gegensatz zu IA-32, wo bei einer Ausnahme eine Funktion entsprechend der Ausnahmenummer aus der *Interrupt Descriptor Table* (IDT) aufgerufen wird, wird beim PowerPC eine Funktion an einem festen Offset (entsprechend der Ausnahme) aufgerufen.

Diese Funktion wird in Assembler-Code geschrieben und muss zunächst den gesamten Zustand sichern. Nachdem das geschehen ist, wird an dieser Stelle eine C-Funktion aufgerufen, da an jedem Offset maximal 256 Byte (= 64 Instruktionen) für die Ausnahmebehandlung zur Verfügung stehen (siehe auch 2.3.2).

Wie in 2.3.2.1 beschrieben, wird beim Auftreten einer Ausnahme die Adressumsetzung abgeschaltet. Es ist jedoch wünschenswert, auch im Kern mit virtueller Adressierung zu arbeiten. Daher ist es üblich, nach dem Sichern des Zustandes mittels der `rfi`-Instruktion in den sogenannten *translated*-Supervisor-Modus umzuschalten. Dort wird die Ausnahme dann behandelt. Danach wird wieder in den unpreviligierten Modus zurückgekehrt.

3.3.2 Systemrufe

Benutzerprogramme müssen für bestimmte Aufgaben die Funktionalität des Betriebssystems nutzen. Dies ist zum Beispiel dann der Fall, wenn sie einen neuen *Thread* erzeugen müssen oder mit einem anderen *Thread* kommunizieren wollen. Das Betriebssystem stellt seine Funktionalität über eine Systemrufschnittstelle bereit.

Erfolgt durch ein Programm ein Systemruf, muss die `sc`-Instruktion ausgeführt werden. Bei L4/V4 jedoch sind in der *kernel info page* (im folgenden KIP) Eintrittspunkte für die Systemrufe festgelegt, die es möglich machen, die Art, wie der Kern „betreten“ werden soll, später zu ändern, ohne dass das Programm neu kompiliert werden muss.

Wird dann in diesem Eintrittspunkt die `sc`-Instruktion ausgeführt, führt das zu einer *System Call Exception* und damit zum Umschalten in den Supervisor-Modus. Damit der Ruf erfolgreich ausgeführt werden kann, muss im Register R2 der Zeiger auf den UTCB abgelegt sein (siehe [Tea05], Seite 120).

3.4 Kern-Debugger

Der in Fiasco integrierte Kern-Debugger (JDB) hängt stark von der konkreten Implementierung des Kerns ab, da die JDB-Klassen durch das C++-Schlüsselwort `friend` an die Kern-Klassen gebunden werden. Der Kern-Debugger ist stark architekturabhängig und selbst für Fiasco-V4 für IA-32 nur teilweise verfügbar. Da der Kern ohne JDB kompiliert werden kann, wurde der JDB zunächst nicht für den PowerPC-Prozessor angepasst.

3.5 Open-Firmware-Konsole

Der Kern arbeitet mit der *Open Firmware* zusammen, um eine Ein- und Ausgabekonsole (insbesondere für die Fehlersuche) bereitzustellen. Da alle Parameter und Rückgabewerte bei einem Aufruf des *Open Firmware*-CI über den Stack (Stack = Kellerspeicher = Stapelspeicher) übergeben werden müssen (und nicht als Referenz), braucht man dafür einen recht großen Speicherbereich. Daher erfolgen Aufrufe der *Open Firmware* nicht über den Kern-TCB-Stack, sondern über einen separaten Stack.

Um Aufrufe des *Open Firmware*-CIs zu ermöglichen, muss der Kern in den Adressraum der *Open Firmware* umschalten. Der Bootloader muss dafür sorgen, dass der von der *Open Firmware* verwendete Speicherbereich vor Zerstörung geschützt wird. Dazu trägt er vor dem Kernstart diesen Speicher als reserviert in die KIP ein.

Tritt während der Ausführung der *Open Firmware* eine Ausnahme auf (zum Beispiel Seitenfehler), so schlägt der Aufruf fehl. Es muss daher sichergestellt werden, dass der benötigte Speicher eingeblendet ist.

4 Design einer architekturneutralen Seitentabellenschnittstelle für Fiasco

In der Diplomarbeit von Alexander Warg [War03] wurde die allgemeine Portabilität von Fiasco untersucht. Dabei wurde die Softwarestruktur des Kerns so verändert, dass er besser auf neue Prozessorarchitekturen übertragen werden kann.

Bisher wurde jedoch kein spezielles Augenmerk auf eine einheitliche Seitentabellenschnittstelle gelegt. Um der großen Anzahl von Seitentabellenschnittstellen in Fiasco nicht für PowerPC eine weitere hinzuzufügen, habe ich eine architekturneutrale Seitentabellenschnittstelle entwickelt, die mit unterschiedlichen Seitentabellenformaten umgehen kann.

4.1 Zielstellung

Seitentabellen werden in einem Betriebssystemkern bei Seitenein- und -ausblendungen sowie bei der Reservierung von virtuellem Speicher modifiziert. Das geschieht zum Beispiel beim Erzeugen oder Zerstören von Prozessen, beim Ändern von Speicherzugriffsrechten oder in einem Mikrokern bei IPC-Operationen.

Bislang ist es bei Fiasco für IA-32 so, dass die Seitentabelle direkt modifiziert wird. Die Implementierungen für AMD64 und ARM bieten jeweils eine eigene proprietäre Schnittstelle. Dieser Umstand erschwert die Portabilität und die Wartung. Außerdem lassen sich neue Funktionen nur schwer integrieren, da jede Variante des Fiasco-Kerns eine spezielle Implementierung benötigt.

Durch die unterschiedlichen Konzepte für Seitentabellen bei verschiedenen Prozessoren ergeben sich für eine kerninterne Seitentabellenschnittstelle mehrere Anforderungen:

- Unterstützung mehrerer Seitengrößen. Die IA-32-Architektur verwendet 4-KB- und 4-MB-Seiten und die IA-64-Architektur ermöglicht sogar die Verwendung von neun verschiedenen Seitengrößen (4 KB bis 256 MB).

- Unterstützung mehrstufiger Seitentabellen. Die AMD64-Architektur arbeitet mit 4-stufigen Seitentabellen (siehe [Fre05]).

- Unterstützung für inverse Seitentabellen (zum Beispiel PowerPC mit *hashed*-Seitentabellen).

Ziel ist es, alle Seitentabellenmodifikationen im Fiasco-Kern unter dieser Schnittstelle zusammenzufassen und eine Abstraktionsschicht für die MMU-Hardware einer Plattform

anzubieten. Durch die Einführung einer Schnittstelle soll die Komplexität des Kerns reduziert werden, der Aufwand für die Wartung des Kerncodes vereinfacht und die Portierung auf neue Plattformen erleichtert werden.

4.2 Ansätze

Der Mach-Mikrokern und Linux wurden schon auf viele Prozessortypen portiert. Eine solche Vielfalt ist ohne gute kerninterne Abstraktionsschichten nicht denkbar. Durch ihre Verbreitung haben diese Schnittstellen ihre Robustheit und Eignung bewiesen. Daher ist ein Blick auf die interne Seitentabellenschnittstelle der beiden Betriebssystemkerne interessant.

4.2.1 Mach

Mach ist ein gutes Beispiel für die klare Trennung der Speicherverwaltung in einen architekturabhängigen (*pmap*-Ebene) und einen architekturunabhängigen Teil. Bei Mach wird der Adressraum in Speicherobjekte zerlegt, für die jeweils dieselben Eigenschaften (zum Beispiel Zugriffsrechte) gelten. Die Größe eines solches Speicherobjekts ist unabhängig von der physischen Seitengröße der zugrundeliegenden Architektur.

Die *pmap*-Ebene übernimmt die Adressumsetzung von virtuellen in physische Adressen und kapselt die MMU-Hardware. Hier gibt es Funktionen zum Erzeugen einer Übersetzung (`pmap_map`), zum Einfügen (`pmap_enter`) und Entfernen (`pmap_remove`) von Seiten und zum Ändern von Seitenzugriffsrechten (`pmap_protect`).

4.2.2 Linux

Im Gegensatz zu vielen anderen Betriebssystemen verfolgt Linux den Weg einer dreistufigen Seitentabelle für den architekturunabhängigen Teil, auch wenn die konkrete Architektur diese nicht unterstützt. Die oberste Ebene in dieser Struktur ist die *Page Global Directory*, gefolgt von der *Page Middle Directory*. Die unterste Ebene sind die *Page Table Entries*.

Die IA32-Architektur verwendet aber nur eine zweistufige Seitentabelle. Um diese auf das 3-stufige System abzubilden, wird die Größe der mittleren Stufe (*Page Middle Directory*) auf die eines Eintrags definiert und bei der Kompilierung mit der ersten Stufe (*Page Global Directory*) zusammengefasst.

Seit der Version 2.6.11 arbeitet der Linux-Kern intern mit einer 4-stufigen Seitentabelle. Damit ist es möglich, den vollen virtuellen Adressraum von 64-bit-Architekturen zu erschließen. Auf der PowerPC-Architektur wird die *hashed*-Seitentabelle vom Linux-Kern als *Cache* für PTEs genutzt. Intern wird eine zweistufige Seitentabelle verwendet.

4.3 Speichermanagement des Fiasco-Kerns

Die Seitentabelle arbeitet eng mit dem Speichermanagement des Kerns zusammen. Die Seitentabellenklasse muss Speicher für die Seitentabelle reservieren. Außerdem muss der Kern Seiten an spezifische virtuelle Adressen *mappen* können.

In Fiasco wird der Kernspeicher durch die Klasse `Kmem` abstrahiert. Diese Klasse erzeugt eine Master-Seitentabelle für den Kernadressraum, die später von allen Prozessen genutzt wird, indem die Einträge dieser Seitentabelle in die Seitentabelle des Prozesses kopiert werden. Die Klasse `Space` repräsentiert einen Prozess mit seinem Adressraum.

Zusätzlich gibt es die Klasse `Kmem_alloc`, die für die Reservierung von Kernspeicher verantwortlich ist.

Damit Prozesse Speicher reservieren können, gibt es in der Klasse `Vmem_alloc` die beiden Funktionen `page_alloc` und `local_alloc`. Sie sind für die Reservierung von kerninternem Speicher nötig. Die erste ermöglicht es dem Prozess, sich vom Kern Speicher an eine von ihm spezifizierte virtuelle Adresse einblenden zu lassen. Ein Beispiel ist hier das Reservieren von ausgenulltem Speicher für den UTCB von *Sigma0*. Mit der zweiten Funktion kann ein Prozess sich eigene physische Seiten in seinen Adressraum einblenden. Sie wird beim Erzeugen des *Rootservers* zum Einblenden des UTCBs verwendet. Außerdem werden diese Funktionen von den *Slab*-Allokatoren verwendet.

4.3.1 Besondere Anforderungen

Besondere Anforderungen an die Schnittstelle ergeben sich aus der Struktur und Funktionsweise von Seitentabellen auf der PowerPC-Architektur. Im Unterschied zur IA-32-Architektur gibt es für alle Adressräume eine gemeinsame Seitentabelle. Gültige Seitentabelleneinträge für einen Adressraum werden über die VSID ermittelt (siehe auch 2.3.1.6). Die VSID übernimmt dabei die Funktion einer Adressraum-ID (ASID) mit Segmentgranularität.

Der Fiasco-Kern geht davon aus, dass Einträge, die in der *Mapping*-Datenbank vorhanden sind, immer über die Seitentabelle zugreifbar sind. Um zum Beispiel den Wurzelknoten einer Seite in der *Mapping*-Datenbank zu finden, wird die physische Adresse dieser Seite benötigt. Aufgrund dieser Annahme und der geschilderten Seitentabellenarchitektur des PowerPC-Prozessors können zwei Arten von Problemen entstehen:

1. In der Seitentabelle muss ein existierender Eintrag verdrängt werden, da die primäre und sekundäre PTEG schon mit jeweils acht gültigen Einträgen gefüllt sind.

2. Es wird auf eine Seite zugegriffen, die in der *Mapping*-Datenbank eingetragen ist, aber aus der Seitentabelle verdrängt wurde. Oder ein Eintrag einer nicht in der Seitentabelle vorhandenen Seite soll aus der *Mapping*-Datenbank ausgetragen werden.

Es gibt zwei Möglichkeiten, diese Probleme zu lösen. Die erste wäre die Einführung von Speicherobjekten als neue kern-interne Abstraktion von virtuellem Speicher. Die L4-Mikrokerne wurden mit dem Ziel entwickelt, die Seitenverwaltung außerhalb des Kerns

durchzuführen. Dafür wurden die sog. *Flexpages* eingeführt, die Speicherobjekte außerhalb des Kerns darstellen. Die hier gemeinten Speicherobjekte können eine beliebige (ein Vielfaches der kleinsten Seitengröße) Größe haben. Die Seitentabellenschnittstelle verwaltet diese Speicherobjekte in einer eigenen Struktur. Zum Beispiel lassen sich größere zusammenhängende *Mappings* als Speicherobjekte betrachten. Die Einführung einer solchen Abstraktion bringt allerdings grundlegende Änderungen an den Kern-Systemen mit sich. Zunächst benötigt man eine Infrastruktur für die Verwaltung der Speicherobjekte. Die *Mapping*-Datenbank müsste für die Verwendung von Speicherobjekten angepasst werden. Außerdem müssen alle Systeme im Kern, welche Speicher reservieren oder freigeben, an die neue Semantik angepasst werden.

Eine einfachere Lösung ist es, die aktuelle Semantik von Speicherseiten im Kern beizubehalten. Statt dessen wird bei Architekturen, auf denen die o.g. Probleme auftreten können, zusätzlich eine Datenstruktur verwaltet, in der Adressumsetzungen eines Adressraums gespeichert werden. Diese „Schatten"-Seitentabelle ist unabhängig von der echten Seitentabelle. Man kann jederzeit zu einer gegebenen virtuellen Adresse die physische Adresse ermitteln, auch wenn zu diesem Zeitpunkt kein Eintrag in der Seitentabelle dafür existiert.

Grundsätzlich sind für die Datenstruktur zwei Ansätze möglich:

1. Sämtliche Adressumsetzungen eines Adressraums werden eingetragen oder

2. es werden nur die aus der Seitentabelle verdrängten Einträge verwaltet.

Der erste Ansatz ist einfacher zu implementieren. Wenn ein Eintrag in die Seitentabelle eingefügt wird, wird dieser Eintrag gleichzeitig in die zusätzliche Datenstruktur eingetragen. Wird jetzt ein Eintrag dieses Adressraums von einem Eintrag eines anderen Adressraums aus der Seitentabelle verdrängt, müssen keine weiteren Maßnahmen ergriffen werden, den verdrängten Eintrag in die „Schatten"-Seitentabelle des entsprechenden Adressraums einzutragen. Bei dieser Variante ist allerdings der Speicherbedarf größer, da für alle Adressumsetzungen ein Eintrag erzeugt werden muss.

Diesen Kritikpunkt vermeidet der zweite Ansatz, in dem in der „Schatten"-Seitentabelle nur aus der Seitentabelle verdrängte Einträge eingetragen werden. Das reduziert den Speicherverbrauch. Allerdings ist hier der Verwaltungsaufwand höher, denn wenn ein Eintrag aus der Seitentabelle verdrängt wird, muss der zu diesem Eintrag gehörende Adressraum ermittelt werden und der Eintrag in seine „Schatten"-Seitentabelle eingetragen werden.

Durch eine solche Struktur wird die hardwareseitig vorgebene Seitentabelle zu einer Art *Cache*-Speicher für Seitentabelleneinträge. Allerdings muss für diese Architektur die Behandlungsroutine für Seitenfehler erweitert werden. Im Falle eines Seitenfehlers muss zunächst in der „Schatten"-Seitentabelle nachgeschaut werden, ob eine gültige Adressumsetzung existiert. Ist das der Fall, dann wird diese wieder in die Seitentabelle eingetragen.

4.4 Annahmen

Die Implementierung des Kerns ist nicht frei von Annahmen über das Verhalten von Funktionen, die die Seitentabelle modifizieren. Diese Annahmen müssen entweder in der neuen Schnittstelle berücksichtigt oder aber an den entsprechenden Stellen im Kern beseitigt werden. Die unterschiedlichen Annahmen lassen sich in explizite und implizite Annahmen teilen.

4.4.1 Explizite Annahmen

Explizite Annahmen drücken sich beim Aufruf einer Funktion schon durch den Funktionsnamen aus. Als Beispiel sei hier die Funktion `page_alloc` aus der Klasse `Vmem_alloc` genannt. Beim Aufruf dieser Funktion (im IA-32-Kern) geht der Aufrufer explizit davon aus, dass hier eine 4-KB-Seite an die von ihm angegebene virtuelle Adresse eingeblendet wird.

Explizite Annahmen lassen sich nicht ohne weiteres beseitigen. Für das genannte Beispiel müsste man zum Beispiel den Aufruf der Funktion `page_alloc` um einen Größen-Parameter erweitern. Das zieht unter Umständen umfangreiche Änderungen im gesamten Kern nach sich. Eine Alternative ist, die explizite Annahme bestehen zu lassen und bei der Implementierung der diskutierten Funktion die Annahme zu erfüllen. Für das Beispiel würde das bedeuten, dass man zum Beispiel immer Seiten der kleinsten verfügbaren Seitengröße reserviert und einblendet.

4.4.2 Implizite Annahmen

Implizite Annahmen im Fiasco-Kern sind dadurch gekennzeichnet, dass der Aufrufer einer Funktion implizit bestimmte Annahmen trifft. Zum Beispiel wird beim Einfügen eines Seitentabelleneintrags davon ausgegangen, dass bestimmte Seitenattribute (`Referenced`, `Valid`) von der aufgerufenen Funktion gesetzt werden.

Implizite Annahmen lassen sich relativ einfach beseitigen oder in eine Funktion übernehmen. Für das Beispiel bedeutet das, dass man entweder alle gewünschten Seitenattribute an die Einfügen-Funktion übergibt oder aber die Einfügen-Funktion so implementiert, dass sie bestimmte Seitenattribute zusätzlich zu den per Parameter spezifizierten setzt.

4.5 Design

4.5.1 Softwarestruktur

Fiasco ist ein Mikrokern, dessen Softwarestruktur nach objektorientierten Aspekten entwickelt wurde. Das soll sich auch in der Seitentabellenschnittstelle fortsetzen.

Aus objektorientierter Betrachtungsweise setzt sich eine Seitentabelle aus zwei Objekten zusammen: Seitentabelle und Seitentabelleneintrag. Eine Seitentabelle besteht aus beliebig vielen, jedoch mindestens einem Seitentabelleneintrag, weil mindestens der

Speicher des aktuell ausgeführten Programms in der Seitentabelle abgebildet sein muss. Die Seitentabellenklasse muss über Funktionen zum

- Einfügen, Löschen, Suchen und Ändern von Seitentabelleneinträgen und
- Aktivieren der Seitentabelle

verfügen. Ein Seitentabelleneintrag muss über Funktionen

- für die Übersetzung von physischen in virtuelle Adressen,
- für die Übersetzung von virtuellen in physische Adressen und
- zum Zugreifen auf die Seiten-Attribute

verfügen.

4.5.2 Entwurf der Kernschnittstelle

Bei den unterschiedlichen Portierungen von Fiasco ist die Seitentabelle in der Klasse `Space_context` repräsentiert. Darin werden für jede Architektur spezifische Funktionen und Methoden implementiert, welche die Seitentabelle modifizieren.

Um eine einheitliche Schnittstelle mit semantisch fest definierten Funktionen zu erhalten und um die Klasse `Space_context` nicht beliebig zu erweitern, wird eine neue Abstraktionsebene in Form der Klasse `Pagetable` eingeführt. `Space_context` (und damit auch `Space`) leitet sich von der neuen Klasse ab. Abbildung 4.1 zeigt die neue Klassenstruktur und die von der neuen Klasse definierten Funktionen und Methoden.

In der neuen Klasse `Pagetable` wird die MMU-Hardware der Prozessorarchitektur abstrahiert und über unterschiedliche Funktionen zur Verfügung gestellt.

Die folgenden Abschnitte beschreiben die Funktionen der architekturneutralen Seitentabellenschnittstelle und stellen sie den in 4.1 genannten Zielen gegenüber.

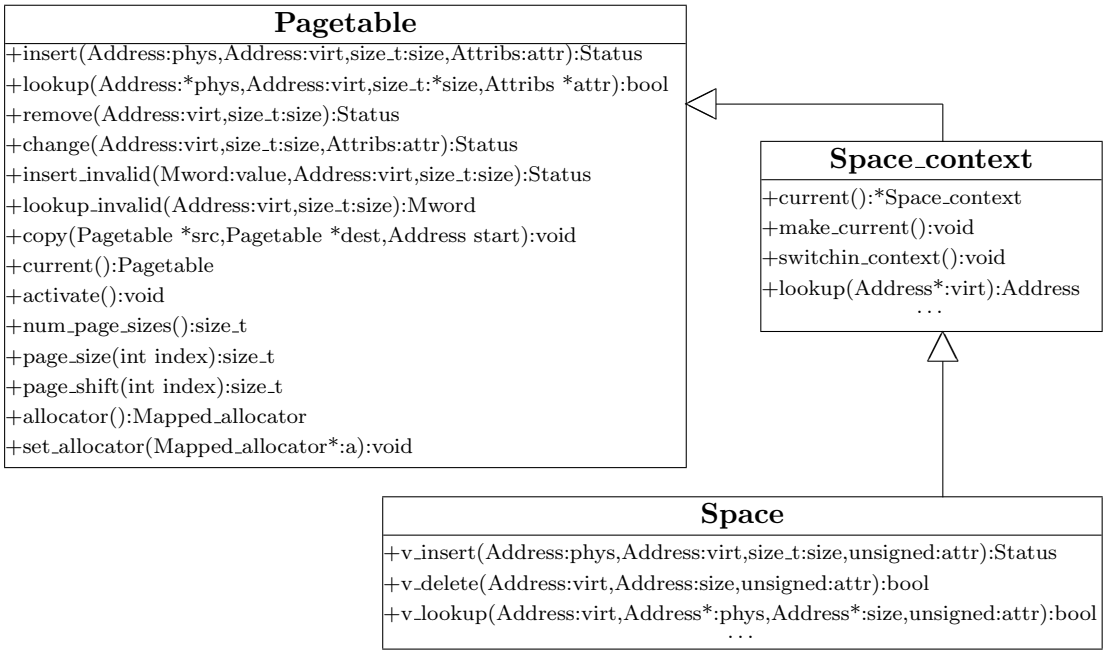
4.5.2.1 Unterschiedliche Seitengrößen

Die architekturspezifischen Definitionen von Konstanten für unterschiedliche Seitengrößen werden durch Funktionen der Seitentabellenschnittstelle zur Ermittlung der Anzahl und der Größe der verfügbaren Seitengrößen ersetzt. Damit werden diese Informationen an zentraler Stelle in der Klasse `Pagetable` zusammengefasst.

Die Funktion `num_page_sizes` liefert einen Wert größer 0, der die Anzahl der Seitengrößen dieser Architektur angibt. Mit `page_size` kann man durch Angabe eines Index die Seitengröße abfragen. Der Index startet wie in C üblich mit 0. Die Seitengrößen sind aufsteigend geordnet, der Index 0 liefert also die kleinste verfügbare Seitengröße.

Um Speicher für eine Seite von einer bestimmten Größe reservieren zu lassen, erwarten die Allokator-Funktionen einen Exponenten als Parameter. Bildet man damit eine Zweierpotenz, dann ergibt das die Größe des gewünschten Speichers in Byte. Daher kann man mit der Funktion `page_shift` mit Hilfe eines Index den richtigen Exponenten zu einer Seitengröße ermitteln.

Abbildung 4.1: Klassendiagramm mit neuer Seitentabellenschnittstelle



Durch die Verwendung dieser Funktionen entsteht kein zusätzlicher Aufwand bei der Ausführung des Kerns, da sich alle Werte während der Übersetzung vom *Compiler* ausrechnen lassen.

4.5.2.2 Unterschiedliche Seitentabellenkonzepte

Die am häufigsten aufgerufenen Funktionen einer Seitentabelle sind das Einfügen und Suchen von Adressübersetzungen. Zusätzlich benötigt man Funktionen zum Löschen und Modifizieren (zum Beispiel Ändern von Zugriffsrechten) von Seitentabelleneinträgen.

Diese Funktionalität wird durch die vier Funktionen `insert`, `lookup`, `change` und `remove` bereitgestellt. Sie sind architekturneutral definiert und werden maximal mit den vier Parametern physische Adresse, virtuelle Adresse, Größe und Seitenattribute aufgerufen. Damit stehen alle Informationen zur Verfügung, die man zum Verwalten der Seitentabelle benötigt. Mit Hilfe des Größenparameters der `insert`-Funktion lässt sich zum Beispiel in einer mehrstufigen Seitentabelle schnell die richtige Hierarchieebene zum Einfügen der Adressübersetzung ermitteln.

Die folgenden Absätze erläutern das Verhalten der vorgestellten Funktionen genauer und definieren die Reaktion bei Randfällen oder Fehlersituationen (siehe 4.5.2.4 für die Definition von Rückgabewerten).

`insert` - Die Funktion `insert` wird mit vier Parametern (physische Adresse, virtuelle Adresse, Seitengröße und Seitenattribute) aufgerufen und fügt eine Adressübersetzung von der übergebenen virtuellen Adresse zur angegebenen physischen

Adresse in die Seitentabelle ein. Dabei können folgende Situationen auftreten:

1. Der Eintrag wird eingefügt und es wird `E_OK` zurückgegeben.
2. An der einzufügenden Stelle existiert bereits ein gültiger Eintrag. In diesem Fall wird `E_EXISTS` zurückgegeben.
3. Der Eintrag kann nicht eingefügt werden, da nicht mehr genug Speicher für die Reservierung weiterer Teile der Seitentabelle zur Verfügung steht. Es wird `E_NOMEM` zurückgegeben.

`lookup` - Die Funktion `lookup` ermittelt die physische Adresse zu der ihr übergebenden virtuellen Adresse. Wenn eine Adressübersetzung in der Seitentabelle gefunden wird, dann wird „wahr“ zurück gegeben. Der übergebene Zeiger für die physische Adresse wird mit der gefundenen Adresse beschrieben. Außerdem werden die ebenfalls als *Pointer* übergebenen weiteren Parameter für die Seitengröße und die Seitenattribute bei einer erfolgreichen Suche mit den entsprechenden Werten belegt.

`change` - Die Funktion `change` ermöglicht es, einen vorhandenen Seitentabelleneintrag zu ändern. Es werden drei Parameter (virtuelle Adresse, Seitengröße und Seitenattribute) übergeben. Dabei können folgende Fälle auftreten:

1. Es wird ein gültiger Eintrag gefunden und die Seitengrößen stimmen überein. Die Seitenattribute werden neu gesetzt und es wird `E_UPGRADE` zurückgegeben
2. Es existiert kein gültiger Eintrag für diese virtuelle Adresse. Es wird `E_INVALID` zurückgegeben.
3. Es existiert ein gültiger Eintrag, aber die Seitengrößen stimmen nicht überein. Es wird `E_INVALID` zurückgegeben.

`remove` - Die Funktion `remove` entfernt zu der übergebenen virtuellen Adresse den Seitentabelleneintrag. Wird ein Eintrag für diese Adresse gefunden, wird er auf Null gesetzt und es wird `E_OK` zurückgegeben. Wenn ein Eintrag gelöscht werden soll, der nicht existiert, wird der Kern-Debugger aufgerufen.

4.5.2.3 Fiasco-spezifische Funktionen

Fiasco-V4 nutzt noch eine provisorische Lösung, um verschiedene Daten zur Prozessverwaltung (zum Beispiel Bereich der KIP, Bereich des UTCB, Prozessnummer) in der Seitentabelle zu speichern. Dazu werden ungültige Einträge eingefügt, in denen nicht eine physische Adresse sondern der zu speichernde Wert kodiert ist. Mit den Funktionen `insert_invalid` und `lookup_invalid` können solche Einträge in der Seitentabelle erzeugt und abgefragt werden.

Bei einem Kontextwechsel muss die Seitentabelle umgeschaltet werden. Auf der IA-32-Plattform wird der Inhalt des CR3-Registers mit der physischen Adresse der neuen Seitentabelle geladen. Auf dem PowerPC-Prozessor müssen dazu jedoch die Segmentregister

neu geladen werden. Diese sehr hardware-spezifischen Vorgänge werden in der Funktion `activate` abgebildet. Sie wird bei jedem Kontextwechsel aufgerufen und übernimmt das Umschalten der Seitentabelle.

Damit die Seitentabelle sparsam mit dem Kernspeicher umgehen kann, erhält sie die Möglichkeit, sich dynamisch Speicher zu reservieren. Das hat zum Beispiel bei mehrstufigen Seitentabellen den Vorteil, dass der Speicher für untere Hierarchieebenen dynamisch reserviert werden kann. Um das zu erreichen, wird der Seitentabellenklasse mit der statischen Funktion `set_allocator` ein Allokator zugewiesen.

Bei der Initialisierung eines neues Adressraums kopiert der Fiasco-Kern die Einträge aus der Kern-Seitentabelle in die Seitentabelle des neuen Adressraums. Diese Funktionalität wird von der statischen Funktion `copy` erbracht. Als Parameter werden zwei Zeiger auf unterschiedliche Seitentabellen und eine virtuelle Startadresse übergeben. Die Startadresse legt fest, ab welcher virtuellen Adresse die Einträge bis zur 4-GB-Grenze kopiert werden sollen. Im obersten Gigabyte blendet damit der Kern seinen Adressraum ein. Bei einem Kontextwechsel stehen dem Kern damit sofort alle Adressumsetzungen zur Verfügung.

4.5.2.4 Konstanten für Seitenattribute

Alle Architekturen, die über MMU-Hardware verfügen, verwalten für Seiten bestimmte Seitenattribute (zum Beispiel *dirty*, *referenced*, *Cache*-Attribute u.a.). Diese werden in für die Adressumsetzung nicht benötigten Bits der Seitentabelleneinträge kodiert. Diese Kodierung ist in hohem Maße architekturabhängig. Es gibt unterschiedliche Wege, einheitliche Konstanten für alle Architekturen zu erhalten. Zwei davon möchte ich hier vorstellen.

Der erste Weg führt über die Einführung einer Indirektionsstufe für solche Werte. In einer architekturunabhängigen Quelldatei werden Konstanten für die unterschiedlichen Seitenattribute definiert und bei einem Aufruf einer Schnittstellenfunktion im Attributsparameter übergeben. Dann wäre es die Aufgabe der architekturspezifischen Implementierung, die architekturneutralen Konstanten in die benötigten Werte zu übersetzen. Durch die Indirektion entsteht allerdings ein Geschwindigkeitsverlust, da die Umsetzung der Werte zusätzliche Prozessorzyklen beansprucht.

Eine weitere Möglichkeit ist es, einen Satz von Namen für Seitenattribute festzulegen, die bei jeder Architektur mit MMU-Hardware zu finden sind. Diese Namen werden dann verwendet, um in der architekturspezifischen Implementierung der Schnittstelle Konstanten mit den benötigten Werten zu definieren. Das kann zum Beispiel als `enum` erfolgen.

Ich habe mich für den zweiten Weg entschieden, da er keine Geschwindigkeitseinbußen durch eine zusätzliche Indirektion verspricht. In Tabelle 4.1 sind die von mir definierten Namen für Seitenattributskonstanten aufgelistet und deren Bedeutung kurz erklärt.

Die Festlegung der Semantik dieser Namen ist wichtig, da es von Architektur zu Architektur Unterschiede bei der Bezeichnung gibt. Bei IBM wird zum Beispiel das *Dirty*-Bit als *Changed*-Bit bezeichnet.

Tabelle 4.1: Architekturneutrale Namen für Seitenattributskonstanten

Name	**Bedeutung**
ENTRY_VALID	gibt die Gültigkeit eines Eintrags an
REFERENCED	gibt an, ob die Seite referenziert wurde oder nicht
WRITABLE	legt fest, ob die Seite geschrieben werden darf
DIRTY	gibt an, ob der Inhalt einer Seite modifiziert wurde
USER	legt fest, dass diese Seite zu einem Benutzerprozess gehört
WRITE_THROUGH	gibt an, ob diese Seite bei Änderung sofort in den Hauptspeicher zurückgeschrieben wird
CACHEABLE	gibt an, ob diese Seite im *Cache* gespeichert werden darf
NONCACHEABLE	gibt an, dass das *Caching* ausgeschaltet sein soll

5 Implementierung

5.1 Fiasco-PPC-Mikrokern

Die vollständige Portierung von Fiasco auf die PowerPC-Architektur konnte bisher nicht abgeschlossen werden. Es gibt Probleme mit der GCC-*Toolchain*, die verhindern, dass der Code für die Ausnahmebehandlung in die Fiasco-Binärdatei gelinkt wird.

5.1.1 Fiasco-Build-System

Das Fiasco-Build-System musste angepasst werden, damit es mit einem PowerPC-Crosskompiler funktioniert. In manchen *Makefiles* mussten Aufrufe von Programmen so angepasst werden, dass das entsprechende Programm der *Crosscompiler-Toolchain* und nicht das von der Entwicklungsplattform verwendet wurde (zum Beispiel statt `ld` Aufruf von `$(LD)`).

5.1.2 Emulator

Die Suche nach einem geeigneten Emulator führte mehrfach in eine Sackgasse. PearPC bzw. Qemu eignen sich nicht, denn sie implementieren nur die minimal benötigte Funktionalität, damit zum Beispiel die Bootloader von Linux oder Mac OS X ausgeführt werden können. Insbesondere fehlt ihnen eine vollständige Implementierung des IEEE-1275-Standards, die mit der geplanten Open-Firmware-Konsole kompatibel ist.

PSIM ist Teil des GNU Debuggers (GDB) und bietet die gewünschten Funktionen. Er emuliert einen CHRP-kompatiblen (**C**ommon **H**ardware **R**eference **P**latform) Rechner mit einer vollständigen Implementierung des *Open Firmware*-Standards. Zusätzlich hat er den Vorteil, dass man ihn direkt aus dem GDB heraus starten kann. Das erleichtert das Debuggen des Kerns deutlich.

Aus diesen Gründen habe ich den PSIM-Simulator bei der Portierung von Fiasco auf den PowerPC-Prozessor verwendet.

Bei der Wahl einer alternativen Implementierung einer Ein-/Ausgabekonsole (zum Beispiel als *Framebuffer*-Gerät) bietet sich die Nutzung von PearPC bzw. Qemu an, da diese auch neuere Versionen des PowerPC-Prozessors emulieren können.

5.1.3 Open-Firmware-Konsole

Die Implementierung einer auf dem *Open Firmware-CI* basierenden Konsole nahm sehr viel Zeit in Anspruch. Es sind keine frei verfügbaren Informationen oder Dokumente

zu finden, in denen das CI beschrieben ist. Erst nach aufwändigem Studium und der Analyse der Quelltexte der Bootloader `yaboot`(siehe [Yab]) und Darwins `BootX` konnte ich die Konsole programmieren.

Die Klasse `OF1275_console` implementiert die allgemeine Konsolenschnittstelle `Console`. Dabei stützt sie sich auf einen *Open-Firmware*-Treiber, der durch die Klassen `Io1275, OF_space und OF1275_ci` implementiert wird.

5.1.4 Bootsequenz

Die Boot-Sequenz von Fiasco für den PowerPC unterscheidet sich leicht von der Variante für Fiasco-V2 für IA-32. Die Unterschiede ergeben sich aus dem Nichtvorhandensein eines Bootloaders (zum Beispiel Grub für IA-32), welcher einzelne Module laden und starten könnte.

5.1.4.1 Bootloader

Wie in Abschnitt 2.4.1 beschrieben, kann die *Open Firmware* ELF-Binärdateien direkt laden und starten. Da jedoch für ein Minimalsystem zusätzlich zum Kern auch noch die *Roottask* und *Sigma0* geladen werden müssen, müssen diese zusammen mit einem Bootloader in ein einziges Binär-Image gepackt werden. Das kann dann von der *Open Firmware* geladen und gestartet werden.

Zum Einsatz kommt hier der *Piggybacker* Bootloader von der Betriebssystemgruppe der Universität Karlsruhe, die ihn auch für die PowerPC-Variante von L4/Pistachio einsetzt (siehe [Pig]).

Nachdem *Piggybacker* von der *Open Firmware* gestartet wurde, initialisiert er zunächst das *Open Firmware-CI*, um auf diesem Wege eine Bootkonsole aufzusetzen. Anschließend werden die einzelnen Module (Kern, *Roottask* und *Sigma0*) an ihre entsprechenden Link-Adressen kopiert. Dann sucht *Piggybacker* im Kern-Modul nach der KIP (über die *magic number L4μk*) und trägt dort den Speicherbereich der *Open Firmware* und des Gerätebaums als reserviert ein. Anschließend wird der Kern gestartet.

5.1.4.2 Startsequenz des Kerns

Von *Piggybacker* wird im Kern die Funktion `_start` aufgerufen. Der Ablauf ist wie folgt:

1. Es wird sichergestellt, dass die virtuelle Adressierung abgeschaltet ist. Für den gesamten vom Kern benötigten Adressraum werden zwei *BAT-Mappings* angelegt (Text und Daten), damit im Kern keine Seitenfehler auftreten können. Die virtuelle Adressierung wird eingeschaltet und die `__main`-Funktion im Kern angesprungen.

2. `__main` initialisiert die Open-Firmware-Konsole, damit die `printf`-Funktion zur Verfügung steht.

3. Die Konstruktoren der statischen Objekte werden ausgeführt.

5.1.5 Roottask und Sigma0

Ohne bestimmte vertrauenswürdige Benutzerprogramme ist ein Mikrokern nicht sinnvoll zu benutzen. Das wichtigste Programm hier ist der Rootpager (*Sigma0*). Er bildet den ersten Adressraum im System. Außerdem bekommt er zunächst allen physischen Speicher zugewiesen, der dann insgesamt oder in Teilen von weiteren Applikationen über das Sigma0-Protokoll angefordert werden kann. Das zweite wichtige Programm ist *Roottask*. Es erhält das Recht, weitere Prozesse zu erzeugen. Später müssen alle weiteren Prozesse von *Roottask* gestartet werden.

Um den Aufwand bei der Portierung zu begrenzen, habe ich mich entschieden, den *Sigma0*-Server und den Pingpong-Server (als *Roottask*) von L4/Pistachio zu verwenden. Die beiden Server sind bereits für die L4/V4 Schnittstelle angepasst. Außerdem liegen sie auch schon in Varianten für den PowerPC vor.

Diese beiden Server, sowie der Bootloader *Piggybacker* waren bereits als Paket unter dem Namen „x2tools“ für den Fiasco-V4 Kern für die IA-32-Architektur vorhanden. Es waren jedoch kleinere Änderungen an den *Makefiles* notwendig, damit das Paket mit einem *Cross-Compiler* übersetzt werden konnte.

5.2 Seitentabellenschnittstelle

Die architekturneutrale Seitentabellenschnittstelle für Fiasco ist im Kapitel 4 beschrieben. Dieser Abschnitt erläutert deren Implementierung am Beispiel des Fiasco-V4-Kerns für die IA-32-Architektur. Im Abschnitt 5.2.2 gebe ich einen Ausblick auf die Implementierung dieser Schnittstelle für den PowerPC-Prozessor.

Die Implementierung ist im Wesentlichen auf zwei Quelldateien aufgeteilt. In einem architekturneutralen Teil (`pagetable.cpp`) werden die Schnittstellenfunktionen definiert. In einem weiteren architekturspezifischen Teil werden diese Funktionen dann implementiert. So wird die MMU-Hardware der Prozessor-Architektur abstrahiert.

5.2.1 IA-32

Dieser Abschnitt erläutert die Implementierung der Seitentabellenschnittstelle für die IA-32-Architektur. Die Schnittstellenfunktionen werden in der Datei `pagetable-ia32.cpp` implementiert. Die in der Datei `pagetable.cpp` definierte Klasse `Pagetable` wird hier zunächst um die öffentliche Definition der Konstanten (siehe Abschnitt 4.5.2.4) erweitert. Die Werte dieser Konstanten werden auf die für die IA-32-Architektur benötigten festgelegt.

5.2.1.1 Struktur der Seitentabellenrepräsentation

Durch die neue Schnittstelle wird die Struktur der Seitentabellenrepräsentation vor den restlichen Teilen des Kern verborgen. Die Definition der Seitentabellenschnittstelle trifft keine Festlegungen über die Art und Weise der Struktur der Seitentabellenre-

präsentation. Sie muss lediglich so gestaltet werden, so dass sie von der MMU-Hardware für die Adressübersetzung genutzt werden kann.

Die verwendeten Strukturen wurden größtenteils aus der vorherigen Implementierung ohne Seitentabellenschnittstelle übernommen und an die Anforderungen der Schnittstelle angepasst. Die Klasse `Pagetable` wurde um ein privates, 1024 Einträge großes Feld von `Unsigned32`-Elementen erweitert. Darin wird das Seitentabellenverzeichnis abgebildet. Einträge aus diesem Feld verweisen später entweder auf eine 4-MB-Seite oder eine Seitentabelle (Klasse `Ptab`) mit ebenfalls 1024 Einträgen.

Die Klasse `Ptab` wird in der Datei `paging-ia32.cpp` definiert und besteht aus einem 1024 Einträge großem Feld von `Unsigned32`-Werten. Zusätzlich werden Hilfsfunktionen festgelegt, die später die Implementierung der Schnittstellenfunktionen vereinfachen. So kann man zum Beispiel schnell den Seitentabellenindex einer gegebenen virtuellen Adresse ermitteln.

5.2.1.2 Implementierung der Schnittstellenfunktionen

Die Definition der Schnittstelle sieht die Festlegung eines Speicher-Allokators vor, damit Speicher für Seitentabellen dynamisch reserviert werden kann. Erst wenn eine Seite nicht mehr als 4-MB-Seite eingetragen werden kann, wird Speicher für eine Seitentabelle reserviert. In diese wird die Seite dann eingetragen.

Die öffentlichen Schnittstellenfunktionen sind so implementiert, dass sie das im Abschnitt 4.5.2.2 definierte Verhalten zeigen. Sie stützen sich dabei auf eine Vielzahl an privaten Hilfsfunktionen, die den Quellcode übersichtlicher machen. Außerdem erleichtern sie den internen Umgang mit der Seitentabelle erheblich. Zu den Hilfsfunktionen gehören zum Beispiel Funktionen zum Ermitteln des Index im Seitentabellenverzeichnis oder zum Überprüfen der Gültigkeit von Einträgen. Die Hilfsfunktionen sind nur sehr kurz. Damit beim Aufruf nicht zusätzlicher Aufwand entsteht, sind sie als *inline*-Funktionen deklariert.

Besondere Beachtung muss man der Implementierung der unterschiedlichen Einfüge-Operationen schenken. Grundsätzlich kann man hier zwischen dem Einfügen von Adressumsetzungen (*map*) und dem Einblenden von TCBs unterscheiden. Im ersten Fall werden gültige Einträge in der Seitentabelle angelegt (für 4-KB- oder 4-MB-Seiten). Dabei ist es möglicherweise erforderlich neuen Speicher für eine Seitentabelle zu reservieren. Für die TCBs benötigt man Funktionen zum Einfügen ungültiger Seiten, damit dort Prozessinformationen in der Seitentabelle gespeichert werden können. Außerdem benötigt man hier eine Funktion zum Kopieren von Seitentabellenverzeichniseinträgen, da so Verweise auf schon existierende Seitentabellen in die Seitentabelle eines Prozesses eingefügt werden.

5.2.1.3 Anpassungen im Kern

Charakteristisch für den Fiasco-V4-Kern war bisher das Fehlen jeglicher Abstraktion für die Seitentabelle. An den Stellen im Kern, an denen eine Änderung an der Seitentabelle nötig war, wurde bisher die Änderung direkt vorgenommen. Das ist auch der

Grund, warum sich die Änderungen auf mehrere Subsysteme des Kerns auswirken.

Umfangreiche Anpassungen waren an den Klassen `Kmem`, `Vmem_alloc`, `Space` und `Space_context` nötig. Kleinere Änderungen gab es in der Klasse `Thread` und `Idt`. Damit der Kern-Debugger wieder funktionsfähig ist, waren kleinere Änderungen an der Klasse `Jdb_dbinfo` erforderlich.

Da die Klasse `Pagetable` als neue Indirektionsstufe eingeführt wurde und sich die Klassen `Space_context` und `Space` von ihr ableiten, waren hier Anpassungen nötig. Die Funktionen `v_lookup`, `v_insert` und `v_delete` der Klasse `Space` rufen die entsprechenden Funktionen der Seitentabellenklasse auf.

5.2.2 PowerPC

Dieser Abschnitt gibt einen Ausblick auf die Schritte und Möglichkeiten, wie die Seitentabellenschnittstelle für den PowerPC-Prozessor implementiert werden kann. Zunächst müssen die Konstanten für die Seitenattribute festgelegt werden. Die nötigen Informationen dafür finden sich in [IBM03] im Abschnitt 7.

5.2.2.1 Struktur der Seitentabellenrepräsentation

Bei der PowerPC-Implementierung sollte die Seitentabelle durch ein Feld von PTEGs abgebildet werden, die jeweils aus acht PTEs bestehen. PTEs können dabei als Datenstruktur `pte_t` definiert werden, die zum Beispiel folgendermaßen aussieht:

```
struct pte_t {
  Unsigned32 upper;
  Unsigned32 lower;
};
```

Der für die Seitentabelle benötigte Speicherplatz lässt sich schon beim Kernstart aus der Größe des verfügbaren Hauptspeichers berechnen. Die Formel für die Berechnung der minimalen Größe lautet (nach [IBM03] Seite 317):

$$\frac{GrößeHauptspeicher}{2 * Seitengröße} * GrößePTEG.$$

Zusätzlich bietet es sich auch hier an, Hilfsfunktionen zu entwickeln, die häufig benötigte Informationen aus einem PTE extrahieren. Damit kann die Implementierung der Schnittstellenfunktionen erheblich vereinfacht werden.

5.2.2.2 Implementierung der Schnittstellenfunktionen

Im Unterschied zur IA-32-Architektur gibt es beim PowerPC-Prozessor nur eine Seitentabelle für alle Adressräume. Gültige Seitentabelleneinträge für einen Adressraum werden über die VSID ermittelt (siehe auch 2.3.1.6), die die Funktion einer Adressraum-ID hat.

In 4.3.1 habe ich die Probleme und Besonderheiten, welche bei der PowerPC-Architektur auftreten können, dargestellt. Dort ist beschrieben, dass bei der Verdrängung

eines Eintrags aus der Seitentabelle, dieser Eintrag in die „Schatten“-Seitentabelle des zugehörigen Adressraums eingetragen werden muss. Um eine aufwändige Suche nach dem richtigen Adressraum zu vermeiden, ist die „richtige“ Wahl der VSID entscheidend. Verwendet man als VSID die physische Adresse des `Pagetable`-Objekts, dann kann man zu einem gegebenen Eintrag das richtige `Pagetable`-Objekt berechnen und ihn in die richtige „Schatten“-Seitentabelle eintragen.

Die zusätzliche Seitentabelle zur Verwaltung der verdrängten Einträge kann als zweistufige Seitentabelle implementiert werden. In ihr werden Verweise auf die verdrängten Einträge gespeichert und können bei Bedarf abgerufen werden.

Die Entwicklung einer geeigneten Ersetzungsstrategie für Seitentabelleneinträge liegt außerhalb des Fokus dieser Arbeit und wird deshalb hier nicht weiter betrachtet. Es bietet sich an, schon bekannte Seitenersetzungsalgorithmen zu verwenden und für diese Problemstellung anzupassen.

6 Ergebnisse und Bewertung

In diesem Kapitel fasse ich die Ergebnisse dieser Arbeit zusammen und bewerte sie anhand unterschiedlicher Kriterien.

6.1 Seitentabellenschnittstelle

Das Ziel dieser Arbeit bestand in der Entwicklung einer architekturneutralen Seitentabellenschnittstelle für Fiasco-V4. Die Schnittstelle wurde dann für die IA-32-Architektur implementiert. Die folgenden Abschnitte überprüfen anhand unterschiedlicher Kriterien, ob die Zielstellungen bei der Entwicklung der Schnittstelle erreicht wurden.

6.1.1 Architekturneutralität

Die Architekturneutralität der Seitentabellenschnittstelle kann an dieser Stelle nur theoretisch diskutiert werden, da die Schnittstelle bisher nur für die IA-32-Architektur implementiert wurde.

In der Schnittstelle wurden die Funktionen in ihrer Semantik so definiert, dass sie ein möglichst breites Spektrum an Anwendungsfällen abdecken. Sie bieten die Funktionalität, die für ein System, welches mit virtuellem Speicher arbeitet, nötig ist.

Der Kern kann durch Anfragen an die Schnittstelle Informationen über die Möglichkeiten der ihm zur Verfügung stehenden MMU-Hardware ermitteln (zum Beispiel Anzahl verfügbarer Seitengrößen oder Seitengrößen). Mit Hilfe der fünf Funktionen `insert`, `lookup`, `change`, `remove` und `copy` wird die Seitentabelle modifiziert. Dabei macht die Schnittstellendefinition keine Annahmen oder Festlegungen über die Implementierung und Definition der Datenstrukturen für die Repräsentation der Seitentabelle.

Der Größenparameter, der bei vielen Funktionen angegeben werden muss, ermöglicht die Unterstützung unterschiedlicher Seitengrößen. Bei Systemen mit mehrstufigen Seitentabellen kann dieser Parameter dazu genutzt werden, die zu ändernde Stelle innerhalb der Seitentabellenhierarchie schnell aufzufinden. Architekturen mit *hashed*-Seitentabellen nutzen diesen Parameter, um die Größeninformation in dem Seitentabelleneintrag zu kodieren.

6.1.2 Implementierung

Die im Abschnitt 4.5.1 beschriebene Softwarestruktur wurde für die Entwicklung der Schnittstelle auf die Klasse `Pagetable` reduziert. Dies hat folgende Gründe:

- Die Struktur von Seitentabelleneinträgen ist zu inkonsistent, so dass keine Gemeinsamkeiten exisitieren, die sich sinnvoll in einer abstrakten Klasse kapseln lassen.

- Durch die unterschiedliche Struktur lassen sich nur innerhalb einer Architektur Funktionen herauslösen und in einer separaten Klasse kapseln. Diese sind aber nicht portabel.

Daher wurde die Definition der Datenstrukturen und der Funktionen für Seitentabelleneinträge dem architekturspezifischen Teil der Implementierung überlassen. Dadurch erhält man eine größere Flexibilität bei der Implementierung.

Bei der IA-32-Architektur belegt die komplette Seitentabelle bei einer Seitengröße von 4 KB normalerweise 4 MB. Da die Speicherzugriffe von Programmen zum größten Teil aber nur auf einen begrenzten Adressbereich fallen, ist es nicht sinnvoll, von vornherein den kompletten Speicher dafür zu reservieren. Beim Erzeugen eines Prozesses wird durch das Erzeugen einer neuen Instanz der Klasse `Pagetable` ein Seitentabellenverzeichnis (Größe 4 KB) im Speicher angelegt. Der benötigte Aufwand, diese Struktur auch dynamisch zu erzeugen, ist zu groß und würde die Ausführungsgeschwindigkeit des Kerns negativ beeinflussen. Außerdem wird sie zum Aktivieren eines neuen Prozesses benötigt und muss daher sofort zugreifbar sein. Die Seitentabellen jedoch (Größe je 4 KB) werden nur bei Bedarf im Speicher angelegt. So vergrößert sich die Seitentabelle dynamisch und nimmt nur den wirklich benötigten Speicher ein. Nachteil im Moment ist, dass eine leere Seitentabelle nicht erkannt und demzufolge auch nicht wieder freigegeben wird. Hat man zum Beispiel einen Prozess, der dynamisch Speicher quer durch seinen gesamten Adressraum reserviert, so wird seine Seitentabelle irgendwann die vollen 4 MB belegen. Erst wenn der Prozess beendet wird, wird dieser Speicher wieder freigegeben.

6.1.3 Messungen

Um die Geschwindigkeit der neuen Implementierung zu bewerten, habe ich auf drei unterschiedlichen Testrechnern Testläufe durchgeführt und mit der original Fiasco-V4 Version verglichen. Die Konfiguration der Testrechner kann Tabelle 6.1 entnommen werden.

Tabelle 6.1: Testrechnerkonfiguration

Prozessor	**Hauptspeicher**
Pentium 3, 450MHz	256 MB
AMD Sempron, 1.5GHz,	256 MB
Pentium 4, 2.8GHz	256 MB

6.1.4 Vergleichstests

Die Vergleichstests der Geschwindigkeit des ursprünglichen Fiasco-V4-Kerns und des Kerns mit neuer Seitentabellenschnittstelle habe ich mit dem Pingpong-*Benchmark*-Programm und einem eigenen kerninternen Mikrobenchmark durchgeführt.

In Tabelle 6.2 sind die Ergebnisse des Pingpong-IPC-*Benchmarks* dargestellt. Der

Tabelle 6.2: Prozessortakte im Pingpong-IPC-Benchmark

Rechner	**IPC-Typ**	**Fiasco-V4 (neu)**	**Fiasco-V4**	$\Delta(in\%)$
Pentium 3	Inter-AS	790	776	1,8
	Intra-AS	552	527	4,9
AMD Sempron	Inter-AS	1583	1594	-0,7
	Intra-AS	1146	1137	0,8
Pentium 4	Inter-AS	8402	8394	0,1
	Intra-AS	7798	7741	0,7

zusätzliche Aufwand durch die Seitentabellenschnittstelle beträgt zwischen 0.1% (Pentium 4, Inter-AS) und 4.9% (Pentium 3, Intra-AS). Der Grund dafür ist, dass der Mehraufwand durch die Seitentabellenschnittstelle im Vergleich zum Aufwand für Kernein- und -austritte gering ist. Der Wert für den Inter-AS-Test auf dem AMD Sempron Prozessor ist sogar geringfügig kleiner als beim ursprünglichen Fiasco-V4-Kern.

Um die Kosten für die Schnittstelle direkt zu messen, habe ich einen kerninternen Mikrobenchmark geschrieben, der eine Sequenz von `lookup - insert - lookup - remove` Operationen in der Seitentabelle nachbildet. Diese Sequenz bildet in begrenztem Maße die Vorgänge in einer IPC-Operation nach. In Tabelle 6.3 sind die Ergebnisse dieses Tests zu sehen. Es zeigt sich ein deutlicher Mehraufwand von etwa 40%. Dieser Mehr-

Tabelle 6.3: Ergebnisse des kernintern Mikrobenchmarks (Prozessortakte)

Rechner	**Fiasco-V4 (neu)**	**Fiasco-V4**	$\Delta(in\%)$
Pentium 3	108.269.668	77.207.796	40
AMD Sempron	99.304.606	71.735.393	38
Pentium 4	128.889.201	93.002.784	38

aufwand kommt durch die zusätzliche Abstraktionsschicht der Seitentabellenschnittstelle zustande. Im ursprünglichen Fiasco-V4-Kern wurden die Seitentabellen direkt durch Speicherzugriffe geändert. In der neuen Variante wird dies über die entsprechenden Funktionsaufrufe gemacht. Zu den Kosten des Funktionsaufrufs kommen noch die Kosten für die Beachtung bestimmter Spezialfälle.

6.1.5 Komplexität des Quellcodes

Ein Maß für die Komplexität einer Software ist die Anzahl der Quellcodezeilen. Ziel bei der Entwicklung der Schnittstelle war es, die Komplexität des Kerns zu reduzieren. Messungen zeigen, dass dieses Ziel erreicht werden konnten. Die Anzahl der Quellcodezeilen ist leicht von ursprünglich 43.385 auf 43.218 gesunken. Die Messungen wurden mit dem Programm `sloccount` (siehe [slo]) durchgeführt. Die Komplexität des Quellcodes wird noch weiter reduziert, wenn später auch der bisher verbliebende *Debug*-Code entfernt worden ist.

Durch die leicht gesunkene Codekomplexität und die Zusammenfassung der Seitentabellenmodifikationen unter einer zentralen Schnittstelle, verbessert sich die Wartungsfreundlichkeit des Kerns. Die Anzahl möglicher Fehlerquellen wird reduziert. Außerdem ermöglicht diese Struktur zukünftig, dass der Fiasco-Mikrokern besser auf andere Architekturen übertragen werden kann.

7 Ausblick und Zusammenfassung

7.1 PowerPC-Portierung

Aufgrund der im Abschnitt 5 genannten Schwierigkeiten startet der Kern bisher nur die virtuelle Adressierung und initialisiert die Open-Firmware-Konsole.

Um eine lauffähige PowerPC-Version von Fiasco zu bekommen, sind aus meiner Sicht als nächstes folgende Schritte nötig:

- Identifizieren des Problems mit der GCC-Toolkette und anschließend Implementierung der Ausnahmebehandlungsroutinen,
- Implementierung der Seitentabellenschnittstelle,
- Implementierung der weiteren architekturspezifischen Teile von Fiasco für den PowerPC.

7.2 Seitentabellenschnittstelle

Um die Stabilität und Eignung der neuen Schnittstelle zu testen, wäre die Implementierung für andere Architekturen das geeignete Mittel. Als sinnvoller Kandidat erscheint mir die PowerPC-Implementierung selbst, da hier eine Variante für ein anderes Konzept von Seitentabellen entstehen würde. Die AMD64-Architektur ist ein weiterer Kandidat, da hier die Eignung für bis zu 4-stufige Seitentabellen intensiv geprüft werden kann.

Bevor man diese Arbeiten beginnt, ist die Entwicklung einer echten Prozessabstraktion sinnvoll. Damit könnte die Seitentabellenschnittstelle weiter vereinfacht werden, denn der *Hack*, Prozessinformationen in nicht gültigen Seiten zu „verstecken“, kann damit entfallen.

7.2.1 FiascoUX

FiascoUX ist eine Variante des Kerns, die auf der UNIX-Systemrufschnittstelle aufbaut. Der Vorteil ist, dass der Kern als normale Anwendung ausgeführt werden kann und damit die Entwicklung von Anwendungen erheblich vereinfacht wird.

Bei der Implementierung der Seitentabellenschnittstelle habe ich bei meinen Arbeiten keine Rücksicht auf die Kompatibilität mit FiascoUX genommen. Die Lauffähigkeit wurde bisher nicht weiter untersucht.

Ein Hinweis darauf, wie FiascoUX mit Seitentabellen umgeht, findet sich in [Ste02].

7.3 Zusammenfassung

Das Ergebnis dieser Arbeit stellt eine gute Grundlage dar, die vollständige Portierung des Fiasco-Mikrokerns auf den PowerPC durchzuführen. Es wurden viele Grundlagen ausgearbeitet, die die Portierung ermöglichen.

Zunächst waren dazu Anpassungen am Fiasco-V4-*Build*-System nötig, damit es in einer Cross-Kompilerumgebung funktionsfähig ist. Danach wurde die Infrastruktur (Emulator, Bootloader, *Roottask* und *Sigma0*) geschaffen, mit der die vollständige Portierung auf die PowerPC-Architektur durchgeführt werden kann.

Der wesentliche Teil der Aufgabe bestand in der Entwicklung einer Seitentabellenschnittstelle, welche die architekturspezifische MMU-Hardware im Kern abstrahiert. Mit dieser Schnittstelle werden alle Modifikationen von Seitentabellen zusammengefasst. Damit werden zukünftige Portierungen des Kerns auf neue Architekturen erleichtert. Zusätzlich verbessert sich die Wartungsfreundlichkeit des Kerncodes, da alle Modifikationen der Seitentabelle nun an einer zentralen Stelle stattfinden.

Dank

An dieser Stelle möchte ich all denen danken, die mir geholfen haben, die zahlreichen Probleme bei dieser Arbeit zu lösen. Insbesondere geht der Dank an meinen Betreuer Michael Hohmuth sowie an Michael Peter, die mir mit viel Geduld und Ausdauer immer wieder bei der Fehlersuche geholfen haben. Besonders bedanken möchte ich mich bei Michael Peter für die Bereitstellung seines modifizierten Qemu-Emulators, der von unschätzbarem Vorteil beim Aufspüren meiner Implementierungsfehler war.

Abkürzungsverzeichnis

API	Abbreviated Page Index
ASCII	American Standard Code for Information Interchange
ASID	Adressraum-ID
BAT	Block Array Translation
CHRP	Common Hardware Reference Platform
CISC	Complex Instruction Set Code
DROPS	Dresden Realtime Operating System
ELF	Executable and Linking Format
IDT	Interrupt Descriptor Table
IPC	Inter-Process-Communication
KIP	Kernel Info Page
MMU	Memory Management Unit
MSR	Machine State Register
Open Firmware CI	Open Firmware Client Interface
PAE	Physical Address Extension
PTE	Page Table Entry
PTEG	Page Table Entry Group
RISC	Reduced Instruction Set Computing
SRR0	Save Restore Register 0
SRR1	Save Restore Register 1
TCB	Thread Control Block
TLB	Translation Lookaside Buffer
UTCB	User Thread Control Block

Literaturverzeichnis

[BBB+90] Baron, R. V., Black, D., Bolosky, W., Chew, J., Draves, R. P., Golub, D. B., Rashid, R. F., Avadis Tevanian, Jr., and Young, M. W. Mach kernel interface manual. Technical report, School of Computer Science, Carnegie Mellon University, `ftp://ftp.cs.cmu.edu/project/mach/doc/unpublished/manual.ps`, 1990. 9

[Fre05] Thorsten Frenzel. Porting Fiasco to AMD64. Technische Universität Dresden, 2005. `http://os.inf.tu-dresden.de/papers_ps/frenzel-beleg.pdf`. 23

[Hei01] Gernot Heiser. *Inside L4/MIPS: anatomy of a high-performance microkernel.* Operating systems group, University of New South Wales, Australia, 2001. 10

[Hoh98] M. Hohmuth. The fiasco kernel: Requirements definition. Technical report, Technische Universität Dresden, 1998. 7

[IBM03] IBM. *PowerPC Microprocessor Family: Programming Environments Manual for 64 and 32-Bit Microprocessors Version 2.0.* IBM Microelectronics Division, 2003. 8, 11, 16, 19, 37

[Lie96] J. Liedtke. *L4 Reference Manual 486 Pentium, Pentium Pro.* GMD - German National Research Center for Information Technology, 1996. 10, 20

[Nan] Nanokernel. `http://en.wikipedia.org/wiki/Nanokernel`. 9

[Pig] Piggybacker bootloader. `http://l4ka.org/projects/pistachio/powerpc/build.php`. 34

[Pis] Pistachio microkernel. `http://l4ka.org/projects/pistachio/`. 10

[Sin] Amit Singh. Hanoi: Open Firmware (Graphical, Macintosh). `http://www.kernelthread.com/hanoi/html/macprom-gfx.html`. 17

[slo] Sloccount. `http://www.dwheeler.com/sloccount`. 42

[Ste02] Udo Steinberg. Fiasco μ-Kernel User Mode Port. Technische Universität Dresden, 2002. `http://os.inf.tu-dresden.de/papers_ps/steinberg-beleg.pdf`. 43

[SUN] SUN Microsystems, `http://playground.sun.com/1275/`. *IEEE 1275 Open Firmware Homepage.* 17

[Tea05] L4Ka Team. *L4 eXperimental Kernel Reference Manual*. System Architecture Group Dept. of Computer Science, Universität Karlsruhe, `http://l4ka.org/projects/pistachio/l4-x2-r5.pdf`, 12. April 2005. 19, 21

[War03] Alexander Warg. Software Structure and Portability of the Fiasco Microkernel. Master's thesis, Technische Universität Dresden, 2003. 23

[Yab] Yaboot Bootloader. `http://yaboot.ozlabs.org/`. 34