UNIVERSITÄT LEIPZIG

Evaluation von Echtzeitbetriebssystemen für den Einsatz in Integrated Modular Avionics

Diplomarbeit am Institut für Informatik
bei Dr. Martin Bogdan
Fakultät für Mathematik und Informatik
Universität Leipzig

von

Martin Christian

Betreuer: Dr. Martin Bogdan
Dipl.-Inf. Thomas Schanne

Tag der Anmeldung: 28. November 2006
Tag der Abgabe: 24. Mai 2007

Zusammenfassung

Ein Betriebssystem muss zum Einsatz in der Luftfahrt verschiedene widersprüchliche Anforderungen erfüllen. Es soll einerseits kostengünstig in Entwicklung und Wartung, andererseits aber sehr zuverlässig im Betrieb sein. Die Kosten können durch Integration mehrerer Avionik-Funktionen auf eine Hardware, offene Standards und einen modularen Aufbau gesenkt werden. Das Betriebssystem muss garantieren können, dass sich Fehler nicht fortpflanzen und Zeitbedingungen immer eingehalten werden. Außerdem sollte es selbst fehlerfrei sein. Das freie Betriebssystem Linux ist eine kostengünstige Alternative zu kommerziellen Betriebssystemen. Für den Einsatz in kritischen Anwendungen der Luftfahrt ist es jedoch nicht geeignet. Unkritische Anwendungen können hingegen mit Linux realisiert werden, wenn es als isolierter Teil eines Echtzeit-Systems zum Einsatz kommt. Die Isolation von Linux erfolgt durch eine virtuelle Maschine. Virtuelle Maschinen kapseln den Zugriff auf die reale Hardware indem sie eine virtuelle Hardware zur Verfügung stellen. Der μ-Kernel Pistachio-embedded stellt eine virtuelle Maschine in Form der Kernelschnittstelle L4 bereit. Außerdem soll er harte Echtzeit-Bedingungen erfüllen können. L4 definiert durch Threads, Adressräume und Kommunikationskanäle eine abstrakte Hardware. Mit Wombat gibt es bereits ein Linux, das auf die L4-Hardware portiert worden ist und somit in einer virtuellen Maschine ausgeführt wird. Wombat basiert auf dem Embedded Systems Framework vom National ICT Australia. Da Pistachio-embedded und das Embedded Systems Framework nicht für die PowerPC-Architektur verfügbar waren, wurden sie im Laufe der Diplomarbeit auf diese Architektur portiert. Der praktische Teil wurde von der EADS Deutschland GmbH durch ein Entwicklungsboard mit PowerPC-Prozessor unterstützt.

Danksagung

Ohne die Unterstützung der EADS Deutschland GmbH, insbesondere von Andreas Schacht und Thomas Schanne, wäre diese Arbeit nicht zustande gekommen. Von daher danke ich Ihnen und der ganzen Abteilung OPEE45 für die Unterstützung, die mir im letzten halben Jahr zuteil wurde.

Einen großen Dank schulde ich auch jenen, die meine häufigen Fragen auf den L4-Mailinglisten der Universität Karlsruhe, der Technischen Universität Dresden und dem NICTA in Sydney beantwortet haben. Von der Universität Karlsruhe danke ich Espen Skoglund, Jan Stoess, Joshua LeVasseur, Uwe Dannowski und Volkmar Uhlig. Von den Open Kernel Labs danke ich Ben Leslie, Carl van Schaik, Matthew Warton und Malcolm Purvis. Vom NICTA und der University of New South Wales danke ich Gernot Heiser, Nicholas FitzRoy-Dale und Sergio Ruocco. Von der Technischen Universität Dresden danke ich Adam Lackorzynski. Von anderer Stelle antwortete mir Matthew R. Dempsky dem ich ebenfalls danke.

Nicht zuletzt danke ich auch Roland Bless von der Universität Karlsruhe für die Veröffentlichung der Latex-Vorlage, die beim Schreiben dieser Arbeit Verwendung fand.

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Zielsetzung der Arbeit 1

1.2 Gliederung der Arbeit 1

2 Anforderungen **3**

2.1 Integrated Modular Avionic 3

2.1.1 Ziele 3

2.1.2 Zertifizierung 4

2.1.3 Standards in anderen Industriezweigen 5

2.2 Partitionierung 5

2.2.1 Räumliche Partitionierung 6

2.2.2 Zeitliche Partitionierung 7

2.3 Echtzeit 8

2.4 Kernel vs. Betriebssystem 10

2.5 Virtualisierung 12

2.6 Anforderungen an einen IMA-Kernel 14

2.6.1 Echtzeit 14

2.6.2 Partitionierung 15

2.6.3 Trusted Computing Base 15

2.6.4 Offene Standards 15

2.6.5 Modularität 16

2.7 Zusammenfassung 16

3 Analyse **17**

3.1 Methodik . . . 18
3.2 Kernel . . . 19
3.2.1 Linux . . . 19
3.2.2 Minix 3 . . . 21
3.2.3 Xen . . . 23
3.2.4 L4 . . . 25
3.3 Auswahl . . . 28
3.4 Zusammenfassung . . . 30

4 Implementierung **31**

4.1 Hardware . . . 31
4.1.1 Board . . . 31
4.1.2 PowerPC-Architektur . . . 31
4.1.3 PowerPC 750Cxe . . . 34
4.1.4 Hardware Mechanismen und IMA-Anforderungen . . . 35
4.2 Software . . . 37
4.2.1 Linux Portierungen auf den μ-Kernel L4 . . . 37
4.3 Arbeitsschritte . . . 39
4.3.1 Toolchain . . . 39
4.3.2 Build-System . . . 41
4.3.3 Plattform . . . 42
4.3.4 Architektur . . . 46
4.3.5 Embedded Systems Framework . . . 49
4.4 Zusammenfassung . . . 51

5 Evaluierung **52**

5.1 IMA-Tauglichkeit . . . 52
5.1.1 Echtzeit . . . 52
5.1.2 Partitionierung . . . 54
5.1.3 Trusted Computing Base . . . 56
5.1.4 Offene Standards . . . 58
5.1.5 Modularität . . . 58
5.2 Fazit . . . 59
5.3 Zusammenfassung . . . 60

6 Zusammenfassung und Ausblick **61**

6.1 Zusammenfassung 61

6.2 Ausblick 62

A Abkürzungen **64**

B Inhalt der CD **67**

C Abhängigkeitsproblem **68**

Literatur **71**

Index **75**

Erklärung **77**

Abbildungsverzeichnis

2.1 Förderierte vs. IMA-Architektur 4
2.2 Generische IMA-Architektur nach [Benn03] 12

3.1 Linux Architektur 19
3.2 Dual-Kernel Architektur 20
3.3 Minix 3 Architektur nach [HBGH+06] 22
3.4 Xen Architektur nach [BDFH+03] 24
3.5 Verhältnis von Adressraum zu Thread 25
3.6 Typische L4-Architektur 27

4.1 Blockschaltbild des Hellas-Boards (Quelle: EADS Deutschland GmbH) 32
4.2 Register des PowerPC 750Cxe (Quelle: [IBM05]) 33
4.3 Address-Translation for PowerPC 750Cxe (Quelle: [IBM05]) 34
4.4 Paging-Verfahren der PowerPC-Architektur (Quelle: [IBM00]) 36
4.5 Arbeitsschritte 40
4.6 Build-Prozess beim Embedded Systems Framework 42
4.7 Das ELF-Image des Pistachio Kernels für PowerPC. 44
4.8 Interrupt-Handler im PowerPC-Code 45
4.9 Hierarchischer Aufbau des Interrupt-Controllers der Hostbridge . . . 45
4.10 Implementierung von get_cycles() 49
4.11 Implementierung von try_lock() 50

5.1 Boot-Vorgang von Pistachio-embedded 53
5.2 BAT-Abbildungen im User-Model 54
5.3 BAT-Abbildungen im Supervisor-Model 55
5.4 Code-Fragment aus: iguana/server/src/memsection.h 56
5.5 Code-Fragment aus: iguana/server/src/memsection.c 57
5.6 Code-Fragment aus: iguana/server/src/pd.c 57

Tabellenverzeichnis

3.1 Ausgeschlossene Kernel 18
3.2 L4 Projekte 26
3.3 Kernel Übersicht 29

4.1 Freie Para-Virtualisierungen des Linux Kernels für L4 37
4.2 Vergleich der System-Calls bei X.2 und N1 API 48

1. Einleitung

Seit einigen Jahren setzt sich das Open Source Betriebssystem Linux in immer mehr Marktsegmenten durch [Kuri03]. Seit kurzem hat es auch den Markt der Luftfahrt-Industrie erreicht [Brie06]. Daher besteht bei der EADS Deutschland GmbH das Interesse, mehr über die Tauglichkeit von Linux für den Einsatz in der Luftfahrt zu erfahren.

1.1 Zielsetzung der Arbeit

Das Ziel dieser Diplomarbeit ist es, den vernünftigsten Weg zu finden, Linux in der Luftfahrt einzusetzen. Vernünftig ist eine Lösung dann, wenn sie die Anforderungen der Luftfahrt-Industrie erfüllt und nach dem momentanen Stand der Wissenschaft und Technik diese Anforderungen am Besten erfüllt. Die Anforderungen an ein Betriebssystem leiten sich aus den erstmal widersprüchlich erscheinenden Bedürfnissen der Industrie nach Kostensenkung und Erhöhung der Sicherheit ab. Die Integrated Modular Avionic (IMA) ist ein Ansatz, der diesen Spagat bewältigen soll. Neben sicherheitsrelevanten Anforderungen werden bei IMA auch Anforderungen an die Software-Architektur gestellt. Die Anforderungen von IMA werden zur Auswahl eines Lösungsansatzes verwendet. Zur Auswahl stehen in dieser Arbeit nur Open Source Lösungen: Linux mit und ohne Echtzeit-Erweiterungen, der μ-Kernel Minix 3, der Virtual Machine Monitor Xen und Implementierungen der μ-Kernelschnittstelle L4.

Für den „praktischen“ Teil dieser Arbeit hat die EADS Deutschland GmbH das Entwicklungsboard der Laser Range Radar Komponente „Hellas“ zur Verfügung gestellt. Darauf soll ein Linux System gemäß den Anforderungen der Integrated Modular Avionic implementiert werden.

1.2 Gliederung der Arbeit

In Kapitel 2 werden die zentralen Begriffe der Arbeit wie IMA, Echtzeit, Kernel, usw. erläutert und das Themengebiet ein- und abgegrenzt. Diese Vorarbeiten führen in Abschnitt 2.6 zur Definition der IMA-Anforderungen. Im 3. Kapitel folgt eine qualitative Analyse verschiedener Open Source Ansätze anhand der IMA-Anforderungen. Die Analyse ergibt, dass ein L4-Kernel am Besten für den Einsatz in

IMA geeignet ist. In Kapitel 4 wird unter den verfügbaren L4-Projekten das Embedded Systems Framework (ESF) vom National Information and Communications Technology Australia (NICTA) ausgewählt. Das ESF basiert auf dem L4 μ-Kernel Pistachio-embedded und bietet mit Wombat ein para-virtualisiertes Linux an. Abschnitt 4.3 beschreibt die Arbeitsschritte die notwendig sind, um das ESF auf das Hellas-Board zu portieren. Warum dieses Ziel nicht vollständig erreicht werden konnte und wie die Ergebnisse zu bewerten sind, wird in Kapitel 5 beleuchtet.

2. Anforderungen

In den Abschnitten 2.1 bis 2.5 werden die zentralen Begriffe der Arbeit erläutert und das Themengebiet ein- und abgegrenzt. Diese Vorarbeiten werden in Abschnitt 2.6 zur Definition der Anforderungen an ein Betriebssystem für den Einsatz in der Integrated Modular Avionic zusammengeführt.

2.1 Integrated Modular Avionic

Avionik ist ein „[...] Kunstw. aus Aviatik u. Elektronik“, sie umfasst die „Gesamtheit der elektronischen Geräte, die in der Luftfahrt verwendet werden.“ [Bibl01]. Die „Integrated Modular Avionic“ (IMA) ist ein Entwurfsprinzip, das sich seit Mitte der 1980er Jahre bei Ausrüstern der Luftfahrt-Industrie durchgesetzt hat [AEEC97, Morg91, Rush99]. Eine IMA-Architektur integriert mehrere Avionik-Funktionen auf einer Hardware, wie es in Abbildung 2.1 zu sehen ist. Bei einer föderierten Architektur würde dagegen jede Avionik-Funktion auf einer dedizierten Hardware ausgeführt werden.

2.1.1 Ziele

Mit einer IMA-Architektur sollen Verbesserungen der Funktionalität, der Sicherheit und der Kosteneffizienz erreicht werden [Rush99, AEEC97, Morg91]:

Funktionalität: Durch die Integration auf einer Hardware können Funktionen besser aufeinander abgestimmt und es können mehr Funktionen auf gleichem Raum untergebracht werden.

Sicherheit: Die Abstimmung der Funktionalität erleichtert das Arbeiten der Besatzung unter Stress. Darüber hinaus könnten in Zukunft Funktionen bei Hardwarefehlern während des Fluges migriert werden.

Kosteneffizienz: Durch den modularen Aufbau können Komponenten in mehreren Projekten verwendet werden. Dies reduziert den Entwicklungsaufwand und die

Kosten einer Zertifizierung. Der modulare Aufbau mit definierten Schnittstellen ermöglicht es, einzelne Module bei Bedarf auszutauschen. Da der Lebenszyklus eines Flugzeugs in Jahrzehnten bemessen ist, Elektronik-Bauteile aber in der Regel einen Lebenszyklus von Jahren haben, verringert dies die Kosten zur Vorhaltung großer Mengen an Ersatzteilen.

Bei einer IMA-Architektur laufen Funktionen mit verschiedenen Sicherheitsanforderungen auf einem Computersystem ab. Daher muss ein Mechanismus vorhanden sein der die Ausbreitung von Störungen und Fehlern in einer Funktion auf andere Funktionen verhindert. Dies wird durch Partitionierung der Ressourcen erreicht und in Abschnitt 2.2 erläutert.

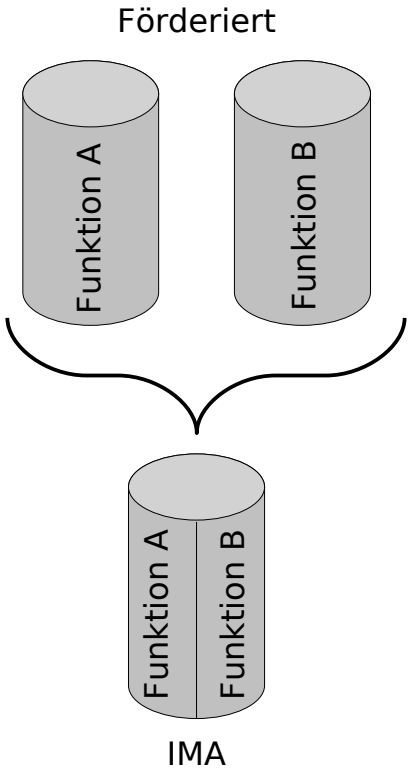

Abbildung 2.1: Förderierte vs. IMA-Architektur

2.1.2 Zertifizierung

Zivile Luftfahrzeuge müssen in Europa von der „Europäische Agentur für Flugsicherheit“ zugelassen werden. Die Zulassung erfolgt nur für das komplette System, nicht für einzelne Komponenten. Die Prüfung eines Systems umfasst jedoch alle seine Subsysteme und Komponenten [RTEU93]. Die Anforderungen zur Zulassung von Software in einem zivilen Luftfahrzeug sind in der Europäischen Union durch die Richtlinie EUROCAE/ED-12B oder in den USA durch RTCA/DO-178B spezifiziert. Beides sind Bezeichnungen für die selbe Richtlinie. ED-12B definiert fünf Grade der Gefährlichkeit von Fehlfunktionen, nach denen zertifiziert werden kann. Je höher die Konsequenzen einer Fehlfunktion sind, desto höher sind die formalen und funktionalen Anforderungen an die Zuverlässigkeit der Software. Da ein Betriebssystem die

unterste Schicht der Software-Architektur bildet, muss es mindestens so zuverlässig sein, wie es von der kritischsten Anwendung gefordert wird.

Für Betriebssysteme hat sich in der zivilen Luftfahrt die Praxis durchgesetzt, den Standard 653-1 der Aeronautical Radio Inc. (ARINC) aus [AEEC03] umzusetzen. Dieser definiert die sprachunabhängige Application/Execution Schnittstelle (APEX):

> „The principal goal of the APEX interface is to provide a general-purpose interface between the application software and the O/S within IMA.“ [AEEC03]

In [AEEC03] wird ein Werkzeug beschrieben, das die Zertifizierung erleichtern soll. Viele Hersteller kommerzieller Betriebssysteme für den Luftfahrtmarkt bieten daher eine APEX Schnittstelle an.

Die Zertifizierung von Betriebssystemen ist nicht Thema dieser Arbeit. Der ARINC Standard 653-1 wird für IMA-Architekturen jedoch als Referenz gesehen, denn er hat sich einerseits in der Praxis bewährt und erfüllt andererseits wichtige Anforderungen an ein IMA-Betriebssystem.

2.1.3 Standards in anderen Industriezweigen

Für den Einsatz von Betriebssystemen in Automobilen gelten ähnlich harte Anforderungen wie in Flugzeugen. Um diese zu erfüllen haben große europäische Automobilhersteller unter Federführung von Siemens den OSEK/VDX Standard entwickelt und als internationalen Standard unter dem Namen ISO 17356 durchgesetzt [OSEK05]. Die Arbeit an OSEK/VDX wurde mittlerweile zu Gunsten der Automotive Open System Architecture (AUTOSAR) eingestellt, die OSEK/VDX in Zukunft ersetzen soll [AUTO05].

Die Anforderungen an die Zuverlässigkeit von Computersystemen sind in der Telekommunikationsindustrie weniger hart, aktuelle Entwicklungen gehen aber in eine ähnliche Richtung. Von Carrier Grade Linux der OSDL werden Anforderungen an ein Linux System spezifiziert, die erfüllt werden müssen, um Quality of Service (QoS) Garantien geben zu können [CGLW06]. Die Entwicklung von Virtualisierungslösungen für Internet-Dienstleistungen verfolgt das gleiche Ziel wie die Partitionierung in IMA. Es geht bei beiden darum, dass sich ein Fehler in einer Anwendung nicht auf die Leistung der anderen Anwendungen auswirken darf. Auf diese Gemeinsamkeiten wird in Abschnitt 2.5 nochmals näher eingegangen.

2.2 Partitionierung

Wie in Abbildung 2.1 zu sehen ist, werden bei IMA mehrere Funktionen auf einer Hardware untergebracht. Zwischen den verschiedenen Funktionen darf es nicht zu einer Fortpflanzung von Fehlern kommen. Sei die Wahrscheinlichkeit für den Ausfall der Funktion f_i gegeben durch $p(f_i)$ und $1 \leq i, j \leq N$. In der Avionik (aber nicht nur dort) wird gefordert, dass das Auftreten von Fehlern unabhängig voneinander ist:

$$\forall i \neq j : p\left(f_i \bigcap f_j\right) = p\left(f_i\right) \cdot p\left(f_j\right)$$

Diese Anforderung wird Partitionierung genannt und in [Rush99] verbal definiert:

> „The behaviour and performance of software in one partition must be unaffected by the software in other partitions.“

Fehler können sich in zwei Dimensionen ausbreiten:

- räumlich: Wenn eine fehlerhafte Funktion Daten oder Code anderer Funktionen überschreibt.
- zeitlich: Wenn eine fehlerhafte Funktion mehr Prozessorzeit in Anspruch nimmt als ihr zusteht.

In beiden Fällen sind andere Funktionen vom Fehler der einen Funktion betroffen. Ein Betriebssystem muss Mechanismen zur Verfügung stellen, die eine Fehlerfortpflanzung in Raum und Zeit verhindern.

2.2.1 Räumliche Partitionierung

Die hier zur Anwendung kommende Definition räumlicher Partitionierung lautet:

> „Spatial partitioning must ensure that software in one partition cannot change the software or private data of another partition (either in memory or in transit) nor command the private devices or actuators of other partions.“ [Rush99]

Es sind zwei Ansätze möglich, räumliche Partitionierung zu erreichen:

Hardware-Partitionierung: Viele Prozessoren sind heutzutage mit einer Memory Management Unit (MMU) ausgestattet. Die MMU stellt einem Betriebssystem Mechanismen zur Verfügung, die den Schutz von Speicherbereichen und den Einsatz virtueller Adressräume ermöglichen. Diese Mechanismen erlauben einem Betriebssystem, Speicherbereiche oder -seiten einem Prozess zuzuordnen und den Zugriffe auf Bereiche anderer Prozesse zu unterbinden. Eine Gefahr für die Partitionierung mittels MMU stellen Bit-Flips dar. Sie werden durch Hoch-Energie-Teilchen, so genannten Single-Event-Upsets (SEU) verursacht. Ein Bit-Flip in der MMU kann dazu führen, dass die Schutzmechanismen durcheinander kommen und die Avionik-Funktionen nicht mehr korrekt arbeiten können. Die Hardware muss daher so ausgelegt sein, dass sie SEU erkennen und beheben kann [Rush99].

Software-Partitionierung: Software Fault Isolation is an approach „to implementing fault isolation within a single address space.“ [WLAG93]. Dem liegt die Idee der Fault Domain zugrunde. Das ist ein Speicherbereich in den Module geladen werden. Linker und/oder Loader stellen sicher, dass kein Speicherzugriff über die Grenze der Fault Domain hinaus gehen kann. Ein Vorteil ist, dass die Kommunikation zwischen verschiedenen Modulen nicht über den Kernel laufen muss und somit schneller ist. Für den Einsatz in der Avionik müsste aber verifiziert werden können, dass zur Laufzeit keine Adresse generiert werden kann, die außerhalb der Fault Domain liegt. Statt Hardware Partitionierung zu ersetzen wird in [Rush99] vorgeschlagen, die Software Ansätze könnten als

Ergänzung dienen. Microsoft Research verwendet einen ähnlichen Mechanismus im Singularity Projekt, die Software Isolated Processes (SIP). Der Ansatz ist rein statisch, d. h. die Sicherheit soll auf Sprach- und Compilerebene erreicht werden. SIP lassen sich in Singularity mit Hardware Isolated Processes kombinieren [AFHH+06].

Ein IMA-System muss eine Hardware- und kann eine Software-Partitionierung implementieren. Ein weiterer Aspekt der Partitionierung ist der Umgang mit Gerätetreibern. Werden Geräte von mehreren Partitionen genutzt, muss sichergestellt werden, dass sie die räumliche Partitionierung nicht verletzen. Es darf z. B. nicht vorkommen, dass eine Anwendung die Daten in einem Cachebuffer überschreibt, die einer anderen Anwendung gehören und noch nicht auf das entsprechende Gerät übertragen wurden. Dies kann z. B. durch eine Management Partition erreicht werden [Rush99].

2.2.2 Zeitliche Partitionierung

Ein Betriebssystem stellt jeder Avionik-Funktion eine bestimmte Menge Zeit zur Verfügung. Es muss sichergestellt werden, dass die Zeiteinteilung des Betriebssystems nicht umgangen werden kann. In dieser Arbeit soll für die Partitionierung der Zeit folgende Definition gelten:

> „Temporal partitioning must ensure that the service received from shared resources by the software in one partition cannot be affected by the software in another partition.“ [Rush99]

Auf einem optimalen IMA-System mit n Avionik-Funktionen und einer Taktfrequenz von s MHz würde jede Funktion so arbeiten können als würde sie auf einem dedizierten System mit $\frac{s}{n}$ MHz ausgeführt. Die wechselweise Zuteilung des Prozessors an die Funktionen wird „Scheduling“ genannt. Scheduling wurde entwickelt, um die Auslastung des Prozessors zu erhöhen und auf Anwendungsebene eine Pseudo-Parallelität zu erreichen. Scheduler können auf der Ebene von Partitionen, Prozessen oder Threads arbeiten. Um diese Entwurfsentscheidung außen vor zu lassen, wird hier von Tasks als Scheduling-Einheit gesprochen.

Pseudo-Parallelität wird dadurch erreicht, dass in regelmäßigen Abständen (Zeitscheiben) ein Timer-Interrupt den Prozessor veranlasst, die Ausführung eines Tasks zu unterbrechen und an einer definierten Stelle im Betriebssystem mit der Ausführung fortzufahren. Dort entscheidet dann der Scheduler ob der alte Task weiterarbeiten darf oder ein anderer Task an der Reihe ist. Betriebssystem und Timer müssen durch Maßnahmen der räumlichen Partitionierung vor Manipulationen durch User-Tasks geschützt werden. Ansonsten könnten Manipulationen am Scheduler oder Timer die zeitliche Partitionierung aushebeln.

In [Rush99] werden zwei Scheduling-Ansätze unterschieden:

Statisches Scheduling: Jeder Task bekommt CPU-Zeit nach einem festen Zyklus zugeteilt. Die Zuteilung findet beim Entwurf des Systems statt. Der bekannteste Scheduling-Algorithmus hierfür ist Round-Robin. Beim statischen Scheduling wird kein Task durch einen anderen unterbrochen.

Dynamisches Scheduling: Es wird zur Laufzeit entschieden welcher Task als nächstes die CPU zugeteilt bekommt. Die Entscheidung wird aufgrund von Prioritäten[1] getroffen. Damit nicht ein fehlerhafter Task mit hoher Priorität das ganze System blockieren kann, wird von ebd. vorgeschlagen, Zeit-Quotas einzuführen. Jeder Task kann dann nur maximal soviel Zeit verbrauchen, wie er Quota hat.

Eine weitere Unterteilung der dynamischen Scheduling-Algorithmen wird in [Chen02] vorgenommen:

- Scheduler mit festen Prioritäten: Rate-Monotonic- und Deadline-Monotonic-Scheduling
- Scheduler mit dynamischen Prioritäten: Earliest-Deadline-First- und Least-Laxity-First-Scheduling

Die Eigenschaften dieser Echtzeit-Scheduler werden in Abschnitt 2.3 diskutiert.

In [AEEC03] kommt ein zweistufiger Scheduler zum Einsatz: Ein statischer Scheduler wird für die Partitionen verwendet. Dies ermöglicht die mathematische Überprüfung der „Schedulabilty“ wie sie in [Chen02] vorstellt wird. Für eine gegebene Menge Tasks und einen bestimmten Scheduling-Algorithmus kann überprüft werden, ob die Zeitbedingungen immer eingehalten werden können.

Innerhalb der Partition kann sowohl ein statischer als auch dynamischer Scheduler (oder keiner) zum Einsatz kommen.

Ein weiteres Problem der zeitlichen Partitionierung sind Unterbrechungen (Interrupts) durch Peripheriegeräte. Ein Gerät, das zu Partition A gehört, sollte den Kernel nicht unterbrechen können, wenn Partition B ausgeführt wird. Sonst würde A Zeit von B „stehlen“. Die zeitliche Partitionierung kann auf zweierlei Weise sicher gestellt werden:

1. Eine Unterbrechung durch Geräte nur zulassen, wenn sie der aktuellen Partition zugeteilt sind.
2. Den Gerätestatus abfragen (Polling) statt unterbrechen (Interrupting) zu lassen.

2.3 Echtzeit

Aus dem Einsatzgebiet in Luftfahrzeugen ergibt sich für viele Anwendungen die Notwendigkeit, innerhalb gegebener Zeitschranken reagieren zu müssen. Dies betrifft nicht nur kritische Systeme wie den Autopiloten oder das Hinderniswarnsystem, sondern auch Multimedia-Anwendungen. Allerdings unterscheiden sich die zeitlichen Anforderungen einer Multimedia-Anwendung von einer Flugsteuerung doch erheblich. Daher geht es nun darum, Begriffe zu definieren, mit denen sich diese Unterschiede exakt beschreiben lassen.

Nach [Stan88] sind die zwei Hauptanforderungen an Echtzeit-Systeme:

[1] Wobei Deadlines und Zykluszeiten als Prioritäten interpretiert werden können.

1. „In real-time computing the correctness of the system depends not only on the logical result of the computation but also on the time at which the results are produced.“

2. „Predictability, not speed, is the foremost goal in real-time-system design.“

Daher wird ein Echtzeit-System von [TiPe05] zusammenfassend definiert als:

> „A real-time system responds in a (timely) predictable way to all individual unpredictable external stimuli arrivals.“

Zur einfacheren Definition der Echtzeit-Anforderungen nehme man folgendes semiformale Modell eines Computersystems an: $C = (\Omega, \Phi, \Pi, s)$, wobei

- Ω der Phasenraum über der Zeit $T = \{0; 1; 2; \ldots\}$ ist, in der sich ein Computersystem befinden kann. Das umfasst den Zustand des Speichers, der Register und der Peripheriegeräte.

- $\Phi : \Omega \times T \to \Omega \times T$ eine Funktion über der Zustandsmenge Ω und der Zeit T ist, die bei jedem Taktschritt den aktuellen Zustand auswertet und das System in einen neuen Zustand überführt:

$$\begin{pmatrix} z_i \\ t \end{pmatrix} \mapsto \begin{pmatrix} z_j \\ t+1 \end{pmatrix}$$

- $\Pi : \Omega \to \Omega$ eine Funktion ist, die Veränderungen des Zustands durch externe Ereignisse (z. B. einen Tastendruck) modelliert. Externe Ereignisse können auch zwischen zwei Taktschritten auftreten und den Zustand verändern.

- s der Startzustand nach Einschalten des Rechners, d. h. zum Zeitpunkt $t = 0$ ist.

Der aktuelle Zustand wird in jedem Zeitschritt (Takt) von Φ neu berechnet. Er kann jedoch auch zwischen zwei Zeitpunkten von Π verändert werden, wenn ein externes Ereignis aufgetreten ist.

Ein Pfad sei eine Trajektorie im Phasenraum: $p_{ij} = z_i \to \ldots \to z_j$. Startet der Pfad im Zustand z_i zur Zeit $t(z_i)$ und erreicht Zustand z_j erstmalig zur Zeit $t(z_j)$, dann ergibt sich die Dauer des Pfades als $\mid p_{ij} \mid = t(z_j) - t(z_i)$.

In [Tane02] wird zwischen weicher und harter Echtzeit unterschieden. Als Kompromiss zwischen beiden wird in [TiPe05] die stabile Echtzeit eingeführt. Für die Analyse der Anforderungen genügt die Zweiteilung in harte und weiche Echtzeit.

Ein Task x sei durch $(q_e, t_e)^T$ definiert, wobei $q_e \in \Omega$ der Zielzustand und $t_e \in T$ die Deadline ist. Sei nun X die Menge der Tasks eines Systems, dann muss gelten:

Weiche Echtzeit: Im Durchschnitt über alle $x = (q_e, t_e)^T \in X$ muss gelten: $(t_e - t_a) \geq \mid p_{ae} \mid$. Eine gelegentliche Verletzung der Deadline ist erlaubt. Multimedia Streaming ist z. B. eine Anwendung mit weichen Echtzeit-Bedingungen.

Harte Echtzeit: $\forall q_a \in \Omega, \forall x \in X : t(q_a) + \mid p_{ae} \mid \leq t_e$.
D. h. es wird keine Überschreitung der Deadline toleriert. Industrieroboter haben z. B. harte Echtzeit-Bedingungen.

Der zeitliche Determinismus, den Echtzeit-Systeme fordern, wird von Π gefährdet. Π modelliert externe Ereignisse, die den Zustand des Systems ändern können. Z. B. Sensoren, die über eine Interruptleitung das Vorhandensein neuer Messdaten signalisieren. Im schlimmsten Fall kann Π das System so lange von seinem Zielzustand ablenken, bis die Deadline verstrichen ist. Dieses Problem lässt sich durch Mechanismen der Partitionierung lösen, wie sie oben beschrieben wurden.

Der Begriff der „Zeit“ wurde bisher außen vor gelassen. Echtzeit bezieht sich immer auf die Zeit, die außerhalb eines Computersystems auf der Erde vergeht. Die Messung der Zeit bezieht sich also auf externe und nicht auf interne Zeitquellen eines Echtzeit-Systems [Chen02]. Die Temps Atomique International (TAI) oder die Universal Coordinated Time (UTC) sind z. B. externe Zeitquellen, wohingegen der Prozessortakt oder ein Simulationsschritt interne Zeitquellen wären.

Beim Entwurf von Echtzeit-Systemen kann man zwei Ansätze ausmachen:

Zeitunbewusste Systeme: Die Echtzeit-Fähigkeiten werden durch die Architektur vorgegeben. Interrupts und Tasks werden mit Prioritäten versehen und von einen Scheduling-Algorithmus dem Prozessor zugeteilt. „The challenge in dynamic scheduling is to allocate priorities to tasks in such a way that overall system behavior is predictable and all deadlines are satisfied.“ [Rush99] Dieser Ansatz hat den Nachteil, dass sich das System der Zeit nicht bewusst ist. Wenn eine Deadline wider Erwarten nicht eingehalten wird, bemerkt es das System nicht. Eine Kontrolle kann dann nur durch eine externe Instanz, z. B. einem Watchdog Timer, vorgenommen werden.

Zeitbewusste Systeme: Eine andere Möglichkeit besteht darin, Algorithmen zu verwenden, die sich der Zeit bewusst sind. Earliest-Deadline-First (EDF) oder Least-Laxity-First (LLF) sind z. B. Scheduling-Algorithmen, die mit konkreten Zeitangaben arbeiten. Diese Systeme können intern überprüfen, ob eine Deadline überschritten wurde und dementsprechend reagieren. Mit einem Schedulability-Test wie er in [Chen02] beschrieben wird, ist es möglich, für eine gegebene Menge von Tasks die Erfüllbarkeit der Echtzeit-Bedingungen zu bestimmen. Es wird auch gezeigt, dass EDF und LLF optimale Scheduler[2] für Systeme mit einem Prozessor sind. Systeme mit einfachen Prioritäten haben dagegen den Vorteil, dass weder die Deadline berechnet, noch die Laufzeit eines Tasks abgeschätzt werden muss.

2.4 Kernel vs. Betriebssystem

Im allgemeinen Sprachgebrauch werden Kernel und Betriebssystem synonym verwendet. Für die Analyse von Betriebssystemen ist jedoch die Trennung und genaue Definition der Begriffe notwendig:

[2] „An optimal scheduler is one which might fail to meet a deadline of a task only if no other scheduler can.“ [Chen02]

Kernel: „Traditionally, the word 'kernel' is used to denote the part of the operating system that is mandadory and common to all other software.“ [Lied95] Dieser kleinste gemeinsame Nenner umfasst die Verwaltung der räumlichen und zeitlichen Ressourcen. Dazu benötigt der Kernel im Gegensatz zu anderen Teilen des Systems die volle Kontrolle über die Hardware. Er wird daher im privilegierten Modus (Ring) des Prozessors ausgeführt. Es gibt zwei Entwurfsansätze für Kernel, die sich in der Frage unterscheiden welche Funktionen in den Kernel gehören:

μ-Kernel: Ein μ-Kernel minimiert die Anzahl der Funktionen, die im privilegierten Modus ausgeführt werden: „[...] a concept is tolerated inside the μ-kernel only if moving it outside the kernel [...] would prevent the implementation of the system's required functionality.“ [Lied95] Dies umfasst mindestens Scheduling, Interrupt Handling und einen Mechanismus zur Verwaltung der Daten, wie z. B. eine Speicherverwaltung. Gerätetreiber können in der Regel ohne vollen Zugriff auf die Hardware implementiert werden und gehören daher nicht in einen μ-Kernel. Beispiele für μ-Kernel sind C5, Mach oder L4Ka::Pistachio.

Monolithischer Kernel: Dieser Ansatz wird in [Tane02] auch als „große Masse“ bezeichnet. Ein monolithischer Kernel enthält neben den Funktionen des μ-Kernels auch Gerätetreiber, Dateisysteme, Netzwerkprotokolle und vieles mehr. Die Code-Basis, die vollen Zugriff auf die Hardware hat, ist hierbei deutlich größer. Beispiele monolithischer Kernel wären Windows XP, Linux oder FreeBSD.

Betriebssystem: Der Kernel ist ein Teil des Betriebssystems. Neben der Verwaltung von Ressourcen, die vorwiegend im Kernel stattfindet, bietet das Betriebssystem eine Abstraktion der Hardware an. Diese Abstraktion wird über Dienste zur Verfügung gestellt. Die Dienste können von Anwendungen durch so genannte System-Calls in Anspruch genommen werden. System-Calls sind im Application Binary Interface (ABI) des Betriebssystems spezifiziert. Das ABI legt für eine bestimmte Hardware-Architektur fest, auf welche Weise (Befehle, Register) ein Dienst in Anspruch genommen werden kann. Es ist von der Programmiersprache unabhängig. Das Application Programming Interface (API) ist dagegen hardwareunabhängig und sprachabhängig. Ein API stellt die Dienste des Betriebssystems durch Funktionsaufrufe in einer Programmiersprache zur Verfügung.

Für den Einsatz in IMA sollte ein Betriebssystem modular aufgebaut und der Kernel durch eine Schnittstelle vom Rest getrennt sein. Das Betriebssystem wäre somit in Komponenten zerlegt, die sich austauschen lassen. So kann z. B. der Kernel ersetzt werden, ohne dass die Anwendungen neu geschrieben werden müssen. Oder der gleiche Kernel kann in Betriebssystemen mit unterschiedlichen APIs zum Einsatz kommen. Der Kernel sollte wiederum in architekturabhängigen und generischen Code getrennt sein. Dies erleichtert die Portierung des Betriebssystems auf eine neue Hardware.

In [Benn03] wird eine generische Software-Architektur für IMA vorgestellt. Sie ist in Abbildung 2.2 zu sehen und definiert drei Schnittstellen:

1. Das *OS Service Interface* entspricht dem üblichen Verständnis eines API. Es ist die standardisierte Schnittstelle zwischen Betriebssystem und Partitionen.

2. Das *IMA Kernel Interface* ist eine standardisierte Schnittstelle zwischen Kernel und Betriebssystem.

3. Das *Hardware Interface* in eine architekturabhängige Schnittstelle zwischen Hardware und Kernel.

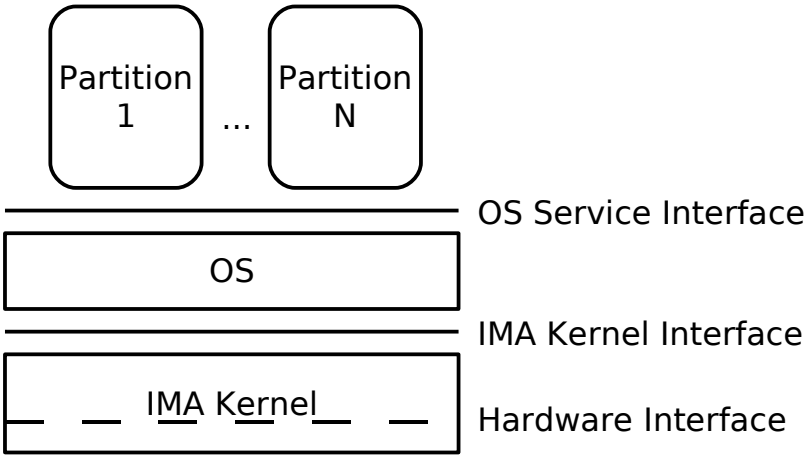

Abbildung 2.2: Generische IMA-Architektur nach [Benn03]

Die generische IMA-Architektur legt im Gegensatz zum APEX-Modell in [AEEC03] keine Details zur Implementierung fest. Es wurde als abstraktes Modell entworfen, um die Anforderungen an einen IMA-Kernel zu bestimmen. Sie soll auch den Überlegungen zur Software-Architektur in dieser Arbeit zu Grunde liegen.

Beim Entwurf von Betriebssystem und Kernel sollte allgemein die Policy vom Mechanismus getrennt werden [Tane02]:

Mechanismus: Ein Mechanismus ist ein Algorithmus dessen Verhalten durch Parameter bestimmt wird. Ein prioritätsbasierter Scheduler wäre z. B. ein Mechanismus.

Policy: Die Policy legt die Parameter des Algorithmus fest und bestimmt somit sein Verhalten. Die Vergabe der Prioritäten für Tasks bestimmt z. B. die Policy des Schedulers.

Besonders IMA-Kernel sollten die Trennung von Mechanismus und Policy berücksichtigen. Denn so lassen sich mit einem Kernel verschiedene Dienstschichten realisieren, die je nach Anwendungsgebiet unterschiedliche Policies verfolgen.

2.5 Virtualisierung

Aus Kostengründen werden bei einer IMA-Architektur Betriebssystem und Kernel in Komponenten mit definierten Schnittstellen zerlegt. So lassen sich Komponenten austauschen und mehrfach verwenden. Die Partitionierung soll sicherstellen, dass

Fehler unabhängig voneinander auftreten. Der modulare Aufbau und die Partitionierung der IMA-Architektur werden in [AEEC03] dazu verwendet, Betriebssysteme innerhalb einer Partition zu realisieren. Diese Betriebssysteme können nicht mehr unbeschränkt auf die Hardware zugreifen, sondern müssen die APEX-Dienste nutzen. Die Betriebssysteme werden also in einer virtuellen Maschine ausgeführt.

Wie man an [DoMa03, BDFH+03] sieht, ist Virtualisierung ein Thema aktueller Forschung, auch wenn die Ursprünge in die späten 70er Jahre des 20. Jahrhunderts zurück gehen [Tane02]. Wenn in jeder Partition eine vollständig unabhängige virtuelle Maschine ausgeführt wird und die Performanz einer virtuellen Maschine nicht von der Performanz der anderen abhängt, dann ist die Partitionierung optimal.

Zur Implementierung der Virtualisierung gibt es verschiedene Ansätze:

Einfache Virtualisierung: Innerhalb eines Betriebssystems wird eine isolierte Umgebung geschaffen, in der dann eine Anwendung z. B. die Instanz eines Servers ausgeführt wird. Der Rest des Systems ist zwar vor Fehlverhalten der isolierten Anwendung geschützt, nicht aber umgekehrt.

Volle Virtualisierung: Ein Virtual-Machine-Monitor oder Hypervisor führt mehrere auch unterschiedliche OS Instanzen parallel aus und sorgt dafür, dass die Ressourcen sauber getrennt bleiben. Ein Vorteil ist, dass die Gast-OS nicht angepasst werden müssen. Dafür müssen Maßnahmen getroffen werden, dass das Gast-OS den Hypervisor nicht umgeht. Das kann einerseits durch laden von speziellen Treibern oder Modulen erreicht werden, andererseits durch umschreiben des Binär-Images zur Ladezeit.

Para-Virtualisierung: Anstatt binäre Maschinenbefehle umzuschreiben, wird bei der Para-Virtualisierung das Gast-OS auf Ebene der Programmiersprache an den Hypervisor angepasst. Der Hypervisor oder Virtual-Machine-Monitor läuft dabei im privilegierten Modus des Prozessors, die Gast-OS werden dagegen in einem nicht privilegierten Modus ausgeführt. Somit kann sichergestellt werden, dass sich die verschiedenen Gast-OS nicht gegenseitig stören. Beispiele für diese Technik finden sich in [WhSG02, BDFH+03, Kais04].

Pre-Virtualisierung: Die Pre-Virtualisierung ist eine Para-Virtualisierung mit dem Ziel, den Aufwand für die Anpassung eines Betriebssystems zu senken. Die Virtualisierung wird dabei vom Compiler vorgenommen. Eine zusätzliche Verarbeitungsstufe des Compilers schreibt privilegierte Hardware-Zugriffe in Aufrufe der virtuellen Hardware um. [LUCC+05]

Bei der Para-Virtualisierung kommt ein Mikro- oder Nano-Kernel[3] als Hypervisor zum Einsatz. Der Hypervisor – oder auch Virtual-Machine-Monitor (VMM) – stellt die virtuelle Umgebung für Gast-Betriebssysteme bereit. Er kümmert sich um die Organisation der Ressourcen und sorgt für eine Isolation der Gast-Systeme. Die virtuelle Umgebung kann entweder eine Teilmenge der Hardware sein oder eine vollständige Abstraktion davon. Xen [BDFH+03] bietet z. B. eine Teilmenge der aktuellen Hardware als virtuelle Maschine an. Die Kernelschnittstelle L4 [L4Ka04, ERTa05] bietet dagegen eine abstrakte Hardware für para-virtualisierte Systeme an. Beide Ansätze werden in Kapitel 3 genauer analysiert.

[3] Die Begriffe Mikro- und Nano-Kernel werden in dieser Arbeit synonym verwendet.

2.6 Anforderungen an einen IMA-Kernel

Ausgehend von der generischen IMA-Architektur in Abbildung 2.2 werden in [Benn03] fünf Anforderungen an einen IMA-Kernel spezifiziert:

Adaptability: Ein IMA-Kernel soll, wie in Abbildung 2.2 zu sehen ist, in zwei Richtungen anpassungsfähig sein:

- Das *OS Service Interface* soll den Einsatz veränderter Paradigmen oder Techniken bei der Anwendungsentwicklung mit minimalem Änderungsaufwand unterstützen.
- Das *Hardware Interface* soll es ermöglichen, dass der IMA-Kernel mit geringem Aufwand auf eine neue Implementierung der Von-Neumann-Architektur übertragen werden kann.

Standardisation: Eine standardisierte Kernel Schnittstelle wie in Abbildung 2.2 soll die unveränderte Verwendung der OS-Dienstschichten auf neuen Implementierungen des Kernels garantieren.

Portability: Ein IMA-Kernel sollte ohne großen Aufwand auf eine neue Plattform portiert werden können.

Partitioning Support: Von zentraler Bedeutung für die Sicherheit in Luftfahrzeugen ist die räumliche und zeitliche Partitionierung von Anwendungen. Ein IMA-Kernel muss dies unterstützen.

Real-Time Performance: Ein IMA-Kernel muss Echtzeit Scheduling anbieten oder zumindest die Implementierung auf Anwendungsebene unterstützen. Außerdem muss es möglich sein, eine Analyse der Worst-Case-Execution-Time durchzuführen.

Die Anforderungen in [Benn03] sind an der Software-Architektur ausgerichtet und haben die Kosteneffizienz zum Ziel. Das wird dadurch deutlich, dass sich drei von fünf Anforderungen ihm widmen: „Portability“ senkt die Kosten einer Portierung, „Adaptability“ senkt die Kosten beim Einsatz neuer Technologien und „Standardisation“ senkt die Kosten durch Austauschbarkeit von Komponenten. In dieser Arbeit soll der Schwerpunkt jedoch stärker auf der Zuverlässigkeit des Systems liegen und nicht bei der Kosteneffizienz. Daher werden „Portability“, „Standardisation“ und „Adaptability“ in die Anforderungen nach Modularität und offenen Standards zusammengefasst. Zusätzlich wird die Anforderung nach einer minimalen Trusted Computer Base (TCB) gestellt, die die Zuverlässigkeit erhöhen soll. Es ergibt sich also folgender Anforderungskatalog:

2.6.1 Echtzeit

Der Kernel muss ein deterministisches Zeitverhalten haben, damit eine Analyse der Worst-Case-Execution-Time durchgeführt werden kann. Dies ist für die Zertifizierung und den Einsatz in kritischen Bereichen der Luftfahrt erforderlich. Ein wichtiger Baustein ist dabei das Scheduling-Verfahren. Ein zeitbewusster Scheduler ist einem reinen Prioritäts-Scheduler vorzuziehen. Der Scheduler darf im Kernel nur den Mechanismus implementieren und muss die Policy dem Betriebssystem überlassen. Wünschenswert wäre, wenn den Partitionen Mechanismen zur Verfügung stünden, die es ermöglichen, weitere Scheduler zu implementieren.

2.6.2 Partitionierung

Der Kernel muss Schutzmechanismen der CPU nutzen, um die Partitionen räumlich und zeitlich zu isolieren. Keine Partition darf durch Absicht oder Zufall eine andere Partition stören oder den Kernel umgehen können. Prozessoren von Intel, AMD, IBM und anderen bieten heute Schutzfunktionen für Speicherbereiche durch eine Memory Management Unit (MMU) an. Maschinenbefehle werden bei der Intel- und PowerPC-Architektur Prozessor Modi zugeteilt. So ist eine Manipulation der MMU-Register nur im privilegierten Modus möglich. Solche Mechanismen muss ein IMA-Kernel unterstützen, wenn sie vorhanden sind.

Eine Unterbrechung durch Geräte, die nicht zur aktiven Partition gehören, soll vermieden oder zumindest durch Quotas eingeschränkt werden. Jede Maßnahme zur Reduzierung der Auswirkungen fehlerhafter Geräte sollte ausgenutzt werden. Nur wenn die Auswirkung externer Ereignisse Π eingeschränkt wird, ist eine Analyse der Worst-Case-Execution-Time möglich.

2.6.3 Trusted Computing Base

Aus der Forderung nach Echtzeit und nach Senkung der Kosten ergibt sich implizit die Forderung nach einer minimalen Trusted Computer Base. Denn die Analyse der Worst-Case-Execution-Time ist aufwändig und daher teuer. Weniger Funktionalität bedeutet dagegen weniger Ausführungs- und Datenpfade. Dies wiederum reduziert in der Regel[4] die Lines-of-Code (LoC), die untersucht werden müssen.

Bei Minix 3 wurde eine minimale Trusted Computing Base mit der Begründung zum Entwurfsziel, dass weniger Code-Zeilen auch weniger Bugs enthalten. [HBGH+06] Dieser Argumentation soll hier Folge geleistet werden. Gerade im Hinblick auf die Zuverlässigkeit und den Aufwand einer Zertifizierung ist ein kleiner Kernel für IMA wünschenswert.

2.6.4 Offene Standards

Standardisierte Schnittstellen sind wichtig, um Komponenten austauschen zu können. Dies spart Entwicklungsaufwand und damit Kosten. Offene Standards ermöglichen, dass mehrere Hersteller eine Komponente anbieten können. Damit ist die Verfügbarkeit von Komponenten über einen langen Zeitraum hinweg gesichert. Da sich die Lebensdauer einer Avionik-Anwendung über Jahrzehnte erstreckt, ist es eine wichtige Anforderung von IMA.

Eine „Adaptability“ an veränderte Anforderungen von Seiten der Anwendung ist im Rahmen eines Community Prozesses gegeben. Dieser kann von Firmen und/oder Forschungsprojekten getragen werden. Eine Beteiligung verschiedener Interessengruppen garantiert auch eine entsprechende Abstraktion der Schnittstelle.

Offene Standards decken also die Anforderungen von „Standardisation“ und „Adaptability“ ab.

[4] Abkürzende Schreibweisen wie sie vor allem in C und C++ verwendet werden, reduzieren zwar den Code, nicht aber seine Komplexität.

2.6.5 Modularität

Aus der generischen IMA-Architektur in Abbildung 2.2 folgt die Forderung nach einer modulare Architektur mit zwei definierten Schnittstellen:

- Das *OS Service Interface* spezifiziert den Zugriff auf Ressourcen und Funktionen des Betriebssystems.
- Das *Kernel Interface* spezifiziert Funktionen, über die Dienste des Kernels angefordert werden können, sowie öffentliche Datenstrukturen des Kernels.

Durch definierte Schnittstellen wird es z. B. möglich, Betriebssystem und/oder Kernel durch neue Implementierungen zu ersetzen, ohne dass die Anwendungen geändert werden müssten.

Die Schnittstelle zur Hardware kann zwar nicht standardisiert aber bis zu einem gewissen Grad abstrahiert werden. So soll ein IMA-Kernel eine Abstraktion der Hardware zur Verfügung stellen, die den Aufwand einer Portierung auf eine neue Architektur oder Plattform verringert. Die Forderung nach „Adaptability“ an neue Hardware Technologien entspricht der Anpassung an eine neue Architektur. Die Forderung nach „Portability“ entspricht der Anpassung an eine neue Plattform. Beides ist hier unter Modularität zusammengefasst.

2.7 Zusammenfassung

Das Entwurfsprinzip der Integrated Modular Avionic (IMA) hat zum Ziel, die Funktionalität und Sicherheit von Flugzeugen zu erhöhen und dabei Kosten zu sparen. Dies wird durch die Integration mehrerer Avionik-Funktionen in einem modular aufgebauten System erreicht. Aus der Zielsetzung und dem Einsatzgebiet von IMA ergeben sich folgende fünf Anforderungen an einen Kernel:

- Echtzeit: Es können Systeme mit deterministischem Zeitverhalten implementiert werden.
- Partitionierung: Die räumliche und zeitliche Ausbreitung von Fehlern wird verhindert.
- Trusted Computing Base: Die Anzahl LoC ist minimal.
- Offene Standards: Die Schnittstellen sind frei verfügbar und ohne Bindung an einen Hersteller.
- Modularität: Definierte Schnittstellen zwischen Kernel, Betriebssystem und Partition ermöglichen, Komponenten auszutauschen und wiederzuverwenden.

3. Analyse

In einer IMA-Architektur muss der Kernel austauschbar sein. Der Kernel ist, wie der Name schon sagt, das Kernstück eines Betriebssystems. Von ihm hängen die wesentlichen Eigenschaften des Systems ab. Daher kommt der Auswahl eines Kernel besondere Bedeutung zu. Anhand der IMA-Anforderungen aus Kapitel 2 werden in diesem Kapitel aktuelle Kernel analysiert. Der Kernel mit den besten Voraussetzungen für den Einsatz in IMA dient als Ausgangspunkt für die Implementierung, die in Kapitel 4 beschrieben wird.

Aufgrund der Vielzahl verfügbarer Kernel muss die Auswahl für diese Arbeit eingeschränkt werden:

1. Der Kernel muss unter einer Open Source Lizenz verfügbar sein. Diese Einschränkung ist aus drei Gründen sinnvoll:
   (a) Die Beschaffung von Informationen und Quellcode ist einfach.
   (b) Keine Non-Disclosure-Agreements (NDA) schränken die Veröffentlichung ein.
   (c) Da das Ziel-Betriebssystem, nämlich Linux, ein Open Source System ist, liegt die Auswahl von Open Source Lösungen nahe.
2. Aus der Zielsetzung der Arbeit ergibt sich, dass nur Kernel in Frage kommen, die es ermöglichen, ein Linux Betriebssystem auszuführen. Man könnte natürlich Linux auf jeden beliebigen Kernel portieren, dies ist jedoch ein zeitaufwändiges Unternehmen und im Rahmen dieser Arbeit nicht möglich.
3. In [Benn03] wurden bereits verschiedene Open Source Kernel auf ihre Tauglichkeit für IMA untersucht. Damals wurde die Kernel-Architektur L4 ausgewählt, da sie den IMA-Voraussetzungen am nächsten kommt. Seit der Veröffentlichung im Jahr 2003 sind neue L4-Projekte entstanden und der der Linux-Kernel wurde weiterentwickelt. Außerdem kamen mit Minix 3 und Xen zwei neue Open Source Systeme hinzu. Außerdem ist für diese Arbeit Linux als Zielsystem vorgegeben und der Schwerpunkt der Anforderungen hat sich leicht

verschoben. Daher wird in dieser Arbeit eine eigenständige Analyse durchgeführt. Die Auswahl in [Benn03] dient jedoch als Ausgangspunkt, denn viele der beschriebenen Kernel wurden nicht weiterentwickelt.

Name	**Ausschluss-Kriterium**
xBSD	Ausschluss durch [Benn03]
Mach	Ausschluss durch [Benn03] und [HMJS+97]
C5	Ausschluss durch [Benn03] unter dem Namen „Chord OS“. Es wird mittlerweile als Open Source Projekt von VirtualLogix weiter entwickelt. Die Architektur hat sich aber nicht geändert.
RTEMS	RTEMS besitzt nur einen Adressraum und bietet keinerlei Partitionierung. Außerdem ist kein Linux verfügbar.
MicroC/ OS-II	Der Source Code für diesen Mikro-Kernel kann durch den Kauf des gleichnamigen Buches [Labr98] erworben werden. Die Nutzung ist für nicht kommerzielle Zwecke frei. Kommerzielle Entwickler müssen eine Lizenz von Micrium erwerben. In einem engeren Sinne ist der Kernel nicht Open Source. Er soll hier aber trotzdem Erwähnung finden, da er in der Wissenschaft eingesetzt werden kann.
VxWorks	Keine Open Source Lizenz
QNX	Keine Open Source Lizenz
Integrity	Keine Open Source Lizenz
PikeOS	Keine Open Source Lizenz

Tabelle 3.1: Ausgeschlossene Kernel

3.1 Methodik

In [MeTa03] wird eine quantitative Methode vorgestellt, Echtzeit-Betriebssysteme entsprechend den Anforderungen eines (Raumfahrt-)Projekts auszuwählen. Dazu wird aus einem Katalog von Anforderungen ein Fragebogen abgeleitet und an die Hersteller von Betriebssystemen geschickt. Dies setzt voraus, dass es einen Ansprechpartner gibt auf den man sich verlassen kann. Bei Open Source Projekten ist diese Herangehensweise leider unbrauchbar. Denn es besteht die Gefahr, dass entweder keiner oder mehrere Entwickler den Fragebogen beantworten – und dies evtl. noch widersprüchlich. Daher wird eine quantitative Erhebung nach sozialwissenschaftlichem Vorbild in dieser Arbeit nicht verfolgt. Statt dessen sollen die Anforderungen anhand von wissenschaftlichen Veröffentlichungen, der technischen Dokumentation und einem ersten Blick in den Quellcode bewertet werden.

Die einfachste Metrik zur Messung der Komplexität von Software ist, die „Lines-of-Code“ (LoC) zu zählen. Um bei der Bewertung der Trusted Computing Base (TCB) eine grobe Abschätzung der Größenordnung verschiedener Kernel zu bekommen, wird in dieser Arbeit folgende triviale LoC-Metrik verwendet:

```
find . -regex '.*\.\(c\|cc\)' -print0 | xargs -0 cat | wc -l
```

Diese Metrik lässt eine unabhängige Einschätzung des TCB-Umfangs zu und stellt die Vergleichbarkeit verschiedener Projekte her, die durch Angaben in Publikationen nicht gegeben wäre.

3.2 Kernel

Eine Auswahl der Kernel, die aufgrund obiger Einschränkungen ausgeschlossen wurden, sind in Tabelle 3.1 zu finden. Diese Aufzählung erhebt keinen Anspruch auf Vollständigkeit.

3.2.1 Linux

Der einfachste Weg zu einem Linux System, führt über einen Linux Kernel. Linux wurde als Mehrbenutzer-UNIX-System von Linus Torvalds entwickelt. Als solches ist es nicht entworfen worden, Echtzeit-Garantien einzuhalten. Vielmehr geht es bei UNIX-Systemen um die faire Verteilung der Ressourcen.

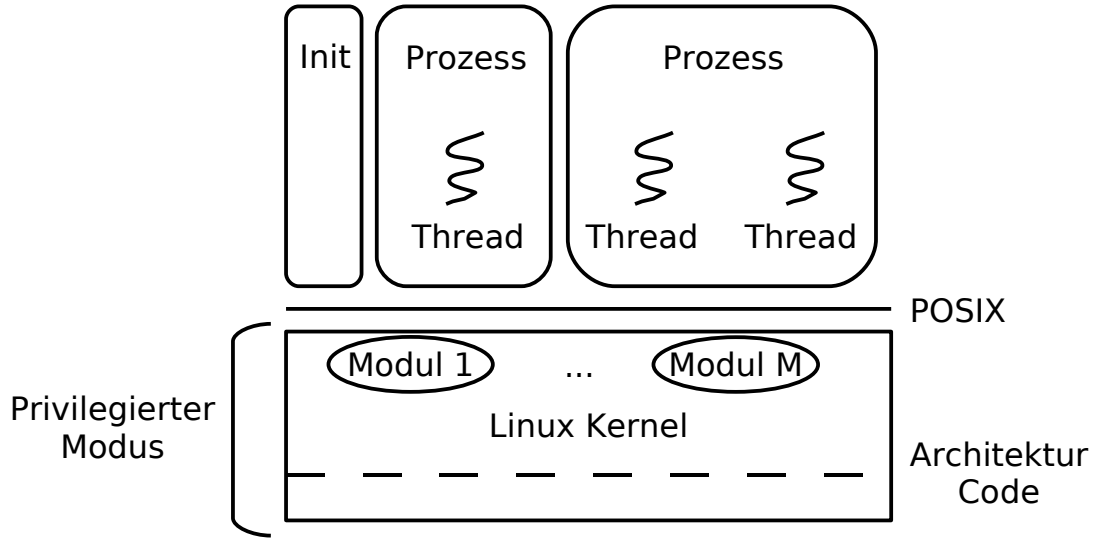

Abbildung 3.1: Linux Architektur

Linux ist ein monolithischer Kernel, wie in Abbildung 3.1 zu sehen ist. Neben Scheduler und Interrupt Handler werden u. a. auch Gerätetreiber und Protokoll-Stacks im privilegierten Modus ausgeführt. Linux implementiert ISO POSIX 2003 und Teile der IEEE POSIX 1003.4 Echtzeit-Erweiterungen:

- `mlock(*addr, len)` erlaubt Speicherseiten vom Swapping ausnehmen
- `sched_setscheduler( pid, policy, *param)` erlaubt die Auswahl zweier Echtzeit-Scheduling Policies: First-In-First-Out und Round-Robin Scheduling. Es wird in [Tane02] darauf hingewiesen, dass beide Scheduler zwar eine höhere Priorität besitzen, aber nicht echtzeitfähig wären. Denn es können weder Deadlines gesetzt noch zeitliche Garantien gegeben werden.

Es haben sich zwei Ansätze durchgesetzt, Linux echtzeitfähig zu machen:

Patching: Mit der Linux Kernel Version 2.6.18 wurden einige Echtzeit-Patches in den Standard Kernel aufgenommen [Corb06]. Es bleibt den Distributionen vorbehalten, diese Eigenschaften zu aktivieren. Anbieter wie Monta Vista oder TimeSys bieten darüber hinaus Kernel an, die noch weitere Echtzeit-Patches enthalten. Sie zielen vor allem darauf ab:

1. die Reaktionszeit (Latenz) des Systems auf Interrupts zu verringern.
2. den Anteil preemptiven Kernel-Codes zu erhöhen.

Durch diese Maßnahmen wird der Linux-Kernel für weiche Echtzeit-Anwendungen tauglich gemacht.

Dual-Kernel: Beim Dual-Kernel Ansatz von RTLinux [Bara97] übernimmt ein μ-Kernel die Kontrolle über Interrupts und Timer. Linux wird nur dann ausgeführt, wenn keine Echtzeit-Tasks lauffähig sind. Die Architektur ist in Abbildung 3.2 dargestellt. Einen ähnlichen Ansatz verfolgt das RTAI/Adeos Projekt. Die Adeos Interrupt-Pipe [Yagh01] wird als Kernelmodul geladen und übernimmt die Kontrolle über das Interrupt-Handling. RTAI implementiert einen eigenen Scheduler, der dem Linux-Scheduler übergeordnet ist. Aus Anforderungen übernimmt das Interrupt-Handling und stellt einen eigenen Scheduler bereit. „This [RTAI/Adeos] is similar to the micro kernel approach described above [RTLinux], but is implemented in such a way as to avoid the claims of the Yodaiken patent.“ [Aeol02]. In beiden Projekten wird sowohl der Echtzeit- wie auch der Linux-Kernel im privilegierten Modus ausgeführt. Dies hat zur Folge, dass ein Fehler in einem Linux-Modul das gesamte System lahmlegen kann. Außerdem erhöht ein zweiter Kernel die Komplexität und damit die Fehleranfälligkeit des Systems.

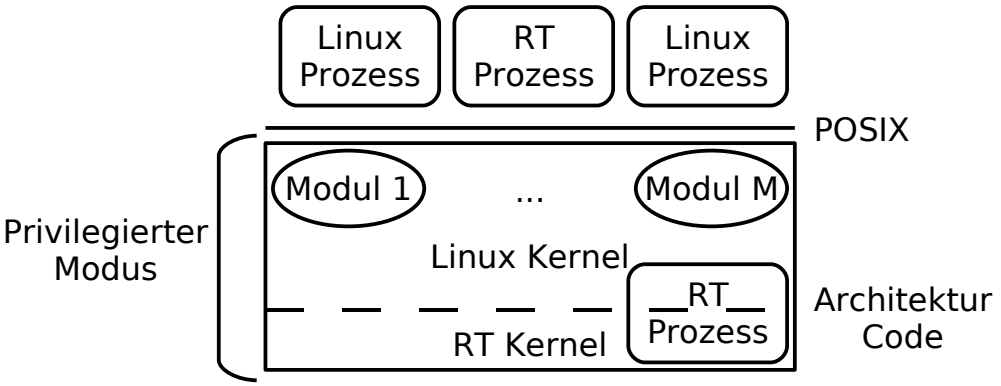

Abbildung 3.2: Dual-Kernel Architektur

Wendet man die IMA-Kriterien auf Linux (mit und ohne Echtzeit Patches) an, so ergibt sich folgendes Bild:

Echtzeit: Der Linux Kernel mit und ohne Echtzeit-Patches kann nur weiche Echtzeit-Bedingungen erfüllen. Der Dual-Kernel Ansatz soll dagegen auch harte Echtzeit-Bedingungen erfüllen können.

Partitionierung: Eine räumliche Partitionierung findet auf Anwendungsebene statt. RTLinux und RTAI/Adeos bieten die Möglichkeit, Echtzeit-Anwendungen im privilegierten Modus auszuführen. Damit könnte selbst ein Fehler in einer Anwendung das System zum Absturz bringen.

Beim Dual-Kernel Ansatz stehen neben den Scheduling-Klassen von Linux weitere Scheduling-Verfahren des Echtzeit-Kernels zur Auswahl. Der Scheduler des Echtzeit-Kernels hat dabei immer höchste Priorität, der Linux-Scheduler wird nur genutzt, wenn keine Echtzeit-Prozesse lauffähig sind.

Unabhängig vom Ansatz, können Linux-Gerätetreiber Interrupts unterdrücken und damit das System beliebig lange blockieren. Echtzeit-Bedingungen können dann nicht mehr erfüllt werden.

Trusted Computing Base: Ein monolithischer Kernel hat den Nachteil, dass alle Gerätetreiber zur TCB hinzugerechnet werden müssen. Bei den Dual-Kern Ansätzen kommt der Echtzeit-Kernel noch hinzu. Falls Echtzeit-Anwendungen im Kernel ausgeführt werden, gehören auch diese zur TCB.

Allein der Linux Kernel ohne Treiber umfasst in Version 2.6.9 für die IA32 etwas weniger als 150.000 LoC. RTLinux und RTAI/Adeos dürften diese Grenze überschreiten.

Offene Standards: Für Anwendungen steht die „quasi offene“ POSIX Schnittstelle zur Verfügung. Die Schnittstellen für Kernel Module sind ebenfalls umfangreich dokumentiert, wenn auch nicht standardisiert.

Modularität: Anwendungen die zum POSIX-Standard konform sind, lassen sich zwischen ebensolchen Betriebssystemen ohne größeren Aufwand portieren. Die Modulschnittstelle des Kernels ist offen, änderte sich in der Vergangenheit jedoch öfters. Es sollte mittlerweile eine gewisse Stabilität erreicht sein, das kann jedoch nicht garantiert werden. Generischer und architekturabhängiger Code sind durch die Verzeichnisstruktur getrennt.

3.2.2 Minix 3

Minix 3 ist ein Betriebssystem, das auf dem gleichnamigen μ-Kernel basiert. Es hat zum Ziel Computer gemäß dem „TV model“ zu bauen: kaufen, einschalten, benutzen [HBGH+06]. Dieses Prinzip wird auch unter dem Begriff „Plug & Play“ beworben. Die Zuverlässigkeit des Betriebssystems soll durch einen Selbstheilungsmechanismus erreicht werden. Ein so genannter „Reincarnation Server“ überwacht die Funktion der Gerätetreiber. Entdeckt er eine Fehlfunktion in einem Treiber, dann versucht er ihn zu ersetzen. Der Austausch des Treibers soll für den Benutzer möglichst transparent erfolgen. Neben der Nutzung auf Personal Computern wird auch der Einsatz in eingebetteten Systemen oder Kleingeräten wie dem „One Laptop per Child“-Projekt angestrebt. Ursprünglich wurde Minix 3 für IA32 entwickelt. In [Alti06] wird beschrieben wie Minix 3 auf die PowerPC-Architektur portiert wurde. Weitere Portierungen sind angekündigt.

Minix 3 wurde als Mehrbenutzer-UNIX-System entworfen. Die Architektur von Minix 3 ist in Abbildung 3.3 dargestellt. Nur der μ-Kernel wird im privilegierten Modus, andere Dienste oder Gerätetreiber werden im unprivilegierten Modus ausgeführt. Die

Dienstschicht stellt eine POSIX konforme Schnittstelle zur Verfügung. Das System-Call ABI des μ-Kernels ist zwar definiert, nicht aber standardisiert. Das liegt sicher auch daran, dass μ-Kernel und Betriebssystem zusammen konzipiert wurden und nicht für den separaten Einsatz gedacht sind.

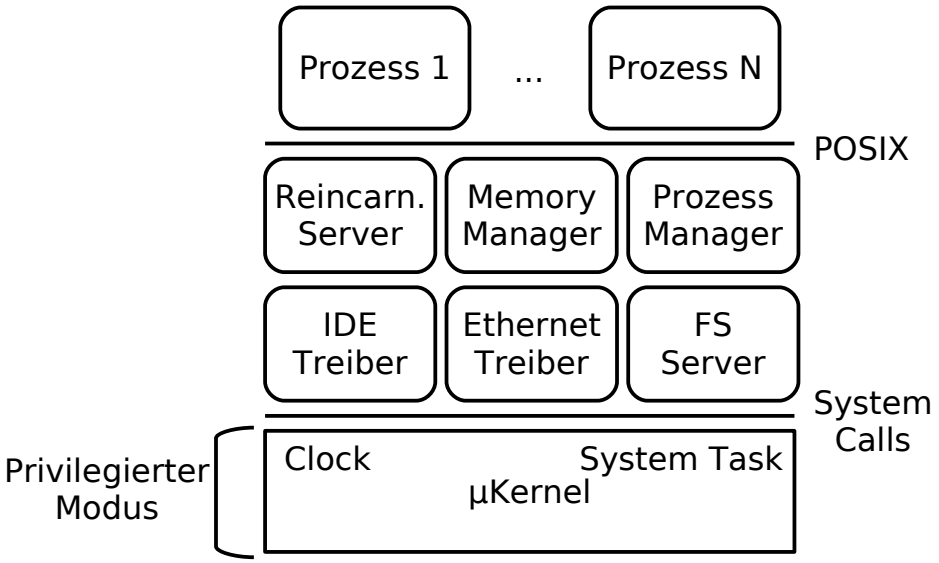

Abbildung 3.3: Minix 3 Architektur nach [HBGH+06]

Echtzeit-Anforderungen wurden beim Entwurf nicht berücksichtigt. So ändert der 16-stufige Scheduler die Priorität von Prozessen, um Fairness zu gewährleisten. Außerdem setzt der Prozess-Manager Swapping ein.

Unter Berücksichtigung der IMA-Kriterien ergibt sich für Minix 3 folgendes Bild:

Echtzeit: In der jetzigen Form ist Minix 3 nicht für den Einsatz in Umgebungen mit harten Echtzeit-Anforderungen geeignet, denn der Scheduler verändert die Prioritäten der Prozesse, um Fairness durchzusetzen. Allerdings wurde der Scheduler so konzipiert, dass er sich einfach erweitern lässt.

Die momentane Implementierung des Prozess-Managers ist für den Einsatz in Echtzeit-Systemen nicht geeignet. Da er aber als Server für den User-Mode konzipiert ist und nur über Nachrichten mit anderen Teilen des Systems interagiert, ließe er sich einfach durch eine Version ohne Swapping ersetzen.

Teile des Minix 3 μ-Kernels sind nicht reentrant. Beim Scheduling oder beim Versenden von Kernel Nachrichten (Message Passing) werden Hardware Interrupts abgeschaltet. Da dies kurze und deterministische Ausführungspfade sind, sollte es kein Hindernis der Echtzeit-Fähigkeit sein.

Partitionierung: Minix 3 nutzt Prozessor-Modi und MMU, um Kernel- von User-Space und Adressräume innerhalb des User-Space zu trennen. Durch die μ-Kernel-Architektur werden auch Geräte-Treiber und Betriebssystem-Dienste im User-Space ausgeführt. Eine räumliche Partitionierung kann somit gewährleistet werden.

Der Clock Gerätetreiber wird im Kernel-Mode ausgeführt und liefert dem μ-Kernel Zeitinformationen. Diese wird u. a. vom Scheduler benötigt um Prozessen die CPU zuzuteilen. Beide können aus dem User Mode nicht manipuliert werden und bilden die Grundlage der zeitlichen Partitionierung.

Trusted Computing Base: Eine minimale TCB wurde beim Entwurf von Minix 3 als wesentlich für die Stabilität des Systems angesehen [HBGH+06]. Der Kernel umfasst für die IA32 etwas weniger als 5.000 LoC. Neben μ-Kernel sind auch Speicher- und Prozessmanager sowie evtl. der Reincarnation Server der TCB zuzurechnen.

Offene Standards: Die Dienstschicht von Minix 3 stellt eine Schnittstelle für Anwendungsprogramme gemäß der „Single UNIX Specification Version 3“ bereit. Die Schnittstelle zwischen μ-Kernel und Betriebssystem ist auf den Seiten des Minix 3 Projekts[1] definiert, aber nicht standardisiert.

Modularität: Minix 3 ist ein Multiserver Betriebssystem. Da die verschiedenen Teile des Systems nur über IPC interagieren können, ist es möglich Server, Treiber, Anwendungen oder gar den μ-Kernel durch eine neue Implementierung zu ersetzen. Außerdem können neue Komponenten hinzugefügt werden, ohne das bestehende System zu beeinflussen.

3.2.3 Xen

Xen wurde als Virtual-Machine-Monitor (VMM) für bis zu 100 para-virtualisierte Gast-Betriebssysteme entworfen [BDFH+03]. Jedem Gast-Betriebssystem wird eine virtualisierte Teilmenge der original Hardware in so genannten Domains bereit gestellt. Damit folgt Xen dem Exokernel Ansatz, der nicht von der Hardware abstrahiert, sondern sie gesichert exportiert [EnKO95]. Auf der IA32 werden in Version 3.0.4 auch Hardware Mechanismen von Intel- und AMD-Prozessoren unterstützt, die eine vollständige Virtualisierung ermöglichen.

Wie physische Hardware besteht die virtuelle Hardware aus CPU, Speicher und Geräten. Interrupts werden durch Event-Kanäle modelliert. Von besonderer Bedeutung ist die Domain 0. Sie dient der Verwaltung des Systems und ist mit dem `root` Nutzer in UNIX-Systemen vergleichbar. Nur Domain 0 hat direkten Zugriff auf die Hardware. Geräte-Treiber müssen daher in zwei Teile geteilt werden (Split-Driver). Das Backend wird in Domain 0 ausgeführt und greift direkt auf die Hardware zu. Das Frontend wird als Geräte-Treiber in einem Gast-Betriebssystem ausgeführt. Die Kommunikation zwischen Back- und Frontend findet über Call-back Handler und System-Calls des VMM statt. Die Architektur von Xen ist in Abbildung 3.4 dargestellt.

Echtzeit: Xen wurde als Hypervisor für virtualisierte Server entwickelt, wie sie vor allem bei Internet Service Providern zum Einsatz kommen. In Version 3.0.4 kann man zwischen zwei Scheduling-Algorithmen wählen: Einem Earliest-Deadline-First und einem Credit-Scheduler. Ersterer ist ein Echtzeit-, letzterer ein fairer Scheduling-Algorithmus. Inwiefern das Interrupt-Handling durch Event-Kanäle deterministisch ist, lässt sich auf den ersten Blick nicht sagen.

[1] http://www.minix3.org/

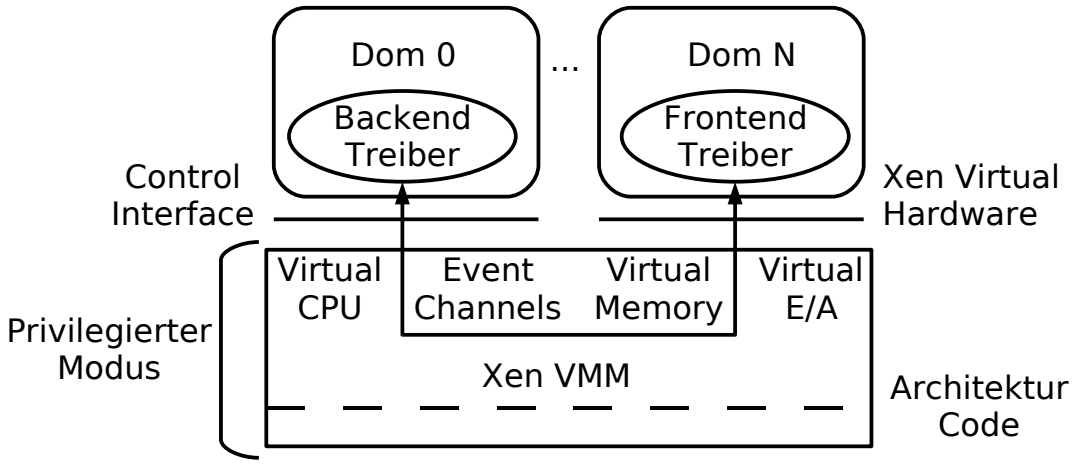

Abbildung 3.4: Xen Architektur nach [BDFH+03]

Partitionierung: Eines des Entwurfsziele von Xen war die „Performance Isolation“. D. h. dass sich das Verhalten einer Domain nicht auf andere Domains auswirken darf. Dies entspricht der Definition von Partitionierung. Außerdem entspricht das Split-Driver Modell zusammen mit Domain 0 der Idee einer „device mangement partition“ aus [Rush99].

Die Implementierung von Xen auf der IA32 nutzt zum Speicherschutz alle verfügbaren Prozessor Modi („Ringe“). Bei anderen Architekturen ist die Verwendung zweier Prozessor-Modi vorgesehen. Die MMU soll dabei nicht nur VMM gegen Gast-OS und Gast-OS untereinander isolieren, sondern auch Mechanismen bereit stellen, die innerhalb einer Domain die Isolation von Gast-OS und Anwendungen ermöglichen.

Trusted Computing Base: Neben dem VMM ist auch Domain 0 der TCB zuzurechnen, da eine Fehlfunktion in Domain 0 das ganze System lahm legen kann. Der VMM umfasst in Version 3.0.4 etwas weniger als 60.000 LoC für die IA32[2].

Offene Standards: Die System-Calls des VMM sind definiert, jedoch nicht standardisiert. An einem Management API wird gearbeitet[3].

Modularität: Gast-OS können beliebig ausgetauscht werden. Geräte-Treiber müssen nach dem Split-Driver Modell umgeschrieben werden. Ein Austausch des VMM wäre theoretisch auch denkbar, sofern System-Calls und sonstige Mechanismen bereit gestellt werden. Unter IMA-Gesichtspunkten ist der größte Nachteil von Xen darin zu sehen, dass keine Hardware-Abstraktion vorhanden ist. Damit lassen sich Partitionen nicht ohne großen Aufwand auf eine andere Hardware portieren.

[2]Unter http://www.xensource.com/products/xen/ werden weniger als 50.000 LoC angegeben.

[3]Informationen zum Xen API können auf dem Wiki des Projekts nachgelesen werden: http://wiki.xensource.com/xenwiki/XenApi

3.2.4 L4

Das in [Lied95] vorgestellte Konzept eines effizienten μ-Kernels bildet die Grundlage der L4-Architektur. Die L4-Architektur stellt dem Anwendungsentwickler Abstraktionen zur Verfügung, die eine architekturunabhängige Entwicklung ermöglichen.

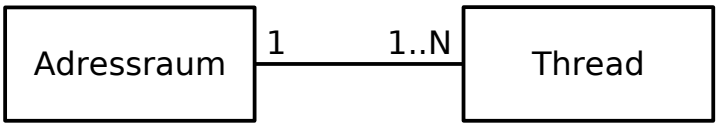

Abbildung 3.5: Verhältnis von Adressraum zu Thread

Threads sind eine Abstraktion parallel ablaufender Aktivitäten. Threads haben eine Priorität in $[0, 255]$. Threads gleicher Priorität werden mit dem Round-Robin Algorithmus der CPU zugeteilt. Wie in Abbildung 3.5 zu sehen ist, gehört jeder Thread zu genau einem Adressraum. Neben dem Adressraum haben Threads noch zwei weitere Abhängigkeiten:

- Der Pager wird benachrichtigt, falls der Thread auf Speicherbereiche zugreift, die sich nicht in seinem Adressraum befinden.
- Der Scheduler kann die Scheduling Parameter des Threads ändern und wird benachrichtigt, wenn das Zeit-Quantum des Threads verbraucht wurde.

Adressräume sind eine Abstraktion von Ressourcen. Über Adressräume wird der Zugriff auf den Hauptspeicher und E/A-Geräte organisiert. Hier kommen jedoch Unterschiede der Hardware-Architekturen zum Tragen. Je nachdem ob die Hardware-Architektur zwei Adressbusse für Speicher und Geräte-Register vorsieht (z. B. IA32) oder einen gemeinsamen Adressbus (z. B. PowerPC), müssen Anwendungen eine oder zwei Mechanismen zum Zugriff auf Ressourcen nutzen. Wie in Abbildung 3.5 zu sehen, müssen Adressräume mindestens einen Thread enthalten. Dieser kann jedoch inaktiv sein.

Kommunikation zwischen Threads findet mittels Inter-Process-Communication (IPC) statt. IPC ist ein synchroner Mechanismus. Zwischen den L4 API-Spezifikationen gibt es erhebliche Unterschiede bei der IPC. In Version V2 wird das Clan-and-Chiefs Modell eingesetzt, das mit Version X.0 verworfen wird. In Version X.2 gibt es einen Time-out für IPC-Nachrichten, der in Version N1 verworfen wird. Dafür kommt in Version N1 mit der „Notification“ ein asynchroner IPC-Mechanismus hinzu.

Jedes L4-System besteht neben dem μ-Kernel aus mindestens zwei Threads in jeweils separaten Adressräumen:

Sigma0: Dem Pager Sigma0 gehört nach Systemstart der gesamte verfügbare Speicher. Er ist auch der Pager vom Root-Task und behandelt dessen Pagefaults. Das Protokoll zur Kommunikation mit Sigma0 ist Teil der L4-Spezifikation.

Root-Task: Über den Root-Task lassen sich einfache Anwendungen oder Dienstschichten für Betriebssysteme implementieren. Er ist mit dem Init-Prozess von UNIX-Systemen vergleichbar.

Mit Sigma1 ist in der Spezifikation ein weiterer optionaler Thread vorgesehen. Er wird jedoch von keinem der in Tabelle 3.2 aufgeführten Projekte verwendet.

Die L4-Architektur wird durch offene Spezifikationen der Kernelschnittstelle festgelegt. Die erste offizielle Version des Kernel API heißt V2 und wurde über die Jahre weiterentwickelt. Jede Stufe wurde dabei in einer eigenen Spezifikation festgehalten. Daher ist bei Implementierungen der μ-Kernelschnittstelle L4 zu beachten auf welcher Version sie beruht.

Name	*API*	*Projekt*	*Architekturen*
L4Ka::Pistachio	X.2	L4Ka	Alpha, AMD64, ARM, IA32, IA64, MIPS, PowerPC32, PowerPC64
NICTA::Pistachio-embedded	N1	NICTA	ARM, IA32, MIPS
OKL4	OKL4	OKL	ARM, IA32
Fiasco	V2/X.0	DROPS	ARM, IA32

Tabelle 3.2: L4 Projekte

In Tabelle 3.2 sind die aktuell aktiven L4-Projekte aufgeführt, die unter einer Open Source Lizenz entwickelt werden. An der Technischen Universität Dresden ist es das „Dresden Real-Time Operating System Projekt“ (DROPS), an der Universität Karlsruhe das L4Ka-Projekt, am National ICT Australia (NICTA) das „Embedded Systems Framework“ (ESF) und von den Open Kernel Labs (OKL) das OKL4-Projekt.

OKL4 wurde im April 2007 erstmalig von den neu gegründeten Open Kernel Labs veröffentlicht. Er war zur Zeit der Entscheidungsfindung noch nicht verfügbar, wird jedoch der Vollständigkeit halber hier aufgeführt. OKL4 basiert auf NICTA::Pistachio-embedded, welches wiederum auf L4Ka::Pistachio zurück geht. Fiasco ist dagegen eine komplett eigenständige Entwicklung.

Sowohl NICTA::Pistachio-embedded wie auch Fiasco wurden für den Einsatz in Echtzeit-Umgebungen entwickelt. Beide sind in ein größeres Framework eingebettet, das eine weitere Dienst-Schicht zwischen μ-Kernel und Anwendung einführt. Insofern ist Abbildung 3.6 als gemeinsamer Nenner von DROPS und Embedded Systems Framework (ESF) vom NICTA zu verstehen.

Unter Berücksichtigung der IMA-Kriterien ergibt sich für die L4-Architektur folgendes Bild:

Echtzeit: In [Ruoc06] wird argumentiert, dass NICTA::Pistachio-embedded harte Echtzeit-Anforderungen erfüllen kann, wenn sich die Entwickler der internen Vorgänge des Kernels bewusst sind. Dazu gehören die Scheduling Entscheidungen, die auf dem Interrupt- und IPC-Pfad getroffen werden. Als problematisch

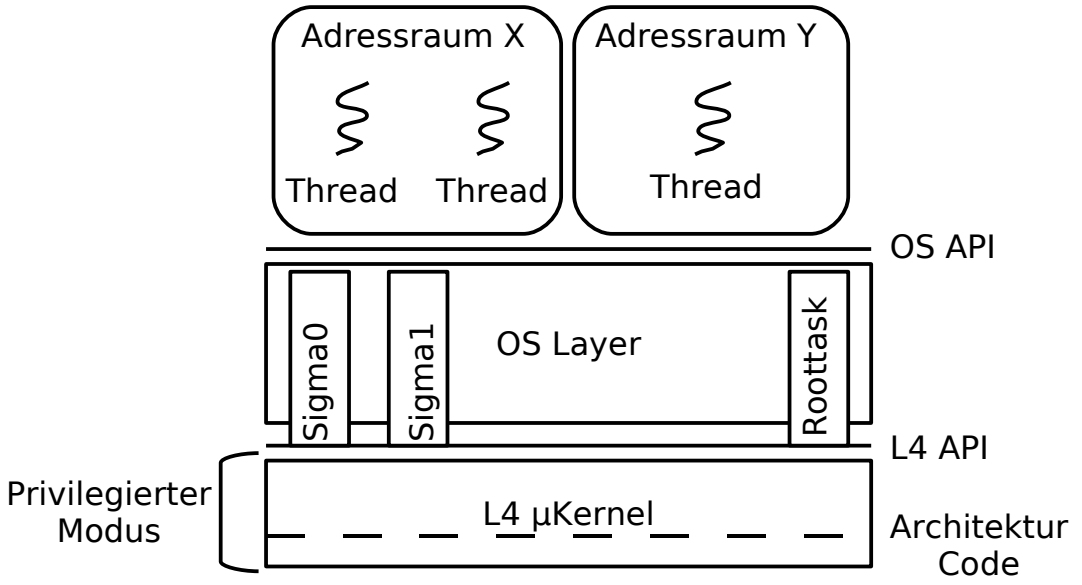

Abbildung 3.6: Typische L4-Architektur

werden tief verschachtelte Seiten-Mappings angesehen, die beim „Unmapping„ rekursiv abgearbeiten werden müssen. Für die Zeit der Unmapping-Operation sind Interrupts gesperrt. Da eine solche Operation sehr lange dauern kann, ist sie für Echtzeit-Systeme unakzeptabel. Für die Zukunft sind Änderungen am Kernel angekündigt, die die Entwicklung von Anwendungen mit harten Echtzeit-Anforderungen erleichtern. Der μ-Kernel Fiasco vom DROPS ist dagegen von vornherein preemptibel konzipiert worden.

Ein klassisches Problem von Echtzeit-Systemen ist die „Priority Inversion“. Bei L4 kann sie durch die synchrone Natur von IPC auftreten: Seien drei Threads A, B und C mit den Prioritäten $P(C) < P(B) < P(A)$ gegeben. Sendet A eine IPC-Nachricht an C, blockiert er. Ist nun B rechenbereit, wird er ausgeführt. C und damit auch A müssen warten.

Hardware Interrupts werden in L4 als “IPC-Calls“ angesehen, die von einem Hardware Thread gesendet werden. Tritt ein Interrupt auf, erzeugt der Kernel einen „pseudo“ Thread, der eine IPC an den Handler verschickt und auf Bestätigung wartet. Der eigentliche Handler wird im unprivilegierten Modus ausgeführt. Selbst wenn der Interrupt Handler die höchste Priorität im System hat, ist die Latenz-Zeit durch den zusätzlichen Kontextwechsel größer als bei monolithischen Kerneln.

Partitionierung: Alle genannten L4-Implementierungen nutzen vorhandene Prozessor-Modi zur Einschränkung des Befehlssatzes und der Zugriffsrechte im User-Modus.

Räumlich: Der Kernel wird von der MMU gegen Manipulation durch User-Threads geschützt. Außerdem werden User-Adressräume gegeneinander abgeschirmt. Innerhalb eines Adressraumes besteht kein räumlicher Schutz.

Zeitlich: Der Kernel-Scheduler ordnet Threads erst nach Prioritäten und bei mehreren Threads gleicher Priorität nach dem Round-Robin Verfahren dem Prozessor zu. Threads können auch von einem Scheduler im User-

Mode verwaltet werden. Dieser legt dann über das Zeitquantum fest, wie lange der Thread den Prozessor nutzen darf. Wenn das Zeitquantum abgelaufen ist, wird der Scheduler vom Kernel benachrichtigt. Der Kernel ermöglicht durch diesen Mechanismus, dass Scheduling-Entscheidungen im User-Mode getroffen werden können. Eine weitere Möglichkeit, einen Scheduler im User-Mode zu implementieren bietet der Mechanismus der „Timeslice-Donation“. Damit kann ein Scheduler den verbleibenden Rest seiner Zeitscheibe einem anderen Thread zuteilen.

Ein Scheduler-Thread muss sich nicht im gleichen Adressraum befinden wie die Threads für die er verantwortlich ist. Der Vorteil eines gemeinsamen Adressraums ist, dass ein Kontextwechsel beim Scheduling eingespart wird. Allerdings besteht dann die Gefahr, dass fehlerhafte User-Threads den Scheduler-Thread manipulieren.

Trusted Computing Base: Neben dem Kernel selbst gehören auch Sigma0, Sigma1 und der Root-Task zur TCB. Die Implementierung des Kernels für IA32 umfasst bei L4Ka::Pistachio etwas weniger als 15.000 LoC.

Offene Standards: Das Kernel API ist durch das jeweilige Referenzhandbuch spezifiziert. Die OS-Dienstschicht hängt vom jeweils verwendeten Projekt ab.

Modularität: Durch die klar definierte Kernelschnittstelle sollten sich Anwendungen leicht von einer Kernel-Implementierungen zu einer anderen portieren lassen. Voraussetzung dafür ist allerdings, dass von jedem Kernel das gleiche API implementiert wird. Leider führt die Offenheit dazu, dass jedes L4-Projekt ein eigenes API entwickelt. Für die OS-Dienstschicht gibt es kein einheitliches API. Auf dieser Ebene ist ein Austausch von Komponenten noch aufwendiger.

L4Ka::Pistachio ist bereits auf viele Architekturen portiert wurden. Die Verzeichnisstruktur trennt hardwareabhängigen (Architektur, Plattform) von hardwareunabhängigem (API) Code. Diese Teilung ist bei allen Nachfolgeprojekten erhalten geblieben. Allerdings ist nur aus dem Quellcode ersichtlich welche Methoden von der Architektur oder der Plattform bereit gestellt werden müssen. Eine Dokumentation ist nicht vorhanden.

Fiasco wurde für die IA32 entwickelt, ist mit Version 1.2 aber auch für ARM verfügbar. 2006 scheiterte der Versuch, Fiasco auf den PowerPC zu portieren [Lang06]. Dies lässt den Schluss zu, dass die Portabilität von Fiasco eingeschränkt ist.

3.3 Auswahl

Um zu entscheiden welcher Kernel für eine Umsetzung in Frage kommt, wird in Tabelle 3.3 die Diskussion aus Abschnitt 3.2 auf einer ordinalen Skala zusammengefasst. Die Bedeutung der „Noten“ auf der fünfstufigen Skala ist folgende:

1: Der Kernel erfüllt die Anforderungen sehr gut.

2: Der Kernel erfüllt die Anforderungen gut.

3: Der Kernel hat das Potential, die Anforderungen zu erfüllen.

4: Der Kernel erfüllt die Anforderungen schlecht.

5: Der Kernel erfüllt die Anforderungen nicht.

Linux hat den Nachteil, als monolithischer Kernel für den Mehrbenutzerbetrieb entworfen worden zu sein. An dieser Schwäche können auch die Echtzeit-Erweiterungen nichts ändern. Minix 3 basiert dagegen auf einem μ-Kernel. Die Implementierung von Scheduler und Prozess-Manager ist jedoch auf den Einsatz in Desktop-Systemen ausgelegt. Durch die modulare Architektur könnten diese Teile jedoch an Echtzeit-Anforderungen angepasst werden. Positiv ist die Zielsetzung einer verbesserten Ausfallsicherheit.

Bei Xen wird besonderen Wert auf die Partitionierung der Domänen gelegt. Eine Domäne kommt einer ARINC-Partition aus [AEEC03] sehr nahe. Positiv ist außerdem, dass ein Echtzeit-Scheduler verfügbar ist. Ob das Interrupt-Handling deterministisch ist, lässt sich auf den ersten Blick nicht sagen. Die Größe des Xen Hypervisors liegt zwischen Minix 3 und Linux. Negativ ist bei Xen, dass es keine Abstraktion von der Hardware anbietet.

Pistachio-embedded und Fiasco sind Implementierungen der L4-Architektur für eingebettete Systeme. Ihnen wird zugeschrieben harte Echtzeit-Bedingungen erfüllen zu können. Als μ-Kernel haben sie wie Minix 3 eine kleine TCB und forcieren eine modulare Architektur des Betriebssystems. Eine ARINC-Partitionierung kann durch Adressräume nur approximiert werden. Hier ist die Lösung von Xen deutlich besser. Dafür abstrahiert L4 von der Hardware und erhöht somit die Portabilität von Anwendungen.

	Echtzeit	*Partitionierung*	*TCB*	*Standards*	*Modularität*
Linux	5	2	5	2	2
Linux+RT	2	4	5	2	2
Minix 3	3	2	1	2	1
Xen	3	1	2	2	4
L4	2	2	1	1	1

Tabelle 3.3: Kernel Übersicht

Die Abstraktion von der Hardware und die Ausführungen in [Ruoc06] sprechen für die Auswahl eines L4-Kernels. Neben den IMA-Anforderungen sprechen auch zwei weitere Faktoren für einen L4-Kernel:

1. Die Verfügbarkeit von Pistachio für den PowerPC-Prozessor, der auf dem Hellas-Board zum Einsatz kommt. Details zum Hellas-Board sind in Abschnitt 4.1 zu finden.

2. Die Zielsetzung der L4-Projekte an der TU Dresden und am NICTA, die Korrektheit des Kernels mathematisch zu verifizieren:

   VFiasco: „The VFiasco project has two primary goals:

   (a) to further develop coalgebraic specification techniques such that these techniques can be applied to real software

(b) to mechanically verify some security relevant properties of a complete μ-kernel (microkernel) operating system running on x86 PCs“. [VFia]

L4.verified „The L4.verified project is providing a mathematical, machine-checked proof of the functional correctness of the L4 microkernel with respect to a high level, formal description of its expected behaviour.“ [L4.v]

Die Bewertung in Tabelle 3.3 und die obigen nicht-IMA-Faktoren legen nahe, einen Kernel der Pistachio-Reihe zu verwenden. Die endgültige Auswahl findet in Abschnitt 4.2 statt. Dabei spielt die Zielsetzung der Arbeit, ein Linux System in der Luftfahrt einzusetzen, eine wichtige Rolle.

3.4 Zusammenfassung

In diesem Kapitel werden vier Kernel unter IMA-Gesichtspunkten bewertet:

Linux: Monolithischer Kernel für ein POSIX Betriebssystem.

Minix 3: μ-Kernel für ein Desktop-Betriebssystem gleichen Namens.

Xen: Ein Hypervisor für para-virtualisierte Gast-Betriebssysteme.

L4: Eine μ-Kernelschnittstelle die eine abstrakte Hardware anbietet.

Linux hat den Nachteil, als monolithischer Kernel für den Mehrbenutzerbetrieb entworfen worden zu sein. An dieser Schwäche können auch die Echtzeit-Erweiterungen nichts ändern. Daher kann er nicht als Grundlage eines IMA-Systems eingesetzt werden.

Minix 3 wäre eine Alternative zu Linux, wenn die Implementierung nicht auf den Desktop zugeschnitten wäre. Es wird durch die μ-Kernel Architektur und den Reincarnation-Server zwar die Ausfallsicherheit erhöht, Scheduler und Prozess-Manager sind dagegen nicht für den Einsatz unter harten Echtzeit-Bedingungen tauglich.

Xen verfolgt als Hypervisor einen anderen Ansatz. Es exportiert eine Teilmenge der Hardware als virtuelle Maschine in so genannte Domains. In den Domains können Gast-Systeme isoliert voneinander ausgeführt werden. Die Isolation durch Domains kommt der Funktionalität einer Partition in [AEEC03] sehr nahe. Innerhalb einer Domain kann z. B. ein para-virtualisiertes Linux zum Einsatz kommen. Über die Echtzeit-Fähigkeiten von Xen ist wenig zu erfahren. Positiv ist jedoch, dass ein Earliest-Deadline-First Scheduler implementiert wurde. Von Nachteil ist hingegen, dass der Hypervisor trotz minimaler Funktionalität recht groß ist und keine Abstraktion von der Hardware anbietet.

Die μ-Kernelschnittstelle L4 stellt dagegen eine abstrakte Hardware zur Verfügung. Auf dieser kann ebenfalls ein para-virtualisiertes Linux zum Einsatz kommen. Gleichzeitig können isoliert davon Echtzeit-Anwendungen ausgeführt werden. Pisatchio-embedded und Fiasco sind Implementierungen der Kernelschnittstelle L4, die harte Echtzeit-Bedingungen erfüllen sollen. Daher beschäftigt sich diese Arbeit mit einer Linux Lösung auf Basis eines L4-Kernels.

4. Implementierung

Die Analyse verschiedener Open Source Kernel in Abschnitt 3.2 legt die Verwendung eines L4-Kernels nahe. Aus den verfügbaren L4 Implementierungen scheinen der Pistachio Kernel und seine Nachfolger am Besten geeignet. Dies legen auch die Hardware-Vorgaben nahe, die in Abschnitt 4.1 unter IMA-Gesichtspunkten betrachtet werden. In Abschnitt 4.2 werden verschiedene Arbeitspakete vorgestellt und eines zur Implementierung ausgewählt. In Abschnitt 4.3 werden die Arbeitsschritte beschrieben und Entscheidungen zu Entwurf und Implementierung dargelegt.

4.1 Hardware

4.1.1 Board

Das Hellas-Board besteht aus zwei Rechnern, einem Verfahrensrechner und einem Schnittstellenrechner. Auf beiden kommt ein PowerPC 750Cxe von IBM zum Einsatz. In dieser Arbeit wurde nur der Schnittstellenrechner genutzt. Eine Teilmenge des Blockschaltbildes ist in Abbildung 4.1 zu sehen. Aus dem Blockschaltbild wurden jene Bauteile entfernt, die spezielle Technologien der Luftfahrt enthalten. Dies stellt insofern kein Informationsverlust dar, da sie in dieser Arbeit keine Verwendung fanden.

Der PowerPC-Prozessor ist über eine Marvell-Hostbridge vom Typ GT-6426x mit SDRAM, Flash und Peripherie verbunden. Die CPU verfügt über eine JTAG-Leitung, die eine befehlsweise Ausführung (Single-Stepping) von Programmen ermöglicht.

4.1.2 PowerPC-Architektur

Der PowerPC 750Cxe implementiert die 32-Bit PowerPC-Architektur. Die PowerPC-Architektur wurde von Apple, IBM und Freescale (damals noch Motorola) als 64-Bit RISC-Architektur mit einer 32-Bit Untermenge entwickelt. Sie ist in drei Ebenen definiert:

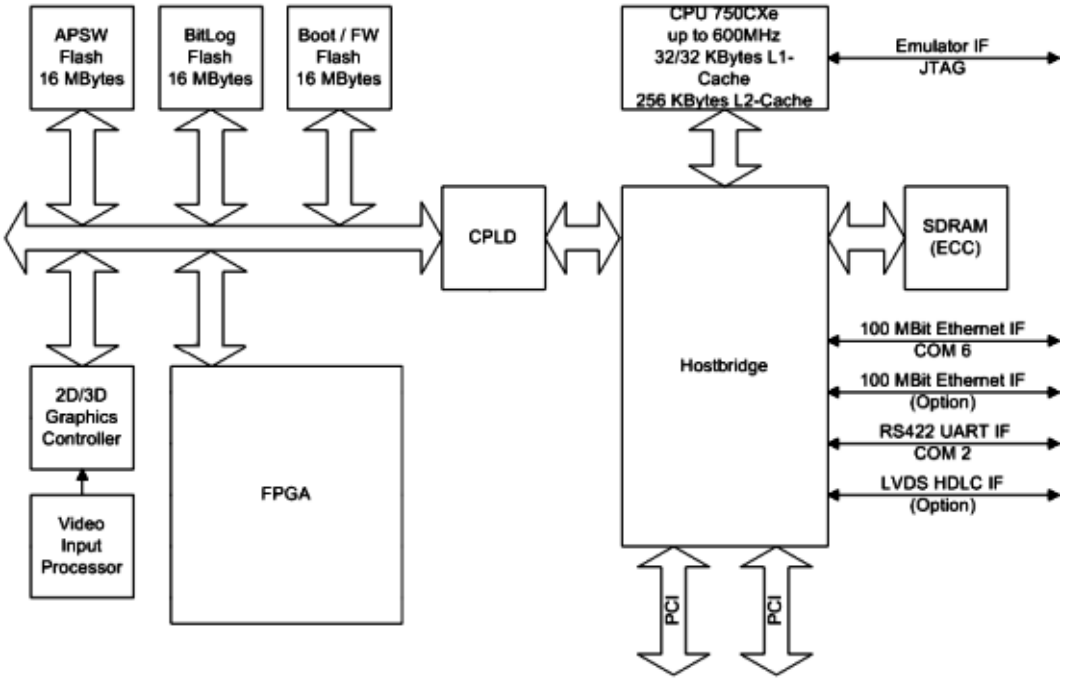

Abbildung 4.1: Blockschaltbild des Hellas-Boards (Quelle: EADS Deutschland GmbH)

UISA: Die *User Instruction Set Architecture* legt 32 Integer-, 32 Fließkomma- und die jeweiligen Konditionsregister, sowie ein Register zum Speichern der Rücksprungadresse und ein Zähler fest. Hinzu kommt ein Satz Arithmetik-, Logik- und Sprungbefehle.

VEA: Die *Virtual Environment Architecture* definiert das zweigeteilte 64-Bit Timebase Register, das der Zeitmessung dient. Außerdem werden Aspekte des Speichermodells für Gerätezugriffe definiert.

OEA: Die *Operating Environment Architecture* definiert das Speichermodell sowie die dazugehörigen Register und Befehle. Gleiches gilt für das Exception-Modell, in dem Unterbrechungen und Fehler behandelt werden. Weitere Register des OEA sind das Machine-State-Register (MSR), die Timebase mit Schreibzugriff und der taktgesteuerte Dekrementer. MSR und Dekrementer werden weiter unten noch genauer beschrieben.

Die verschiedenen Ebenen stehen in einer Untermengen-Beziehung: UISA $\subset$ VEA $\subset$ OEA. D. h. ein Prozessor der die OEA implementiert, implementiert auch die VEA und UISA.

Die PowerPC-Architektur legt keine Hardware-Implementierung fest. Jede Implementierung der PowerPC-Architektur muss jedoch die Register und Befehle der UISA bereitstellen. Dadurch wird die Binär-Kompatibilität von Anwendungen gewährleistet.

Der PowerPC 750Cxe implementiert die OEA wie in Abbildung 4.2 zu sehen ist. Darüber hinaus bietet er noch weitere Mechanismen zur Performanz-Messung und Energie-Kontrolle.

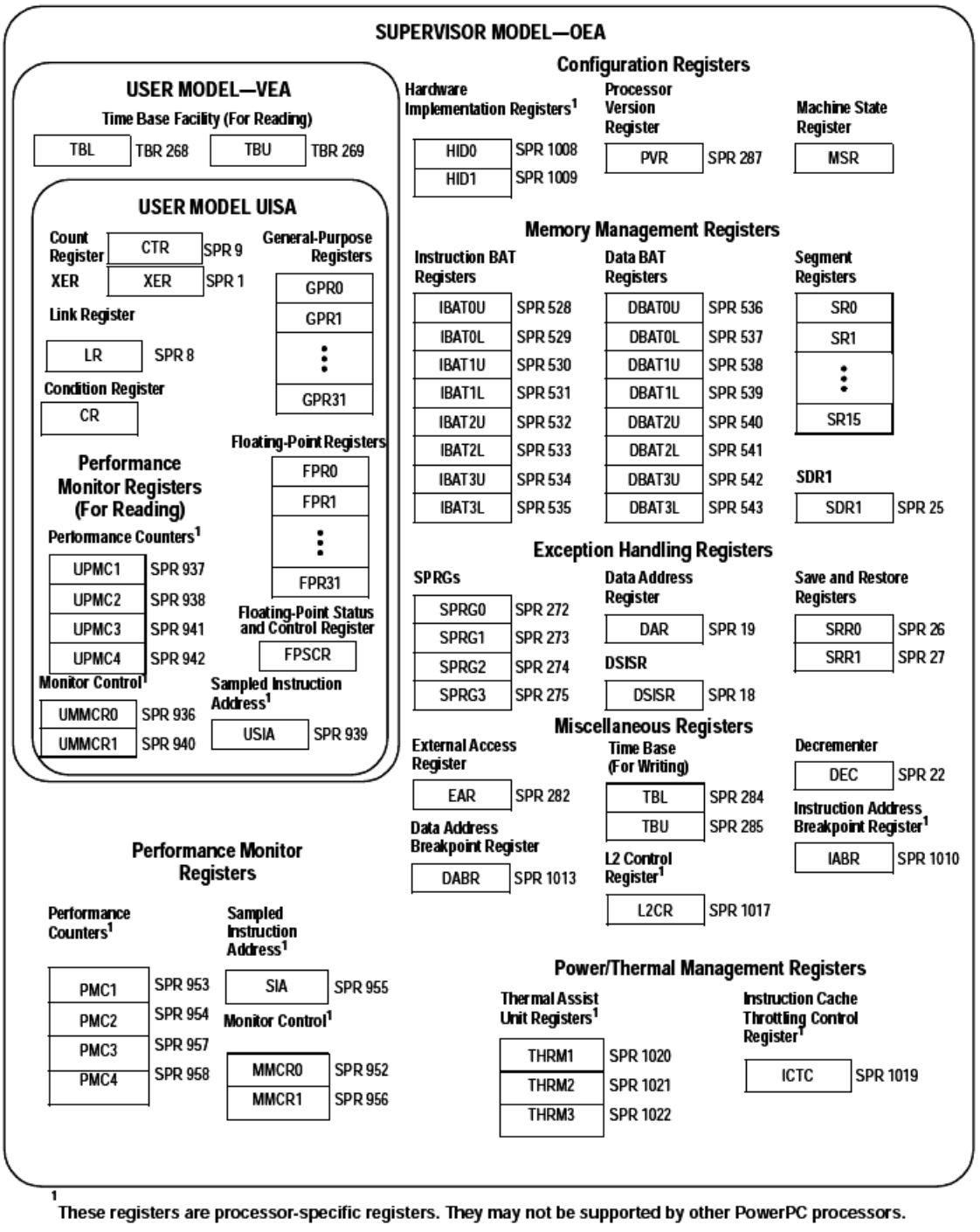

Abbildung 4.2: Register des PowerPC 750Cxe (Quelle: [IBM05])

4.1.3 PowerPC 750Cxe

Auf dem Hellas-Board kommt ein PowerPC 750Cxe zum Einsatz. Er hat neben den Registern aus Abbildung 4.2 noch folgende Merkmale:

- Prozessortakt: 600 MHz
- Bustakt: 100 MHz
- Zwei separate Memory Management Units für Daten und Befehle
- L1 Caches: 8-Wege mengenassoziativ pro MMU
- L2 Cache: 2-Wege mengenassoziativ + 256 KB SRAM Datenspeicher
- Translation-Lookaside-Buffer (TLB) mit 128 Einträgen pro MMU
- Energie- und Temperaturkontrolle
- Sprungvorhersage
- Monitor zur Performanz-Messung

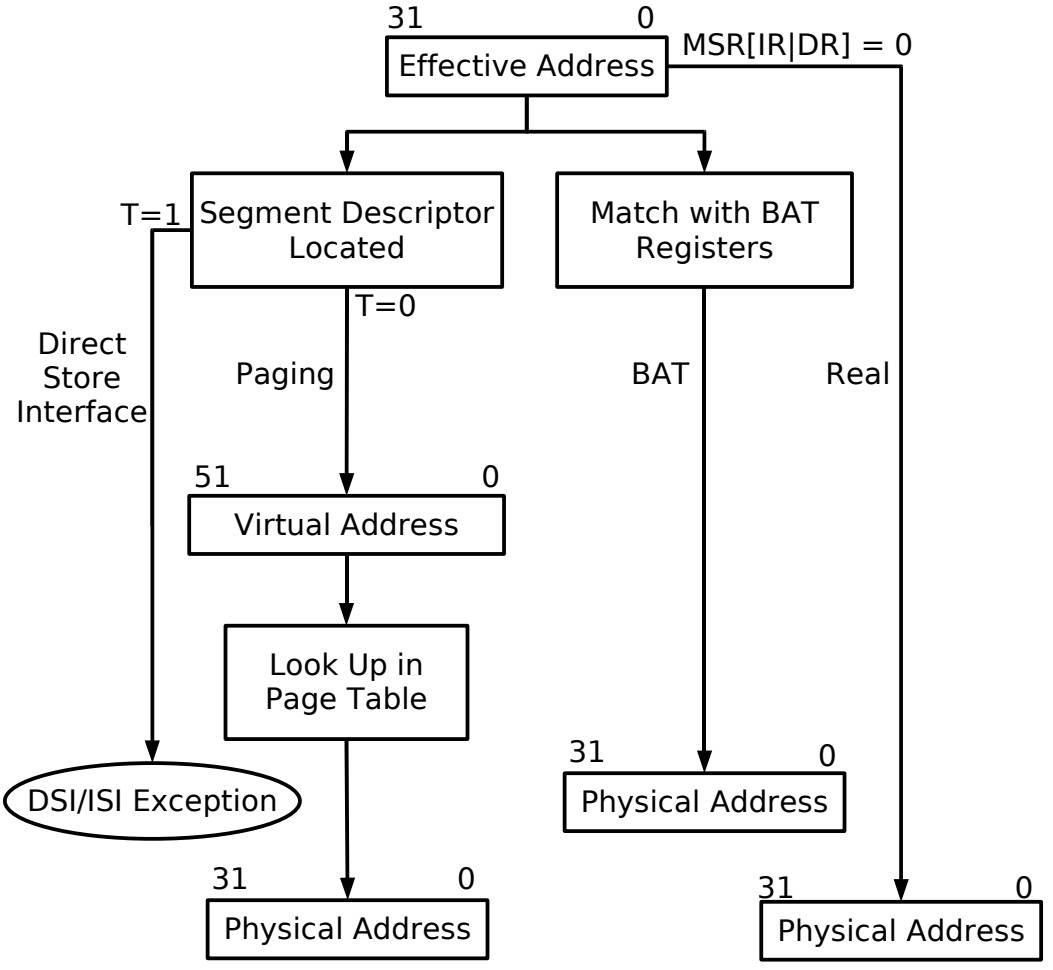

Abbildung 4.3: Address-Translation for PowerPC 750Cxe (Quelle: [IBM05])

Software auf dem PowerPC verwendet so genannte „effektive Adressen“. Wenn keine Übersetzung von Speicheradressen (Addresstranslation) aktiviert wurde, entspricht die effektive Adresse der physikalischen Adresse. Über ein Flag im Machine-State-Register kann die jeweilige MMU für Daten und Befehle aktiviert werden. Dies führt wie in Abbildung 4.3 zu sehen ist zur Aktivierung der Block-Address-Translation (BAT) und des Paging. Ist für eine effektive Adresse ein BAT- und ein Paging-Eintrag vorhanden, dann wird der BAT-Eintrag genutzt. Das Direct-Store-Interface aus Abbildung 4.3 wird vom PowerPC 750Cxe nicht unterstützt. Wenn es bei einem Zugriff adressiert wird, löst die MMU eine Exception aus. BAT und Paging funktionieren folgender Maßen:

BAT: Eine Block-Address-Translation (BAT) definiert eine Abbildung $E \longrightarrow P$ von effektiven[1] Adressen E in physikalische Adressen P. E und P bestehen aus einem kontinuierlichen Speicherblock, der in seiner Größe zwischen 128 KB und 256 MB variieren kann. Eine Abbildung wird über zwei Register definiert, dem oberen und unteren BAT-Register. Es gibt je vier solche Register-Paare für Daten- und Befehlsadressen. Neben der Abbildung lassen sich über die BAT-Register auch Schutzrechte vergeben und das Caching festlegen.

Paging: Die PowerPC-Architektur spezifiziert in der OEA für 32-Bit Prozessoren ein segmentiertes Paging-Verfahren wie es in Abbildung 4.4 zu sehen ist. Effektive Adressen E werden durch Paging auf physikalische Adressen P abgebildet. Dabei werden die Adressen E in 4 KB große Seiten gruppiert. Die niederwertigsten 12 Bit werden maskiert und ergeben den Abstand (Offset) einer Adresse zu ihrem Seitenrahmen. Seiten werden über eine invertierte Seitentabelle einem physikalischen Seitenrahmen zugeordnet. Die physikalische Adresse ergibt sich durch Addition der Adresse des Seitenrahmens mit dem Offset. Die Segmentierung kommt bei der Berechnung des Eintrags in der invertierten Seitentabelle zum Tragen. Neben dem Page Index (PI), der sich direkt aus der effektiven Adresse (EA) ergibt, wird durch die obersten 4 Bit der EA ein Segmentdeskriptor (-register) ausgewählt. In diesem ist die 24 Bit große Virtual Segment ID (VSID) gespeichert. VSID und PI ergeben den Schlüssel für die Suche im TLB und der Seitentabelle. Die Suche in der Seitentabelle erfolgt über eine primäre und eine sekundäre Hashfunktion. Wird dem TLB eine neue Seite hinzugefügt, erfolgt die Ersetzung nach dem Least-Recently-Used (LRU) Algorithmus.

4.1.4 Hardware Mechanismen und IMA-Anforderungen

Ein Kernel die IMA-Anforderung nur gewährleisten, wenn die Hardware entsprechende Mechanismen zur Verfügung stellt. Dies gilt vor allem für die Partitionierung und das Echtzeit-Verhalten. Daher müssen die IMA-Anforderungen auch bei der Auswahl der Hardware-Architektur eine Rolle spielen. Die Hardware des Hellas-Boards lässt sich zwar für diese Arbeit nicht verändern, dennoch ist es wichtig zu wissen, welche Mechanismen die Hardware zur Erfüllung der IMA-Anforderungen bereit stellt:

[1] In [IBM05] wird als „effektive“ Adresse jene bezeichnet, die vom Anwendungsprogramm genutzt wird. Als „virtuelle“ Adresse wird dagegen die 52-Bit Adresse bezeichnet, die temporär zur Bestimmung des Seitenrahmens genutzt wird.

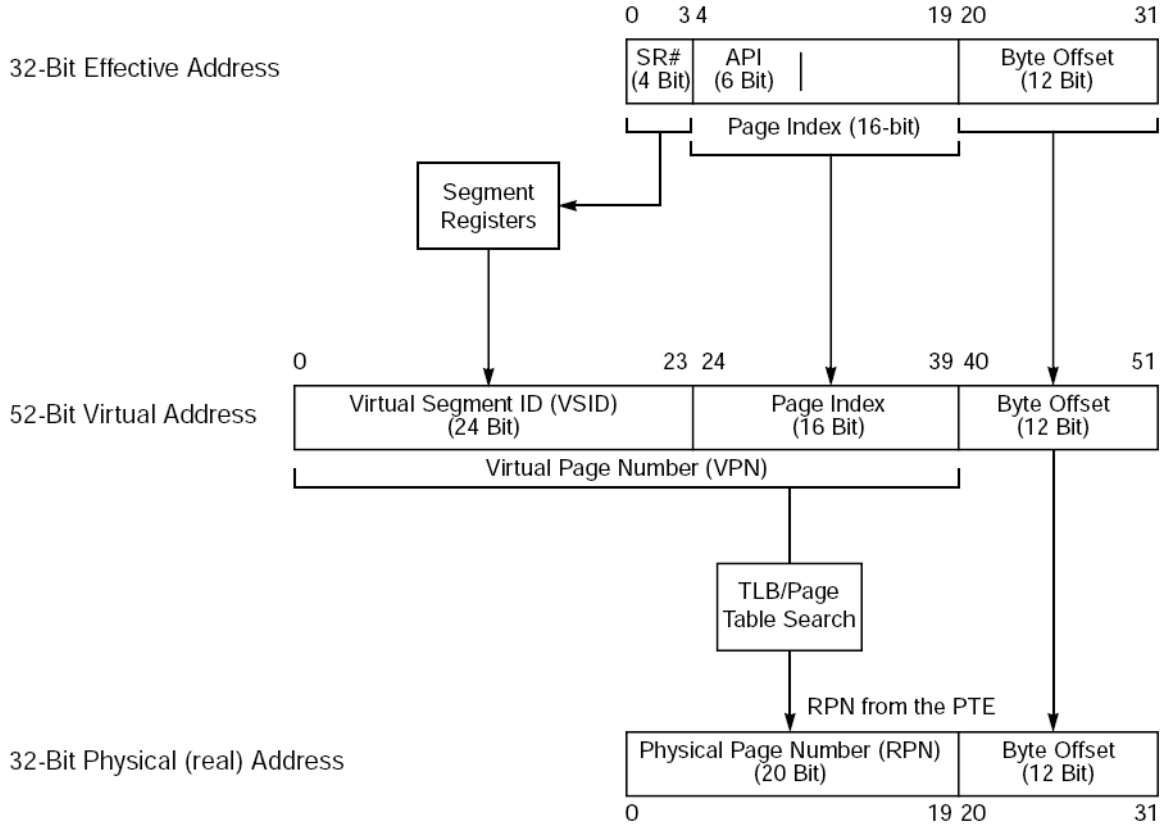

Abbildung 4.4: Paging-Verfahren der PowerPC-Architektur (Quelle: [IBM00])

Echtzeit: Das Dekrementer-Register wird mit einer Frequenz von Bus-Takt/4 dekrementiert. Er kann im Supervisor-Mode mit einem beliebigen ganzzahligen Wert $< 2^{32}$ geladen werden. Dies ermöglicht die Implementierung von Schedulern mit festen wie auch variablen Zeitscheiben. Wenn ein Scheduler variable Zeitscheiben nutzen kann, ist es ihm möglich, die nächste Unterbrechung dann festzulegen, wenn eine Änderung (z. B. ein Task-Wechsel) eintritt. Dadurch können unnötige Unterbrechungen vermieden werden. Die Präzision des Schedulers wird dabei allein vom Bus-Takt festgelegt. Aus dem Bus-Takt von 100 MHz beim Hellas-Board ergibt sich eine Scheduling-Präzision von $0.04\mu s$:

$$\frac{1}{\text{Dekrementer-Takt}} = \frac{4}{\text{Bus-Takt}} = 0.04\frac{\mu s}{\text{Takt}}$$

Partitionierung: Die Einteilung von Register- und Befehlssatz in Supervisor- und User-Model schützt die Kernfunktionen eines Systems vor absichtlicher oder unabsichtlicher Manipulation durch Anwendungsprogramme. Dies ist die Grundlage für eine räumliche und zeitliche Partitionierung.

Räumliche Partitionierung: Die räumliche Partitionierung wird durch die Block-Address-Translation und Paging unterstützt. Bei der BAT ist es möglich auf Block-Ebene Zugriffsrechte zu vergeben. Beim Paging können sowohl für Segmente wie auch für Seiten Zugriffsrechte vergeben werden.

Zeitliche Partitionierung: Die zeitliche Partitionierung wird durch den Dekrementer unterstützt. Wenn der Dekrementer unterläuft, d. h. das höchst-

wertige Bit von 0 auf 1 springt, wird ein Interrupt[2] ausgelöst. Bei einem Interrupt springt der PowerPC-Prozessor an eine definierte Stelle im Hauptspeicher. An diese Stelle hat der Kernel beim Booten spezielle Routinen geladen. Die Ausführung kehrt somit an den Kernel zurück, der nun über das weitere Vorgehen entscheiden kann.

Offene Standards: Die PowerPC-Architektur ist als Buch oder kostenlos als PDF-Dokument verfügbar. Der Entwurf sieht unterschiedliche Implementierungen der Architektur vor.

Modularität: Die drei Ebenen der PowerPC-Architektur gewähren Prozessor-Herstellern Freiheit bei der Implementierung und garantieren Binär-Kompatibilität für Anwendungsentwickler. Die Kernel Entwicklung bleibt jedoch an die Besonderheiten der Hardware-Implementierung gebunden.

Trusted Computing Base: RISC-Programme sind im Allgemeinen länger als CISC-Programme, da der RISC-Befehlssatz keine komplexen Befehle zur Verfügung stellt. Dafür ist ein RISC-Befehlssatz orthogonal und lässt sich damit leichter analysieren. Es wird für diese Arbeit angenommen, dass der Unterschied zwischen RISC- und CISC-Befehlssatz unwesentlich für die Trusted Computing Base ist.

4.2 Software

Aus der Analyse in Kapitel 3 geht hervor, dass die L4 μ-Kernel den IMA-Kriterien am Besten entsprechen. In Tabelle 3.2 werden die verfügbaren L4-Implementierungen aufgelistet. Die Frage welche Implementierung in dieser Arbeit zum Einsatz kommen soll, hängt auch von den verfügbaren Linux Portierungen ab.

4.2.1 Linux Portierungen auf den μ-Kernel L4

Die Abstraktionen von L4 stellen eine eigene virtuelle Hardware bereit. Damit Linux nicht mehr auf die echte sondern auf die virtuelle L4-Hardware zugreift, muss der Linux Kernel auf die L4-Architektur portiert werden. Dieser Prozess wird Para-Virtualisierung genannt. Ein para-virtualisierter Linux Kernel wird dann im unprivilegierten Modus (dem User-Model beim PowerPC) des Prozessors ausgeführt. Die frei verfügbaren para-virtualisierten Linux Kernel sind in Tabelle 4.1 aufgeführt.

Name	*μ-Kernel*	*Voraussetzung*
Afterburner	L4Ka::Pistachio	GNU Compiler Collection
Wombat	NICTA::Pistachio-embedded, OKL4	Embedded Systems Framework
L4Linux	Fiasco	L4Environment
User Mode Linux	–	Linux

Tabelle 4.1: Freie Para-Virtualisierungen des Linux Kernels für L4

[2]Was üblicher Weise Interrupt heißt, wird in der PowerPC-Architektur Exception genannt.

Theoretisch ließe sich jedes Linux aus Tabelle 4.1 mit jedem L4-Kernel aus Tabelle 3.2 kombinieren. Die möglichen Kombinationen lassen sich aber einschränken wenn man bedenkt, dass

- nur L4Ka::Pistachio für den PowerPC verfügbar und der Versuch einer Fiasco Portierung auf den PowerPC gescheitert ist [Lang06].
- es sinnvoll ist nur den Kernel zu betrachten für den das virtualisierte Linux geschrieben wurde.

Darum stehen nur Pistachio oder Pistachio-embedded zur Auswahl bei den L4-Implementierungen. Letzterer unterstützt zwar nicht die PowerPC-Architektur, er hat jedoch sehr viele Gemeinsamkeiten mit Pistachio, da er auf diesen zurück geht. Für beide Kernel muss der Plattform-Code für das Hellas-Board geschrieben werden. Er findet daher bei der folgenden Diskussion alternativer Arbeitspakete keine Erwähnung:

Afterburner und L4Ka::Pistachio: In persönlicher Kommunikation mit Joshua LeVasseur legt dieser dar, dass folgende Schritte notwendig sind, um Afterburner auf den PowerPC zu portieren:

- Afterburner Parser schreiben (bevorzugt in ANTLR[3]).
- Frontend für den In-Place Virtual-Machine-Monitor (IPVMM) schreiben.
- Evtl. Anpassungen am IPVMM-Backend.

L4Linux auf L4Ka::Pistachio: Für die IA32 Architektur existieren Portierungen des Linux Kernels 2.2.x und 2.4.x auf L4Ka::Pistachio. Beide werden nicht mehr unterstützt und außer dem Quellcode gibt es kaum Dokumentation. Mit L4Linux-2.6 existiert eine Portierung des aktuellen Linux Kernels. L4Linux-2.6 setzt auf dem L4Env Framework und dem Fiasco Kernel auf. Folgende Arbeitsschritte sind notwendig um L4Linux auf dem PowerPC einsetzen zu können:

- L4Env an das X.2 Kernel API anpassen
- L4Linux an das X.2 Kernel API anpassen
- Schreiben PowerPC-Subarchitektur für Linux.
- Evtl. sind in L4Env Abhängigkeiten zu IA32 zu beseitigen.

UML auf L4Ka::Pistachio: Hierbei müsste der Linux Kernel auf den L4 μ-Kernel portiert und eine Subarchitektur für PowerPC angelegt werden. Da UML bereits eine Subarchitektur für PowerPC besitzt, könnte man Teile davon wiederverwenden. Allerdings ist UML eine Portierung von Linux auf Linux. Es erwartet daher eine Linux-Umgebung. Die Anpassungen an L4 würden einer vollständigen Portierung gleichkommen. Die Diplomarbeit [Hohm96] könnte dabei als Vorlage dienen. Sie zeigt jedoch auch, dass dies ein Thema für eine eigenständige Abschlussarbeit ist.

[3]Homepage: http://www.antlr.org/

Wombat auf NICTA::Pistachio-embedded: Am NICTA hat man das X.2 API von L4 gemäß den Anforderungen von eingebetteten Systemen angepasst und unter dem Namen N1 veröffentlicht [ERTa05]. Das PowerPC ABI von Pistachio wurde noch nicht an das N1 API angepasst. Das NICTA verwendet darüber hinaus ein neues Build-System und bietet durch eine Sammlung von Bibliotheken (Framework) eine gewohnte Umgebung für die Entwicklung in C. Die architekturabhängigen Teile dieses Frameworks müssen für den PowerPC neu geschrieben werden. Zuletzt fehlt dem Linux Kernel noch die Subarchitektur für den PowerPC. Zusammengefasst sind bei dieser Lösung folgende Punkte abzuarbeiten:

- Anpassen der PowerPC Implementierung an das Build System.
- Anpassen des PowerPC ABI an das N1 Kernel API.
- Schreiben der architekturabhängigen Bibliotheksfunktionen.
- Schreiben der PowerPC-Subarchitektur für Linux.

L4Linux oder UML auf den L4Ka::Pistachio zu portieren, wäre eine umfangreiche Aufgabe, die im Rahmen dieser Arbeit nicht machbar scheint. Das Afterburner-Projekt für L4Ka::Pistachio würde eine intensive Auseinandersetzung mit dem Aufbau von Compilern allgemein und dem C-Compiler der GNU Compiler Collection (GCC) im Speziellen bedeuten. Das scheint jedoch eher ein Thema für eine separate Arbeit zu sein.

Für den Einsatz in IMA liegt es nahe, ein Projekt zu wählen das sich den Einsatz in eingebetteten Systemen auf die Fahnen geschrieben hat. Wombat Linux, basierend auf dem μ-Kernel Pistachio-embedded und dem Embedded Systems Framework (ESF) vom National Information and Communications Technology Australia (NICTA), scheint für diese Arbeit der vernünftigste Weg zu sein, Linux einzusetzen. Ein weiterer Grund der zu dieser Entscheidung beitrug ist die Zielsetzung von NICTA. Durch das Projekt L4.verified und dem seL4 API soll ein mathematisch verifizierter Kernel entwickelt werden [Elph04]. Das Ergebnis soll in NICTA::Pistachio-embedded einfließen und somit Stabilität und Sicherheit garantieren. Gerade für den Einsatz in IMA und der damit verbundenen Notwendigkeit zur Zertifizierung ist dies eine sehr hilfreiche Entwicklung.

4.3 Arbeitsschritte

Kleine überschaubare Arbeitsschritte erlauben, den Fortschritt eines Projekts zu kontrollieren und den Aufwand bei der Fehlersuche gering zu halten. In Abbildung 4.5 werden die Schritte dieser Diplomarbeit zusammengefasst. Die IMA-Anforderungen werden in Abschnitt 2.6 entwickelt. Sie führen in Abschnitt 3.2 zur Auswahl der L4-Architektur. In Abschnitt 4.2 wird entschieden, Wombat Linux mit Embedded Systems Framework auf dem μ-Kernel Pistachio-embedded einzusetzen. In diesem Abschnitt werden die nachfolgenden Arbeitsschritte beschrieben.

4.3.1 Toolchain

Bei L4Ka::Pistachio wie auch NICTA::Pistachio-embedded kommt die GNU Compiler Collection (GCC) zum Einsatz. Da die Entwicklungsumgebung auf einem Rechner

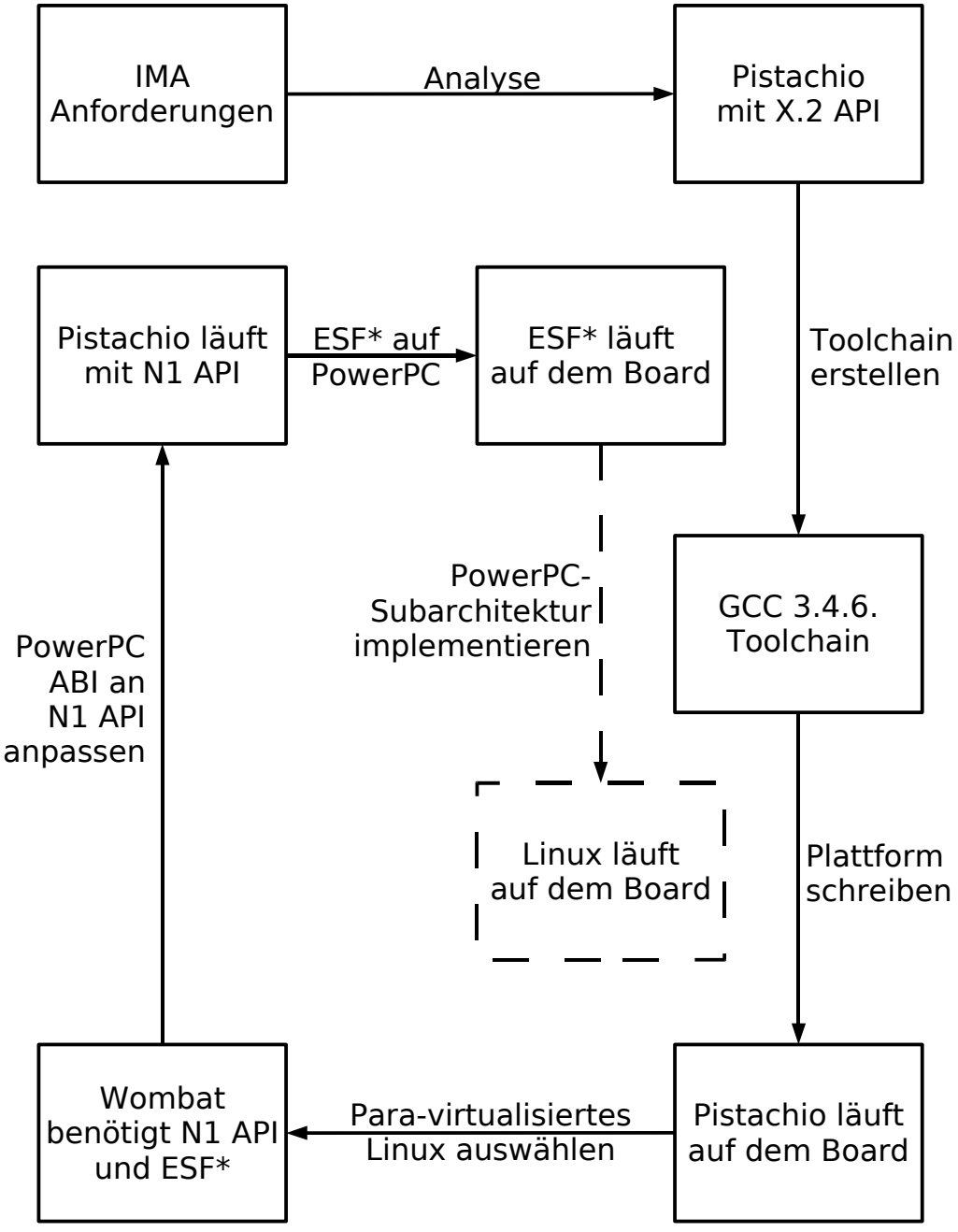

Abbildung 4.5: Arbeitsschritte

mit IA32 CPU läuft, das Hellas-Board aber einen PowerPC-Prozessor hat, muss ein Cross-Compiler zum Einsatz kommen. Ein Cross-Compiler wird durch Entwicklungs- und Zielplattform spezifiziert:

Entwicklungsplattform: 32-Bit Intel-Architektur mit Linux/Windows XP

Zielplattform: 32-Bit PowerPC mit Image im Executable and Linking Format (ELF)

Eine GCC-Toolchain für C besteht aus drei Komponenten:

1. Den Binärcode-Werkzeugen aus dem Paket `binutils`.
2. Dem C-Compiler aus der GNU Compiler Collection (`gcc`).
3. Der Standard C-Bibliothek (`libc`).

C-Compiler und C-Bibliothek sind wechselseitig voneinander abhängig. Diese Abhängigkeit kann nach [Schi03] durch folgende Vorgehensweise aufgelöst werden:

1. Einen einfachen C-Compiler ohne Unterstützung für die C-Bibliothek bauen.
2. Den einfachen C-Compiler nutzen um die C-Bibliothek zu bauen.
3. Den vollständigen C-Compiler bauen.

Zwischen C++-Compiler und Standard C++-Bibliothek existiert die gleiche wechselseitige Abhängigkeit. Auch hier führt diese Vorgehensweise zur Lösung.

Die C- und C++-Bibliotheken sind standardisiert und sollen die Anwendungsentwicklung vereinfachen. Für die Entwicklung von Pistachio-embedded und dem Embedded Systems Framework genügt jedoch ein einfacher C- und C++-Compiler. Die Funktionen der Standard C- und C++-Bibliothek werden dazu nicht benötigt.

4.3.2 Build-System

Das L4Ka-Projekt nutzt die GNU Build-Werkzeuge, NICTA setzt im ESF dagegen auf SCons, das in Python geschrieben wurde. Zeitlich fand der Wechsel des Build-Systems erst statt, nachdem das PowerPC ABI an das N1 API angepasst worden war. Das Build-System vom ESF bietet jedoch zwei Möglichkeiten, den Kernel zu bauen und erleichtert somit die Umstellung:

- `kernel_env.Pistachio()` ruft über ein Shell Skript die GNU Build-Werkzeuge auf.
- `kernel_env.Application("pistachio")` nutzt den SCons Build-Prozess.

Diese Arbeit unterstützt beide Methoden zum Bauen des PowerPC-Kernels.

Zuerst muss das Build-System auf die neue Architektur und Plattform eingestellt werden. Da der Quellcode für die PowerPC-Architektur von L4Ka::Pistachio auf den GNU Build-Prozess eingestellt war, müssen auch Anpassungen am Quellcode vorgenommen werden, damit der Code unter SCons kompiliert werden kann:

- Bei L4Ka automatisch eingebundene Header-Dateien müssen in den Quellcode-Dateien nun manuell eingebunden werden.
- Beim Erzeugen von „tcb_layout“ gerät der Compiler in eine Abhänigkeitsschleife. Das Problem wird durch die E-Mail in Anhang C beschrieben. Da keine Änderungen am Code außerhalb von Architektur und Plattform vorgenommen werden sollen, kann das Problem nur gelöst werden, indem `kernel/include/glue/v4-powerpc/space.h` in zwei Dateien aufgeteilt wird. Details können dem beiliegenden Quellcode in Anhang B entnommen werden.

4.3.3 Plattform

Der Plattform-Code wurde zu Anfang dieser Arbeit für L4Ka::Pistachio geschrieben, kann aber ohne zusätzliche[4] Änderungen in NICTA::Pistachio-embedded übernommen werden.

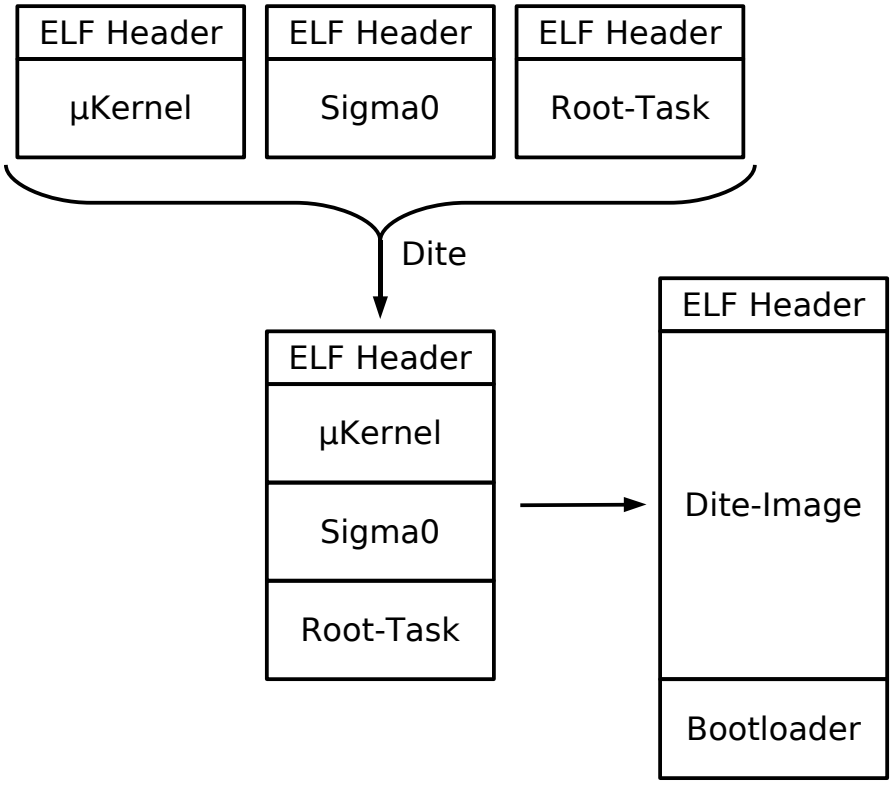

Abbildung 4.6: Build-Prozess beim Embedded Systems Framework

Der Plattform-Code des Kernels besteht aus vier Teilen:

Bootloader: Nach Einschalten des Hellas-Boards beginnt der Prozessor mit dem Befehl an Adresse `0xfff00000` den Boot-Vorgang. Ab dieser Adresse wird der Flash-Speicher eingeblendet, der das Boot-Image enthält. Der Build-Prozess durchläuft, wie in Abbildung 4.6 zu sehen ist, drei Stufen. Zuerst werden

[4] Abgesehen von den Änderungen durch den Wechsel des Build-Systems sind keine Änderungen am Quellcode notwendig.

die einzelnen System-Teile gebaut. Ein Werkzeug namens „Dite“ verschmilzt die verschiedenen Images zu einem Dite-Image. Das Boot-Image kapselt das Dite-Image und enthält zusätzlich den Bootloader Code. Damit der Code des Bootloaders ausgeführt werden kann, muss das Boot-Image vom ELF- in das Binärformat umgewandelt und in den Flash-Speicher geschrieben werden. Der Flash-Speicher des Hellas-Boards wird mit „SingleStep“, einem kommerzielles Werkzeug von Wind River, beschrieben.

Der Bootloader initialisiert die Marvell-Bridge und lädt das Dite-Image entsprechend der Angaben im ELF-Header in den Hauptspeicher. Dann springt der Bootloader in den Start-Code des Kernels. Die Adresse ist ebenfalls im ELF-Header hinterlegt.

Kernel Start: Nachdem der Bootloader in den Kernel gesprungen ist, beginnt die Initialisierung durch den Start-Code in Datei `kernel/src/glue/v4-powerpc/init.cc`. Dabei werden interne Datenstrukturen gefüllt und das Multi-Threading vorbereitet. Die Initialisierung endet mit der Übergabe des Prozessors an den Scheduler.

Interrupt Handling: Der Interrupt-Controller variiert von Plattform zu Plattform. Daher wird die Verwaltung des Interrupt-Controller im Plattform-Code des Kernels implementiert. Details zur Implementierung finden sich weiter unten im Text.

Linker Script: Das Linker Script legt fest, welche Teile des Kernels an welche Position im Hauptspeicher geladen werden müssen. Abbildung 4.7 zeigt den Aufbau der Pistachio-Images für den PowerPC. Die verschiedenen Positionen im Kernel-Image können durch Symbole referenziert werden. So kann z. B. über das Symbol `_start_data` auf den Datenbereich des Kernels zugegriffen werden. Die einzelnen Teile des Kernels werden dabei an Seitengrenzen ausgerichtet.

Es sollen hier noch zwei zentrale Themen des Plattform-Codes besprochen werden:

1. Das Interrupt-Handling beim PowerPC.
2. Der Umgang mit der Abhängigkeit der PowerPC-Implementierung von Open Firmware.

Interrupts

Der PowerPC hat nur eine Interrupt-Leitung. Damit können externe Interrupts entweder ganz oder gar nicht zugelassen werden. Zu den externen Interrupts zählt auch der Interrupt des Dekrementers. Allerdings springt der Prozessor verschiedene Routinen an, je nachdem ob der Interrupt von der Hostbridge oder dem Dekrementer ausgelöst wurde. Tritt ein Interrupt von der Hostbridge auf, sichert der Exception-Handler zuerst den Zustand des aktuell ausgeführten Threads und springt dann zum generischen Interrupt-Handler der PowerPC-Architektur. In Abbildung 4.8 ist der Interrupt-Handler der PowerPC-Architektur aus der Datei `kernel/src/glue/v4-powerpc/except_handlers.cc` zu sehen. Der Handler im Architektur-Code ruft wiederum den Handler der Plattform auf. Dabei übergibt er ihm immer die IRQ-

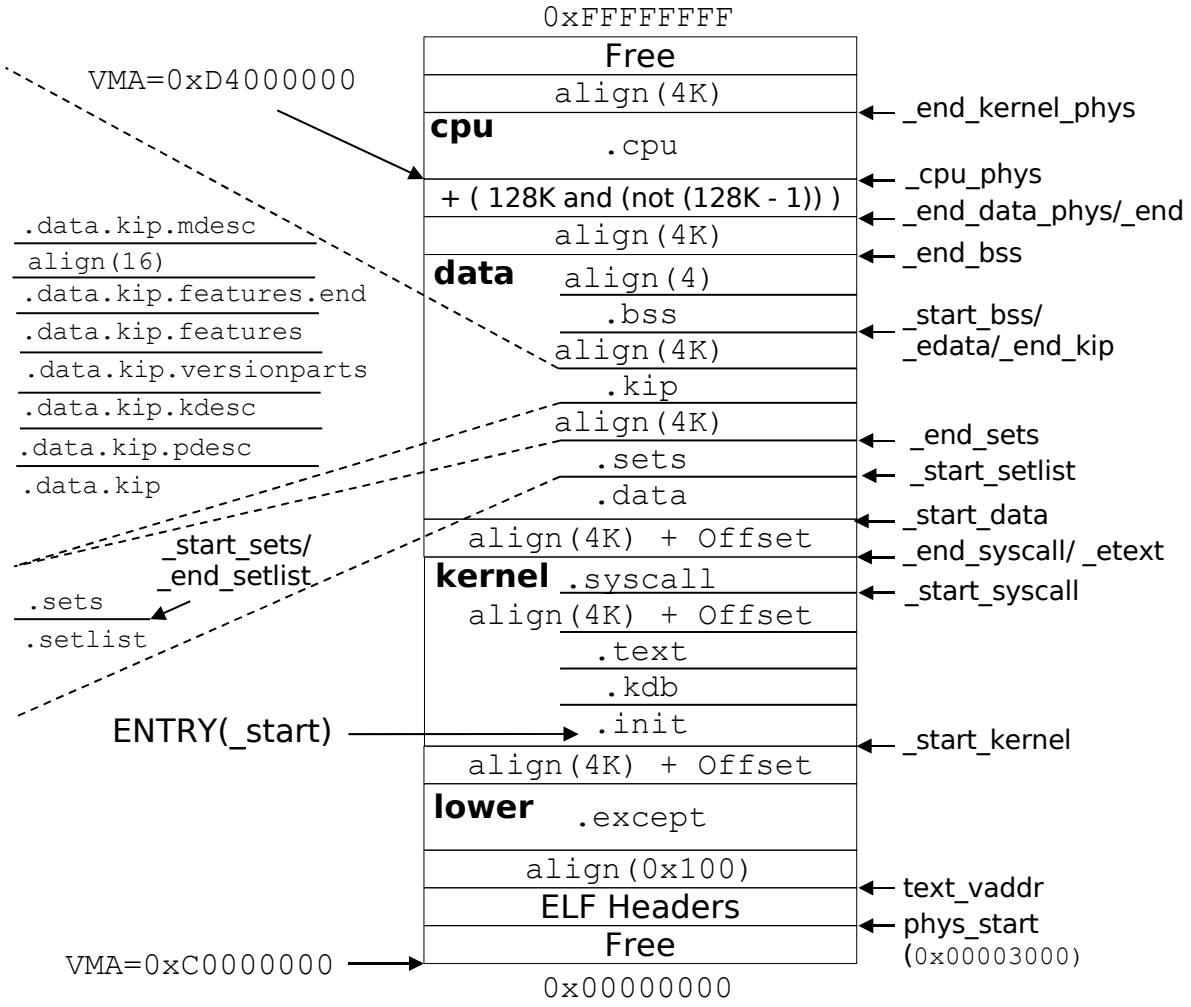

Abbildung 4.7: Das ELF-Image des Pistachio Kernels für PowerPC.

Nummer 0. Das ist bei einer einzigen Interrupt-Leitung des Prozessors auch verständlich. Es ist Aufgabe des Interrupt-Handlers im Plattform-Code, herauszufinden welche Komponente den Interrupt ausgelöst hat.

Die Marvell-Hostbridge hat einen hierarchisch aufgebauten Interrupt-Controller. Dadurch können Subsysteme selektiv de-/aktiviert werden. So lange sich kein User-Mode Handler für einen Interrupt angemeldet hat, erwartet der Kernel, dass kein Interrupt auftritt. Erst wenn sich ein Handler registriert hat, wird der entsprechende Interrupt demaskiert. Sollte zuvor ein Interrupt auftreten, bricht der Kernel mit einer Fehlermeldung ab.

Abbildung 4.9 zeigt einen Ausschnitt des Interrupt-Controller. Der Main-Controller wie auch die Interrupt-Controller der Subsysteme bestehen aus Mask- und Cause-Registern. Über das Mask-Register wird die Weitergabe von Interrupts an die nächsthöhere Ebene de-/aktiviert. Tritt ein Interrupt auf, kann über das Cause-Register der Auslöser in Erfahrung gebracht werden. Für diese Arbeit wurde nur ein Interrupt berücksichtigt: Wenn ein Zeichen über die serielle Schnittstelle eintrifft, wird im ersten Multi-Protocol-Serial-Controller (MPSC) der Interrupt Received-Control-Character (RCC) ausgelöst. Wenn der RCC-Interrupt im MPSC-Mask Register freigegeben ist,

```
EXCDEF( extern_int_handler )
{
  if( EXPECT_FALSE(ppc_is_kernel_mode(srr1)) ) {
    srr1 = processor_wake( srr1 );
    frame->srr1_flags = srr1;
  }

  get_interrupt_ctrl()->handle_irq( 0 );
  except_return();
}
```

Abbildung 4.8: Interrupt-Handler im PowerPC-Code

wird er an das höhere Main-Cause-Register weitergeleitet. Wenn nun wiederum die Interrupts des ersten MPSC freigegeben sind, wird der Interrupt an den Prozessor gemeldet. Wenn dieser nun seinerseits externe Interrupts zulässt, wird eine Exception ausgelöst und der Handler für externe Interrupts ausgeführt.

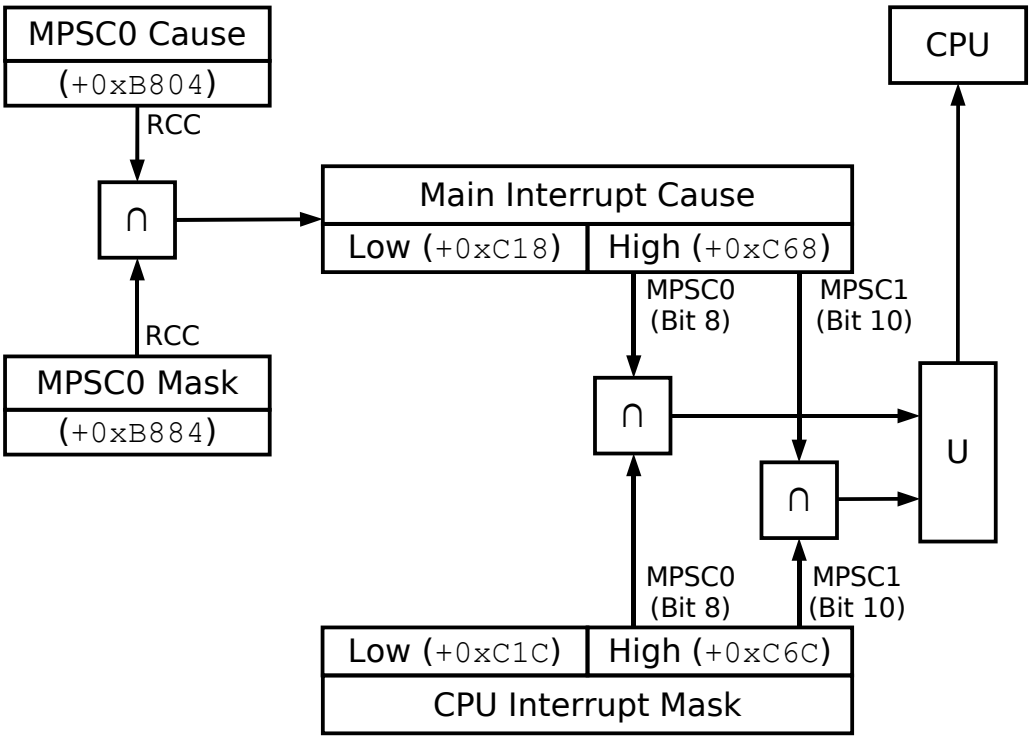

Abbildung 4.9: Hierarchischer Aufbau des Interrupt-Controllers der Hostbridge

Open Firmware

Der Open Firmware Standard definiert Funktionen und Datenstrukturen eines Bootloaders und ist in IEEE-1275 spezifiziert [Comm94]. Die Open Firmware kommt u. a. auf der „Common Hardware Reference Platform“ (CHRP) des PowerPC zum Einsatz, auf der auch manche Apple-Systeme basieren. Das Hellas-Board verwendet dagegen keine Firmware. Der Prozessor lädt den ersten Befehl aus dem Flash-Speicher

von Adresse `0xfff00000`. Die PowerPC-Implementierung von L4Ka::Pistachio erwartet jedoch Informationen über die Hardware in einem Geräte-Baum (Device-Tree) gemäß der Open Firmware Spezifikation. Es gibt nun zwei Alternativen für den Umgang mit dieser Abhängigkeit:

1. Open Firmware kann als spezifische Plattform betrachtet werden. Zugriffe auf Funktionen und Datenstrukturen der Open Firmware finden nur im plattformabhängigen Teil des Kernels statt.

2. Der Linux Kernel für den PowerPC erwartet vom Bootloader einen „flachen“ Device-Tree. D. h. neben dem Wurzelknoten besteht der Baum nur aus Blättern. Diese liefern Informationen zum Interrupt-Controller, den CPUs, Speicher, usw. Wenn das Board keine Open Firmware besitzt, muss der Bootloader einen flachen Device-Tree erstellen [HeBr05]. Auf jeden Fall muss der Speicherbereich des Device-Tree reserviert werden, damit der Kernel ihn nicht überschreibt oder für Nutzerprogramme frei gibt.

Da NICTA::Pistachio-embedded auf eingebetteten Systemen zum Einsatz kommen soll, deren Hardware-Ausstattung sich in der Regel nicht ändert, ist der Aufwand für eine Open Firmware nicht gerechtfertigt. Aus diesem Grund wurde der erste Weg gewählt, Open Firmware als separate Plattform zu betrachten.

4.3.4 Architektur

Der Architektur-Code ist in zwei Verzeichnissen organisiert. Der reine Architektur-Code ist in `arch/powerpc`, der Code zwischen Architektur und L4 API ist in `glue/v4-powerpc` zu finden.

Die Initialisierung des Kernels ist in `glue/v4-powerpc/init.cc` implementiert. Bei L4Ka::Pistachio wurde ein System mit Open Firmware vorausgesetzt. Diese Abhängigkeit wurde aus der Architektur entfernt und in die Plattform verschoben.

Es ist ein wesentlicher Teil dieser Arbeit, das PowerPC ABI an das N1 API in [ERTa05] anzupassen. Erst dadurch ist es möglich Pistachio-embedded auf dem PowerPC einzusetzen. Das PowerPC ABI von Pistachio war auf das X.2 API in [L4Ka04] ausgelegt. Die Veränderungen zwischen X.2 und N1 API sind in [ERTa05] aufgeführt und lassen sich folgender Maßen zusammenfassen:

- Die Inter Process Communication mit langen Nachrichten wurden entfernt.
- Die asynchrone Benachrichtigung (Notification) wurde hinzugefügt.
- Die Behandlung der Zeit wurde aus dem Kernel entfernt.
- Die Weiterleitung von Nachrichten wurde auf Thread-Basis eingeführt.
- Die Anzahl Nachrichtenregister wurde verringert.

Ein Teil dieser Änderungen wirkt sich auf die System-Calls aus. Die Logik der System-Calls ist im generischen Code des Kernels implementiert. Im Architektur-Code wird das Application Binary Interface (ABI) implementiert. Dies umfasst die

Registerbelegung bei Ein- und Austritt in/vom Kernel. Das ABI ist daher von der Hardware-Architektur abhängig. Die Implementierung des ABI ist im Verzeichnis `glue/v4-powerpc` zu finden, da es sich um die Schnittstelle zwischen API und Architektur handelt. Die Ein- und Austrittsroutinen des ABI müssen an die Parameter des N1 API angepasst werden. Die Veränderungen der System-Calls sind in Tabelle 4.2 zusammengefasst. Der Parameter `is_local` im System-Call `ExchangeRegisters` ist im X.2 API nicht dokumentiert, wird aber im ABI verwendet. Der System-Call `SystemClock` wurde ersatzlos gestrichen und ist daher nicht mit aufgeführt. Die folgenden System-Calls bleiben unverändert:

- `ThreadSwitch`
- `Unmap`
- `ProcessorControl`
- `MemoryControl`

Neben diesen rein syntaktischen Änderungen, müssen im Architektur-Code auch semantische Änderungen vorgenommen werden:

- Das N1 API kennt keine Lokale Thread ID (LTID) mehr. Das PowerPC ABI von X.2 sieht allerdings vor, dass der User-Thread-Control-Block (UTCB) an der Speicheradresse LTID liegt. Diese Abhängigkeit wurde aufgehoben. Thread ID und UTCB sind nunmehr frei[5] wählbar.
- Die neue Semantik der System-Calls hat viele Änderungen am UTCB notwendig gemacht. Außerdem werden im PowerPC-Code von Pistachio aus unerfindlichen Gründen negative Indizes für die Felder im UTCB verwendet. Daher wurde der UTCB von der ARM-Implementierung übernommen. Nach Änderungen in diversen Dateien erfolgt der Zugriff auf den UTCB nun durchgehend mit positiven Indices.
- Bei Pistachio-embedded kamen weitere Funktionen des Kernel-Debuggers hinzu. Der Kernel-Debugger (KDebug) ist nicht offiziell Teil des N1 API. Er ist aber für die korrekte Funktionsweise der PowerPC-Implementierung unerlässlich. KDebug bietet Dienste an, die über einen speziellen System-Call in Anspruch genommen werden können. Momentan wird z. B. die Ein- und Ausgabefunktionalität der `libc` über den Kernel-Debugger realisiert.
- Dem Thread-Control-Block von Pistachio-embedded wurden zwei Methoden hinzugefügt, die im Architektur-Teil implementiert werden müssen:
  - `copy_user_regs(tcb_t *src)`
  - `copy_regs_to_mrs(tcb_t *src)`

  Erstere Methode übernimmt die gesicherten Register des Quell-Threads als eigene, letztere kopiert sie in die Message-Register des UTCB.

[5] KIP- und UTCB-Area dürfen sich nicht überlappen und im Kernel-Bereich darf er auch nicht liegen.

System-Call		X.2 API	N1 API
IPC	in:	to_tid from_tid timeout	to_tid from_tid –
	out:	from_tid	from_tid
Schedule	in:	dest_tid time_control processor_control prio preemption_control	dest_tid ts_len total_quantum processor_control prio
	out:	result time_control –	result rem_ts rem_quantum
ThreadControl	in:	dest_tid space_tid scheduler_tid pager_tid – – utcb_location	dest_tid space_tid scheduler_tid pager_tid snd_redir_tid rcv_redir_tid utcb_location
	out:	result	result
ExchangeRegisters	in:	dest_tid control usp uip uflags uhandle pager_tid (is_local)	dest_tid control usp uip uflags uhandle pager_tid –
	out:	result control usp uip uflags pager_tid uhandle	result control usp uip uflags pager_tid uhandle
SpaceControl	in:	space_tid control kip_area utcb_area redirector_tid	space_tid control kip_area utcb_area –
	out:	result control	result control

Tabelle 4.2: Vergleich der System-Calls bei X.2 und N1 API

4.3.5 Embedded Systems Framework

Das Embedded Systems Framework (ESF) besteht aus dem L4 μ-Kernel, einigen Bibliotheken, der Dienstschicht Iguana und Wombat, einem para-virtualisierten Linux. In dieser Arbeit werden μ-Kernel und Linux immer separat erwähnt, weswegen sich „ESF“ immer nur auf die Bibliotheken und Iguana bezieht.

Nicht alle Bibliotheken sind von der Hardware-Architektur abhängig. Folgende Bibliotheken enthalten jedoch architekturspezifischen Code:

`libc`: Dies ist die Standard C-Bibliothek. Es müssen mehrere architekturabhängige Datentypen und -strukturen definiert und Funktionen implementiert werden:

- Definition architekturabhängiger Datentypen.
- Implementierung der erweiterten Sprungfunktionen.
- Implementierung der Assembler Start-Routinen (CRT0) für Bootloader, Root-Server, Iguana-Server und Iguana-Anwendungen.
- Definition und Implementierung der System-nahen E/A-Schnittstelle. Dabei wird die E/A-Funktion des Kernel-Debuggers verwendet.

```
typedef uint32_t cycle_t;

static inline cycle_t get_cycles( void )
{
  cycle_t cycles;
  /* We assume that the kernel configured upmc1 for cycle counting. */
  asm volatile ( "mfspr %0, %1" : "=r" (cycles) : "i" (CYCLE_REG) );
  return cycles;
}
```

Abbildung 4.10: Implementierung von get_cycles()

`libcycles`: Diese Bibliothek dient zur Performanz-Messung. Es wird eine architekturspezifische Funktion zum Auslesen des Prozessor-Zyklus-Zählers benötigt. Die Implementierung ist in Abbildung 4.10 zu sehen.

`libiguana`: Diese Bibliothek kapselt den Zugriff auf Dienste des Iguana-Servers. Es muss der Startcode für Iguana-Threads in Assembler implementiert werden.

`libl4`: Diese Bibliothek kapselt das System-Call ABI durch bereitstellen eines architekturunabhängigen API für C und C++. Dem C/C++-Programmierer wird durch Funktionen und Datentypen die Nutzung der μ-Kernel Schnittstelle erleichtert. Das API entspricht größten Teils der Spezifikation in [ERTa05]. Diese Bibliothek war bereits für L4Ka::Pistachio mit PowerPC-Unterstützung verfügbar. Daher mussten „nur“ die Änderungen und Neuerungen des ABI eingearbeitet werden:

- Anpassen des System-Call ABI an die Funktionen des N1 API.
- Hinzufügen von Funktionen zur Nutzung der neuen Kernel-Debugger Dienste.
- Überarbeiten der Funktionen zum Zugriff auf den UTCB.
- Löschen des System-Calls `SystemClock` aus dem ABI.

`libmutex`: Diese Bibliothek stellt die Funktionalität für wechselseitigen Ausschluss (Mutex) zur Verfügung. Basierend auf den Beispielen in [IBM00] wurde die Funktion `try_lock` in Assembler implementiert. Wie in Abbildung 4.11 zu sehen ist, kam der vom PowerPC-Prozessor bereit gestellte Mechanismus zum Reservieren von Speicherstellen (bzw. -blöcken) zum Einsatz.

```
try_lock:
  lwarx  %r5, 0, %r3  /* r5 = mtx->holder       */
  cmpw   0, %r5, %r4  /* if (mtx->holder == me) */
  beq-   2f           /*   goto return_success  */
  cmpwi  0, %r5, 0    /* if (mtx->holder != 0)  */
  bne    3f           /*   goto return_fail     */
1:                    /* else                   */
  stwcx. %r4, 0, %r3  /*   mtx->holder = me     */
  bne-   3f           /* if (!success)          */
                      /*   goto return_fail     */
2:                    /* return_success */
  li %r3, 1
  blr
3:                    /* return_fail */
  li %r3, 0
  blr
```

Abbildung 4.11: Implementierung von try_lock()

Das ESF erwartet neben obigen Bibliotheken auch die Implementierung eines Timer-Treiber im unprivilegierten Modus. Welcher Treiber zum Einsatz kommt, hängt vom Zeitgeber ab, den die Hardware bereit stellt. Mit Timebase- und Dekrementer-Register bietet der PowerPC-Prozessor bereits alle Mechanismen an, die ein Timer-Treiber benötigt. Daher wurde für diese Arbeit ein solcher Treiber im Verzeichnis `libs/drv_powerpc_timer` implementiert. Es ist eine „Quick-and-dirty“ Implementierung mit vielen Nachteilen:

- Es kann sich kein User-Mode Treiber für den Dekrementer-Interrupt registrieren, da er vom μ-Kernel intern verwendet wird. Ein Timer-Thread muss somit aktiv auf den Time-out warten. Damit der Timer-Thread sofort aktiv werden kann, wenn der Time-out eintritt, müsste er eine höherer Priorität haben als andere User-Threads. Das kann er aber nicht, denn der Timer-Thread kann nicht von einem Interrupt „aufgeweckt“ werden, sondern muss aktiv warten.

Damit würde er wiederum alle Threads mit geringerer Priorität blockieren. Deswegen muss der Timer-Thread mit gleicher Priorität ausgeführt werden, wie andere User-Threads. Dies führt jedoch zu einer großen Varianz der Reaktionszeit, da ein Time-out nur festgestellt werden kann, wenn der Timer-Thread gerade aktiv ist.

- Selbst wenn der Timer gerade nicht benötigt wird, wartet er aktiv darauf, dass er zurück gesetzt wird. Diese Lösung musste gewählt werden, weil der System-Call `L4_WaitForever()`, aus Gründen die nicht mehr ausgeforscht werden konnten, zurückkehrt.

Die Implementierung des Timers ist hauptsächlich dazu gedacht, die Abhängigkeit von Iguana zu erfüllen. Sie ist keinesfalls für den Einsatz in Echtzeit-Systemen geeignet!

4.4 Zusammenfassung

Von der EADS Deutschland GmbH wurde für diese Diplomarbeit das Entwicklungsboard der Laser Range Radar Komponente „Hellas“ zur Verfügung gestellt. Auf dem Hellas-Board kommt als Prozessor ein PowerPC 750Cxe von IBM zum Einsatz. Die PowerPC-Architektur erlaubt einem Scheduler die Timer-Interrupts über ein Register zu programmieren. Zwei Memory Management Units stellen dem Kernel Mechanismen zur Durchsetzung der Partitionierung zur Verfügung.

Von den L4-Kerneln aus Kapitel 3 ist nur Pistachio auf den PowerPC portiert worden. Am National ICT Australia (NICTA) wurde Pistachio für den Einsatz in eingebetteten Systemen weiterentwickelt und unter dem Namen „Pistachio-embedded“ veröffentlicht. Pistachio-embedded implementiert das N1 API, das aufgrund der veränderten Anforderungen aus dem X.2 API von Pistachio abgeleitet wurde. Der μ-Kernel wird am NICTA im Rahmen des Embedded Systems Framework (ESF) eingesetzt. Das ESF stellt eine Reihe Bibliotheken, sowie die Dienstschicht Iguana bereit. Für Pistachio-embedded mit ESF ist ein para-virtualisiertes Linux namens „Wombat“ verfügbar. Allerdings wird die PowerPC-Architektur vom NICTA nicht unterstützt. Um die Kombination aus Pistachio-embedded, ESF und Wombat auf dem Hellas-Board einsetzen zu können, sind folgende Arbeitsschritte notwendig gewesen:

- Erstellen einer GCC-Toolchain
- Anpassen des Build-Systems an PowerPC-Architektur und Hellas-Board
- Anpassen des PowerPC ABI an das N1 API
- Implementieren der architekturabhängigen Bibliotheksfunktionen für den PowerPC
- Implementieren der PowerPC-Subarchitektur für Linux

Aus Zeitmangel konnte der letzte Arbeitsschritt nicht mehr durchgeführt werden. Alle anderen Arbeitsschritte wurden im Rahmen dieser Arbeit erfolgreich abgeschlossen. Der Iguana-Server und einige Testanwendungen können nun auf dem Hellas-Board ausgeführt werden. Um die Testanwendungen, die nicht erfolgreich ausgeführt werden können, geht es in Abschnitt 5.1.

5. Evaluierung

Im Rahmen dieser Arbeit wurde erreicht, dass

1. Pistachio-embedded nun die PowerPC-Architektur unterstützt.
2. das Embedded Systems Framework (ESF) die PowerPC-Architektur unterstützt.
3. das ESF auf dem Hellas-Board läuft, wie in Abbildung 5.1 zu sehen ist.

Die Evaluierung muss in dieser Arbeit qualitativ durch die gewonnene Erfahrung erfolgen. Für eine empirische Evaluierung fehlte einerseits die Zeit. Andererseits ist es fraglich, ob sie in dieser Arbeit sinnvoll durchführbar gewesen wäre. Die Messung der Interrupt-Latenz und des Jitters mit/ohne Last werden gewöhnlich als Kennzahlen bei Echtzeit-Betriebssystemen angeführt. Die Erhebung der Kennzahlen macht allerdings nur Sinn, wenn sie vergleichbar sind. Vergleichbarkeit ist nur gegeben, wenn alle Messungen auf der gleichen Hardware durchgeführt werden. Dazu müssten verschiedene Kernel für das Hellas-Board verfügbar sein. Da das Hellas-Projekt VxWorks als Betriebssystem einsetzt, hätte der VxWork-Kernel als monolithische Referenz verwendet werden können. Dann hätten Benchmarks auf beiden Systemen implementiert werden müssen. Dies hätte eine Einarbeitung in VxWorks notwendig gemacht. Die Auswahl und Implementierung aussagekräftiger Benchmarks, wie sie in [HMJS+97] verwendet wird, braucht einige Zeit. In einer Diplomarbeit kann leider keine Portierung und ausführliche Evaluierung zusammen realisiert werden. Daher ist in dieser Arbeit nur eine qualitative Evaluation der Ergebnisse möglich.

Im Abschnitt 5.1 werden die IMA-Kriterien auf die PowerPC-Implementierung von Pistachio-embedded und das ESF angewendet. Die Schlussfolgerungen daraus werden in Abschnitt 5.2 diskutiert.

5.1 IMA-Tauglichkeit

5.1.1 Echtzeit

Ob die Ausführungen zur Echtzeit von [Ruoc06] in der Praxis zutreffen, muss weitestgehend offen bleiben. Festgestellt werden kann, dass der Scheduler mit festen Zeit-

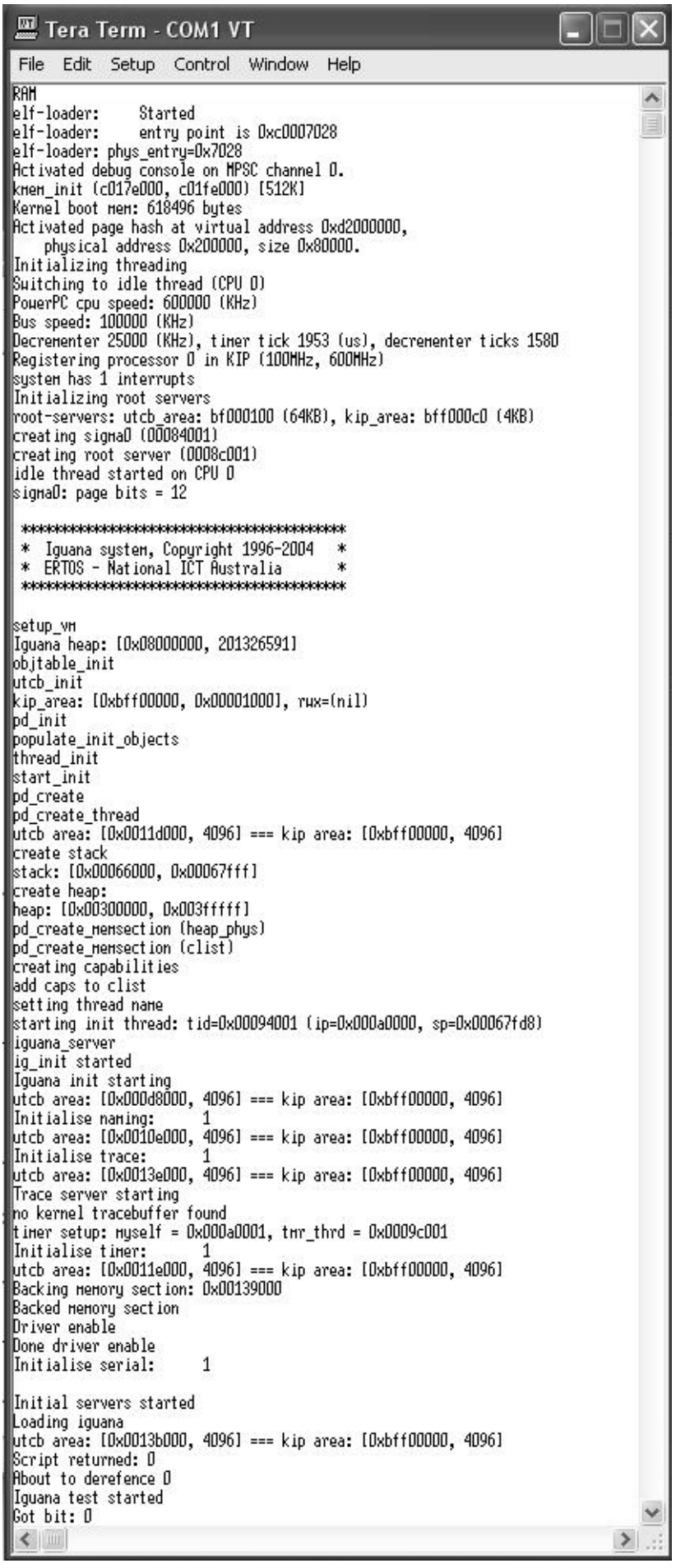

Abbildung 5.1: Boot-Vorgang von Pistachio-embedded

scheiben arbeitet. Die Länge ist in der Datei `pistachio/kernel/include/glue/v4-powerpc/config.h` definiert: `#define TIMER_TICK_LENGTH (1953)`. Maßeinheit sind μ-Sekunden. D. h. eine Zeitscheibe ist etwa 2 ms lang.

Die Arbeitsweise des PowerPC-Prozessors wird durch das Machine-State-Register (MSR) festgelegt. In der Datei `pistachio/kernel/include/arch/powerpc/msr.h` sind verschiedene Prozessor-Zustände definiert. Nur im User-Mode sind externe Interrupts von vornherein erlaubt. Die Ausführungspfade im Kernel-Mode sind also nicht preemptibel.

5.1.2 Partitionierung

Die PowerPC-Implementierung nutzt sowohl Block-Address-Translation (BAT) wie auch Page-Address-Translation. Das Kernel-Image wird beim Boot-Vorgang in den unteren physikalischen Speicherbereich nach den Exception-Handlern geladen. Effektiv beginnen die Adresse des Kernels dagegen bei `0xC0000000`. Die Abbildung von effektiver auf physikalische Adresse wird durch BAT erreicht.

Abbildung 5.2 zeigt das Layout des Speichers im User-Model des PowerPC. Die ersten drei Gigabyte des effektiven Adressraums sind mit Ausnahme der Hostbridge-Adressen frei verfügbar. Das letzte Gigabyte ist für den Kernel reserviert. Der Kernel-Code wird über den Eintrag in IBAT0 und die Register der Hostbridge über den Eintrag in DBAT2 abgebildet. Die Abbildung der Hostbridge durch ein BAT-Register im User-Model ist eine Schwachstelle der Implementierung. Denn sie erlaubt jedem Thread lesenden und schreibenden Zugriff auf die Hostbridge. Dieser Schritt war notwendig, da der System-Call `MemoryControl` für den PowerPC nicht implementiert wurde. Dieser System-Call ist aber u. a. dazu gedacht, die Caching-Attribute von Speicherseiten zu kontrollieren. Bei Speicherseiten im Hauptspeicher ist Caching wünschenswert, weswegen es standardmäßig aktiviert ist. Für den Zugriff auf die Register der Hostbridge sollte Caching dagegen deaktiviert sein. Ohne `MemoryControl` System-Call ist dies jedoch nicht erreichbar. Daher musste ein DBAT-Eintrag erstellt werden, der auch Treibern im User-Model direkten Zugriff auf die Hostbridge ermöglicht.

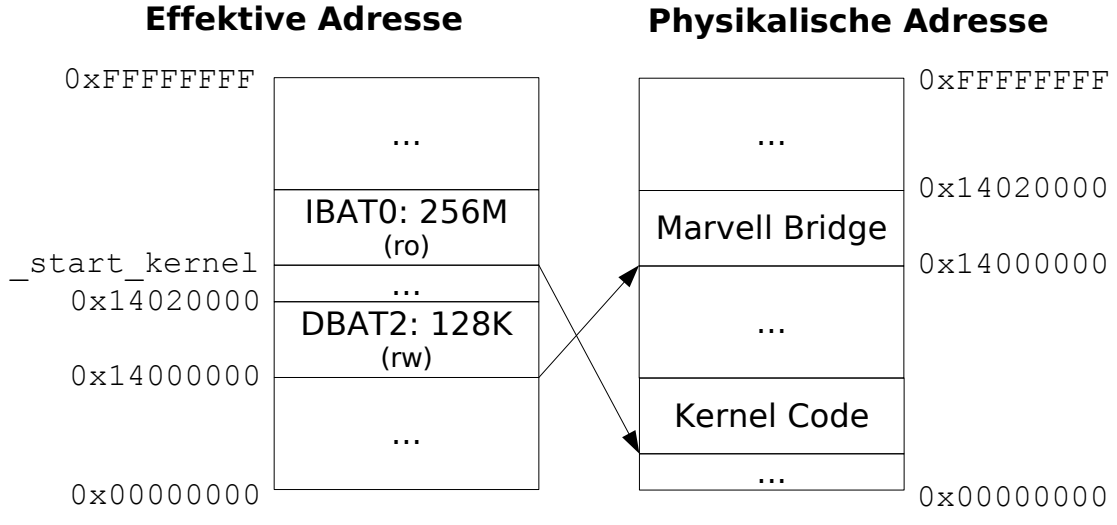

Abbildung 5.2: BAT-Abbildungen im User-Model

Die aktiven BAT-Einträge im Supervisor-Model sind in Abbildung 5.3 zu sehen. Die Bedeutung der Einträge in Daten-BAT (DBAT) und Befehls-BAT (IBAT) sind folgende:

IBAT0: Abbildung auf den Kernel-Code.

IBAT1: Identische Abbildung der ersten 128 KB auf die Exception-Handler.

DBAT0: Abbildung auf die Kernel-Daten.

DBAT1: Abbildung auf die Seiten-Tabelle der MMU. Bereich und Größe wird beim Start des Kernels festgelegt.

DBAT2: Identische Abbildung auf die Register der Hostbridge.

DBAT3: Abbildung auf Prozessor-Daten bei Mehrprozessor-Systemen.

Die IBAT-Einträge erlauben nur lesenden, die DBAT-Einträge lesenden und schreibenden Zugriff.

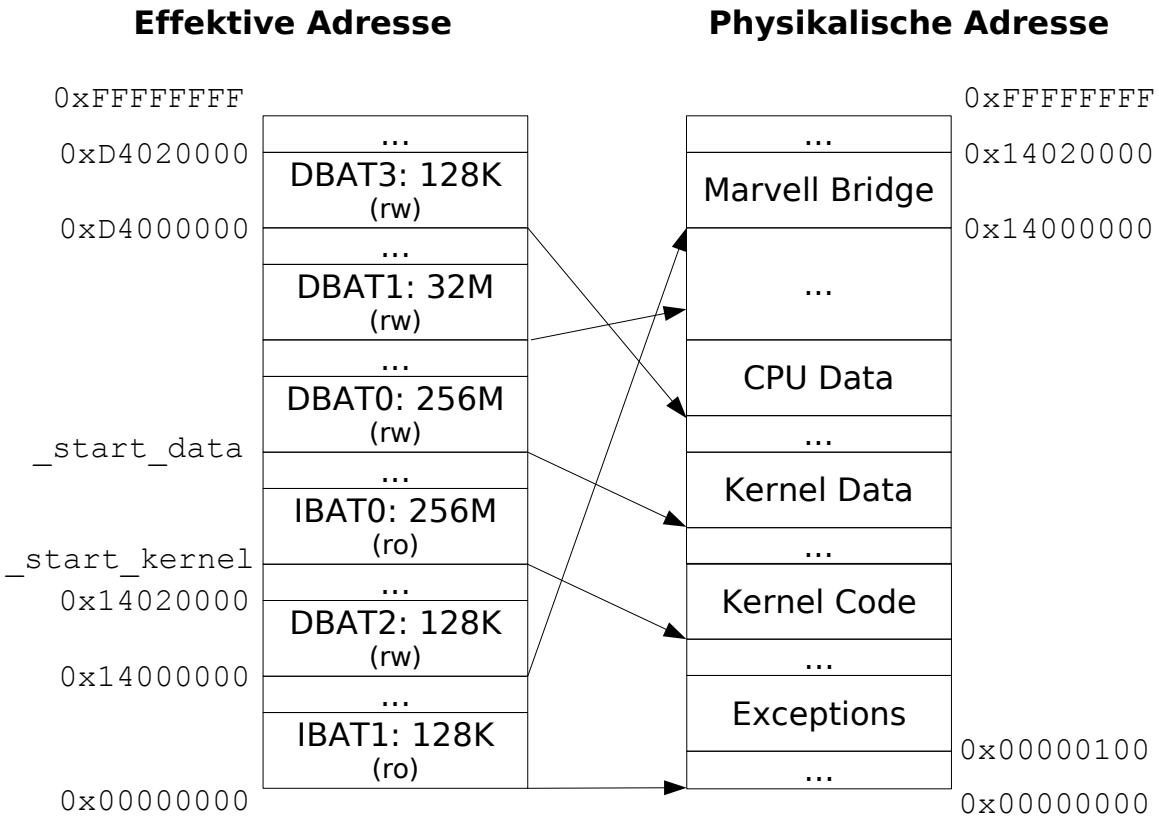

Abbildung 5.3: BAT-Abbildungen im Supervisor-Model

Im Laufe dieser Arbeit wurde ein Bug im Paging-System entdeckt: Sei P ein Pager, A ein Thread mit P als Pager und p_0 eine beliebige Seite. Wenn A das erste Mal auf p_0 lesend zugreift, wird ein Pagefault generiert und an P geschickt. P erlaubt nun A lesenden Zugriff auf p_0 indem es die Pagefault-Nachricht entsprechend beantwortet. A kann jetzt weiter ausgeführt werden und greift kurz danach schreibend auf p_0 zu. P erlaubt A daraufhin auch schreibenden Zugriff auf p_0. Nun sollte A normal ausgeführt

werden können. Tatsächlich erzeugt es aber weitere Pagefaults. Nach Debugging und persönlicher Kommunikation mit Ben Leslie liegt nahe, dass der Fehler im Handling des TLB liegt. Wenn im TLB der alte Eintrag nicht ungültig gemacht wird, geht die MMU weiterhin davon aus, dass auf die Seite nur lesend zugegriffen werden kann und löst eine Pagefault-Exception.

5.1.3 Trusted Computing Base

Zur TCB gehören alle grundlegenden Komponenten des Systems. Beim ESF umfasst dies den μ-Kernel, die Bibliotheken, den Iguana-Server und Sigma0-Pager. Mit der LoC-Metrik aus Abschnitt 3.1 ergeben sich für die PowerPC-Architektur und das Hellas-Board folgende Werte:

Pistachio-embedded: ca. 11.000 LoC

Bibliotheken: ca. 39.000 LoC

Iguana: ca. 8.000 LoC

Das ergibt eine TCB von ca. 58.000 LoC. Vom Umfang entspricht dies dem Xen-Hypervisor. Wobei beim Xen-Projekt neben dem Hypervisor auch Domain0 zur TCB gehört. Auf jeden Fall liegt der Wert deutlich unter dem des Linux Kernels.

Von „trusted“ kann aber leider keine Rede sein. Denn Iguana konnte nur nach Beheben diverser Bugs und durch einige Workarounds zum Laufen gebracht werden. Zum Beispiel ist die Struktur `internal_memsection` in Abbildung 5.4 vom Typ `memsection_list`. Diese Liste wird in Abbildung 5.5 mit Referenzen auf sich selbst

```
[...]
struct memsection_list
{
  struct memsection_node* first;
  struct memsection_node* last;
};

struct memsection_node
{
  struct memsection_node* next;
  struct memsection_node* prev;
  struct memsection data;
};
[...]
```

Abbildung 5.4: Code-Fragment aus: iguana/server/src/memsection.h

initialisiert. Später im Code wird diese Liste erweitert, wobei die ersten Referenzen als Start- und Stop-Anker erhalten bleiben. Die Funktion `pd_init` aus der Datei `iguana/server/src/pd.c` durchläuft diese Liste und übernimmt die Einträge

```
[...]
struct memsection_list internal_memsections =
{ (void *)&internal_memsections, (void *)&internal_memsections };
[...]
```

Abbildung 5.5: Code-Fragment aus: iguana/server/src/memsection.c

als Speicher-Objekte in eine Tabelle. Beim Durchlaufen der Liste wird im original Code nicht darauf geachtet, dass Start- und Stop-Anker keine Referenzen zu Knoten sind, sondern Referenzen zur Liste selbst. Dies führt später zu einem Pagefault, wenn auf die `memsection` zugegriffen wird, auf die der angebliche Knoten verweist. Der Workaround in Abbildung 5.6 prüft jeden Eintrag auf Ungleichheit mit `internal_memsection` bevor er als Speicher-Objekt in die Tabelle von Iguana übernommen wird. Dieses Beispiel illustriert den Eindruck den der Code an vielen

```
[...]
if ( node != (void*)&internal_memsections )
  objtable_insert(&node->data);
[...]
```

Abbildung 5.6: Code-Fragment aus: iguana/server/src/pd.c

Stellen macht: „Quick-and-dirty“. Daher ist anzunehmen, dass Iguana noch weitere unentdeckte Bugs enthält.

Im Verzeichnis `libs/check` befindet sich der Quellcode des Unit Test Framework „Check“[1]. Für manche Bibliotheken existieren Test-Suits im jeweiligen Unterverzeichnis. Es wurden nur die Tests für Komponenten durchgeführt, an denen etwas geändert wurde:

- Die Test-Suite für die Standard C-Bibliothek `libc`, ohne Test des Dateisystems.
- Die Test-Suite für die Mutex-Bibliothek `libmutex`.

Zusätzlich sind im Verzeichnis `example` einige Tests zu finden, die nicht auf „Check“ basieren. Von diesen Tests wurden folgende erfolgreich durchgeführt:

- Der Iguana-Test in `example/iguana`.
- Der Semaphor-Test in `example/semaphor`.
- Der Timer-Test in `example/timer`.

[1] Homepage: http://check.sourceforge.net

Fehlgeschlagen sind dagegen folgende Tests:

- Der Speicher-Management-Test in `example/memsection` bricht nach 15 Schleifendurchläufen mit einer Fehlermeldung auf dem IPC-Pfad im Kernel ab. Halbiert man die Größe des allozierten Speichers, verdoppelt sich die Anzahl der Schleifendurchläufe. Die Ursache dafür ist ungeklärt. Es liegt nahe, dass der Fehler auf einen Bug im Paging-Subsystem des Kernels zurück geht.
- Der Thread-Management-Test in `example/threadsmash` bricht bei etwa 20 Threads mit einer Fehlermeldung ab. Diese besagt, dass der Kernel nicht genug Speicher hat, um einen Thread-Control-Block anzulegen. Die Ursache für diesen Fehler könnte ein Bug im Kernel-Memory-Manager sein oder auf eine fehlerhafte Initialisierung des Kernel-Speichers zurück gehen.

Während der Suche nach anderen Fehlern wurde außerdem festgestellt, dass die „Global Thread ID“ im User-Thread-Control-Block nicht gesetzt wird.

5.1.4 Offene Standards

Das original L4 API von Jochen Liedtke trägt den Namen V2. Es kommt nur noch im DROPS-Projekt zum Einsatz. V2 war auf die 32-Bit Intel-Architektur zugeschnitten, weswegen die Arbeit an einem neuen API begonnen wurde. Die experimentellen APIs der X-Reihe sollten die Portierbarkeit des Kernels erhöhen. Das erste experimentelle API X.0 wird ebenfalls vom DROPS-Projekt genutzt. Das L4Ka-Projekt setzt dagegen das X.2 API ein, das als V4 API offizieller Nachfolger des V2 API werden sollte. Am NICTA hat man das X.2 API an die Bedürfnisse von eingebetteten Systemen angepasst und unter dem Namen N1 veröffentlicht. Im Repository der Entwickler von Pistachio-embedded ist bereits die Rede von einem N2 API, das aber noch nicht veröffentlicht wurde. Die Open Kernel Labs haben im Frühjahr 2007 die L4-Distribution OKL4 veröffentlicht und mit ihr das gleichnamige API. Einen Überblick über die verfügbaren APIs gibt [L4-H].

So scheint es, dass jedes L4 Projekt meint, ein eigenes Kernel API veröffentlichen zu müssen. Dies kann man durchaus als Nachteil offener Standards ansehen. Diese Veränderungen beruhen aber größtenteils auf einem Lernprozess. So wurde die X-Reihe geschaffen, weil das V2 API zu sehr auf die IA32 zugeschnitten war. Die N-Reihe soll den Bedürfnissen eingebetteter Systeme besser gerecht werden. Es bleibt nur zu hoffen, dass sich das Kernel API für eingebettete Systeme bald stabilisiert und von allen Projekten akzeptiert wird.

5.1.5 Modularität

Die Vielfalt an APIs wirkt sich natürlich auch negativ auf die Modularität aus, denn L4-Anwendungen sind nicht ohne Änderungen portierbar. Auch die verfügbaren Kernel können nicht einfach ausgetauscht werden, denn sie implementieren unterschiedliche APIs. Jedes API, das nach V2 entwickelt wurde, geht auf dieses zurück. Weder die experimentellen noch die N1 APIs wurden komplett neu entwickelt. Sie basieren alle auf einem Vorgänger-API. So sind zumindest die Grundkonzepte Threads, Adressräume und IPC überall vorhanden. Eine Portierung von API zu API ist daher keine komplexe Aufgabe, auch wenn sie besser zu vermeiden wäre.

5.2 Fazit

Die Implementierung des System-Call `MemoryControl` fehlt und der PowerPC-Code des Kernels enthält noch einige Bugs. Daher kann das Entwicklungsstadium der PowerPC-Implementierung von Pistachio-embedded und der Dienstschicht Iguana als Alpha-Version eingestuft werden. Wenn die bekannten Fehler behoben sind, kann mit dem Beta-Test begonnen werden. Momentan ist das Embedded Systems Framework für den PowerPC jedoch nicht zum Einsatz in IMA geeignet!

Auch das Vertrauen in die Trusted Computing Base ist gering. Denn die wechselseitigen Abhängigkeiten im Quellcode sind sehr verworren. Die Implementierung der Methoden des Thread-Control-Block ist über diverse Dateien verstreut und nicht dokumentiert. Generell ist die Dokumentation des Quellcodes dürftig. Die Suche nach den oben beschriebenen Bugs wäre eine gute Gelegenheit den Quellcode neu zu strukturieren. Dabei sollen Methoden der Softwaretechnik zum Einsatz kommen, wie z. B. Klassen- und Zustandsdiagramme.

Trotz all dieser Mängel war die Wahl des Embedded Systems Framework richtig. Der Weg zu einem para-virtualisierten Linux war im Rahmen dieser Arbeit zeitlich jedoch nicht machbar. Für den L4 Ansatz sprechen folgende Erkenntnisse:

- Das L4 API definiert eine eigene Hardware-Architektur, die von der realen Hardware abstrahiert. Ein erfahrener Entwickler kann das ESF in weniger als zwei Monaten auf eine neue Plattform portieren. Dem Kernel eine neue Architektur hinzuzufügen dauert mindestens ein halbes Mann-Jahr. Mit IA32, ARM und nun PowerPC sind die gängigsten Architekturen jedoch bereits implementiert.
- Innerhalb eines Jahres könnte das ESF und Pistachio-embedded für PowerPC zur Einsatzreife gebracht werden. Die Implementierung der PowerPC-Subarchitektur für Wombat Linux wird mindestens weitere sechs Monate dauern.
- Mit OKL4 wurde auch eine umfangreiche Dokumentation veröffentlicht. Da nun Vollzeit-Entwickler an dem Kernel arbeiten, kann man auf eine Verbesserung der Quellcode-Dokumentation hoffen. Allerdings unterscheidet sich das OKL4 API deutlich vom N1 API. Daher ist der Aufwand zur Anpassung des PowerPC ABI an das OKL4 API umfangreicher als die Anpassung, die im Rahmen dieser Arbeit durchgeführt wurde.
- Wombat Linux wird in einem eigenen Adressraum ausgeführt. Fehler im Linux Kernel können theoretisch nicht das ganze System zum Absturz bringen. Ob sich diese Erwartung in der Praxis erfüllt, müsste man durch systematische Tests verifizieren.

Ein Schwachpunkt der L4-Architektur ist, dass Adressräume nicht in Containern zusammengefasst werden können. Mit solchen Containern könnten Partitionen im Sinne von [AEEC03] erstellt werden. Bisher gibt es zwei Möglichkeiten, ARINC-Partitionen zu erstellen:

- Eine Partition wird durch einen Adressraum realisiert. Innerhalb der Partition existiert dann aber kein Speicherschutz. Jeder Thread kann Daten und Code der anderen Threads, insbesondere auch den des Scheduler-Threads, überschreiben.

- Eine Partition wird durch mehrere Adressräume modelliert. Damit wären die verschiedenen Teile voneinander geschützt, insbesondere auch der Scheduler-Thread. Die Nachteile dieser Variante sind zusätzliche Kontextwechsel innerhalb der Partition und eine komplexere Implementierung.

Die Domains des Xen-Projekts kommen den ARINC-Partitionen aus [AEEC03] viel näher, da ein hierarchischer Scheduler implementiert wurde. Die Domains werden vom Scheduler des Hypervisors verwaltet, und innerhalb einer Domain kommt der Scheduler des Gast-Kernels zum Einsatz.

5.3 Zusammenfassung

Im Rahmen dieser Arbeit wurde erreicht, dass der μ-Kernel Pistachio-embedded und das Embedded Systems Framework von NICTA die PowerPC-Architektur unterstützen. Da bereits bei Pistachio die Implementierung des System-Call `MemoryControl` fehlt und im PowerPC-Code von Pistachio-embedded mehrere Bugs gefunden wurden, ist ein Einsatz in IMA noch nicht möglich. Die Dienstschicht Iguana konnte nur nach diversen Bugfixes und Workarounds zum Laufen gebracht werden. Daher muss das Projekt als Entwicklungsversion im Stadium „Alpha“ angesehen werden. D. h. es sind noch nicht alle Funktionen implementiert und die Software wurde teilweise noch nicht getestet.

Pistachio-embedded nutzt statische Zeitscheiben und ist nicht preemptibel. Ob es in der Praxis harte Echtzeit-Bedingungen einhält, konnte nicht getestet werden.

Die Partitionierung findet bei L4-Systemen auf Ebene von Adressräumen statt. Eine Partitionierung nach [AEEC03] kann nur approximiert werden, da Adressräume nicht in Container zusammengefasst werden können. Somit kann eine ARINC-Partition entweder innerhalb eines Adressraums realisiert oder über verschiedene Adressräume verteilt werden. Innerhalb einer Partition kann Scheduling durch einen Scheduler-Thread und den Mechanismus der „Timeslice-Donation“ erfolgen.

Die Offenheit des L4 API hat momentan fünf verschiedene Versionen zur Folge. Die Vielfalt der Kernel APIs führt wiederum dazu, dass sich Teile des Systems nicht austauschen lassen.

Eine alternative Lösung wäre Xen gewesen. Es bietet mit dem Domain-Konzept eine Partitionierung, die jener in [AEEC03] näher kommt als L4. Aber erstens bietet Xen keine Abstraktion der Hardware an und zweitens gibt es keine Implementierung für die 32-Bit Architektur des PowerPC. Daher ist trotz der vielen Schwächen und Mängel das Embedded Systems Framework die richtige Wahl für diese Arbeit gewesen.

6. Zusammenfassung und Ausblick

6.1 Zusammenfassung

In dieser Arbeit wird dargelegt, dass eine Para-Virtualisierung der vernünftigste Weg ist, Linux in der Luftfahrt einzusetzen. Ausgehend von einer Architektur, die dem Entwurfsprinzip der Integrated Modular Avionic folgt, ergeben sich für ein Betriebssystem folgende Anforderungen:

- Das System muss harte Echtzeit-Bedingungen erfüllen.
- Räumliche und zeitliche Partitionierung muss die stochastische Unabhängigkeit von Fehlern garantieren.
- Eine minimale Trusted Computing Base verringert potentielle Bugs und den Aufwand bei der Zertifizierung.
- Offene Standards gewährleisten die Austauschbarkeit von Komponenten.
- Eine IMA-Architektur ist aus wohl definierten Komponenten aufgebaut.

Ein reines Linux System kann diese Anforderungen nicht erfüllen. Durch die monolithische Architektur wird erstens zuviel Code im privilegierten Modus ausgeführt und zweitens eine Portierung des Kernels aufwändiger. Nur durch den Einsatz eines zweiten Kernels können harte Echtzeit-Bedingungen erfüllt werden. Dies steigert jedoch die Komplexität des Systems.

Ausgehend von der Analyse in [Benn03] werden folgende Kernel in Betracht gezogen:

- Der μ-Kernel Minix 3 wurde für ein Mehrbenutzer-Betriebssystem gleichen Namens entwickelt.
- Xen ist ein Hypervisor für para-virtualisierte Gast-Betriebssysteme.
- L4 ist ein Kernel API mit diversen Implementierungen. Für den Einsatz in eingebetteten Systemen wurden die μ-Kernel Fiasco und Pistachio-embedded entwickelt.

Minix 3 wurde für den Einsatz auf dem Desktop entwickelt. Harte Echtzeit-Bedingungen kann es nicht erfüllen, denn die Implementierung des Prozess-Managers nutzt Swapping und der Scheduler verändert die Prioritäten der Prozesse. Durch seine μ-Kernel Architektur hätte Minix 3 jedoch grundsätzlich das Potential zu einem Echtzeit-Betriebssystem.

Xen bietet eine ähnliche Partitionierung an, wie sie in [AEEC03] spezifiziert ist und hat einen Echtzeit-Scheduler implementiert. Inwiefern das Interrupt-Handling deterministisch ist, lässt sich auf den ersten Blick nicht sagen. Nachteilig für die Portierbarkeit ist jedoch, dass Xen auf eine Abstraktion der Hardware verzichtet. Es exportiert vielmehr eine Teilmenge der realen Hardware in die virtuellen Maschinen.

Die Kernelschnittstelle L4 definiert hingegen eine abstrakte Hardware bestehend aus Adressräumen, Threads und Kommunikationskanälen. Die L4-Implementierungen Pistachio-embedded und Fiasco sollen nach [Ruoc06] harte Echtzeit-Bedingungen erfüllen können. Die Architektur als μ-Kernel minimiert die Trusted Computing Base und erhöht zusammen mit der Hardware-Abstraktion die Modularität. Daher wurde für diese Arbeit eine Lösung auf Basis der L4-Architektur gewählt.

Wombat ist eine Para-Virtualisierung von Linux für die Kernelschnittstelle L4. Es wurde zusammen mit dem Embedded Systems Framework (ESF) und dem μ-Kernel Pistachio-embedded am National ICT Australia entwickelt und setzt diese voraus. Das Entwicklungsboard wurde für diese Arbeit von der EADS Deutschland GmbH gestellt und wird mit einem PowerPC-Prozessor betrieben. Die Portierung von Pistachio-embedded, dem ESF und Wombat auf die PowerPC-Architektur umfasst folgende Arbeitsschritte:

1. Erstellen einer GCC-Toolchain
2. Anpassen des Build-Systems an PowerPC-Architektur und Board
3. Anpassen des PowerPC ABI an das N1 API
4. Implementieren der architekturabhängigen Bibliotheksfunktionen
5. Implementieren der PowerPC-Subarchitektur für Wombat Linux

Im Rahmen dieser Arbeit wurden die ersten vier Arbeitsschritte erfolgreich abgeschlossen. Pistachio-embedded und ESF können auf dem Entwicklungsboard ausgeführt werden. Bereits der PowerPC-Implementierung des Vorgängers von Pistachio-embedded fehlt die Implementierung des System-Calls `MemoryControl`. Darüber hinaus sind im PowerPC-Code von Pistachio-embedded einige Bugs bekannt, und die Dienstschicht Iguana konnte nur nach diversen Bugfixes und Workarounds zum Laufen gebracht werden. Für die Implementierung der PowerPC-Subarchitektur des Linux Kernels hat am Ende die Zeit gefehlt.

6.2 Ausblick

Um mit Beta-Tests beginnen zu können, müssen die bekannten Fehler beseitigt und die fehlenden Teile des Kernel implementiert werden. Die nächsten Arbeitsschritte in Richtung dieses Ziels wären:

- Die beiden Bugs im Code der Seitenverwaltung beheben. Wobei die Dateien reorganisiert und der Code dokumentiert werden sollte.
- Den Bug im Kernel-Memory-Manager beheben und die Funktionsweise im Quellcode dokumentieren.
- Die Stelle finden, an der die Global Thread ID im User-Thread-Control-Block gesetzt werden sollte, es aber nicht wird und den Bug beheben.
- Den System-Call `MemoryControl` implementieren, dokumentieren und testen.

Nach Abschluss dieser Arbeiten kann der Beta-Test beginnen. In einer systematischen Reihe von Tests sollte die Korrektheit der Kernel-Funktionen überprüft werden. Darauf könnten spezielle IMA-Tests folgen, die Echtzeit- und Partitionierungseigenschaften überprüfen. Beispielsweise können Messungen der Interrupt-Latenz zur Bewertung der Echtzeit-Eigenschaften vorgenommen werden. Dabei sollten sich die Werte eines ausgelasteten Systems nicht signifikant von den Werten eines leer laufenden Systems unterscheiden. Die Partitionierung kann z. B. durch die Implementierung asozialer Threads und die Messung der Auswirkungen bewertet werden. Wenn der Kernel ausgiebig getestet wurde, müsste die Dienstschicht Iguana überarbeitet werden. Zum Testen von Kernel und Iguana kann das Unit Test Framework „Check“ genutzt werden, das bereits in das ESF integriert wurde.

Aufgrund der aktuellen Entwicklungen stellt sich allerdings die Frage, ob sich dieser Aufwand lohnt. Denn mit OKL4 wurde im Frühjahr 2007 eine komplett überarbeitete Version von Pistachio-embedded, dem ESF und Wombat veröffentlicht. OKL4 wird von den Open Kernel Labs[1], einer Ausgründung des NICTA, entwickelt und unter BSD-Lizenz vertrieben. OKL4 implementiert das gleichnamige Kernel API, eine Weiterentwicklung des N1 API. Eine Alternative wäre, diese Entwicklung zu nutzen und den PowerPC-Code entweder neu zu schreiben oder zu überarbeiten, so dass er in OKL4 integriert werden kann.

Auch von anderen Projekten gibt es Neuigkeiten. Im Laufe dieser Arbeit wurde die Portierung von Minix 3 auf den PowerPC veröffentlicht [Alti06] und in den Code der Entwicklungsversion integriert. Wenn ein Echtzeit-Scheduler und ein Prozess-Manager ohne Swapping implementiert würden, könnte mit Minix 3 ein weiterer μ-Kernel als Referenz für umfangreiche IMA-Tests genutzt werden.

Der Hypervisor Xen wurde bisher nur auf die 64-Bit Architektur des PowerPC 970 portiert. Im Quellcode ist zwar die Implementierung der 32-Bit Architektur vorgesehen, inwiefern daran gearbeitet wird, ist jedoch nicht ersichtlich.

[1] Homepage: http://www.ok-labs.com/

A. Abkürzungen

Abkürzung	Beschreibung
ABI	Das Application Binary Interface legt die Ein- und Austrittspfade des Kernels sowie die Belegung der Register mit Parametern fest.
ANTLR	Another Tool for Language Recognition
APEX	Die Application/Execution Schnittstelle wird vom ARINC Standard 653-1 spezifiziert.
API	Das Application Programming Interface kapselt die ABI durch Funktionsaufrufe in einer Programmiersprache.
ARINC	Die Aeronautical Radio Inc. entwickelt Spezifikationen für die Luftfahrt-Industrie.
AUTOSAR	Die Automotive Open System Architecture entwickelt Spezifikationen für die Automobil-Industrie.
CHRP	Die Common Hardware Reference Platform wurde als Standard für PowerPC-Rechner entwickelt.
CPU	Die Central Processing Unit ist das Herzstück eines Rechners.
DROPS	Das Dresden Real-Time Operating System Project entwickelt ein Betriebssystem für harte Echtzeit-Anwendungen auf Basis des L4 μ-Kernels Fiasco.
EA	Die effektive Adresse ist bei der PowerPC-Architektur die Adresse mit der Software Daten referenziert. Sie muss nicht mit der physikalischen Adresse übereinstimmen.
EADS	Die European Air, Defence and Space Company ist ein europäischer Luft- und Raumfahrt Konzern.
ELF	Das Executable and Linking Format ermöglicht das Laden von komplexen Programmen oder Betriebssystemen.
ESF	Das Embedded Systems Framework von NICTA umfasst μ-Kernel Pistachio-embedded, diverse Bibliotheken, die Dienstschicht Iguana und das para-virtualisierte Linux Wombat.
DO-178B	Bezeichnung für ED-12B in den USA.
ED-12B	Standard zur Zertifizierung von Software für die zivile Luftfahrt.
EUROCAE	European Organisation for Civil Aviation Equipment

GCC	Die GNU Compiler Collection ist eine Sammlung von Werkzeugen zur Software-Entwicklung, die unter einer Open Source Lizenz zur Verfügung gestellt werden.
IEEE	Institute of Electrical and Electronics Engineers
IBM	International Business Machines Corporation
IMA	Die Integrated Modular Avionic ist ein Entwurfsprinzip in der Luftfahrt-Industrie.
IPC	Die Inter-Process-Communication ist ein synchroner Mechanismus zur Kommunikations in Betriebssystemen.
IPVMM	Ein In-Place Virtual Machine Monitor kommt im Afterburner Projekt zum Einsatz. Die Para-Virtualisierung wird dabei vom Compiler vorgenommen.
L4Ka	Das L4 Projekt an der Universität Karlsruhe hat den μ-Kernel Pistachio entwickelt.
LoC	Lines-of-Code ist eine triviale Metrik zur Bewertung von Software-Komplexität.
LRU	Least-Recently-Used ist eine Ersetzungsstrategie für Cache-Speicher.
MMU	Die Memory Management Unit ist Teil des Prozessors. Sie stellt Mechanismen zum Umgang mit virtuellem Speicher und zum Schutz von Speicherbereichen zur Verfügung.
MPSC	Der Multi-Protocol-Serial-Controller ist ein Baustein auf der Hostbridge von Marvell, die im Hellas-Board zum Einsatz kommt.
NDA	Ein Non-Disclosure Agreement ist ein Vertrag, der Informationen vor der Veröffentlichung schützt.
NICTA	Das National Information and Communications Technology Australia ist ein Forschungsinstitut.
OEA	Die Operating Environment Architecture ist die unterste Schicht der PowerPC-Architektur. Sie soll die Portabilität von Betriebssystemen gewährleisten.
OKL	Die Open Kernel Labs sind eine Firma, die aus dem NICTA heraus gegründet wurden.
OSEK	Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug
PI	Der Page Index ist Teil der Adresse einer Speicherseite beim Paging der PowerPC-Architektur. Mehr Informationen finden sich in Abschnitt 4.1.
RTCA	Die Radio Technical Commission for Aeronautics entwickelt Empfehlungen für die Zulassung von Fluggeräten, die wie z. B. DO-178B in Standards münden können.
SEU	Bei einem Single-Event-Upset verursacht ein Hoch-Energie-Teilchen eine Veränderungen in einem Halbleiter-Bauteil. Dies kann z. B. zu einem Bit-Flip führen.
TCB	Die Trusted-Computing-Base bezeichnet den Teil einer Software-Architektur, der unbedingt korrekt funktionieren muss. Fehler in der TCB können das gesamte System lahmlegen.
TCB	Der Thread-Control-Block ist eine Klasse im μ-Kernel Pistachio.

UISA	Die User Instruction Set Architecture ist die oberste Ebene der PowerPC-Architektur. Sie soll Binärkompatibilität für Anwendungen gewährleisten.
UML	User-Mode-Linux ist ein para-virtualisiertes Linux zur Ausführung im User-Mode von Linux.
UTCB	Der User-Thread-Control-Block ist eine Klasse im μ-Kernel Pistachio. Im Gegensatz zum Thread-Control-Block hat ein User-Thread vollen Zugriff auf den UTCB.
VDX	Vehicle Distributed eXecutive verfolgte wie OSEK die Definition einer standardisierten Schnittstelle für die Automobil-Industrie.
VEA	Die Virtual Environment Architecture liegt in der PowerPC-Architektur zwischen UISA und OEA. Sie spezifiziert Aspekte der Speicher- und Zeitverwaltung.
VMM	Ein Virtual Machine Monitor (oder auch Hypervisor) verwaltet und organisiert die virtuellen Maschinen, auf denen Gast-Systeme ausgeführt werden.
VSID	Die Virtual Segment Identification kommt beim Paging der PowerPC-Architektur zum Einsatz. Mehr Informationen finden sich in Abschnitt 4.1.

B. Inhalt der CD

Die beiliegende CD enthält den Quellcode des Embedded Systems Framework mit den Änderungen aus dieser Diplomarbeit:

+- Iguana		
|	|- apps	Enhält eine Shell für Iguana.
|	|- build	In diesem Verzeichnis wird das System-Image kompiliert.
|	|- configs	Enhält Konfigurationsdateien für verschiedene Boards.
|	|- doc	Enthält die Dokumentation zur Iguana Dienstschicht.
|	|- example	Enthält Stand-alone Tests als Beispiel-Code.
|	|- iguana	Enthält den Quellcode der Iguana Dienstschicht.
|	|- libs	Enthält den Quellcode der verschiedenen Bibliotheken.
|	|- linux_apps	Enthält Anwendungen für Wombat.
|	|- loaders	Enthält den Boot-Loader.
|	|- package.cfg	Beschreibt das Iguana-Paket.
|	|- packages	Listet die Unterpakete auf.
|	|- pistachio	Enthält den Quellcode des μ-Kernels.
|	|- SConstruct	Ist das Äquivalent zum „Makefile“ für SCons.
|	|- test	Enthält Konfiguration des Unit Test Frameworks „Check“.
|	|- tools	Enthält Werkzeuge und Erweiterungen des Build-Systems.
|	+- wombat	Enthält den para-virtualisierten Linux Kernel 2.6.9.
|- Iguana.README		
+- Crosstools.README		

C. Abhängigkeitsproblem

Date: Tue, 23 Jan 2007 18:42:06 +0100
From: "Christian, Martin, OPEE45" <Martin.Christian@eads.com>
To: <l4ka@ira.uni-karlsruhe.de>
Subject: Solving a dependency circle

This is not really a L4 question! However, it's a real problem for me, while adapting the PowerPC arch to NICTAs L4.N1 API. The include files create a circle of dependency. There are common solutions to this, however this ones seems to be special or at least out of my mind.

I created this simple example:

<Alpha.h>

```
#ifndef ALPHA_H_
#define ALPHA_H_

#include "Alpha_inline.h"
class Beta;

class Alpha {
public:
        inline void doSomething( Beta* b );
};

#endif /*ALPHA_H_*/
```

</Alpha.h>

<Alpha_inline.h>

```
#ifndef A_INLINE_H_
#define A_INLINE_H_
```

```
#include "Alpha.h"
#include "Beta.h"

/* Inline methods */
inline void Alpha::doSomething( Beta* b ) {
        b->helpAlpha();
}

#endif /*A_INLINE_H_*/
</Alpha_inline.h>


<Beta.h>
#ifndef BETA_H_
#define BETA_H_

#include "Alpha.h"

class Beta {
private:
        Alpha alpha;
public:
        inline Alpha* getAlpha();
        inline void useInline();
        inline void helpAlpha();
};

/* Inline methods */
inline Alpha* Beta::getAlpha() {
        return &alpha;
}

inline void Beta::useInline() {
        this->getAlpha()->doSomething( this );
}

inline void Beta::helpAlpha() {
        for ( volatile int i=3D0; i<1000; i++ );
}

#endif /*BETA_H_*/
</Beta.h>



<main.cpp>
#include "Beta.h"

int main( int argc, char** argv ) {
```

```
        Beta b;

        b.useInline();
        return 0;
}
</main.cpp>

Compile with: gcc -Wall main.cpp -o dependency

The solution would be to include "Alpha_inline." in main.cpp. However,
the file belongs to the "API", which means I can't change it.

For the real problem, use this mapping for arch=powerpc:
Alpha -> GLUE(pgent.h)
Alpha_inline -> GLUE(pgent_inline.h)
Beta -> GLUE(space.h)
main.cpp -> i. e. API(exregs.cc)

Would someone know a solution? For some reason I haven't discovered,
L4Ka::Pistachio avoids this problem.

Cheers,

Martin.
```

Literatur

[AEEC97] Airlines Electronic Engineering Committee. ARINC Report 651-1: Design Guidance For Integrated Modular Avionics, November 1997. Prepared by the Airlines Electronic Engineering Committee.

[AEEC03] Airlines Electronic Engineering Committee. ARINC Specification 653-1: Avionics Application Software Standard Interface, Oktober 2003. Prepared by the Airlines Electronic Engineering Committee.

[Aeol02] Aeolean Inc. Introduction to Linux for Real-Time Control. Introductory Guidelines and References for Control Engineers and Managers 2002-12-11T1531, National Institute of Standards and Technology, 2002.

[AFHH+06] M. Aiken, M. Fähndrich, C. Hawblitzel, G. Hunt und J. Larus. Deconstructing Process Isolation. In *ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, Oktober 2006.

[Alti06] I. A. Alting. MinixPPC: A port of the MINIX OS to the PowerPC platform. Diplomarbeit, Vrije Universiteit Amsterdam, September 2006.

[AUTO05] AUTOSAR GbR. Requirements on Operating System. Internet: `http://www.autosar.org`, Februar 2005. Version 2.0.1.

[Bara97] M. Barabanov. A Linux-based Real-Time Operating System. Diplomarbeit, New Mexico Institute of Mining and Technology, Juni 1997.

[BDFH+03] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt und A. Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the 19th ACM Symposium on Operating Systems Principles*, New York, USA, 2003. ACM Press, S. 164–177.

[Benn03] M. D. Bennett. *A Kernel For IMA Systems*. Dissertation, University of York, Juli 2003.

[Bibl01] Bibliographisches Institut. *Duden – Fremdwörterbuch*. Bibliographisches Institut, Mannheim. 7.. Auflage, 2001.

[Brie06] V. Briegleb. Der Pinguin geht auf U-Boot-Jagd. heise newsticker: `http://www.heise.de/newsticker/meldung/76165`, Juli 2006.

[CGLW06] Carrier Grade Linux Working Group. Carrier Grade Linux Requirements Definition Overview. Internet: `http://www.osdl.org`, Februar 2006. Version 3.2.

[Chen02] A. M. K. Cheng. *Real-Time Systems: Scheduling, Analysis, and Verification*. John Wiley & Sons. 2002.

[Comm94] IEEE 1275 Technical Committee. *IEEE Standard 1275-1994 — Standard for Boot (Initialization Configuration) Firmware: Core Requirements and Practices*. IEEE. 1994.

[Corb06] J. Corbet. The 2006 Linux Kernel Summit. Internet: `http://lwn.net/Articles/KernelSummit2006/`, Juli 2006.

[DoMa03] L. Dozio und P. Mantegazza. Linux Real Time Application Interface (RTAI) in low cost high performance motion control. Internet: `http://www.aero.polimi.it/~rtai/documentation/conferences/motioncontro% l2003.pdf`, März 2003. Vortrag auf der Motion Control 2003, einer Konferenz der Associazione Nazionale Italiana per l'Automazione.

[Elph04] K. Elphinstone. Future directions in the evolution of the L4 microkernel. Technical Report 0401005T-1, National ICT Australia, Oktober 2004.

[EnKO95] D. R. Engler, M. F. Kaashoek und J. O'Toole Jr. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, Copper Mountain Resort, Colorado, Dezember 1995. S. 251–266.

[ERTa05] Embedded Real-Time and Operating Systems Group. *NICTA L4-embedded Kernel Reference Manual, Version NICTA N1*, Document Revision 5. Auflage, November 2005.

[HBGH+06] J. N. Herder, H. Bos, B. Gras, P. Homburg und A. S. Tanenbaum. MINIX 3: a highly reliable, self-repairing operating system. *SIGOPS Operating Systems Review*, 40(3), Juli 2006, S. 80–89.

[HeBr05] B. Herrenschmidt und B. Bruce. Booting the Linux/ppc kernel without Open Firmware. Linux Kernel Dokumentation: `http://lxr.linux.no/source/Documentation/powerpc/booting-without-of.tx% t?v=2.6.18`, November 2005.

[HMJS+97] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg und J. Wolter. The Performance of μ-Kernel-Based Systems. In *SOSP '97: Proceedings of the 16th ACM Symposium on Operating Systems Principles*, New York, USA, Oktober 1997. ACM Press, S. 66–77.

[Hohm96] M. Hohmuth. Linux-Emulation auf einem Mikrokern. Diplomarbeit, Technische Universität Dresden, August 1996.

[IBM00] IBM. *PowerPC^{TM} Microprocessor Family: The Programming Environments for 32-Bit Microprocessors*. International Business Machines (IBM) Corporation. Februar 2000.

[IBM05] IBM. *IBM PowerPC®750CX/CXe/CXr RISC Microprocessor. User's Manual. Version 1.3.* International Business Machines (IBM) Corporation. März 2005.

[Kais04] R. Kaiser. PikeOS - eine Mikrokern-basierte Umgebung für partitionierte Systeme. In *Linux Automation Konferenz*, 2004.

[Kuri03] J. Kuri. Embedded-Spezialist Wind River tritt Open-Source-Labs bei. heise newsticker: `http://www.heise.de/newsticker/meldung/42522`, Dezember 2003.

[L4-H] L4-Headquarters. Home of the L4 community. Internet: `http://www.l4hq.org/kernels/`. Besucht am 10.05.2007.

[L4Ka04] System Architecture Group L4Ka. *L4 eXperimental Kernel Reference Manual, Version X.2*, Document Revision 5. Auflage, Juni 2004.

[L4.v] L4.verified. Internet: `http://www.ertos.nicta.com.au/research/l4.verified/`. Besucht im April 2007.

[Labr98] J. J. Labrosse. *Microc/OS-II.* R & D Books. 1998.

[Lang06] M. Lange. Schritte zur Portierung des Fiasco Mikrokerns auf PowerPC. Diplomarbeit, Technische Universität Dresden, November 2006.

[Lied95] J. Liedtke. On μ-kernel construction. In *SOSP '95: Proceedings of the 15th ACM symposium on Operating System Principles*, New York, USA, 1995. ACM Press, S. 237–250.

[LUCC+05] J. LeVasseur, V. Uhlig, M. Chapman, P. Chubb, B. Leslie und G. Heiser. Pre-Virtualization: Slashing the Cost of Virtualization. Technical Report 2005-30, Fakultät für Informatik, Universität Karlsruhe (TH), November 2005.

[MeTa03] P. Melanson und S. Tafazoli. A Selection Methodology for the RTOS Market. In R. A. Harris (Hrsg.), *Proceedings of DASIA 2003 (ESA SP-532)*. Published on CDROM, Juni 2003.

[Morg91] M. Morgan. Integrated Modular Avionics for Next-Generation Commercial Airplanes. *IEEE Aerospace and Electronic Systems*, 6(8), August 1991, S. 9–12.

[OSEK05] OSEK/VDX. OSEK Operating System. Internet: `http://portal.osek-vdx.org`, Februar 2005. Version 2.2.3.

[RTEU93] RTCA und EUROCAE. RTCA/DO-178B: Software Considerations in Airborne Systems and Equipment Certification, Januar 1993. Also known as EUROCAE/ED-12B.

[Ruoc06] S. Ruocco. Real-Time Programming and L4 Microkernels. In *Proceedings of the 2006 Workshop on Operating System Platforms for Embedded Real-Time applications*, Juli 2006.

[Rush99] J. Rushby. Partitioning for Safety and Security: Requirements, Mechanisms, and Assurance. NASA Contractor Report CR-1999-209347, NASA Langley Research Center, März 1999. Also to be issued by the FAA.

[Schi03] R. Schiele. Building and Using a Cross Development Tool Chain. In A. J. Hutton, S. Donovan und C. C. Ross (Hrsg.), *Proceedings of the GCC Developers Summit 2003*, 2003, S. 213–222.

[Stan88] J. A. Stankovic. Misconceptions about Real-Time Computing: A Serious Problem for Next-Generation Systems. *IEEE Computer*, 21(10), Oktober 1988, S. 10–19.

[Tane02] A. S. Tanenbaum. *Moderne Betriebssysteme*. Pearson Studium, München, Deutschland. Übersetzt von Uwe Baumgarten, 2., überarbeitete Auflage. Auflage, 2002.

[TiPe05] M. Timmerman und L. Pernell. RTOS State Of The Art – Understanding RTOS Technology And Markets? RTOS Evaluation Project DSE-RTOS-EVA-001b, Dedicated Systems Experts, September 2005.

[VFia] Verified Fiasco. Internet: `http://os.inf.tu-dresden.de/vfiasco/`. Besucht im April 2007.

[WhSG02] A. Whitaker, M. Shaw und S. D. Gribble. Denali: Lightweight Virtual Machines for Distributed and Networked Applications. Technical Report 02-02-01, University of Washington, März 2002.

[WLAG93] R. Wahbe, S. Lucco, T. E. Anderson und S. L. Graham. Efficient software-based fault isolation. In *SOSP '93: Proceedings of the 14th ACM Symposium on Operating Systems Principles*, New York, USA, 1993. ACM Press, S. 203–216.

[Yagh01] K. Yaghmour. Adaptive Domain Environment for Operating Systems. Internet: `http://opersys.com/ftp/pub/Adeos/adeos.pdf`, Opersys, Inc., 2001.

Index

ABI, *siehe* Application Binary Interface
Addresstranslation, 34
API, *siehe* Application Programming Interface
Application Binary Interface, 11
Application Programming Interface, 11
ARINC, 4
Avionik, 3

Betriebssystem, 10–12
Bootloader, 42

Echtzeit, 8–10, 14, 52
  harte, 9
  weiche, 9
Embedded Systems Framework, 49–51
ESF, *siehe* Embedded Systems Framework

GCC, 39–41

Hellas-Board, 31

Integrated Modular Avionic, 3
  Anforderungen, 14
  Architektur, 3
Interrupts, 43–45

Kernel, 10
  Mikro-, 11
  Monolithischer, 11

L4, 25–28
libc, 49
libcycles, 49
libiguana, 49
libl4, 49
libmutex, 50
Linux, 19–21, 37–39

Mechanismus, 12
Minix, 21–23
Modularität, 16

Open Firmware, 45–46

Partitionierung, 5–8, 15, 54–56
  räumlich, 6
  zeitlich, 7
Pistachio-embedded
  Architektur, 46–47
  Plattform, 42–46
  System-Calls, 46–47
Policy, 12
PowerPC, 31–37
  750Cxe, 34
  Architektur, 31, 36
  BAT, 35
  Dekrementer, 36
  Paging, 35
  Registerset, 32

Scheduling, 7
SCons, 41–42
Standards, 15
System-Calls, 11

Timer, 50
Trusted Computing Base, 15, 56–58

Virtualisierung, 12–13
  Para-Virtualisierung, 13
  Pre-Virtualisierung, 13

Xen, 23–24, 60

Zeit, 10
  bewusste Systeme, 10
  unbewusste Systeme, 10
Zertifizierung, 4–5

Erklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Leipzig, den 24. Mai 2007