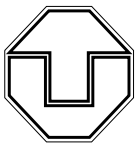
Technische Universität Dresden



Fakultät Informatik
Institut für Systemarchitektur
Professur Betriebssysteme

Diplomarbeit

Software Structure and Portability of the Fiasco Microkernel

Alexander Warg

28. Juli 2003

Betreuender Hochschullehrer: Prof. Dr. Hermann Härtig
Betreuender Mitarbeiter: Dr. Michael Hohmuth

All trademarks are the property of their respective owners.

Acknowledgements

I like to thank everybody, who supported my work on the FIASCO microkernel. Especially, I like to thank: my supervisor Michael Hohmuth, for introducing me into the depths of FIASCO's source code; Udo Steinberg and Frank Mehnert, who did numerous implementation details on IA-32 and accepted my design principles; Prof. Dr. Hermann Härtig, who always found time for a discussion; Adam Lackorzynski, who build the ARM cross compilers and never disabled my login; as well as all other members of the OS Groups at TU Dresden and University of Karlsruhe.

Erklärung

Hiermit erkläre ich, die vorliegende Arbeit selbständig verfaßt und keine anderen als die angegebenen Literaturhilfsmittel verwendet zu haben.

Declaration

I hereby declare that this thesis is a work of my own, and that only cited sources have been used.

Dresden, den 28. Juli 2003

Alexander Warg

Contents

Contents I

List of Figures IV

1 Introduction 1

1.1 Problem Statement 2

1.2 Approach 3

1.3 Document Structure 3

1.3.1 UML Class Diagrams 4

2 Fundamentals 5

2.1 The ARM Processor Architecture 5

2.1.1 Privileged Modes and Banked Registers 6

2.1.2 Exceptions 6

2.1.3 Memory Management Unit 7

2.1.4 StrongARM and XScale 7

2.2 Related Work 9

2.2.1 How to Achieve Portability 9

2.2.2 ARM Related Work 10

2.3 State of Fiasco 12

3 Design 14

3.1 Requirements 14

3.2 Overall Design of Fiasco 15

3.3 L4-Independent Hardware Abstractions 16

3.3.1 Native Data Types 16

3.3.2 Drivers 17
3.3.3 Generic Page-Table Interface 19
3.3.4 Standard C Library 19
3.4 L4-Specific Components 22
3.4.1 Basic L4 Abstractions 22
3.4.2 Hardware Layer of L4's Basic Abstractions 24
3.4.3 Exception Handling 26
3.4.4 L4-ABI Abstractions 26
3.4.5 Kernel Memory Management 29
3.5 New JDB Design 31
3.6 StrongARM Specific Design 33
3.6.1 Exception Handling 33
3.6.2 ARM Kernel Address Space 34

4 Implementation **35**
4.1 Polymorphism 35
4.2 Bootstrap 37
4.2.1 Stage 1, Boot Subsystem 37
4.2.2 Stage 2, In-Kernel Bootstrap 39
4.3 The Build System 40
4.4 StrongARM Implementation Details 43

5 Future Work **45**
5.1 General Issues 45
5.2 StrongARM Topics 46

6 Summary **48**

A Architecture-Specific Hooks **A-1**
A.1 Kernel-External Hooks A-1
A.1.1 Proc Class A-1
A.1.2 Atomic Operations A-2
A.2 Kernel-Internal Hooks A-3
A.2.1 Page_table Class A-3
A.2.2 Kmem Class A-5
A.2.3 Context Class A-5

A.2.4 Thread Class A-6
A.2.5 Kernel_thread Class A-6
A.2.6 In-Kernel System-Call Bindings A-7
A.2.7 Cpu Class A-12
A.2.8 Fpu Class A-12
A.2.9 Timer Class A-12
A.2.10 Pic Class A-13
A.2.11 Mapdb Class A-13
A.2.12 Startup Constructor A-14
A.2.13 Boot_console Class A-14
A.2.14 kdb_ke Module A-14

Acronyms **B-1**

Bibliography **C-1**

List of Figures

2.1 Address flow on ARM with FCSE 8

3.1 Layering of Subsystems 16
3.2 Design of the Generic CPU Abstractions 18
3.3 Design of the Console-I/O Subsystem 20
3.4 Generic Page-Table Interface 21
3.5 Hierarchy of the Thread Abstraction 23
3.6 Hierarchy of the Space Abstraction 24
3.7 Platform Hooks of the Thread Abstraction 25
3.8 Design of the L4 ABI Types 27
3.9 Encapsulated System-Call Parameters 28
3.10 Kernel Binding for the Id-Nearest System Call 29
3.11 Model of the Memory-Management System 30
3.12 Class Structure of the New JDB 32
3.13 Exemplary Subclasses of Jdb_module 32
3.14 StrongARM Address-Space Layout 34

4.1 Memory Aliasing Example 43

List of Programs

4.1 Explicit Compile-Time Polymorphism (Interface) 37
4.2 Explicit Compile-Time Polymorphism (Implementation) 38
4.3 Simple Static Constructors Example 40
4.4 Prioritized Static Constructors Example 41
4.5 Example Modules File 42

Chapter 1

Introduction

Portability and software engineering are two key terms that have become very significant for today's software. For example, the complete philosophy of Java is based on the idea of write once and run anywhere, which is the most extreme form of portability. The software engineering community proposes the principle of object-oriented design (OOD), which manifests itself in programming languages that directly support object-oriented programming (OOP). The approach of OOD is claimed to be useful for distributed work on software projects and to result in maintainable code. The OOD approach helps to define autonomous components with well-defined interfaces, which are the base for good testability.

Nowadays developers of application software commonly accept the principles of software design and the importance of portability. Many modern application programs run on various computer platforms and operating-systems; these are often implemented in object-oriented programming languages or with object-oriented techniques. The operating-systems world seems to be somewhat slower in accepting new principles; the importance of portability is known commonly, whereas a common basic approach for the design of operating systems is not established.

The microkernel approach, which was devised in the early 1980's, has the potential to become the base for operating-system design. Microkernels support modern component-based and modularized design of the operating-system personalities built atop them. However, the two key issues, portability and software engineering, can also be applied to the microkernel itself.

The first microkernels suffered from their poor performance. For instance, Mach (see [Accetta et al., 1986]) was not accepted because of its bad performance. These first-generation microkernels where not really small and had a lot of functionality built into the kernel.

The microkernels of the second generation, such as L4, aimed to be highly efficient. Jochen Liedtke claimed in 1995 that efficient microkernels are per se non-portable (see [Liedtke, 1995]) and must be implemented in assembly language to be able to use specific features of the underlying hardware.

Today the number of architectures that are on the market is growing extremely fast and the field of embedded systems is becoming increasingly important. With the growing number of potential target platforms, the maintenance effort increases too fast if the implementation is done from scratch for every platform; each new implementation is very prone to errors and the various implementations do not benefit from bug fixes in one particular implementation. Thus, the claim that portability is not an issue for microkernels no longer holds true.

Hazelnut from Liedtke's operating-systems group at the University of Karlsruhe proved that it is possible to implement a microkernel in a high-level language (HLL). The runtime overhead of Hazelnut, which is implemented in C, was very low (see [Hazelnut, 2000]).

Today's picture of microkernel implementations has already changed. Current microkernels, such as Fiasco, Hazelnut, and Pistachio, are implemented in HLLs. This development made the microkernels far more maintainable and less prone to errors.

Hazelnut and Pistachio, from University of Karlsruhe, have already focused on portability and use a common code base for different target architectures, whereas Fiasco, from TU Dresden, currently supports only one target architecture in the main branch.

1.1 Problem Statement

The known portable L4 implementations do not use object-oriented principles to achieve portability, because polymorphism is thought to be a performance problem. The kernel implementations either suffer from an excess of conditionally compiled code (`#ifdef` constructs) or from inherent code duplication.

Another deficiency of Pistachio and Hazelnut is the restricted real-time capability. These kernels use a global interrupt lock for synchronization, which results in long and partly unbounded interrupt latencies. In contrast, the Fiasco microkernel was designed to provide good real-time capabilities. The focus was on high preemptability, as well as short and bounded interrupt latencies, but **not** on portability.

One way to overcome the deficits mentioned before is to improve the preemptability of Pistachio. The other solution is to make Fiasco more portable. The latter possibility has the advantage that the Fiasco microkernel was developed in Dresden, resulting in a high level of Fiasco knowledge in the Operating Systems Group in Dresden.

Additionally, the ongoing VFiasco[1] project could benefit from clear hardware abstractions.

[1] VFiasco is the approach to formally proof the correctness of Fiasco. The formal proof is an important step to use Fiasco in systems with very high security demands.

1.2 Approach

This paper presents a completely object-oriented design that is intended to improve the portability of the FIASCO source code. The design incorporates all features of object orientation, such as inheritance and polymorphism. I assumed that there is no a-priory impact on the performance of the compiled kernel due to the usage of OOD and OOP to achieve a better portability, which is not true offhand.

In reality, one of the main challenges in applying modern software design principles, such as OOD, to microkernels is to keep the performance impact as low as possible (at best zero). However, the things that make OOP attractive, such as polymorphism, have a problematic influence on the efficiency of the resulting software.

Most C++ compilers generate an extra level of indirection and impede source-level inlining in the case of polymorphism. However, the extra level of indirection results in extra instructions and extra memory accesses on method invocation, and source-level inlining is the base for efficient realization of fine-grained interfaces.

The additional overhead that is introduced through OOP is commonly accepted in application development, because the advantage of better maintainability pays more than the *minor* performance impact. However, in the microkernel community the most important design goal is to achieve maximum performance and to have as little influence on running applications as possible. Microkernels of the first generation, such as MACH, where not accepted because of their unsatisfactory performance.

1.3 Document Structure

The rest of this paper is divided into four chapters. The following chapter provides a foundation that should help you better understand this thesis. You may just skip Chapter 2 completely or pick up some parts if you feel familiar with the mentioned topics.

The Design chapter (Chapter 3) contains the object-oriented design of FIASCO and describes how portability is actually improved. Chapter 4 focuses on issues such as the efficient implementation of the polymorphic design, the build system, and the boot sequence. The Design and the Implementation chapter are split into two main parts, one that deals with portability in general and another one that reflects the issues dedicated to the ARM port, which was part of the task for my thesis.

Finally, an outlook to open topics is given in Future Work (Chapter 5), and the Summary (Chapter 6) briefly recapitulates the important issues of this paper.

1.3.1 UML Class Diagrams

All the class diagrams use a common style to communicate extra information. The pictured classes are colored white in the case of generic classes (i.e., not dedicated to a certain architecture). The gray shaded classes are hardware specific implementations or interfaces.

Further information is encoded in the font that is used for the class title and for the operations. The class title is set italic in the case of abstract classes that cannot be instantiated. The operations of a class are set bold and italic if they are abstract or in C++ terms pure virtual. Final implementations of operations are set in normal font. Class-scope operations are underlined.

Chapter 2

Fundamentals

2.1 The ARM Processor Architecture

A main point of my work was porting of the FIASCO microkernel to ARM and especially to StrongARM.

The ARM architecture is nowadays one of the mostly used architectures in embedded systems. The big market share is due to the simple design that leads to low-cost and low-power implementations.

This section introduces the ARM architecture from the operating systems point of view; the user-level view and the programming model are not covered in detail.

The ARM is fundamentally a 32-Bit RISC architecture, augmented with CISC strengths. The condensed properties of ARM are as follows:

- 32-Bit load/store architecture
- High performance at low cost and power consumption
- Conditional execution of all instructions
- Parallel shift and ALU operation
- Multiple register load/store instructions (slow on Intel XScale)
- Hardware-loaded TLB
- Multiple page size support
- Domains, which can be taken as a variation of address-space identifiers (ASIDs)

The following sections shall just give short overview of the ARM architecture. For complete documentation see [ARM Ltd., 2000, StrongARM, 2001, Intel XScale, 2002, Intel PXA, 2002a, Intel PXA, 2002b].

Mode	Privileged	Registers
User	–	R0–R15
FIQ	×	R0–R7, R8_fiq–R14_fiq, R15
IRQ	×	R0–R12, R13_irq–R14_irq, R15
Supervisor	×	R0–R12, R13_svc–R14_svc, R15
Abort	×	R0–R12, R13_abt–R14_abt, R15
Undefined	×	R0–R12, R13_und–R14_und, R15
System	×	R0–R15

Table 2.1 *ARM Operating Modes.*

2.1.1 Privileged Modes and Banked Registers

The ARM architecture supports seven operating modes, which are summarized in Table 2.1. One of these modes is the non-privileged user mode; the other six are privileged modes for operating-system execution. Modes can either be switched under the control flow of privileged software or via exceptions, which include the explicit software interrupt (SWI).

The ARM uses banked registers to preserve the minimal machine state on mode switches. Banked registers are general-purpose registers that have an extra instance for a dedicated mode of operation. The privileged operating modes that can be entered via exceptions have at least a banked stack pointer (SP), link register (LR), and saved program status register (SPSR). The fast interrupt (FIQ) mode, has extra five banked registers, to facilitate extremely fast interrupt handling without the need to save registers to memory.

2.1.2 Exceptions

Exceptions result in a defined change of the flow of control. It is irrelevant whether the exception is due to reasons internal or external to the processor; the CPU assigns a fixed address in the exception vector to the instruction pointer and switches to a privileged mode (see Section 2.1.1), corresponding to the exception that occurred.

The ARM architecture supports the listed exceptions.

Reset The reset exception is raised when the processor's reset input is asserted, and the execution starts at address `0x00000000` in *Supervisor* mode.

Undefined instruction This exception is raised if an attempt is made to execute an *undefined* instruction, or if a coprocessor instruction shall be executed and no coprocessor responds. The processor switches to the *Undefined* mode.

Software Interrupt (SWI) An SWI enters the *Supervisor* mode. It is raised explicitly by executing the `swi` instruction.

Prefetch Abort If the memory system signals an abort on an instruction prefetch, the *prefetch abort* exception is raised. The execution continues in *Abort* mode.

Data Abort The memory system signals an abort on a data access. Again, the processor is put into *Abort* mode.

Interrupt Request (IRQ) The *IRQ* exception is raised when the IRQ signal of the CPU is asserted. IRQs can be masked by setting the appropriate bit in the current program status register (CPSR). This exception ends up in the *IRQ* mode.

Fast Interrupt Request (FIQ) The *FIQ* exception is raised when the FIQ signal of the CPU core is asserted. An *FIQ* exception results in a switch to the *FIQ* mode, which provides extra five banked registers. The FIQ signal should be used for a small number of interrupts that require very fast handling.

2.1.3 Memory Management Unit

The ARM architecture defines virtual memory support with a hardware-walked page table. An implementation may provide a translation look-aside buffer (TLB), which can either be unified for code and data or split into an instruction-prefetch translation look-aside buffer (ITLB) and a data-access translation look-aside buffer (DTLB). In summary, the virtual memory system of ARM is based on hardware loaded TLBs, provides multiple page sizes and a variation of address-space identifiers (ASIDs), the domains.

Page tables have two levels: the first level is referred to as page directory (PD), and the second level as leaf page table (LPT). The page directory contains 4096 entries and a leaf page table consists of 256 or 1024 entries.

2.1.4 StrongARM and XScale

The StrongARM SA-1100 is a low-power, low-cost, and high-speed implementation of the ARM architecture. It specifies not only the CPU core, but also a set of peripheral controllers that are integrated within a single package.

The StrongARM CPU core implements a modified Harvard architecture with separate caches and TLBs for instruction and data streams.

The XScale application-processor family is the successor of StrongARM. The processors provide a set of extensions to the ARM architecture, as well as increased clock rates. Interesting enhancements, from the operating systems point of view, are better cache and TLB control (e.g., cache and TLB pinning).

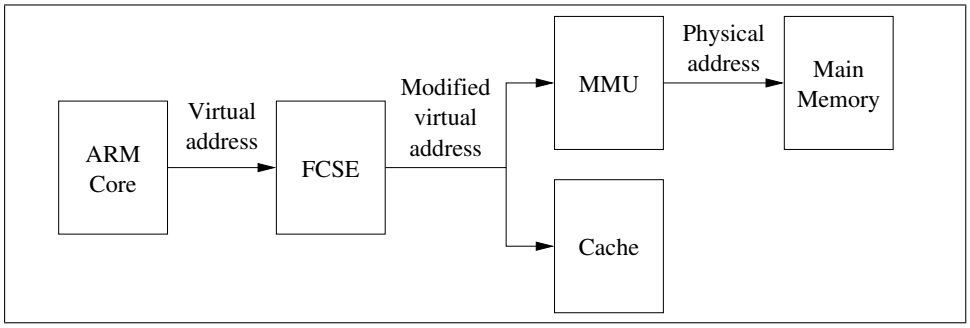


Figure 2.1: *Address flow on ARM with FCSE.* (According to [ARM Ltd., 2000], Figure 6–1 on page B6–3)

Cache Architecture

The modified Harvard architecture makes it possible to achieve clock speeds of 200 MHz and more. The processor provides, beside its standard instruction and data caches, a write buffer, a mini-data cache, and a data-prefetching read buffer.

The caches are **fully virtual**, which results in problems that are described in later on, in Section 2.2.2. Fully virtual means indexed and tagged with virtual addresses.

The XScale processors provide almost the same cache features like SA-1100; only the data-prefetching read buffer is missing. As compensation for the missing read buffer, XScale features larger caches and the ability to pin cache lines.

Fast Context Switch Extension (FCSE)

The fast-context-switch extension (FCSE) of the StrongARM provides a means to re-map the lowest 32 MByte of an address space. The FCSE is controlled with a PID register that contains the ID of the currently running process. The contents of the PID register are used to calculate the *modified virtual address* for accesses to the first 32 MByte of the address space; this *modified virtual address* is then fed to the fully virtual caches and the MMU for the usual translation process. Figure 2.1 shows the address flow in the ARM MMU system with FCSE.

The calculation is as follows, where P is the current PID, A_{access} is the accessed address, and $A_{virtual}$ is the *modified virtual address* fed to the memory system.

$$A_{virtual} = \begin{cases} P \cdot 32 \text{ MByte} + A_{access} & \text{if } A_{access} < 32 \text{ MByte} \\ A_{access} & \text{if } A_{access} \geq 32 \text{ MByte} \end{cases}$$

As evident from the equation, only the lowest 32 MByte of the address space are affected by the FCSE; the rest of the address space is the normal virtual address space.

The FCSE shall be very useful for small address spaces on StrongARM, however, there is a discussion about that later on.

2.2 Related Work

There is a lot of work to do in the field of portable operating systems, especially in the open-source community. The most common open-source operating system is probably Linux, which runs in version 2.4 on at least 17 different platforms (see [Linux, 2003]). Other classical operating systems that have a focus on portability are FreeBSD (6 platforms, [FreeBSD, 2003, McKusick et al., 1996]) and NetBSD (53 platforms, [NetBSD, 2003, Kesteloot, 1995]).

The aforementioned systems have the common feature that all are based on a monolithic kernel, where most of the drivers as well as file systems run in privileged mode. Nevertheless, in the microkernel community there is also upcoming interest in the direction of portability. A recent microkernel that has a clear focus on portability is Pistachio, the implementation of the L4 version X.2 interface from University of Karlsruhe (see [L4Ka, 2003]). Other projects in the field of portable microkernels are the Sartoris microkernel (see [Sartoris Developers Group, 2003]) and "The KeyKOS" microkernel (see [Bomberger et al., 1992]).

2.2.1 How to Achieve Portability

There exist several different approaches to achieve the design goals of easy portability and maintainability. In the case that only two different platforms have to be supported, the obvious way is to split the source code into two parts, an architecture-specific and a generic part. This very simple approach proves not to be useful if more than two platforms should be supported, because there may be code that can be shared among some platforms but is useless for others; for example, the VGA driver, which fits for IA-32 and IA-64 but not for ARM. Such drivers must, with the two-part scheme, go into the architecture-specific part, and therefore must be duplicated.

The monolithic kernels that I mentioned at the beginning of Section 2.2 are structured in a more sophisticated manner. Their source code consists of: a completely generic part; an architecture-dependent part, which is often subdivided into sub-architectures; and the device drivers.

The architecture dependent part contains code that is specific to the target processor as well as device drivers for devices that are dedicated to machines based on the target processor. The device-driver part contains drivers for hardware devices that are available on at least two architectures. Drivers are mostly classified according to their purpose (e.g., Network, Video, and Sound).

The Pistachio microkernel has a quite similar structure, it is subdivided into five parts: an architecture-specific part (mostly CPU specific), a platform-specific part (e.g., PC99 or EFI), an API-specific part, a generic part, and a glue part. There is no part for device drivers, which is seemingly unproblematic because a microkernel does not contain any device drivers. However, at least for debugging purposes even a microkernel needs drivers for I/O hardware. Moreover, there are further devices that need a driver in the micro kernel: first, the interrupt controller, for enabling

and disabling interrupts; second, the system timer, which is needed to trigger events such as time-slice ends or IPC timeouts.

Another problem of this hard partitioning is that two different API implementations, such as version 2 and version X.0, cannot share any code, in spite of the fact that they vary only at very few locations.

The structure of Pistachio obviously goes into the direction of aspect-oriented design (ASOD) (see [ASOD, 2003]). The *crosscutting* concerns, target architecture, target platform, and system-call API, are separated cleanly. A design goal was a modular microkernel construction kit, which features easy construction of new APIs and easy porting to new architectures and platforms.

However, Pistachio is implemented in plain C++ and without the use of inheritance and polymorphism. C-preprocessor macros and conditional compilation (`#ifdef`s) are used, to compose the different components and to enable specific features that are scattered throughout several functions.

2.2.2 ARM Related Work

Address Space Switches

The frequent address-space switches usually performed in microkernel operating systems are one of the trickiest challenges with ARM. The most problematic feature is the fully virtual caches (see Section 2.1.4) that do not support any form of ASID tagging, neither on StrongARM nor on XScale. Additionally, the StrongARM architecture suffers from its rudimentary cache-control mechanisms that allow only complete invalidation of the instruction cache.

The fully virtual caches have the effect that cache coherency must be ensured by software. With a naive approach, the caches must be flushed[1] on each address-space switch. The direct and even more the indirect costs of this operation result in a major performance impact. The direct cost for flushing the cache is 1,000–18,000 cycles. The indirect costs, which result from the lost cache and TLB working set, are about 45 cycles per TLB miss and about 70 cycles per cache miss. In the worst case, the costs are up to 75,000 cycles ($\approx 350\mu s$ on a 200-MHz processor).

General Solution The ARM architecture provides means to mitigate the problem of missing ASIDs in the TLBs and even in the caches. The solution is to use the ARM domains. A sophisticated description of the ARM domains can be found in [ARM Ltd., 2000, Wiggins, 1999, Wiggins and Heiser, 2000].

In principle, the ARM domains provide efficient access-control changes for large and non-contiguous regions of virtual memory. This mechanism supports fast switches among different address spaces, with the restriction that concurrently active address spaces must have no overlap in their mapped memory regions. As long as the address spaces meet this restriction, switches among them can be

[1] written back and invalidated

made by reloading the domain access control register (DACR) with the appropriate access rights.

The real profit results from the fact that the StrongARM architecture observes domain access rights even for cached data and not only for cache misses; thus the kernel can mostly avoid the extremely expensive cache flushes on address-space switches.

The conclusion is that it is possible to circumvent the high expenses for address-space switching, as long as the kernel ensures the condition of no overlap among the mappings of active address spaces.

The general implementation idea is based on a caching page table, which contains non-overlapping regions of different address spaces at the same time. The page-table entries of the different address spaces are tagged with different ARM domains, and isolation of the address spaces is enforced with an appropriate mask in the DACR. More information about the principles can be found in Section 4.3 of [Wiggins, 1999]. The aforementioned concepts also hold true for the XScale architecture.

Non-overlapping Address Spaces One reason for virtual address spaces is to support transparent multiprocessing with programs that make use of the same address ranges. Therefore, the nature of processes running in different address spaces is that they may have overlaps in their mappings.

This condition stands in hard contradiction to the assumption of no overlap in mappings, made for fast address-space switches. One approach to reduce or even remove the overlap is the single-address-space operating system (SASOS)[2] approach. SASOSs are a rather special class of operating systems, but the microkernel approach should not be restricted to such a niche.

A more general way to reduce contention for address-space ranges is to use the FCSE of StrongARM, which allows a transparent re-mapping of the lowest 32 MBytes of the virtual address space (see Section 2.1.4). The drawback of the FCSE is the restriction to the lowest 32 MByte of the 4-GByte virtual address space. This means that only a variation of small address spaces benefits from the FCSE. However, with the target systems of ARM processors in mind, which are embedded systems, the limitation of processes to a 32 MByte address space seems to be not unrealistic.

In addition, tasks with larger address spaces may benefit from the fast address-space switch mechanism as long as the static part of the address space is limited to the lowest 32 MByte and the memory management component of the operating system avoids overlaps in dynamically allocated memory, such as stacks and heap.

Limited Number of Domains A problematic issue is the limited number of available ARM domains. Only 16 domains are provided by the ARM architecture.

[2] MUNGI from University of New South Wales (UNSW) is a SASOS based on L4 technology, see [Wilkinson et al., 1995]

The fact that every running task needs to get a domain assigned makes it obvious that potentially more than 16 domains would be necessary to execute more than 16 tasks.

The solution for this problem is to assign domains to running tasks dynamically. The dynamic allocation of a limited resource always yields the following key issues:

Preemption to withdraw an assigned resource

Scheduling for choosing a victim to be preempted and to select a candidate that gets a free resource

Thrashing, which occurs if there are more candidates in the current working set than there are available resources

In terms of ARM domains, the keywords are **domain preemption, domain thrashing, and domain preemption strategy.** [Wiggins et al., 2002, Wiggins and Heiser, 2000, Wiggins, 1999] discuss these terms in more detail.

2.3 State of Fiasco

Michael Hohmuth, from the TU Dresden, initially wrote the Fiasco microkernel. It was the first implementation of the L4 microkernel interface in a HLL (C++).

At the very beginning of my work, the Fiasco microkernel was already modularized and well-defined interfaces were used among the modules. The main reason for this structure was to achieve testability of separate modules. For such off-kernel testing of modules, the dependencies among the modules must result in a directed *acyclic* graph. If this property is not satisfied, all modules participating in a circular dependency can only be tested as a whole. The same applies to portability too; more details on this shall follow in Section 3.1.

The source code of the Fiasco microkernel was already partitioned into subsystems, modules, and submodules. The different **subsystems** are almost complete self-contained parts of the microkernel, such as a library for simple memory management, the reduced C library, or the main kernel image itself. Each subsystem consists of one or more **modules** that encapsulate logical units. For example, the `thread` module contains the data structures and methods necessary to handle L4 threads. The interfaces of these modules are well defined and hide the implementation details. Again, a module can aggregate one or more **submodules**, which are used to further subdivide a module into smaller logical blocks (e.g., `thread-syscall` and `thread-ipc` are two submodules of `thread`).

When I initially started to work with Fiasco's source code, the property of an acyclic directed graph was violated, the interfaces were not designed for portability among different architectures, and architecture and platform specific code was spread all over the kernel.

Indeed the situation proved to be better than that. The circular dependencies were almost entirely introduced by logging features that use hooks into the FIASCO kernel debugger (JDB), and on the other hand JDB depends on nearly everything of the microkernel. Interfaces were well defined, even if mostly not machine independent and often coarse grained.

The subsystem-module-submodule structure was not designed for portability; the initial partitioning was based only on logical blocks, and later refactoring aimed at the elimination of certain circular dependencies for meliorating testability.

All in all, portability was a new goal that needed further partitioning, restructuring, and new probably more fine-grained interfaces.

Chapter 3

Design

The main goal is to achieve portability among different computer architectures and platforms, while reusing most of Fiasco's existing source code and keeping the performance impact as low as possible.

The performance aspect is very important in a microkernel operating system. Older microkernel operating systems often suffered from the bad performance of the underlying kernel. As the past has shown, the Mach microkernel (see [Accetta et al., 1986]) was not accepted due to the performance issue. The L4 microkernels were actually designed to be highly efficient. Thus L4Linux, a Linux running in user space, suffers from a performance loss of only five percent (see [Härtig et al., 1997]).

In condensed form, the design objectives are:

- **Portability** among various hardware platforms
- No performance impact at the L4 interface
- Low maintenance effort for the different supported platforms
- No `#ifdef` constructs to select specific implementations

3.1 Requirements

This section is about portability in general, but in the terms of Fiasco's existing structure.

To achieve a certain degree of portability, all machine-specific code must be factored out into separate modules or submodules. Appropriate interfaces to the architecture-dependent parts have to be defined. These two things can be brought together under the term *encapsulation of machine-specific code*, which is clearly a necessary issue. The more tricky part is to determine the appropriate granularity

for the encapsulation, which must be somewhere between taking the whole kernel as machine specific and hiding single assembler statements behind a general interface. The resulting granularity is a tradeoff between the porting effort and runtime efficiency (performance).

Not only the right granularity, but also the dependencies among the *grains*, in our terms modules and submodules, are of high importance. To get an easily portable system, there should be no circular dependencies (see Section 2.3). In the case where no circular dependencies exist, the porting work can be started at **low-level modules**, which do not depend on anything else, and go toward the **top-level modules**, which have no other modules that depend on them. The module structure and the avoidance of circular dependencies also gain the overall testability and the testability of a partially ported kernel.

Altogether, to create portable software, circular dependencies among different modules should be removed entirely and a suitable granularity for the encapsulation must be found.

3.2 Overall Design of Fiasco

When I started this work, I decided to use a two-part partitioning of every subsystem (generic & architecture-specific). This approach was determined to be infeasible; even for microkernel design, the weaknesses described in Section 2.2.1 are not acceptable.

The resulting design is now similar to Linux. The source code is formally fragmented into a generic and an architecture-specific part. Nevertheless, the actual structure is far more fine-grained and based on the subsystem-module-submodule scheme of Fiasco, which is introduced in Chapter 2.

Code that is not completely generic is fully factored out into separate modules or submodules and `#ifdef`s are almost completely banned. Modules and submodules that cannot be confined to one specific architecture, but to two or more architectures, are located in the generic parts of the source code.

It turned out that there are several orthogonal concerns. For example, there are modules that are dedicated to: a certain hardware device, which is available on more than one architecture; a specific API version; or the word width of the target processor. These semi-specific submodules are labeled with special suffixes that point out their concern.

Figure 3.1 on the following page shows the composition of the final boot image. The subsystems have the following tasks:

Kernel + JDB is the implementation of the L4 interface and the Fiasco kernel debugger (JDB).

Boot is the early bootstrap code, which loads the kernel into virtual memory and transfers control to it (see Section 4.2.1 for more information).

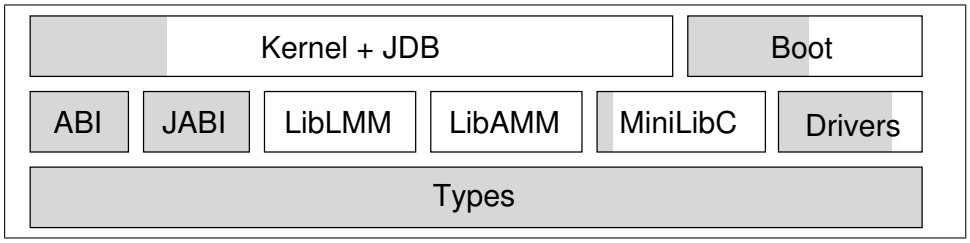


Figure 3.1: *Layering of Subsystems.* This picture shows the composition of the Fiasco kernel image out off different subsystems. The shaded areas give the estimated amount of architecture and/or platform dependent code in the pictured subsystems.

ABI contains L4-ABI specific type definitions (described in Section 3.4.4).

JABI contains user-level interface specifications for the JDB.

LibLMM is a generic implementation of a list-based memory manager.

LibAMM provides functions to manage regions of virtual address spaces.

MiniLibC contains the C-library used for the kernel and the Boot subsystem.

Drivers is a library of drivers for console I/O and general processor features.

For a better understanding of the rest of this chapter, there are three main terms that are used as follows:

kernel image is used whenever the complete bootable image of the kernel is meant, it consists at least of the subsystems you can see in Figure 3.1,

kernel always refers to the implementation of the L4 interface, which is composed of the Kernel, ABI, LibLMM, LibAMM, MiniLibC, and Drivers subsystems,

JDB refers to Fiasco's kernel debugger, which actually is an integrated part of the kernel, but virtually forms a stand-alone debugger and can be decoupled from Fiasco.

3.3 L4-Independent Hardware Abstractions

This section deals with hardware abstractions that are not dedicated to porting of L4 or Fiasco. The subsequently described parts are generally useful for portable software and especially portable operating systems.

3.3.1 Native Data Types

The most important prerequisite for the portability of machine-tight code is to have clearly defined **native data types**. These native data types belong to the

group of unstructured integral types and can be classified in **fixed-width** and **fixed-meaning** data types. The former have a fixed number of bits, independent from the target architecture. The latter have a special purpose and may vary in their width according to the target architecture.

The fixed-width types must be defined for each processor architecture and for each supported compiler, because the C++ standard specifies no data types with a concrete width. The definition according to the used compiler can be dropped, because FIASCO currently only supports the GNU compilers, GCC and G++. On the other hand, the architecture specific definitions cannot be omitted.

The second category, which may also differ from one architecture to another, is the fixed-meaning types. Members of this class are, for example, a type with the width of a general-purpose register or a virtual memory address. The C++ standard already defines some of these types, such as `void*` for addresses, but it lacks the definition of others.

Because of the importance of these native types, not only for the kernel implementation but for nearly every subsystem, they are defined in a dedicated subsystem; this subsystem is called the *Types* subsystem and forms the lowest level of architecture dependent code.

3.3.2 Drivers

The Drivers subsystem provides the next level of hardware abstraction. This subsystem contains several hardware-specific device drivers, which can be classified into different levels again.

Processor Driver

The lowest-level driver provides a very small interface to the central processing unit (CPU). Figure 3.2 on the following page shows the actual interface of the CPU, the `Proc` class, which is reduced to the common base of all CPUs FIASCO should run on. The `Proc` class contains the subsequently mentioned functionality:

- IRQ control methods, which control and request the acceptance of hardware interrupts (IRQs); the strangest method in this group is probably `irq_chance`, which gives a pending IRQ the chance to come through
- A spin-loop support method (`pause`), which must be used in tight spin loops, and should protect the CPU from consuming too much energy and prevent blocking of hyper-threaded[1] CPUs
- Sleep-mode support (`halt`), which puts the processor into a sleep mode until the next IRQ

[1] duplication of execution context but not the execution units on a single processor, to increase the utilization of the execution units (see [Intel, 2003])

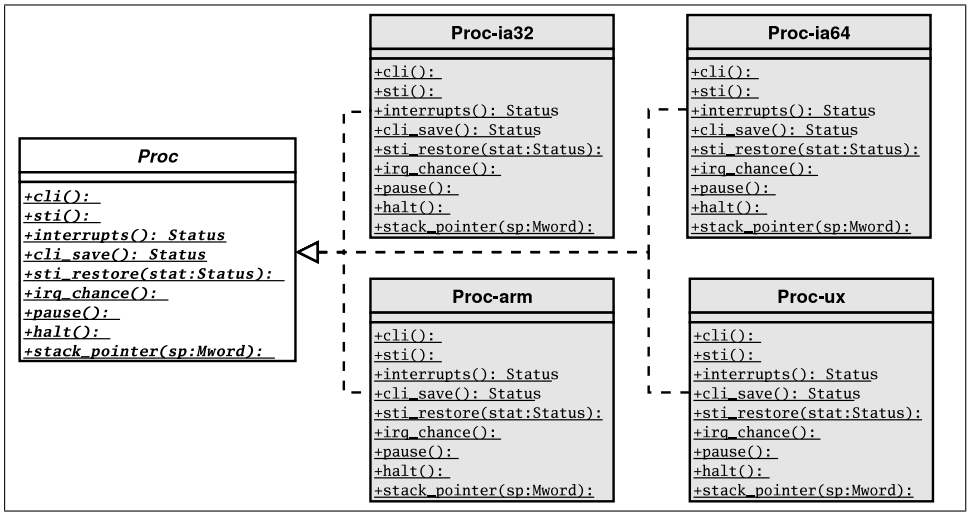


Figure 3.2: *Design of the Generic CPU Abstractions.* This figure shows the design of the very generic CPU interface, which is used not only by the microkernel but also by other subsystems.

- Methods to access/manipulate the stack pointer of the local CPU

For the complete interface definition see Section A.1.1 on page A-1.

Atomic Operations

The atomic operations are an integral part of Fiasco. The lock-free and wait-free (see [Hohmuth and Härtig, 2001]) implementation requires the use of operations that read and manipulate data in the main memory atomically, such as *compare and swap* or *test and set*. These atomic operations are defined in an extra module and come in two different flavors: multi-processor safe, and uniprocessor safe. The operations are hidden behind a type-safe interface, which is implemented with C++ function templates. The generic and type-safe wrapper functions make use of machine-specific low-level operations. Appendix A in Section A.1.2 contains the interface definition for these type-unsafe atomic operations that must be implemented for a specific target processor.

Port-I/O Driver

Another low-level driver is the port-I/O driver. This driver is only useful on architectures that use IA-32-compatible devices. Such devices use a special physical address space distinct from the normal memory address space, the I/O-ports. For instance, IA-64, as a supported architecture, may use an IA-32 compatible VGA card. The port-I/O abstractions are a means to write processor independent device drivers, no matter how the port address space is accessed on the underlying

platform.

Console-I/O Driver

Additionally to the aforementioned device drivers, a console-I/O abstraction exists. This abstraction provides a means for debugging input and output. Figure 3.3 on the following page illustrates the complete design of the console-I/O system.

No I/O facilities in the kernel are necessary if the microkernel is used in a production system, because all the device interaction is implemented in user-level device drivers. Nevertheless, the development of the microkernel and the applications atop it often needs debugging. To support a well-featured kernel debugger, the console-I/O system forms the abstraction layer between the various I/O hardware and the kernel debugger.

The basic console abstraction (class `Console`) is designed to cover any kind of character-based input and/or output device. Specialized abstractions for designated hardware classes are also defined. For instance, the serial UART device drivers for StrongARM and the 16550 UART, as used in most IA-32 computers, are bases on the `Uart` class definition.

3.3.3 Generic Page-Table Interface

One of the fundamental principles of L4 microkernels is memory protection via address spaces. The underlying hardware platform must provide a mechanism for enforcing isolation of user-level programs. The processor must have the ability to allow or disallow access to a certain area of the main memory. However, the protection mechanism is often combined with an address-translation mechanism (virtual memory).

Figure 3.4 on page 21 pictures a generic interface for the combination of address-translation and protection mechanisms. A complete documentation of the page-table interface is given in Section A.2.1 on page A-3.

The original IA-32 Fiasco used the `Space` class as interface for the hardware page-table structure. However, I designed the page-table interface completely independent from L4's address-space abstraction. The main reason for the decoupling is the problematic address-space switches on ARM, which is described in Section 2.2.2. The solution outlined in this section is based on a caching page table (CPD), which caches mappings from multiple address spaces.

3.3.4 Standard C Library

The standard C library is one of the most basic components that are necessary for creating complex software in C or C++. Although it is possible to implement software without using a standard C library, it is very convenient to use their well-known abstractions.

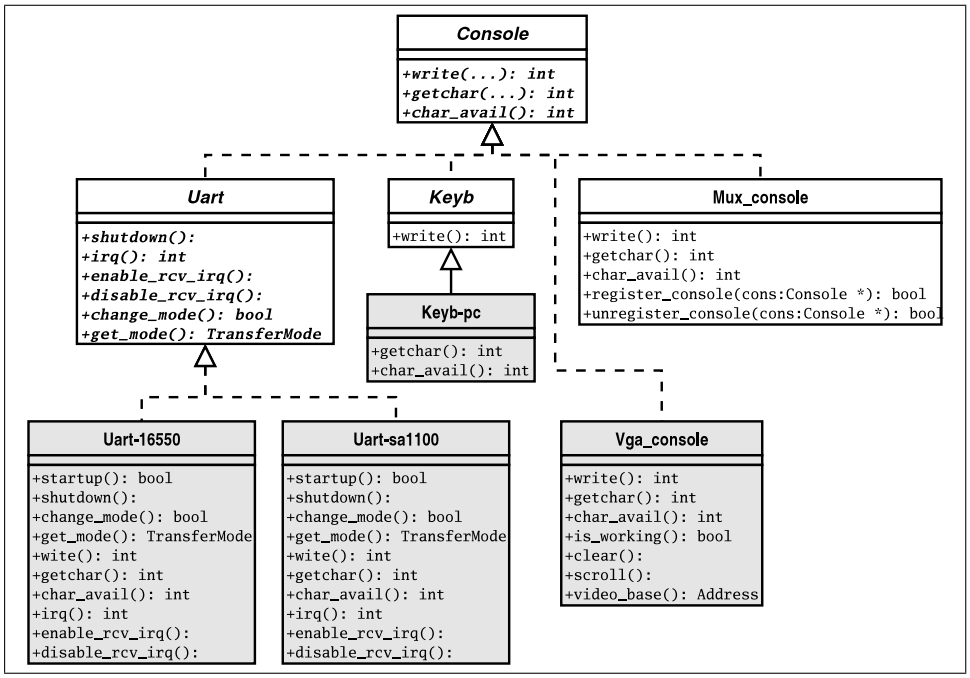


Figure 3.3: *Design of the Console-I/O Subsystem.* The pictured class hierarchy has the abstract interface for character-based I/O devices as root. This interface is on the one hand implemented directly, for instance by `Mux_console` or `Vga_console`. On the other hand, it is refined for specific classes of hardware, such as `Uart` for serial UART devices.

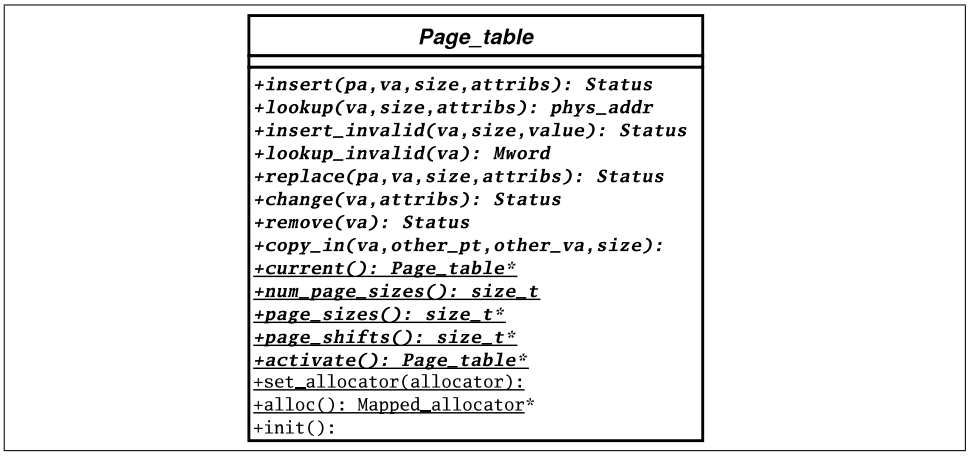


Figure 3.4: *Generic Page-Table Interface.*

There are several different C libraries freely available, but they are mostly based on an underlying operating system (OS) and this is problematic if the OS kernel itself is the destination. Moreover, only a small subset of the features current C libraries support is really useful for kernel and in particular microkernel implementation. The following features are identified to be necessary for Fiasco's implementation.

- String handling functions, such as `memcpy` and its relatives
- Character-type functions as usually provided by `ctype.h` (mostly used in JDB)
- Assertions, which are widely used to catch error conditions that are caused by disregarded interface constraints
- Static construction and destruction (important for static C++ objects)
- `setjmp` and `longjmp`, which are used for in-kernel page-fault recovery
- Basic input and output functionality, as for example `printf` and `getchar`

Other features, such as I/O via file handles or file-system access, are useless for Fiasco, because L4 microkernels do not know anything about abstractions like files.

At the beginning of my work, the main tree of Fiasco was built against the C library from the OSKit v0.6 (see [Group, 1999]), which is available only for IA-32. During the work for my term paper [Warg, 2002], I already implemented a minimal C library for the IA-64 port of Fiasco, which is based on the *diet libc* (see [von Leitner, 2003]). As the goal of this paper is generally improved portability, a single C library that shares as much code as possible among the supported architectures had to be developed.

The starting point for the new C library was the one used for IA-64. This library is extended to the IA-32 and the ARM architecture, which was a minor effort, because only type and limit definitions and the `longjmp` implementation are architecture-specific.

The minimal C library is almost completely self-contained. The only exceptions are the console-I/O functions, which need a kind of back-end driver to pass the output to or read the input from. The remaining part of the library does not rely on any external functionality. The design target for the standard-I/O component was to have a simple and well-defined back-end interface, which makes it possible to reuse the library for almost any kind of low-level software. The idea is to provide a slim glue layer to bring the C library and an I/O driver together.

C-Library Back End

There are two possible flavors for an I/O back end: character oriented or string oriented. The OSKit v0.6 C library uses a character-oriented back end. In other words, an output or input-call-back function is invoked for every single character. I considered the overhead that is introduced by a character-oriented interface as too high. In particular, devices with hardware buffers that have to be flushed before the driver returns control to the C library suffer from this kind of interface. I finally preferred the string-oriented back end because the character-oriented interface is only a special case thereof and the calling and flushing overhead is reduced from once per character to once per string.

3.4 L4-Specific Components

Section 3.3 focused on the general components that are very important for the design of maintainable software. This section is equally important, because it describes the design of the L4 specific components.

3.4.1 Basic L4 Abstractions

The fundamental abstractions of the L4 interface, **threads** and **address spaces** (see [Liedtke, 1996]), form the base of Fiasco's kernel design. Additional functionality to manage the hierarchical flex-page mapping relations is necessary to implement the `unmap` system call; this functionality is provided by the mapping database, which is already fully hidden behind an abstract interface. Moreover, the mapping database design and implementation itself is very complex (see [Grützmacher, 1998]), which caused me to leave it almost completely untouched. The only caveat is the restriction to 4-GByte virtual address space and the lack of support for multiple page sizes; but it is beyond the scope of this diploma thesis to redesign the mapping database.

The overall design of the basic abstractions (threads and address spaces) is the one

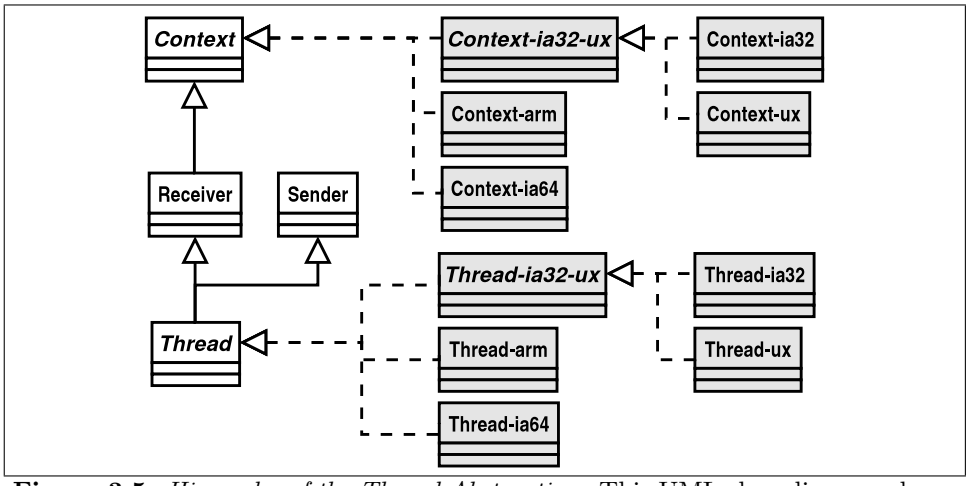


Figure 3.5: *Hierarchy of the Thread Abstraction.* This UML class diagram shows the portability-relevant parts of FIASCO's L4-thread design. The leftmost hierarchy was the initial design and is described in the [Hohmuth, 2003a]. The interfaces of these classes are extended with only a few abstract operations that form the hooks into the architecture specific parts. These architecture-specific parts (the shaded classes) implement the previously defined hooks in a machine-specific manner. The quite complex class structure results from the fully preemptable design of FIASCO. You should look into Section 3.4.1 for a more detailed explanation.

described in [Hohmuth, 2003a]. To satisfy the portability requirements modules and submodules that contained architecture dependent code are subdivided into further fragments. Well-defined hooks, which are declared in the generic parts, provide the connectivity between these newly created fragments. The concrete definition of the *hardware-abstraction hooks* is given in Section 3.4.2. Figure 3.5 and Figure 3.6 on the following page illustrate the class hierarchies of the thread and the address-space abstractions.

The completely preemptable design of FIASCO causes these complex class hierarchies for the basic L4 abstractions. In fully preemptable software, the access to shared resources has to be synchronized. The implementation of FIASCO is based on lock-free and wait-free synchronization primitives (see [Hohmuth and Härtig, 2001]). The wait-free locking scheme uses locking with helping, to avoid priority inversion, and thus depends on switching to the lock-holders execution context. Some thread and address-space operations depend on the locking primitives, to protect shared data structures. The aforementioned dependencies would create a cyclic graph that is avoided by splitting the abstractions of threads and address spaces into two parts. As stated in Section 3.1, circular dependencies should be avoided completely, to maintain testability and portability. The one (low-level) part does not allow any manipulation of shared data structures, but provides the means to switch among different execution contexts (`Context` and `Space_context`). The other (high-level) part encapsulates the

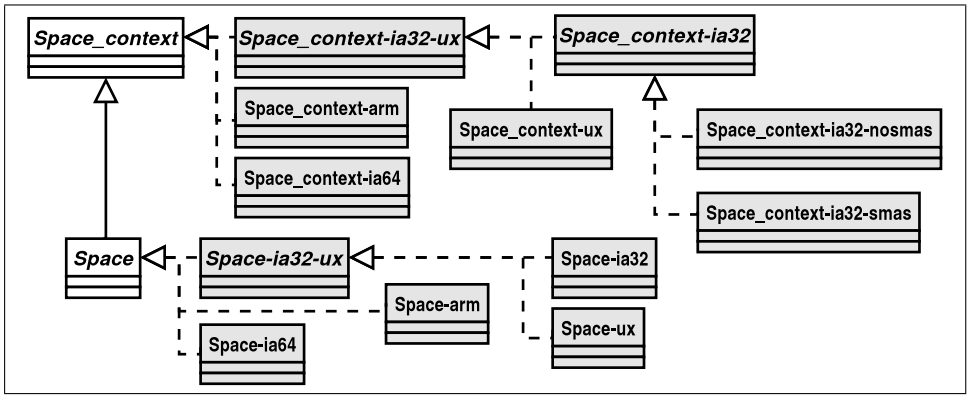


Figure 3.6: *Hierarchy of the Space Abstraction.* Analog to Figure 3.5 on the preceding page, this class diagram shows FIASCO's design of the L4 address-space abstraction. Again, the two level concept (`Space_context` and `Space`) results from the preemptability of FIASCO (see Section 3.4.1). The gray shaded classes are architecture-specific specializations or implementations of the generic interfaces.

manipulation operations (`Thread` and `Space`).

3.4.2 Hardware Layer of L4's Basic Abstractions

The last section described on the general design issues with the basic L4 abstractions. On the other side, the two first-class abstraction of L4 are coupled tightly with a kind of hardware context.

Threads

The thread abstraction provides an execution context, which runs on a certain CPU and can be preempted transparently. From L4's point of view, a thread is subject to a scheduling strategy and can be addressed for IPC. In relation to the underlying hardware, a thread is the execution context of the CPU. For switching among threads, the CPU state of the current thread must be saved and the state of the target thread must be restored. The state of a CPU is obviously completely dependent on the processor architecture and there are no proper means in today's HLLs to express context switches, so the actual switching of the execution context has to be done in architecture-specific functions.

The execution context of a thread is composed of the following parts:

- CPU context, which is the content of the registers and the processor status
- FPU context, which consists of the FPU registers and state (provided that the target architecture features an FPU)
- The address space in which the thread is executing

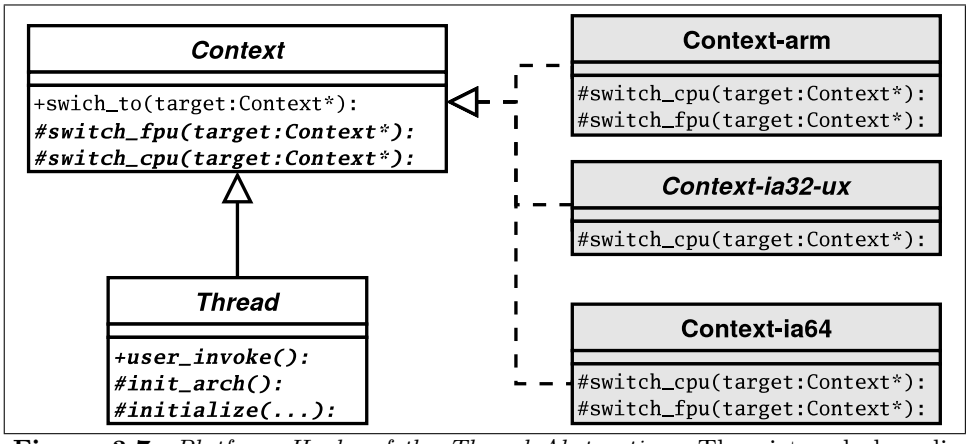


Figure 3.7: *Platform Hooks of the Thread Abstraction.* The pictured class diagram shows the class hierarchy that represents the polymorphic hardware abstraction of an L4 thread. In comparison to Figure 3.5 on page 23, this figure shows the details about the hooks that must be implemented for the different target architectures. You can see, the class Context-ia32-ux has no implementation for `switch_fpu`; this method is implemented in two further specializations Context-ia32 and Context-ux, which are omitted for the lack of space.

The latter part is coupled directly with L4's address-space abstraction. I will treat this later on. The former two parts are encapsulated in two hooks in the `Context` class (see Figure 3.7). The hook `switch_fpu` must save the FPU state or prepare lazy saving mechanisms. After dealing with the FPU the actual CPU context must be switched, which is done in the hook `switch_cpu`. The CPU switching hook must also call the function to switch the address space (`call_switchin_context`) immediately after switching to the target stack. This is necessary, because on thread creation a newly created thread does not leave the `switch_cpu` function as usual, but drops directly into `user_invoke`, which manages the transition to user mode. For a detailed description of the hooks into the architecture-specific part see Section A.2.3 on page A-5 and Section A.2.4 on page A-6.

The FPU context is treated separately because not all platforms feature an FPU, whereas others have a very large FPU state and/or support lazy state handling. On all platforms that feature an FPU, the state is stored into a specifically allocated buffer, private to the thread. It is not necessary to save FPU state on the kernel stack; because the kernel never uses the FPU and thus no nested state saving is needed.

The hooks that are declared in the class `Thread` (see Figure 3.7) are used for thread creation. `Thread::init_arch` is called from the generic constructor of `Thread`, and has to initialize architecture specific thread state (e.g., specific bits in the thread's processor state). The function `Thread::initialize` virtually performs the `lthread_ex_regs` system call. This hook has a generic implementation that fits for most architectures. Nevertheless, IA-32 is an exception, because it uses

varying principles for kernel entries (`int/iret` and `sysenter/sysexit`) that need special handling in the case of `lthread_ex_regs`.

Address Spaces

Address space switching, which is initiated by the call to `call_switchin_context`, follows a similar scheme like thread switching. A generic switching function (`switchin_context`) uses a hook into the architecture specific part (`make_current`). The function `make_current` is responsible for switching the MMU to another address space and to do the proper operations to keep caches and TLBs consistent.

The architecture-specific implementations of L4's address-space abstraction become redundant, once the low-level page-table interface is implemented on all architectures, as proposed in Section 3.3.3.

3.4.3 Exception Handling

In principle, exceptions must be handled according to their origin. Exceptions that originate from user applications, called **user exceptions**, are mostly passed back to user space. Version 2 and version X.0 of L4 use two different concepts for doing this. In version 2, page faults are delivered via IPC to a dedicated pager thread and all other exceptions are passed via native exception emulation, whereas version X.2 delivers all exceptions via IPC. The latter concept is much easier to implement in a generic fashion and therefore preferred for ARM.

The other type of exception is that caused by the kernel itself. The code of the kernel is assumed correct and thus raises only a very limited number of exceptions at well-known locations. The kernel raises virtually only page faults. Page faults can be raised during the long-IPC transmission; such page faults are on behalf of the user application and can be accounted to the user exceptions. They are passed to the pager thread as usual.

All current ports of Fiasco use lazy mechanisms for TCB allocation. These mechanisms are based on page faults in a special TCB area in the kernel address space. Such **kernel page faults** must be completely hidden from the user applications.

Platforms that make use of kernel page faults must provide the corresponding handlers in the architecture dependent part of the thread abstractions.

3.4.4 L4-ABI Abstractions

L4's application binary interface (ABI) can be divided into two basic parts, the ABI types and the system-call conventions.

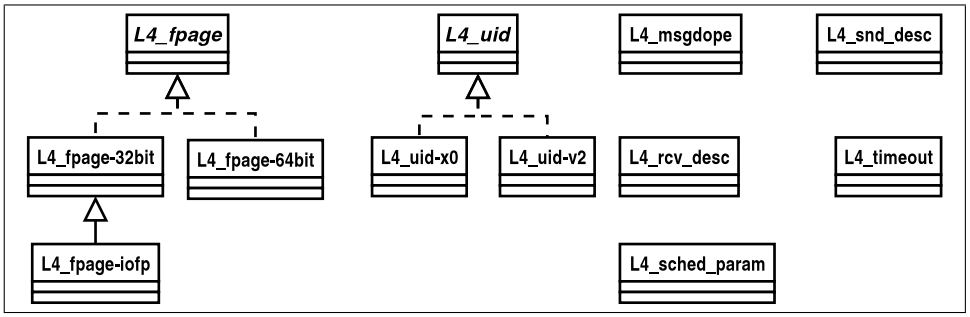


Figure 3.8: *Design of the L4 ABI Types.* This class diagram shows the L4 ABI types and their specific implementations, all interface details are hidden. The boxes with cursive titles are abstract interface definitions and the boxes with normal titles represent implementations. As you can see, some types have the interface definition separated from the implementation and others do not. The separation is always done, when more than one implementation exists on the currently supported architectures and ABI versions.

ABI Types

The L4 specification defines a number of data types as part of the microkernel ABI. You can find this specification in [Liedtke, 1996]. The ABI types transfer almost the same content on any architecture and on the supported ABI versions. The fact that makes them relevant for portability is the differing layout among different machines. Varying word widths and the possibility of highly efficient implementations, dedicated to the target architecture, constitute the variation in the binary layout of these data types. For example, the L4-UID type that is used on IA-32 was initially optimized to calculate the address of the TCB of the corresponding thread with only one 32-Bit-`AND` and one 32-Bit-`OR` operation.

To be able to implement generic system-call logic, this diversity in the data layout must be hidden behind generally defined interfaces. Figure 3.8 gives an overview of these ABI types. As illustrated there, some of the data types do not have any specialized implementation yet and others do. For instance, `L4_msgdope` has just one generic implementation, whereas `L4_fpage` has specializations for 32-Bit architectures and the IA-32-specific I/O flex pages.

Problem with C Bit fields

In the first place, C bit fields seem to be a perfect means to represent L4's ABI data types. However, there is a serious limitation regarding C bit fields: neither the C nor the C++ standard defines a concrete ordering of bit-field members. The result of this freedom is that the GNU compilers for big-endian and little-endian machines generate differing binary representations for the same bit-field declaration.

The implementations of Hazelnut and Pistachio use subtle preprocessor macros

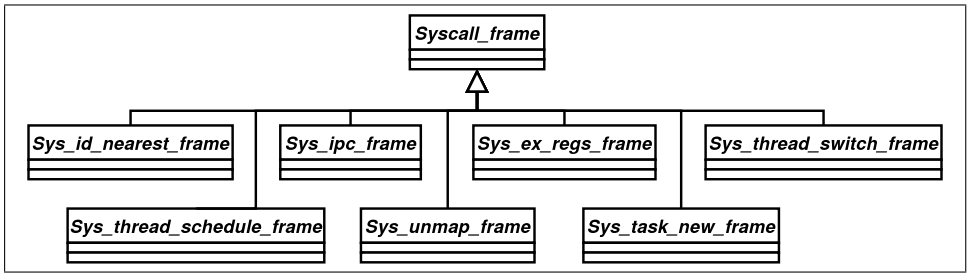


Figure 3.9: *Encapsulated System-Call Parameters.* The implementation of every system call uses its dedicated `sys_XXX_frame` interface to access the parameters. These interfaces are based on the ABI-type definitions presented in Figure 3.8 on the page before and kept completely generic with respect to the underlying architecture. The parameter encapsulations are even aware of the currently supported API versions (v2 and X.0).

for defining bit fields in an endian-independent manner. I considered this solution infeasible, because the C standard also lacks a concrete definition for bit fields wider than an `unsigned int`, and a separate macro for each number of bit-field members is necessary. The implementation of Fiasco therefore does not use C bit fields for representing ABI data types. Fiasco is based on C++ classes with access functions that use mask and shift operations, which are independent of the endianness and compiler.

System-Call Conventions

Another completely architecture specific part of the L4 specification is the system-call ABI, which is the mapping of the system-call parameters to the processor registers or to memory. Again, the obvious way to keep the system-call logic generic is to define an abstract interface for every L4 system call, which provides a means to access the parameters independent from the underlying architecture. I will refer to this layer of abstraction with the term: **in-kernel system-call bindings**. Figure 3.9 gives an overview of the complete set of interface definitions. Figure 3.10 on the facing page shows the interface and the existing implementations for the `id nearest` system call. The class hierarchy of the other system calls is analogous to that of `id nearest`. The complete description of the interfaces of all in-kernel system-call bindings is located in Section A.2.6 on page A-7 and the following.

As a nice side effect, the encapsulation of the ABI types and of the system-call conventions made it possible to easily integrate support for the version-X.0 ABI into Fiasco. Only the `L4_uid` data type, the in-kernel system-call bindings, and the `task new` system call had to be re-implemented.

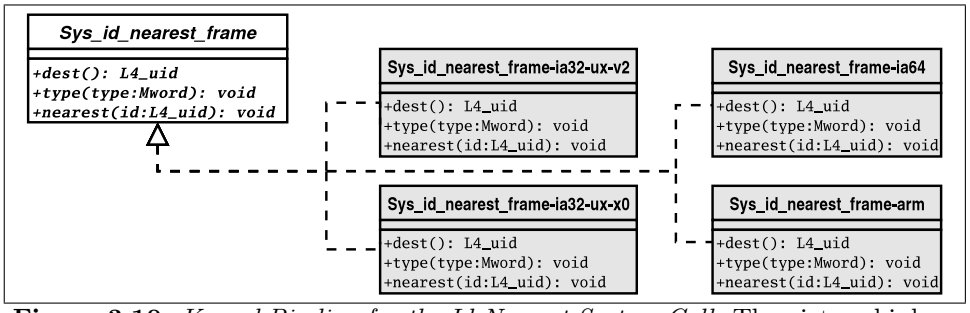


Figure 3.10: *Kernel Binding for the Id-Nearest System Call.* The pictured inheritance tree representatively stands for all in-kernel system-call bindings. As can be seen, each supported architecture and API version has to provide its specific implementation.

3.4.5 Kernel Memory Management

To implement the basic abstractions of L4, a dynamic memory management is necessary. The kernel needs to allocate memory for the following objects:

- Thread control blocks (TCBs)
- Page tables and Page directories
- Mapping nodes, which are an integral part of the mapping database
- FPU context, if the target machine features an FPU

The design of the memory management has to meet various demands: first, single memory pages at arbitrary virtual addresses must be allocated (**page-level allocation**). Secondly, the kernel needs to allocate pages at specific locations in virtual memory (e.g., for TCBs), this is called **virtual-page allocation**. Finally, the **object allocation** is responsible for allocating various memory objects of arbitrary size. Figure 3.11 on the following page illustrates the design of the memory-management subsystem. You can see the three aforementioned parts and their interaction. The overall design of the in-kernel memory management is based on the design of the IA-64 port of Fiasco, which is described in my term paper [Warg, 2002].

The trichotomy of the memory management is based on the logical separation of the different parts and the avoidance of circular dependencies among the address-space and the allocator components. In a few words, the page-table implementation depends on page-level allocation, for allocating memory for page tables. Furthermore, the virtual-page allocation needs the page-table implementation to map the allocated pages to the requested virtual addresses. Thus, page-level and virtual-page allocation cannot be provided by one component.

The third allocator is separated, because a completely different strategy is necessary to manage allocation of many small and highly dynamic objects; in Fiasco, such allocation is basically done with slab allocators.

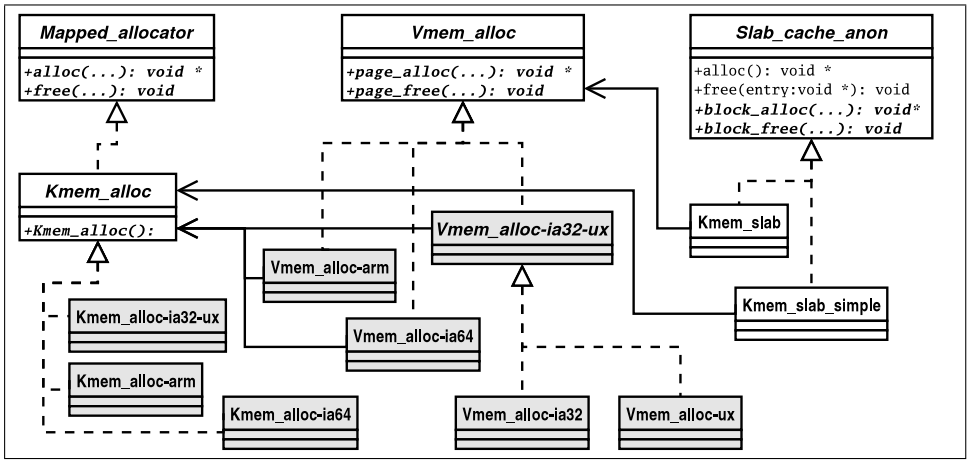


Figure 3.11: *Model of the Memory-Management System.* This figure shows the class structure of FIASCO's memory management. The three main components of the memory management are `Mapped_allocator` for **page-level allocation**, `Vmem_alloc` for **virtual-page allocation**, and `Slab_cache_anon` for **object allocation**. The depicted classes do not show the complete interface definitions.

Page-Level Allocator

The page-level allocator provides the lowest level of memory allocation. All other allocators make use of this allocator either directly or indirectly.

The provided interface consists of methods for allocating and freeing blocks of physically adjacent memory, which is mapped at arbitrary locations in the kernel's address space. Moreover, a method to request the physical address of a previously allocated block, and a method to request the virtual address of a physical frame exist. The latter two methods require that the physical memory lies within the responsibility of the allocator.

In view of portability, this interface allows the implementation of an allocator for any model of physical memory. In particular, the model that is currently used on IA-32[2] is easily implementable. Nevertheless, also models where user-level software has to supply the kernel with memory, as proposed by Andreas Haeberlen in [Haeberlen, 2003] can be supported behind this interface.

Virtual-Page Allocator

The virtual-page allocator is not really an allocator that manages memory resources. `Vmem_alloc` uses the page-level allocator to actually allocate a block of memory at an arbitrary address and then uses the page-table interface (on IA-32 it is directly manipulated) to map the page to a requested address.

[2] the physical memory is linearly mapped into the kernel's address space as a whole

FIASCO uses this mechanism to allocate TCBs, and for allocating larger blocks of memory[3] that do not have to be physically contiguous.

The extra implementation for IA-32, which manipulates the page-table structures directly, should be removed in favor of a generic implementation, once the low-level page-table interface is implemented on IA-32.

3.5 New JDB Design

This section shall not describe the design of a completely functional kernel debugger. The design is restricted to basic debugger core and has the following goals:

- Portability (architecture independence)
- Modularity
- Easy extensibility
- Look and feel of the original JDB
- Coexistence with the original JDB

The two last points are very important. The look and feel should be well known to the people that are accustomed to the kernel debugger. However, the more important goal is a smooth transition from the old kernel debugger to the new modularized and portable debugger. This can only be achieved by a temporary coexistence, because the old kernel debugger provides features for kernel and user-level development that cannot be ported in one piece.

The current IA-32 implementation accomplishes the coexistence through the integration of the `Jdb_core` methods `Jdb_core::has_cmd` and `Jdb_core::exec_cmd` into the traditional event loop.

The class diagram in Figure 3.12 on the next page pictures the design of the basic JDB abstractions. The small and architecture-independent core (`Jdb_core`) provides the command execution and input-parsing environment. The debugger modules provide the actual debugger functionality. These modules must be derived from the class `Jdb_module`, and are not required to deal directly with console input or input-parameter parsing.

All debugger modules register themselves at the debugger core with use of static constructors (see Section 4.2.2). A debugger module must provide at least one command (`Jdb_module::Cmd`) and is member of exactly one category (`Jdb_category`). The category is used to classify the debugger help screen.

The actual improvement of portability is achieved through the modular design. This design allows the programmer to port the debugger functionality that he cur-

[3] i.e., larger than one page

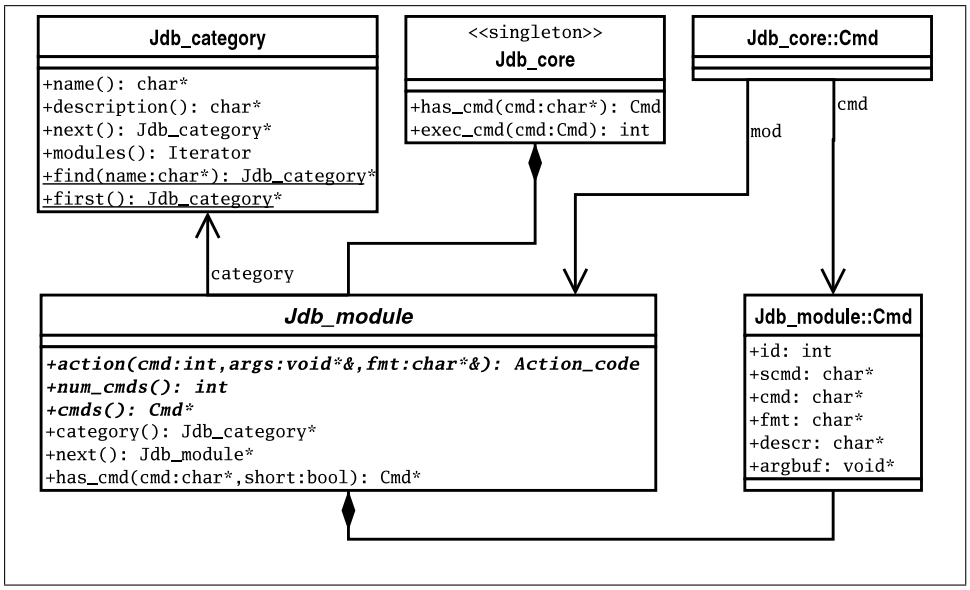


Figure 3.12: *Class Structure of the New JDB.* Figure 3.13 shows exemplary subclasses of `Jdb_module`, which are actually implemented in FIASCO.

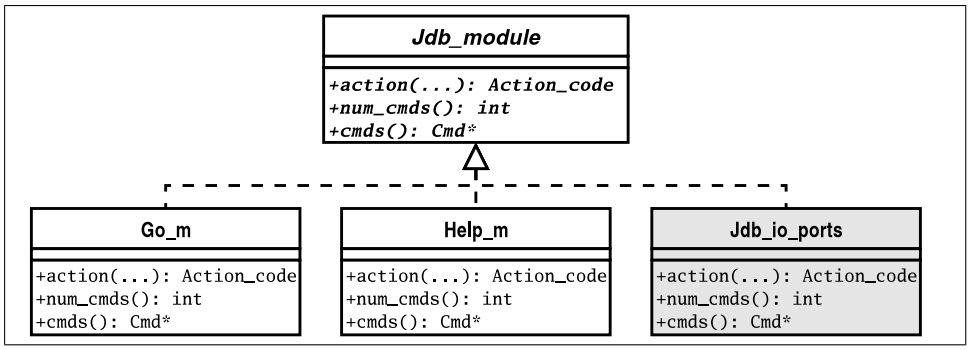


Figure 3.13: *Exemplary Subclasses of Jdb_module.* The pictured class graph shows existing JDB modules.

rently needs, and eases the implementation of machine-specific extensions without interfering with other architectures.

3.6 StrongARM Specific Design

This section contains very specific design issues that can already be taken as a kind of implementation of the generic design mentioned before. However, it is a part of the design that was made in order to implement the ARM port of Fiasco and is therefore located here.

3.6.1 Exception Handling

This section builds upon the discussion of Section 3.1 from [Wiggins, 1999]. The discussion is concerned with the possibilities for handling exceptions and the usage of the privileged modes. In summary, the following two alternatives are evaluated:

- Handling of all exceptions in a single mode of operation (see Section 2.1.1)
- Handling of exceptions in the operating mode corresponding to the exception

The first alternative comes with additional overhead for saving the banked registers (see Section 2.1.1) and switching to the final operating mode. The latter alternative implies that an un-banked register is used as stack pointer, to avoid problems with in-kernel exceptions: assuming, the processor is already in a privileged mode (e.g., the *Abort* mode) and an event causes a mode switch, for example an IRQ; the execution continues with a different stack pointer and not as supposed on the thread's kernel stack. A seemingly simple solution is to load the mode-private stack pointer immediately after entering the different operating mode with the value of the stack pointer of the previous mode. The problem is that the banked registers of other modes are usually not accessible without an explicit switch to the corresponding mode.

I decided to use a single operating mode for all exceptions, because the compilers used do not support the use of an un-banked register as stack pointer and a re-implementation in assembly language is out of the question. The decision seems to be contrary to that from Adam Wiggins in [Wiggins, 1999], but Gauntlet was implemented completely in assembly language and therefore could use an un-banked register as stack pointer and execute in the operating mode corresponding to the raised exception (see Section 2.1.1).

Which operating mode is the best for kernel execution? System calls, which are triggered by well-defined prefetch aborts, as well as normal page faults switch automatically to the *Abort* mode. The *Abort* mode should be the choice, because no extra mode switches are necessary for these performance-critical kernel entries. The disadvantage of the *Abort* mode is that a bit of additional overhead for IRQs and FIQs is introduced, which could be also considered critical; an extra switch

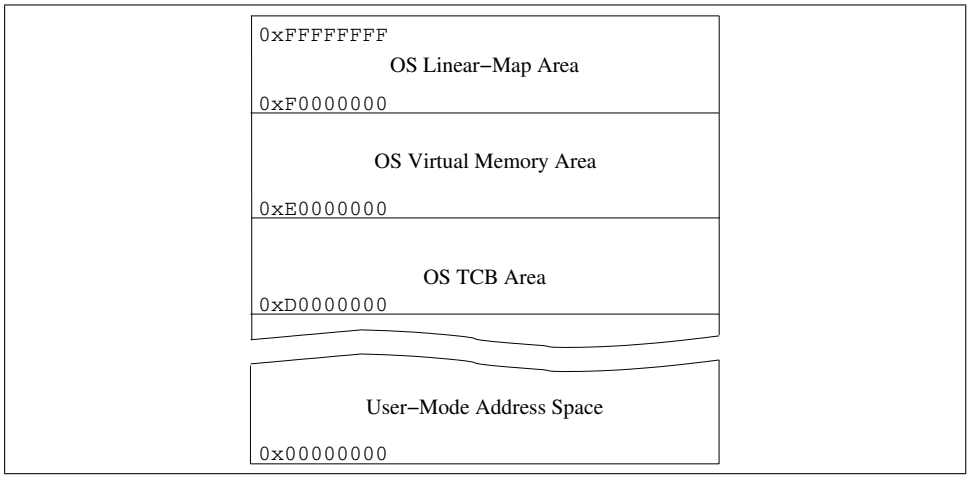


Figure 3.14: *StrongARM Address-Space Layout.* The shown diagram represents the address-space layout as it is currently used on the StrongARM processors. In comparison to the IA-32 address space, the user-kernel boundary is moved up to `0xD0000000`, because StrongARM has its physical RAM at `0xC0000000`. This restriction is not problematic, because the ARM implementation uses version X.0 UIDs, which allows only 2048 Tasks with 64 Threads each. With a TCB size of 2 KBytes, the TCBs fit into the 256 MByte OS TCB Area.

from the IRQ respective the FIQ mode to the *Abort* mode must be executed and FIQ handlers cannot use the extra five registers (see Table 2.1).

3.6.2 ARM Kernel Address Space

The layout of the kernel address space of StrongARM is slightly different from that of other 32-Bit architectures. Usually the uppermost Gigabyte of the address space is allocated for the kernel address space. The StrongARM architecture defines a physical memory map where DRAM is located in a partition above `0xC0000000` (3 GByte) in the physical address space. Hence, Sigma0, which is specified to run in one-to-one mapped memory, must execute above 3 GByte. To make this possible I moved the user-kernel boundary up to `0xD0000000`. If the device features more than 256 MByte of memory or the memory is scattered over the uppermost gigabyte, only the RAM from `0xC0000000` up to `0xD0000000` can be mapped one-to-one, the applications must map the remaining RAM to lower addresses in order to access it. Figure 3.14 shows the complete address-space layout that I have implemented on StrongARM.

Chapter 4

Implementation

In the design chapter, I largely focused on the software-technological point of view. Nevertheless, in the case of a microkernel, such as FIASCO, it is not adequate to have a nice object-oriented design (OOD). One of the perpetual goals is to achieve very high performance. Thus, this chapter does not contain only some war stories about the fiddly implementation process, but it describes the basic concepts of FIASCO's fast polymorphic implementation.

4.1 Polymorphism

Some members of the OS community toughly defend their opinion that C++ ends in too slow code and is not feasible for OS programming. The same people often use C to implement object-oriented code and use structures with function pointers to have a means for polymorphic classes. Examples are the `fops` structure in the Linux kernel, as well as the `kdb_console_t` structure in Pistachio's kernel debugger.

FIASCO, as the first HLL (C++) implementation of the L4 microkernel interface, has proven the contrary. Even the faster but not real-time capable L4 implementations from the University of Karlsruhe (Hazelnut and Pistachio) are implemented in C++.

In spite of the fact that C++ is successfully used for the implementation of operating systems, polymorphism is a red rag for most microkernel people. This point of view originates from the additional level of indirection, which all known C++ compilers introduce to implement virtual functions.

Indeed polymorphism has two occurrences, **runtime polymorphism** and **compile-time polymorphism**. Runtime polymorphism allows different implementations of a certain interface to be exchanged at runtime. This is useful when objects with specific implementations of the same interface coexist in a running system. The commonly used way to implement such runtime polymorphism is to

use an extra level of indirection. This level can be created explicitly, by the use of function pointers, or implicitly using lazy binding by marking a C++ member function as `virtual`.

If the runtime polymorphism is wanted, the extra level of indirection is accepted. On the other hand, the additional overhead is not and cannot be accepted if only one implementation of an interface is used at runtime. In this case, the polymorphism is only a software-technological means. In the design of FIASCO, the complete set of architecture-specific subclasses belongs to this category of polymorphism, which could be resolved at compile time. This compile-time polymorphism does not need an additional level of indirection for method invocation.

The manifest possibilities to resolve the aforementioned problem are:

- Definition of interfaces in C++ include files and multiple implementations in different C++ source files
- Using a C++ compiler that recognizes compile-time polymorphism automatically[1]
- Using the C++ template mechanism
- A language extension for explicitly specifying compile-time polymorphism

The first solution works perfectly as long as no inline functions are used and the *derived* classes do not extend the interface. The latter means that specific classes that are derived from a certain super class are not able to define extra operations or attributes. The problem with inlining is that C++ inline functions must be implemented in the header file where they are declared, because the compiler needs the source code of an inline function at every invocation of it. Furthermore, it is not feasible to use non-inline functions, because the introduced overhead cannot be tolerated for the fine-grained encapsulation that is presently used in FIASCO.

The second solution is not possible with the available C++ compilers; I am not aware of any compiler that performs such an optimization. Furthermore, it is not trivial to detect compile-time polymorphism automatically. The compiler has to do a complex data-flow analysis on the complete source code. In the general case, it is not sufficient to work on a per-file basis, as usually done. After all, developing such a compiler is clearly beyond the scope of this work.

The *template solution* is a possible way to implement compile-time polymorphism. Andrei Alexandrescu describes the principles of this solution in [Alexandrescu, 2001] under the term *Policy-Based Class Design*. With this approach, generic code is located in class templates and the machine specific components are defined as template parameters. With respect to the existing source code of FIASCO, this alternative requires major restructuring.

Due to the just mentioned difficulties, the last solution remains as a possible opportunity. Even this way requires an extra processing of the source code, but expensive

[1] Uwe Dannowski from University of Karlsruhe is currently working that topic.

Program 4.1 *Explicit Compile-Time Polymorphism (Interface).* This piece of code looks like a normal C++ class declaration. The peculiar piece is the preprocessor term 'INTERFACE:' that instructs Preprocess to handle the subsequent code as interface definition. Program 4.2 on the next page shows the counterpart to the interface section, the implementation section. The two sections can also be placed in a single file, but for realization of compile-time polymorphism they have to be put into separate files.

file: 'entry_frame.cpp'

```
INTERFACE:
...
class Sys_id_nearest_frame : public Syscall_frame
{
public:
 L4_uid dest() const;
 void type( Mword type );
 void nearest( L4_uid id );
};
```

auto-recognition of compile-time polymorphism is not necessary. Another thing that led me to the explicit solution was Preprocess (see [Hohmuth, 2003b]), the special C++ preprocessor that is already used for the implementation of Fiasco. Only minor extensions to Preprocess were necessary to support explicit compile-time polymorphism.

Program 4.1 and Program 4.2 on the following page show an example how compile-time polymorphism can be implemented with the help of Preprocess.

4.2 Bootstrap

The boot procedure of Fiasco is subdivided into two major stages. The first stage runs in an environment that is provided by the platform's boot loader. The second stage is the actual kernel bootstrap itself.

4.2.1 Stage 1, Boot Subsystem

This boot stage is bound very tightly to the underlying platform and the used boot loader. Thus, it is not really possible to provide a platform and architecture-independent Boot subsystem. The purpose of the Boot subsystem is to set up the CPU and the MMU for the execution of the kernel.

The main steps that are necessary for initializing the kernel's execution environment are the following:

Program 4.2 *Explicit Compile-Time Polymorphism (Implementation).* This code snippet presents one possible implementation of the abstract interface shown in Program 4.1 on the preceding page. The preprocessor combines the two files at compile-time, thus another implementation can be selected by choosing another implementation file at the invocation of Preprocess. The key term 'IMPLEMENTATION[*ext*]:' initiates the implementation section of the submodule with the given extension. The implementations of the methods are marked with the keyword 'IMPLEMENT', which implies that the respective interface is already declared elsewhere.

file: 'entry_frame-ia32-ux-x0.cpp'

```
IMPLEMENTATION[ia32-ux-x0]:
...
IMPLEMENT inline L4_uid Sys_id_nearest_frame::dest() const
{
  return L4_uid( esi );
}

IMPLEMENT inline void Sys_id_nearest_frame::type( Mword type )
{
  eax = type;
}

IMPLEMENT inline void Sys_id_nearest_frame::nearest( L4_uid id )
{
  esi = id.raw();
}
```

- Loading of the actual kernel
- Enabling the MMU and populating the appropriate mappings
- Transferring control to the kernel code

Depending on the underlying platform, the loading of the kernel can also be done after enabling virtual memory.

The only part that is shared among the supported architectures is an integrated ELF interpreter, which can handle statically linked ELF images. The current versions of Fiasco embed the kernel's ELF image into the actual bootable image.

The ARM boot image also contains additional images for user-level applications, such as Sigma0 and the root task.

4.2.2 Stage 2, In-Kernel Bootstrap

The in-kernel bootstrap of FIASCO is based on two mechanisms: static constructors, and the actual `main` function. Static constructors are used, because C++ initializes global objects automatically through static constructors. With respect to portability, the mechanism of static constructors provides a perfect means for modularizing the kernel while the modules need to be initialized at boot up.

Static Constructors

A static constructor in my terminology is a trivial C function that initializes a statically defined data structure, such as a global object. The static constructors have to be executed before the actual program enters the `main` function, because statically defined objects must already be in an initialized state at this point of execution. The nature of statically defined objects is that their definitions may be distributed among different source-code files and therefore among the created object files (linker units), which implies that only the linker knows the complete list of all static constructors that have to be executed.

If the statically defined object is a C++ object, the compiler puts a pointer to the object's constructor into the appropriate list in the object file. The linker finally combines the constructor lists of the different object files. However, there are C data structures and hardware objects that are not directly supported by the C++ compiler. To have a means for initializing these objects, without the knowledge of the main program, `static_init.h` defines four macros that allow the use of normal C functions as static constructors. These *static-init* macros put a pointer to the constructor function into the appropriate list in the object file. Program 4.3 on the next page and Program 4.4 on page 41 show examples for the definition of static constructors as they are used in FIASCO.

There are actually two flavors of static constructors: constructors that have to be executed in a defined order due to dependencies among the initialized objects, and constructors of objects that have no dependencies among each other. The former kind will be called **prioritized static constructors**, and the latter group is the **simple static constructors**. The only difference in the declaration is the additional priority (Program 4.4).

Startup Constructor

One of the integral parts of the boot sequence is the static constructor `startup_system`. This constructor is defined with the highest possible priority and therefore is the very first function that is executed during boot up. The job of `startup_system` is to initialize all mandatory components of the microkernel, which cannot be configured out. For instance, the memory management, IRQs, and the interval timer are components that are initialized explicitly in `startup_system`. These mandatory components often have dependencies among each other and therefore depend on their initialization order.

Program 4.3 *Simple Static Constructors Example.* The three code snippets depict the principles that FIASCO uses to declare static constructors for certain objects. Snippet (a) is a statically defined instance of a normal C++ object. Snippet (b) shows the method of choice for initializing singleton objects. In Snippet (c) a normal C function is marked as static constructor. Each of these ways does also support an additional priority, like in Program 4.4 on the next page.

(a) Global C++ Object

```
class C {
public:
 C();
};

C::C() {
 /* do some init stuff */
}
...
C object_of_c;
```

(b) Singleton Global C++ Object

```
#include "static_init.h"
class C {
public:
 static void init();
};

C::init() {
 /* do some init stuff */
}
...
/* mark C::init() as static */
/* constructor */
STATIC_INITIALIZE(C);
```

(c) C Function

```
#include "static_init.h"
void constructor_func() {
 /* do the work */
}
...
STATIC_INITIALIZER(constructor_func);
```

Each of these components could also declare its own static constructor with the appropriate priority to satisfy the dependencies among them, but this is much more complicated to understand and not necessary because the components are mandatory parts of the microkernel and well known at programming time.

4.3 The Build System

This section gives an overview of the build system. The build system is based on GNU Make. A big part of the Makefile logic is actually necessary for creating the source-file dependencies; Make uses these dependencies to build the minimal set of files. Concerning portability, the part that is responsible for the architecture-dependent build process is of interest.

There are actually two parts that control the architecture-specific build. The first part is located in files named `Makeconf.<arch>`, and defines tools, such as compil-

Program 4.4 *Prioritized Static Constructors Example.* The mode of action of these examples is almost the same like in Program 4.3 on the facing page. The only difference is the priority `X`, which is additionally specified.

(a) Global C++ Object

```
#include "static_init.h"
class C {
public:
 C();
};

C::C() {
 /* do some init stuff */
}
...
C object_of_c INIT_PRIORITY(X);
```

(b) Singleton Global C++ Object

```
#include "static_init.h"
class C {
public:
 static void init();
};

C::init() {
 /* do some init stuff */
}
...
/* mark C::init() as */
/* static constructor */
STATIC_INITIALIZE_P(C,X);
```

(c) C Function

```
#include "static_init.h"
void constructor_func() {
 /* do the work */
}
...
STATIC_INITIALIZER_P(constructor_func,X);
```

ers, assembler, and linker. The second part, the Modules File (`Modules.<arch>`, see `src/README` in the Fiasco source directory), defines the subsystems and their exact composition. The two aforementioned files exist for each supported target architecture. Program 4.5 on the next page shows an example excerpt from the IA-32 Modules File.

The final linking of each subsystem is defined in files named `Makerules.<subsystem>`. If there are architecture specific rules necessary to do the final linking the `Makerules.<subsystem>` file must include a *Makerules* file from the directory of the appropriate architecture.

The target architecture, and hence the used Modules File, is determined by the configuration variable `CONFIG_XARCH`. All configuration variables are defined in the file `globalconfig.out`. This file can be modified either by hand or via the interactive configuration tool, which handles dependencies between the configuration options and keeps the file consistent.

Program 4.5 *Example Modules File.* This excerpt from the IA-32 Modules file shows the declaration of the subsystems that are build for IA-32 and two examples for the subsystem composition. The variable *SUBSYSTEMS* contains the list of subsystems. The build target for every subsystem must be assigned to a variable named <*subsystem*>. The modules that constitute a subsystem have to be assigned to *INTERFACES_*<*subsystem*>. If a specific module is composed of different submodules, the list of submodules must be assigned to <*module*>*_IMPL*.

```
SUBSYSTEMS = JABI ABI DRIVERS KERNEL CRT0 BOOT LIBK LIBAMM \
             LIBLMM CHECKSUM CXXLIB MINILIBC LIBKERN TCBOFFSET
...
# ABI Subsystem
#
ABI            := libabi.a
VPATH          += abi/$(CONFIG_XARCH) abi
INTERFACES_ABI := l4_types kip
l4_types_IMPL  := l4_types l4_types-$(CONFIG_ABI) \
                  l4_types-32bit l4_types-iofp
kip_IMPL       := kip kip-ia32
...
# DRIVERS subsystem
#
DRIVERS            := libdrivers.a libgluedriverslibc.a
VPATH              += drivers/$(CONFIG_XARCH) drivers
PRIVATE_INCDIR     += drivers/$(CONFIG_XARCH) drivers
INTERFACES_DRIVERS := mux_console console filter_console \
                      keyb io uart vga_console
uart_IMPL          := uart uart-16550
keyb_IMPL          := keyb keyb-pc
io_IMPL            := io io-ia32
CXXSRC_DRIVERS     := glue_libc.cc
```

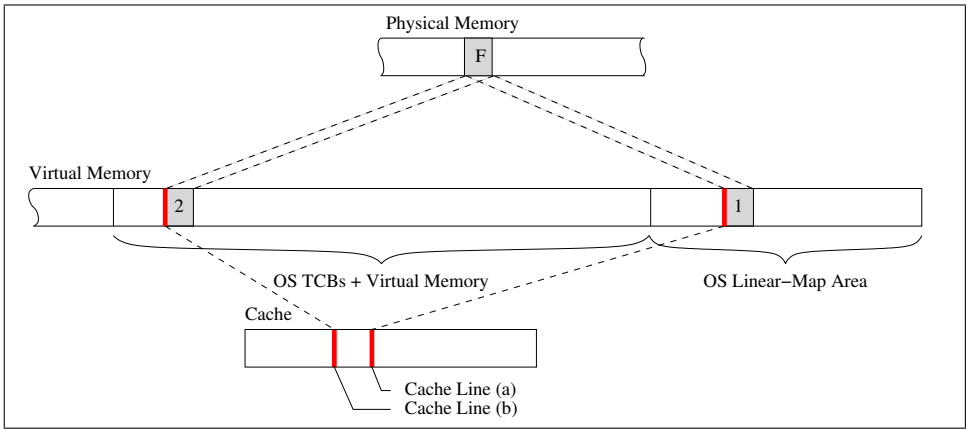


Figure 4.1: *Memory Aliasing Example.* This figure shows how contents of the physical frame F may reside in multiple cache lines at the same time, provided that the frame is aliased at different virtual addresses.

4.4 StrongARM Implementation Details

This section covers the details about the StrongARM implementation. The focus is on the memory-system architecture, which brought up tricky issues.

Page Tables and Fully Virtual Caches

Hardware-walked page tables in conjunction with fully virtual caches yield a major problem. Manipulations on page tables must be enforced to the main memory before any access to the modified virtual memory regions. In particular, this means that manipulations on the currently active page table must be written back to memory instantly. Page tables of inactive tasks are written back implicitly; on StrongARM, caches must be written back and invalidated completely before switching over to another page table, because the caches do not support ASIDs and are tagged and indexed with virtual addresses.

Fiasco uses an ARM cache-control instruction in its page-table implementation to write back the single cache line that contains the manipulated page-table entry. This instruction avoids unnecessary pollution of the data cache, which occurs if the instruction sequence for complete cache write back is used.

Aliasing and Fully Virtual Caches

The ARM reference manual says a physical page must not be mapped at more than one virtual address at the same time (aliased), except the caches and the write buffer are disabled for this page. Figure 4.1 depicts that aliasing in combination with fully virtual caches results in inconsistent memory contents.

Fiasco and Aliasing On the one hand, FIASCO uses lazy mechanisms for TCB allocation that are based on aliasing of memory pages. On the other hand, programmers did not care about aliasing, because it was not an issue on supported architectures other than StrongARM.

The lazy memory allocation uses a read-only zero page, which is mapped into the TCB area when a read instruction accesses a TCB for the first time; the zero page may be mapped multiple times into the TCB area, if multiple TCBs were accessed by read instructions. Nevertheless, the aliasing of the zero page is harmless, because the zero page is always mapped read only, and hence, cannot cause inconsistent views of the main memory. According to Figure 4.1 on the page before, cache line a and cache line b always contain the same value and never become dirty, because write access is denied.

A major problem arose from aliases that exist because implementers did not care about. The virtual-page allocator uses `Kmem_alloc` to allocate a page at an arbitrary virtual address and maps it to another virtual address. Thus, `Vmem_alloc` creates an alias for the allocated page. The cache contents may become inconsistent if the first few bytes of the newly allocated page are modified. According to Figure 4.1 on the preceding page, cache line a contains stale allocator data, and cache line b contains up-to-date TCB data. If first cache line b and then cache line a is written back, the next read operation will see the stale data, whereas the actual TCB data is lost.

The current solution is, flushing the cache after the re-mapping operation in `Vmem_alloc`, which removes the stale cache contents before any access to the remapped page can occur. A better solution would be, using cache-control instructions to invalidate the cache location with the stale allocator data. However, the latter solution requires certain changes in the implementation of the page-level allocator.

Chapter 5

Future Work

5.1 General Issues

There are still open topics that affect all target architectures. An important issue is the implementation of the generic page-table interface for all architectures, because this would remove further duplicated code: The L4 address-space abstractions could be implemented in a generic manner, and the virtual-page allocator, `Vmem_alloc` (see Section 3.4.5 on page 29) could get rid of the special IA-32 implementation, which manipulates page-table structures directly.

Currently no clean and flexible protocol exists to hand the machine's physical memory layout into the kernel (at least in version 2 and X.0), which is the cause of extra code in `Kmem_alloc`. The main difference among the implementations of the page-level allocator is the initialization routine that claims the memory for all kernel allocators. The page-level allocator needs to know where free memory is located and must mark the claimed memory as used, in order to advise Sigma0 to keep the memory reserved for the kernel.

Another open issue is a mapping database that supports multiple page sizes (even more than two page sizes), and large and sparse address spaces, in particular larger than 4 GByte. The support of arbitrary page sizes could increase the TLB coverage on systems that feature multiple pages sizes in their TLB. In particular, IA-64 and ARM could gain application performance.

An interesting topic is the system timer. One-shot timers, which are also known as aperiodic timers, can help to substantially decrease the frequency of timer interrupts while maintaining timer accuracy. On architectures that provide fast and efficient programming of timer events (e.g., StrongARM) this could gain overall performance. On architectures that suffer from their inefficient timer programming the abstraction should be easily implementable with periodic timers.

In addition to the aforementioned topics, which are mostly about functionality, there is an open issue with the structure of Fiasco's source code. The refactored

source code is scattered over many small files, which often leads to confusion while navigating through the code. Many functions have different implementations corresponding to different concerns, and hence finding the relevant piece of code may be quite difficult. There are possible solutions for this problem that should be evaluated:

- A directory structure, in which the architectures are the first level and the subsystems the second level
- Farther refactoring of big classes, such as `Thread` into smaller subclasses
- Symbolic links in the build directory that point to the source directories of the current architecture

These solutions are not mutual exclusive, they should rather be combined to achieve better source-code properties.

5.2 StrongARM Topics

There are several features of the StrongARM architecture that are currently unused. Fast address-space switches, as explained in Section 2.2.2 on page 10, should be implemented, which would improve the application performance substantially.

The use of the *minicache* and the read buffer, which are available on StrongARM, should further reduce the influence of the microkernel on the cache-miss rate of the user-level applications. For instance, the TCBs and the page tables could be cached within the minicache.

StrongARM also provides an efficiently programmable timer circuit. The timer events are determined by the contents of a few memory-mapped registers. The use of one-shot timers for scheduling and timeout events should be evaluated with respect to the interrupt frequencies and the user-level application performance.

An open issue of the StrongARM port is the implementation of the exception delivery mechanism (see Section 3.4.3). I would prefer a method similar to that used in version X.2, where all exceptions are delivered via IPC to a dedicated exception handler thread.

A very big topic is the complete user-level environment. Currently there is no appropriate C library for ARM user-level applications. Hence, at the moment there is only a Sigma0 and a simple test server running. Both are linked against the MiniLibC from the kernel and are build directly with the kernel.

The lack of time and a *malicious* C++ compiler prevented me from measuring IPC performance. The user-level test program crashed with strange memory accesses, which turned out to come from bogus register contents. The compiler did not save allocated registers before system calls, even though all registers are in the clobber list of the assembler statement for the system call.

Another field of open work is the unit and functional testing of the ARM components. The architecture-independent part of the kernel can be tested on any architecture; the unit tests that already exist are feasible for that. However, a set of low-level tests should be implemented to verify for example the implementation of the page-table interface or of atomic operations.

Chapter 6

Summary

This chapter shall give a brief summary of what was achieved during my work. First, I will treat the software-technological changes. The second part deals with the ARM related topics.

The most important step towards more portability was the refactoring of the source code in completely generic and machine or hardware-dependent units. The refactoring is based on class definitions with generic interfaces and implementations. The generic implementations make use of generic interfaces that are specialized by compile-time polymorphic inheritance. This kind of polymorphism can be completely resolved at compile time and introduces no runtime overhead.

It turned out that Preprocess is a lot more versatile than thought. It helped for modularizing completely orthogonal aspects of Fiasco. The resulting design goes into the direction of ASOD, which may be worth it considering in the future (see [Coady et al., 2001]). An open issue with respect to ASOD is the tool support and the complexity of the description for intermixing the different aspects.

The build system, as used now, supports the composition of modules from different submodules and uses Preprocess (already used before my work) for resolving the compile-time polymorphism. A file per target architecture, which can internally depend on further configuration options, determines the assembly of the final image.

Another important property of the source code is the absence of circular dependencies among the different modules. This property is the base for unit testing, which is very helpful for maintaining functionality while changing the implementation behind the interfaces, and simplifies the porting work. In the absence of circular dependencies the modules and their dependencies result in an directed acyclic graph; porting can be started at the low-level modules, which do not depend on anything else and go towards the top-level modules, whereas each level can be tested independently of the higher levels.

The independence of the system-call logic from the underlying processor archi-

tecture and partially also from the provided ABI version is achieved through the complete encapsulation of L4's ABI types and the system-call parameters into C++ classes. The ABI types are based on mask and shift operations and not on C bit fields, what makes them aware of little and big-endian architectures; the use of C++-source-level inlining should provide compiler results as efficient as the use of bit fields. The in-kernel system-call bindings provide the system-call logic with a completely generic mechanism to access the system-call parameters.

I provided a generic page-table interface, which should ease the transfer to new architectures or new page-table layouts (e.g., on IA-64) even without a complete understanding of the class hierarchies of Fiasco's address-space abstractions.

A problem when porting Fiasco to a new architecture was the kernel debugger (JDB). The original JDB was dedicated to IA-32 and had a monolithic design that complicated, if not impeded, a port to a new architecture. The Fiasco kernel debugger is now easily removable from the kernel. I provided generic dummy modules for disabling the logging features, which depend on JDB and made the kernel implementation dependent on the kernel debugger, which is problematic anyway. Additionally, I designed and implemented a basic modularized kernel debugger, which eases the porting work and supports the implementation of easily pluggable modules.

The ARM port, which I did during this work, is fully integrated in the main branch of Fiasco. It runs on the iPAQ H3800 with a StrongARM processor. It is not yet well tested because of missing user-level applications.

Appendix A

Architecture-Specific Hooks

This appendix contains the description for all interfaces that need an architecture-specific implementation.

A.1 Kernel-External Hooks

A.1.1 Proc Class

Low-level (L4 independent) CPU abstraction.

`void` **Proc::cli** ( )

This function must disable external interrupts on the local CPU.

`void` **Proc::sti** ( )

This function must enable external interrupts on the local CPU.

`Proc::Status` **Proc::interrupts** ( )

This function must return the state (enabled/disabled) interrupts on the local CPU.

`Proc::Status` **Proc::cli_save** ( )

This function must disable the interrupts on the local CPU and returns the previous state (enabled/disabled).

`void` **Proc::sti_restore** (`Proc::Status state`)

This method must restore the interrupt state from *state*.

`void` **Proc::pause** ( )

This method must be implemented to prevent the CPU from overheating and/or blocking concurrent hyper threads in the case of tight spin loops.

`void` **Proc::halt** ( )

This method should put the CPU into sleep mode, which is left by a later IRQ.

`void` **Proc::stack_pointer** (`Mword sp`)

This method has to set the CPU's stack pointer to *sp*.

A.1.2 Atomic Operations

Operations for atomic read-write access to the main memory.

```
bool up_cas_unsafe ( Mword *ptr,
                     Mword oldval,
                     Mword newval )
```

This method is must be implemented to do a uniprocessor *compare and swap* operation on the machine word that is addressed by *ptr*. The function must return *true* on successful operation, and *false* else.

```
bool smp_cas_unsafe ( Mword *ptr,
                      Mword oldval,
                      Mword newval )
```

This method is must be implemented to do a multiprocessor-safe *compare and swap* operation on the machine word that is addressed by *ptr*. The function must return *true* on successful operation, and *false* else.

```
bool up_cas2_unsafe ( Mword *ptr,
                      Mword *oldval,
                      Mword *newval )
```

This method is must be implemented to do a uniprocessor *compare and swap* operation on the two adjacent machine words that are addressed by *ptr*. The function must return *true* on successful operation, and *false* else.

```
bool smp_cas2_unsafe ( Mword *ptr,
                       Mword *oldval,
                       Mword *newval )
```

This method is must be implemented to do a multiprocessor-safe *compare and swap* operation on the two adjacent machine words that are addressed by *ptr*. The function must return *true* on successful operation, and *false* else.

`bool` **up_tas** (`Mword *lock`)

This function must be implemented to do a uniprocessor atomic *test and set* operation on the machine word addressed by *lock*.

`bool` **smp_tas** (`Mword *lock`)

This function must be implemented to do a multiprocessor-safe atomic *test and set* operation on the machine word addressed by *lock*.

A.2 Kernel-Internal Hooks

A.2.1 Page_table Class

`void*` **Page_table::operator new** (`size_t`)

Allocate memory for a new page table.

`void` **Page_table::operator delete** (`void *`)

Free the memory of the given page table.

`void` **Page_table::init** ( )

Initialize the paging mechanisms for the kernel.

Page_table::Page_table ( )

Create a new (empty) page table.

```
Page_table::Status Page_table::insert ( P_ptr<void> pa,
                                        void *va, size_t s,
                                        Page::Attribs a )
```

Insert a mapping from virtual address *va* to physical address *pa* with the size *s* and attributes *a*. If there is already a mapping for the given virtual address, *E_EXISTS* must be returned.

```
Page_table::Status Page_table::replace ( P_ptr<void> pa,
                                         void *va, size_t s,
                                         Page::Attribs a )
```

Replace the mapping for virtual address *va* with the new values given in *pa, s,* and *a*.

`Page_table::Status` **Page_table::change** (`void *va, Page::Attribs a`)

Change the access rights of the given address *va* to *a*. If there is no mapping for *va*, *E_INVALID* must be returned.

`Page_table::Status` **Page_table::remove** (`void *va`)

Remove the mapping for the virtual address *va*. If there is no mapping for *va*, *E_INVALID* must be returned.

`P_ptr<void>` **Page_table::lookup** (`void *va, size_t *s, Page::Attribs *a`) `const`

Returns the mapping for the virtual address *va*. If *s* is not *null*, the size of the found page is returned in *s*. If *a* is not *null*, the page attributes of the found page are returned. The returned pointer is *null* if there is no valid mapping.

`Page_table::Status` **Page_table::insert_invalid** (`void *va, size_t s, Mword val`)

Insert the given value (*val*) as invalid mapping at address *va* and with size *s* into the page table. If there is already a valid mapping, *E_EXISTS* must be returned.

`Mword` **Page_table::lookup_invalid** (`void *va`) `const`

Returns the invalid mapping at virtual address *va*. If there is a valid mapping, *(Mword)-1* must be returned.

`void` **Page_table::copy_in** (`void *my_base, Page_table *o, void *base, size_t size`)

Copy all mappings from *o*, starting with *base* and within the size *size*, to this page table, starting at *my_base*.

`Page_table*` **Page_table::current** ( )

Return a pointer to the currently active page table.

`size_t const` **Page_table::num_page_sizes** ( )

Returns the number of supported page sizes.

`size_t const*const` **Page_table::page_sizes** ( )

Returns a constant array with *num_page_sizes* entries that contains the supported page sizes in bytes (starting with the smallest).

`size_t const*const` **Page_table::page_shifts** ( )

Returns a constant array with *num_page_sizes* entries. Each entry must contain the number of page-offset bits according to the page size returned by *page_sizes*.

`Page_table*` **Page_table::activate** (`P_ptr<Page_table> page_table`)

Activate the given page table and return a pointer its virtual address. This means the given page table must be activated in the MMU and all operations that are necessary to ensure memory consistency must be executed.

A.2.2 Kmem Class

The `Kmem` class represents the static (read only) abstraction of the kernel address space. The following constants must be defined:

`Kmem::mem_tcbs` Start address of the TCB area

`Kmem::mem_tbcs_end` End address of the TCB area

`Kmem::mem_user_max` End address of the user address space

`Kmem::mem_kernel_max` End address of the kernel address space

`Kmem::ipc_window_start` Start address of the IPC window

`Kmem::ipc_window_end` End address of the IPC window

`Mword` **Kmem::is_kmem_page_fault** (`Mword pfa, Mword error`)

Checks for kernel page fault.

`Mword` **Kmem::is_tcb_page_fault** (`Mword pfa, Mword error`)

Checks for page fault in TCB area.

`Mword` **Kmem::is_ipc_page_fault** (`Mword pfa, Mword error`)

Checks for user-mode page fault.

`Mword` **Kmem::is_io_bitmap_page_fault** (`Mword pfa, Mword error`)

Checks for page fault in the IA-32 I/O bitmap.

A.2.3 Context Class

Low-level part of Fiasco's thread abstraction.

`void` **Context::switch_fpu** (`Context *target`)

This hook must save the FPU context of the current thread and restore the FPU context of the target thread. On architectures with lazy FPU handling this method has to prepare the FPU according to their current owner.

`Mword` **Context::switch_cpu** (`Context *target`)

`switch_cpu` has to switch the execution context of the CPU and call the function `call_switchin_context`, which handles address-space switch over.

`void` **Context::switchin_context** ( )

On most architectures, this function calls `Space_context::switchin_context`. IA-32 requires additionally that the new stack pointer is written into the TSS.

A.2.4 Thread Class

High-level part of FIASCO's thread abstraction.

```
bool Thread::initialize ( Address ip, Address sp,
                          Thread *pager,
                          Thread *preempter,
                          Address *o_ip,
                          Address *o_sp,
                          Thread **o_pager,
                          Thread **o_preempter,
                          Mword *o_eflags )
```

This method has to prepare the user-level state of the thread, according to the given parameters. It is basically the implementation of the `thread_ex_regs` system call.

`void` **Thread::user_invoke** ( )

This method has to manage the transition to user-mode for newly created threads.

`bool` **Thread::handle_sigma0_page_fault** (`Address pfa`)

This method becomes redundant with the implementation of a generic page-table interface. At the moment the IA-32 version uses CPU feature flags to decide whether super pages shall be used.

A.2.5 Kernel_thread Class

Special kernel thread, derived from the `Thread` class. This thread is responsible for the kernel startup and finally implements the *idle loop*.

`void` **Kernel_thread::free_init_call_section** ( )

Should make the Initcall sections of the kernel unaccessible to the kernel. This means a complete un-mapping in FIASCO/UX, filling with undefined opcodes on IA-32, and do nothing on ARM.

`void` **Kernel_thread::bootstrap_arch** ( )

Initializing some architecture specific kernel-thread stuff, may be empty.

`void` **Kernel_thread::init_workload** ( )

Is only specialized on native IA-32, because of the old way sigma0 gets the address of the kernel info page (i.e., via an initial stack).

A.2.6 In-Kernel System-Call Bindings

The following sections describe the in-kernel system-call bindings for each system call. These bindings must be implemented to read the system-call parameters from the appropriate registers or memory locations.

Sys_ipc_frame

Binding for the version 2 and version X.0 IPC system call.

`void` **Sys_ipc_frame::rcv_source** (`L4_uid id`)

Set the IPC source for the recipient.

`L4_uid` **Sys_ipc_frame::rcv_source** ( )

Get the IPC source for the recipient.

`L4_uid` **Sys_ipc_frame::snd_dest** ( ) `const`

Get the destination for the IPC.

`Mword` **Sys_ipc_frame::has_snd_dest** ( ) `const`

Does the IPC have a destination.

`Mword` **Sys_ipc_frame::irq** ( ) `const`

Get the IRQ destination of the IPC.

`void` **Sys_ipc_frame::snd_desc** (`Mword w`)

Set the send descriptor.

`L4_snd_desc` **Sys_ipc_frame::snd_desc** ( ) `const`

Get the send descriptor.

`void` **Sys_ipc_frame::rcv_desc** (`L4_rcv_desc d`)

Set the receive descriptor.

`L4_rcv_desc` **Sys_ipc_frame::rcv_desc** ( ) `const`

Get the receive descriptor.

`L4_timeout` **Sys_ipc_frame::timeout** ( ) `const`

Get the message timeout.

`Mword` **Sys_ipc_frame::msg_word** (`unsigned index`) `const`

Get the given register message word.

`void` **Sys_ipc_frame::set_msg_word** (`unsigned index, Mword value`)

Set the given message word to the given value.

`void` **Sys_ipc_frame::copy_msg** (`Sys_ipc_frame *to`) `const`

Copy this msg to the given IPC data.

`L4_msgdope` **Sys_ipc_frame::msg_dope** ( ) `const`

Get the msg dope.

`void` **Sys_ipc_frame::msg_dope_set_error** (`Mword err`)

Set the error code.

`void` **Sys_ipc_frame::msg_dope** (`L4_msgdope d`)

Set the msg dope.

`void` **Sys_ipc_frame::msg_dope_combine** (`L4_msgdope d`)

OR some extra bits to the msg dope.

`unsigned const` **Sys_ipc_frame::num_reg_words** ( )

Number of words transmitted in registers.

Sys_id_nearest_frame

Binding for the version 2 and version X.0 ID nearest system call.

`L4_uid` **Sys_id_nearest_frame::dest** ( ) `const`

Get the dest parameter of the syscall.

`void` **Sys_id_nearest_frame::type** (`Mword type`)

Set the return type of the syscall.

`void` **Sys_id_nearest_frame::nearest** (`L4_uid id`)

Set the result of the syscall.

Sys_ex_regs_frame

Binding for the version 2 and version X.0 ex-regs system call.

`Mword` **Sys_ex_regs_frame::lthread** ( ) `const`

Get the lthread parameter of the syscall.

`Mword` **Sys_ex_regs_frame::sp** ( ) `const`

Get the stack pointer parameter.

`Mword` **Sys_ex_regs_frame::ip** ( ) `const`

Get the instruction pointer parameter.

`L4_uid` **Sys_ex_regs_frame::preempter** ( ) `const`

Get the preempter ID.

`L4_uid` **Sys_ex_regs_frame::pager** ( ) `const`

Get the pager ID.

`void` **Sys_ex_regs_frame::old_eflags** (`Mword oefl`)

Set the old eflags (x86) or processor status word (other CPUs).

`void` **Sys_ex_regs_frame::old_sp** (`Mword osp`)

Set the old stack pointer.

`void` **Sys_ex_regs_frame::old_ip** (`Mword oip`)

Set the old instruction pointer.

`void` **Sys_ex_regs_frame::old_preempter** (`L4_uid id`)

Set the old preempter ID.

`void` **Sys_ex_regs_frame::old_pager** (`L4_uid id`)

Set the old pager ID.

Sys_thread_switch_frame

Binding for the version 2 and version X.0 thread-switch system call.

`L4_uid` **Sys_thread_switch_frame::dest** ( ) `const`

Get the dest id of the switch.

`Mword` **Sys_thread_switch_frame::has_dest** ( ) `const`

Returns true whether dest is valid.

Sys_thread_schedule_frame

Binding for the version 2 and version X.0 thread-schedule system call.

`L4_sched_param` **Sys_thread_schedule_frame::param** ( ) `const`

Get the scheduling parameters.

`L4_uid` **Sys_thread_schedule_frame::preempter** ( ) `const`

Get the preempter ID.

`L4_uid` **Sys_thread_schedule_frame::dest** ( ) `const`

Get the destination ID.

`void` **Sys_thread_schedule_frame::old_param** (`L4_sched_param op`)

Set the old scheduling params.

`void` **Sys_thread_schedule_frame::time** (`Unsigned64 t`)

Set the consumed time.

`void` **Sys_thread_schedule_frame::old_preempter** (`L4_uid id`)

Set the old preempter.

`void` **Sys_thread_schedule_frame::partner** (`L4_uid id`)

Set the partner of a pending IPC.

Sys_unmap_frame

Binding for the version 2 and version X.0 flexpage-unmap system call.

`L4_fpage` **Sys_unmap_frame::fpage** ( ) `const`

Get the fpage to unmap.

`Mword` **Sys_unmap_frame::map_mask** ( ) `const`

Get the mask, say rights for the unmap.

`bool` **Sys_unmap_frame::downgrade** ( ) `const`

Returns true if the operation is a downgrade.

`bool` **Sys_unmap_frame::self_unmap** ( ) `const`

Returns true if also the current space flushes the fpage.

Sys_task_new_frame

Binding for the version 2 and version X.0 task-new system call.

`Mword` **Sys_task_new_frame::mcp** ( ) `const`

Get the mcp of the new task (if created active).

`L4_uid` **Sys_task_new_frame::new_chief** ( ) `const`

Get the new chief of the task (if created inactive).

`Mword` **Sys_task_new_frame::sp** ( ) `const`

Get the stack pointer of the thread 0 (active).

`Mword` **Sys_task_new_frame::ip** ( ) `const`

Get the instruction pointer of thread 0 (active).

`Mword` **Sys_task_new_frame::has_pager** ( ) `const`

Is a pager specified (if not then create inactive task).

`L4_uid` **Sys_task_new_frame::pager** ( ) `const`

Get the pager id (active).

`L4_uid` **Sys_task_new_frame::dest** ( ) `const`

Get the task id of the new task.

`void` **Sys_task_new_frame::new_taskid** (`L4_uid id`)

Set the new tasks ID.

A.2.7 Cpu Class

Abstraction for kernel internal CPU initialization and features.

`void` **Cpu::init** ( )

Initialize the CPU (e.g., Map ARM interrupt vector table)

A.2.8 Fpu Class

This class handles the state of the floating point unit (FPU). An implementation for machines without an FPU exists, in other cases the following methods need an implementation.

`void` **Fpu::init** ( )

Initialize the FPU.

`void` **Fpu::save_state** (`Fpu_state *s`)

Save the current FPU state into *s*.

`void` **Fpu::restore_state** (`Fpu_state *s`)

Restore the FPU state from *s*.

`void` **Fpu::disable** ( )

Disable the FPU, subsequent use may cause an exception, which can be used to implement lazy FPU handling.

`void` **Fpu::enable** ( )

Enable the FPU.

A.2.9 Timer Class

This class encapsulates access to the system timer. The following hooks must be implemented.

`void` **Timer::init** ( )

Initialize the system timer circuit.

`void` **Timer::acknowledge ( )**

Acknowledge a timer IRQ.

`void` **Timer::enable ( )**

Enable the system timer. Timer IRQs are subsequently generated.

`void` **Timer::disable ( )**

Disable the system timer. (No more timer IRQs.)

A.2.10 Pic Class

This class is the abstraction for the platform's interrupt controller.

`void` **Pic::disable_locked (** `unsigned irq` **)**

Disable (mask) the specified IRQ. (This operation is only used with locally disabled IRQs, thus is not required to care about locking.)

`void` **Pic::enable_locked (** `unsigned irq` **)**

Enable (unmask) the specified IRQ. (This operation is only used with locally disabled IRQs, thus is not required to care about locking.)

`void` **Pic::acknowledge_locked (** `unsigned irq` **)**

Acknowledge the specified IRQ. (This operation is only used with locally disabled IRQs, thus is not required to care about locking.)

`Pic::Status` **Pic::disable_all_save ( )**

Disable (mask) all specified IRQs and return the previous state.

`void` **Pic::restore_all (** `Pic::Status state` **)**

Restore the state of all IRQs according to the given state.

A.2.11 Mapdb Class

The `Mapdb` class represents the mapping database. The entries of the mapping database may have differing layouts and alignment constraints on different architectures, and hence have to be defined per target architecture.

The following structure represents an entry of the mapping database. The declaration of this structure can differ in the sizes and in the order of the members.

```
struct Mapping_entry
{
```

```
    unsigned space:11;
    unsigned size:1;
    unsigned address:20;
    unsigned depth:8;
  };
```

A.2.12 Startup Constructor

A static constructor with the priority `STARTUP_INIT_PRIO` must initialize the mandatory parts of the kernel. These parts are, for example:

- `Boot_info::init()`
- `Config::init()`
- `Kip::init()`
- `Pic::init()`
- `Boot_console::init()`
- ...

A.2.13 Boot_console Class

This class has to handle the in-kernel console drivers during the boot process.

`void` **Boot_console::init** ( )

This method must initialize the console for kernel messages.

A.2.14 kdb_ke Module

Abstraction for explicit kernel-debugger invocation.

`bool` **kdb_ke** (`const char *msg`)

This function must enter the kernel debugger and print out the given message.

Acronyms

ABI application binary interface

API application programming interface

ASID address-space identifier

ASOD aspect-oriented design

CISC complex-instruction-set computing

CPD caching page table

CPSR current program status register

CPU central processing unit

DACR domain access control register

DTLB data-access translation look-aside buffer

EFI extensible firmware interface

ELF executable and linker format

FCSE fast-context-switch extension

FIQ fast-interrupt request

FPU floating point unit

HLL high-level language

IPC inter-process communication

IRQ hardware interrupt

ITLB instruction-prefetch translation look-aside buffer

I/O input/output

JDB Fiasco kernel debugger

LPT leaf page table

MMU memory management unit

OO object oriented

OOD object-oriented design

OOP object-oriented programming

OS operating system

PD page directory

PID process identifier

RAM random-access memory

RISC reduced-instruction-set computing

SASOS single-address-space operating system

SPSR saved program status register

SWI software interrupt

TCB thread control block

TLB translation look-aside buffer

UART universal asynchronous receiver/transmitter

UML unified modeling language

UNSW University of New South Wales

VGA video graphics adapter

Bibliography

[Accetta et al., 1986] Accetta, M. J., Baron, R. V., Bolosky, W., Golub, D. B., Rashid, R. F., Tevanian, A., and Young, M. W. (1986). Mach: A new kernel foundation for unix development. In *USENIX Summer Conference*, pages 93–113, Atlanta, GA.

[Alexandrescu, 2001] Alexandrescu, A. (2001). *Modern C++ Design : Generic Programming and Design Patterns Applied*. Addison-Wesley.

[ARM Ltd., 2000] ARM Ltd. (2000). *ARM Architecture Reference Manual*. ARM Limited.

[ASOD, 2003] ASOD (2003). Aspect-Oriented Development, Home Page. URL: `http://www.asod.net`.

[Bomberger et al., 1992] Bomberger, A. C., Frantz, A. P., Frantz, W. S., Hardy, A. C., Hardy Norman, Landau, C. R., and Shapiro, J. S. (1992). The KeyKOS Nanokernel Architecture. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 95–112. USENIX Association.

[Coady et al., 2001] Coady, Y., Kiczales, G., Feely, M., Hutchinson, N., Suan, J. O., and Gudmudson, S. (2001). Position Summary: Aspect-Oriented System Structure. In *The 8th Workshop on Hot Topics in Operating Systems (HotOS)*.

[FreeBSD, 2003] FreeBSD (2003). FreeBSD Home Page. URL: `http://www.freebsd.org`.

[Group, 1999] Group, T. F. R. (1999). *The OSKit: The Flux Operating System Toolkit*. University of Utah, Department of Computer Science.

[Grützmacher, 1998] Grützmacher, L. (1998). Entwurf und Implementierung einer Mapping-Datenbank für L4.

[Haeberlen, 2003] Haeberlen, A. (2003). Managing Kernel Memory Resources from User Level. Master's thesis, System Architecture Group, University of Karlsruhe.

[Härtig et al., 1997] Härtig, H., Hohmuth, M., Liedtke, J., Schönberg, S., and Wolter, J. (1997). The performance of μ-kernel-based systems. In *16th ACM*

Symposium on Operating System Principles (SOSP), pages 66–77, Saint-Malo, France.

[Hazelnut, 2000] Hazelnut (2000). Hazelnut – performance evaluation. Available from URL: `http://www.l4ka.org/projects/hazelnut/eval.asp`.

[Hohmuth, 2003a] Hohmuth, M. (2003a). The Fiasco kernel: System Architecture. Technical Report ISSN 1430-211X TUD-FI02-06, Dresden University of Technology. Unpublished.

[Hohmuth, 2003b] Hohmuth, M. (2003b). *Preprocess - A preprocessor for C and C++ modules*. http://os.inf.tu-dresden.de/ hohmuth/prj/preprocess/.

[Hohmuth and Härtig, 2001] Hohmuth, M. and Härtig, H. (2001). Pragmatic non-blocking synchronization for real-time systems. In *USENIX Annual Technical Conference*, Boston, MA.

[Intel, 2003] Intel (2003). *Prescott New Instructions Software Developer's Guide*. Intel Corporation.

[Intel PXA, 2002a] Intel PXA (2002a). *Intel PXA250 and PXA210 Application Processors, Developer's Manual*.

[Intel PXA, 2002b] Intel PXA (2002b). *Intel PXA250 and PXA210 Application Processors Operating System Developer's Guide*.

[Intel XScale, 2002] Intel XScale (2002). *Intel XScale Microarchitecture for the PXA250 and PXA210 Application Processors, User's Manual*.

[Kesteloot, 1995] Kesteloot, L. (1995). Porting BSD UNIX to a New Platform. URL:http://www.teamten.com/lawrence/291.paper/291.paper.html.

[L4Ka, 2003] L4Ka (2003). L4Ka Website of the University of Karlsruhe. URL: `http://www.l4ka.org`.

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

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

[Linux, 2003] Linux (2003). Linux-Kernel Home Page. URL: `http://www.kernel.org`.

[McKusick et al., 1996] McKusick, M. K., Bostic, K., Karels, M. J., and Quarterman, J. S. (1996). *The Design and Implementation of the 4.4BSD Operating System*. Addison-Wesley Longman, Inc.

[NetBSD, 2003] NetBSD (2003). NetBSD Home Page. URL: `http://www.netbsd.org`.

[Sartoris Developers Group, 2003] Sartoris Developers Group (2003). Sartoris Project Description. URL: `http://sartoris.sourceforge.net`.

[StrongARM, 2001] StrongARM (2001). *Intel StrongARM SA-1110 Microprocessor Developer's Manual*.

[von Leitner, 2003] von Leitner, F. (2003). diet libc Web Page. URL: `http://www.fefe.de/dietlibc`.

[Warg, 2002] Warg, A. (2002). Porting of Fiasco to IA-64. Dresden University of Technology.

[Wiggins, 1999] Wiggins, A. (1999). The Design and Implementation of the L4 Microkernel on the StrongARM SA-1100.

[Wiggins et al., 2002] Wiggins, A., Chapman, M., Uhlig, V., Sayle, A., and Heiser, G. (2002). The Benefits of Sharing TLB Entries. Technical report, School of Computer Science & Engeneering, UNSW, Australia; University of Karlsruhe, Germany.

[Wiggins and Heiser, 2000] Wiggins, A. and Heiser, G. (2000). Fast Address-Space Switching on the StrongARM SA-1100 Processor. In *Proceedings of the 5th Australian Computer Architecture Conference (ACAC)*, pages 97–104, Canberra, Australia. IEEE CS Press.

[Wilkinson et al., 1995] Wilkinson, T., Murray, K., Russel, S., Heiser, G., and Liedtke, J. (1995). Single address space operating systems. UNSW-CSE-TR 9504, Univ. of New South Wales, School of Computer Science, Sydney, Australia.