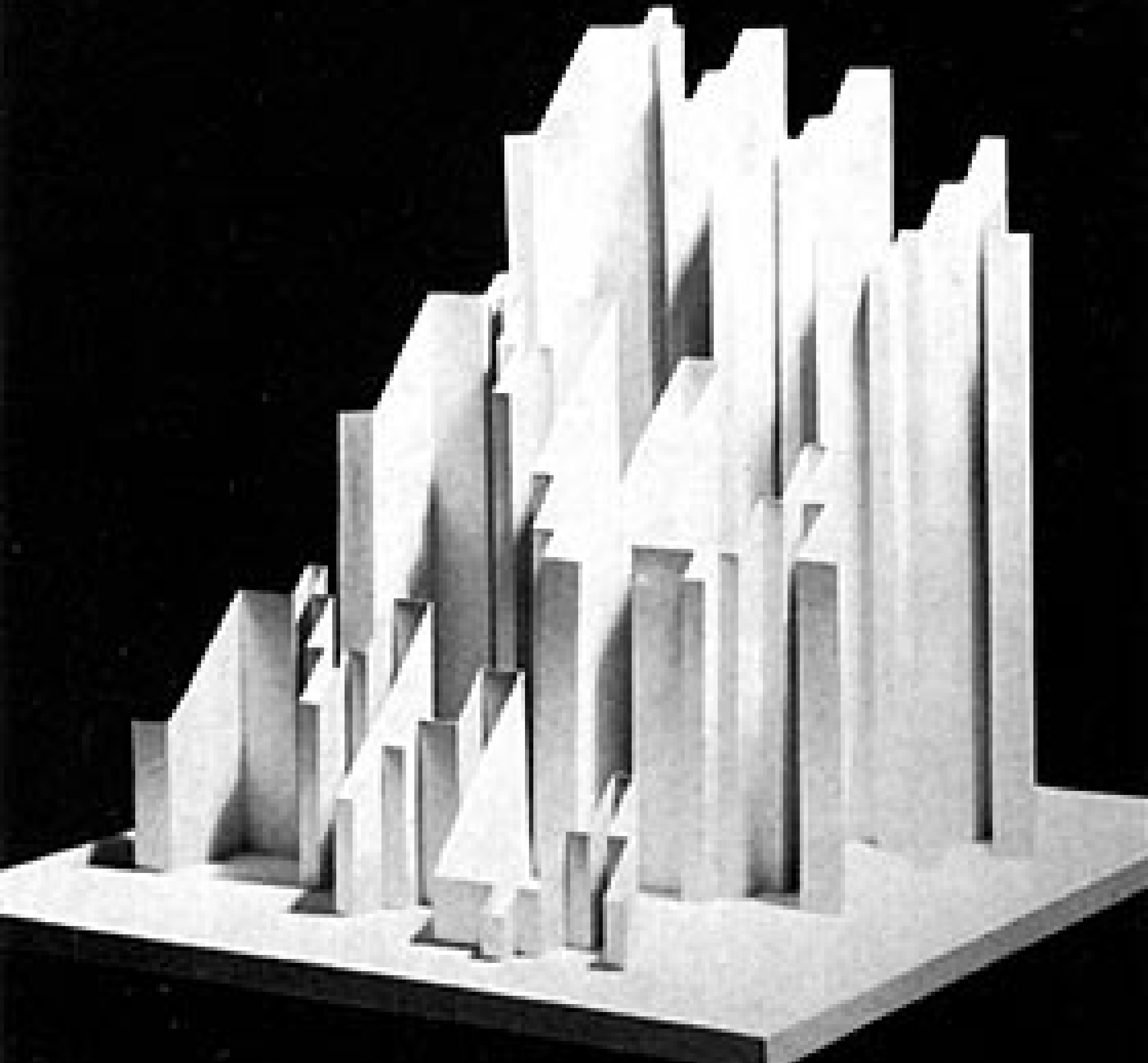
Advancing Operating Systems via Aspect-Oriented Programming



Michael Engel

Philipps-Universität Marburg
Fachbereich Mathematik und Informatik

Advancing Operating Systems via Aspect-Oriented Programming

Dissertation

zur Erlangung des Doktorgrades

der Naturwissenschaften

(Dr. rer. nat.)

vorgelegt von

Michael Engel

aus Zell/Mosel

Marburg/Lahn 2005

vom Fachbereich Mathematik und Informatik
der Philipps-Universität Marburg
als Dissertation am 4. Juli 2005 angenommen.

Erstgutachter: Prof. Dr. Bernd Freisleben

Zweitgutachter: Prof. Dr. Wolfgang Merzenich

Tag der mündlichen Prüfung am 26. Juli 2005

Abstract

Operating system kernels are among the most complex pieces of software in existence today. Maintaining the kernel code and developing new functionality is increasingly complicated, since the amount of required features has risen significantly, leading to side effects that can be introduced inadvertedly by changing a piece of code that belongs to a completely different context.

Software developers try to modularize their code base into separate functional units. Some of the functionality or “concerns” required in a kernel, however, does not fit into the given modularization structure; this code may then be spread over the code base and its implementation tangled with code implementing different concerns. These so-called “crosscutting concerns” are especially difficult to handle since a change in a crosscutting concern implies that all relevant locations spread throughout the code base have to be modified.

Aspect-Oriented Software Development (AOSD) is an approach to handle crosscutting concerns by factoring them out into separate modules. The “advice” code contained in these modules is woven into the original code base according to a pointcut description, a set of interaction points (joinpoints) with the code base.

To be used in operating systems, AOSD requires tool support for the prevalent procedural programming style as well as support for weaving aspects. Many interactions in kernel code are dynamic, so in order to implement non-static behavior and improve performance, a dynamic weaver that deploys and undeploys aspects at system runtime is required.

This thesis presents an extension of the “C” programming language to support AOSD. Based on this, two dynamic weaving toolkits – TOSKANA and TOSKANA-VM – are presented to permit dynamic aspect weaving in the monolithic NetBSD kernel as well as in a virtual-machine and microkernel-based Linux kernel running on top of L4. Based on TOSKANA, applications for this dynamic aspect technology are discussed and evaluated.

The thesis closes with a view on an aspect-oriented kernel structure that maintains coherency and handles crosscutting concerns using dynamic aspects while enhancing development methods through the use of domain-specific programming languages.

Zusammenfassung

Betriebssystem-Kernel zählen zu den komplexesten Software-Systemen, die heutzutage verwendet werden. Die Wartung des Kernel-Codes und die Entwicklung neuartiger Funktionalität wird zunehmend komplizierter, da die Menge an benötigten Eigenschaften eines Kernels stetig steigt. Dies kann zu unabsichtlich in den Code eingebrachten Nebeneffekten führen, wenn ein Teil des Systems geändert wird, der zu einem anderen Kontext gehört.

Software-Entwickler modularisieren ihre Code-Basis in einzelne funktionale Einheiten. Einige Funktionalität – auch “concern” genannt – die im Kernel benötigt wird, gliedert sich nicht sauber in die vorgegebene Modularisierungsstruktur ein; die zugehörigen Codestücke sind dann oft über das gesamte System verteilt und ihre Implementierung ist mit der Implementierung anderer Funktionalität vermischt. Diese sogenannten “crosscutting concerns” (querschneidende Belange) sind besonders schwierig in der Handhabung, da Änderungen an einem solchen crosscutting concern erfordern, dass alle die über die gesamte Codebasis verteilten relevanten Stellen gefunden und modifiziert werden.

Aspekt-orientierte Software-Entwicklung (Aspect Oriented Software Development, AOSD) ist ein Ansatz zur einfacheren Handhabung von crosscutting concerns, der den zugehörigen Code in separate Module herausfaktorisiert. Der in diesen Modulen enthaltene sogenannte ”Advice-Code” wird dann mit der ursprünglichen Code-Basis nach Vorgabe einer sogenannten Pointcut-Beschreibung verwoben, die aus einer Menge an Interaktionspunkten mit dem ursprünglichen Code, den sog. “Join Points” besteht.

Um AOSD im Rahmen von Betriebssystemen zu verwenden, müssen aspektorientierte Werkzeuge zur Unterstützung für den dort vorherrschenden prozeduralen Programmierstil sowie eine unterstützende Infrastruktur für das Weben von Aspekten zur Verfügung stehen. Viele Interaktionen mit dem Kernel sind dynamisch, daher ist ein dynamischer Weber erforderlich, der Aspekte zur Laufzeit in das System einfügen und wieder aus diesem entfernen kann.

Diese Arbeit stellt eine Erweiterung der Programmiersprache “C” vor, die aspektorientierte Programmierung unterstützt. Darauf aufbauend werden zwei Werkzeugsätze für dynamisches Weben – TOSKANA und TOSKANA-VM – vorgestellt, die dynamische Aspekte im monolithischen NetBSD-Kernel bzw. in einem Linux-Kernel realisieren, die auf einer Infrastruktur aufbauen, die aus einem L4-Microkernel und einer unterstützenden virtuellen Maschine besteht. Auf der Basis von TOSKANA werden Anwendungen für diese dynamische Aspekttechnologie vorgestellt und bewertet.

Ein Ausblick auf eine aspekt-orientierte Kernelstruktur, die mit Hilfe von dynamischen Aspekten Kohärenz sicherstellt und crosscutting concerns handhabt und zugleich die Systementwicklung durch den Einsatz domänenspezifischer Programmiersprachen erleichtert, schließt die Arbeit ab.

Contents

List of Figures

1. Introduction

“Felix qui potuit rerum cognoscere”

— Latin Proverb

1.1 The State of Operating Systems Research

Operating systems are at the heart of most electronic systems today. The traditional use of an operating system controlling server installations and workstation computers is only one among the many possible applications. This particular market is usually dominated by Microsoft products, with only a few competitors remaining that mostly base their operating system development on Unix or a Unix-like system such as Linux.

By comparison, a larger number of operating systems are deployed in embedded systems. Their applications range from simple small-scale tasks like capturing sensor data on a low-end eight-bit microcontroller that provides very limited amounts of computational power and memory to complex products like cellular phones, which nowadays include capabilities like the support for multiple wireless network protocols, a virtual execution environment for dynamically downloadable applications and multimedia features like audio and video players, integrated digital cameras and even GPS navigation services. Modern mobile phones offer CPU performance and memory capacities that are comparable to workstations from the mid-nineties.

Embedded applications use a rather large variety of operating systems that range from minimalist solutions like Berkeley’s TinyOS [70] for sensor nodes, which fits into a few kilobytes of memory, up to full-featured multimedia platforms with graphical user interfaces like the Symbian Operating System used on various cellular phones. At the top end of this area, traditional PC and workstation operating systems are being scaled down for embedded use, like Microsoft Windows for Pocket PC and the various embedded Linux variants. The embedded market features short-lived product cycles and cost-driven rapid changes in hardware platforms, making adaptations of the operating system software platform a regular necessity.

“Innovation” is an overused notion in computer-related areas. Often, innovation is not defined by technological achievements but rather by factors that differentiate a product from its competition. Marketing-driven factors are often more important than new concepts, so that current consumer electronics devices end up being a farrago of features crammed into a tiny amount of space at lowest possible cost. This often not only has a negative impact on hardware quality, but also the software functionality required to operate such a device is astonishingly complex, resulting in software quality problems which often make the end user a beta tester just to have a product on the market before the competition.

Real *academic innovation* in the operating system area, however, is usually neither emerging from commercial offerings nor from popular open-source systems. In the commercial area, Microsoft is often cited for just copying ideas from existing systems, whereas the aim of the more popular open-source systems (like Linux) is more or less to re-invent the wheel by replicating a system according to well-tested Unix/POSIX specifications.

In his often-cited polemic from 2000, Rob Pike – one of the original developers of Unix [112] and developer of the Plan 9 [107] and Inferno [42] operating systems at Bell Labs – declared systems software research irrelevant [106]. Even though Pike deliberately paints a pessimistic picture of the situation, various points he brings up are worth considering. One of his observations is that in recent years, most of the research done in the area of operating systems revolves around phenomenology – i.e., too much measurement is going on instead of innovation in concepts. While this produces a vast amount of research papers, the changes in system implementation that are derived are mostly on a microscopic scale while losing control of the large picture.

One of the central points of his essay is that operating systems research and development should be a combination of engineering, design and art – but the *art* of building systems is mostly gone and replaced by mimicking or evaluating well-tested approaches. The community of system developers must therefore accept and explore unorthodox ideas. This is, however, a two-edged sword – creating a new operating system from scratch is a huge endeavor, whereas integrating novel concepts into one of the mainstream open source systems is often hindered by extremely conservative approaches to far-reaching changes in these projects.

A central conclusion Pike draws is that research in operating systems and market capitalization should be separated. This, however, is what partially lead to the current situation – whereas new impulses for operating systems in the late 1980s and early 1990s came from academia and were rapidly subsumed into commercial products (think of the BSD Unix development and the Mach project, which showed up early in commercial products like SunOS, NeXTstep and OSF/1), nowadays companies fund most open source development efforts, leaving only a narrow space for developments outside the corporation-controlled pathways.

To be successful outside of its own realm, academic research in operating system topics has to take the lead in ideas and technology again – but the lead has to be large enough and the ideas sufficiently compelling for industrial and open-source projects to pick them up. The problem behind this is that the development of operating system code today is much more complex than it was in the 1980s and thus hard to handle in a typical university research group. Here, results from software engineering research are required in order to reduce that complexity. The development method used in open-source system software projects today is mostly unchanged from the humble beginnings of Unix more than thirty years ago. These rather primitive toolsets consisting of a C compiler, linker and a make utility used for developing kernel code will not be replaced in their entirety (since low-level interaction with the hardware is a necessity in a kernel), but rather improved and augmented with software engineering technologies like component-based systems, aspect-orientation, refactoring and domain-specific languages in order to make handling this complex software system easier for the developer. If these methods gain broad acceptance in the development process, one might again expect more novel ideas in systems to be realized by academic research groups.

Remarkably, Rob Pike has left Bell Labs in 2002 to work for Google, which is rumored to work on its own large-scale distributed operating system.

1.2 Current Trends

Despite Pike's rather pessimistic view of the research situation in operating systems, there is OS research going on in academia as well as in the industry. Most of the research is driven by the fact that the requirements for functionality implemented in microprocessor-based systems ranging from tiny embedded applications like nodes in sensor networks to large-scale supercomputers [3] and mainframes [99] are constantly rising. In turn, this demands sunstantially more functionality as well as flexibility from the system software controlling these devices. An operating system today has to be adaptable to new environmental challenges, thereby taking great care not to break existing applications. As a consequence, flexibility and adaptability (while ensuring compatibility) are two major challenges for operating system developers today.

At the low end of the performance spectrum, a typical embedded system some years ago usually communicated using a simple protocol over slow serial lines. Today's embedded systems are often required to provide TCP/IP networking, wireless connectivity and remote administration at the lowest possible cost. Inexpensive 32-bit microcontrollers now make it possible to provide a full-featured Unix-like system in a disposable device[1].

In the midrange, server systems are thriving for increased availability and ease of setup and administration while providing high file system and network throughput. The typical personal computer today has evolved from a sophisticated typewriter replacement into an always-connected multimedia and communication machine providing large amounts of computing power and disk space. The multimedia requirements bring about that the system software has to implement some sort of soft-realtime behavior in order to provide for synchronized media rendition, while on the other side being always connected to a public network demands improved security measures from the system to ensure privacy of the user and integrity of software and data on the PC.

Notebook computers gained most of the features of their desktop counterparts, but show additional constraints. Whereas desktop systems usually neglect the power usage of the system, running on battery required power-saving mechanisms implemented in hard- and software while providing an environment compatible to the operating systems and applications used on the larger desktop systems. Being able to integrate into changing and

[1]After all, these embedded CPUs provide several times the computing power and memory than a typical PDP-11 Unix was developed on.

unknown network environments using wireless technologies, an operating system has to provide methods to cope with this nomadic network connection style.

At the top end of the spectrum, mainframe users nowadays are not contempt with using a complex proprietary operating system; these users readily expect the development and execution environment (which is usually Unix-based) to be available on their mainframe in order to reuse existing code as well as to make use of the vast amount of open-source software available for Unix-like systems. Thus, existing systems are being adapted to run on these systems which are architecturally quite different from normal PCs and workstations. A premier example for this is Linux on IBM's zSeries mainframes [99].

A contemporary operating system is often scalable over this broad range of hardware environments and has to provide solutions for the specific problems of all levels. In the following sections, some of these challenges for operating system developers – which ultimately lead to the requirements for more efficient programming methodologies to handle some of these problems, which this thesis is about – are exemplified.

1.2.1 Configurability

A typical system from ten years ago did not have to care much about configurability, since the hardware choices as well as the connectivity options that had to be supported by a typical desktop or server OS were rather limited. Today, there is an abundance of external as well as internal add-on components for a computer system, which are often "hot-pluggable", i.e., installable and removable at runtime.

Another level of configurability comes into play when scalability is required. For example, in-kernel data structures like the process table or file allocation structures for which it is sufficient to be statically allocated for a single scenario are either over-dimensioned for single-user systems or too limited and inefficient for use on systems with thousands of simultaneous users.

1.2.2 Ubiquituos Networking

The number of connectivity options has also increased considerably. A typical computer from the early nineties usually was able to communicate with other systems using a modem attached to a serial line or a 10 MBit Ethernet connection. Today's computer systems, however, feature a vast amount of connectivity options built-in – Ethernet with speeds ranging from 10 MBit/s to 1 GBit/s, wireless 802.11 networking, Bluetooth, Infrared (IrDA), to name just the most common options.

Related to the increase of connectivity options, the number of supported network protocols increased. Whereas computers from the early nineties typically supported either IPv4 or NetBIOS protocols with a limited range of application-level protocols, today often ten or more protocols have to be implemented together, ranging from IPv4 and IPv6 over GSM, GPRS and UMTS cellular protocols and 802.11x as well as Bluetooth short-range wireless protocols to infrared support (IrDA) and several specific embedded protocols like the CAN-bus protocol used in automotive systems.

On the application level, applications like video conferencing, voice-over-IP, and file sharing pose new requirements for the networking stack like support for quality-of-service, high throughput and efficient operation when connected to a large number of peers.

Being connected to a worldwide network of computers does not only bring advantages. Security of computer systems is one of the most important topics today. With the current amount of viruses and worms spreading over the Internet, ensuring the integrity of data on the system and protecting the privacy of its users is one of the most important concerns for OS developers nowadays.

1.2.3 Virtualization

Virtualization itself is no new invention. Today, standard PCs are sufficiently powerful to support virtualization, a technology that creates the illusion of separate, protected execution enviroments supporting distinct operating system instances or application domains on a single machine.

While not being a novel technology by far, virtualization finds new applications in networked environments. While virtual machines (VMs) were used on mainframes to provides multiple users with private instances of their respective single-user operating systems, VMs today allow the use of an operating system inside another (typically of a different OS) in order to run legacy applications, create virtual server environments to consolidate hundreds or thousands of web server instances on one large machine at an ISP's site or generate a secure execution environment for untrusted code, e.g. in the context of ad-hoc grid software applications [58].

Virtual machines were originally set up by experienced administrators. The challenge of deploying VMs into personal computers is to create a lightweight, easily administrable system that is able to support a wide range of hardware and system software configurations while providing the greatest amount of security for the user by employing appropriate isolation mechanisms.

1.2.4 Autonomic Computing

Autonomic Computing [81] is a new challenge for software development on all levels – from system software to middleware and applications. The components of a system are expected to have the ability to control and organize themselves to meet unforeseen changes in the hard- and software environment. The basic principles employed to increase availability, reliability and performance of a computer system are self-configuration, self-healing, self-optimization, and self-protection.

Installing, configuring, and integrating large, complex systems is challenging, time-consuming, and error-prone even for experienced administrators. Autonomic systems will perform *self-configuration* automatically according to directives that set the overall aims for system operation. Newly added components will integrate themselves seamlessly into the system, while the rest of the system will adapt to their presence.

Identifying, tracing and determining the cause of problems in a complex computing system often takes human experts days or even weeks to fix. Still worse, sometimes problems mysteriously disappear without any trace. An autonomic computing system detects, diagnoses, and repairs localized problems that result from bugs or failures in either hard- or software. When knowledge about the configuration of the system is available, a component doing problem analysis is able to gather information from log files and additional monitoring facilities. The system would then try to apply *self-healing* by matching the diagnosis result against available software patches and apply the patch, if possible on the fly without restarting the system.

Complex systems often have hundreds of tunable parameters that must be adjusted for the system to perform optimally. Through *self-optimization*, autonomic systems continually seek ways to improve their operation, thereby trying to achieve better performance or cost-efficiency.

Finally, *self-protection* can have two different meanings. Autonomic systems will defend the system as a whole against large-scale problems that arise from malicious attacks or related failures that self-healing measures cannot correct. Furthermore, the system will be able to anticipate problems based on readings of sensors and take measures to correct or avoid these problems.

Implementing autonomic behavior in systems means to deal with topics such as quality of service, energy consumption, fault tolerance, and security. These in turn require changes to the operating system since control over the hardware is required to gain access to the parameters defining the corresponding behavior of the particular components.

The various "self" technologies require adaptation of the kernel to dynamically changing requirements. This level of adaptability enables the operating system to cope with problems such as the failure of a system component, a denial of service attack coming in over the network, adaptation to different load requirements, or the addition of new peripherals to the system.

1.3 Structure of the Thesis

Aspect-oriented software development (AOSD) is a programming paradigm that facilitates designing code for complex situations in which functionality related to more than one problem domain has to be implemented simultaneously by creating a new level of modularization for these so-called *crosscutting* problems.

The changed demands on operating systems create new challenges for kernel developers. Implementing the new requirements involves the modification of a large portion of code scattered throughout the system, since most of these concerns are either crossing layers inside the operating system kernel or have to be implemented in a set of components that realize similar functionality (e.g. diverse file system implementations or various supported binary executable formats) simultaneously. Consequently, an aspect-oriented approach is the obvious way to handle the complex interactions required to implement novel solutions.

Being envisioned by Gregor Kiczales in 1997 [82], AOSD is beginning to find widespread acceptance among object-oriented software developers. In mostly procedural programming environments, AOSD is still relatively unexplored. Its broad usability, however, makes it useful in many common situations in software development – especially for complex systems like an operating system kernel.

This thesis shows how aspect-oriented software development can be applied to procedural systems by defining the specific problem domain and proposing a “C” language extension to support AOSD. Based on this specification, two toolkits for using AOP in kernel space were developed, which are showcased by several applications.

AOSD and language technologies, e.g., domain-specific languages (DSLs), can be used to handle complex problems that usually use up valuable development time. Since most of the AOSD development nowadays takes place in object-oriented systems in user space, AOSD in kernel space poses a specific set of requirements that stem from the mostly procedural structure of existing kernel source code as well as the asynchronous, event-oriented nature of a large part of the code in a typical kernel.

Combining two areas of computer science like operating systems and software engineering to create a form of synergy between the two requires quite a bit of background in both areas to be explained. Both topics are of practical nature, so the implementation and application of aspect-oriented technologies in an operating system context using real-world software systems plays a central role in this thesis. Consequently, descriptions of the implementation of the various software components used to explore AOP in an operating system context and of applications for typical kernel mode tasks based on these components take up a large amount of space.

Chapter 2 characterizes the context in which aspect-oriented software development takes place in this thesis by describing the relevant structural aspects of operating system kernels with a special focus on BSD-derived systems like NetBSD.

Chapter 3 describes problems that show up when designing and modifying software. Starting from the assumption that operating system kernels are among the largest and most complex pieces of software in existence today, observations on the modularity structure of a kernel are presented that lead to crosscutting concerns in the implementation.

Chapter 4 shows methods to solve the problem of implementing crosscutting problems in kernels in a modular way using AOSD. Here, special emphasis is put on aspects for procedural languages, which are in common use in kernel code.

Chapter 5, then, concentrates on AspectC, an extension of the C programming language to support AOP. Since no usable implementation of AspectC existed so far, a prototypical compiler was implemented using code-transformation technology. Despite its prototypical nature, the compiler is designed from ground up to support AOP in large, real-world applications like an OS kernel.

Chapter 6 gives an overview of the various possible approaches to deploy aspects dynamically into a running software system. Ideas and mechanisms for various approches, from the level of reconfigurable hardware up to virtual machine environments, are discussed and evaluated.

Chapters 7 and 8, describe the toolkits developed in order to use dynamic AOP in kernel mode and extensions to the software generation toolchain (compilers, linkers etc.) in order to provide enhanced AOP functionality. TOSKANA is the first approach to in-kernel dynamic AOP, implemented for the NetBSD kernel. Based on experience with TOSKANA, the virtual-machine based TOSKANA-VM system takes a different approach to the deployment of dynamic aspects to overcome some of TOSKANA's initial limitations.

Chapter 9 describes kernel-mode applications of AOP ranging from cross-layer networking problems to security for Grid environments by creating "sandboxes" for non-trustworthy processes

Chapter 10 discusses related work.

Chapter 11, finally, summarizes and concludes the thesis. The vision of an operating system that extensively uses AOP technology is given as an outlook into future research in this area.

1.4 Acknowledgements

Foremost, doing research for a doctoral thesis not only requires sufficient funding, but also an environment in which it is possible to try out novel ideas and possibly change the direction of research if one reaches a dead end. This is especially important when working on a topic with no immediate industrial results. Thus, the first big "thank you" goes to my thesis advisor, Bernd Freisleben, for letting me work on my favorite topic "operating systems" and providing incentives to take a look over the rim of the tea cup in order to avoid the narrow-mindedness one develops when being deeply immersed with a specific topic for a long time.

Working on research topics is not possible without a solid foundation of knowledge. As a crossover of software engineering, programming languages and operating systems, this thesis required familiarity with all these topics. Here, my gratitude goes to Wolfgang Merzenich, who provided me with excellent lectures on compiler technology and the theoretical foundations of computer science as well as having an open ear for my research progress and problems that showed up.

Research today no longer takes place in the proverbial ivory tower – we are in the lucky situation to have rapid worldwide communication media available at our fingertips. So, a special thank you shall go to all the numerous friends, colleagues and people on the net with whom I had discussions on thesis-related (and other...) topics. I hope I wasn't too much a pain in the neck with my focussing on the OS topic. Here, a special "Thanks!" goes to the Software Technology group at Darmstadt University of Technology, especially Mira Mezini and Michael Haupt, for providing much valuable information and discussion on the topic of aspect-oriented programming.

Perhaps a funny coincidence, there are two more persons named Bernd and Wolfgang involved. Wolfgang Schmidt and Bernd Gläser, two of my teachers in gymnasium, not only imparted the basic required knowledge in Mathematics and Physics to take up CS studies, but also were responsible for my fascination with computer science by providing information and interesting projects far beyond the requirements of the regular curriculum. I wished today's students could get such a thorough introduction to the topic and see where

the fun in it is.

Doing research and writing a thesis without the proper environment is impossible – a special “thanks!” goes to the professors, my fellow Ph.D. students as well as the technical and administrative staff of the department of Mathematics and Computer Science at Marburg University.

A final “thank you” goes to all Unix developers – the original team at Bell Labs as well as the thousands of programmers, researchers and open-source developers worldwide. You managed to make a highly interesting topic in computer science available to the general public by providing the source code to your work freely instead of keeping your secrets in a black box. Being able to take a look at your work enabled many students to turn a boring theoretical topic into a rewarding hands-on experience. Without your contributions, this thesis would not exist.

2. NetBSD System Structure

"Make everything as simple as possible –
but not any simpler!"

— Albert Einstein

2.1 Introduction

In its more than 35 years of history, Unix-based operating systems have influenced design decisions for a lot of other operating systems. However, Unix itself is a moving target, which adapts well to new requirements. In this process of adapation, Unix acquired new technology like TCP/IP networking, support for various file systems and dynamically loadable kernel components. This chapter gives an overview of the structure of the NetBSD kernel, which was derived from the line of Unix operating systems developed at Berekely University. Special emphasis is placed on those parts of the kernel that are relevant in the context of aspect-oriented development.

2.2 Unix Kernel Structure

NetBSD is one of the open source software projects that originated from the versions of Unix developed at the University of California at Berkeley beginning in the year 1978, culminating in the release of 4.4BSD in 1994. Starting in 1991, a group of researchers adapted the 4.3 version of BSD Unix to run on industry-standard x86-based computer systems. Based on this development, the three open source projects FreeBSD, NetBSD and OpenBSD were created, each with different goals, but all based on the final 4.4BSD codebase from Berkeley.[1]

Traditionally, the kernel of an operating system derived from Unix [112] handles tasks like process and memory management, device control, file system handling and network communication. Each of these tasks is further subdivided into several variants; e.g., a typical kernel supports multiple different file systems, various network protocols, a large array of devices and bus systems and classes of processes with different requirements ranging from realtime response to efficient batch processing.

[1]This is an extremely condensed version of the developments around BSD Unix. For detailed information, see [115], [94] and [39].

This complex and variable set of tasks is reflected in the source code structure of a kernel, exemplified by the current version 2.0 of the BSD-derived [88] NetBSD kernel in figs. 2.1 and 2.2. Here, functional subsystems of the kernel are contained in speparate source code directories under the `/usr/src/sys/` top-level source code hierarchy.

The most important subdirectories in the source tree are:

arch	This directory contains all architecture-dependent code of the system. NetBSD currently supports 55 different hardware architectures based on 15 different CPU families, ranging from embedded use to workstation and server deployment. Unsurprisingly, the arch directory is where most of the NetBSD source code is situated.
compat	NetBSD provides binary compatibility with a lot of legacy and current platforms running on one of the supported CPUs. This directory contains wrappers and emulation code for various platforms.
dev	Another large part of the kernel code is made up of device drivers and support routines for defining and controlling bus structures, which are all located in the dev subdirectory
***fs**	NetBSD supports a large range of filesystems for physical storage media (like ufs and FAT), network access (NFS, smbfs) and virtual file system representations (unionfs, ptyfs). This code is distributed over various directories containing “fs” in their name. On the top level of the kernel source tree. More complex implementations use their own subdirectory for historical reasons, while less complex file systems are collected in the fs and miscfs directories.
kern	This directory contains the core kernel code – system calls, device switch structures, VFS and vnode code, scheduling, executable format support and IPC support.
lib	The lib directory contains libraries of common code for in-kernel use like compression libraries and supporting libraries for standalone (boot) code.
lkm	Like most modern Unix-like systems, NetBSD supports the extension of the kernel through the use of modules that can be loaded dynamically at runtime. The lkm directory contains support code for module handling as well as example and production code for various kernel modules.
net*	True to its name, NetBSD supports a plethora of networking protocols. For historical reasons, each network protocol family that is supported resides in its own directory in the root of the kernel source tree. Support for IP V4 and V6, IEEE 802.11, Appletalk, ISO networking protocols and many lesser-used protocols is included here.
stand	This directory contains standalone code that is used for boot loaders and the kernel debugger.

uvm NetBSD abandoned the traditional BSD virtual memory system, which was historically very VAX-centric, and replaced it with a Mach-inspired new VM engine called UVM [37]. The uvm directory contains all machine-independent code for virtual memory management.

This main modularization structure of the kernel code source tree is further subdivided in some of the directories, e.g., each architecture supported is contained in a separate directory inside `arch` and several small file system implementations are contained in the `fs` and `miscfs` directories.

2.3 System Evolution

The NetBSD kernel is written almost completely in ANSI C, only about 9% of the code is written in assembler – of which in turn 90% are machine-dependent parts. Fig. 2.3 summarizes the distribution of functionality in a snapshot of the current development version.

Consisting of about 6,000 source files containing nearly 3 million lines of C and assembler source code (including comments and empty lines), the NetBSD kernel is a rather complex system.

Many of the concepts behind NetBSD, like the representation of all objects in the system as files and the handling of all data in the system as unstructured byte streams, date back to the very first Unix implementations of the early 1970's – some concepts, like implementing the shell as a separate user-mode process, stem further back to the original Unix authors' experience with the MULTICS [102] and MIT Compatible Time-Sharing System (CTSS) [36] operating systems.

Wherever necessary, however, Unix adopted new methods and structures throughout its now 35 years of existence. These additions and modifications mostly fitted well into the Unix model, but often represented extensive changes throughout the system's source code. Today, Unix itself is no homogeneous piece of software, currently there are many operating systems that are either derived from the original Ball Labs Unix source code or reimplemented according to published documentation and interface standards. The evolution and cross-dependencies between various systems conforming to Unix definitions is shown in fig. 2.6, adapted from its original version at [135]. These cross-dependencies result in code reused or adapted from a different source code structure, so that modifications coming from a different source branch regularly end up being scattered throughout the code and tangled with code that implements different functionality.

NetBSD itself is derived from the line of Unix systems developed at Berkeley University. After the Computer Systems Research Group (CSRG) in Berkeley was disbanded, several projects set out to create free, open-source versions of BSD Unix. The FreeBSD project started off with 386BSD, an early port of 4.4BSD to Intel-based PCs [80] by Bill Jolitz, with the aim of optimization for this architecture. The NetBSD project chose another direction

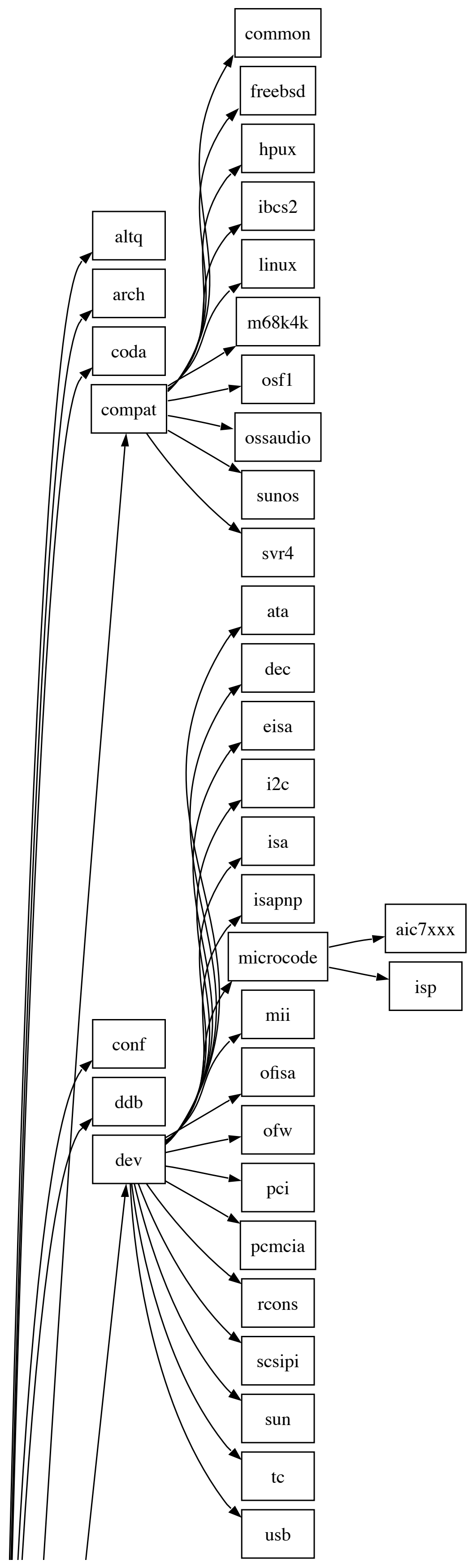


Figure 2.1: Structure of the NetBSD kernel source tree (part 1)

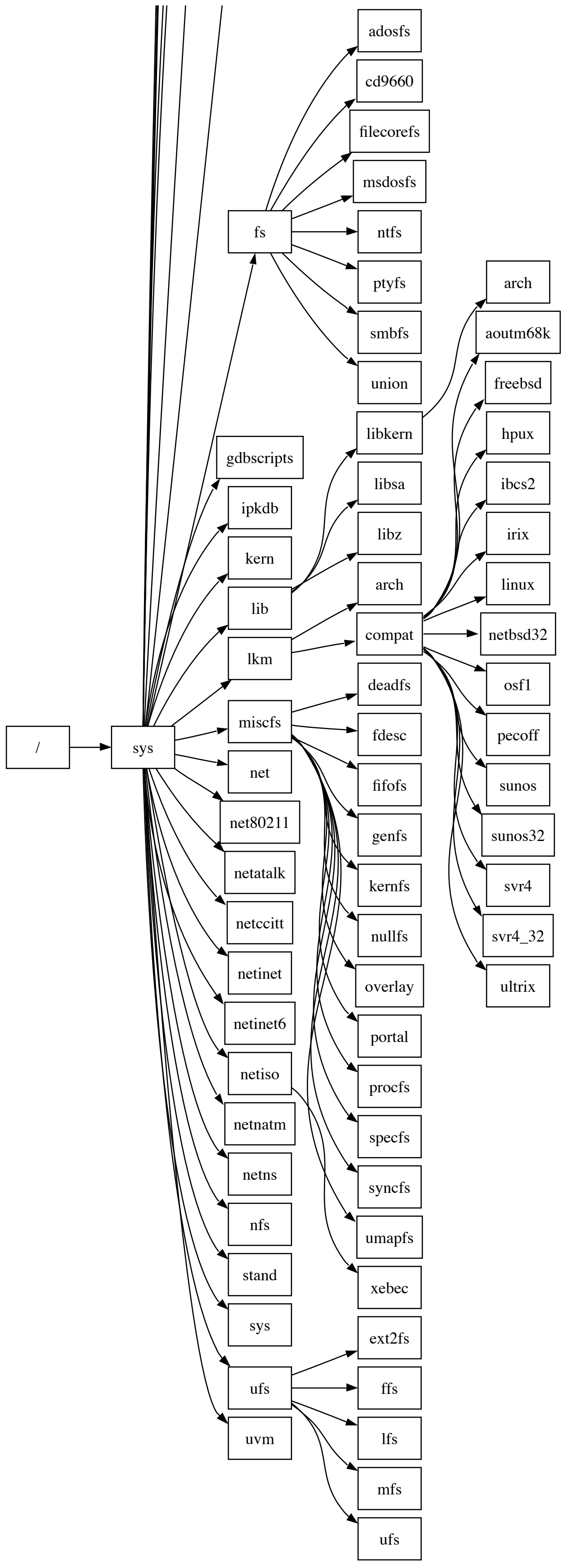


Figure 2.2: Structure of the NetBSD kernel source tree (part 2)

Part of NetBSD	Nr. of C files	Lines of C code	Asm. files	Lines of Assem.
Basic kernel code	98	87229	0	0
Device Drivers	1157	838157	0	0
File System Code	133	105408	0	0
Networking	268	239051	0	0
Libraries	154	25718	205	19163
Arch. independent	2278	1490932	212	24268
Arch. dependent	3039	1215513	412	217082
Complete system	5317	2706445	624	241350

Figure 2.3: Statistics of NetBSD 2.0 kernel code structure

and focussed on portability, making NetBSD the probably most ported operating system. Later in the development, OpenBSD split from NetBSD in order to concentrate on system security issues. A recent development is DragonFlyBSD. Forked from FreeBSD 4, DragonFlyBSD concentrates on integrating advanced concepts into the BSD kernel base like virtualization, microkernel- structures and process checkpointing.

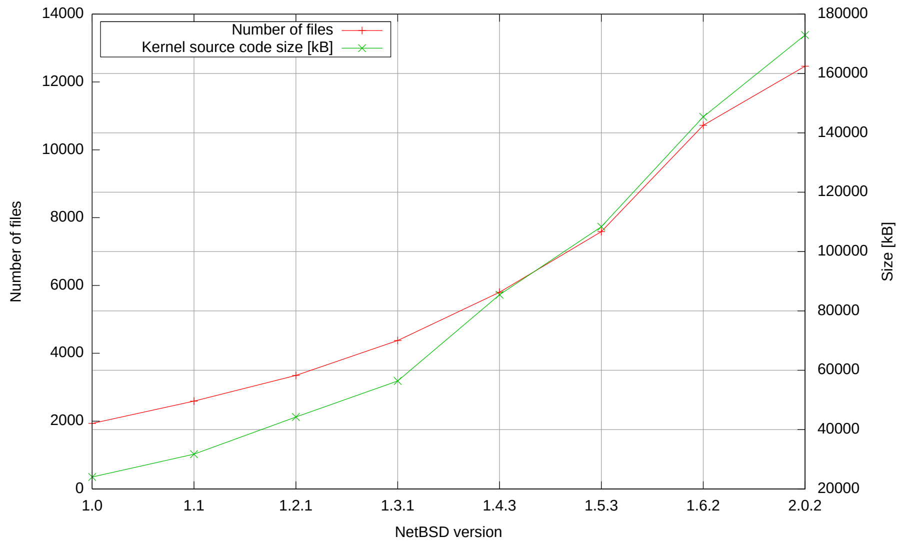


Figure 2.4: NetBSD lines of code

Even today, after more than ten years of separate development, code is exchanged between all of the BSD-derived systems (and related systems, like Apple's Darwin kernel foundation for MacOS X) on a regular basis.

The evolution of various open-source software projects has been analyzed in [9] and [62]. However, no analysis for NetBSD has been performed so far.

NetBSD releases were historically numbered $v.x.y$, with a change in v indicating system redesigns, a change in x stands for a major version change and a change in y is a subversion containing bug-fixes and minor improvements. In the following, each of the final releases

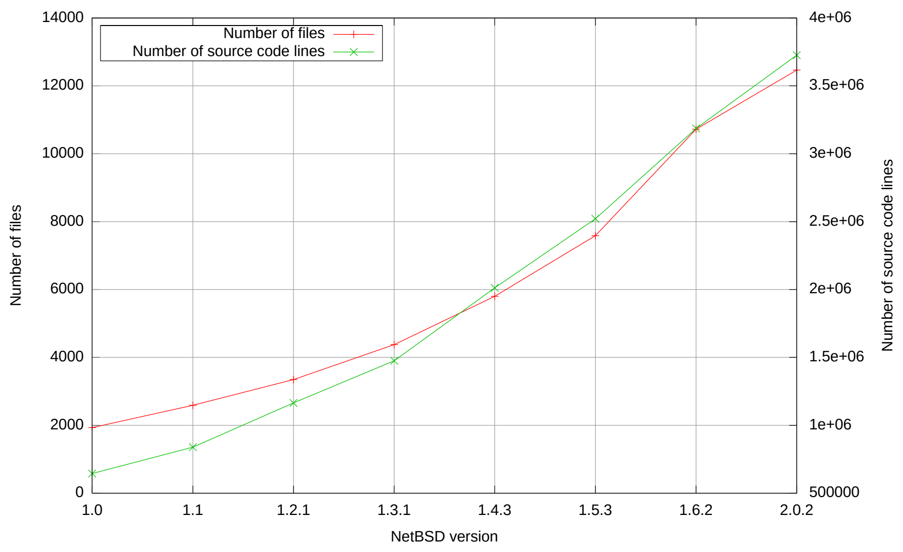


Figure 2.5: NetBSD number of files

of a NetBSD major version, beginning with 1.0 and ending with the current 2.0.2 are analyzed. Recently, the NetBSD numbering scheme has been modified, so that changes in major versions, which were formerly indicated by a change in the second digit, are now denoted by a change in the first digit – so, the next NetBSD release will be called 3.0. This is a purely marketing-driven change as other project have since evolved to much larger numbers: FreeBSD is at 5.4, OpenBSD at 3.8 and the Linux kernel at 2.6.

Of interest for the application of software technology tools is the overall code complexity. The complexity measures compares the number of source files with the kernel source code size in kBytes, shown in fig. 2.4, the number of files compared to the number of source code lines (C source, header files and assembly) shown in fig. 2.5 as well as the increase in the number of architectures and the number of of functional units, denoted by the number of directories in the NetBSD kernel tree containing source code files. This is shown in fig. 2.7.

Comparing these graphs, one sees exponential growth in the number of source code files, lines of code and even in the number of supported architectures. The number of NetBSD developers, however, has not increased linearly. No exact historical numbers could be found. The current number of NetBSD developers is 287, but this includes all system developers, not only persons working on the kernel.

As a consequence, the time between final stable releases of the system is steadily increasing, as shown in fig. 2.8[2]. This could be an indication that the complexity of the system, especially the crosscutting concerns that have to be managed, require new approaches to software development.

[2]2.0.2 is not yet considered to be the final 2.0 release

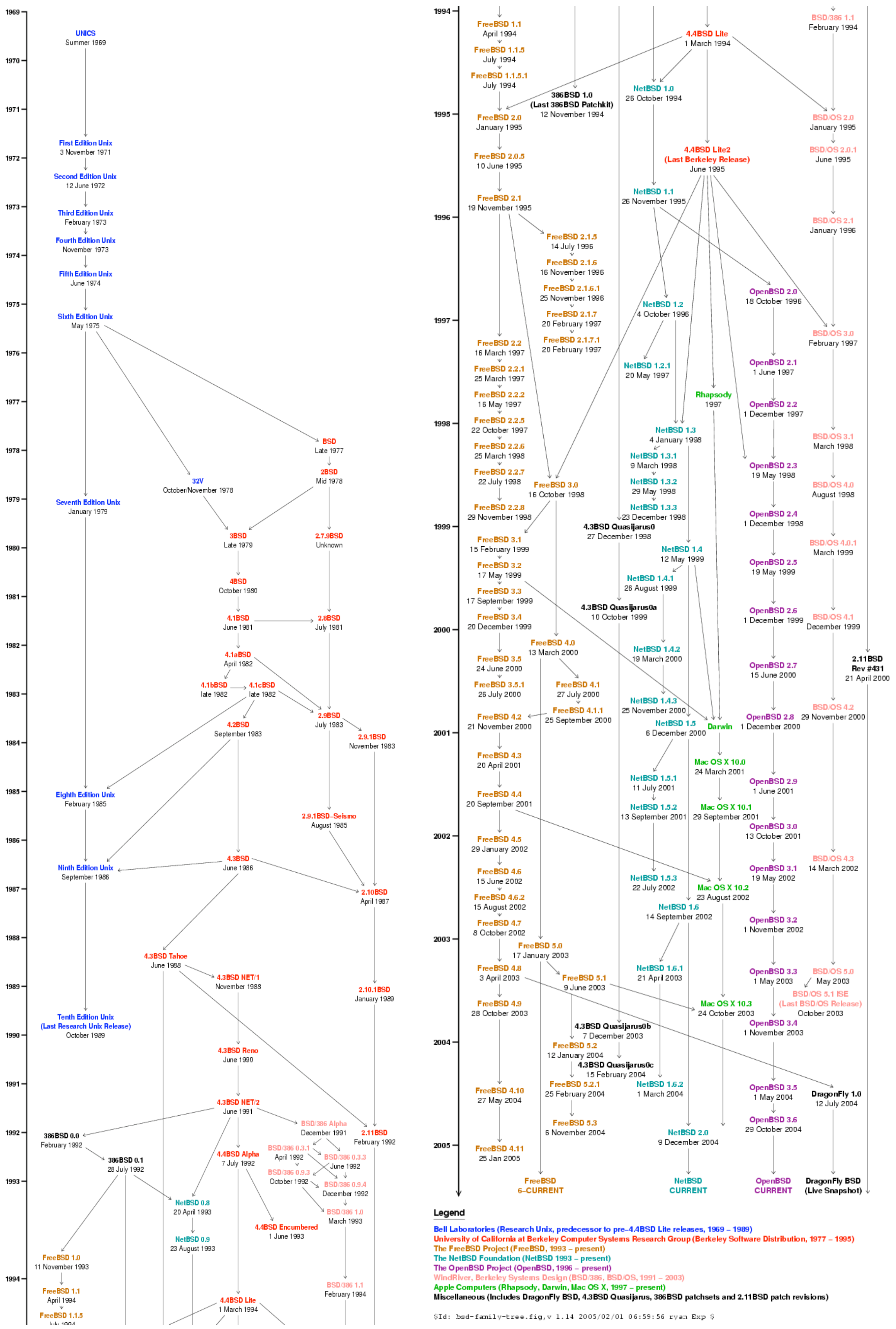


Figure 2.6: Unix evolution

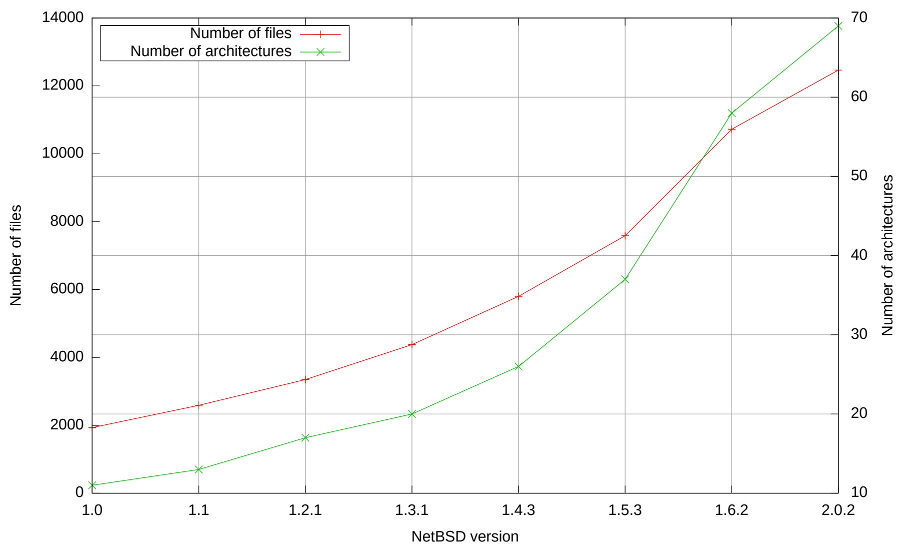


Figure 2.7: NetBSD number of architectures

2.4 Layers

Several subsystems of the kernel provide a functionality that is well suited to be further subdivided into smaller layers, each implementing a strictly defined set of functions relying on services from the lower levels and providing functionality to upper layers using well-defined APIs.

Since issues concerning functionality that spans several layers are one of the important sources for cross-cutting concerns, the two largest layered structures inside an operating system kernel – the network protocol stack and the virtual file system – are discussed below.

2.4.1 Layered Networking Structure

One well-known example for layered software design is the ISO/OSI network protocol stack [77] depicted in fig. 2.9, which defines a seven-layer model for network communication. This model comprises all functionality required by network code implementations, ranging from definitions of physical properties of the transmission medium up to application-specific protocols.

The layered model has the advantage of defining logical communication channels between two layers on each communication endpoint. Real data exchange between hosts, however, does only occur on the physical layer. In order to send and receive information, the software components in each layer use well-defined interfaces to the next upper and lower layers, through which the real information flow takes place, as shown in fig. 2.10.

Real-world implementations of network protocols, however, mostly employ a more simple method of organization[3]. Functionality equivalent to the three upper layers of the ISO

[3]The ISO networking protocols have been implemented (cf. the `netiso` source directory), but have never

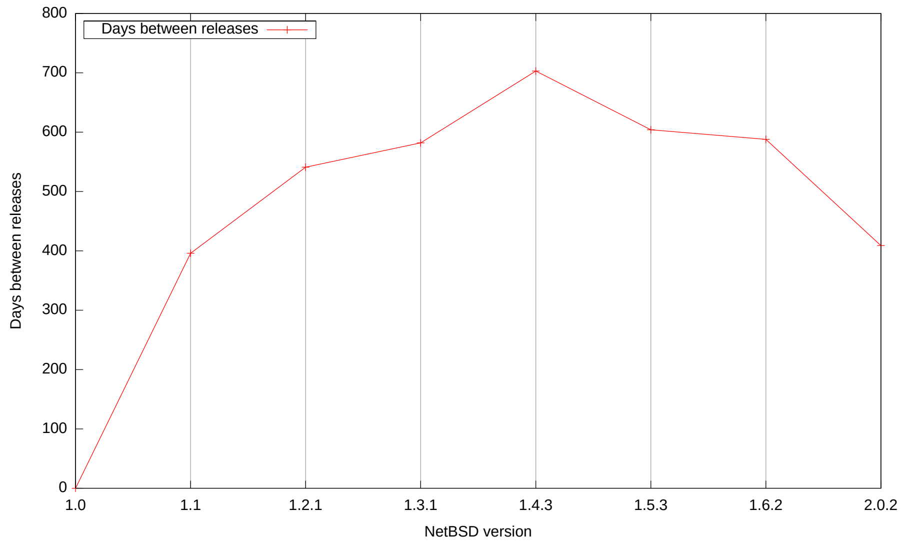


Figure 2.8: Number of days between NetBSD releases of final versions

model (application, and, if implemented, presentation and session layer) are usually relegated to user mode applications like e.g. a web server or ssh client[4], leaving layers from pyhsical to transport to be implemented by the hardware and corresponding kernel drivers and the networking stack.

The NetBSD TCP/IP network stack, shown in fig. 2.11 (taken from `http://www.micro-controller.ru/arc/rtcs.htm`), follows the layered model without strict adherence to the OSI model. On the hardware level, device drivers for Ethernet, serial lines, WLAN controllers and so on take care of the physical layer and part of the data link layer. Addressing at the data link layer is handled by the ARP module, while data encapsulation, authentication and service functions of the data link layer reside in various modules like PAP, CHAP and IPCP.

The network layer is handled by IP, while the transport layer is represented in TCP, UDP and related protocols, while higher-level services on ISO layers 5 and above are mostly handled by user-space applications. In addition to the TCP/IP protocol stack shown here, NetBSD supports a large number of additional network protocols on all layers. Describing these would exceed the realm of this thesis.

The network subsystem provides a general-purpose framework within which network services are implemented. These facilities include a common interface to the socket level that allows the development of network-independent application software, a consistent interface to the hardware devices used to transmit and receive data, network-independent support for message routing, and memory management.

The network subsystem is logically divided into three layers. These three layers manage three tasks: interprocess data transport, internetwork addressing and message rout-

been in common use

[4]However, there are examples to this rule. In Linux, there exists an implementation of an in-kernel web server that is orders of magnitude faster than user-mode web server applications.

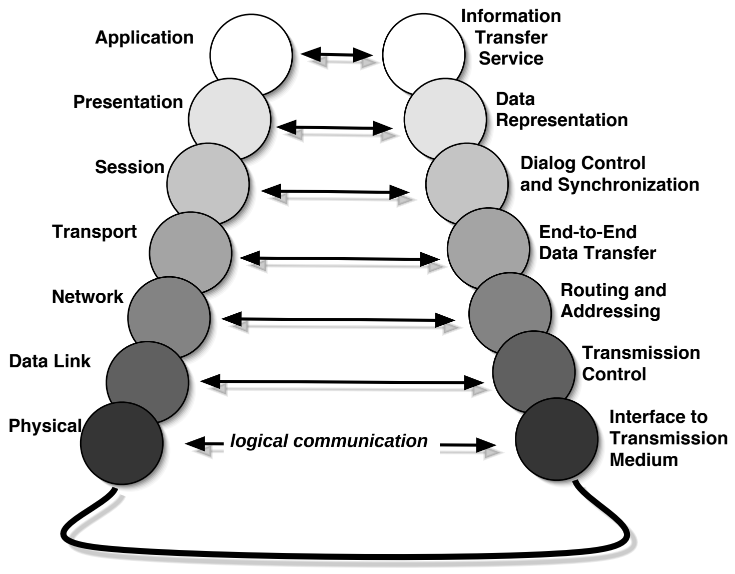


Figure 2.9: ISO 7-layer-model

ing, and transmission-media support. The first two layers are made up of modules that implement communications protocols; the software in the third layer generally includes a protocol sublayer, as wenn as one or more network device drivers.

The topmost layer in the network subsystem is called the *transport layer*. The transport layer must provide an addressing structure that permits communication between sockets and any protocol mechanisms necessary for socket semantics, such as reliable data delivery. The second layer, the *network layer*, is responsible for the delivery of data destined for remote transport or for network-layer protocols. The *network interface layer* or *link layer* at the bottom of the stack is responsible for transporting messages between hosts connected to a common transmission medium. The network-interface layer is mainly concerned with driving the transmission media involved and doing any necessary link-level protocol *encapsulation* and *decapsulation*.

The internal structure of the networking software is not directly visible to the user (and application). Instead, all networking facilities are accessed through the socket layer. Each communication protocol that permits access to its facilities exports a set of user request routines to the socket layer.

The layering described here is a *logical layering*. The software that implements network services may use more or fewer communication protocols according to the design of the network architecture being supported. For example, raw sockets often use a null implementation at one or more layers. At the opposite extreme, *tunnelling* of one protocol through another uses one network protocol to encapsulate and deliver packets for another protocol and involves multiple instances of some layers.

Early versions of BSD were only used as end systems in a network and played the role of sender or receiver of data. A simple data flow through the various network stack layers for an end system is shown in fig. 2.12. The reliability of the BSD Unix network stack,

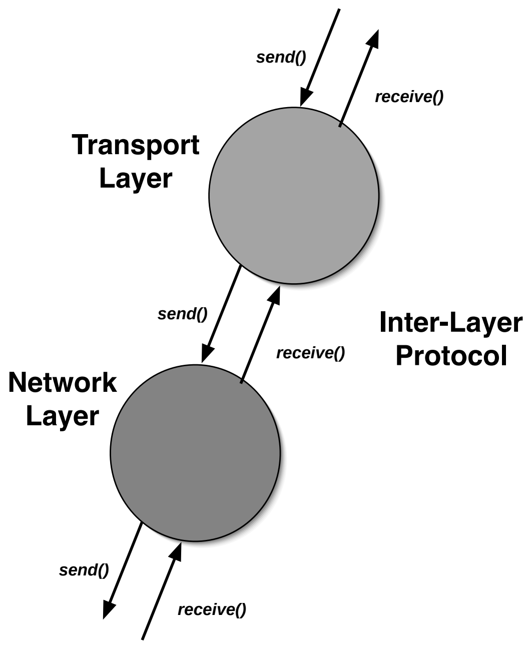


Figure 2.10: Inter-layer information exchange

however, lead to their widespread use in the core of the Internet, so BSD systems had to implement additional functionality such as bridging, routing, protocol conversion, and data en- and decryption. This makes the possible data flow in a system increasingly complex, a fact that is also represented in the code. However, the basic idea that is still valid is that there are only four real data paths through a network node:

- **Inbound** – Destined for a user-level application
- **Outbound** – From user-level application to the network
- **Forward** – Whether bridged or routed the packets are not for this node but to be sent on to another network or host
- **Error** – A packet has arrived that requires the network subsystem itself to send a response without the involvement of a user level application

Inbound data received at a network interface flow upward through communication protocols until they are place in the receive queue of the destination socket. Outbound data flow down to the netowrk subsystem from the socket layer through calls to the transport-layer modules that support the socket abstraction.

Certain properties of the network functionality, however, can not be fit inside this strictly directed layer model. Examples for this crosscutting functionality are quality of service (QoS) control and packet filtering. One approach to solve this problem is to replace the traditional network stack architecture by network heaps [16]. This solves some common performance problems and eliminates crosscutting problems by changing the modularization structure of the system at the expense of more complex programming.

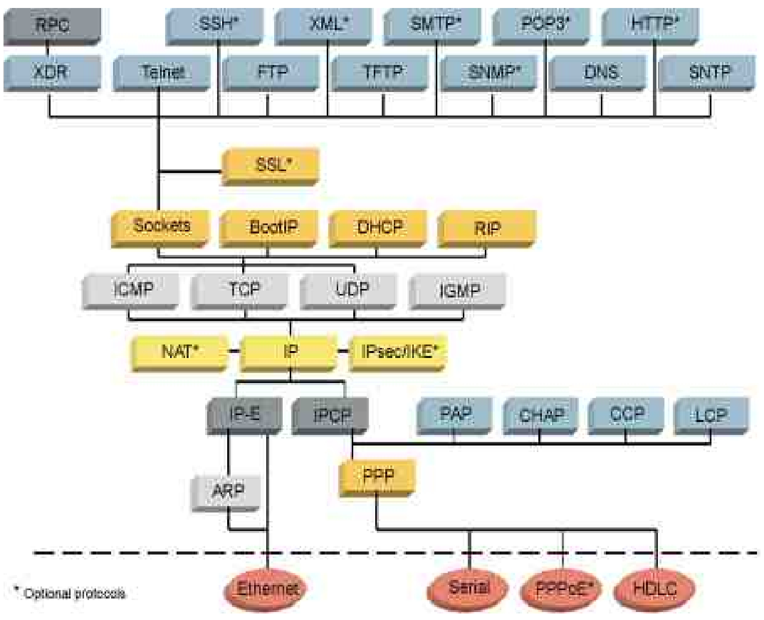


Figure 2.11: NetBSD network stack components

2.4.2 Virtual File System Structure

Another example for layered organization is the virtual file system. Here, processes are provided a common API to access files and directories independent of their physical (local disk, network, optical disk, etc.) location and organization structure (ufs, NFS network file system, FAT). Common functionality need not be replicated in each single file system implementation, and the various filesystem implementation are in turn provided a consistent interface to the virtual file system layer.

Early Unix systems used file system entries that directly referenced the local file system inode. This was a feasible approach as long as only one file system implementation existed, but today's requirements demand a more flexible system, so the architecture had to be generalized.

The virtual file system, vfs, is the kernel interface to file systems. The interface specifies the calls for the kernel to access file systems. It also specifies the core functionality that a file system must provide to the kernel.

The focus of vfs activity is the vnode, which is in turn the focus of all file activity in NetBSD. There is a unique vnode allocated for each active file, directory, mounted-on file, fifo, domain socket, symbolic link and device. The kernel has no concept of a file's underlying structure and so it relies on the information stored in the vnode to describe the file. Thus, the vnode associated with a file holds all the administration information pertaining to it.

When a process requests an operation on a file, the vfs interface passes control to a file system type dependent function to carry out the operation. If the file system type dependent function finds that a vnode representing the file is not in main memory, it dynamically allocates a new vnode from the system main memory pool. Once allocated,

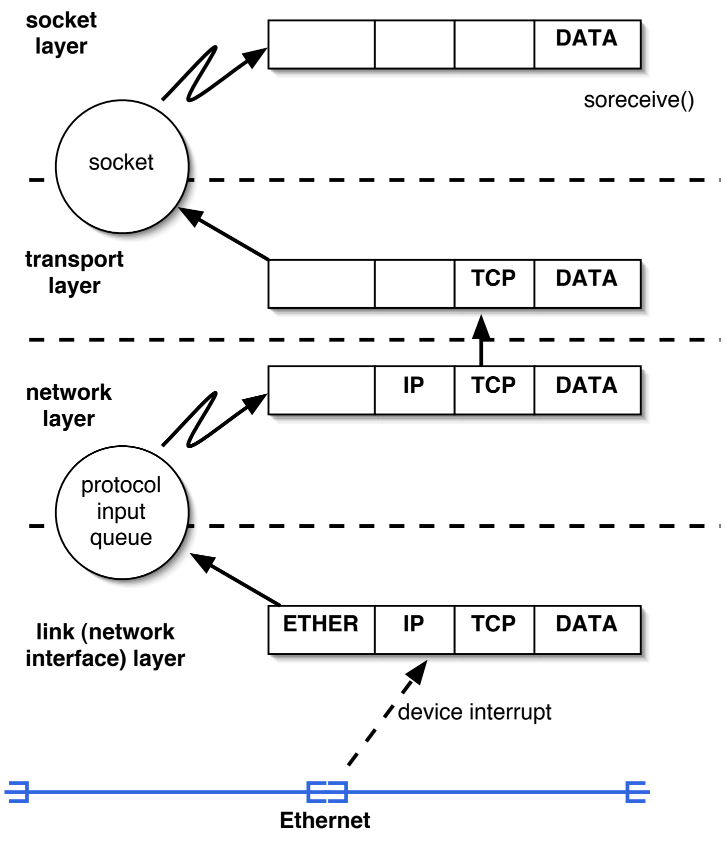


Figure 2.12: NetBSD network stack data flow

the vnode is attached to the data structure pointer associated with the cause of the vnode allocation and it remains resident in the main memory until the system decides that it is no longer needed and can be recycled.

Files and file systems are inextricably linked with the virtual memory system, so the vnode also contains the data maintained by the virtual memory system.

In addition to layering a single file system on top of the vfs abstractions, the uniform interfaces provided by the file system modules allow for more complex interaction – file systems can be "stacked" on top of each other. This seems at first absurd, but has many applications in conjunction with pseudo file systems. One application for this technology is stacking a writeable cache-only file system on top of a read-only file system, so that all writes to files are only temporary and the original system state can be restored after a restart. Another application is to provide access control as a stacked file system layer for file systems like FAT that have no notion of file owners and permissions. The case illustrated in fig. 2.13 shows yet another layering that provides a transparent, distributed network file access using a special virtual file system between the vnode layer and the local block devices.

Virtual file system operations, like networking operations, often cannot be restricted to operate solely on one layer. In the vfs case, cross-layer issues include e.g. disk bandwidth scheduling, file system entry aliasing (i.e., providing access to a file using different file

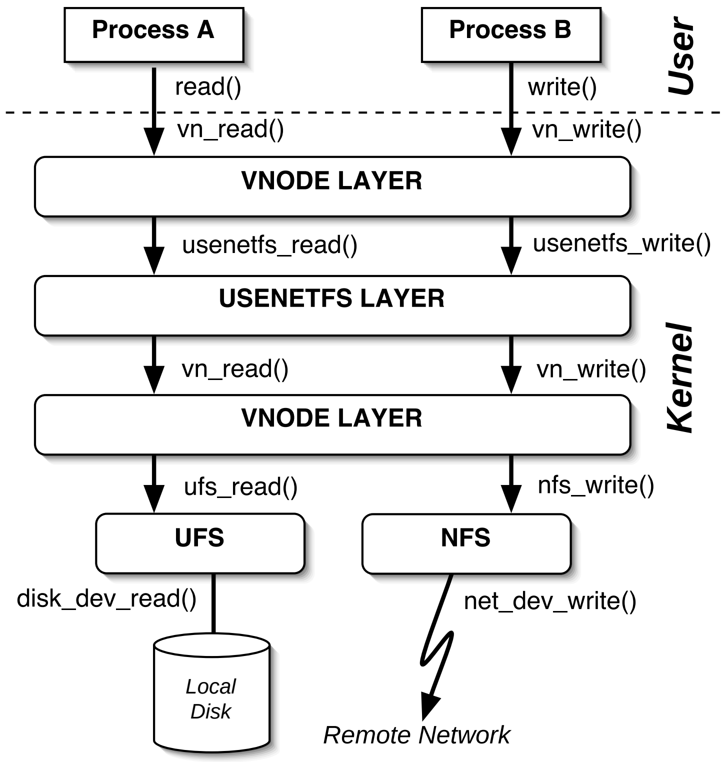


Figure 2.13: Stacked file systems in NetBSD

names), and mounting respectively unmounting of file systems.

2.5 Contexts

In addition to handling various different tasks, each of which has to be provided in several varieties, kernel code also faces another dependency. While traditional, single-threaded user mode applications rely on the notion of running their code in one single, well-defined context, operating system code usually does not run on behalf of itself, but rather in reaction to events which are generated from the hardware as well as from user (and kernel) mode processes running on top of it.

2.5.1 Startup and Control

After the kernel is loaded into memory, the kernel initializes various internal data structures, detects the hardware configuration of the specific system it runs on and configures itself accordingly and finally prepares to start the first user mode process. While all other processes in a Unix system are created using the `fork` system call from inside a running process, this first process is hand-crafted by the kernel by explicitly creating process-table and virtual memory management information. This first process, traditionally called `init` and having the process id (pid) of 1, is started and control is transferred to the running

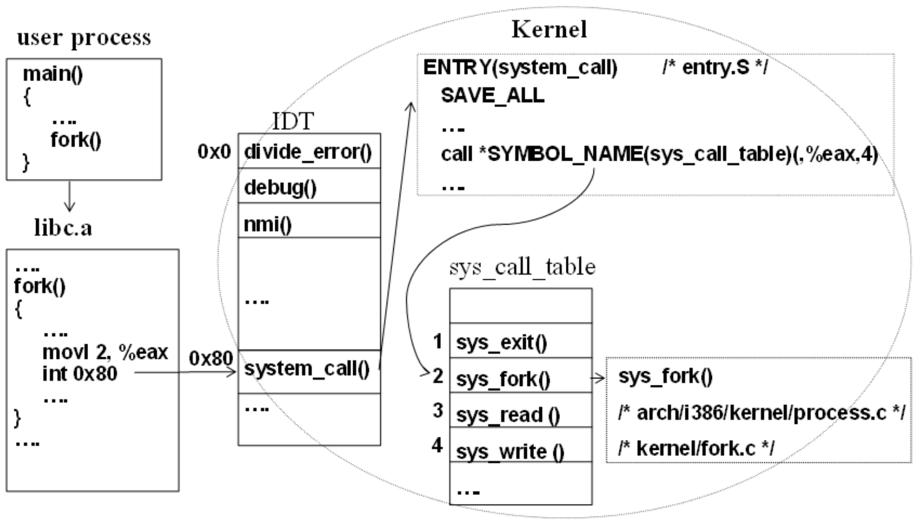


Figure 2.14: System call dispatch

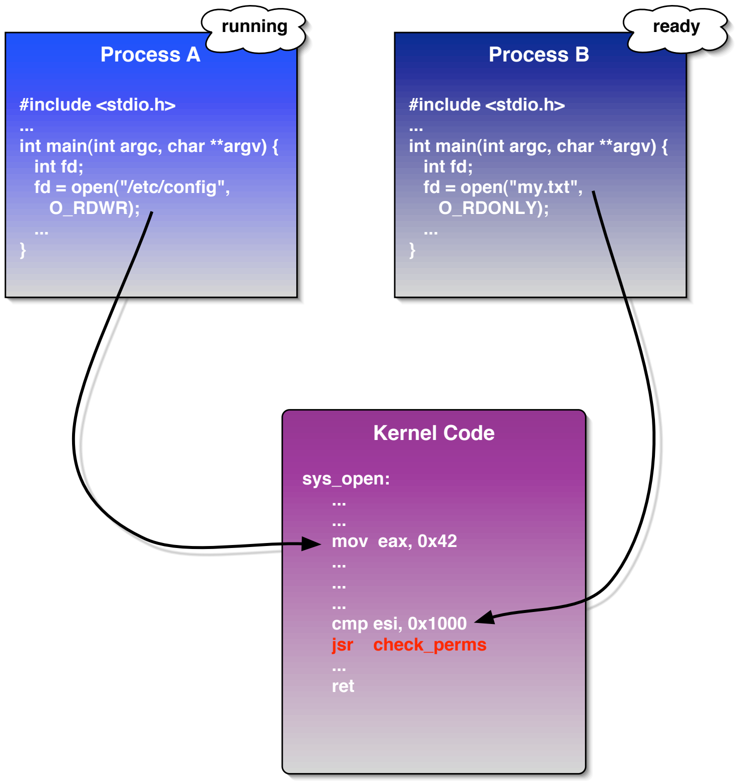
init process in user mode. Init, in turn, reads a configuration file and starts up all further user mode activity (like network daemons, processes handling local logins, etc.) on the system. From the moment init starts running in user mode, the kernel mostly reacts to actions from processes and messages (usually represented by interrupt requests) from the hardware, essentially like a huge library commonly used by many processes.

The consequence of this operation model is that code in the kernel itself is always executed in a certain *context* and so has to keep different state, accordingly. The most important contexts are process contexts and hardware contexts.

2.5.2 Process Contexts

When a process executes a system call, control is transferred from user mode to kernel mode using a special machine code instruction like `trap` or related mechanisms like software interrupts or call gates. An example for system call dispatch on x86 systems is shown in 2.14. This mechanism interrupts the code flow of the current process, changes the execution mode of the processor to a privileged state and transfers control to a location in kernel virtual memory corresponding to the address of the *system call handler*. This handler analyzes the system call number that is passed on the stack or in a processor register and further dispatches the flow of control to the appropriate in-kernel system call function. This system call code – and all functions called on behalf of this code at this moment – is said to be running in the *context* of the process that executed the system call.

Since Unix is a multitasking system, several user mode processes are permitted to run (semi-)concurrently and it is not uncommon for two unrelated processes to be executing the kernel code of a common system call (like, e.g., `read`) at the same time, since processes might also be interrupted or rescheduled while running in kernel mode. An example of two processes executing in different locations inside the `sys_read` system call is shown in fig. 2.15. As a consequence, the exactly same code, residing in the same memory locations



```
#include <stdio.h>
...
int main(int argc, char **argv) {
   int fd;
   fd = open("/etc/config",
      O_RDWR);
   ...
}
```

```
#include <stdio.h>
...
int main(int argc, char **argv) {
   int fd;
   fd = open("my.txt",
      O_RDONLY);
   ...
}
```

```
sys_open:
      ...
      ...
      mov  eax, 0x42
      ...
      ...
      ...
      cmp esi, 0x1000
      jsr    check_perms
      ...
      ret
```

Figure 2.15: Two processes in kernel mode executing sys_read()

in kernel virtual memory, must behave differently according to the context of the process that originally initiated its execution. The impact of this on providing AOP functionality is discussed in a later chapter.

Exceptions are another common case of control transfer from user to kernel mode initiated by the actions of a running process. Some exceptions occur regularly, like page faults indicating to the virtual memory subsystem of the kernel to retrieve further pages from the on-disk executable or the swap space area of the hard disk. Other exceptions, like arithmetic exceptions (division by zero, overflow, etc.), occur unwantedly and usually indicate a programming error in the user mode process that has to be handled by the kernel – usually, by terminating the operation of the faulting process. These exception handlers, too, run in the context of the corresponding user mode process.

2.5.3 Asynchronous (Hardware) Contexts

Some parts of the kernel do not operate synchronously in response to user mode actions, but rather have to react asynchronously to unpredictable external events, usually signalled by devices through some sort of interrupt mechanism (see fig. 2.16).

Interrupt handler code running in response to external interrupts is running in a special *interrupt context* and usually run uninterrruptible by temporarily turning off all interrupt handling (like in Linux) or adjusting the interrupt level of the processor so that only

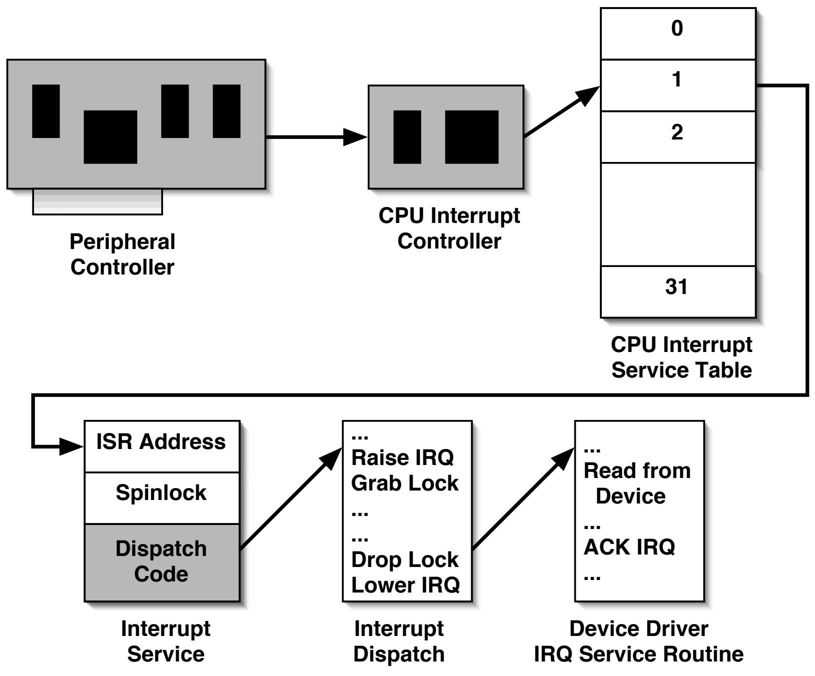


Figure 2.16: Interrupt handling

higher-privileged interrupts may be processed. Current trends, however, try to minimize the time spent in an interrupt handler and relegate most of the traditional bottom-level work of the handler to special in-kernel *interrupt handler threads* that run independently of the interrupt context.

2.6 Summary

Unix kernels are complex software systems that evolved from a simple implementation of anticipatory concepts into a software system typically spanning millions of lines of code. The flexible architecture of Unix has permitted the system to acquire new functionality along the way, like layered architectural frameworks for network and file system implementation, which nowadays seem like a natural fit for the Unix structure. Compared to most application software, a kernel has to handle many more tasks concurrently, while always being alert to asynchronous, external events that change the flow of control in the system. Much of the data kept inside the kernel depends on changing conditions and thus is expected to dynamically change over time. The evolution of the system as well as the asynchronous nature of control flow in a kernel have implications for the development of kernel-mode code, so that writing and evolving kernel code is a demanding task compared to application-level development.

3. In-Kernel Crosscutting Concerns

“If builders built buildings the way programmers wrote programs, then the first woodpecker that came along would destroy civilization.”

— Weinberg’s Second Law

3.1 Introduction

Complex systems like operating system kernels have to fulfill many differing requirements or concerns. While software developers try to separate different functionality into distinctive modules, some of the functionality required does not fit into this modularization structure. Code implementing this functionality ends up being scattered over the system’s code base and tangled with code implementing different functionality as a so-called “crosscutting concern”. This chapter gives an overview of crosscutting concerns, with special emphasis on the structure of kernel code.

3.2 Modularity and Concerns in Kernel Code

The opening keynote of the first AOSD conference [67] credited Dijkstra for reminding software developers of the Roman principle of divide-and-conquer. In other words, the *separation of concerns* (SOC) is an ages-old concept that arose from the observation that many large problems are actually easier to handle if they can be broken down into smaller sub-pieces, especially if the solutions to the single sub-problems are relatively easy and together form a solution to the large problem.

Separation of concerns can occur in different ways – by process, by notation, by organization, and by language mechanisms. One of the problems encountered when developing software is that the single hierarchical structures offered by current (object oriented, procedural or functional) programming languages are too limited to effectively separate all concerns in complex systems.

The AOSD Wiki Glossary [4] defines a concern as follows:

“A concern is an area of interest or focus in a system. Concerns are the primary criterium for decomposing software into smaller, more manageable and comprehensible parts that have a meaning to a software engineer. Examples of concerns include requirements, use cases, features, data structures, quality-of-service issues, variants, intellectual property boundaries, collaborations, patterns and contracts."

Concerns arise throughout the software lifecycle, involving formal and informal artifacts of all kinds. Some apply within individual artifacts, but many span multiple artifacts and lifecycle phases. For example, a requirement concern defined in a requirements document spans the architecture, design, code, tests and documentation that address that requirement.

Concerns generally have both intension and extension. The intension embodies the meaning, or intent, such as "the collision-avoidance feature" or "the Linux variant", and might be expressed formally as a query or predicate. Ideally, it is stable even as the software changes. The extension enumerates the software elements that pertain to the concern, which evolve as the software evolves.

In operating system kernels, imperative programming languages still are in common use. All kernels of Unix-based systems today are implemented in a variant of the C programming language; experiments with including object oriented programming styles, mostly C++, into kernel code, have not found widespread acceptance. With the exception of an Objective C-based device driver framework in the OpenStep operating system, which was later replaced by the C++-based IOKit in the followup MacOS X operating system, no widespread use of object oriented languages in mainstream kernels is known.

Thus, a definition of concerns and modularity for procedural languages, and especially in a kernel context, is required in order to discover and solve problems related to crosscutting.

Software developers often use the term "module" to represent a concern that is to be implemented, so modularity is another term for talking about separation of concerns. The term modularity is used when talking about formal artifacts, such as code, and refers to achieving a high degree of clarity in the separation between the individual units (modules). The code that implements a concern is a module if:

- it is textually local,
- there is a well-defined interface that describes how it interacts with the rest of the system,
- the interface is an abstraction of the implementation, in that it is possible to make material changes to the implementation without violating the interface,
- an automatic mechanism enforces that every module satisfies its own interface and respects the interface of all other modules, and
- the module can be automatically composed – by a compiler, loader, linker etc. – in various configurations with other modules to produce a complete system.

Traditional languages and modularization mechanisms suffer from a limitation called the Tyranny of the Dominant Decomposition: the program can be modularized in only one way at a time, and the many kinds of concerns that do not align with that modularization end up scattered across many modules and tangled with one another.

The structure of a typical kernel is described in chapter 2. Here, a hierarchical division of the source code tree is performed according to major (obvious) functional requirements, like implementing memory management, device drivers or file systems. This separation into distint directories is the most coarse unit of modularization below the project level. Fig. 3.1 shows how the semantic units in a program source code are further broken down.

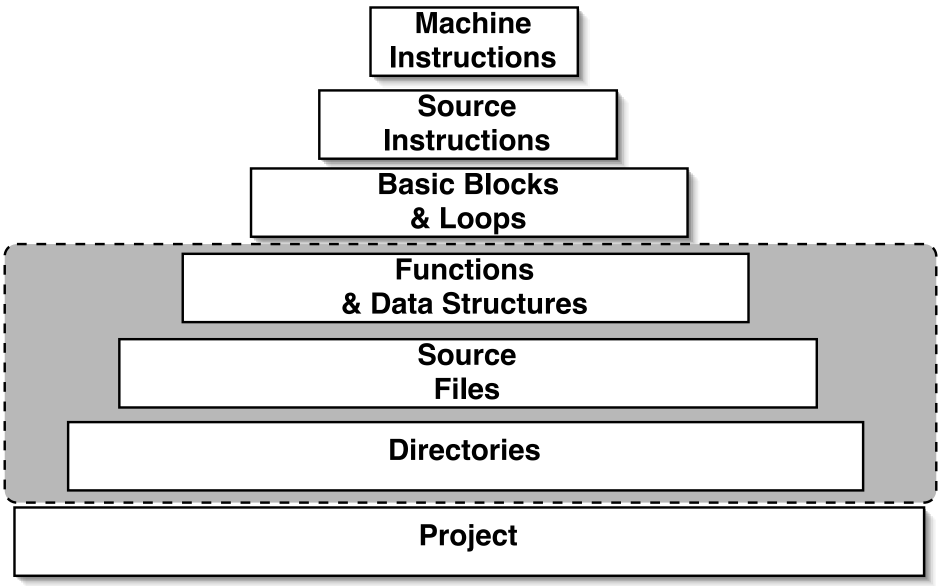


Figure 3.1: Modularity pyramid

The source code of a project consists of a top-level directory which contains several subdirectories for major functional units. Inside each directory, several files exist that divide the task up into source code files that implement a subset of the functionality. Inside the source files, two levels of modularity can be found – code and data modules. Code modules on this level are expressed as functions, whereas data modules in a procedural language are expressed as structs or records of basic data types.

The various levels of granularity are of importance for the semantics of join points, discussed in the next chapter. Granularity levels of importance for the state of aspect oriented programming available are marked with a grey box in fig. 3.1.

To illustrate the modularity inherent in the NetBSD kernel, the various levels of modularity going, bottom-up, from the machine instruction to the project level, are shown in fig. 3.2.

The most fine-grained level is a single machine instruction, `mov %eax, 0`, which is one of the instructions generated by the C compiler for the source code instruction `return 0;`. This instruction is located inside a basic block, which is the body of a loop `while (--fnlen >= 0)`. The loop is part of the function `int isofncmp(fn, fnlen, isofn, isolen, joliet_level)`, which in turn is contained in the source file `cd9660_util.c`. The source file, then, is contained in a directory `src/sys/fs/iso9660/`, which is part of the NetBSD kernel project.

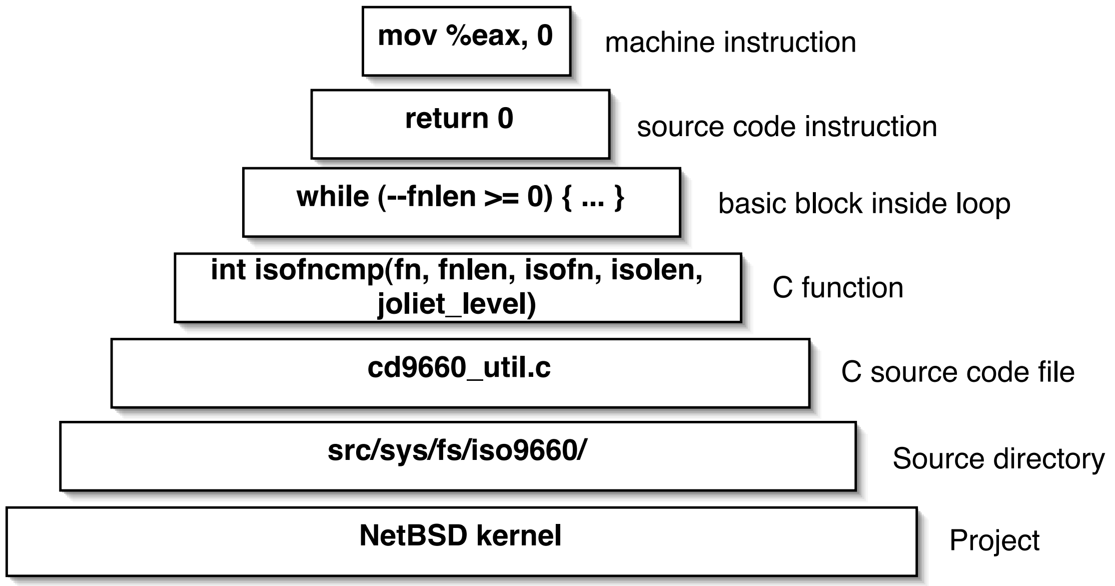


Figure 3.2: Modularization hierarchy of a machine instruction

3.3 A Definition of Crosscutting Concerns

3.3.1 An Example: Telecommunications

The designer of a software system usually selects a modularization structure that adequately represents the *core concerns* – i.e., the properties defined by the main functional requirements – of the system. For example, in a telecommunications application, the routing of calls would be a core concern, but the system structure would be crosscut by calls to timing and billing functions. A simplified version of the telephone call routing function might look like this:

```
void route_call(phonenumber *source, phonenumber *dest) {
        int status;

        status = dial(dest);

        if (status != CALL_OK) {
                if (status == BUSY)
                        return(EBUSY);
                if (status == NOSUCHNUMBER || status == NETERROR)
                        return(EFAIL);
        }

        start_time(source);

        if (! Is_0800_Number(dest))
                start_accounting(source);

        do_call(dest);

        if (! Is_0800_Number(dest))
21                    end_accounting(source);
22
23            end_time(source);
24
25            hangup(dest);
26    }
```

In this piece of code, the main functionality clearly is the call routing, performed by the `route_call` function. However, keeping logs of the connection time of the call (to provide a connection log to the customer) as well as keeping track of the connection fees (which may require a complex implementation themselves, as fees are dependent on destination, time-of-day, etc.) are required side-effects of the call routing.

Time keeping is activated in lines 13 and 23, respectively. Here, simple calls to functions that record a beginning and ending time of a call are performed. The accounting functionality is a bit more complicated. Since calls to certain numbers are free of charge, the accounting function is only called for numbers where charges apply. This is also one of the reasons for separating timekeeping from accounting.

The code presented here, thus, implements three concerns:

- *Call routing*, the core concern
- *Timekeeping*, and
- *Accounting*

In this implementation, the calls to functions for timekeeping and accounting are implementing a different functionality than call routing by calls to functions in the respective modules. Thus, accounting and timekeeping are *crosscutting concerns* related to the main functionality and modularization structure of the example telecommunications application. Using a procedural programming language, they cannot be represented adequately in the modularization structure of the source code.

3.3.2 Scattering and Tangling

In large software systems, software elements are usually not cleanly separated. Instead, the code implementing a certain concern is usually *scattered* throughout the code and *tangled* with lines of code implementing a different concern. The representation of a concern is scattered over an artifact if it is spread out rather than localized. The representation of concerns are scattered within an artifact if they are intermixed rather than separated. Scattering and tangling often go together, even though they are different concepts.

Crosscutting is defined by the AOSD Wiki as “a structural relationship between representations of a concern. In this way it is similar to other kinds of structure, like hierarchical structure and block structure.” Crosscutting is thus not equivalent to scattering and tangling of code – while a crosscutting concern may be implemented by code scattered throughout the code base and tangled with code that implements different concerns, the existence of tangled and scattered code does not necessarily identify a crosscutting concern. Rather, it can also be bad programming style and the concerns at hand could have been easily represented by the existing modularization structure.

3.3.3 Crosscutting in Related Research

Formal definitions of crosscutting concerns have been published in [45] and [141]. These analyses of crosscutting concerns in software systems have, however, only been applied to systems implemented in object-oriented languages. Object-oriented languages employ a different approach to modularization, so the crosscutting concerns identified in these papers are not usable for the analysis of kernel code.

A practical approach to identify crosscutting concerns based on reverse-engineering technologies of large software systems is described in [19] and [20]. This analysis starts from the assertion that source code implementing cross-cutting concerns (CCCs) tends to involve a great deal of duplication. First of all, since such code cannot be captured cleanly inside a single abstraction, it cannot be reused. Therefore, developers are forced to write the same code over and over again, and are tempted to simply copy and paste the code and adapt it slightly to the context. Alternatively, the developers may use particular coding conventions and idioms, which also exhibit similar code.

As a consequence, clone detection techniques were evaluated for identifying CCC code, since they automatically detect duplicated code in the source code of an application.

While this approach has shown success in the particular software system investigated – a 19,000 line subsystem of a 10 million line-of-code system – the detection of cloned code usually works only for larger pieces of code. Duplicated code alone also does not need to be a reliable indicator for crosscutting concerns. As seen above, scattering and tangling of code are no reliable indicators for a crosscutting concern. Code duplication can, in the same ways, also simply be an indicator for bad programming style, where the repetition of code would be simply removable without breaking the given modularization structure.

The simple telecommunications example above shows another problem with the duplication approach. In this case, only short, simple pieces of code (minimally, a single function call) represent a tangled crosscutting concern. Finding occurences of these short function calls tend to be impossible with the duplication approach.

3.3.4 Identifying Crosscutting

The preceding sections tried to define crosscutting by an analysis of the symptoms that apply to crosscutting concerns. However, all of these symptoms are not sufficient to identify code belonging to a crosscutting concern. Scattering and tangling as well as code duplication are analyzed using *syntactical* methods. The modularization structure of a system, which is violated by a crosscutting concern, however, is not cleanly expressible in syntactic terms, but it is rather a *semantic* structure.

Identifying crosscuts against this semantic structure requires advanced tools for semantic analysis. While there are some approaches defining the semantics of programming languages [29, 5], no semantic description detailing the modularization structure for large software projects exists.

In an operating system context, thus, an automated analysis of existing code to identify and factor out code related to crosscutting concerns is not feasible. Tools that identify tangling, scattering and code duplication, however, can assist a human expert who is revising the code in identifying possible candidates for crosscutting concerns.

3.4 The Trouble with Macros

The C language is special in that certain constructs in C source code are not handled by the language compiler itself, but rather by a preprocessing component that processes the C source files before the compiler scans and parses the source code. As a result, the compiler frontend never sees the defined function-alike macros, but only the expanded form.

Examples for this can be found in numerous places inside the kernel. The header file `src/sys/arch/i386/include/disklabel.h` defines macros like this:

```
#define DISKUNIT(dev)    ((minor(dev) / OLDMAXPARTITIONS) % __I386_MAXDISKS)
#define DISKPART(dev)    ((minor(dev) % OLDMAXPARTITIONS) + \
    ((minor(dev) / (__I386_MAXDISKS * OLDMAXPARTITIONS)) * OLDMAXPARTITIONS))
#define DISKMINOR(unit, part) \
    (((unit) * OLDMAXPARTITIONS) + ((part) % OLDMAXPARTITIONS) + \
     ((part) / OLDMAXPARTITIONS) * (__I386_MAXDISKS * OLDMAXPARTITIONS))
```

When used in C code, the macros DISKUNIT, DISKPART and DISKMINOR look like normal function calls, like in this example from `src/sys/arch/pmax/pmax/disksubr.c`:

```
bp->b_dev = makedev(major(dev), DISKMINOR(DISKUNIT(dev), labelpart));
```

After being processed by the C preprocessor, however, the above code looks like this:

```
bp->b_dev = makedev(major(dev),
    ((( ((minor(dev) / OLDMAXPARTITIONS) % __I386_MAXDISKS))
          * OLDMAXPARTITIONS) + ((labelpart) % OLDMAXPARTITIONS) + \
     ((labelpart) / OLDMAXPARTITIONS) * (__I386_MAXDISKS * OLDMAXPARTITIONS)))
```

No function calls remain, as the macro substitution replaced what was formerly looking like a simple function call with a complex, in-line arithmetic expression. So, macro calls can fool the observer into thinking that certain constructs of the source code are function calls, but in reality these are rather textually expanded preprocessor macros.

This code transformation has severe implications for the identification of crosscutting concerns in the code. Whereas the non-preprocessed code seems to clearly exhibit function calls, the compiler is presented with the macro code, in which name replacement and replacement of elements by previously defined macros has taken place. As a consequence, these locations may not be identifiable for aspect-oriented tools and provide obstacles to the unsuspecting developer.

3.5 OO in Kernel Mode

Since the implementation of most kernels still uses strictly procedural languages, the methods available to the programmer to implement data abstraction are restricted. Nevertheless, in many situations, object oriented approaches make implementing useful concepts like polymorphism more transparent to the programmer. Accordingly, in some places of kernel code, an object-oriented programming style is imitated by a combination of procedural techniques.

3.5.1 Object Orientation Using Procedural Languages

When defining object-oriented abstractions in C, classes are implemented by combining traditional C data structures with functions that operate on the data in the structure. This is realized by adding a set of function pointers to the structure, like in this simple example:

```c
struct myobject {
    int value;

    int (* get_value) (void);
    void (* set_value) (int);
    int (* add_value) (int);
}
```

Here, an object `myobject` is defined. It consists of an instance variable `value` and three member functions, `get_value`, `set_value`, and `add_value`, which are referenced using C-style function pointers.

The object is instantiated by defining a variable of the structure data type:

```c
struct myobject o;

o->set_value(42);
o->add_value(23);
printf("%d\n", o->get_value());
```

Overriding inherited methods in newly created instances is also possible by simply redefining the function pointers.

One important concept to be implemented is polymorphism, the idea of allowing the same code to be used with different types, resulting in more general and abstract implementations. In C code, this behavior can be achieved by using `union`s, which allow a programmer to identify part of a structure as different variable types:

```c
struct example {
    int type;
    union value {
        int i;
        char c;
    };
    int (*set) (void *value);
};
```

This simple emulation of object-oriented semantics, however, does not offer all features of real object-oriented languages. For example, data encapsulation is not enforced by the compiler, so that all code in a system can effectively access data in a struct that is intended to be private to this pseudo-object.

A more detailed introduction to object-oriented programming in procedural languages is given in [118]. In the following section, an example of object-oriented structures in the NetBSD kernel is presented.

3.5.2 Example: Console Handling

The so-called "console" is the main point of user interaction in a Unix-based system. Its manifestations on the hardware side range from a simple, "dumb" serial-line ASCII terminal to high-resolution color graphics adapters connected to one or more screens. On the application side, a multitude of different control mechanisms, usually using so-called "escape sequences" have been invented throughout Unix history, which have to be supported (emulated) to run applications that use text-mode menus, the curses libraries, etc.

Further requirements to console code are support for screen power management and virtualization, i.e., displaying several virtual console screens alternatively on one physical device. Thus, the kernel code implementing the console has to be extremely flexible.

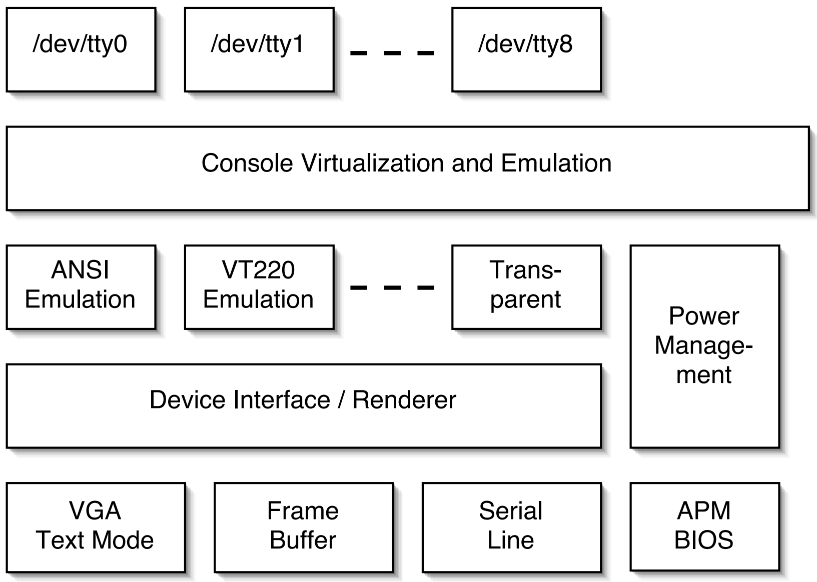


Figure 3.3: Console subsystem structure

The structure of the NetBSD console code is shown in fig. 3.3. The subsystem provides various emulation modules to the software using it, multiplexed over virtual console devices. On the hardware side, the console code provides support for various console hardware types as well as an interface to the system power management.

In console code, several object-oriented approaches are used. A good example for this is the `terminal_emulator` object:

```
struct terminal_emulator {
    char *name;
    /* Terminal emulation routines */
    int  (*term_init)        __P((struct vconsole *vc));
    int  (*putstring)        __P((char *string, int length, struct vconsole *vc));
    int  (*swapin)           __P((struct vconsole *vc));
    int  (*swapout)          __P((struct vconsole *vc));
    int  (*sleep)            __P((struct vconsole *vc));
    int  (*wake)             __P((struct vconsole *vc));
    int  ( *scrollback)      __P(( struct vconsole *vc ));
    int  ( *scrollforward)   __P(( struct vconsole *vc ));
    int  ( *scrollbackend)   __P(( struct vconsole *vc ));
    int  ( *debugprint)      __P(( struct vconsole *vc ));
    int  ( *modechange)      __P(( struct vconsole *vc ));
    int  ( *attach )         __P(( struct vconsole *vc, struct device *dev,
                                   struct device *dev1, void *arg ));
};
```

The data contained in this structure is very simple – it is just a pointer to a string containing the console name (e.g., /dev/tty0). The other entries in this struct all refer to member functions that are redefined using the function pointers to indicate the correct function for the current console instance.

A similar construct can be found in the definition of the rendering engine struct, struct render_engine, in the file src/sys/arch/acorn32/include/vconsole.h.

Combining code and data in C structs has implications for crosscutting concerns. In order to differentiate crosscutting functionality involving pseudo object-oriented structures in the kernel from normal procedure calls, the context of a function call has to be observed, since the struct effectively introduces a new level of modularization.

3.6 Crosscutting Concern Example: Disk Quota

Explaining crosscutting concerns with simple examples serves as an introduction to the topic. However, this approach does not give an in-depth insight into crosscutting in real code. Thus, this section presents an example from actual BSD kernel code. Crosscutting in quota management is first described in the FreeBSD kernel in [32].

It is often necessary to split up disk space between multiple users. Disk quotas are designed to track disk utilization and enforce limits for all users and groups. The inherent structure of quota is a set of low-level disk space related operations that consistently monitor/limit all disk usage.

Quota checking is implemented using a function call to the chkdq function. This call crosscuts operations that consume and free disk space in file systems that offer support for this functionality.

Fig. 3.4 shows how code that implements quota checking is scattered throughout various file systems. In every file system implementation, several files are affected, indicated by the number in square brackets. The implementation of quota is not restricted to one

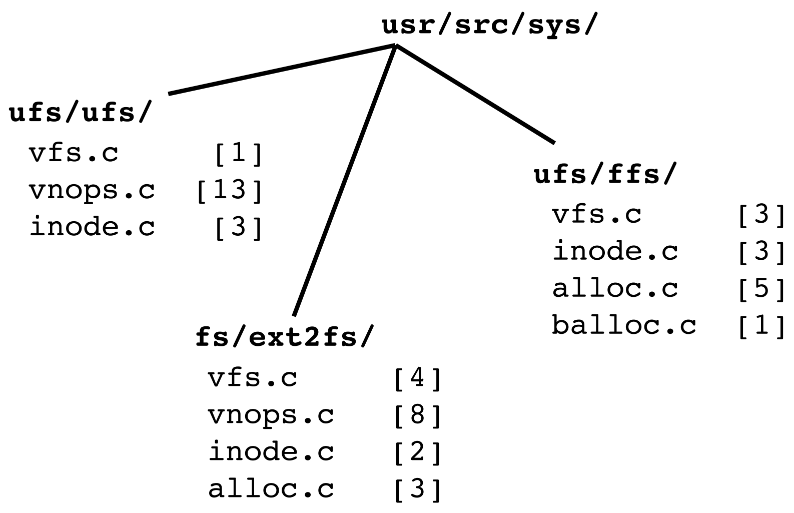


Figure 3.4: Quota code in various file systems

single location in the file, but rather can be found in several locations each.

The first example of a piece of code in which quota management is implemented as a crosscutting concern occurs is from the function `ufs_chown`, contained in the file `ufs/ufs/vnops.c`. The `chown` function implements the functionality known from the corresponding user mode command and system call. Changing the ownership of a file has direct implications on the quota management, since changing a file's owner to a different user ID has to subtract the disk space used by this file from the previous user and add it to the new user's quota.

```
        if ((cred->cr_uid != ip->i_uid || uid != ip->i_uid ||
            (gid != ip->i_gid &&
             !(cred->cr_gid == gid || groupmember((gid_t)gid, cred)))) &&
            ((error = suser(cred, &p->p_acflag)) != 0))
                return (error);

        ogid = ip->i_gid;
        ouid = ip->i_uid;
        if ((error = getinoquota(ip)) != 0)
                return (error);
        if (ouid == uid) {
                dqrele(vp, ip->i_dquot[USRQUOTA]);
                ip->i_dquot[USRQUOTA] = NODQUOT;
        }
        if (ogid == gid) {
                dqrele(vp, ip->i_dquot[GRPQUOTA]);
                ip->i_dquot[GRPQUOTA] = NODQUOT;
        }
        change = DIP(ip, blocks);
        (void) chkdq(ip, -change, cred, CHOWN);
        (void) chkiq(ip, -1, cred, CHOWN);
        for (i = 0; i < MAXQUOTAS; i++) {
                dqrele(vp, ip->i_dquot[i]);
                ip->i_dquot[i] = NODQUOT;
```

```
25              }
26
27              ip->i_gid = gid;
28              DIP_ASSIGN(ip, gid, gid);
```

Listing 3.1: Crosscutting quota example #1

Here, the code in lines 1. . . 8 and 27 upwards implements functionality related to the main concern of the function, namely changing the file's (vnode's) ownership. Inbetween lines 9. . . 25, quota management is introduced as a crosscutting concern. Here, disk quotas are released using the `dqrele` function if various conditions related to user and group IDs of the file (which are not relevant to the crosscutting discussion itself) are met.

The second excerpt is taken from the function `ffs_alloc` in the file `ufs/ffs/alloc.c`. This function is responsible for allocating a block in the FFS file system. Here, disk quota checking using a call to the `chkdq` function in line 5 crosscuts the file-system specific checks for available space on the partition.

```
 1              if (size == fs->fs_bsize && fs->fs_cstotal.cs_nbfree == 0)
 2                      goto nospace;
 3              if (cred->cr_uid != 0 && freespace(fs, fs->fs_minfree) <= 0)
 4                      goto nospace;
 5              if ((error = chkdq(ip, btodb(size), cred, 0)) != 0)
 6                      return (error);
 7              if (bpref >= fs->fs_size)
 8                      bpref = 0;
 9              if (bpref == 0)
10                      cg = ino_to_cg(fs, ip->i_number);
11              else
12                      cg = dtog(fs, bpref);
13              bno = ffs_hashalloc(ip, cg, (long)bpref, size,
14                                           ffs_alloccg);
```

Listing 3.2: Crosscutting Quota Example #2

Further down in the same function, another example for crosscutting can be found. Here, the quota for the specific user is restored to its original value using the `checkdq` function in line 10 if the allocation of a block in the file system has failed if free blocks are available.

```
 1              if (bno > 0) {
 2                      DIP_ADD(ip, blocks, btodb(size));
 3                      ip->i_flag |= IN_CHANGE | IN_UPDATE;
 4                      *bnp = bno;
 5                      return (0);
 6              }
 7              /* Restore user's disk quota because allocation failed. */
 8              (void) chkdq(ip, -btodb(size), cred, FORCE);
 9      nospace:
10              ffs_fserr(fs, cred->cr_uid, "file␣system␣full");
11              uprintf("\n%s:␣write␣failed,␣file␣system␣is␣full\n", fs->fs_fsmnt);
12              return (ENOSPC);
```

Listing 3.3: Crosscutting Quota Example #3

Other pieces of code related to crosscutting quota management throughout many file system implementations look alike and can be identified by searching for calls to functions matching the pattern `%dq%`, such that a semi-automated tool could be used in this case to factor out quota management into a separate module.

3.7 Summary

The current methods for modularization in procedural languages are relatively restricted, even when considering approaches to object oriented modelling of data structures using procedural approaches. In particular, the modularization methods are not sufficient to express crosscutting properties for the functionality required in an operating system kernel. Crosscutting concerns make developing and evolving system software harder, so that methods to create an additional level of abstraction to refactor crosscutting concerns into more manageable entities are required. This approach is explained in the following chapter.

4. Aspect-Oriented Programming

"They took the credit for your second symphony
Rewritten by machine on new technology
And now I understand the problems you can see"

— *Video Killed the Radio Star* Lyrics

4.1 Introduction

Crosscutting concerns in operating system kernels manifest themselves in different ways – as "traditional" existing crosscutting problems in the modularization structure, crosscutting due to cross-layer interaction, crosscutting and interaction between several kernel extensions affecting (at least partially) the same modules and crosscutting problems that show up when trying to extend the kernel. As a consequence, aspect-oriented methods for use in kernel development have to be applicable to the modification of legacy components as well as the implementation of new functionality.

After discussing a recently published analysis of aspect-oriented software development by Forrester Research [143], the following sections of this chapter describe the basic concepts and notions of AOP as well as various types of join points in common use and analyze the use of each in an operating system context. In the remaining sections, static and dynamic AOP approaches are discussed and evaluated in their relevance for solving crosscutting problems in kernel space.

4.2 AOP — considered useful!

Recently, a study by Forrester Research [143] was published that stated broad and general criticism of aspect-oriented software development and compared AOP approaches with the use of the "Go To" statement in procedural languages.

Thus, a discussion of solving crosscutting concerns using aspect-oriented methods would not be complete without refuting at least some of the arguments of that report. This is done in the next section by outlining the special advantages the use of AOP in an operating system kernel context.

Alike all technologies that are relatively new and unseasoned, aspect-oriented software development is not always met with enhusiasm. Recently, a Forrester report by Carl

Zetie[143], one of Forrester's vice presidents, has caused some turmoil in the AOP community by describing aspect-oriented development methods as the equivalent of the GOTO statement for object-oriented programming, which was badmouthed in Edsger W. Dijkstra's famous 1968 letter "Go To Statement Considered Harmful" to the editor of Communications of the ACM [40]. A general discussion of the merits and shortcomings of AOP is outside the range of this topic, but a discussion of several topics mentioned in the report with special consideration of the operating system context concentrated upon can actually turn some of the topics considered by the report's author as shortcomings of AOP into agruments pro AOP in this special context.

In the following sections, the Forrester report is analyzed from a viewpoint of an operating systems developer intending to use AOP.

The shortcomings of AOP when "applied to ordinary IT projects staffed by average programmers and not AOP gurus" are described as :

1. The effects of AOP can be subtle and can change the behavior of code in ways that are not always easily determined simply by reading the code.

Quite the opposite is the case in AOP systems. The approach of aspect-oriented system design is to clear up code that can behave in non-obvious ways due to tangling and scattering of code pieces related to different concerns. One critical point, however, may be induced from here: the current possibilities for defining pointcuts in most aspect-oriented languages often rely on only using syntactic information expressed by regular expressions to define sets of join points for an aspect. This method can introduce problems since the introduction of a new function that matches, but is unrelated to an existing, defined pointcut changes the behavior of the system in unpredictable ways.

Here, on the one hand an advanced method to describe pointcuts – optimally by augmenting the syntactic pointcut description by some kind of semantic information – can clear up confusing issues; on the other hand, keeping to a strict naming scheme in the development of a project, e.g. by adhering to conventions for prefixes when naming functions and variables, can readily help to clear up pointcut matching issues.

2. In some implementations, the introduction – or removal – of an aspect can break existing code in unanticipated ways.

Since this is already clearly described as implementation-specific behavior, implementations – especially in kernel code – have to avoid unexpected side effects and issue compiler warnings or errors. The behavior described, however, is certainly not confined to aspect-oriented systems, but can e.g. also show up in plain object-oriented systems.

3. It reintroduces global effects: the coupling of code fragments that are widely separated in the code body, as well as in the programmer's attention span.

These "global effects" cited here are already inherent in the code. The effects, however, are not obvious when the respective code is scattered and tangled throughout the code base. The amount of global changes that can be introduced depends on the join point models and pointcut functions available in the particular AOP tool. Current tools restrict the possibilities of change in the control flow to reproducable events. The introduction of global effects here is not comparable to a cross-function or cross-module Go To, which could e.g. disorganize the ordering of stack frames, causing problems when returning from functions. This is not possible with reasonable AOP semantics.

Finally, Zetie's main recommendations for AOP developers can be summarized as follows:

1. Only allow a small group of highly skilled developers to write pointcuts and advice, and absolutely forbid side effects in advice code. Implement strict code inspections to ensure that advice code is well-behaved.

It is of course debatable that one should only permit code changes to be made by a small group of experts anyway. Neglecting this, this is exactly the way most extensive changes involving important structural changes in operating system kernels are performed anyway, if a deduction from the mailing lists of various open-source operating system kernel can be generalized.

With Linux development, for example, there exists a hierarchy of persons responsible for overall structure of several system components, where final changes to generic kernel topics or the introduction of API changes are being supervised by the Linux developer with the undoubtedly longest Linux kernel development experience, Linus Torvalds himself. Changes affecting major subsystems of the kernel, like networking, the SCSI layer or the VFS system, must be approved by a trusted person responsible for this subsystem who has to approve the integration of changes prior to their integration in the official Linux source tree. Smaller changes and extensions, like the addition of a new device driver for a USB device, can occur by a programmer further down in the decision matrix, as these changes are unlikely to affect overall kernel operation. Other open-source projects, like the various BSD-derived systems, replace the final decision by a single developer by a commitee decision by an elected group of talented, seasoned developers.

This distinction is especially important inside the kernel, as in theory every single change in an unimportant part of the kernel code may affect the system as a whole. Most Unix-derived operating systems traditionally run all of their kernel code in the same privilege level.

Other systems, like VMS and Windows-NT derived systems, have the ability to run certain parts of the kernel code at a lower priority level if the CPU supports several protection rings with staged priorities. More recent approaches using virtualization and micro kernels to create a system with more fine-grained privilege control are described in later chapters.

2. Be extremely suspicious of code that uses aspects. When baffling bugs occur, remind yourself that aspects are a likely cause. Test the problem code in an aspect-free test harness to isolate the cause.

Due to timing depencencies as well as asynchronous behavior in the kernel induced by external events (interrupts etc.), baffling bugs are effects kernel developers have to face on a regular basis anyways. In this case, aspects can actually make debugging this code *easier*, since the code that can be affected by the effects described can actually be factored out into a single aspect component and thus side-effects and timing dependencides can be localized in a far more efficient way compared to code that is scattered and tangled throughout the code base.

3. Carefully monitor the defect rate and the time needed to fix systems that use aspects. This is the only way to determine whether aspects are costing you more than they are worth.

This is as general an advice as can be when discussing the use of *any* technology for software development. Since AOP inside kernel code is not yet in common use – most AOP tools exist for object-oriented application code development – an evaluation phase certainly has to start as soon as the tools developed here, like the AspectC compiler, and in the rest of the AOP community are sufficiently mature and in widespread use. This thesis may, however, partly lay the foundation for evaluation of AOP in this context by providing tools and example applications for aspect-oriented software development inside a commonly used Unix-like kernel.

Overall, the Forrester report shows up some valid points about certain implementations of aspect-oriented systems. Since AOP is still a relatively young technology, methods as well as implementations are expected to overcome the limitations shown, especially when more semantic information is included in the weaving process. The points debated, however, can not generally be applied to aspect-oriented development methods. Especially when developing kernel code, experienced programmers used to handle crosscutting problems can gain benefits from using aspect-oriented approaches to code maintenance and development.

4.3 Using AOP Inside the Kernel

A complete overview of aspect-oriented software development is beyond the range of this thesis. A comprehensive overview of AOSD theory, implementations and applications can be found in [54]. The following sections contain an overview of AOP terms and methodologies appropriate for kernel development. Whenever necessary, special consideration is given to the programming environment inside the kernel code in addition to a general definition in an AOP context.

4.3.1 Join Points

Join points are those elements of the programming language semantics which the aspects coordinate with. Nowadays, there are various join point models around and still new under development. They heavily depend on the underlying programming language and AO language.

In a number of presently available AOP languages, a join point is a region in the dynamic control flow of an application. Thus, a join point can, for instance, represent

- a call to a method,

- execution of a method,
- the event of setting a field,
- the event of handling an exception

Join points can be picked up by an AOP program by using pointcuts to match on them. Depending on the pointcut language the AOP language provides, it may be possible to pick up more or less of those join points. Since join points are dynamic, it may be possible to expose runtime information such as the caller or callee of a method from a join point to a matching pointcut.

Many AOP languages implement aspect behavior by weaving hooks into the join point shadows, which is the static projection of a join point onto the program code.

4.3.2 Pointcuts

In most AOP languages, a pointcut is a predicate over dynamic join points, meaning that given a certain dynamic join point, a pointcut can either match this join point or not (at runtime). Another view of pointcuts is that they represent sets of join points. A pointcut may expose runtime information to a piece of advice.

4.3.3 Join Point and Pointcut Types

The expressiveness of an AOP tool's join point model determines the granularity of the join points available, and how those join points are matched. Each AOP tool offers a number of primitive pointcuts for matching join points. Some primitive pointcuts match only join points of a specific kind (for example, method executions). Other pointcuts can match any kind of join point, based on a common property of those points (for example, all join points in a certain control flow). The kinds of join points, and their specific pointcuts, that are relevant for use in a kernel context can be grouped as follows:

- **Invocation** – Points when methods and other code elements are called or executed
- **Initialization** – Points in the initialization of classes and objects
- **Access** – Points when fields are read or written
- **Exception handling** – Points when exceptions and errors are thrown and handled

In addition, the following categories of kindless pointcuts are supported:

- **Control flow** – Join points within certain program control flows
- **Containment** – Join points that correspond to places in the code contained within certain classes or methods
- **Conditional** – Join points at which a specified predicate is true

The main trade-off here is in expressiveness vs. simplicity. A more complete and fine-grained set of pointcuts allows more of the interesting points of program execution to be accessible to aspects. For example, an aspect related to persistence might need access to an object's initialization join points. But such completeness brings with it additional complexity and a steeper learning curve. Many programmers do not distinguish between calls and execution, and even fewer need to understand the subtleties of initialization. Also, many commonly used aspects are termed auxiliary, in that they are not tightly coupled to the application functionality. Common auxiliary aspects, such as monitoring and logging, typically make use of only coarse-grained pointcuts like method executions.

4.3.4 Advice Code

The code to be executed at the location of a join point shadow that is matched by a given pointcut description is called *advice code*. Advice code interacts with the primary functionality at function call level and can, in current implementations, be run *before*, *after* or *around* the call.

The usual semantics for executing advice code is to execute the advice as a separate function. This implies that the advice code has only limited access to the context of the current pointcut; all data structures and variables that are to be accessed have to be explicitly passed to the advice function.

A different approach not found in common use is inlining the advice code instead of treating it as a separate function. This permits the advice code to access more of the program state surrounding the join point shadow than a function invocation, but creates new problems as the compiler has to ensure that all variables not declared locally in the advice code or aspect have to be declared in the code surrounding each join point shadow.

4.4 Weaving

Weaving can be described as the process of coordinating aspects and non-aspects (i.e., the base code of a software system). In an aspect-oriented language, one part of weaving is ensuring that advice executes at the appropriate dynamic join points. Weaving can be done explicitly or implicitly, and can be done at a variety of times ranging from by-hand weaving when code is written, through compile-time, post-compile time and load time, up to runtime. The general weaving process is depicted in fig. 4.1.

4.5 Static AOP

Originally, the weaving process only took place at compile time. Using this approach, the advice code woven in is not modifiable once the program is compiled and linked. This method of applying aspects is named *static AOP*.

Static weaving, as shown in fig. 4.2, requires the analysis of the pointcuts defined in the various aspect specifications before compiling the base code. The aspect compiler

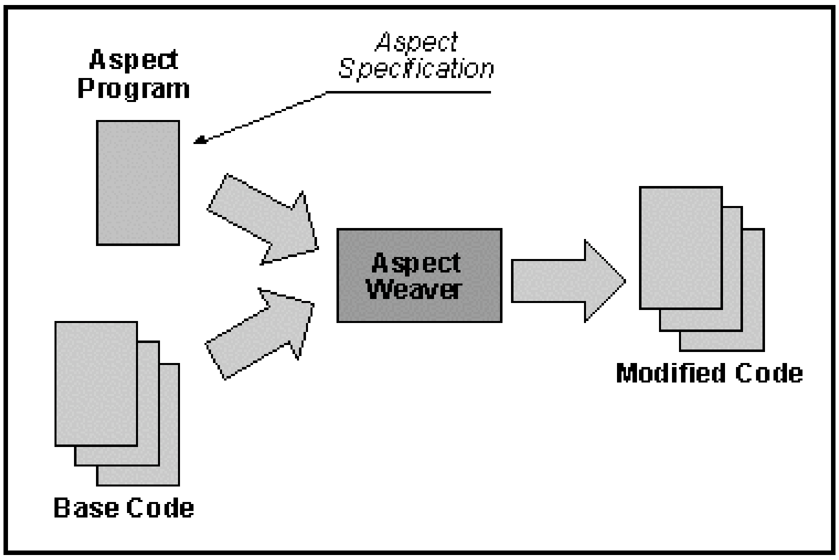


Figure 4.1: The weaving process

extracts sets of join points from the pointcut definitions given and identifies these so-called join point shadows in the code. At the identified locations, the source code is modified by augmenting the original code according to the advice code given. Additionally, the aspect compiler inserts calls to support routines in the code base, e.g., routines that keep track of the function call stack in order to implement `cflow` pointcuts.

Like early object-oriented compilers, the first static AOP compilers worked as a sort of preprocessor that generated code in the language of the base system that had to be compiled using the regular compiler for the basis language. Current compilers integrate the AOP processing in the compiler front end to achieve faster compile times.

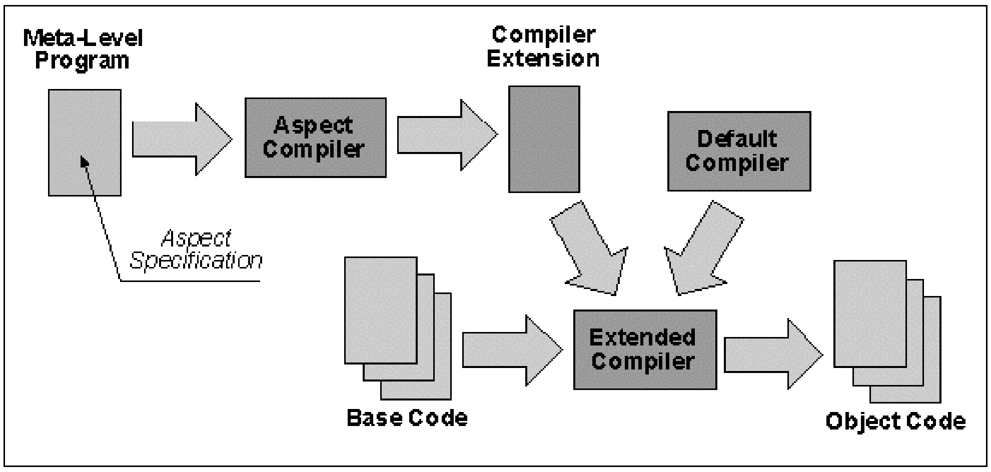


Figure 4.2: Static weaving

Static AOP only has limited use in operating system environments. Especially in kernel space, code that is executed will belong to different (user mode) process contexts, so aspects intended to modify only process-context related information in kernel mode would have to employ expensive checks to ensure only the correct process is interfering with the advice code.

4.5.1 Link-time Weaving

Static weaving can also take place in the linking stage of a system instead of the compile stage, e.g. in the PIN system [92].

Link-time weaving moves the renaming of functions that occurs in a preprocessing stage in compiler-based static weaving to the link stage. Here, the linker renames functions which match a pointcut description and links in the appropriate advice code using the original function names. The code calling the functions, then, gets redirected to the advice code which, in turn, can call the advised function using the renamed identifier.

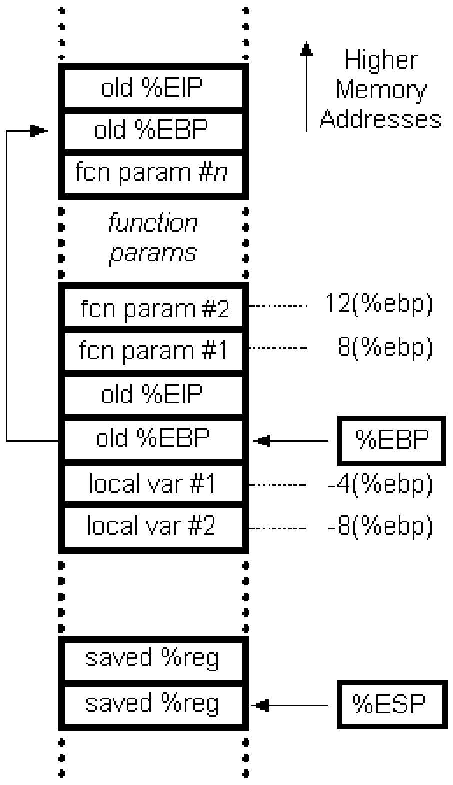
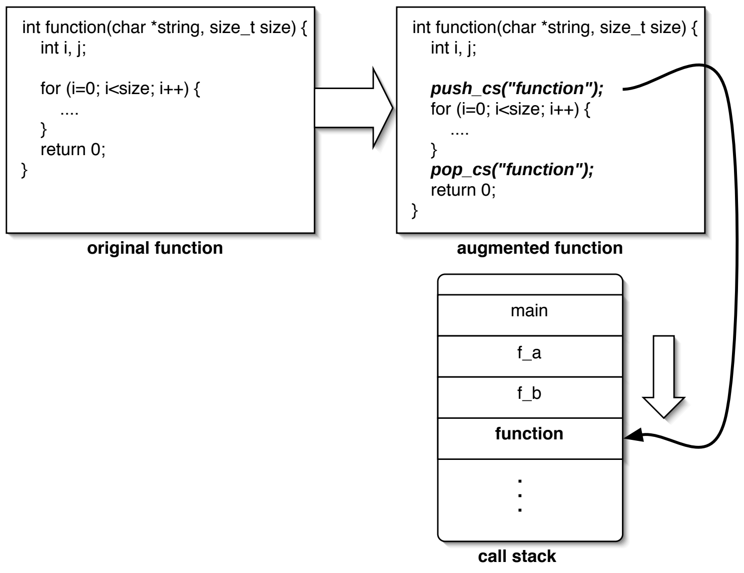


Figure 4.3: Call stack and stack walking

4.5.2 Not Quite That Static

Static weaving can provide a limited amount of dynamic context for aspects. One of the pointcut types requiring dynamic analysis of the program context is `cflow`. cflow is applied to a pointcut description (in the most simple case, a function call) and captures related pointcuts if and only if these pointcuts occur when the stack context of the current execution indicated that the code currently executed is called from a function matching the pointcut description given to cflow.



```
int function(char *string, size_t size) {
    int i, j;

    for (i=0; i<size; i++) {
        ....
    }
    return 0;
}
```

```
int function(char *string, size_t size) {
    int i, j;

    push_cs("function");
    for (i=0; i<size; i++) {
        ....
    }
    pop_cs("function");
    return 0;
}
```

Figure 4.4: Separate stack for cflow

The implementation of `cflow` requires runtime support, usually in the form of a small library. For cflow to work as required, the program has to analyze the state the running program currently is in related to the momentary nesting of functions. This analysis can take place in one of two ways:

One way is to provide runtime support to perform a call stack analysis. The current stack frames are analyzed beginning from the current stack pointer upwards until either a function matching the cflow description is found or the search arrived at the bottom of the stack, i.e., the main function of the program. Fig. 4.3 shows a typical call stack on a x86 CPU. In order to walk the stack, for each hierachy level the saved `%ebp` register has to be retrieved, the `%eip` value (instruction pointer) of this function retrieved and compared with the symbol table of the program to determine which function it represents. If no match is found in the current level, the procedure repeats for the next higher level.

Another approach implements a separate stack that represents the function call hierarchy. In order to maintain this stack, code has to be inserted at the entry and exit of functions that push and pop information on and from this separate stack, as shown in fig. 4.4. This method makes searching more efficient, because walking the call stack is simplified at the expense of stack operation overhead at function entry and exit.

Another dynamic property in static AOP is expressed by the `proceed` statement, which is used to call the original function captured by the current `around` join point from inside the advice code. Proceed is especially difficult to implement since on the one hand it can be called zero, one or even more times in an advice function and on the other hand `proceed` can match one of several functions with differing return types, depending on the context. In these cases, the AOP compiler must ensure that either the return type is identical for all functions matched or the return type is ignored.

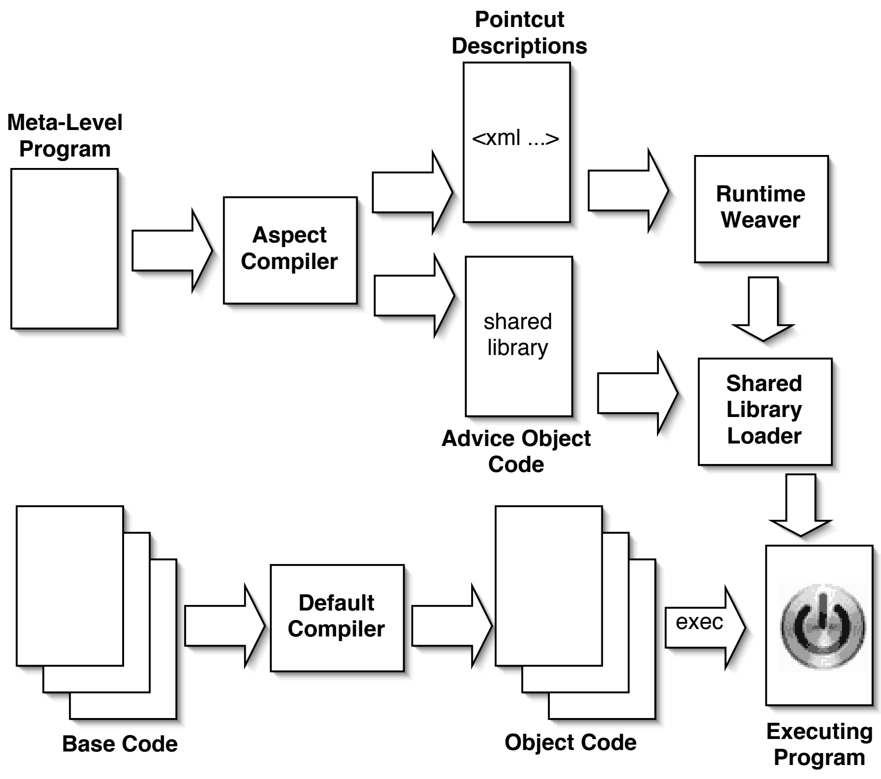


Figure 4.5: Dynamic weaving

4.6 Dynamic AOP

Dynamic AOP, on the contrary, moves the weaving process to the run time of the executable. This enables the programmer to activate and deactivate aspects at runtime and optionally also change the dependency of aspects from other aspects at runtime. In dynamic AOP, the task of the AOP compiler is reduced to check the validity of the aspect code, convert the pointcut descriptions given into a format appropriate by the dynamic weaver and compile the advice code to dynamically loadable libraries.

The result of the compilation process, shown in fig. 4.5, is a pointcut description, e.g. in XML, and a set of related shared library modules containing the advice code to be woven. The advice code modules are delivered with the executable along with the pointcut descriptions. This method also allows the aspect code written after the system code was developed without recompiling and – if sufficient information on the program structure is available – to develop aspects for a program for which source code is not available.

Dynamic AOP shifts the deployment phase to the runtime of the software. This has the implication that aspects can be added and removed to the system on demand, creating a much more flexible environment for modifying and exending a system. The component responsible for the addition and removal of aspects is the dynamic weaver. The weaver takes a pointcut description and shared library modules and modifies the original program by inserting hooks to the advice code at the join point shadows found.

4.7 Summary

Despite being criticized by some analysts, aspect-oriented programming is a useful tool to overcome the crosscutting problems inherent in operating system code. Aspect-orientation, however, is no single concept; it rather consists of many facets, not all of which are applicable to the procedural environment prevalent in kernel code. The following chapter shows how the set of aspect-oriented approaches described in this chapter can be integrated with the most common systems programming language, “C”.

5. A C Extension Supporting AOP

"The most important thing in the programming language is the name.
A language will not succeed without a good name.
I have recently invented a very good name and now
I am looking for a suitable language."

— Donald Knuth

5.1 Introduction

While early research in AOP has concentrated on extending object-oriented [65, 123] as well as functional [82] languages, maintaining and supporting existing large software systems implies that the developer has to work on systems implemented in a procedural language. While many legacy application systems are implemented in Fortran or COBOL, most procedural system software code is implemented in C.

An extension of C to support aspects is required to use AOP inside the kernel. Since no working AspectC compiler and no formal specification of the language exist so far, an AspectC compiler implementation based on a concise grammar for AspectC (extrapolated from the examples in Coady's work [32]) has been developed as part of this thesis.

Various approaches for implementing an AspectC compiler are discussed. The prototypical AspectC compiler, based on source code analysis and rewriting tools, is then presented and examples of static weaving using the AspectC compiler are shown.

Finally, a set of experimental extensions to AspectC is presented that support an extended notion of local storage for aspects, and explicit initialization functions as well as support for dynamic AOP are presented.

5.2 AspectC

The C programming language is not only used for maintaining legacy systems, but it is still a popular choice for developing new applications. Statistics gathered by the open source support site SourceForge [122] show that of the 85649 projects registered at SourceForge on June 14th, 2005, 14989 projects are implemented in C. This number is in the same range as the number of projects using Java (15220) or C++ (15616) and comprises more than 17% of all projects.

A first approach to exploring the usefulness of an AOP-extended version of C was published in Coadys Ph.D. thesis [33], in which an approach to use an aspect-extended version of C, AspectC, is defined and used for solving example crosscutting problems. The language structure of AspectC resembles the aspect, pointcut and advice code definitions supported by AspectJ [65]. These first examples of AspectC usage ware based on the state of AOP research in about 2000 and supported only static aspects. However, neither a complete formal definition nor a version of an AspectC compiler was ever released.

While there is no proposed standard for AspectC at the moment, an interest group for AspectC was formed at AOSD 2005 in Chicago, which will define a definitive standard for AspectC, based on comparable constructs in AspectJ and AspectC++ adapted to procedural systems. This chapter is one of the first results of the AOSD 2005 discussions.

Even with the envisioned increased use of domain-specific languages (DSLs) in operating system development, AspectC can still prove to be an important building block for future systems, since domain-specific languages might be translated into AspectC instead of directly compiling to native or byte code. A suitable compiler and runtime implementation for these DSLs will take care of the problems inherent in C, such as weak type checking, dereferencing of invalid pointers and missing array bounds checking.

5.3 Supported Join Points and Pointcuts

AspectC in its current form provides support for *function call* and *function execution* join points. While a function call join point annotates the code of the caller, the function execution join point annotates the code of the callee.

The following types of pointcuts, derived from the list of useful join point and pointcut types for procedural languages in the previous chapter, are supported in AspectC:

call (function-prototype-pattern)

`Call` selects all function call join points, where the signature matches *function-prototype-pattern*.

execution (function-prototype-pattern)

`Execution` selects all function execution join points, where the signature matches *function-prototype-pattern*.

within (function-prototype-pattern)

withincode (function-prototype-pattern)

`Within` and `withincode` select all join points, in which the code executed is defined inside a function that matches the *function-prototype-pattern*.

args(type or id, ...)

`Args` selects all join points with arguments that are instances of *type* or *typeof(id)*.

cflow(pointcut)

`Cflow` selects all join points occuring in the control flow of a join point *P*. Here, P can again be a pointcut specification or reference.

! pointcut

This construct selects the inverse set of join points to the set described by *pointcut*.

pointcut0 && pointcut1

pointcut0 || pointcut1

(pointcut)

These pointcuts combine the set of join points described by the AND- respectively OR-combination of the set of join points matching the two pointcut descriptions pointcut0 and pointcut1. Expressions may also be nested using brackets.

5.4 A Grammar for Static AspectC

This grammar definition for AspectC is the first formal definition of the AspectC language extensions to the C standard. Formerly, the only sources for the AspectC syntax were examples from [33], [34], and [32]; these, however did not demonstrate the complete language specification for AspectC and were inconsistent in the syntax used. For this formal specification, the existing examples were analyzed, unified and formalized, with missing parts added from similar constructs in AspectC++ and AspectJ.

AspectC is designed as a superset of ANSI C with support for C99 and GNU-C extensions. As such, the structure of the base language remains nearly unchanged – with the exception of *proceed* in advice code functions, which is discussed below.

5.4.1 Aspect Specification

Aspects are specified in separate AspectC source files with a proposed common file extension of `.ac`. Each file may contain one or more *aspect definitions* as follows:

```
aspect(name) {
   /* local storage declarations */

   /* pointcut definitions */

   /* advice code definitions */
}
```

Listing 5.1: Aspect definition

5.4.2 Grammar

The grammar of AspectC extends the regular ANSI C grammar at the top level. The top-level grammar definition for `external-declaration`, which defines function declarations, is extended by a case for a top-level `aspect` definition. For the sake of clarity, the grammar shown here omits the actions of the parser and only gives the AspectC grammar extensions in EBNF:

```
external-declaration:
      aspect-definition

aspect-definition:
      ASPECT identifier { aspect-decl-list }

aspect-decl-list:
      aspect-decl-item
|     aspect-decl-list aspect-decl-item

aspect-decl-item:
      pointcut-definition
|     advice-definition

pointcut-identifier:
      identifier

pointcut-definition:
      POINTCUT pointcut-identifier ( parameter-type-list-opt ) : pointcut-expr ;

advice-definition:
      BEFORE ( parameter-type-list-opt ) : pointcut-expr compound-statement
|     AFTER  ( parameter-type-list-opt ) : pointcut-expr compound-statement
|     declaration-specifiers AROUND around-specifier-opt
              ( parameter-type-list-opt ) : pointcut-expr compound-statement

around-specifier:
      RETURNING ( parameter-declaration )

pointcut:
      pointcut-identifier ( aspect-parameter-list-ellipsis-opt )
|     pointcut-specifier ( parameter-declaration )
|     ARGS ( parameter-type-list-opt )
|     CFLOW ( pointcut )

pointcut-specifier:
      CALL
|     EXECUTION
|     WITHINCODE

pointcut-expr:
      pointcut
|     ! pointcut-expr
|     ( pointcut-expr )
|     pointcut-expr && pointcut-expr
|     pointcut-expr || pointcut-expr

identifier:
      id
|     ident_regexp

type-specifier:
      type_regexp
```

```

aspect-parameter-decl:
        parameter-declaration
|       identifier
|       dotdot

aspect-parameter-list:
        aspect-parameter-decl
|       aspect-parameter-list , aspect-parameter-decl

aspect-parameter-list-ellipsis:
        aspect-parameter-list
|       aspect-parameter-list , ...

direct-declarator:
        direct-declarator ( parameter-type-list )
|       direct-declarator ( identifier-list-opt )
|       direct-declarator ( aspect-parameter-list-ellipsis-opt )
```

5.5 A Real-World AspectC Example

To illustrate the use of AspectC, the following listing shows a real-world example using static aspects to solve a crosscutting concern in file system prefetching, adapted from [33]:

```
aspect sequential_mapped_file_prefetching {
    pointcut vm_fault_cflow( vm_map_t map ):
        cflow( call( int vm_fault( map, .. )));

    pointcut ffs_read_cflow( struct vnode* vp, struct uio* io_info,
                             int size, struct buff** bpp ):
        cflow( call( int ffs_read( vp, io_info, size, bpp )));

    /* plan the prefetching and allocate the pages */
    before( vm_map_t map, vm_object_t object, vm_page_t* pagelist,
        int* length, int faulted_page ):
        call( int vnode_pager_getpages( object, pagelist,
                                        length, faulted_page ))
        && vm_fault_cflow( map )
    {
        if ( object->declared_behaviour == SEQUENTIAL ) {
            vm_map_lock( map );
            plan_and_alloc_sequential_prefetch_pages( object, pagelist,
                                                      length, faulted_page );
            vm_map_unlock( map );
        }
    }

    /* divert to ffs_read */
    int around( vm_object_t object, vm_page_t* pagelist,
                int* length, int faulted_page ):
        call( int ffs_getpages( object, pagelist, length, faulted_page ))
    {
        if ( object->behaviour == SEQUENTIAL ) {
            struct vnode* vp = object->handle;
            struct uio* io_info = io_prep( pagelist[faulted_page]->pindex,
                                           MAXBSIZE, curproc );
            int error = ffs_read( vp, io_info, MAXBSIZE, curproc->p_ucred );
            return cleanup_after_read( error, object, pagelist,
                                       length, faulted_page );
        } else {
            proceed ( object, pagelist, length, faulted_page );
        }
    }

    /* page flip buffer pages */
    after( struct uio* io_info, int size, struct buf** bpp ):
            call( t%%% block_read(..) )
            && vm_fault_cflow(..)
            && ffs_read_cflow( struct vnode*, io_info, size, bpp )
    {
        flip_buffer_pages_to_allocated_vm_pages( (char *)bpp->b_data, size,
                                                     io_info );
    }
}
```

Prefetching is an optimization designed to reduce the overhead of fetching pages from disk by acquiring additional pages that may be required in the near future. Since prefetching is a heuristic method, it may only take place when it is cost effective to do so. The virtual memory system thus suggests pages for prefetching, but the file system decides whether or not to actually get them. The inherent structure of prefetching is shaped by specific execution paths that retrieve pages from disk. Prefetching crosscuts virtual memory and file systems, coordinating high-level allocation and low-level de-allocation of prefetched pages.

The code presented here defines two pointcuts – `vm_fault_cflow` and `ffs_read_cflow`. `vm_fault_cflow` matches all join points in the control flow of the function `vm_fault`, whereas `ffs_read_cflow` does the same for the function `ffs_read`. Both provide the necessary parameters to the advice code.

Based on these pointcut definitions, three advice functions are defined. The first one is a `before` advice that matches all join points matched by the `vm_fault_cflow` pointcut if and only if they occur in a call to the function `vnode_pager_getpages`.

The second advice defines a `call` join point directly and matches all calls to the function `ffs_getpages`, defining an `around` advice for these. Here, for sequential access mode, a new piece of code that handles the prefetching is introduced, whereas in the case of random access simply the originally matched function is being called via `proceed()`.

Finally, the third advice uses the most complex pointcut description. In addition to matching join points that match the two previously defined pointcuts `vm_fault_cflow` and `ffs_read_cflow`, the third condition applied is that this set of join points is further restricted to such join points that are within a call to a function described by the regular expression `t%%%block_read()`. This expression matches all functions having a name beginning with the letter "t" and ending in "block_read". Like in the previous case, this implements an `around` advice. The difference in this case is that the originally matched function is *never* called, it is rather replaced by a call to the function `flip_buffer_pages_to-`

_allocated_vm_pages.

5.6 Approaches to Compiling Aspect Code

One of the goals in developing an AspectC compiler was to create a working compiler in a short amount of time, so the prototypical AspectC compiler described here was constructed reusing as many open source components as possible.

An important goal when construction the compiler has been to support not only standard ANSI C code, but real-world systems. In order to compile the kernel of an open source operating system, support for GCC extensions has to be provided.

Three approaches to creating a first version of an AspectC compiler are discussed below, citing the advantages and disadvantages of each method.

5.6.1 GCC Front End

Since AspectC aspects may contain arbitrary C code in their advice code definitions, the AspectC grammar had to be an extension of a working grammar. ANSI-C grammars in yacc/bison [79] or ANTLR [103] format can easily be found, however, fully working grammars with support for GNU C extensions are surprisingly hard to come by.

The primary source for a grammar, of course, is the GCC compiler collection itself. The original GCC grammar, however, requires support code from the GNU C libraries and intense understanding of the GCC-internal data formats and algorithms. Thus, using this grammar as a basis would have sharply contradicted the approach of creating a compiler in a short amount of time.

5.6.2 XML-based AOP Tools

A different approach to weaving static aspects is to use the compiler intermediate representation of the compiled program in a structured form, the abstract syntax tree (AST). GCC provides the option of creating a dump of the internal AST representation, which in turn can be converted to a processable XML description by tools like GCC-XML [93]. A simple AST for the expression `a+b*c` is shown in fig. 5.1.

Providing the aspect code as well as the code the aspects are to be woven into in an intermediate XML format has the advantage that weaving the aspects can now be assisted by the use of standard XML query and transformation tools. Tasks like resolving method names that are specified using regular expressions in a pointcut can be delegated to the XML query tool like XQuery [25] or XSLT [14] and weaving the advice code itself is done by an XML transformation tool.

Approaches to using XML-based transformation tools can be found in [47] and [117]. However, the XML approach only handles the problem of actually weaving in advice code; the parsing of aspect code itself is not solved.

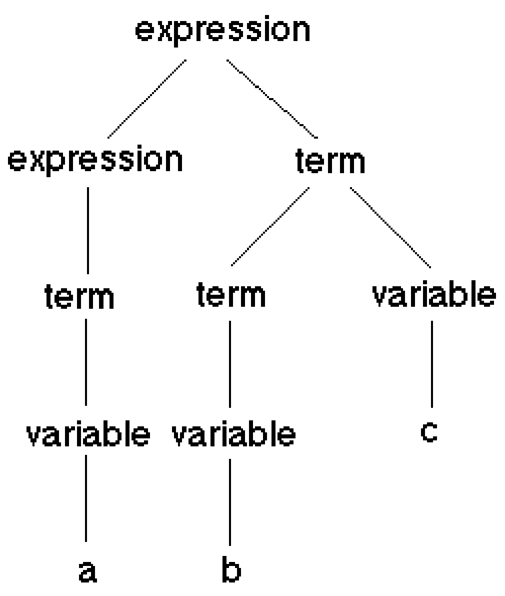


Figure 5.1: A simple abstract syntax tree

5.6.3 Source Code Transformation

The third approach that has been considered is using a source code transformation tool to implement the AspectC compiler. Code transformation tools analyze the structure of a program using a parser and generate an augmented version of the program source. This augmentation is the output of a static weaver, so a transformation tool that is able to handle GNU C syntax would also be able to do most of the weaving.

Two implementations of a source code translation framework are available that come with a complete GNU C grammar: cgram [60] and CIL [100]. Both provide similar functionality, but use quite different approaches. While cgram uses a Java-based parser and transformation engine, CIL makes use of the parsing capabilities of the OCaml language and an existing C parsing toolset to implement C code transformation. Both tools provide support for real-world code. Cgram is used by Intel to transform the Linux kernel source code into a form parseable by the high-performance Intel C compiler, icc, whereas CIL is used in various large-scale code transformation and code security applications applied to projects like the Linux kernel and the Apache web server.

5.7 The Static AspectC Compiler

Since no implementation of an AspectC compiler existed so far, a prototypical compiler was created in the context of this thesis. The focus of this first implementation was to create a working compiler in the shortest possible amount of time by using existing open source tools for code analysis and transformation. In addition to parsing standard C syntax, the use of a code transformation tool for real-world kernel source code implies that all extensions used in the source have to be supported in the tools used.

The CIL infrastructure from UC Berkeley was finally chosen as the basis for the AspectC compiler prototype since it has good documentation, runs out of the box and provides a large-scale real-world example of C code transformation, while supporting most GNU C extensions, demonstrated by its use in analyzing Linux kernel source code.

5.7.1 The CIL Infrastructure

CIL (C Intermediate Language) uses a high-level representation of the C language that enables its user to easily analyze C programs as well as to perform source-to-source transformations, performed by a set of tools written in OCaml [111] and Perl [140].

On the one hand, CIL resides on a lower level than the commonly used abstract syntax trees, on the other hand it manages to retain a higher level than intermediate languages commonly used in compilers and transformation toolkits as CIL maintains type information as well as a close relationship with the original source program.

CIL provides a large set of code transformation tools along with the base infrastructure. Among these tools are OCaml programs to log function calls, replace all `return` instructions in a function with a single function exit point, detect stack overflows and generate a function call graph. Some of these tools, like the function call logger, are directly applicable to weaving AspectC.

Along with CIL, a number of valuable support tools are available. First, there is a compiler driver that integrates a CIL preprocessing and transformation stage which behaves just like the original gcc compiler driver. Second, CIL provides a whole program merger, which is able to merge all files compiled into a single source file, thus simplifying the weaving step for AspectC.

CIL has been tested exensively by compiling the Linux kernel, the GIMP image processing toolkit and the SPECint 95 benchmarks, so the inelegant but legal constructs showing up in kernel-level code are supported. Based on CIL, CCured [101] is an infrastructure that analyzes a C program to determine the smallest number of run-time checks that must be inserted in the program to prevent all memory safety violations. The resulting program is memory safe, meaning that it will stop rather than overrun a buffer or write over memory that it should not touch. While not directly useful for the AspectC compiler implementation, the CCured source code provided many hints how to handle special transformation cases using CIL.

Thus, CIL was selected as the most appropriate solution available to base the AspectC compiler on. Examples for code transformation using CIL are given in the following sections.

5.7.2 AspectC Compiler Structure

The AspectC compiler works as a precompiler, similar to early C++ and ObjectiveC compilers [127, 83]. The overall compilation procedure is shown in fig. 5.2.

The AspectC compiler implementation is based on a version of the grammar shown in

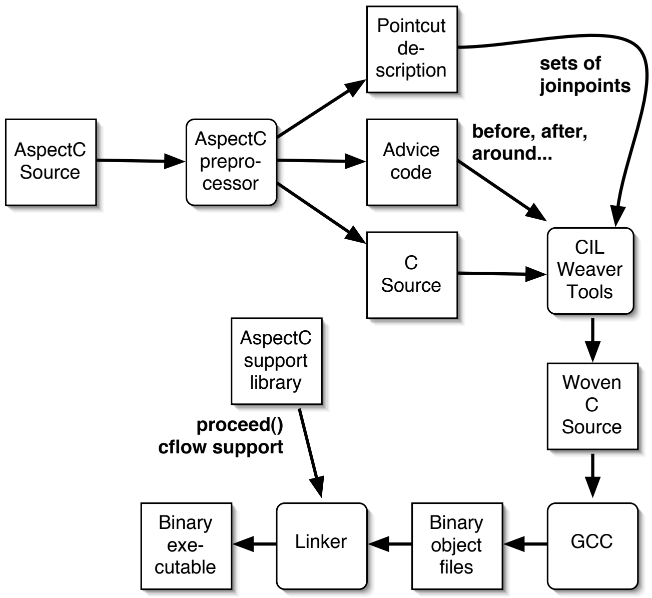


Figure 5.2: Compiling AspectC using CIL

section 5.3.2, adapted to run with the OCaml-based *ocamlyacc* parser tool CIL is based on.

After parsing the aspect definitions, a set of pointcut descriptions is generated which then control the various weaving scripts adapted from the examples provided by CIL. Along with the pointcuts, the advice code for each aspect to be woven is separated, leaving the pure C source code the aspects are to be woven into as a third component. These components are handed to the CIL-based weaver tools, which search the source for the pointcuts described using OCaml-provided regular expression matching and then transform the C source using separate tools for `before`, `after` and `around` advice.

The result of this step is C source code with calls to advice code statically woven into. This code is then handed to the regular gcc compiler to create a set of object files. These binary object files, in turn, are then handed to the linker. Since AspectC requires support functions for the dynamic properties of the woven aspects to be available, the linker adds the AspectC support library, providing functions to implement `proceed()` and support for maintaing the function call stack for `cflow()`. From the object files, the AspectC support library, and the regular system and program libraries, the final woven binary executable file is created.

5.8 Weaving Static AspectC

The weaving process is the most interesting piece of the AspectC compiler. In order to illustrate the results of the weaving process, examples for each advice type are given in the following sections. For each advice type (before, after, around), a piece of original code with an aspect definition is presented first, followed by the C code with the aspect statically woven into.

5.8.1 *Before* Advice

The source code shown in listing 5.2 consists of the `main` function and two additional functions, `foo` and `bar`. At the top of the source, an aspect `test` is defined that defines a pointcut `IdPointcut0` matching calls to functions foo and bar. The advice function implements a simple logging advice that generates a messages whenever one of the functions in the pointcut is called.

```
aspect Test {
    pointcut IdPointcut0 (int x, int y):
        call( int foo (x, y, ...) ) || call( void bar (x, y) );

    before(int k, int l): IdPointcut0(k, l) {
        fprintf (stderr, "DEBUG:␣calling␣(foo|bar)(%d,␣%d,␣...)\n", k, l);
    }
}

int foo (int a, int b, int c)
{
    doSomethingFoo ();
}

void bar (int a, int b)
{
    doSomethingBar ();
}

int
main ()
{
    bar (1, 2);
    doSomethingMain ();
    bar (3, foo (4, 5, 6));

    exit (0);
}
```

Listing 5.2: AspectC source code

The source code shown above is transformed using CIL into the code shown in listing 5.3. For each join point matched, the original function calls to `foo` and `bar` are replaced by calls to the advice code functions, named `bar_test_before` and `foo_test_before`. As required by the `before` advice semantics, the advice functions in turn call the original functions matching the respective pointcut.

```
static int foo__Test__before__0 (int, int, int);
static void bar__Test__before__0 (int, int);

int
foo (int a, int b, int c)
{
   doSomethingFoo ();
}

void
```

```
bar (int a, int b)
{
   doSomethingBar ();
}

int
main ()
{
   bar__Test__before__0 (1, 2);
   doSomethingElse ();
   bar__Test__before__0 (3, foo__Test__before__0 (4, 5, 6));
   exit (0);
}

static int
foo__Test__before_0 (int _p0, int _p1, int _p2)
{
   int k = _p0;
   int l = _p1;
   {
      fprintf (stderr, "DEBUG:␣calling␣(foo|bar)(%d,␣%d,␣...)\n", k, l);
   }
   return foo (_p0, _p1, _p2);
}

static void
bar__Test__before_0 (int _p0, int _p1)
{
   int k = _p0;
   int l = _p1;
   {
      fprintf (stderr, "DEBUG:␣calling␣(foo|bar)(%d,␣%d,␣...)\n", k, l);
   }
   bar (_p0, _p1);
}
```

Listing 5.3: Source code after weaving `before` advice

5.8.2 *After* Advice

Whereas the location of the join point shadow for weaving `before` advice is unambiguous, weaving `after` advice is more complex, since a function may have several exit points using the `return` statement (see listing 5.4). Here, the transformation tools provided by CIL already provide a tool to coalesce all returns from a function into a single return statement by inserting jump instructions from the original locations of all return statements. This is done using the C `goto` instruction – however, in this case this is not critical, since the goto changes code flow in a strictly controlled way. Thus, Dijkstra's critique of goto [40] does not apply here.

```
int f(int x) {
        if (x>42)
                return x-31;
        else if (x<4)
```

```
                return x+33;
        else
                return x*3;
}
```

Listing 5.4: C function with several return statements

This code is transformed into code using exactly one return, so the `after` advice can be easily inserted by the weaver before line 17 in listing 5.5.

```
1  int f(int x )
2  { int __retres ;
3   {
4    if (x > 42) {
5      __retres = x - 31;
6      goto return_label;
7    } else {
8      if (x < 4) {
9        __retres = x + 33;
10       goto return_label;
11     } else {
12       __retres = x * 3;
13       goto return_label;
14     }
15   }
16   return_label: /* CIL Label */
17         return (__retres);
18  }
19 }
```

Listing 5.5: Transformed C function with one single return statement

5.8.3 *Around* Advice

Weaving `around` advice works similar to weaving `before` advice. The difference to before is that the code of the original function is not called unconditionally at the end of the advice code, but rather inside the advice function using the `proceed()` statement. Previous examples showed that in some cases, the original function is not called at all. Weaving around advice replaces all occurences of `proceed` in advice code with a call to the AspectC support library described in the following section.

```
aspect Test {
    pointcut pc0 (int x, int y):
        call( int foo (x, y, ...) );

    around(int k, int l): pc0 (k, l) {
        if ( k < 5)
            proceed(10, l, 5);
        else if (k >= 5 && k < 20)
            proceed(k+5, l-1, 6);
    }
}

int foo (int a, int b, int c)
```

```
{
    doSomethingFoo ();
}

int
main ()
{
    foo (4, foo (5, 1, 2) , 6);

    exit (0);
}
```

Listing 5.6: Example for `around` advice

Listing 5.6 shows source code implementing `around` advice. The pointcut `pc0` captures the call of the function `foo`; around the function invocation, an advice function is woven. This advice function checks the first input parameter of the original function `foo`. If this parameter is less than 5, the original function is called using `proceed` with the first and third parameter changed to 10 and 5, respectively. The original second parameter remains unaltered. If, however, the first parameter's value lies between 5 and 20, inclusive, the original function `foo` is called with a differing set of parameters: the original first parameter is incremented by 5, the second is decremented by 1, and finally the third parameter is set to 6.

In all other cases, i.e. for values of k greater than 20, the original function *is not called at all*, since no `proceed` statement is executed by default.

```
foo__Test__ around(int _p0, int _p1, int _p2) {
        if ( _p0 < 5)
            _ac_lib_proceed("foo", "int", "int:int:int", 10, _p1, 5);
        else if (_p0 >= 5 && _p0 < 20)
            _ac_lib_proceed("foo", "int", "int:int:int", _p0+5, _p1-1, 6);
    }
}

int foo (int a, int b, int c)
{
    doSomethingFoo ();
}

int
main ()
{
    foo__Test__around__0 (4, foo__Test__around__0 (5, 1, 2) , 6);

    exit (0);
}
```

Listing 5.7: Woven `around` advice

The resulting C code after running the AspectC compiler is shown in listing 5.7. The advice function is now called `foo__Test__around` (composed of the original function, the aspect name, and the advice type). This function uses name mangling to obtain its unique set of parameters mapped from the original passed values. The original function `foo` is called depending on the value of the first parameter, now called _p0, like described above. The proceed function is implemented by calling the library function _ac_lib_proceed. In addition to the parameters passed to proceed specified in the advice code, three additional parameters are passed as C strings which contain the original function name and the signature of this function, separate for input and output parameter(s).

5.9 Support Library

To create an executable program with woven aspects, the support library for AspectC is linked in addition to the regular system and program libraries of a software system. This library provides the support functions required for analyzing dynamic context. The two major tasks implemented here are proceed() support to select the correct function to call from inside an advice and cflow() support, which provides functions for keeping track of the current function call stack that is evaluated in the cflow pointcuts.

Support for proceed is implemented in the function _ac_lib_proceed. This function takes three parameters in addition to the original parameter list of the function at the join point. While the first three parameters are fixed (the C strings containing the function name and input/output parameter signatures), the remaining parameters are of arbitrary number, depending on the particular join point. Thus, these parameters have to be processed using the C library's varargs functionality. A simplified version of this function's implementation is shown in listing 5.8.

```
#include <sys/varargs.h>

void *_ac_lib_proceed(char *_f_name, char *_out_signature,
                               char *_in_signature, va_alist) {
    _ac_lib_stackframe *proceed_stack = *_ac_lib_current_stack;
    va_list args;
    void *param, *retval;
    int argno = 0;

    if (! _ac_lib_check_valid_function(_f_name)) {
        _ac_lib_runtime_error(_AC_LIB_LOG,
                "Nonexistant␣join␣point:␣%s", _f_name);
        return((void *)0);
    }
    if (!_ac_lib_check_signature(_f_name, _out_signature, _in_signature) {
        _ac_lib_runtime_error(_AC_LIB_LOG,
                "Invalid␣signature␣for␣join␣point:␣%d", _f_name);
        return((void *)0);
    }

    va_start(args);

    while (param = va_arg(ap, _ac_lib_lookup_signature(_in_signature)) != NULL) {
        _ac_lib_push(proceed_stack, param);
        args++;
    }
    va_end(args);

    retval = _ac_lib_call_function(_f_name, proceed_stack);
    return(_ac_lib_mangle_retval(retval));
}
```

Listing 5.8: Implementation of the `proceed` support function

The `proceed` support first checks for the validity of the function name and signatures passed. If any of these do not match, a kernel error is logged and no further action taken. However, the kernel is not halted – rather, the original code is permitted to continue without further execution of the advice code. If the checks succeed, the rest of the parameters passed is retrieved using the `varargs` macros and a new stack frame is constructed from these parameters and the function at the join point is called. After the call, the return value is retrieved and returned following a validity check.

The second important function of the support library implements the functionality necessary for `cflow`. The semantics of a cflow pointcut is that it captures the set of join points that occur in the *control flow* of the given function. In other words, the set of join points that occur in conjunction with `cflow` in a pointcut description are only active when the function passed as parameter to `cflow` occurs on a higher level in the current function call hierarchy than the current join point.

Since this behavior is dynamic and unpredictable, runtime support is required to analyze the function call hierarchy. This support function, `_ac_lib_cflow`, is shown in listing 5.9.

```
int _ac_lib_cflow(char *_f_name, char *_out_signature, char *_in_signature) {
    _ac_lib_stackframe *cur_stack = *_ac_lib_current_stack;
    char *current_function;

    if (! _ac_lib_check_valid_function(_f_name)) {
        _ac_lib_runtime_error(_AC_LIB_LOG,
                "Nonexistant␣join␣point:␣%s", _f_name);
        return((void *)0);
    }
    if (!_ac_lib_check_signature(_f_name, _out_signature, _in_signature) {
        _ac_lib_runtime_error(_AC_LIB_LOG,
                "Invalid␣signature␣for␣join␣point:␣%d", _f_name);
        return((void *)0);
    }
    /* Walk the stack from bottom to top, return success
     * when function name matches */
    while ((current_function = _ac_lib_get_stackframe(cur_stack).f_name
            != (char *)0) {
        if (strncmp(current_function, _f_name,
            min(strlen(current_function), strlen(_f_name)) == 0) {
                /* strncmp == 0 -> strings match */
                return _AC_LIB_TRUE;
        }
        cur_stack = _ac_lib_stack_walk_up(cur_stack);
    }
    return _AC_LIB_FALSE;
}
```

Listing 5.9: Implementation of `cflow` support

Similar to the `proceed` support, first the function name and signatures are checked. After this, the stack is walked from bottom to top, comparing the function name of the current stack level with the function name specified in the `cflow` call. If the compare matches, a status code indicating success is returned. If the top of stack is reached without a match, negative acknowledge is generated.

In future versions of AspectC, the support library will include functionality to provide more dynamic context to advice code like introspection capabilities and temporal join points.

5.10 Extensions to the AspectC Syntax

The original AspectC syntax proposed in Coady's examples [33] only provides basic AOP functionality. The development of AOP-based code for kernel space usage, however, requires some additional functionality, while some other functionality is provided as convenience functions to the programmer. The following paragraphs describe novel extensions to AspectC that provide support for dynamic aspects as well as local storage and initialization functions for aspects that augment aspect modules with a functionality similar to first-class objects. An example for the use of local storage to provide temporal dependencies for aspects using static aspects and an optimized version of this code using dynamic AOP is presented as a use case.

5.10.1 Init Functions for Aspects

In many circumstances, aspect code requires initialization of internal parameters or execution of startup functions. In standard AspectC, this can be handled by creating `before` advice for the `main` function. Introducing a separate syntactical construct for aspect initialization, however, clears up the semantics of this function and make aspects even more similar to objects, which usually provide a constructor method for object initialization.

In static AspectC, this initialization operator is implemented as an `around` advice with a join point of the program's entry function – in the case of NetBSD, this is `kernel_entry`, which is the equivalent to the `main` function in normal C programs. Dynamic AspectC, discussed below, requires execution of the init function when the aspect is loaded. Here, a special support library for dynamic aspects is responsible for calling the init function. Further details on this functionality are described in the following chapter.

5.10.2 Local Storage for Aspects

Providing local storage for aspects enables several pieces of advice inside an aspect to coordinate themselves to record more complex states. Local storage for aspects, however, cannot be provided on the normal program stack. While creating the storage objects on the normal program heap would be possible, AspectC allocates a special per-aspect heap for these objects in order not to interfer with the regular program operation in unpredictable ways. References to aspect-local variables inside advice code are, in turn, replaced with operations dereferencing pointers to the aspect-local heap, shown in fig. 5.3.

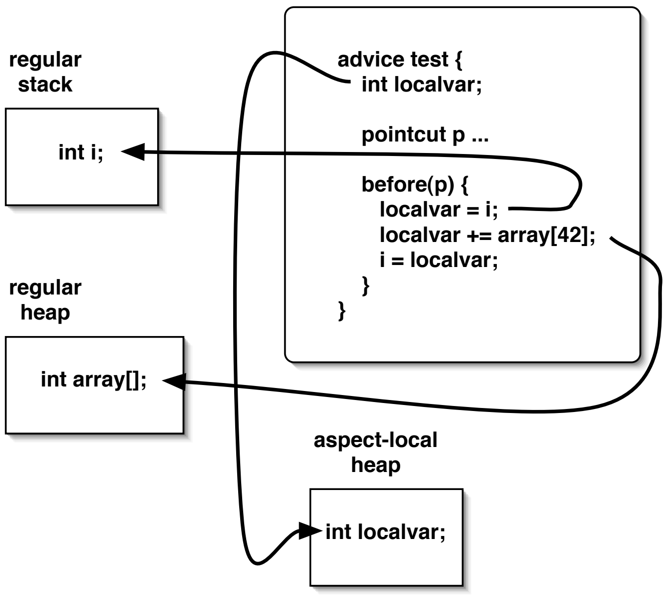


Figure 5.3: Local storage for aspects

Together with the init function, these extended AspectC aspects behave similar to first-class objects in OOP languages, but do not provide reliable data encapsulation.

5.10.3 Use Case: A Sense of Time for Aspects

Implementing local storage for AspectC aspects permits the programmer to capture state between the invocation of several advice code functions belonging to one aspect. The following use case illustrates this.

Consider an application where writes to a certain file descriptor have to be intercepted in order to perform additional functionality before the data is output to the file descriptor. This functionality could be translating the data that is to be output to a different language, implementing additional handshaking before sending data, or filtering information log output according to several severity levels. The code for writing data to the file descriptor is scattered throughout the code base using the `write` system call; in addition, the file descriptor referring to the output stream is obtained by the `open` system call and invalidated by calling `close`.

`Open` returns a file descriptor that subsequent calls to write and close use to indicate the output stream to write data on and invalidate. In Unix, file descriptors are re-used, so a certain value of a descriptor may point to a different output stream at a later point in time. If the aspect defined should only affect writes to one well-defined output stream (e.g., we only want to translate output to the console, but no output to disk files or the network), special care has to be taken to only supply the correct incovations of write with advice.

The AspectC using local storage is shown in listing 5.10.

```
aspect(translate) {
  int valid_descriptor;
  int assigned_descriptor;

  pointcut pc_open {
    call(open(char *name, ..);
  };

  pointcut pc_close {
    call(close(int fd);
  };

  pointcut pc_write {
    call(write(int fd, void *buf, int len);
  };

  init() {
    valid_descriptor = 0;
  }

  around(pc_open) {
    if (strncmp(name, "/dev/tty", 8) == 0) {
      valid_descriptor = 1;
      assigned_descriptor = proceed();
    }
  }

  around(pc_close) {
    if (valid_descriptor == 1 && fd == assigned_descriptor)
      valid_descriptor = 0;
  }

  around(pc_write) {
    if (valid_descriptor && fd == assigned_descriptor) {
      translate(buf);
      write(fd, buf, len);
    }
  }
}
```

Listing 5.10: Temporal dependencies in AspectC

Here, two aspect-local variables, `valid_descriptor` and `assigned_descriptor`, are used to keep track of the file descriptor and indicate its validity. The advice routines around `open` and `close` check for the correct file being associated with the file descriptor and set the local variables accordingly. The third advice function, then, calls the `translate` function only if the assigned file descriptor matches the one passed in this invocation and the validity of the descriptor is signaled by the value of `valid_descriptor`. Thus, local storage for aspects permits advice code to keep state in between advice code invocation, but using this state to control advice invocation involves a significant overhead for *each* `write` system call invoked.

5.10.4 Dynamic AspectC

The previous discussion of AspectC concerned static weaving. An extension of AspectC to support dynamic aspects uses an identical pointcut and aspect definition language with simple extensions inside the advice code that allow activation and deactivation of aspects at runtime.

The structure of the compiler, however, is changed significantly. Instead of performing a code transformation, aspect code is now compiled into separate dynamically loadable objects and pointcut descriptions are compiled into a description format suitable for a separate dynamic weaver component.

Aspects in AspectC are named. While static weaving made no use of the aspect identifier, the name of an aspect is used for activation and deactivation of specified aspects. In order to permit this, advice code for dynamic AspectC may use two additional calls to a support library: *ac_activate()* and *ac_deactivate()*, which validate and invalidate advice code execution or the validity of a pointcut. Weaving dynamic AspectC is now delayed to the execution phase of a system; thus, the object files containing advice code and pointcut descriptions generated by the AspectC compiler are handed to the runtime weaver.

5.10.5 Expressing Context Dependencies Using Dynamic AOP

One of the special conditions regarding operating system kernel code is that code executed inside the kernel is in most cases running in either a user-mode process context or an asynchronous (interrupt) context, as described in chapter 2.

A new join point type that captures the process context of code eases the formulation of pointcuts. For example, a pointcut could employ security policies for network code that only affect a certain kind of processes, e.g. a web browser.

There are some other useful contexts in an operating system environment. User and group belonging to an executing piece of code to implement access restrictions are useful for restricting operations nonprivileged users can execute; network contexts would restrict pointcuts to the processing flow of a specified network connection.

An example for a pointcut matching the `sys_open` system call only when the executing process' name matches "`ls`" is as follows:

```
pointcut(example):
      cflow(sys_open) && process_name("ls");
```

Support for context-specific join points is currently implemented in prototypical form.

5.10.6 Use Case: Temporal Dependencies

Another important property of in-kernel code is the temporal change of system status. A problem that arises is that the kernel reuses descriptors like file descriptors and process IDs over the uptime of the system.

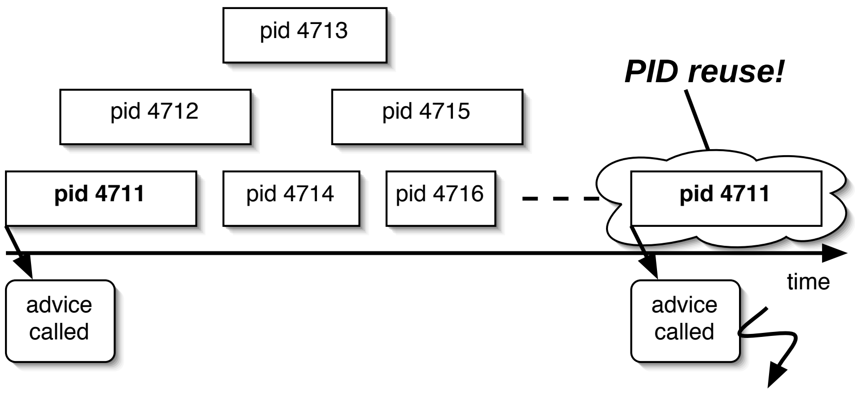


Figure 5.4: PID reuse evokes unwanted aspect call

Without support for lifetime management for pointcuts, naïve pointcut descriptions could try to capture an event related to a specific process ID and would at a later point of time be inadvertedly match again when the kernel reuses the matching process ID, shown in fig. 5.4.

A new join point type that is invalidated after matching the object would be a simple solution to this problem. An advanced concept could be implemented using transactional join points, which ensure that reuse of context descriptions in the kernel is automatically detected.

If the previous example demonstrating time dependency in static aspects is extended to use dynamic aspects, the implementation code looks as shown in listing 5.11.

```
aspect(translate) {
  int valid_descriptor;
  int assigned_descriptor;

  pointcut pc_open {
    call(open(char *name, ..);
  };

  pointcut pc_close {
    call(close(int fd);
  };

  pointcut pc_write {
    call(write(int fd, void *buf, int len);
  };

  init() {
    valid_descriptor = 0;
  }

  around(pc_open) {
    if (strncmp(name, "/dev/tty", 8) == 0) {
      ac_activate("pc_write");
    }
  }

  around(pc_close) {
    if (valid_descriptor == 1 && fd == assigned_descriptor)
      ac_deactivate("pc_write");
  }

  around(pc_write) {
      translate(buf);
      write(fd, buf, len);
  }
}
```

Listing 5.11: Dynamic AspectC implementation

Compared to the previous implementation, the `around(pc_write)` advice does not have to check for a valid condition any longer; this condition is rather handled by the `pc_open` and `pc_close` advice code that dynamically activate and deactivate the `pc_write` pointcut depending on the file context. The benefit here is twofold – the development of aspect code does not have to deal with keeping extensive state and the advice for `pc_write` is only called when activated and does not require activation for every write call executed, returning early since the context is most often incorrect for the advice.

Dynamic aspects, however, introduce the danger of deadlocks in aspects since timing dependencies are introduced when advice code activation and deactivation intersect for different aspects. No research into deadlock avoidance for aspects has been performed so far. It is assumed, however, that the well-known mechanisms for deadlock prevention in operating systems apply here [41].

5.11 Evaluation

The AspectC compiler presented here is a rapidly developed prototype. Emphasis was put on robustness, C standards and GCC extensions conformance and a short development cycle. A consequence of using the CIL infrastructure as a preprocessing stage is that all source code has to be parsed twice – the first time by the AspectC compiler generating pure C code and the second time by the C compiler that transforms the woven code into object code. Thus, compiling AspectC programs currently takes more than twice the time of compiling equivalent pure C programs. For a prototype implementation, however, this compile time overhead remains in an acceptable range.

The overhead of static join points is negligible, since the function names are rewritten by the AspectC compiler to directly call the advice code wherever possible. Thus, the runtime penalty for simple cases of `before`, `after` and `around` advice is in the optimal case in the range of a function call plus the time required by the advice code itself. This is a comparable overhead to using virtual functions in C++.

Implementing `proceed` implies a slightly higher overhead, since the proceed implementation of the AspectC support library, shown above, performs some extensive consistency checks. However, this overhead is also static – in the range of three to four function invocations.

The only case where the runtime overhead of a construct is depending on the current conditions is the implementation of `cflow`. Since the function currently being executed can be at an almost arbitrary stack level, the performance of cflow depends on the distance of stack levels from the most recent invocation of the function specified in the cflow to the bottom of the stack. Still worse, the most common case is that the cflow does not match. In this case, the stack has to be traversed completely to the topmost function invocation, generating significant overhead that scales linearly with the function nesting level. This problem cannot be solved easily, since replacement of the stack walk with a hash table or similar constructs does not take recursive invocations of functions into account. One possible optimization is to introduce a separate call stack that only contains function names that quickly matched. This, however, requires additional memory space and stack frame handling functionality added to all function entry and exit points.

5.12 Summary

Most extensions of existing programming languages concentrate on extending object oriented base languages like C++ and Java with aspect-oriented constructs. However, an implementation of AOP functionality for the most widespread systems programming language, C, did not exist so far. The existence of industrial-strength code analysis tools for ANSI C with GNU extensions permitted the rapid development of a compiler for AspectC, an aspect-oriented extension of the C language, as a precompiler. While a standalone compiler is sufficient for implementing static AOP, dynamic AOP requires runtime support for weaving and unweaving. The first in-kernel toolkit for dynamic AOP, TOSKANA, is presented in chapter 7.

■

6. Execution Interaction

*"Programs must be written for people to read,
and only incidentally for machines to execute."*

— Abelson and Sussman

6.1 Introduction

Introducing dynamic aspects into native code directly executed on the CPU in a privileged execution mode – as required for the execution of operating system kernel code – can occur on different levels in the execution hierarchy of a system. This chapter describes various approaches and discusses the advantages and disadvantages of each approach as well as giving an estimation of the overhead induced.

Providing support for dynamic aspects in native code revolves around a simple basic principle – the access to code in the flow of operation (in the case of advice-augmented functions) and to data structures (for extending existing data structures) has to be intercepted and redirected to the appropriate advice code inside the "execution layer" (i.e., the part of the system responsible for execution of the particular code).

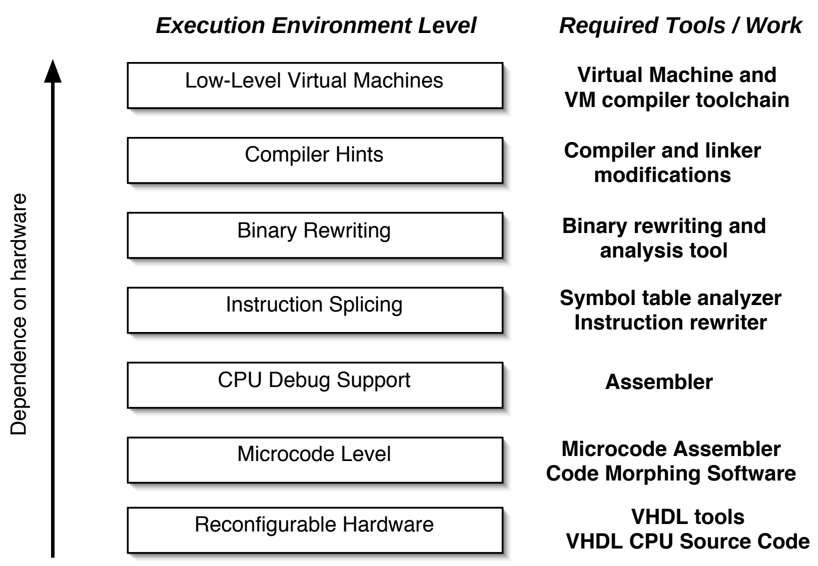
This already raises a significant problem, since information on the locality of the join points is related to certain locations in the original source code, not in the binary form resulting from compilation. Accordingly, a weaving tool that is located in the execution layer providing support for dynamic aspects has to reconstruct as much of the information as possible.

Parts of the contents of this chapter have been published in [49].

6.2 Deployment Approaches

Dynamic aspect deployment for native code can occur in several different execution environments. The environments are categorized according to their proximity to the hardware layer, depicted in fig. 6.1.

Starting from the level of reconfigurable hardware and going "upwards" towards more software-related layers, the possibilities to influence the execution of code and accordingly implement aspect deployment vary greatly. Since the aim is to provide aspect support for native code, approaches using complex virtual machines like the JVM or .NET as a basis are



Execution Environment Level
Required Tools / Work
Dependence on hardware
Low-Level Virtual Machines
Virtual Machine and VM compiler toolchain
Compiler Hints
Compiler and linker modifications
Binary Rewriting
Binary rewriting and analysis tool
Instruction Splicing
Symbol table analyzer Instruction rewriter
CPU Debug Support
Assembler
Microcode Level
Microcode Assembler Code Morphing Software
Reconfigurable Hardware
VHDL tools VHDL CPU Source Code

Figure 6.1: Execution layer position and tools required

not considered in this context (see e.g. [15] for further information on this topic). Aspect support in each of these execution layers requires a different set of tools and involves a special set of restrictions and complications which are discussed below.

6.2.1 Reconfigurable Hardware

Reconfigurable hardware that can be implemented in FPGA[1] chips is a common practice in providing flexible hardware components that are reprogrammable according to the demand of a specific application. Programming reconfigurable hardware today no longer takes place on the level of circuit diagrams, but rather advanced programming languages like VHDL or Verilog are used to generate a chip design from a “soft” description. FPGA technology has advanced so far that implementing a complete 32-bit RISC-style CPU in an FPGA is easily feasible, which allows for the flexibility required for improved support for aspects in the CPU execution layer.

An FPGA implementation would introduce an additional stage in the commonplace pipelined structure of instruction execution. Depending on the joinpoint type, i.e. the particular class of execution to be intercepted (instruction execution, data read or write access), the instruction fetch, data read or write or branch target calculation stage will be preceded by an optionally inserted pipeline stage that checks if a condition for a currently deployed aspect (that is defined using a special instruction of the CPU) is satisfied and executes a low-overhead branch instruction (instead of a vastly more expensive context switch) to the advice code.

A representation of the improved pipeline structure to support dynamic aspects can be seen in fig. 6.2. Aspect support intercepts the pipeline flow after the instruction fetch stage

[1]Field Programmable Gate Array

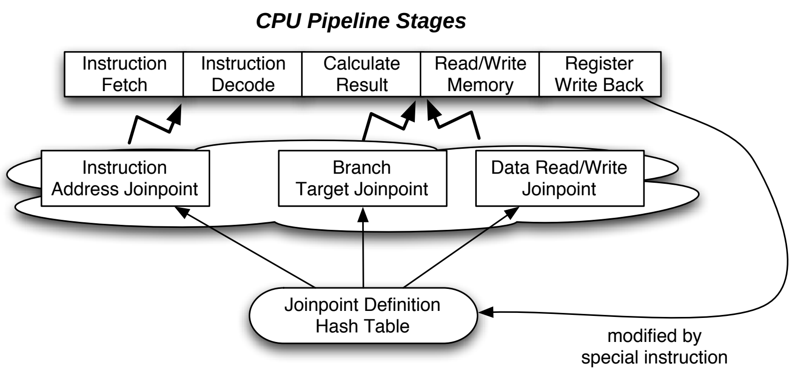


Figure 6.2: 5-stage CPU pipeline with added aspect support

and in between the result calculation stage (where branch target addresses are calculated) and the read/write memory stage (where the actual data access is executed), respectively.

The dynamic introduction of pipeline stages during the execution of an instruction slows down the operation of the CPU, so the methods for retrieving the locations of joinpoints have to be very efficient, possibly by creating on-chip hash tables to improve lookup speed.

Adding joinpoint definition functions to the CPU will require a modified compiler toolchain. This does not induce significant overhead, since an aspect-supporting compiler has to be created anyway.

6.2.2 CPU Debug Support

Some CPUs, most notably Intel's Pentium 4 and IBM's Power 4 and Power 5, also support additional functionality originally designed for supporting debugging tasks. This support usually consists of a set of special *monitoring registers* that generate an exception every time a memory address defined in one of these registers is accessed, either by instruction execution or by a data read or write operation.

Whenever such a condition occurs, the CPU executes special functionality to execute the debug handler – which for dynamic AOP could be the advice code itself. Unfortunately, only four debug registers exist that can monitor from one to 16 bytes of memory each, so the number of deployable aspects would be very limited. It is unknown if future CPUs will contain a larger set of debug registers. Since implementing these registers potentially slows down native code execution[2], the future availability of sufficiently large set of registers for dynamic AOP use is not guaranteed.

Additionally, on detection of a monitored condition, the CPU takes a trap, changing the operating mode to a different context. This results in an overhead of about 200-300 instruction times, making calls to aspect code relatively expensive.

[2]Every command executed has to be checked against all possible addresses in these registers.

6.3 Microcode Level

Going one level up, the behavior of machine-level instruction execution can also be influenced by changing the CPU-internal code that is used by the CPU in order to execute the given machine instruction. In older CPU designs, this code is usually *microcode* that controls the low-level operation of the CPU related to the specific functional units (this model is e.g. used in digital VAX processors); in more modern designs a well-known instruction set (like the x86 instructions) are implemented in form of a highly optimized emulator program that creates the behaviour of the desired instruction set on top of an effective and simple minimal CPU. This model is used by Transmeta's Crusoe CPU, which simulates a modern x86 CPU in software running on top of a VLIW (Very Large Instruction Word) CPU.

Compared to the interception based on introducing pipeline stages, a microcode-based solution works on a software basis, albeit one usually not accessible to the system programmer. Similar to the pipeline extensions, the microcode functions responsible for instruction fetch, branch target calculation and data read or write operations have to be extended to intercept the microcode operation at active join points. Like in the case of using reconfigurable hardware, a special instruction can be used to define join point locations and types to the microcode level.

Implementing this solution requires support to microcode or emulation-level development tools as well as the source code of the microcode layer or emulator. The implementation could prove to be difficult since the space available for microcode storage inside a CPU respectively the memory reserved for emulator usage is usually very restricted.

6.4 Binary Code Manipulation

Binary code manipulation is a method that modifies the text section, containing the program machine instructions, of a running program. This can either be implemented as *code splicing*, which changes instructions in the program's memory space directly, or using *binary rewriting* technology, which analyzes and transforms the code of a program prior to its execution on the CPU.

6.4.1 Code Splicing

The basic method for inserting dynamic join points shadows into native code is *code splicing*. Code splicing is a technology that replaces the bit patterns of instructions in native code with a branch to a location outside of the predefined code flow, where additional instructions followed by the originally replaced instruction and a jump back to the instruction after the splicing location are inserted.

Fine-grained code splicing is able to insert instrumentation code with the granularity of a single machine instruction. As shown in fig. 6.3, splicing replaces one or more machine instructions at a join point with a jump instruction to the advice code with the effect that the advice code is executed before the replaced instruction.

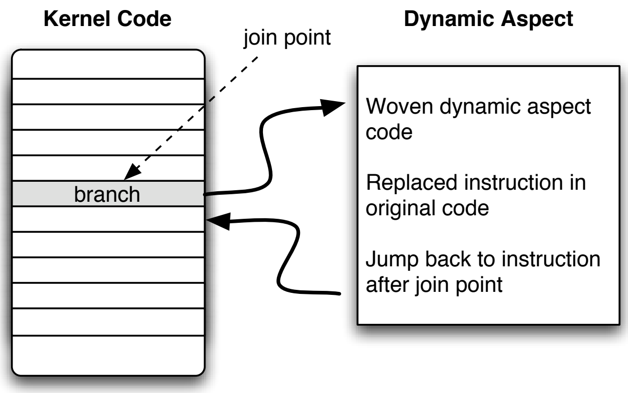

Kernel Code
join point
Dynamic Aspect
branch
Woven dynamic aspect code
Replaced instruction in original code
Jump back to instruction after join point

Figure 6.3: Code splicing

When modifying native code, some precautions have to be taken to avoid corrupting activities currently running in kernel mode.

Since the execution of kernel functions may usually be interrupted at any time, it is desirable to make the splicing operation atomic. This, however, can be quite problematic, depending on the particular architecture. On most x86 processors, for example, the *jump* instruction that redirects control flow to the dynamic aspect is at least five bytes long, but atomic writes can only be handled up to word size (4 bytes)[3].

Another problem may be that more than one instruction had to be replaced by the jump instruction in the splicing process and the second of these replaced instructions is a branch target. A branch to that address would then end up trying to execute part of the target address as operation code with unpredictable results.

Since only few bytes are being replaced, this situation could only occur in functions written in assembly language. However, this situation is unlikely to occur.

6.4.2 Binary Rewriting and Emulation

Binary rewriting is a technology that takes a binary, analyzes its operation and semantic behavior to some extent and rewrites the code contained in the binary. Originally, this technology was intended to execute code for one specific CPU on a CPU with a different instruction set; one example is Digital's FX!32 [26], which was used to run Windows NT applications compiled for a x86 CPU on an Alpha-based NT system.

The binary rewriting technology performs an analysis of the program structure that is executed, a binary rewriter can also used to implement aspect deployment by "rewriting" the original native code into code for *the same* instruction set, inserting calls to aspect code on the fly in the generated native code instruction stream. The binary rewriter is able to capture the complete state of the CPU during the rewriting process, thus interception capabilities similar to the approaches using reconfigurable hardware or microcode are

[3]Recent x86 CPUs allow the atomic replacement of an eight byte word using a different instruction

available.

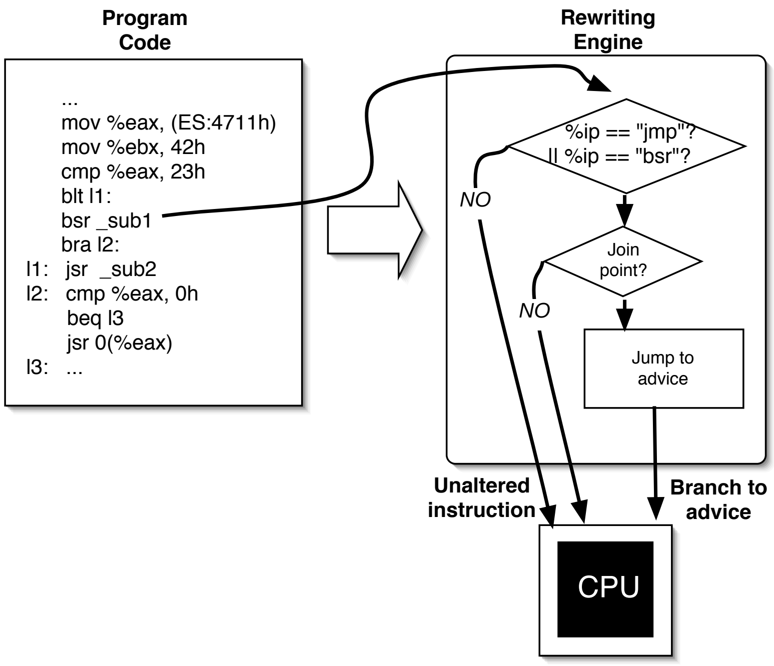


Figure 6.4: Binary rewriting

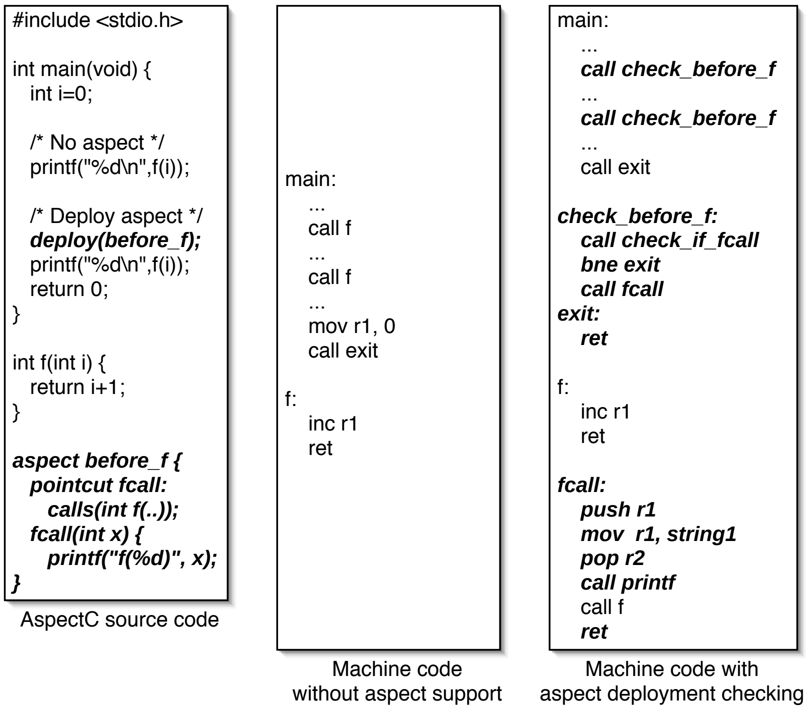
The principal use of binary rewriting for weaving aspects is shown in fig. 6.4. The original program's instruction stream is intercepted by the rewriting engine. Every instruction to be executed is checked for patterns indicating a possible join point – e.g., `branch` and `jump subroutine` instructions for `call` join points. If the current instruction does not match these patterns, it is passed unaltered to the CPU for execution. If, however, the pattern matches, the branch or jump target is checked against a hash table of active join points. If a join point is matched, instead of the original instruction, a jump to the appropriate advice code function is inserted into the instruction flow, causing the advice function to be executed.

However, rewriting instructions in the kernel level may prove difficult, since the basic kernel execution has to be controlled by an instance of the binary rewriting tool, thereby creating a very thin abstraction layer on top of the CPU. Due to different timing behaviour of the instruction rewriting tool in comparison to code running on the bare CPU, timing-critical sections inside a kernel may not be suitable for execution with a rewriter. Binary rewriting might, however, prove to be a valuable tool for dynamic aspect deployment in user mode applications.

An extended discussion of the binary rewriting approach can be found in chapter 8, where it is analyzed from the viewpoint of virtualization.

6.5 Compiler Hints

The obvious approach to solving this problem is to introduce additional code at all possible join point locations that actually checks if this specific join point is enabled in the current context. This, however, generates significant overhead in the generated binary code. For deploying *before* advice, a static call to a advice activation function has to be introduced which checks for activation conditions and then optionally executes the advice code, resulting in a significant execution delay in cases where advice code is not being activated (which is the regular case) of at least a function call and return plus the overhead required by performing the check itself.



```
#include <stdio.h>

int main(void) {
   int i=0;

   /* No aspect */
   printf("%d\n",f(i));

   /* Deploy aspect */
   deploy(before_f);
   printf("%d\n",f(i));
   return 0;
}

int f(int i) {
   return i+1;
}

aspect before_f {
   pointcut fcall:
      calls(int f(..));
   fcall(int x) {
      printf("f(%d)", x);
}
```

```
main:
    ...
    call f
    ...
    call f
    ...
    mov r1, 0
    call exit

f:
    inc r1
    ret
```

```
main:
    ...
    call check_before_f
    ...
    call check_before_f
    ...
    call exit

check_before_f:
    call check_if_fcall
    bne exit
    call fcall
exit:
    ret

f:
    inc r1
    ret

fcall:
    push r1
    mov  r1, string1
    pop r2
    call printf
    call f
    ret
```

Figure 6.5: Aspect deployment with compiler hints

With about 20,000 functions contained in a typical monolithic operating system kernel (NetBSD 2), the overhead in code size would not be negligible. Figure 6.5 shows the transformation of a simple C source code into machine code without aspects and with an added dynamic aspect check and advice code (aspect support code and generated machine code for the aspect is shown in bold italic font).

6.6 Low-Level Virtual Machines

In contrast to well-known virtual machine approaches like the Java Virtual Machine and .NET, *low-level virtual machines* (LLVMs) provide a virtual execution environment that is very similar to the native instruction set of a real CPU. As a consequence, LLVM instructions can be interpreted by or translated into native code on the fly with a comparably low overhead and only minimal runtime support.

Compared to the binary rewriting approach detailed above, the low-level virtual machine induces a performance loss due to the requirement of translating the virtual instructions to instructions suitable for the system's CPU. This overhead, however, can be minimized by using a just-in-time compilation approach that implements the execution of VM instructions as a set of calls to optimized functions performing the VM instruction's operation in native code.

Furthermore, a specialized compiler toolchain targeting the virtual machine is required. This is, however, readily available for a LLVM based on the GNU Compiler Collection.

6.7 Summary

Introducing aspects into binary code is possible on various levels, ranging from hardware-software codesign approaches to efficient, low-level virtual machines. The various approaches differ in feasibility of their implementation as well as the features offered.

The most feasible approaches for dynamic aspect weaving in the context of this thesis are binary code manipulation using code splicing, which can strongly benefit from similar base technologies already introduced in code instrumentation toolkits, and weaving using a low-level virtual machine. Both approches have been implemented as prototypes in the TOSKANA and TOSKANA-VM systems and are described in the following two chapters.

7. Dynamic Kernel Aspects: TOSKANA

*"Unix was not designed to stop people from doing stupid things,
because that would also stop them from doing clever things."*

— Doug Gwyn

7.1 Introduction

In order to create applications using aspect-oriented technology in kernel space, a toolkit had to be devised and implemented that provides basic dynamic AOP functionality inside the kernel. TOSKANA, the "Toolkit for Operating System Kernel Aspects with Nice Applications", is a dynamic aspect toolkit using code splicing methods for the monolithic NetBSD kernel.

Since TOSKANA relies on the NetBSD loadable kernel module (LKM) structure to dynamically insert advice code into the kernel address space, an overview of the NetBSD kernel module mechanisms is prepended before of TOSKANA itself is described.

Parts of this chapter have been published in [50].

7.2 NetBSD Modules

In early versions of Unix, adding functionality to an existing kernel involved changing a configuration file and rebuilding the kernel binary, followed by a system reboot. This is especially annyoing with modern systems, where hardware (like USB devices) can be added and removed on the fly. A solution to this problem is to use *loadable kernel modules*, which are object files in the standard ELF [131] binary format that can be loaded into and unloaded from kernel space at runtime with the help of a special set of kernel module tools.

Kernel modules are not restricted to implementing device drivers. Instead, a kernel module can provide any function that is possibly running in kernel context, since it has complete direct access to the system hardware as well as all kernel memory – i.e., all code and data structures.

Since NetBSD kernel modules form the basis for the deployment of dynamic aspects in the TOSKANA system, some example code (adapted from the NetBSD lkm documentation [76]) follows that describes their functionality.

The loadable kernel module interface was originally designed to be similar in functionality to the loadable kernel modules facility provided by SunOS. The lkm(4) facility is controlled by performing ioctl(2) calls on the /dev/lkm device, but since all operations are handled by the modload(8), modunload(8) and modstat(8) programs, so the user should never have to interact with /dev/lkm directly. The NetBSD kernel must be compiled with the LKM option in order to make use of LKMs.

7.2.1 A Simple Device Driver Module

This example shows a driver module, `fibo.o`, that simply calculates Fibonacci numbers, so no real hardware interaction is performed here. The driver will provide eight minor devices, `/dev/fibo0` to `/dev/fibo7`. Each of these devices offer the following functions:

```
static int fibo_open(dev_t, int, int, struct proc *);
static int fibo_close(dev_t, int, int, struct proc *);
static int fibo_read(dev_t dev, struct uio *, int);
```

The device provided by this driver can be opened and closed, and data can be read from it; write operations are not supported. To make the device available, the kernel needs to know that a new character device with the 3 functions listed above is to be added. Therefore, a struct `cdevsw` (character device switch) has to be populated:

```
static struct cdevsw fibo_dev = {
        fibo_open,      /* open function */
        fibo_close,     /* close function */
        fibo_read,      /* read function */
        (dev_type_write((*))) enodev,    /* write function not implemented */
        (dev_type_ioctl((*))) enodev,    /* ioctl function not implemented */
        (dev_type_stop((*))) enodev,     /* stop function not implemented */
        0,
        (dev_type_poll((*))) enodev,     /* poll function not implemented */
        (dev_type_mmap((*))) enodev,     /* mmap function not implemented */
        0
};
```

Here, `enodev` is a generic function that simply returns the error number `ENODEV` (Operation not supported by device) which implies that no operations besides `open`, `close` and `read` are supported. So, for example, whenever a process tried to write to the device, the call will fail with `ENODEV`.

Furthermore, the kernel needs to know how the module is named and where to find information about operations provided by the module. This task is simplified by the lkm interface: the preprocessor macro `MOD_DEV` can be used to provide this information.[1]:

```
#if (__NetBSD_Version__ >= 106080000)
MOD_DEV("fibo", "fibo", NULL, -1, &fibo_dev, -1);
#else
MOD_DEV("fibo", LM_DT_CHAR, -1, &fibo_dev)
#endif
```

[1]The MOD_DEV macro was changed after the release of NetBSD 1.6.2; this is reflected by the #ifdefs in the code shown.

The code defines a name of “fibo” for the kernel module, which is a character major device . The driver requires a dynamic major device number from the kernel (minor devices are handled by the module itself) and finally, information about the supported operations is passed to the kernel in `fibo_dev`.

In order to ensure proper unloading of the module, a global reference counter of opened minor devices has to be kept:

```
static int fibo_refcnt = 0;
```

In addition, some information about each minor device is required:

```
struct fibo_softc {
        int             sc_refcnt;
        u_int32_t       sc_current;
        u_int32_t       sc_previous;
};

#define    MAXFIBODEVS     8

static struct fibo_softc fibo_scs[MAXFIBODEVS];
```

The driver will provide 8 minor devices. Each minor device stores information about how often it was opened (in our example each minor device can only be opened once for the sake of simplicity), the current number and the previous number for calculating the Fibonacci numbers.

Each kernel module needs to have an entry point which is passed to ld(1) by modload when the module is linked. The default module entry point is named *xxx*`init`. If the init function cannot be found, an attempt to use the function `modulename_lkmentry` will be made, where modulename is the filename of the module being loaded. In general, the entry function will consist entirely of a single DISPATCH line, with DISPATCH being a preprocessor macro defined in sys/lkm.h to handle loading, unloading and stating. Thus, the fibo_lkmentry function will look like this:

```
int
fibo_lkmentry(struct lkm_table *lkmtp, int cmd, nt ver)
{
        DISPATCH(lkmtp, cmd, ver, fibo_handle, fibo_handle, fibo_handle);
}
```

7.2.2 Module Functionality

Now, a handler function for the module is required to do module specific tasks when loading, unloading or stating the module. The name of this handler function is passed to DISPATCH to tell the kernel which function it has to call. A pointer to the module entry in the LKM table and an integer representing the desired command (LKM_E_LOAD, LKM_E_UNLOAD or LKM_E_STAT) are passed to the handler function. The handler is called after the module is linked and loaded into the kernel with the LKM_E_LOAD command. After this, the driver needs to check whether the module was already loaded into the kernel and initializes the data structures. When unloading the module, the handler is called with the LKM_E_UNLOAD command and a check is introducted to see if the module is not required any more (e.g. check if all devices are closed for char/block driver modules) before confirming the unload command:

```
static int
fibo_handle(struct lkm_table *lkmtp, int cmd)
{
  switch (cmd) {
  case LKM_E_LOAD:
          /* check if module was already loaded */
          if (lkmexists(lkmtp))
                    return (EEXIST);

          /* initialize minor device structures */
          bzero(fibo_scs, sizeof(fibo_scs));
          printf("fibo:␣Fibonacci␣driver␣loaded␣successfully\n");
          break;

  case LKM_E_UNLOAD:
          /* check if a minor device is opened */
          if (fibo_refcnt > 0)
                     return (EBUSY);
          break;

  case LKM_E_STAT:
                 break;

  default:
                 return (EIO);
  }

  return (0);
}
```

The open function is quite simple, since most housekeeping functionality is already provided by the NetBSD kernel – e.g. the kernel will automatically allocate a vnode(9) for the device. The parameters for the open function are the major and minor device numbers (using the major and minor macros), the flag and mode arguments as described in open(2) and a pointer to the struct proc of the process that executes the open system call.

The first action in `open` is to check if the minor device number received when the device was opened is not out of range, and if the minor device is not already opened. Then, the minor device data (the Fibonacci starting numbers: = 1, 0 + 1 = 1, 1 + 1 = 2, 1 + 2 = 3, ...) is initialized and the minor device and the global module reference counter increased:

```
static int
fibo_open(dev_t dev, int flag, int mode, struct proc *p)
{
      struct fibo_softc *fibosc = (fibo_scs + minor(dev));

      if (minor(dev) >= MAXFIBODEVS)
          return (ENODEV);

      /* check if device already open */
```

```
        if (fibosc->sc_refcnt > 0)
                return (EBUSY);

        fibosc->sc_current = 1;
        fibosc->sc_previous = 0;

        /* increase device reference counter */
        fibosc->sc_refcnt++;

        /* increase module reference counter */
        fibo_refcnt++;

        return (0);
}
```

The `close` function uses the same parameters with the same meanings as the open function described above. It is used to free the internal data structures of a minor device opened before. The close function does not need to check whether the device was actually opened before or to release the vnode associated with the device; rather, only module specific information has to be deallocated. In the case of the Fibonacci module example, this means decreasing the minor device and the global module reference counters:

```
static int
fibo_close(dev_t dev, int flag, int mode, struct proc *p)
{
    struct fibo_softc *fibosc = (fibo_scs + minor(dev));

    /* decrease device reference counter */
    fibosc->sc_refcnt--;

    /* decrease module reference counter */
    fibo_refcnt--;

    return (0);
}
```

The main functionality of the module is provided in the `read` function. This function has three parameters: the device major and minor numbers like in the open and close functions, a flag field indicating whether the read should be done in a non-blocking fashion and a pointer to a struct uio. A struct uio typically describes data in motion, in case of a read(2) system call data moved from kernel-space to user-space[2]:

```
static int
fibo_read(dev_t dev, struct uio *uio, int flag)
{
    struct fibo_softc *fibosc = (fibo_scs + minor(dev));

    if (uio->uio_resid < sizeof(u_int32_t))
        return (EINVAL);

    while (uio->uio_resid >= sizeof(u_int32_t)) {
```

[2]The uio concept is unique to NetBSD and may be unfamiliar to driver developers for other Unix-based operating systems like Linux. See the NetBSD uiomove(9) man page for more information.

```
        int error;

        /* copy to user space */
        if ((error = uiomove(&(fibosc->sc_current),
                             sizeof(fibosc->sc_current), uio))) {
                     return (error);
        }

        /* prevent overflow */
        if (fibosc->sc_current > (MAXFIBONUM - 1)) {
                 fibosc->sc_current = 1;
                 fibosc->sc_previous = 0;
                 continue;
        }

        /* calculate */ {
                 u_int32_t tmp;

                 tmp = fibosc->sc_current;
                 fibosc->sc_current += fibosc->sc_previous;
                 fibosc->sc_previous = tmp;
        }
    }

    return (0);
}
```

The first operation performed by `read` is to check whether the process requests less than sizeof(u_int32_t) bytes (usually, 4 bytes). The read function always reads a multiple of four byte blocks; in order to keep the example code simple and reading less than 4 bytes at a time is disallowed (uio-¿uio_resid holds the number of remaining bytes to move to user-space, automatically decreased by uiomove).

The read function copies the current Fibonacci number into the user-space buffer, checks for a possible overflow (only the first 42 Fibonacci numbers fit into u_int32_t), and calculates the next Fibonacci number. If there is enough space left in the user-space buffer, the function loops and restarts the process of moving, checking and calculating until the buffer is filled up to the possible maximum or uiomove returns an error. Thus, a read(2) system call on this device will never return 0, and, as a consequence, will never reach an end-of-file (EOF), so the device will generate Fibonacci numbers forever.

7.2.3 Loading the Module

In addition to the code for the device driver module itself, a shell script is required that will be executed when the module is successfully loaded to create the required device nodes in `/dev`. This shell script gets passed three arguments: the module id (in decimal), the module type (in hexadecimal) and the character major device number (this differs for other types of LKMs such as system call modules):

```
if [ $# -ne 3 ]; then
   echo "$0␣should␣only␣be␣called␣from␣modload(8)␣with␣3␣args"
   exit 1
```

```
fi
```

First, the script checks whether all three command line arguments are present and exits with error code if not.

```
for i in 0 1 2 3 4 5 6 7; do
    rm -f /dev/fibo$i
    mknod /dev/fibo$i c $3 $i
    chmod 666 /dev/fibo$i
done
exit 0
```

Finally, the required special device nodes are created. In order to test the module developed, the module can be compiled and loaded using the following command (this needs to be run as superuser):

```
modload -e fibo_lkmentry -p fibo_post.sh fibo.o
```

If everything went well, the modstat(8) program presents output similar to this:

```
Type     Id  Off Loadaddr Size Info     Rev Module Name
DEV       0   29 dca4f000 0004 dca4f260   1 fibo
```

In order to test the new kernel module, a small test program is required that does nothing more than reading a 32bit unsigned integer value from /dev/fibo0 and printing the value to standard output:

```
#define DEVICE  "/dev/fibo0"

int
main(int argc, char **argv)
{
    u_int32_t val;
    int fd, ret;

    if ((fd = open(DEVICE, O_RDONLY)) < 0)
        err(1, "unable␣to␣open␣" DEVICE);
    while ((ret = read(fd, &val, sizeof(val))) == sizeof(val))
        printf("%u\n", val);
    if (ret < 0)
        err(2, "read(" DEVICE ")");
    close(fd);
    return 0;
}
```

When this sample test program is run, it will output Fibonacci numbers below Fib(11) until it is interrupted or killed. To unload the kernel module, the following command has to be executed with superuser privileges:

```
modunload -n fibo
```

7.3 TOSKANA — Aspects for NetBSD

Based on the NetBSD kernel module support described in the previous sections, TOSKANA provides a weaver for dynamic aspects as well as the required support tools and libraries for developing and compiling aspects inside the NetBSD kernel.

7.3.1 Structure of TOSKANA

TOSKANA – the "Toolkit for Operating System Kernel Aspects with Nice Applications" – is a system for deploying dynamic aspects into an operating system kernel as a central part of a computer system. TOSKANA provides *before*, *after* and *around* advice for in-kernel functions and supports the specification of pointcuts as well as the implementation of aspects themselves as dynamically exchangeable kernel modules.

The TOSKANA system provides the necessary tools for weaving and unweaving aspects into a running NetBSD kernel. One of the design goals was to avoid any modification to the underlying NetBSD kernel source code, so that dynamic AOP can be deployed on demand in a production system without having to apply refactoring methods. The basic aspect deployment method used is fine-grained code splicing, which is able to insert instrumentation code with the granularity of a single machine instruction.

TOSKANA consists of several components. The central part is a dynamic weaver running in user space that used services of the kernel module loader and the kernel aspect library module to dynamically load und remove kernel modules into and from a running NetBSD kernel. These kernel modules contain advice code, which is generated either by the system C compiler using a special header file (low-level dynamic aspect macros) or by an extension of the AspectC compiler described in chapter 5, which provides pointcut descriptions along with the dynamically loadable aspect modules.

The current implementation of TOSKANA is available for NetBSD running on x86-based systems. Its basic mechanisms, however, are applicable to a large range of CPUs and operating systems, so that its application is not restricted to this environment. In fact, a port of TOSKANA to PowerPC-based hardware running Apple's BSD- and Mach-based MacOS X operating system is in progress.

7.3.2 Access to Kernel Internals

In order to get hold of the required symbol and context information for weaving and unweaving aspects, TOSKANA requires insight into internal kernel structures such as the symbol table.

Older Unix-like systems only provided access to kernel memory space to code executing in kernel context. Beginning with the BSD-derived SunOS 4, Unix system developers introduced an interface as a special device file that permits privileged applications (usually running unter the root UID) to gain access to kernel virtual memory space. This device, commonly named `/dev/kmem`, is also available in NetBSD and permits read and write access to kernel space virtual memory without creating a kernel module.

In NetBSD 2.0, another device was added as `/dev/ksyms` that in addition permits access to the symbol table of the running kernel. A user mode program can either access the symbol table using a set of `ioctls` – special function calls to a device file that can control operating parameters or retrieve additional out-of-band information not contained in the device's data stream – or by directly interpreting the ELF symbol table that is available by `mmap`ing `/dev/ksyms`, i.e. directly accessing the contents of the symbol table as a memory area in a user mode application[3]. In the ELF standard [131], the kinds of symbols listed in fig. 7.1 are defined.

Name	Description
STT_NOTYPE	not specified
STT_OBJECT	data object (variable, array etc.)
STT_FUNC	function or other executable code
STT_SECTION	relocation section
STT_LOPROC	processor-specific
...	...
STT_HIPROC	...definitions

Figure 7.1: ELF symbol types

Of interest for dynamic aspect weaving are the `STT_FUNC` and `STT_OBJECT` symbol entries, describing the virtual addresses of in-kernel functions and data objects (variables, arrays etc.), respectively.

Experience with `/dev/ksyms` showed, however, that accessing the symbol information via the provided `ioctl` calls did not provide all available symbols in the binary. Access to all symbol information contained in the ELF binary was only available using the symbol table directly. An arbitrary excerpt of the symbol table output, created with the GNU tool *readelf*, is shown in fig. 7.1. The information required for dynamic aspect weaving is contained in the `FUNC` entries that associate an address in kernel virtual memory space with the start of a function.

```
Symbol table '.symtab' contains 22895 entries:
Num:  Value Size Type Bind  Vis     Ndx Name

...

805: c010c9f8 24 FUNC LOCAL DEFAULT ABS deflate_compress
806: c010ca10 24 FUNC LOCAL DEFAULT ABS deflate_decompress
807: 00000000  0 FILE LOCAL DEFAULT ABS crypto.c
808: c010ca28 48 FUNC LOCAL DEFAULT ABS nanouptime
809: c073aae0  4 OBJ  LOCAL DEFAULT ABS crypto_drivers
810: c073aae4  4 OBJ  LOCAL DEFAULT ABS crypto_drivers_num
```

Listing 7.1: Excerpt from the NetBSD symbol table

[3]It is also possible – but more complicated – to directly obtain the symbol information from kernel virtual memory.

7.3.3 Code Splicing in TOSKANA

Inserting dynamic join points shadows into native code in TOSKANA uses *code splicing* technology. Code splicing replaces bit patterns of instructions in native code with bytes that represent the instruction code of a branch. The target of this branch is a location outside of the predefined code flow; here, advice code, the originally replaced instruction and a transfer of control returning to the instruction after the splicing location are inserted.

TOSKANA uses fine-grained code splicing. Fine granularity implies that the splicing technology is able to insert instrumentation code with the granularity of a single machine instruction. Splicing replaces one or more machine instructions at a join point with a jump instruction to the advice code. As a consequence, the woven advice code is executed before the replaced instruction. Fig. 7.2 details the process; here, the dynamic aspect woven is modified on demand to include the correct instruction to return to the original code.

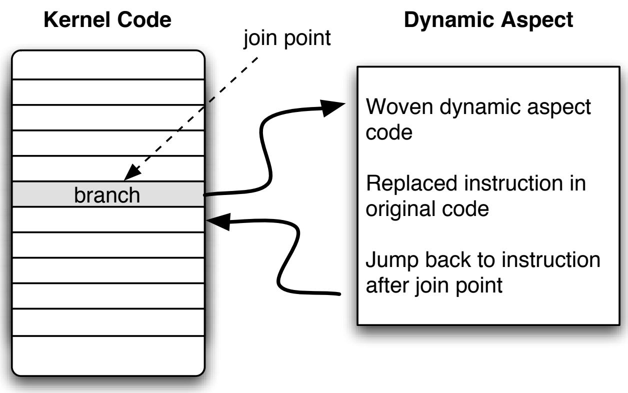

Kernel Code
join point
Dynamic Aspect
branch
Woven dynamic aspect code
Replaced instruction in original code
Jump back to instruction after join point

Figure 7.2: Code splicing

Special precaution has to be taken to avoid corrupting current kernel actions when native code is modified.

The execution of kernel functions may usually be interrupted asynchronously before the execution of a machine instruction (on pipelined processors, this means before an instruction fetch takes place). To avoid race conditions that are hard to discover and debug, a splicing operation should be able to atomically change all bytes comprising the instruction to be replaced.. The instruction that replaces the original one is a *jump* instruction. This instruction redirects control flow to the dynamic aspect; on x86-compatible processors, it uses at least five bytes. On these CPUs, atomic writes can usually be handled in word or double word quantities (4 and 8 bytes, respectively) using the `FIST` or `CMPXCHG8` instructions, so special care has to be taken in order not to corrupt the instructions following the jump instruction.

If, during splicing, more than one instruction has to be replaced with a jump instruction, problems may arise if the second of these replaced instructions is a branch target, i.e., the address of this second instruction is references elsewhere in the code with a jump instruction. A branch to that address would then end up trying to execute a bit pattern that is part of the target address as operation code instead of the original, valid operating code. This may result in problems like program crashes due to an illegal operation code encountered by the CPU or – worse – silent data corruption if the replaced bit pattern is a valid operation code.

However, this situation could only occur in functions written in assembly language, since only five bytes are being replaced. In most cases, an instruction of a high-level language translates more than one machine instruction. No such function could be found inside the x86-related and general parts of the NetBSD kernel, such that the discussed issue does not affect the implementation of TOSKANA.

7.3.4 Weaving

Code splicing is the basic functionality used by the weaver to insert join point shadows. For each of the three possible advice variants, a different action must be undertaken in order to assure correct behavior of the dynamic aspect.

Weaving "before"

Weaving a *before* advice is the simple case. Code splicing is applied to the first instruction of the function involved, this instruction is replaced by a *jmp* instruction to the advice code[4]. After the end of the advice code, the instructions that were formerly at the beginning of the function the aspect was deployed in are inserted (see fig. 7.3). The `ASPECT` macro reserves space for these instructions by inserting `nop` operations before returning from the advice code.

Weaving "after"

Weaving an *after* advice is more difficult. While a *before* advice only has one possible entry point that is easily identifiable from the symbol table, the number of ways to return from a function and the location of the return instructions can not be determined from the symbol table. Thus, to find all possible return points of a function, the memory area the function occupies has to be searched for the pattern of the x86 *ret* instruction (`0xc3`).

Instructions on x86 machines can be of arbitrary length (from 1 to 16 bytes) and need not be aligned to word or longword boundaries. Thus, the weaver must take care not to accidentally identify a value of 0xc3 that is part of something else than a return instruction. To support this functionality, the weaver uses a table containing x86 instruction length information. Starting from the symbol table address entry of the function, the weaver only searches the first byte of possible instructions for the value 0xc3.

[4]The splicing code takes care of padding the replaced code segment with `nop` if required.

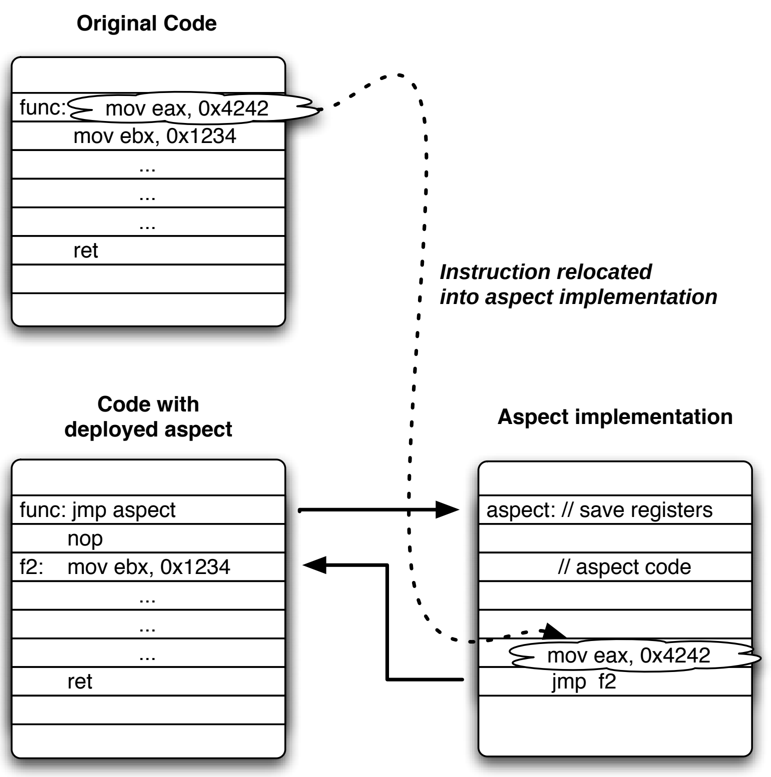


Figure 7.3: Implementing "before"

Weaving "around"

Weaving an *around* advice basically combines the methods used for weaving a *before* and an *after* advice. An additional problem lies in the implementation of the *proceed* macro, as each return instruction of the original function is replaced by a jump instruction to the same specific target address. This return address can not be determined at weaving time, since the advice code may contain more than one *proceed* macro.

As shown in fig. 7.4, the code flow gets redirected dynamically. While the initial call of the function works just like a *before* advice, handling the return case changes depending on the dynamic execution of the *proceed* statements contained in the advice code. Initially, all returns from the function are simply replaced by a jump to a return instruction which is the default in case no *proceed* macro at all is used in the advice code. Whenever the advice code reaches a *proceed* macro, the macro calls a function in the aspect support library module that replaces the target addresses in all positions where originally a return instruction was located with the address of the instruction directly after the `jmp` instruction to the function code itself.

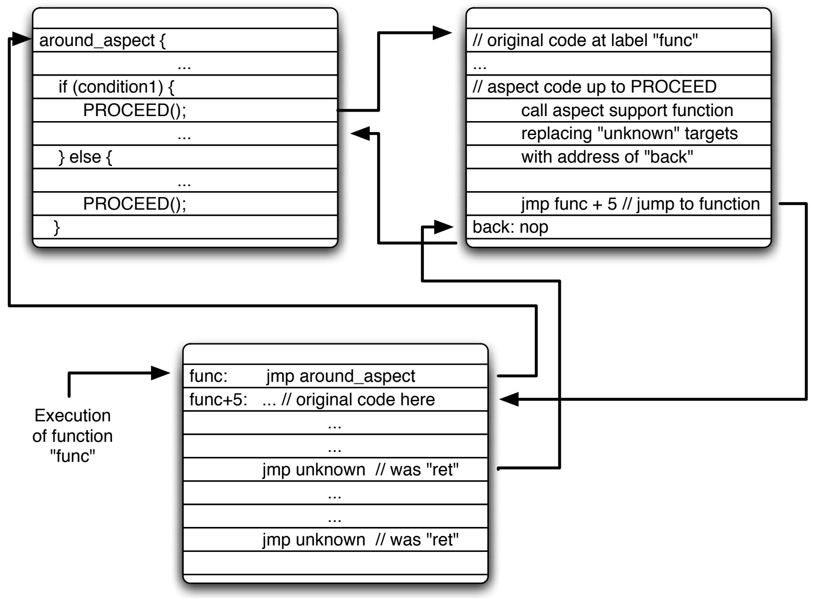

around_aspect {
...
if (condition1) {
PROCEED();
...
} else {
...
PROCEED();
}
// original code at label "func"
...
// aspect code up to PROCEED
call aspect support function
replacing "unknown" targets
with address of "back"
jmp func + 5 // jump to function
back: nop
Execution of function "func"
func: jmp around_aspect
func+5: ... // original code here
...
...
jmp unknown // was "ret"
...
...
jmp unknown // was "ret"

Figure 7.4: Implementing “proceed”

7.3.5 Inlined Functions

One particular problem encountered was the use of functions in NetBSD that are explicitly tagged as “inlined”. With the standard *gcc* compiler optimization setting of `-O2`, code explicitly tagged as “inline” is inlined and a peephole optimizer is applied to the resulting code [130]. This has two implications for applying aspects – first, inlined functions are no longer visible in the kernel, since there is no symbol table information about them and second, the optimizer run that follows the inlining is permitted to re-order instructions, thereby crossing inlined function boundaries, so that in the end the instructions of the inlined function may be mixed up with instructions from the calling context.

Compiling the NetBSD kernel with the `-O0` option disables all inlining (explicit as well as implicit) and all other optimizations. The kernel size for our configuration, however, nearly doubled in size from about 5.5 to more than 9.5 MB. Since further problems executing non-optimized code might occur and the original aim was to deploy aspects into a non-modified kernel, this approach was not pursued further.

This problem is resolved when using more recent versions of GNU gcc and binutils[5]. These contain improved debugging support, which actually enables TOSKANA to provide advice for inlined functions. The weaver, however, had to be extended to be aware of jumps outside of the inlined function context replacing the return instructions used in the original function.

[5]gcc version 3.4.0 and a prerelease version of binutils 2.15

7.3.6 TOSKANA Implementation

TOSKANA consists of a set of interacting user mode programs and kernel modules. The central point of user interaction is the dynamic weaver, `/sbin/weave`. `Weave` is implemented as a daemon running permanently in the background. At startup, `weave` checks if the *aspectlib.ko* support library module is loaded and inserts it into the kernel if needed using `/sbin/kextload`. Then, the current kernel symbol table is read out of the ELF symbol table information in `/dev/ksyms`.

Aspect modules, shown in listing 7.2, are implemented as kernel extension modules. They rely on functions provided by the support library module that instruct the weaver to modify kernel memory via `/dev/kmem` in order to deploy the aspect hooks[6].

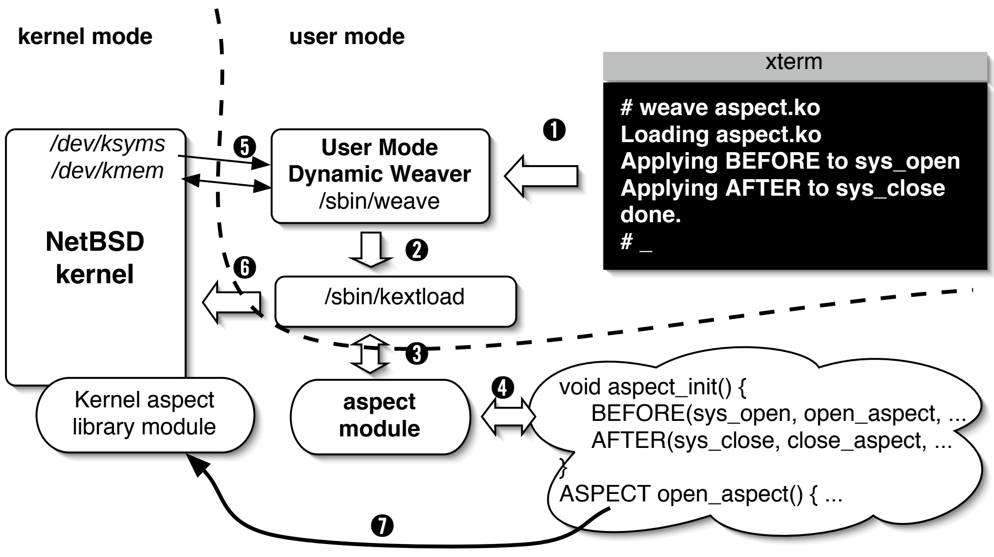
```
#include <sys/aspects.h>
...
void aspect_init(void) {
    /* deploy three aspects */
    BEFORE(sys_open, open_aspect);
    AFTER(sys_open, close_aspect);
    AROUND(func, some_aspect);
}

ASPECT open_aspect(void) {
    ...
}

ASPECT close_aspect(void) {
   ...
}

ASPECT some_aspect(void) {
   ...
   PROCEED();
   ...
}
```

Listing 7.2: A sample aspect module

Fig. 7.5 shows the components and tasks involved in weaving an aspect. Weaving an aspect module is initiated from user mode (1) by invoking the `weave` command. The weave command then checks for availability and integrity of the aspect module (which is built like a standard kernel module, described earlier in this chapter) and extracts the symbolic pointcut and joinpoint information from the module binary.

The join points are checked for validity against the kernel symbol information provided in `/dev/ksyms` (5). If the aspect module exists and contains valid weaving meta-information, the weaver instructs the NetBSD kernel module loader, `/sbin/kextload` to load the aspect module into kernel virtual memory (2). The module loader, then, treats the aspect module like normal (non device driver) kernel modules (4), opens it (3) and loads

[6]This is a modification of kernel space memory. An alternative implementation would be to modify kernel virtual memory directly from the support library module. Using the `kmem` device, however, is less complex since the `kmem` device driver already takes care of setting page protection bits.



```
# weave aspect.ko
Loading aspect.ko
Applying BEFORE to sys_open
Applying AFTER to sys_close
done.
# _
```

```
void aspect_init() {
    BEFORE(sys_open, open_aspect, ...
    AFTER(sys_close, close_aspect, ...
}
ASPECT open_aspect() { ...
```

Figure 7.5: Weaving a dynamic aspect into the kernel

the module into kernel memory, executing the modules initializtion function (6). The initialization function, in turn, calls a support library function that is contained in the kernel aspect library module for each advice (7) – implemented as a function in the same module – that is to be deployed.

Aspect weaving is performed by the NetBSD kernel support library. During kernel runtime, advice functions can be activated and deactivated at runtime from other advice functions as well as from dedicated kernel components (e.g., the scheduler, which can activate and deactivate advice that depends on process context).

Name	Parameters	Function
activate	aspect, advice	activate advice in aspect
deactivate	aspect, advice	deactivate advice in aspect
weave	aspect	weave aspect
unweave	aspect	unweave aspect
proceed	various	proceed with parameters at join point
check_symbol	symbol_name	check if symbol exists
get_symbol_value	symbol_name	get address of symbol
get_loaded_aspects	–	get list of loaded aspects

Figure 7.6: Functions of the TOSKANA support library

The calls to the support library are implemented as macros that specify aspect weaving behavior. Currently, macros for `BEFORE`, `AFTER` and `AROUND` advice are provided. The aspect modules are compiled like standard NetBSD kernel modules. The only addition is the inclusion of the `#include <sys/aspects.h>` header file that provides the macro definitions to interface to the support library. The support library supplies some additional functions to aspect modules, listed in fig. 7.6.

7.4 Evaluation

Two distinct values are of interest when gathering performance statistics of TOSKANA – the time it takes to deploy a dynamic aspect into a running system and the loss of performance when an aspect is deployed.

While the first value can be determined by Unix timestamp measurements (using the time(1) utility), the overhead of a deployed dynamic aspect is analyzed by native code analysis using the published values for instruction timing [75].

7.4.1 Deployment Overhead

The deployment of dynamic aspects is an interesting process to analzye. In the case of a system with several thousands of dynamic aspects deployed, deployment overhead is critical, especially if a system has to be restarted and all aspects have to be deployed again in order to reach a known state.

Deployment overhead is composed of two major parts – the time required by the system module loader to load the aspect into kernel virtual memory and the time required by the in-kernel aspect support library to actually weave the aspects. Additional overhead results from the weaver's reading of the modified kernel symbol table. Averaged timing results for the NULL aspect and a test aspect module containing no functionality except for 10 kB of dummy code that weaves a BEFORE, AFTER and AROUND advice, taken with the Unix time(1) command as well as the `gettimeofday` function, are presented in fig. 7.7. These values were measured on a 1.8 GHz Pentium 4 system using 512 MB RAM which was running a prerelease of NetBSD 2.

Task	Time Null aspect	Time Test aspect
Module loading	1.17s	1.23s
Aspect weaving	0.22s	0.27s
Reading symbol table	1.35s	1.35s

Figure 7.7: Performance of aspect weaving

7.4.2 Execution Overhead

The overhead for executing a deployed dynamic aspect is important, since an inefficient implementation may slow down the system considerably in cases where a dynamic aspect is deployed in often-used places or a large number of dynamic aspects are spread throughout the system. To determine this overhead, the NULL aspect as a simple aspect doing nothing is used. It is the most simple test case possible and serves as a model to calculate aspect execution overhead as well as to execute aspect deployment timing measurements.

The NULL aspect, shown in fig. 7.8, only involves two additional operations - a jump to the deployed advice and immediately following the replaced instruction a jump back to the code location after the splicing point.

Timing of these operations depends on the exact type and revision of the processor used. Since our development mostly takes place inside of a x86 emulator, real instruction timings are not available. With an overhead of only two respectively four instructions – each consisting of a jump instruction to the advice code and one back to the original code, and doubled for the case of “around” – for a deployed aspect (and none for a non-deployed aspect, of course), the performance impact is negligible.

Task	Nr. of instructions
before	2
after	2
around	4
proceed	15 + nr. of ret instructions × 22

Figure 7.8: Overhead calculations for supported advice types

The only case demanding more runtime overhead is the proceed macro, since this involves patching the new return address into the advice code at runtime. Clearly, this depends on the number of returns from a function. With an overhead of 15 instructions fix cost (to call the library) and 22 instructions per return to patch the return address, for usual cases of three to five returns from a function this is in the range of performance loss due to a wrong branch prediction, so it does not have a significant impact.

7.5 Micromeasurements

The measurement results presented here concentrate on micro-measurements of various important parts of TOSKANA's functionality, thus giving an overview of the performance of the overhead introduced by code splicing, weaving as well as activating and deactivating aspects. Macro measurements could also be performed using the measurement technology described below, however, macro measurements are not rerally useful for a number of reasons. The applications presented in chapter 9 that are implemented using TOSKANA draw their improvements, with respect to performance, not from using aspect-oriented technology, but rather from the application itself. This is the case for CXCC as well as for the Appropriate Byte Counting implementation. The third example presented, Grid application confinement, is a security topic. Performance results on a macroscopic scale would rather measure the properties of the Systrace implementation used.

7.5.1 Evaluation Method

Evaluating the performance characteristics of changes to kernel code is a complex task – on the one hand, measuring the impact of a single modification undisturbed by other tasks running in parallel is difficult; on the other hand, getting reliable measurements out of a system running on the bare hardware poses a completely new set of problems.

The measurement method selected for evaluating TOSKANA was chosen to be as close to measuring effects on real hardware as possible. Since no hardware-based CPU emulator was available for the measurements and introspection utilities like Sun's DTrace [23] or KProbes [109], which would be well suited for measuring kernel timing, are currently not available for NetBSD, an approach using the Xen virtual machine [7] was used. For evaluation purposes, NetBSD is running as an unprivileged guest domain on the Xen hypervisor. For this purpose, a modified version of the NetBSD kernel suitable for running on top of Xen was implemented and the entire system was installed on a 2 GHz Pentium M laptop.

The measurements presented below use the remote gdb (GNU debugger) infrastructure provided by Xen. A remote gdb server running in the privileged domain 0 (in this case, a Linux system), is configured to control the NetBSD system running in an unprivileged domU domain. Since Xen has full control over interrupt assignments and memory mappings, the remote gdb server can insert breakpoints at arbitary locations in the execution of any code executing in the domU domain, including the kernel.

The system provided by Xen provides two methods to measure performance. On the one hand, a microsecond-resolution timer is provided in each execution domain that is incremented only when Xen schedules the respective domain to run. On the other hand, the virtual CPU model provided by Xen implements a cycle counter that also only counts the instructions executed in the context of this particular domain. Both values were used for evaluation and, in addition, compared to each other in order to ascertain the validity of the values measured.

The implementation of counters through hardware features of the CPU, together with the domain multiplexing provided by Xen, ensures that the results gathered are as similar as possible to a system using NetBSD running on the bare hardware. Compared to a full-system emulation like qemu [11] which only provides a behavioral model of a system but not accurate timing emulation, running the code multiplexed on the real CPU ensures that all measurements are performed in a cycle-accurate way.

7.5.2 Performance

To evaluate the effects of using TOSKANA in a NetBSD system, a set of measurements was performed. Of special interest for the evaluation was the infrastructural overhead for deploying and using aspects, the cost of saving and restoring state modified by advice code and a simple example to ascertain the results of the previous measurements.

Basic Values

In order to obtain a reference, the basic duration of two specific system calls were measured. The first system call is `sys_getuid`, a call that simply returns the user ID related to the calling process; the second system call is `sys_chmod`, which is used to change the access mode of inodes (files and directories) in a file system. To avoid the influence of I/O operations on the results, all operations were performed on a RAM disk. All measurements were performed in single user mode one hundred times to eliminate the influence of processes running in parallel as well as to filter out inaccuracies.

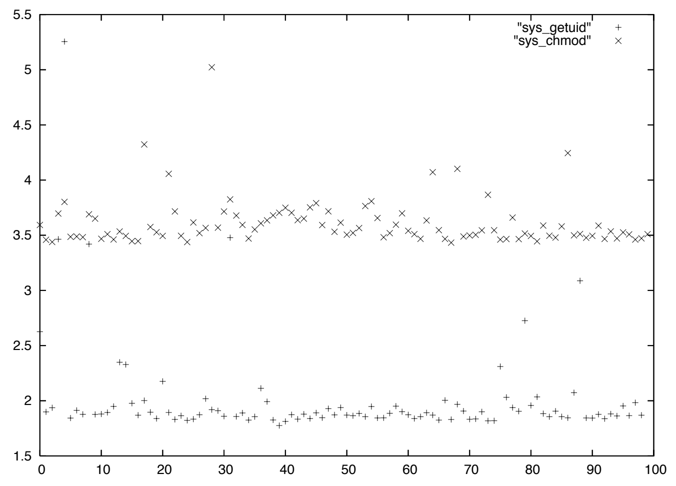


Figure 7.9: Duration [μs] of the sys_getuid and sys_chmod system calls over 100 iterations

Fig. 7.9 shows the distribution of 100 successive calls to the `sys_getuid` function[7]. In the same diagram, timing values for the `sys_mknod` function are given. From the diagrams we can see that while the average time for executing `sys_getuid` is 1.996 μs, the file system access overhead brought the average `sys_chmod` execution time up to 3.610 μs.

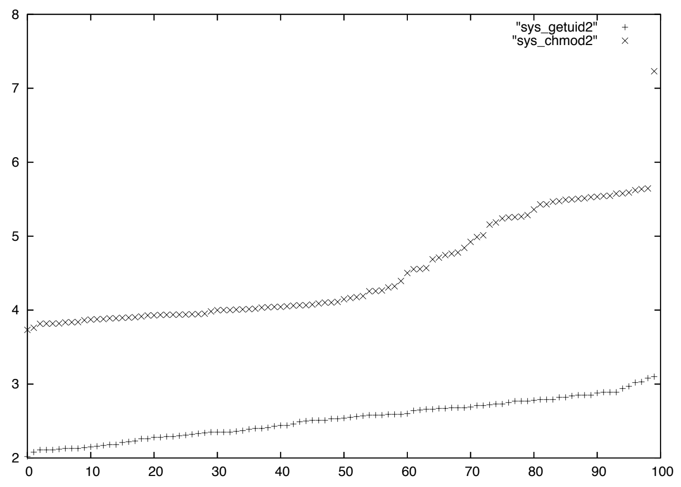


Figure 7.10: Timing [μs] of the sys_getuid and sys_chmod system calls with 1–100 joinpoints

Overhead of Advice Invocation

The next interesting question is what the overhead of invoking the deployed advice code is. Here, calls to an empty advice function are inserted in the code flow of the two system calls. The number of inserted advice calls was increased from 1 to 100; each of these runs was again averaged over 100 iterations. The results, depicted in fig. 7.10, show a linear increase in system call execution time for both system calls up to about 50 advice calls per system call invocation. While the `sys_getuid` call continues to scale in a linear way,

[7]Like all other measurements, these results were taken out of the middle of a larger series in order to eliminate differences due to caching.

the values for `sys_chmod` start growing faster than linear. We attribute these results due to caching effects in the processor; a more precise evaluation of this effect would require a precise CPU model to analyze cache usage and bus contention. The average overhead for calling a single advice function over all samples was 13.99 ns. The time required for a single invocation varied from 8 to 23 ns; again, this is most likely the effect of caching and different behavior in the CPU's branch prediction.

Overhead of Simple Operations

For this measurement, a simple operation (increasing a variable and referencing a local variable) was used, thereby evaluating the overhead of saving context for an advice invocation while providing only minimal execution overhead. The results are quite similar to the previous measurements[8]; the average time for an advice invocation grew from 13.99 to 17.23 ns with samples ranging from 12 to 29 ns.

Thus, the insertion of a simple instruction does not have significant effects on timing; more extreme caching effects will probably only show up with very large advice code functions.

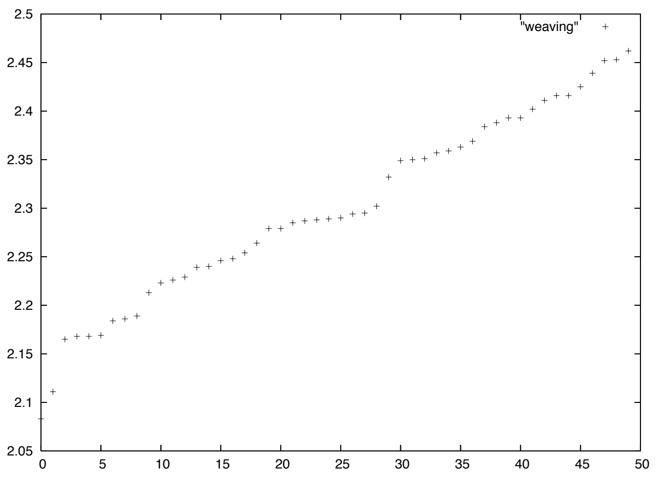


Figure 7.11: Time [s] to deploy an aspect relative to the number of symbols involved

Deployment Overhead

This measurement evaluates the time required to dynamically insert a new aspect into the running system. Measurements were performed with respect to the number of symbols that had to be referenced in a pointcut, since symbol lookup has the most significant on TOSKANA's weaving performance. Figure 7.11 shows the time it takes to weave an aspect that references from one to 50 symbols in the kernel. Since weaving is initiated from user mode, reading the kernel's symbol table initially is the most time-consuming step. In current versions of TOSKANA, the symbol table is not cached, since no mechanism exists with which the kernel informs the weaver about recently loaded kernel modules[9]. This results in an overhead of about 2 seconds to extract the symbols using the `/dev/ksyms` special device. However, this lookup is only required the first time an aspect is used in the

[8]The diagram is not shown since no significant differences can be observed from the diagram

[9]Here, this is only relevant modules other than the aspect modules.

system. After extracting the symbol table, the time required to weave referenced symbols grows linear with the number of symbols.

7.6 Summary

TOSKANA provides the first implementation of kernel mode dynamic aspects in existence. Whereas static AOP in kernel mode and dynamic AOP in user mode have been implemented before, dynamically modifying running kernel code poses special requirements to the stability of the code and careful design in order not to disturb regular kernel interaction.

However, TOSKANA's use of binary code manipulation through code splicing restricts the supportable join point types to those types whose semantics can be recovered from the machine code and symbol table. An advantage of this approach is that aspects can be applied to a completely unmodified kernel – as long as the symbol table is provided, TOSKANA can be deployed into an already running system.

An alternative approach would be to introduce more semantic information into the binary, e.g. by supplying extended symbol table information, as discussed in chapter 6, section “Compiler Hints”. This, however, requires a recompilation of kernel code, which can change the timing behavior and memory requirements of the kernel, making this approach less attractive.

8. AOP in Virtualized Systems

"La perfection est atteinte non quand il ne reste rien à ajouter, mais quand il ne reste rien à enlever."

— Antoine de Saint-Exupery

8.1 Introduction

One of the most featured recent trends in operating systems recently is the virtualization of systems. The idea itself is not new, first examples of virtualized systems can be found in the late 1960s in IBM's VM/370 system [99]. Important later developments in this area include the UCSD P-System [105] and the Smalltalk-80 [63] virtual machine. In general computing, however, the topic of virtual machines lay dormant for a long time, since the performance hit taken by virtualizing the execution environment was too large for use on older systems.

The increase of computing power since the mid 1990s, however, has again brought virtualized systems into the mindset of computer scientists. Similar to the discussion on the various levels of dynamic aspect deployment in chapter 5, the virtualization of a system can occur on several different layers.

This chapter describes the design and implementation of a VM-based dynamic aspect facility for operating systems, TOSKANA-VM.

Parts of this chapter have been published in [51].

8.2 Virtualization

The original meaning of "virtual machine" is the creation of a number of different identical execution environments on a single computer, each of which exactly emulates the host computer. This provides each user or group of users with the illusion of having an entire computer that is their private machine, isolated from other users, all on a single physical machine. The host software which provides this capability is often referred to as a virtual machine monitor or hypervisor.

The second, and now more common, meaning of virtual machine is a piece of computer software that isolates the application being used by the user from the computer. Because versions of the virtual machine are written for various computer platforms, any application written for the virtual machine can be operated on any of the platforms, instead of having to produce separate versions of the application for each computer and operating system. The application is run on the computer using an interpreter or Just In Time compilation.

One of the basic principles of virtualized systems that permits several systems to run concurrently on a system without negatively influencing system integrity is the protection of tasks against malicious or accidential changes to the system state (memory contents, interrupt contexts, resource usage etc.) by other tasks running at the same time. This protection can occur on various layers – from sandboxing in the Java Virtual Machine (JVM) to using hardware-assisted mechanisms in micro- and exokernels and providing a completely simulated system environment in emulators. Each of these approaches has their own set of benefits and drawbacks, which are discussed below along with a description of the approach with focus on its applicability for weaving dynamic aspects.

8.2.1 High-Level Virtualized Systems

Premier examples for a high-level virtualization of systems are virtual execution environments running in user-level, like Sun's Java Virtual Machine [72] and Microsoft's CIL, which is the virtual machine at the basis of the .NET system.

For the sake of simplicity of compiler development, these machines provide a stack-based computing model. The instructions provided by the virtual machine are called "byte codes".

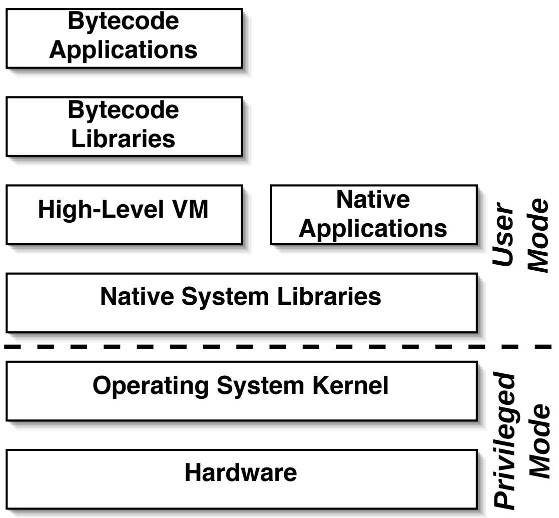


Figure 8.1: Structure of a system using a high-level VM

The structure of a typical system using a high-level VM is shown in fig. 8.1. Here, all code in the system except the applications and libraries running on top of the VM, are represented as native code directly executed by the system's CPU. The VM itself is a regular user-mode process and has no special privileges.

Most dynamic AOP research concentrates on implementing AOP in high-level virtual machines. Steamloom [15], for example, is an implementation of dynamic AOP for the IBM

Jikes Research Java Virtual Machine. Implementing a complete operating system in a high-level VM, however, is relatively awkward. In the JVM, the byte codes compiled from Java source code do not support access to memory locations using pointers.

8.2.2 Paravirtualization

Paravirtualization is a virtualization technique that presents the abstraction of virtual machines with a software interface that is similar but not identical to that of the underlying hardware, i.e., the paravirtualized machine provides the identical instruction set to the underlying CPU. This requires operating systems to be explicitly ported to run on top of the virtual machine monitor (VMM) but may enable the VMM itself to be simpler and for the virtual machines that run on it to achieve higher performance.

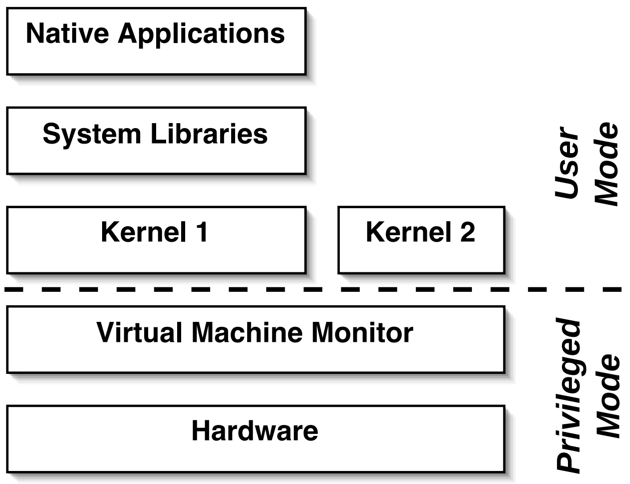


Figure 8.2: A VMM implementing paravirtualization

The performance loss of paravirtualized systems typically is less than 10 % of the original system's speed. This performance gain is achieved by *not* emulating most instructions, but rather executing them directly on the CPU. Only privileged operations like interrupt control and device access, which could change the state of the hardware and thus cause the system to hang, are either emulated by the VMM or removed from the systems running on top of the VMM completely and delegated to strictly controlled routines provided by the VMM.

Using this approach, the operating system instances running on top of the paravirtualization VMM can be run completely in user mode. If the approach of emulating privileged instructions is used, the attempted execution of such a privileged instruction by the CPU causes a special exception to occur, which in turn calls a support routine in the VMM. This support routine parses the bit pattern of the instruction that caused the exception, checks for a valid operation, and emulates the instruction accordingly. This approach causes significant overhead when privileged instructions are executed. Since these privileged instructions occur relatively uncommon, however, most of the original CPU's execution speed is retained.

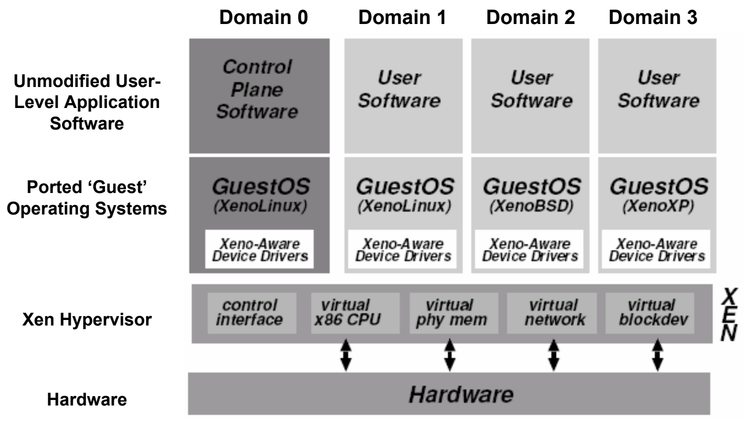


Figure 8.3: Architecture of a Xen-based system

Xen [7] is a VMM implementing paravirtualization for standard x86-based PCs. It provides secure isolation between VMs, resource control and quality-of-service handling at speeds close to the native machine's performance. "Guest" kernels, e.g. Linux, that are to be run on top of Xen have to be adapted, due to the similarity of Xen to the underlying hardware, however, the porting effort is reasonable. The architecture of a typical Xen-based system is shown in fig. 8.3 (taken from [7]). One of the guest operating systems is running as "domain 0", which owns the privileges and has the necessary software tools installed to create, delete and control other domains. Each domain uses its own instance of a guest kernel, running a separate set of user-mode applications on top. Resource management is handled by Xen, which has control over virtual and physical memory, provides a virtual network adapter to interconnect the various OS domains and makes virtual block devices available to all domains.

Other projects that make use of paravirtualization include IBM's rHype hypervisor [74], Microsoft's Singularity operating system [73], and the Denali isolation kernel [142], all of which are detailed in chapter 10.

Paravirtualization is a technology that provides many advantages to operating system developers. However, it does not provide the ability to intercept the execution of arbitrary machine instructions, since these are directly executed by the CPU. Thus, paravirtualization can not be directly used to implement dynamic AOP, but rather needs support, e.g. from the CPU debug unit, in order to weave dynamic aspects.

8.2.3 System-Level Virtual Machines

One virtual machine acting as an execution layer for all applications running on top of the operating system is the Dis virtual machine [42], implemented in Bell Labs' Inferno operating system [43]. The programming language used in Inferno is Limbo [44]. The Inferno kernel, however, is implemented in ANSI C.

Another set of system-level virtual machines is applicable at an even lower level. VVM [108], the "virtual virtual machine", uses an additional step of indirection. Instead of implementing a new virtual machine for each application domain, the VVM virtualizes the virtual machine itself. New specifications of VM adapted for an application domain are loaded on demand in the VVM. These specifications are called “VMlets”. A VMlet contains in a high level language the complete description of the execution environment (bytecodes, syntax, operating system services, API and so on).

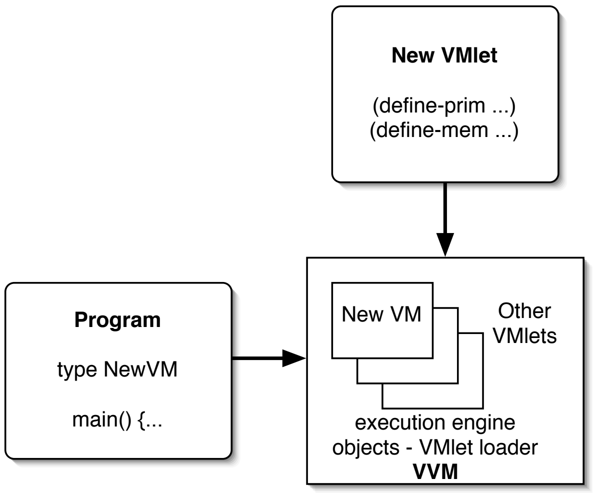


Figure 8.4: Architecture of the “Virtual Virtual Machine” (VVM)

The VMlets are themselves encoded in compact bytecoded programs. They are not just declarative, but imperative and may alter their own execution environment. For instance, according to the environment where they are loaded they may restrain the visibility of given functionalities. Thus, the VVM can be seen as a VM, executing VMlets, that transform the VM to the one contained in the VMlet (see fig. 8.4). Bytecoded applications can then be run on this specialized VM.

The low-level virtual machine, LLVM [86], is a virtual machine system consisting of a compilation strategy for lifetime program optimization, a virtual instruction set similar to RISC processors, and a C and C++ compiler infrastructure based on gcc. Its primary intended use is optimization of programs at compile-time, link-time and run-time, while remaining transparent to the developer. Since this involves analysis and modification of running programs, the LLVM infrastructure provides facilities useful for the weaving of dynamic aspects. LLVM is the VM at the basis of the TOSKANA-VM project and is described in more detail later in this chapter.

The Singularity OS uses the .NET CIL virtual machine to implement operating system functionality. The impact on performance and functionality of the system, however, can not yet be evaluated since no code is published.

Since system-level virtual machines reside on a layer closer to the hardware, it is possible to weave aspects into kernel code, provided the kernel code is implemented in the VM’s bytecodes. TOSKANA-VM uses LLVM in conjunction with the L4 microkernel as support to implement the weaving of dynamic aspects into a L4Linux kernel compiled to bytecodes.

8.2.4 Microkernel-Based Systems

On levels closer to the hardware layer, other virtualization approaches have been developed. The complexity and growing feature set of traditional operating system kernels lead to a rethinking of what constitutes a kernel. Starting with ideas of minimal kernel functionality, e.g. by Brinch Hansen [66], this development lead to the creation of the Mach project at CMU [1] in the late 1980s. Despite being used commercially by several operating systems vendors (Digital OSF/1, NeXTstep, MacOS X) as the base of their Unix-like systems, first-generation microkernels generally suffer from performance problems, as the overhead of crossing the kernel protection boundary for simple functionality like message passing was too high (see fig. 8.5).

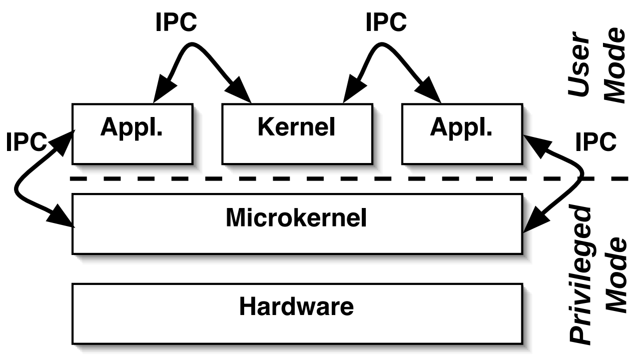


Figure 8.5: A microkernel-based virtualized system

Analyzing the shortcomings of these first implementation approaches, Liedtke started to design the structure of a second-generation microkernel [90], addressing most of the problems in Mach. Based on experience from the L3 kernel project at GMD [89], the L4 microkernel was designed as a minimal kernel, supporting only interrupt handling, task management and a very efficient implementation of inter-process communication (IPC). Kernel tasks implemented in kernel space in first-generation microkernels like memory management, file system handling and device drivers are relegated to user-space processes that communicate with the microkernel exclusively using IPC and shared memory pages.

The API of L4, however, is far too restricted to directly support user mode applications. Like Mach-based systems, a Unix-like system was adopted on top of L4 to serve as one common API. Using Linux as a basis, L4Linux was created as a kernel personality by replacing only the hardware-related parts of the Linux kernel with L4 abstractions invoked using L4 IPC messages.

Other implementations of kernel personalities for L4 are available (e.g., the TU Dresden DROPS [10] realtime kernel) and can run concurrently with L4Linux. Together with the LLVM virtual machine infrastructure, described above, L4 and L4Linux provide the foundation for the VM-based TOSKANA-VM dynamic aspect toolkit.

8.2.5 Exokernels

The idea of minimizing kernel functionality lead to further experiments in minimalism, trying to reduce the functionality of the operating system as far as possible. Traditional operating system functionality can then be implemented as user-mode libraries, bringing operating systems back to their origins as libraries of code commonly used by different applications on early computer systems like IBM's 7094 [36].

This approach, called "exokernels", differs from the concept of paravirtualization in that the OS running on top of the basic execution layer is composed of a set of libraries instead of a traditional monolithic kernel.

While the exokernel approach is interesting, no mainstream implementation has been tried so far. Kaashoek et al. implemented a first version of an exokernel in the MIT Atheos project [53], but the idea of exokernels never caught on. On the one hand, the implementation of an operating system as a set of libraries on top of the exokernel provides possibilities to modularize aspects, on the other hand, this encapsulation again might create a new set of crosscutting concerns to be handled.

8.2.6 Emulation Using Binary Translation

Binary translation is a technology that analyzes the machine instructions of a program prior to their execution and translates them, either into instructions for a different instruction set architecture or into optimized code using the same instruction set. Since the translation process retrieves semantic information from the binary being translated, this technology can also be used to weave dynamic aspects into the native execution stream.

Research projects implementing binary translation include UQBT and Cifuentes [27, 28], which provide a universal translation framework that performs semantic analysis of the machine code to be translated and is extensible to support many source as well as target machines.

One commercial implementation of binary translation is FX!32 by Digital Equipment [26], which was used to execute x86 Windows NT binaries on Alpha processors.

Transmeta [134] processors use an approach that is midway between hardware support and binary translation. The Crusoe microprocessor consists of a hardware VLIW core as the core engine and a software layer called "Code Morphing" software. The Code Morphing software acts as an intermediate layer translating x86 instructions to native Crusoe instructions. In addition, the Code Morphing software contains a dynamic compiler and code optimizer used to improve translation performance.

Binary translation implies startup costs when running a program, since at least parts of the program have to be translated prior to execution. These costs are minimized through the use on on-demand technology. On the one hand, code is only translated when it is actually scheduled for execution at runtime, on the other hand, the translator keeps a cache of the native instruction represenation of already translated pieces of code, such that translation usually only occurs once for code executed multiple times, e.g., in a loop. Many binary translators also make the translated instructions persistent across program runs, providing a near-native representation of the original binary code over time. The translation, however, has to adapt calling conventions, register use, branch predition mechanisms and similar CPU-level semantic information to the peculiarities of the host machine, so that full native emulation speed, compared to executing the compiled source code for the native machine executing the binary translator, is not achievable.

Since binary translation usually takes place at load time, weaving of dynamic aspects it is mostly suited for user-mode applications. Binary translation inside an OS kernel would require an existing low-level infrastructure as a basis for the translator. This could be provided by a paravirtualization VMM. No such projects are currently known, however.

8.2.7 Hardware Virtualization Using Emulators

A related, but different approach is taken by hardware virtualization systems implemented in emulators like VMware [139], VM/370 [99], Bochs [87], Qemu [11], PearPC and a large variety of other emulators for older computer systems like Bob Supnik's simh system [21]. These systems supply the illusion of an existing system, emulated in excrucial detail, in order to run unmodified original software for that system on top of the emulator.

While emulating older systems like Digital's PDP-11 and VAX systems can be done in speeds approximating or exceeding the original system's speed on modern hardware, emulating contemporary hardware implies a large performance drawback. PearPC, for example, emulates a PowerPC running at approximately 40 MHz when running on a current 2 GHz Athlon 64 system.

Emulating complete systems is a way to run legacy applications or to enable device driver development for novel OS concepts. The lack of speed in the emulation process can partially be regained by using just-in-time compilation technologies, similar to the binary translation technology described above. Since the manipulation of operating system code is the goal of using dynamic AOP, code adaptations to a kernel prior to using aspects are acceptable; thus, a completely precise emulation like one provided by emulators is not required.

8.3 TOSKANA-VM: A First Prototype

A first prototypical implementation of an aspect-aware virtual machine for operating system kernel is based on the L4 microkernel, a modified version of the LLVM virtual machine, and a modified version of the L4Linux kernel personality compiled to LLVM bytecode instead of native machine code.

The approach presented here deviates from traditional operating system construction approaches in that not OS kernel functionality is seen as the basis of the system, but rather an aspect-supporting virtual execution environment that can be used to flexibly build kernel functionality on top of it.

8.3.1 Microkernel-Based Systems

Compared to traditional monolithic operating system kernels, a microkernel-based system divides the functionality it provides into several system components that are cleanly separated from each other. At the base of the system, the microkernel itself provides only the absolute minimum functionality of a kernel – in the case of L4 used in TOSKANA-VM, this is restricted to memory management, task management and interprocess communication primitives. All other functionality usually contained in a monolithic kernel is delegated to so-called *kernel personalities*, which are essentially user-mode tasks running as microkernel applications.

8.3.2 L4 System Structure

The basis for all interaction with the hardware in a L4-based system [90] is the microkernel itself. L4 has complete control over the hardware and is the only process in the system running in privileged ("kernel" or supervisor) CPU mode. All other parts of the system run under control of the microkernel in non-privileged user mode. This includes all operating system personalities described in the next paragraph.

L4 is optimized for fast inter-process communication, which is extensively used throughout the system. IPC messages can be sent and received by any task in the system; messaging in L4 is synchronous, so the delivery of IPC information is guaranteed by L4 as soon as the IPC call returns to the caller. In addition, L4 supports a method to share address spaces between different tasks running on top of L4. Using the *flexpages* system, one task can share parts of it's virtual address space with another task with page-sized granularity.

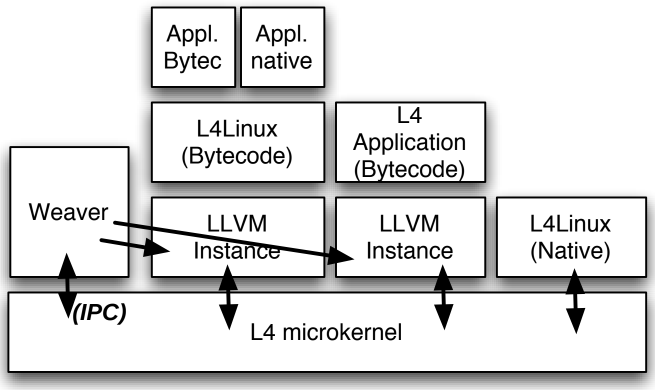


Figure 8.6: L4/LLVM-based TOSKANA-VM System

In order to support virtualization of OS instances, L4 provides abstractions for timer and interrupts as well as virtual memory management and encapsulates these as IPC messages.

An overview of the system is given in figure 8.6. On top of the microkernel, the infrastructure consists of one or more instances of LLVM running in their own address spaces and the weaver as a separate process that handles communication with the VM instances.

Running on top of L4, one or more instances of LLVM execute, this in turn can run L4Linux instances compiled to bytecode or other L4-based applications compiled with LLVM as their target.

The dynamic aspect weaver is a separate component running on top of L4. It receives join point description and weaving/unweaving requests from other tasks in the system and instructs a LLVM instance to add or remove the join point description to respectively from its table of active join points. Then, the weaver inserts the specified advice code in the address space of the particular LLVM instance using shared memory obtained via L4 flex pages.

In this L4-based system, tasks compiled to native code can execute in parallel to the LLVM instances. These tasks provide no support for dynamic aspects.

8.3.3 OS Instances

Since L4 only provides basic kernel functionality, an additional layer of software, shown in fig. 8.6, is required to implement the features required by application programs that L4 is lacking. This is realized in the form of an *operating system personality* that provides the standard interfaces of a traditional monolithic OS kernel to applications running on top of it. In the case of L4, a port of Linux as a personality is available, called *L4Linux* [68]. L4Linux is a modified version of Linux 2.6 in which all critical hardware accesses (interrupt control, page table handling, timer control) is removed and replaced by IPC calls to the underlying L4 microkernel that performs these operations on behalf of the personality (if permitted by the security guidelines).

L4Linux runs as an ordinary user mode process, thus several L4Linux instances are not able to interfere with each other. As a consequence, a virtualization on the level of L4Linux instances running in parallel is feasible and already used in several applications [136, 48].

8.4 The Low-Level Virtual Machine

LLVM is a virtual machine infrastructure consisting of a RISC-like virtual instruction set, a compilation strategy designed to enable effective program optimization during the lifetime of a program, a compiler infrastructure that provides C and C++ compilers based on the GNU compiler collection and a just-in-time compiler for several CPU architectures. LLVM does not implement things that one would expect from a high-level virtual machine. It does not require garbage collection or run-time code generation. Optional LLVM components can be used to build high-level virtual machines and other systems that need these services.

8.4.1 LLVM Instruction Set

The LLVM code representation is designed to be used in three different forms: as an in-memory compiler (providing the intermediate represenation IR), as an on-disk bytecode representation (suitable for fast loading by a Just-In-Time compiler), and as a human readable assembly language representation. This allows LLVM to provide a powerful intermediate representation for efficient compiler transformations and analysis, while providing a natural means to debug and visualize the transformations. The three different forms of LLVM are all equivalent.

The LLVM representation aims to be a light-weight and low-level while being expressive, typed, and extensible at the same time. It aims to be a "universal IR" of sorts, by being at a low enough level that high-level ideas may be cleanly mapped to it (similar to how microprocessors are "universal IRs", allowing many source languages to be mapped to them). By providing type information, LLVM can be used as the target of optimizations: for example, through pointer analysis, it can be proven that a C automatic variable is never accessed outside of the current function.

8.4.2 The LLVM Infrastructure

As fig. 8.7 illustrates, the LLVM compilation strategy exactly matches the standard compile-link-execute model of program development, with the addition of a runtime and offline optimizer. Unlike a traditional compiler, however, the .o files generated by an LLVM static compiler do not contain any machine code at all – they contain LLVM code in a compressed format.

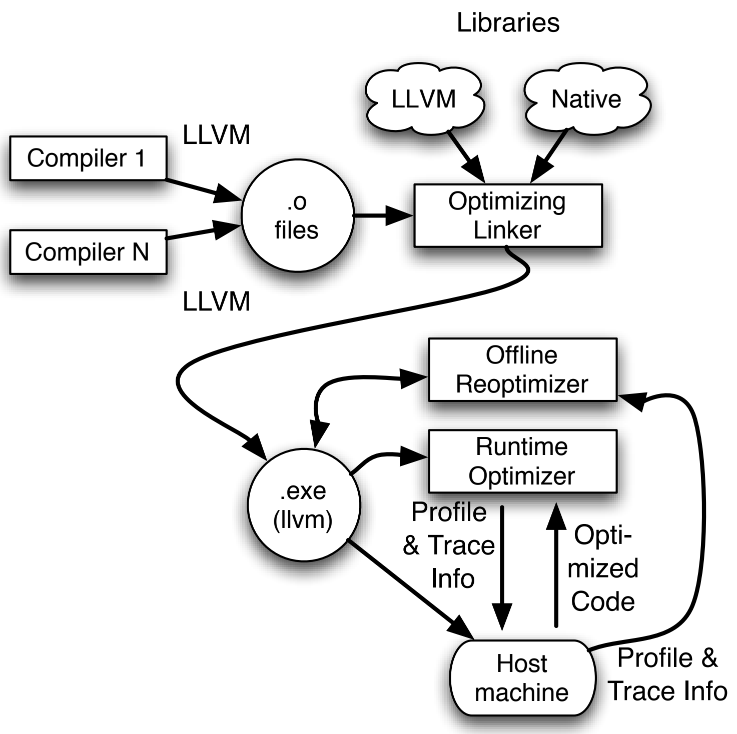


Figure 8.7: LLVM compilation infrastructure

The LLVM optimizing linker combines LLVM object files, applies interprocedural optimizations, generates native code, and links in libraries provided in native code form. An interesting feature of the executables produced by the optimizing linker is that they contain compressed LLVM bytecode in a special section of the executable itself.

Once the executable has been produced, the developers (or end-users) of the application begin executing the program. Trace and profile information generated by the execution can be optionally used by the runtime optimizer to transparently tune the generated executable.

The system heavily depends on having a code representation with the following qualities:

The bytecode is be high-level and expressive enough to permit high-level analyses and transformations at linktime and in the offline optimizer when profile information is available. Without the ability to perform high-level transformations, the representation does not provide any advantages over optimizing machine code directly.

Also, the code has a dense representation, to avoid inflating native executables. Additionally, it is useful if the representation allows for random access to portions of the application code, allowing the runtime optimizer to avoid reading the entire application code into memory to do local transformations.

Finally, the code is low-level enough to perform lightweight transformations at runtime without too much overhead. If the code is low-level enough, runtime code generation can be cheap enough to be feasible in a broad variety of situations. A low-level representation is also useful because it allows many traditional optimizations to be implemented without difficulty.

8.5 Implementation Issues

When handling aspect interaction with native code, the interception of machine instruction execution is usually not supported by standard CPUs. Consequently, the join point model is quite restricted and dynamically inserting join point shadows is an intricate task, since self-modifying code is used in the process.

Using a virtual machine instead of executing code directly on the CPU provides improved methods of detecting possible join points, since all instructions are now either directly interpreted by the VM or translated into short native code segments by the just-in-time (JIT) compiler.

Using LLVM gives the aspect weaver enhanced control over the execution of (VM bytecode) instructions. Instead of having to rely on self-modifying code at program runtime, the infrastructure executing the (byte-)code itself can now be instructed to intercept instruction execution, thereby providing much more detailed information about the current state of the machine.

Based on LLVM, TOSKANA-VM is able to supply a broader range of join point types. The types currently implemented are described below, accompanied with an explanation of the basic VM functionality executed to support them.

Execution and Call

LLVM provides two function call instructions, which abstract away the calling conventions of the underlying machine, simplify program analysis, and provide support for exception handling. The simple call instruction takes a pointer to a function to call, as well as the arguments to pass in (which are passed by value). Although all call instructions take a function pointer to invoke (and are thus seemingly indirect calls), for direct calls, the argument is a global constant (the address of a function). This common case can easily be identified simply by checking if the pointer is a global constant. The second function call instruction provided by LLVM is the invoke instruction, which is used for languages with exception handling.

When using code splicing, no clean distinction can be made between *execution* and *call* join points, since the only point at which the weaver can be certain that the respective function was actually executed is within the code of the function itself.

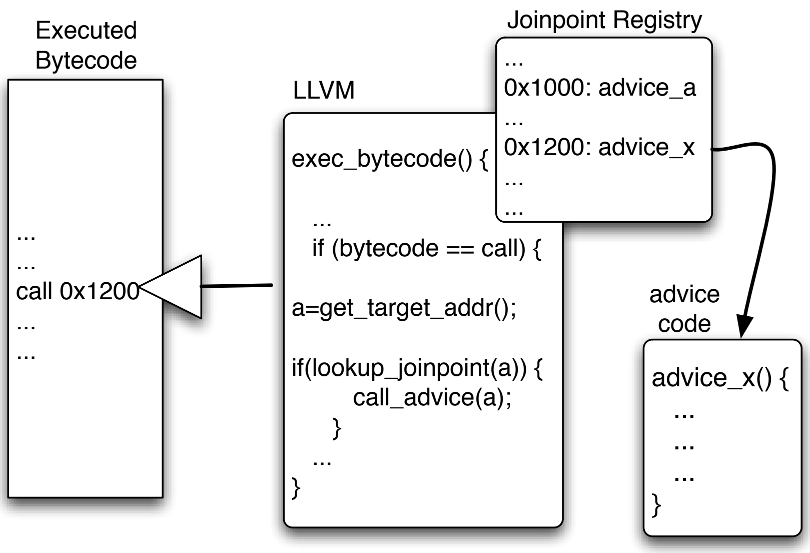


Figure 8.8: A *call* join point

Implementing *call* join points, as depicted in fig. 8.8, requires an interception of the call instruction described above. The target of the call instruction has to be compared against a list of active join points and the related advice code is called appropriately.

Execution join points – illustrated in fig. 8.9 – however, have a different semantics, since they exist at a point when the body of code for an actual method is executed. Thus, interception of the call is not sufficient; rather, the bytecode instruction pointer of the currently executing instructions has to be monitored. As soon as program execution enters (or leaves) the range of addresses defined by the particular function, advice code can be called.

Variable Assignment and Access

Capturing variable assignment and access was not possible using splicing, since accesses to variables in native code not only occur using a direct address reference (which could be detected), but more commonly using pointers to variables contained in registers or calculated target addresses that were not available to the splicing process.

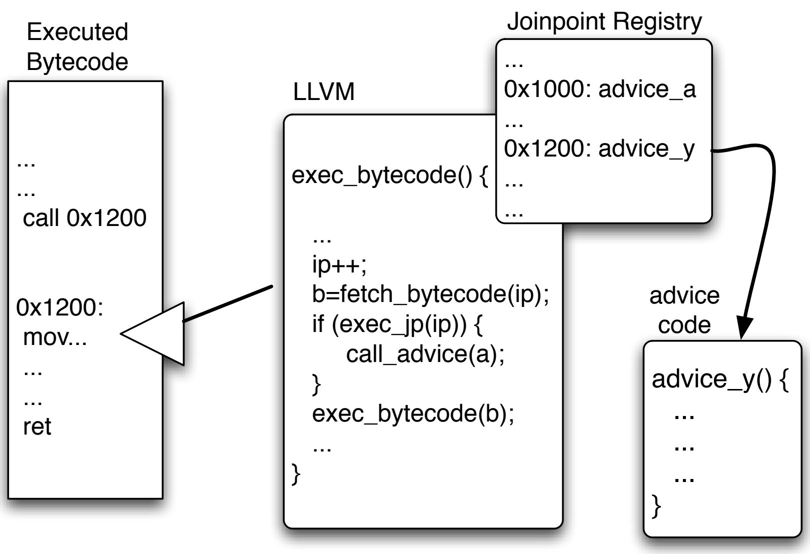


Figure 8.9: An *execution* join point

In LLVM, however, the final address of every read or write instruction executed by the VM is well-known. Hence, interception of read accesses (i.e., variable read) and write accesses (i.e., variable assignment) can be intercepted and corresponding advice code can be executed as the address of the variable is well-known from the symbol table.

A problem with join points triggering on variable accesses lies in variables that are stored in registers for the sake of faster access times. Here, the VM provides a mapping from memory addresses to register contents; however, a tracking of variables in registers in the VM is necessary which is time-consuming. Thus, currently only *volatile* variables and variables to which an address operator (&) has been applied can be used as variable access join points.

8.6 Evaluation

One drawback of the TOSKANA-VM structure is that aspects cannot be deployed into the L4 microkernel itself. While this may not pose a significant restriction for most use cases, the pervasive use of aspects on all levels of the system is desirable. An approach using only a minimal execution layer – exokernels or improved paravirtualization with hardware support for interception instruction execution – could provide a less limited set of functionality here.

Recently, the L4 developers have abandoned the L4Linux approach in order to concentrate on pre-virtualization technologies in the "afterburning" project [137]. This pre-virtualization technology using compile-time hints to support virtualization is a technique to build virtual machines that combines the best features of the two prominent virtualization technologies, pure virtualization and para-virtualization. Since this technology is quite new and unproven, its relevance to implementing dynamic aspects has not yet been evaluated.

8.7 Summary

The TOSKANA-VM approach implements support for dynamic aspects on a higher abstraction level compared to the original TOSKANA implementation by relocating the weaving functionality into a microkernel-based virtual machine that executes kernel personality code compiled into VM byte codes. This introduces two small disadvantages – the microkernel code itself cannot be subjected to aspect deployment, and the kernel personalities on top of microkernel and VM suffer from a performance hit due to the interpretation or just-in-time compilation overhead of the VM, respectively.

This performance hit can be reduced by using a virtual machine that is optimized for in-kernel usage semantics, e.g., by directly supporting locking and page table operations as atomic VM instructions. Furthermore, performance could be improved by using domain-specific languages for large functional units of the kernel like the network stack, scheduling functions and device drivers, which could be translated into higher-level byte codes, which in turn perform better. A concept for an aspect-oriented multi-server OS using domain-specific languages for various components is presented in chapter 11.

■

9. Applications

"To the systems programmer,
users and applications serve only to provide a test load."

— Unix fortune

9.1 Introduction

No technology is useful on its own, after all AOP in kernel mode is intended to be a tool that makes handling crosscutting concerns easier for the programmer. This chapter describes several applications extending kernel functionality that are either improving performance of the system or adding useful functionality.

Related work has explored the use of static AOP to solve in-kernel crosscutting problems like path-specific optimization [34], disk quota management and prefetching [31], page daemon activation and blocking control in device drivers [30], interrupt synchronization [124], product family management and aspect-oriented refactoring of architecture-dependent kernel code [59].

The applications presented in this chapter cover a broad, novel range of problems. Though the actual implementation of the aspect code running in TOSKANA uses the macro API set described in chapter 6, many of the examples shown here use the AspectC notation to improve readability. To obtain an impression of the original TOSKANA programming model, some examples (notably, those concerning autonomic computing are being presented in the original TOSKANA syntax.

Two important structural properties of the NetBSD kernel discussed in chapter 2 are cross-layer networking applications, demonstrated by an improvement of congestion control mechanisms, and cross-layer functionality in file systems, demonstrated by an optimization of virtual memory allocation for file system operations using optimized data structures.

Grid confinement is an application for aspects that not only crosscuts all system call entry and exit points in the kernel, but in addition is a use case for the novel notion of process context join points.

Autonomic computing has many facets, all of which show crosscutting behavior. For each of the central areas of the autonomic computing initiative – self-configuration, self-healing, self-optimization and self-protection – a short aspect-oriented example using TOSKANA is shown that demonstrates the essence of the particular specific autonomic functionality.

The chapter closes with application ideas from various areas that show a broad range of use cases for aspects ranging from audio control to process checkpointing.

9.2 Cross-Layer Congestion Control

RFC 3465 [2] is a proposal for a modification to the algorithm for increasing TCP's congestion window that improves performance of the network stack as well as security.

TCP uses two algorithms for increasing the congestion window. During steady-state, TCP uses the Congestion Avoidance algorithm to linearly increase the value of the congestion window size, `cwnd`, shown in fig. 9.1. At the beginning of a transfer, after a retransmission timeout or after a long idle period (in some implementations), TCP uses the Slow Start algorithm to increase `cwnd`, exponentially. According to RFC 2581, slow start bases the cwnd increase on the number of incoming acknowledgments. During congestion avoidance, RFC 2581 allows more latitude in increasing `cwnd`, but traditionally implementations have based the increase on the number of arriving ACKs.

"Appropriate Byte Counting" (ABC) is a set of modifications to these algorithms to increase `cwnd` based on the number of bytes being acknowledged by each arriving ACK, rather than by the number of ACKs that arrive. The ABC algorithm improves performance by mitigating the impact of delayed ACKs on the growth of the window size.

Simultaneously, the ABC algorithm provides window size growth in direct relation to the probed capacity of a network path, resulting in a more appropriate response to ACKs that only cover small amounts of data, i.e., ACKs of less than a full segment size, compared to the standard ACK counting method.

This more appropriate cwnd growth can improve both performance and can prevent inappropriate cwnd growth in response to a misbehaving receiver. In some cases, however, the modified cwnd growth algorithm causes larger bursts of segments to be sent into the network. This can, in extreme situations, lead to a non-negligible increase in the drop rate and reduced performance.

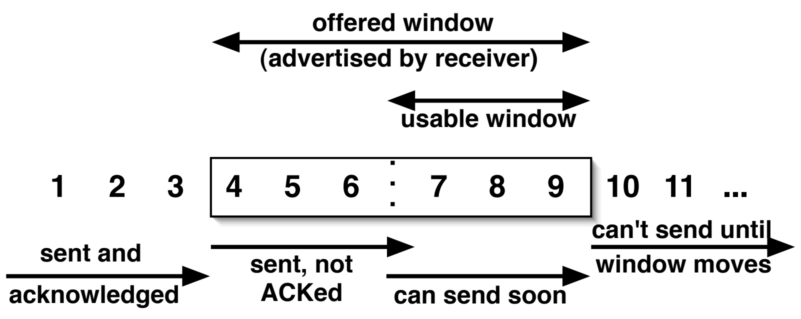


Figure 9.1: TCP congestion window in operation

9.2.1 Crosscutting Properties

To implement congestion window management, the original BSD TCP/IP stack implementation counts the number of incoming as well as outgoing packets. The counting takes place in the routines that handle TCP input and output and in functions that handle packet timeouts and buffer space management (e.g., in case of overflows). The packet counting is scattered over these functions and tangled with the respective code. A new implementation that implements ABC instead of simple packet counting affects the same locations.

In detail, the implementation of appropriate byte counting affects the following kernel source files:

- **src/sys/netinet/tcp_input.c** – 5 locations

  The function affected most is the tcp_input function itself, which handles the reception of TCP packets. Here, the behavior of the cwnd calculation is adapted.

  The other locations where the ABC implementation crosscuts the functionality set the variable `tp->snd_wacked` which counts the number of acknowledged packets, according to the number of correctly or incorrectly received packets.

- **src/sys/netinet/tcp_output.c** –1 location

  In this file, only the function `tcp_output` is affected. Here, at the end of the function, the variable `tp->snd_wacked` has to be set according to the number of ACK packets sent.

- **src/sys/netinet/tcp_subr.c** – 1 location

  Here, the `tcp_quench` function also has to keep count of the number of acknowledged packets.

- **src/sys/netinet/tcp_timer.c** – 6 locations

  In `tcp_timer.c`, the function `tcp_revert_congestion_state` resets the congestion state indicators after a predefined timeout. Here, statistics about the number of acknowledged packets are kept and the counter is reset after a timeout.

9.2.2 AOP Implementation

The main functionality is to appropriately adjust the byte count. Listing 9.1 shows the relevant excerpt from the advice function that calculates the window size:

```
/*
 * Open the congestion window.  When in slow-start,
 * open exponentially: maxseg per packet.  Otherwise,
 * open linearly: maxseg per window.
 */
if (tp->snd_cwnd <= tp->snd_ssthresh) {
        u_int abc_sslimit =
            (SEQ_LT(tp->snd_nxt, tp->snd_max) ?
             tp->t_maxseg : 2 * tp->t_maxseg);

        /* slow-start */
        tp->snd_cwnd += tcp_do_abc ?
            min(acked, abc_sslimit) : tp->t_maxseg;
} else {
        /* linear increase */
        tp->snd_wacked += tcp_do_abc ? acked :
            tp->t_maxseg;
        if (tp->snd_wacked >= tp->snd_cwnd) {
                tp->snd_wacked -= tp->snd_cwnd;
                tp->snd_cwnd += tp->t_maxseg;
        }
}
tp->snd_cwnd = min(tp->snd_cwnd,
                    TCP_MAXWIN << tp->snd_scale);
```

Listing 9.1: Advice code implementing ABC

The functionality most scattered throughout the source files, however, is the code to reset the variable counting the number of acknowledged packets. Resetting the `tp->snd_cwnd` counter is handled by a very simple advice function (listing 9.2):

```
advice reset_wacked (struct tcpcb *tp) {
    tp->snd_wacked = 0;
}
```

Listing 9.2: Resetting snd_wacked

Since appropriate byte counting can adversely affect the performance of TCP transfers, it is important that the ABC algorithm can be activated and deactivated on demand. This activation mechanism makes use of TOSKANA's dynamic aspects. In the current version, however, activation and deactivation has to be performed manually. An autonomic functionality that determines the appropriate time to switch the behavior would have to observe TCP input and output performance and activate the aspect accordingly. This monitoring functionality, in turn, is another crosscutting concern.

9.2.3 Optimizing TCP Throughput for Wireless Networks

The TCP/IP protocol suite, though ubiquituos, is not optimized for the use in wireless hop-to-hop networks, where an underlying broadcast medium is present. Here, the protection of the network from excessive load is required to ascertain a working system for all nodes. One way to enable this is to modify the congestion control provided by TCP/IP. Compared to the previous example, however, this approach is not universally applicable to hosts implementing TCP/IP. Rather, it is an optimization only useful on low-bandwidth shared broadcast media like IEEE 802.11-based WLANs. The end-to-end congestion control provided by TCP, therefore, is not an ideal protocol for wireless hop-to-hop networks. A more appropriate approach is to implement a hop-to-hop congestion control strategy, as described below.

The CXCC (cooperative crosslayer congestion control) approach, described in [116], uses implicit feedback from the next node along a route, obtained by overhearing this node's transmissions. Hence, it does not require additional overhead. The implicit feedback is

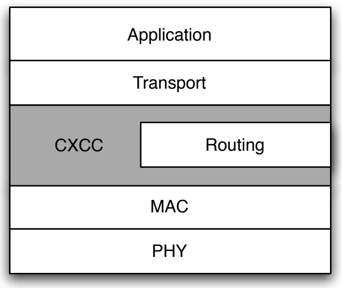


Figure 9.2: Cross-layer properties of CXCC

combined with a tight queue length limitation at each node, which yields a congestion control method that not only avoids explicit feedback wherever possible, but additionally does not require any explicit congestion control action at all: the congestion control itself, too, happens implicitly, just by following the protocol's forwarding rules. As an immediate result, the algorithm is not only able to perform congestion control for connection-oriented, TCP-like traffic, but it works equally well for connectionless, UDP-like traffic and can easily be extended to other forms of communication such as multicasting, flooding and geocasting.

CXCC is not intended to replace the functionality of the layers it encompasses. Instead, the responsibilities of these layers and their interfaces are slightly changed. An overview of the altered architecture is shown in figure 9.2.

CXCC alleviates network congestion by limiting the number of packets queued for each flow in each single node. This queue length constraint is not maintained by dropping packets exceeding the limit, but by pausing the transmission of a flow by an intermediate node once the limit has been reached by its successor on the path to the destination. In the basic variant of CXCC, only a single packet of each flow is allowed to be queued at each hop along the route. To decide whether a packet may be sent, a node has to know whether the downstream node still has a packet of the same flow in the queue. This information could be sent explicitly, but such a mechanism would induce additional traffic and increase the congestion of the network. In CXCC, the shared nature of the medium is used to gain the necessary information at no additional cost. Because every transmission is a de-facto broadcast and the next hop generally is within communication range, the forwarding of a packet by the downstream node can be overheard. This indicates that the next packet may be transmitted.

At the same time, this implicit forwarding notification can serve as an acknowledgment, indicating successful packet delivery to the next hop. Common MAC layer acknowledgments are therefore no longer required. This yields an additional benefit: during the usual DATA-ACK-sequence as used in 802.11, collisions may occur as well with the data packet transmission as with the ACK, therefore an area around both sender and receiver is affected by the transmission. Although the ACK packet is very small, it may well collide with and destroy any larger packet whose transmission is in progress. Eliminating the need for the transmission of an explicit ACK reduces the area of the network affected by the transmission.

Crosscutting Properties.

CXCC acts as a kind of wrapper around the routing layer of the TCP/IP stack. Thus, it has to interfere with packet transport from the transport to the routing and from the routing to the MAC layer and handle the queue lengths and ACK generation accordingly.

The following files are affected by the implementation of CXCC:

- **src/sys/netinet/ip_input.c** – 3 locations

  The function `ip_input` handles IP input and defragmentation. Here, packets have to be sent to the CXCC implementation insted of directly to the transport layer.

- **src/sys/netinet/tcp_input.c** – 5 locations

  Here, the `tcp_input` function has to be modified in order to implement the modified ACK processing.

- **src/sys/netinet/tcp_output.c** – 6 locations

  In `tcp_output`, the function `tcp_output` generates ACKs. This generation behavior is changed.

- **src/sys/netinet/ip_state.c** – 4 locations

  Functions in `ip_state` handle TCP option processing and parts of the TCP timeout handling. Here, changes to functions `fr_tcpstate` and various window size handling functions have to be made.

- **src/sys/netinet/tcp_userreq.c** – 5 locations

  In `tcp_userreq`, user issued connection requests (e.g., termination) are handled. Here, ACK handling for forcibly closed connections has to be modified in several functions.

AOP Implementation.

The "wrapped" implementation of CXCC is handled by one aspect consisting of several advice functions. These functions handle, in turn, redirection of traffic between the transport and routing layers as well as between the routing and MAC layers. ACK generation and surveillance of the broadcast medium for ACKs generated by remote receivers are realized.

Dynamic aspects are required to guarantee correct operation of the congestion control subsystem for hosts that maintain connections to wired and wireless hop-to-hop networks at the same time, like wireless access points. Depending on the destination of the network packet in question, CXCC congestion control must be either active or inactive. This can be achieved by activating and deactivating the aspect; otherwise, two mostly similar network stacks with different congestion control mechanisms would have to be employed.

```
/*
 * Check for ACK/NACK
 */
```

```
pointcut data_packet_received(int type, struct ackpkt *nmo):
    cflow(tcp_input)
    && calls(int tcp_recv(struct ackpkt *nmo, int type, ..));

before(data_packet_received):
{
    if (type == NPT_NACK) {       // This is a NACK
      set_header(nmo->getHead())->retry() = 0;
      // Reset retry count, so the real packet and not a RFA is sent
      nmo->stopTimer();        // Resend immediately
      sendPacket();
    } else {                             // This is an ACK
      nmo->shift();
      if (type == NPT_DATA) { // Impl. ACK via data pkt => upd. tmout
        influence_timeout(nh->queue_delay());
      } else if (type == NPT_RFA) { // Impl. ACK via RFA => upd. tmout
        influence_timeout(nh->queue_delay());
      }
    }
}
```

Listing 9.3: Advice code implementing ACK control

Listing 9.3 shows an excerpt from the handling of positive and negative acknowledgements. Here, the retry delay timer is stopped whenever a negative ACK for a packet arrived, causing the packet in question to be retransmitted as soon as possible. In case of a positive ACK, the timeout values for ACK handling are modified for two kinds of implicit ACKs occuring. This handling is an advice function activated whenever a joinpoint described by the pointcut `data_packet_received`, which matches all calls to the `tcp_recv` function to handle received packets inside the execution of `tcp_input`, is encountered.

9.3 Cross-Layer File System/VM Optimization

File system management and virtual memory are closely interdependent in Unix-based systems. Memory pages are used to buffer blocks from disk device drivers and prefetch information from sequentially read files. In addition, Unix allows user mode processes to map a file's contents sparsely into the process virtual memory space. As a consequence, every read or write operation on a file system involves changes in the virtual memory state. The memory pages related to the specific file being processed are usually maintained in a simple tree-based data structure, which provides suboptimal search performance. Replacing this implementation with a more efficient tree, such as a Red-Black tree, can improve file handling performance.

9.3.1 Red-Black Trees

A red-black tree is a type of self-balancing binary search tree. A red-black tree's implementation is complex compared to that of other tree forms, but provides a good worst-case running time for its operations and is efficient in practice: it can search, insert, and delete in $O(logn)$ time, where n is the number of elements in the tree.

In a red-black tree, as seen in fig. 9.3, each node has a color attribute, the value of which is either red (shown as light grey) or black. In addition to the ordinary requirements imposed on binary search trees, the following additional requirements of any valid red-black tree hold.

1. A node is either red or black.
2. The root is black.
3. All leaves are black (or null).
4. Both children of each red node are black.
5. The paths from each leaf up to the root contain the same number of black nodes.

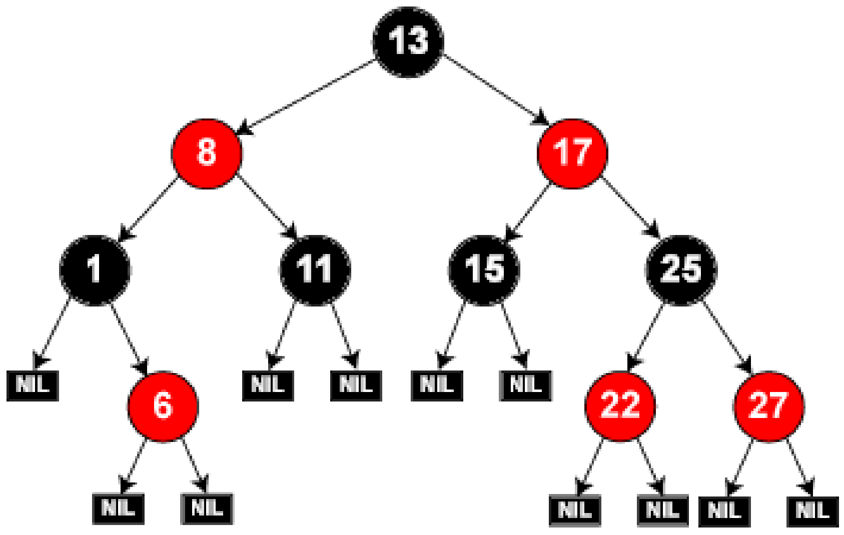


Figure 9.3: A red/black tree

These constraints enforce a critical property of red-black trees: that the longest possible path from the root to a leaf is no more than twice as long as the shortest possible path. The result is that the tree is roughly balanced. Since operations such as inserting, deleting, and finding values requires worst-case time proportional to the height of the tree, this theoretical upper bound on the height allows red-black trees to be efficient in the worst-case, unlike ordinary binary search trees.

Red-black trees are used to optimize entry retrieval in the in-kernel virtual memory mapping structures, `vm_map`. This speeds up memory allocation and memory referencing if there are many `vm_map` entries.

9.3.2 Crosscutting Properties

The implementation of red-black trees affects only one source file, `uvm_map.c`, in 24 separate locations. The tree management crosscuts all functionality and is scattered over all functions operating on virtual memory pages like creating, preparing, modifying and deleting VM entries and checking for the correct alignment of memory pages.

The functions affected by introduction of red-black-trees are:

- **uvm_map_clip_start** – 5 locations

  This function ensures a correct alignment of `vm_entry_start`, resulting in reliable page alignment strategies.

- **uvm_map_clip_end** – 4 locations

  Analogous to `uvm_map_clip_start`, the uvm_map_clip_end function encures that the end of allocated VM entries has a corrent alignment.

- **uvm_map_prepare** – 5 locations

  This function starts a new uvm mapping by preparing the appropriate data structures, e.g. when a new process is created.

- **uvm_map_enter** – 4 locations

  Using this function, a new uvm page entry is created.

- **uvm_map_replace** – 6 locations

  This function exchanges a reserved uvm entry (which is prepared, but not yet assigned) with real mappings.

Throughout these functions, red-black-tree operations have to be performed in order to correctly index all uvm entries.

The example shown in listing 9.4 is from the function `uvm_map_enter`, which creates a new uvm mapping:

```
/*
 * map entry is a valid blank!   replace it.   (this does all the
 * work of map entry link/unlink...).
 */

if (newents) {
        last = newents->prev;

        /* critical: flush stale hints out of map */
        SAVE_HINT(map, map->hint, newents);
        if (map->first_free == oldent)
                map->first_free = last;

        last->next = oldent->next;
        last->next->prev = last;

        /* Fix RB tree */
        uvm_rb_remove(map, oldent);

        newents->prev = oldent->prev;
        newents->prev->next = newents;
```

```
		map->nentries = map->nentries + (nnewents - 1);

		/* Fixup the RB tree */
		{
			int i;
			struct vm_map_entry *tmp;

			tmp = newents;
			for (i = 0; i < nnewents && tmp; i++) {
				uvm_rb_insert(map, tmp);
				tmp = tmp->next;
			}
		}
	} else {

		/* critical: flush stale hints out of map */
		SAVE_HINT(map, map->hint, oldent->prev);
		if (map->first_free == oldent)
			map->first_free = oldent->prev;

		/* NULL list of new entries: just remove the old one */
		uvm_map_entry_unlink(map, oldent);
	}

	uvm_tree_sanity(map, "map_replace␣leave");
```

Listing 9.4: Excerpt from `uvm_map_enter`

In this function, calls to the various `uvm_rb` functions are scattered and tangled with the ordinary uvm entry creation.

9.3.3 AOP Implementation

The aspect for red-black-tree searches implements a basic red-black-tree data structure along with advice functions operating on the tree. The advice functions are:

- **uvm_map_spacefits** – checks if the uvm map has free capacity
- **uvm_rb_augment** – adds vm information to a tree entry
- **uvm_rb_space** – checks the amount of free space in the tree
- **uvm_rb_subtree_space** – same for a given subtree
- **uvm_rb_fixup** – traverses tree elements
- **uvm_rb_insert** – inserts an element into the tree
- **uvm_rb_remove** – removes an element
- **_uvm_tree_sanity** – ensures tree integrity

The complete implementation of the aspect spans more than 500 lines, mostly consisting of advice code that implements the tree operations. The C code implementing the red-black trees was adapted from an ANSI C implementation found in [133].

9.4 Grid Application Confinement

In Grid systems [56], unused computing resources available on the Internet worldwide are used to solve distributed computing problems. When deploying applications in an ad-hoc manner on systems participating in a Grid [121], this implies running unknown and untrusted code on these machines with the potential danger of capturing spyware or other kinds of malicious software.

This software, however, can be sandboxed so that the process is running in a mode restricted by either a permit- or a deny-list of system calls and system call parameters. Using this method, an untrusted process can only read and write files and directories permitted by its sandboxing policies, regardless of Unix permissions or access control lists. In addition, access to other system resources, like process status, system load, disk space etc., can be controlled.

9.4.1 Native Code in Ad-Hoc Grid Applications

For Globus-based Grid jobs [55] written in Java, the actions of Grid applications can be sufficiently restricted by the JVM sandbox model [78]. Many applications, however, have to rely on legacy binary code for certain functionality or simply to improve performance. This legacy code is usually integrated into the application using shared libraries that communicate with the Java code using the Java Native Interface (JNI) [128]. This code executing natively on the CPU of the system running the Grid application is not restricted by the Java sandbox and can, in turn, execute all operations a regular native executable on the specific operating system can – including reading arbitrary files and producing buffer overflows in system libraries.

Several approaches to confining native code exist; these range from running the code in separate virtual machine instances [91] and using small-scale virtualization solutions for kernel-level confinement to monitoring and restricting system calls. The approach of restricting system calls relative to a process context is implemented in the Systrace package [110] that is available for various BSD-based operating systems and Linux. The structure of a Systrace-based sandbox is shown in fig. 9.4.

Normally, an application has access to any system call it requests. In Unix-based systems, nothing prevents a program from searching through “interesting” public readable files in users’ home directories or opening a network connection that sends unsolicited emails to other hosts. Such behaviour, of course, is unacceptable for a Grid application; however, the simple solution of restricting the native parts of the application by denying file system and network access is too limiting, since native code may well read and write temporary files or connect over the internet in order to check for a valid run-time license.

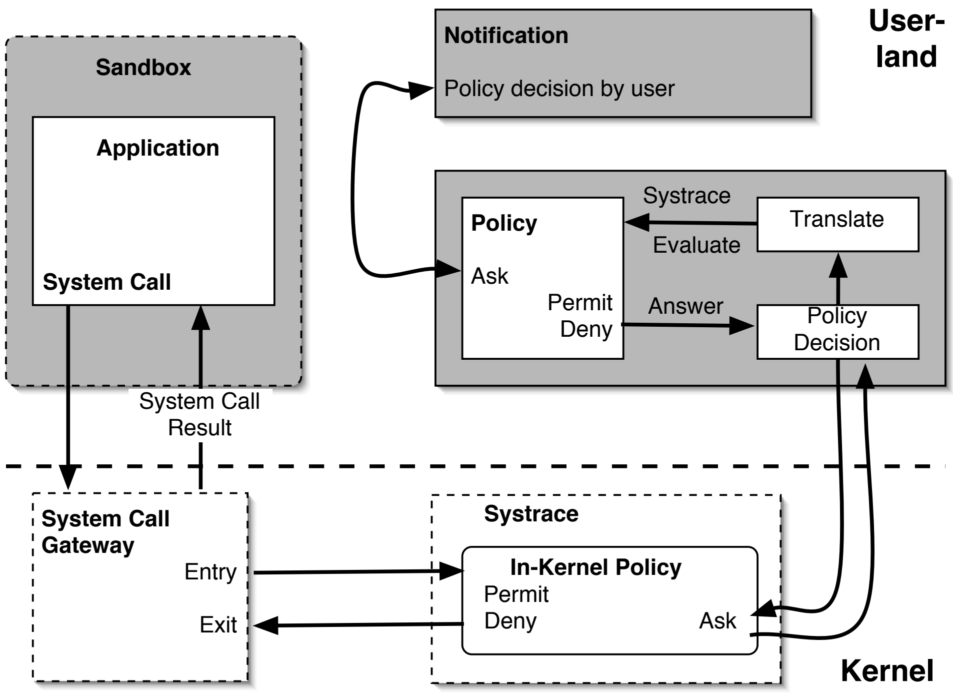


Figure 9.4: Sandboxing in systrace

As a consequence, policies have to defined that tell the Systrace facility which system call actions are permitted for a specific process. A systrace policy comprises a set of rules. Each rule controls a system call and its paramters, speciying whether or not this system call is allowed. A simple rule allowing the use of the “chdir()” and “stat()” system calls looks as follows:

```
chdir: permit
stat: permit
```

A rule containing the keyword “deny” would accordingly prevent these system calls to be executed. Rules can also control parameters of a system call:

```
fsread: filename eq "/tmp/gridjob" then permit
fsread: filename match "/etc/*" then deny[enoent]
```

Based on these rules, the program is allowed to read the file “/tmp/gridjob”, but trying to read files that match the wildcard expression “/etc/*” (i.e., system configuration files) will result in an `ENOENT` error code returned instead.

One additional requirement for the use of systrace for ad-hoc grid deployment is the ability to only activate its functionality for a specific shared object – the dynamically loaded native JNI-based code – in a process context. Systracing the complete Java VM environment including all loaded shared libraries would restrict the functionality of the already sandboxed Java code too far.

9.4.2 Systrace Implementation

An implementation of the systrace functionality can use one of several ways to insert control mechanisms. Looking at the path of a system call reveals several potential mechanisms. Applications initiate system calls by writing to specific registers and invoking software interrupts (in BSD on x86 systems, the "int" instruction). Applications usually do not execute system calls directly, they usually relay this task to the standard C library (libc), which handles setting up and initializing the calls. In fact, a large part of the C library simply consists of functions that provide a wrapper for system calls like *open(), read()* or *write()*. The path of a system call from the application down to the kernel is shown in fig. 9.5. A user mode application, `/bin/cp`, calls the `write()` library function, which selects the appropriate system call via the `eax` register.

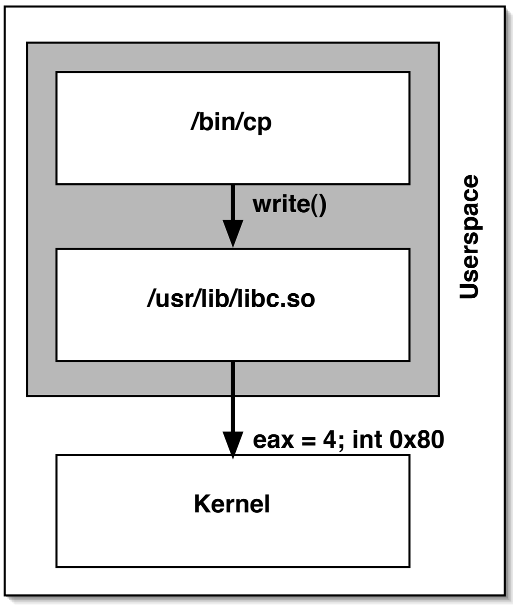


Figure 9.5: System call routing using libc

System calls can be intercepted and modified in each of these layers. Interceptiog system calls in the library layer would be trivial, as a specially restricted libc could be enforced for dynamic linking by setting the `LD_PRELOAD` environment variable. This mechanism, however, can be easily cicrumvented by an application programmer – the application simply has to invoke a system call directly instead of going through libc. In addition, this method does not work for statically linked programs.

Thus, the kernel layer is the natural place to intercept system calls. It is the only place where is can be assured that every system call is caught, no matter which way it was invoked. Every system call enters the kernel via the syscall gateway, which acts as a kind of interrupt handler for the software interrupts the system calls use.

Upon invocation, the gateway reads a register (`eax` on x86 processors) to determine the requested system call number, which is an index into the system call table that contains pointers to individual kernel functions. The gateway parses that number and then initiates the correct function which performs the task specified by the system call. In order to reject a system call, systrace must intercept it before it is executed, so it hooks into the syscall gateway.

A part of systrace's functionality is implemented in a user space program which evaluates the policy files. It communicates with the kernel using the `/dev/systrace` device special file by reading kernel messages from the device and invoking `ioctl` calls in order to send messages to the kernel. The user space program is not relevant for this discussion, since the aspect-oriented implementation described below is concerned with the crosscutting problems in kernel mode.

9.4.3 Crosscutting Properties

The code introduced by systrace consists of about 2900 lines of context diffs that contain modifications to four source code files:

- **systemcalls.c** – Implementation of central system call code (3 locations)

  The function affected here is the system call dispatcher. The systrace helper function that checks for permissions and valid parameters is crosscutting the dispatcher functionality;

- **kern_exec.c** – Functions related to loading and starting programs from a file system (4 locations)

  Here, the function implementing the "exec" system call is affected. Various pieces of code implementing object loading for several binary formats (ELF, a.out) as well as code providing special treatment for interpreter-based executables like shell scripts, are crosscut. The systrace code fragments check for valid command line parameters, permission of program execution and access to files passed by file handles from the parent process.

- **kern_exit.c** – Functions related to terminated and finishing processes (3 locations)

  In this file, the function that implements the "exit" system call is crosscut. A process may exit due to normal or abnormal (forced) termination. Here, systrace functionality that calls a helper function to check return values and clear up outstanding signals as well as functionality that releases ruleset information related to this process is crosscutting the exit functionality.

- **kern_fork.c** – Functions related to process creation (2 locations)

  Finally, in `kern_fork.c`, the "fork" system call itself is affected. Fork is used to create an identical copy of the calling process. The systrace functionality crosscutting fork checks for valid permissions to execute fork at all, for restrictions on the open files passed via file descriptors from the parent and for possible restrictions on return values from fork.

The systrace functionality requires changes in `sys_exec`, `sys_exit` and `sys_fork` functions implementing the particular system call. In the system call dispatcher itself, the permission to call a specific system call at all is controlled.

The interaction of systrace with the system call dispatcher is shown in listing 9.5. Execution of the system call itself is performed by the call to the corresponding entry in the system call table via a function pointer: `error = (*callp->sy_call)(p, args);`. If systrace is active, the support functions `systrace_enter` and `systrace_exit` are called before and after the system call execution itself:

```
        if ( (error == 0) && (ISSET(p->p_flag, P_SYSTRACE)) ) {
                error = systrace_enter(p, code, args, p->p_retval);

                if (error == 0)
                  error = (*callp->sy_call)(p, args);
        } else
          error = (*callp->sy_call)(p, args);

        /*
         * MP SAFE (we may or may not have the MP lock at this point)
         */
        have_mplock = userret(p, &frame, sticks, have_mplock);

        if (ISSET(p->p_flag, P_SYSTRACE)) {
                systrace_exit(p, code, args, p->p_retval, error);
        }
```

Listing 9.5: Systrace Functionality in `systemcalls.c`

In addition, three new files are introduced implementing the systrace core functionality:

```
kern_systrace.c
systrace.h
tree.h
```

These functions provide the in-kernel routines for systrace as well as the interfacing to the user-mode components.

9.4.4 AOP Implementation and Evaluation

The TOSKANA-based implementation of systrace reuses code from the newly introduced source code file, `kern_systrace.c`, to implement the aspect. The advice code consists of about 2200 lines of C code. Most of the advice code is manually refactored from the existing scattered and tangled original implementation of systrace. About 1700 of the 2200 lines consists of a large helper function that coordinates the communication with the user mode tool that parses the rulesets and the in-kernel ruleset evaluation itself.

The dynamic aspects provided by TOSKANA are useful for implementing systrace, since the functionality provided by systrace is inherently dependent on the process context. Consequently, the systrace aspect is activated and deactivated depending on the process that currently is in the running state. This way, processes that are not confined into a sandbox, which are probably the majority of processes running on a system, do not have to suffer a performance hit.

9.5 Autonomic Computing

The area of autonomic computing consists of a multitude of different approaches to handling the central problem to make computers less dependent on human maintenance and supervision, thereby creating systems that are more reliable, have less downtime and are more immune against malicious and accidental attacks on system integrity – or, short, to employ "self"-technologies like self-monitoring, self-configuration, self-optimization and self-healing. To implement autonomic functionality, information from many different hard- and software components is required, which in turn results in various crosscutting concerns to be solved. In this section, four examples implementing basic autonomic computing functionality, each providing support for one "self"-principle using dynamic aspects are presented.

The following examples are less complex and self-contained, thus they are shown using the original TOSKANA macro set that references the in-kernel support library described in chapter 7 instead of relying on the AspectC compiler to generate the macro descriptions.

9.5.1 Self-Configuration

In most current computer systems, plug-and-play functionality is essential, since many devices and bus systems are designed to add and remove devices like USB memory sticks, network adapters and input devices or Firewire optical drives on the fly.

While adding new components only adds information that does not directly affect running processes, the removal of a component from a running system may pose a problem if the device or a functionality implemented on top of the device (like a file system on a USB memory stick) is in use by some running process.

Depending on the type of device that is removed, the system may lose access to open files, network connections or a data source. Removing a device containing a file system may not lead to immediate error messages.

This example is a use case for self-configuration because the system adapts itself to a changed hardware configuration. Since the device may be removed on the fly without rebooting the system, dynamic adaptation is required. The cross-cutting functionality that is affected here is the call of the `VOP_OPEN` function in the NetBSD kernel, which occurs in 43 locations in architecture-specific, basic kernel, file system and device driver source files.

A code fragment that catches the removal of a storage device from the system and signals an error to kernel code (which is spread over various file systems) that is trying to open a `vnode` on the associated file system by returning an error code is shown in listing 9.6.

```
#include <aspects.h>

volatile int device_id = 0;
volatile int current_device;

void aspect_init(void) {
    AROUND(usb_storage_removed, remove_aspect);
    AROUND(VOP_OPEN, vop_aspect);
}

ASPECT remove_aspect(void) {
    device_id = PROCEED();
}

ASPECT vop_aspect(void) {

    /* Copy vnode from stack and convert it to */
    /* device number */
    current_device = DEVICE(LOCAL(1));

    /* If it is the one that was removed, return */
    /* error code */
    if (device_id == current_device)
        return(ENODEV);
    else
        PROCEED();
}
```

Listing 9.6: Implementation of self-configuration

9.5.2 Self-Healing

Main memory requirements in a running system are unpredictable, whereas the maximum amount of available virtual memory is limited by the amount of main memory plus the amount of swap space configured at system startup time. While it is possible for a system administrator to add swap space manually to a running system, autonomic behavior that adapts to current memory requirements, adding and removing swap files on the fly, is preferable.

Allocating new virtual memory in NetBSD is handled by calling `uvm_map`. If the system runs out of virtual memory space, an error code of `ENOMEM` is returned to the allocating function. To work around this situation, a call to `uvm_map` has to be retried around every invocation of the function in case of insufficient virtual memory. Calls to `uvm_map` are contained in the base kernel code, architecture-specific parts, compatibility functions and some higher-level virtual memory functions.

This example illustrates the self-healing principle since the system autonomically adds new swap space if required. Virtual memory requirements are dynamically changing with the process load. The cross-cutting functionality that is affected here is the call of the `uvm_map` function, which occurs in 334 locations in architecture-specific, basic kernel, compatibility layer and virtual memory system source code.

Dynamically increasing virtual memory space involves placing an advice around the execution of `uvm_map`, adding virtual memory space and retrying the operation in case of a failure. A code fragment illustrating this functionality is shown in listing 9.7.

```
#include <aspects.h>

int error;
```

```

void aspect_init(void) {
    AROUND(uvm_map, map_aspect);
}

ASPECT map_aspect(void) {
    error = PROCEED();

    if (error == ENOMEM) {
        /* no memory available */
        kernel_alloc_swapfile(SWAPFILE_SIZE);

        /* try again */
        error = PROCEED();
    }
    return error;
}
```

Listing 9.7: Implementation of self-healing

9.5.3 Self-Optimization

Information on available free blocks in a file system is maintained in a free block bitmap that indicates the status for each block in the filesystem. To calculate the number of available free blocks on a given filesystem, the operating system skims through the free block list and counts the number of bits indicating free blocks, creating an overhead of O(n) at readout while maintaining O(1) at insertion time.

When the system detects that the operation of reading out the free block count in a filesystem occurs more frequently than updating the free block bitmap, a readout overhead of O(1) can be achieved by dynamically switching the free block calculation so that the amount of free blocks is now calculated prior to every bitmap update instead of recalculating it on every call to a readout function.

A skeleton showing the basic implementation for this self-optimization functionality is shown in listing 9.8. For reasons of simplicity, only the switch to precalculating the number of free blocks is shown; the reverse action can be implemented in a similar manner.

This example illustrates self-optimization functionality because the calculation of free blocks is dynamically shifted to a location where it causes the least system overhead. The system adapts dynamically to changing conditions, since it checks if the update or free block check operation is occurring more frequently. The cross-cutting functionality here is the call of the `update_bitmap` and `read_free_blocks` functions, which occur in basic kernel, file system and device driver source code.

```
#include <aspects.h>

volatile int count_update = 0;
volatile int count_read = 0;
/* -1 indicates never called */
volatile int free_blocks = -1;

```

```
void aspect_init(void) {
    BEFORE(update_bitmap, update_aspect);
    AROUND(read_free_blocks, read_free_aspect);
}

ASPECT update_aspect(void) {
    count_update++;

    if (count_read > count_update) {
        free_blocks = calc_free_blocks();
    }
}

ASPECT read_free_aspect(void) {
    count_read++;
    if (count_update == 0 || free_blocks == -1) {
        PROCEED();
    } else
        return free_blocks;
}
```

Listing 9.8: Implementation of self-optimization

9.5.4 Self-Protection

Implementing enhanced security features in layers above the operating system kernel contains the danger that malicious users or code may be able to circumvent the security measures by using kernel-level functionality.

In NetBSD kernel code, the accessibility of a file in a filesystem is determined by calling the function VOP_ACCESS with a set of parameters that describe the vnode of the file to check, the type of access required, a description of the user credentials and a description of the process which is checking the credentials.

Calls to VOP_ACCESS can be found in all file system implementations, in the virtual file system layer itself, in the kernel core exec functions that handle execution of user mode programs, in kernel terminal device handling code, in the in-kernel NFS server code and in various modules that provide binary compatibility with other Unix variants in NetBSD. In the entire kernel code, 78 instances of calls to VOP_ACCESS can be found.

If additional security functionality is introduced to the system, e.g. an additional check of a smart-card reader to permit access to a specific file or set of files, VOP_ACCESS does not provide sufficient semantics to implement the required changes directly. Instead, before each of the calls to VOP_ACCESS that are scattered around the kernel source code, an additional check would have to be introduced to check valid authentication.

This examples illustrates the self-protection principle, since the system is changed to permit access to certain vnodes only when an extra authentication has succeeded. Example code illustrating this functionality is shown in listing 9.9.

```
#include <aspects.h>

```

```
volatile int is_authenticated = 0;

void aspect_init(void) {
    AROUND(authenticate_start, authstart_aspect);
    BEFORE(authenticate_end, authend_aspect);
    BEFORE(VOP_ACCESS, access_aspect);
}

ASPECT authstart_aspect(void) {
    if (PROCEED() == 0)
        /* authentication OK */
        is_authenticated = 1;
    else
        /* some error occured */
        is_authenticated = 0;
}

ASPECT authend_aspect(void) {
    /* no longer authenticated */
    is_authenticated = 0;
}

ASPECT access_aspect(void) {
    if (! is_authenticated)
        return EPERM;
}
```

Listing 9.9: Implementation of self-protection

9.6 Dynamic Aspects in a Process Context

The extension of join point semantics to include support for in-kernel contexts like a process context has implications on activation and deactivation of dynamic aspects.

In order to provide aspects local to a process context, the activation and deactivation of the aspect has to occur in the scheduler. Since process context dependency was required for some of the other application examples, a first implementation of context dependency was developed as an extension to TOSKANA.

The deployment implementation is located in `kern/kern_sync.c` and extends the function `awaken` with the code in lines 16. . . 27 in listing 9.10.

```
 1  __inline void
 2  awaken(struct lwp *l)
 3  {
 4
 5          SCHED_ASSERT_LOCKED();
 6
 7          if (l->l_proc->p_sa)
 8                  sa_awaken(l);
 9
10          if (l->l_slptime > 1)
11                  updatepri(l);
```

```
        l->l_slptime = 0;
        l->l_stat = LSRUN;
        l->l_proc->p_nrlwps++;

        /*
         * Activate dynamic aspects for current process
         */
        if (l->l_proc->numaspects > 0) {
           toskana_aspect *current = *(l->l_proc->aspects);
           int counter;

           for (counter = 0; counter < l->l_proc->numaspects; counter++) {
               toskana_deploy(aspect);
               aspect++;
           }
        }
        /*
         * Since curpriority is a user priority, p->p_priority
         * is always better than curpriority on the last CPU on
         * which it ran.
         */
        if (l->l_flag & L_INMEM) {
                setrunqueue(l);
                                KASSERT(l->l_cpu != NULL);
                need_resched(l->l_cpu);
        } else
                sched_wakeup(&proc0);
}
```

Listing 9.10: Activating Dynamic Aspects in the Scheduler

This implementation currently is unoptimized as to efficient activation and deactivation of aspects. The overhead involved currently scales linear with the number of aspects to be deactivated from the previous process context and to be activated for the new process context. Here, an optimization using patterns of instructions to modify – similar to a just-in-time compiled dynamic weaver – or a kernel modification that activates and deactivates in-kernel aspects using duplicate pages with and without the woven aspect would greatly enhance performance.

9.7 Application Ideas

The following sections describe some ideas for applications that have been analyzed for their feasibility, but not implemented in the context of this thesis. However, the examples serve to give an impression of the wide range of applications for in-kernel aspects.

9.7.1 Variant Symlinks

A variant symlink is a symlink that has a variable name embedded in it. This allows some interesting operations to be performed that are not possible with a standard symlink. Variables embedded in the symlinks can include user and or environment specific information among other things.

The first implementation of variant symlinks appeared in the mid 1980s in Apollo's DomainOS, but has never found widespread acceptance. Here is an example for the use of variant symlinks:

```
 1  $ ln -s 'a${var}b' test
 2
 3  $ cat > axxb << EOF
 4  Hi!
 5  EOF
 6
 7  $ varsym var=xx
 8  $ cat test
 9  Hi!
10  $
```

The command in line 1 creates a symlink named `test`, which contains a variant symlink target, `a${var}b`. The `var` part of the target is replaced by whatever is configured in the kernel using the `varsym` command in line 7. In between, the commands in line 3. . . 5 create a file `axxb`. The `cat` command in line 8, then, accesses the symlink. The kernel deferecenes the link target and replaces the variable part with the current definition of `var` to produce the final file name `axxb`, which is then dereferenced using the `namei` in-kernel function.

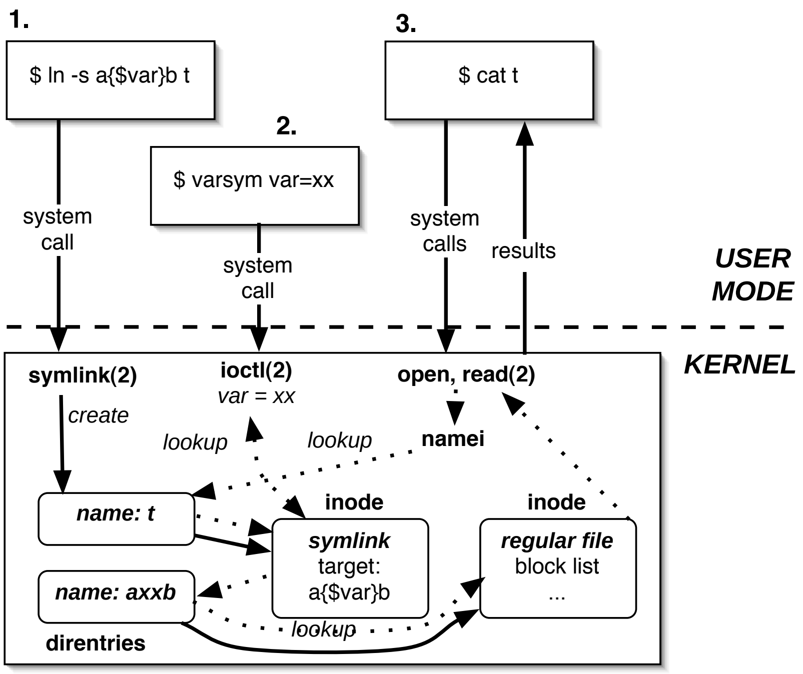


Figure 9.6: Kernel/user mode interactions for variant symlinks

The interactions on user and kernel level are shown in fig. 9.6. Step 1 creates the variant symlink using the regular symlink system call. This creates an inode containing the symlink target `a{$var}b` and a corresponding directory entry `t`. Next, the user mode program varsym sets the in-kernel resolution for the symbol var to the value xx in the context of the current process group. Then, any access to the file t from the same process group results in a lookup of the directory entry for t, which results in the inode being returned. Since the inode is a symlink, the kernel performs an internal symlink resolution using the namei function. namei recognizes that the symlink is variant and consults the in-kernel process group state to check if the variant part can be resolved. Since there is a definition var = xx set, the symlink resolves to axxb, which is then looked up and returned using the regular namei mechanisms in the kernel.

Implementing this functionality crosscuts in several areas – first, all file systems implementing symlinks have to support this functionality in their corresponding vfs namei implementation. The bookkeeping for the variant symlink resolving has to occur in a special device implementing the ioctl, and the creation of a symlink has to recognize that the symlink is variant and set a special flag inside the inode's create function. Finally, the functionality of variant symlinks is relative to a process group (i.e., one user setting a variant symlink resolution in a shell context may not disturb other users' settings), which has to be taken into account when defining the pointcuts.

9.7.2 Audio Volume Regulation

The mixing of several sources is necessary in desktop computer systems, since several applications should be able to simultaneously generate audible signals. One example is running a MP3 player application running on a notebook, while permitting system alert sounds (e.g., for a low battery condition) to be heard by the user. A possible scenario is shown in fig. 9.7.

Each application provides a separate control for its audio volume. The in-kernel audio driver component takes all audio streams and mixes them together to produce a common output signal. Optionally, a multiplexing of the output to various devices is possible – e.g., system beeps only sound on the internal speaker, whereas MP3 and DVD audio are routed to an external amplifier.

Audio volume regulation over several sources is a concern crosscutting process and device driver contexts. Each context producing audio output has to be able to dynamically set its relative output volume, the central mixer has to take all these volume settings into account when calculating the resulting output signal and, in addition, has to re-route some of the signals to dedicated sources.

A first analysis of the NetBSD kernel sources was performed. Parts of the kernel source code affected by the volume regulation are the general audio mixing framework dev/audio.c, the console audio notifier that handles the console alert function (bell) dev/audiobell.c, the OSS compatibility libraries for emulating other Unix systems compat/ossaudio.c, the USB audio driver dev/usb/uaudio.c and various ISA/PCI sound card drivers, e.g. dev/isa/sb_isa.c.

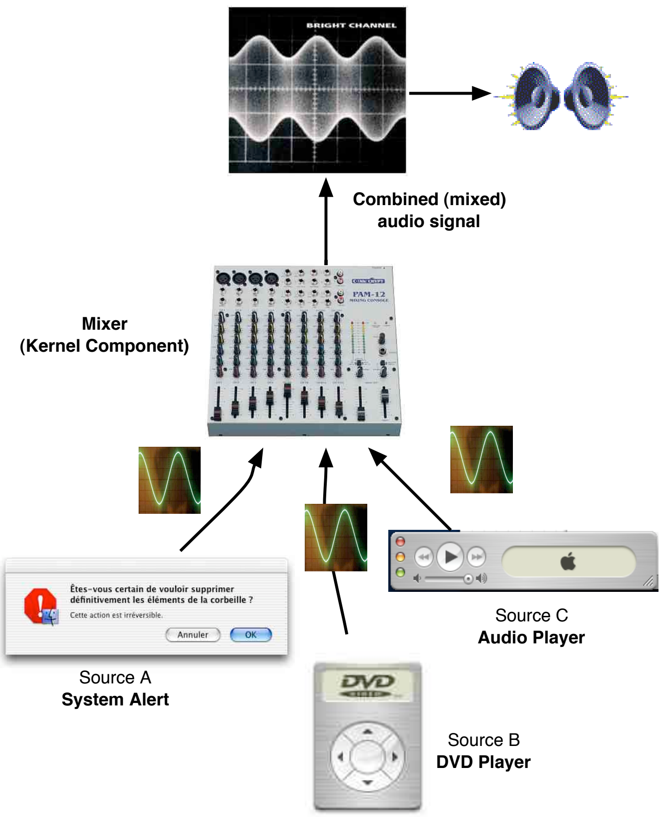

BRIGHT CHANNEL
Combined (mixed) audio signal
Mixer (Kernel Component)
PAM-12
Êtes-vous certain de vouloir supprimer définitivement les éléments de la corbeille ?
Cette action est irréversible.
Annuler
OK
Source A System Alert
Source C Audio Player
Source B DVD Player
DVD

Figure 9.7: Audio mixing in the kernel

9.7.3 Energy Awareness

An increasing number of devices today is powered by a battery – mobile phones, portable MP3 players, notebooks computers etc. For these devices, it is extremely important to use energy as economically as possible. Conservation methods involve scaling the CPU frequency, turning off unused peripheral devices, adapting wireless transmission power, controlling display brightness and saving the system and application state to non-volatile memory when not using the device for extended periods.

CPU scaling alone is already a crosscutting concern. Many functions inside the kernel, like timeout calculation, scheduling, and device control, require a reliable time base. Some timing functions are realized using exactly calibrated busy waiting loops that unexpectedly change the duration their execution takes when the CPU frequency is changed in a running system.

A first analysis of the NetBSD kernel source has shown that implementing CPU frequency scaling affects various device drivers (e.g. for serial lines, IDE disk drives, various network drivers) as well as general CPU timing routines. Among the files affected are ISA clock timer calibration `arch/i386/isa/clock.c`, hardware timer calibration in the ACPI subsystem `dev/acpi/acpica/Subsystem/hwtimer.c`, the kernel timer routines `kern/kern_time.c`, various network and mass storage device drivers (e.g., `dev/ic/tulip.c` and `dev/ieee1394/`

`fwohci.c`, PCI timing routines `arch/pi6/pci/ichlpcib.c`, and the altq network load leveler `altq/altq_subr.c`

9.7.4 Process Checkpointing and Migration

Process checkpointing and migration is an essential technology for distributed systems. Checkpointing allows the complete state of a process running on one system to be captured at a given instant. Once this state is captured, the complete virtual memory image of the process can be transmitted to a different computer along with the status or even "frozen" to non-volatile storage for later use.

Applications for this technology include fault tolerance, workstation clusters where compute-intensive background jobs adapt to the current interactive load of a system, and pervasive computing, in which an application and its data is made avaiable worldwide to a roaming user.

While there were some experiments with process migration, often based on microkernel systems, like Dejan Milojicic's Mach-based task migration system [95], process migration on Sprite [46] and the Winner system running on Digital Unix [6], checkpointing never found widespread use in mainstream operating system. One of the reasons for this may be the complex implementation due to the amount of crosscutting concerns involved.

The state that has to be captured for successful process migration involves almost all areas of the kernel. The state includes the code for the program, the program's static and dynamic data, the procedure call stack, contents of general purpose CPU registers, the current program counter and the operating system resources in use.

Fig. 9.8 exemplifies some of the in-kernel state of a process as stored in the process control block entries. This information is read and written in different parts of the kernel. In addition to this information that is available directly from the in-kernel process state, the per-process state for network connections, open files, user and group permission context, scheduling priority, process group memberships (and potentially more) has to be recorded, transferred to the target system and reinstated there. The gathering of information on the sending machine and replay on the target are extremely crosscutting, since the in-kernel checkpointing functionality has to record information from all areas cited above.

A process may optionally contain even more state information, e.g. accounting information for network and processor usage, location of on-screen windows, and I/O buffer contents, which are not directly kept in the process control block. Finding, gathering and restoring this information requires additional aspects. An extensive analysis on all information making up the complete process state, however, has not been performed.

In addition to the crosscutting in code described above, a special set of problems, which have implications on an aspect-oriented implementation of process migration, can occur when deploying the checkpointed process on the target machine:

Often, the migrated process keeps its own process ID in a variable. On the target machine, however, this PID may already be used, so a kernel function has to dynamically provide a mapping (proxy) between the real new PID and the processes' kept value. Here, aspects

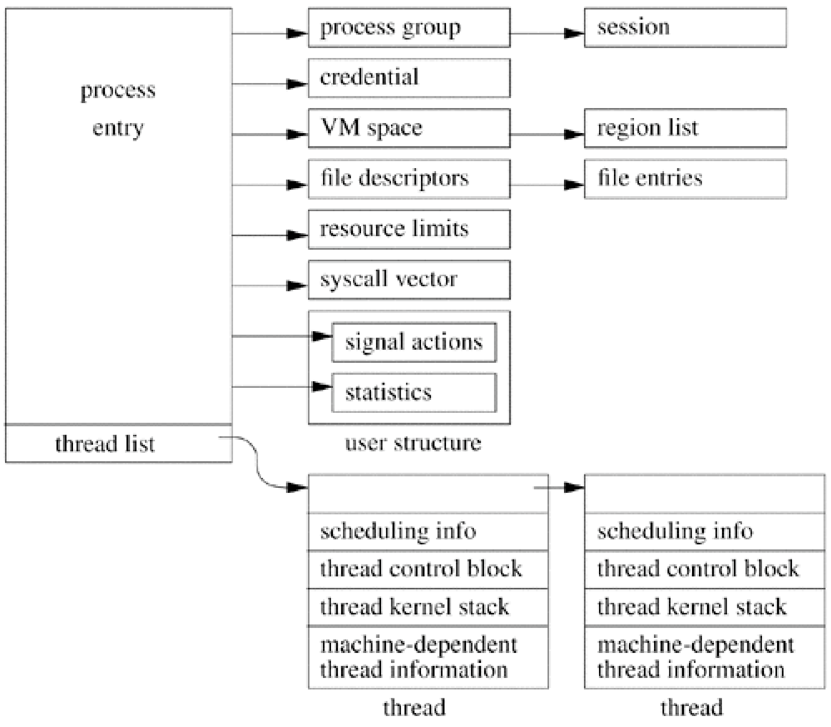


Figure 9.8: NetBSD process state

on the originating and destination machine have to coordinate themselves to configure the proxy instance.

When network connections are used, the machine the process originally ran on must take care of either re-routing the packets destined for the migrated process to the new target or some sort of mobile IP management must be used. During the migration phase of the process, network timeouts may occur which have to be handled appropriately by initiating retransmissions. Temporal dependencies in aspects can help handling timeout situations, e.g. by retriggering the sending of network packets if an aspect detects that a deadline was missed.

File system access must be transparently re-routed, e.g. by using a networked file system. However, shared file accesses and locking between nodes has to be handled appropriately, e.g., a mechanism for transferring file locks over the network has to be implemented. The same is valid for other I/O streams like console and audio, which have to be provided by appropriate proxy objects. Here, aspects on the source and destination system have to communicate to ensure correct mapping of permissions on both systems.

Process migration on top of a mainstream kernel may prove to be one of the sought-after “killer applications” for dynamic kernel-mode AOSD. The problems involved and the related implementation of process migration are, however, complex enough to justify their own thesis.

9.8 Summary

The number of applications for AOP in kernel code is as big as the number of crosscutting concerns. This chapter presented a selection of interesting, novel crosscutting problems from various areas of the kernel, highlighting important sources for crosscutting concerns. AOP-based solutions using static as well as dynamic AOP technology were exemplified and evaluated. In addition, even more ideas for aspect-oriented implementations of common in-kernel problems were presented.

10. Related Work

"There's no sense being exact about something
if you don't even know what you're talking about."

— John von Neumann

10.1 Introduction

Where can related work be located best in this thesis? Since the topic covers a crossover of two large topics of computer science – operating system research and aspect-oriented technology – references would best be located scattered throughout the text, tangled with the related novel approaches presented.

That's a crosscutting concern!

Since it is appropriate for a thesis covering an aspect-oriented topic, related work is factored out of the single chapters into this chapter, and join point shadows are marked throughout the text to provide the context that section is to be woven into. Unfortunately, there seems to be no weaver for LaTeX documents so far, so the task of (statically) weaving the related work aspects into the text flow is left as a manual task to the reader.

Research related to AOP in kernel can be found in many areas. Research in AOP in an operating systems context and in dynamic aspect-oriented approaches as well as procedural language extensions supporting aspects form one part of related work. Directly connected to the dynamic aspect deployment methods used in TOSKANA are program instrumentation technologies, whereas technologies implementing virtualization technology are relevant to the TOSKANA-VM approach. This is also related to the discussion on intercepting instruction execution on various levels from hardware to high-level virtual machines, like the Transmeta Crusoe code morphing approach.

10.2 Instrumentation Tools

Kernel instrumentation is a technology that aims to perform run-time system analysis, including performance measurement, debugging, code coverage, run-time installation of patches, and run-time optimizations. Static kernel instrumentation incurs a high overhead, since instrumentation points are statically compiled into the existing code, using up compute time for checks if a given instrumentation point is actually activated. Dynamic kernel instrumentation, however, can implement instrumentation points at run-time into an existing kernel binary, thus reducing the overhead and providing a greater level of flexibility.

10.2.1 KernInst

KernInst [129] is a tool that enables a user to dynamically instrument an already-running unmodified Solaris operating system kernel in a fine-grained manner. It has been successfully applied to kernel performance measurement and run-time optimization.

The basic principles used by KernInst to dynamically insert instrumentation points into running native code are similar to the approaches TOSKANA uses, and KernInst can also be seen as implementing some basic autonomic computing functionality – kernel tuning is explicitly given as one application area.

However, KernInst is not suited as a basis for dynamic aspect deployment. The space available for code patches and associated data is very limited in KernInst, so the deployment of a large number of aspects is not feasible.

10.2.2 Sun DTrace

A recent development extending the ideas of KernInst is DTrace from Sun Microsystems [23]. Like KernInst, DTrace is able to dynamically insert instrumentation code into a running Solaris OS kernel.

Unlike other solutions for dynamic instrumentation that execute native instrumentation code, DTrace implements a simple virtual machine in kernel space that interprets byte code generated by a compiler for the "D" language, which is an extension of C specifically developed for writing instrumentation code.

D was especially designed to make instrumenting the Solaris kernel safe in all circumstances. To avoid the danger that a piece of instrumentation code creates loops that hang-up kernel operation, only forward branches are permitted by the virtual machine. As a result, the functionality of D programs is relatively restricted. While this in principle provides a lot of security when dynamically inserting code into random spots in the kernel, the execution model provided by DTrace is too restricted to implement general advice code. In addition, D does not provide an instrumentation similar to an "around" advice. Overall, DTrace contains too many tradeoffs to be used in a more general implementation of dynamic AOP like TOSKANA. An open-source implementation of DTrace in the forthcoming OpenSolaris 10 will nevertheless be interesting to analyze.

10.2.3 GILK

GILK [104] is a dynamic instrumentation toolkit for the Linux kernel. It allows a standard Linux kernel to be modified during execution using instruments implemented as kernel modules. The GILK authors face many problems similar to the ones that occured during the design and development of TOSKANA, e.g. the variable-length instructions of the x86 architecture when replacing instructions in binary code. While the basic technological approaches of GILK are quite similar to the technologies TOSKANA uses, GILK is restricted to developing instrumentation code for Linux, whereas TOSKANA can be applied to cover a broader range of in-kernel applications.

10.3 Extensible Operating Systems

An extensible operating system lets an application safely tailor the operating system's behavior to the needs of the application. How to best build a system that explicitly allows applications to insert code into the kernel has been an active research topic for many years, with several different methodologies explored.

10.3.1 SUPERZAP

SUPERZAP (or IMASPZAP) (http://www.cs.niu.edu/csci/567/ho/ho4.shtml) is the IBM supplied utility for applying patches to object modules. SUPERZAP allows to inspect and modify instructions in load modules (without leaving any audit trail). With some positive thinking, SUPERZAP may be viewed as a tool for extending operating systems by machine code patching techniques. However, even though some of the basic technologial approaches SUPERZAP uses are similar to the approaches TOSKANA uses, its main application is supplying patches to a live operating system. While this is also possible using TOSKANA, it is rather a side-effect of the code manipulation technology employed and not a central feature.

10.3.2 SPIN

SPIN [13] is an operating system that allows applications to dynamically insert low-level system services, so called spindles, into the kernel, such that they can receive notification of, or take action on, certain kernel events. Spindles, as well as SPIN itself, are written in Modula-3, and the compiler is the only user mode process that is allowed to insert spindles at runtime into the kernel (by performing compilation at runtime). This has the drawback that the time to compile and optimize a spindle has to be spent for each spindle insertion; dynamically linking a spindle into the running kernel also takes time. Furthermore, there may be semantic difficulties when multiple handlers are installed on a single event. While SPIN is an interesting approach that could possibly be used as a basic technology for a TOSKANA-like aspect system, it never gained widespread acceptance. One advantage of SPIN is that – due to the use of Modula-3 as its implementation language – many of the problems TOSKANA has to handle in C code are non-existent, so SPIN can actually provide a more secure approach to extending operating systems. However, to our knowledge, manipulation of already existing pieces of functionality in SPIN is quite hard, whereas it is one of the common usage scenarios for TOSKANA.

10.3.3 VINO

The Vino operating system [120] allows applications to dynamically insert code that replaces the implementation of a method on any object. Alternatively, code can be installed to run when a specified kernel event (such as a network packet arrival) occurs. The granularity at which the system can be extended is dependent upon the objects and events exported by the system. The complexity and performance overheads associated with Vino prevent it from being as efficient as SPIN, for example. Also, like SPIN, Vino does not include any features for dynamic system reconfiguration.

10.3.4 KEA

The Kea operating system [138] provides means through which applications can dynamically reconfigure, replace or extend existing services in an incremental and safe manner. Kea incorporates several features, so called portals and portal remappings, whereby a system can be easily reconfigured. Kea provides a mechanism (called IDC) which has all the semantics of a procedure call, but lets a calling thread continue execution in a different domain. Portals are the Kea abstraction used to represent an IDC entry point. They provide a handle into both a domain and an entry point within that domain, and once created they can be freely passed around between domains. The most important property of portals, and the one that makes reconfiguration possible, is that they can be dynamically remapped to a different domain/entry-point at runtime. Compared to TOSKANA, KEA only offers an incremental model to extend kernel functionality. This requires reimplementing complete kernel subsystems whereas using TOSKANA, relatively small changes, e.g. to improve the performance of certain code paths, can be implemented directly into an existing subsystem structure.

10.3.5 SLIC

SLIC [61] allows to insert trusted extension code into existing operating systems with minor or no modifications to operating system source code. Conceptually, SLIC dynamically “hijacks” various kernel interfaces (such as the system call, signal, or virtual memory paging interfaces) and transparently reroutes events which cross that interface to extensions located either in the kernel (for performance) or at the user-level (for ease of development). Extensions both use and implement the intercepted kernel interface, enabling new functionality to be added to the system while the underlying kernel and existing application binaries remain oblivious to those extensions. SLIC dynamically interposes extensions on kernel interfaces by modifying jump tables or by binary patching kernel routines. While the basic approaches of SLIC are comparable to the splicing approach used by TOSKANA, the hijacking can only occur to kernel interfaces, not to low-level functions inside the kernel.

10.4 AOP Compiler Tools

10.4.1 AspectC

AspectC was proposed as an extension of the ANSI C language by Coady [33, 31], but no implementation is available so far. In addition, the language has no formal definition so far, but it was rather described using example code, which, unfortunately, is inconsistent between publications on AspectC.

AspectC, however, provides the basic idea for using aspect-orientation in kernel mode. The lack of a readily available AspectC compiler resulted in the development of the AspectC prototype, described in chapter 5.

10.4.2 AspectC++

AspectC++ [123] is an extension of the C++ language that supports static join points as well as dynamic join points modelled after the AspectJ [65] approach. While static join points are named entities in the static program structure, dynamic join points are events that happen during the program execution.

The AspectC++ infrastructure considers the following kinds of C++ program entities as static join points: classes, structs, and unions, namespaces as well as all kinds of functions (member, non-member, operator, conversion, etc.).

Static join points are described by match expressions. For example, "% ...::foo(...)" is a match expression that describes all functions called foo (in any scope, with any argument list, and any result type). More information on match expressions is given below. Note that not all of these static join point types are currently supported as a target of advice.

The following kinds of events that might happen during the execution of a program are considered as dynamic join points: function calls, function execution, object construction, and object destruction.

AspectC++ is used in research and commercial embedded operating system projects like the PURE [124, 59] embedded realtime kernels. In the context of TOSKANA and TOSKANA-VM, however, AspectC++ proved not to be usable due to the compiler generating C++ classes from the AspectC++ source code, which requires in-kernel runtime support. Much inspiration on the modelling of join points and extending the AspectC compiler described in chapter 5 come from AspectC++, however.

10.4.3 xtc

xtc [64] is a program transformation toolkit similar to the CIL toolkit used by the AspectC compiler. Currently, xtc is used by the Princeton PlanetLab project to manage large-scale modifications to the Linux kernel source code [17].

Xtc could be used as another alternative to the CIL toolkit used in the TOSKANA AspectC compiler implementation. However, the rewriting methods offered by CIL fit the requirements of AspectC weaving more readily.

10.4.4 XWeaver

XWeaver [113] is the name of an aspect weaver for C/C++. The aspects to be processed by XWeaver are written in the AspectX language. The development of XWeaver and the definition of the AspectX language began in August 2003 at the Automatic Control Laboratory of ETH Zurich.

XWeaver is an extensible, customizable and minimally intrusive aspect weaver. It is extensible in the sense that the aspect weaving process is controlled by a set of rules which are independent of each other and are each implemented in a dedicated module. The weaver can be extended by adding new modules that cover new rules to handle additional types of aspects. Similarly, the weaver is customizable because existing rules can be tailored to the needs of a particular project by updating the corresponding rule module. XWeaver is minimally intrusive in the sense that it gives a high degree of control over the structure of the modified code and it generates code which differs as little as possible from the base code. For this reason, XWeaver is aimed at applications that must undergo a qualification process such as mission-critical applications.

XWeaver, however, uses aspect specifications in a special XML format that break the continuity of the C code the aspects are to be woven into. In order to gain acceptance for kernel development, a solution extending C in a more "natural" way has been prefered in this thesis.

10.5 AOP in Kernel Space

Since aspect-orientation is well suited to work on large software systems, research on using AOP in kernel space has been the topic of some projects. The existing developments, however, concentrate on providing static AOP functionality mostly using AspectC or AspectC++ as a basis.

10.5.1 a-kernel

Research on aspect-orientation in operating system kernels was initiated by [33], where problems that crosscut the traditional layered operating system structure were identified using FreeBSD as an example.

Based on AspectC, an aspect-oriented extension of C, the a-kernel project tries to determine whether aspect-oriented programming is a suitable method to improve OS modularity and thereby reduce complexity and fragility that is associated with the implementation of an operating system.

In [31], [34], [32], and [30], various cross-cutting concerns are implemented as static aspects using AspectC in the FreeBSD kernel. In addition, an analysis of code evolution implementing cross-cutting concerns between different versions of FreeBSD is undertaken and the evolution is remodelled as static aspects.

Further development in this project resulted in the concept for RADAR [125], which is a proposal for a low-level infrastructure for system analysis using dynamic aspects in OS code. This is an interesting development similar to TOSKANA, although the application area is targeted to analysis. Currently, no experience with implementing the system seems to exist.

10.5.2 CiAO

In [124] and [59], the design and implementation of an aspect-oriented family of operating systems is described which provide, based on aspects and static configuration, a full encapsulation of architectural properties of an OS. CiAO is targeted at the area of embedded systems, ranging from deeply embedded devices up to Unix-like operating systems, requiring a so far unattained level of adaptability and configurability. The language used for developing operating systems of the CiAO family is AspectC++, which provides AspectJ-like functionality for C++.

CiAO, however, is targeted to embedded systems. A general use in operating systems may be beyond the scope of that system. In addition, the use of AspectC++ is only possible in systems newly developed in an object-oriented style. The legacy systems this thesis is concerned with are, however, mostly written in procedural C, so the CiAO approaches are not applicable here.

10.6 Dynamic AOP

There are several approaches for implementing dynamic aspect-orientation. Of interest for this thesis are solutions that work on C code as well as solutions that operate in the execution layer.

10.6.1 Arachne

Arachne [119] is a dynamic weaver for C programs. It uses the μDyner AOP infrastructure for writing and dynamically deploying aspects into running user mode C applications without disturbing their service. The implementation of join points, however, requires source instrumentation of the program to reduce the cost of dynamic weaving. μDiner provides a special *hookable* source-level annotation with which the developer of the base program annotates points in the program at which later on dynamic adaptation is permitted. Here, only functions and global variables can be declared to be hookable.

The aspect code is written using a special extension of C that provides the syntax for specification of join points and advice types. Aspects are compiled into dynamically loadable libraries that can be woven and unwoven using the Arachne *weave* and *deweave* tools.

While Arachne is an interesting approach comparable in its functionality to TOSKANA, its use is restricted to weaving aspects in user mode programs. Since these can be easily controlled by a running kernel without changing state, Arachne would require extensive modifications to be used in kernel code.

10.6.2 Steamloom

The Steamloom virtual machine [15] is an extension to an existing Java virtual machine. It follows the approach to tightly integrate support for aspect-oriented programming with the virtual machine implementation itself, instead of delivering a set of Java classes providing such support. Steamloom comprises changes to the underlying VM's object model and just-in-time compiler architecture. Its approach to dynamic weaving is to dynamically modify method bytecodes and to schedule those methods for JIT recompilation afterwards.

Steamloom was built to evaluate the benefits that can be gained by implementing AOP support in the execution layer instead of at application level. Various measurements [69] have shown that both flexibility and performance significantly benefit from such an integrative approach. Using Steamloom for dynamic aspects in kernel code, however, would require a kernel written in Java. Some approaches to create a Java-based operating system exist [96], but none of these has shown significant success.

10.7 Low-Level Virtual Machines

Since modifications of the execution layer are rather problematic to implement – modifying microcode is difficult and source code to Transmeta's code morphing software is not freely available – another solution is the introduction of a very thin hardware abstraction layer on top of the CPU instruction set. These virtual machines are simpler and more similar to a normal CPU than JVM or the .NET Intermediate Language engine. Two such projects are the Low Level Virtual Machine LLVM [86] and the Virtual Virtual Machine [108].

10.7.1 LLVM

LLVM [86] is an infrastructure consisting of a RISC-like virtual instruction set, a compilation strategy designed to enable effective program optimization during the lifetime of a program, a compiler infrastructure that provides C and C++ compilers based on the GNU compiler collection and a just-in-time compiler for x86 and Sparc machines. LLVM does not imply things that one would expect from a high-level virtual machine. It does not require garbage collection or run-time code generation. Optional LLVM components can be used to build high-level virtual machines and other systems that need these services.

LLVM forms the basis of the TOSKANA-VM toolkit and is described in more detail in chapter 8.

10.7.2 VVM

Instead of designing and implementing a new virtual machine for each application domain, the goal of VVM [108] is to virtualize the virtual machine itself. VVM supports so-called “VMlets” that contain a specification of a virtual machine implemented using VVM.

No large-scale operating system has been developed to run on top of VVM, however, so the feasibility of this approach still has to be determined. VVM is, however, interesting in the context of domain-specific languages used to implement kernel components, since an adaptive virtual machine would be able to provide tailored VMs for each domain-specific language that are compatible on a common level.

10.7.3 z/VM

Another low level virtual machine that is explicitly used to virtualize a single physical machine into distinct partitions each running its own operating system instance is IBM’s z/VM[99], used on z-Series[1] mainframe systems. z/VM supports a large number of operating systems running in parallel on top of the virtual machine[2] and would be an ideal target for implementing dynamic aspect support in the execution layer. The source code for z/VM, however, is not available from IBM for further evaluation.

10.7.4 rHype

The Research Hypervisor Project [74] creates a hypervisor environment that is suitable for client operating systems that have had minor changes applied to them in order to run on the small number of machine abstractions the hypervisor presents. This is different than hypervisors that emulate the hardware completely so that no modifications to the client operating system are necessary. Research hypervisor project uses what is popularly known as ”para-virtualization” [7].

The main objective of rHype is to create a coding base by which the hypervisor that is designed is easily portable to other Instruction Set Architectures (ISA) as well be customized to target a specific implementation of an ISA and therefore keep the resulting code and binary as small as possible.

It provides a from-scratch implementation and design decisions that allow the core trusted code base to be tiny, making it appropriate for secure design. The memory management architecture is designed to enable large pages, to be cache efficient on common operations and friendly to future self-virtualizing I/O devices.

RHype can be seen as a reduced and open source implementation of the proven LPAR environment used in IBM’s commercial hypervisors like z/VM. Its use in dynamic AOP systems, however, is difficult, since the paravirtualization used in rHype does not permit the interception of arbitrary instructions – only privileged instructions are currently interceptable

[1]formerly S/390 series
[2]A test has shown that 40,000 parallel Linux instances are possible

using CPU hardware support. Thus, the current form of rHype is not suitable for use with TOSKANA.

10.7.5 Xen

Xen [7] is an open source virtual machine monitor, developed by the University of Cambridge. It is intended to run up to 100 full featured OSs on a single computer. Operating systems must be explicitly modified ("ported") to run on Xen (although compatibility is maintained for user applications). This enables Xen to achieve high performance virtualization without special hardware support.

Xen uses paravirtualization to achieve high performance (i. e. low performance penalties, typically around 2%, with worst-case scenarios at 8% performance penalty; this is in stark contrast to perfect emulation solutions which typically entail performance penalties of 20%), even on its host architecture (x86) which is notoriously uncooperative with traditional virtualization techniques. Unlike traditional virtual machine monitors, which provide a software-based execution environment exactly like the simulated hardware, Xen requires the porting of guest operating systems to the Xen API. Currently, this porting has happened for NetBSD, GNU/Linux, FreeBSD and Plan 9.

Intel has contributed modifications to Xen support their Vanderpool architecture extensions. This technology will enable unmodified guest operating systems to run within Xen virtual machines, if the host system supports the Vanderpool or the Pacifica extensions (Intel's, and AMD's, respectively, extensions to natively support virtualization).

Like rHype, Xen uses paravirtualization, so that the same restrictions for use in TOSKANA apply here.

10.7.6 Denali

The Denali isolation kernel [142] is an operating system architecture that safely multiplexes a large number of untrusted Internet services on shared hardware. Denali's goal is to allow new Internet services to be "pushed" into third party infrastructure, relieving Internet service authors from the burden of acquiring and maintaining physical infrastructure. The isolation kernel exposes a virtual machine abstraction, but unlike conventional virtual machine monitors, Denali does not attempt to emulate the underlying physical architecture precisely, and instead modifies the virtual architecture to gain scale, performance, and simplicity of implementation. The overhead of virtualization in Denali is small, it has shown to successfully scale to more than 10,000 virtual machines on commodity hardware.

Denali could prove to be an interesting alternative to TOSKANA for Grid application confinement. For general dynamic AOP applications, however, Denali provides insufficient methods for interfering with the kernel operation itself, since the isolation functionality is mostly restricted to user mode applications.

10.8 Summary

Dynamic AOP in kernel mode has some forerunners – especially static AOP in kernel mode and dynamic instrumentation technologies for operating systems. The dynamic AOP technologies presented in this thesis, however, use a novel combination of code manipulation based on aspect semantics that has not been realized in kernel mode before. The special requirements of kernel programming, like asynchronous events and changing process contexts, make dynamic AOP in kernel mode significantly harder compared to the various user-mode dynamic AOP approaches.

11. Conclusions

"As for the future, your task is not to foresee it, but to enable it."
— Antoine de Saint-Exupery, *The Wisdom of the Sands*

11.1 Achievements

While aspect-oriented software development is still a relatively new technology, it already has found broad acceptance in designing and maintaining application code. A widespread adoption in system level code development has not yet taken place. While first approaches show up in experimental or special-function operating systems, the "mainstream" kernel development procedures stayed quite conservative.

This thesis started out to explore the possibilities and limitations of AOP for kernel code, inspired by early attempts to use static AOP in the FreeBSD kernel [31] as well as dynamic instrumentation in Solaris' DTrace toolkit [23]. The following goals were reached in the context of this thesis:

- **Identification of crosscutting properties in procedural code**

  This thesis started out by giving an overview of the problems involved with crosscutting concerns in procedural languages and especially in the structural context of an operating system kernel.

- **Definition of an AOP subset suitable for C**

  After defining the nature of crosscutting concerns for procedural systems, approaches to adapt AOSD technologies to these systems were shown and their limitations were outlined.

- **Creation of a prototypical AspectC compiler**

  The slow adoption of AOP technologies in kernel development is in part also related to missing or non-mature tools. Especially for system-level code, support for many compiler extensions and system adaptations is required in order to work on millions of lines of production-quality code. With the creation of a prototypical AspectC compiler based on proven code transformation tools that is able to compile large real-world software systems like the NetBSD kernel, this thesis provided a first step in providing an aspect-oriented toolset for C programming. Furthermore, this compiler is the basis for the currentintegration of AspectC as a compiler frontend in the ubiquituos GCC compiler toolchain.

- **Development of a dynamic AOP toolkit for NetBSD**

  While static aspects provide a tool for many useful applications, the functionality provided by static AOP is limiting for applications inside the kernel. Kernel code reacts to external, asynchronous events like hardware interrupts, user input and page faults. Handling a crosscutting concern with a static aspect that is only relevant in a few special cases implies that checking for applicability of the aspect generates an unnecessary overhead for the majority of the use cases of this code. This, in turn, will slow down the complete operation of the system. As a consequence, dynamic aspects – aspects that can be dynamically activated and deactivated at system runtime – provide a tool that allows for better integration of AOSD principles with the perfomance requirements of a kernel. This thesis presented TOSKANA, the first implementation of dynamic aspects in the NetBSD kernel.

- **Creation of a microkernel- and VM-based dynamic AOP toolkit**

  Based on experience with TOSKANA, a different approach – TOSKANA-VM – for deploying dynamic aspects inside a kernel is presented that is based on a microkernel executing one or more kernel personalities in user mode. In order to overcome some of TOSKANA's shortcomings, TOSKANA-VM introduced a lightweight virtual machine below the kernel personalities that handles the deployment and undeployment of dynamic aspects.

- **Novel applications for AOP in kernel code**

  A new technology is mostly useless without applications. To illustrate the usefulness of static and dynamic AOP inside a kernel, a set of applications with a broad range of topics was chosen – cross-layer networking, crosscutting concerns between file system and process management and optimization inside the virtual file system. In addition, further crosscutting concerns inside the kernel were presented.

11.2 Future Work

A thesis covering several larger software development projects leaves some open ends. While TOSKANA itself is a working prototype, stability and usability improvements, especially an integration with the AspectC compiler system, are required. In this context, the AspectC compiler has to be brought from a prototype stage to a fully integrated compiler, possibly implemented as part of the GNU compiler collection.

Identifying crosscutting concerns that can be refactored into aspects – so-called "aspect mining" is one of the requirements for the widespread adoption of AOP, especially when maintaining existing large software systems. Current aspect mining approaches require either the help of a human expert [19, 20] or are restricted to a very small part of crosscutting problems that are easy to identify [18, 24]. For a large-scale refactoring of kernel code, both approaches are too limited; instead, using a semantic model of kernel tasks and related code patterns could improve the quality of machine-based aspect mining since it makes the identification of tangled and scattered code easier and provides a relation to the code's functional context.

The approach used in TOSKANA-VM is an experiment based on a very volatile and rapidly changing structure. The L4 microkernel itself is in a constant state of change. Currently, discussions ensue about the abandonment of the L4Linux kernel personality in favor of a new pre-virtualization technology, “Afterburning” [137], that provides a compiler-supported OS virtualization approach. This shows one of the current trends in operating system, which our future research tries to leverage: the combination of improved support for programming language at runtime with pervasive deployment of virtualization technology and microkernel-based approaches can redefine the notion of the execution environment which is currently quite close to existing hardware. This has direct implications on the development of operating systems, since these will achieve the ability to gain more platform independence. This enables more flexible uses in distributed environments like Grid systems, where moving complete OS installations ad hoc over the network is one approach to application deployment. Implementing parts of the OS as components on top of a virtualization infrastructure, in turn, creates new crosscutting problems. Here, aspect-oriented approaches can serve to handle this increased complexity of interactions.

The following section, derived from a first sketch published in [52], describes a vision of an aspect-oriented operating system, based on functional components implemented in various mainstream as well as domain-specific languages that handle their crosscutting concerns using dynamic, language-agnostic aspects. The basis for this system is a combination of an extremely lightweight controlling component (derived from micro- and exokernel research) with an extensive framework supporting static and dynamic aspects through kernel-integrated compiler and transformation technology.

11.3 Vision: An AOP-based Multi-Server OS

The aim of applying novel software engineering technologies to an existing source code base is to create a microkernel-based multiserver operating system that reuses as many of the components as possible while reducing the amount of manual refactoring and code changes as much as possible.

The successful construction of a multiserver operating system kernel has to combine several software engineering technologies in order to create the blueprint for a system that mainly consists of recycled source code components.

With support from expert knowledge, the first step is to analyze the modularization structure of the existing code. As much of the OS kernel functionality is contained in layers, these provide a good starting point for analysis. A simplified view of a possible decomposition is shown in fig. 11.1. In the multi-server view on the right side, the arrows indicate the various crosscutting interdependencies created by decomposing the layered monolithic structure into independent multi-server components.

The next step is to decompose the modules detected into separate entities. In this process, crosscutting concerns have to be detected. Technologies for the detection of crosscuts include code similarity analysis and the creation of function call graphs. Interdependencies between code in different modules are indicators for possible crosscutting concerns. Again, expert knowledge is required here in order to categorize the concerns.

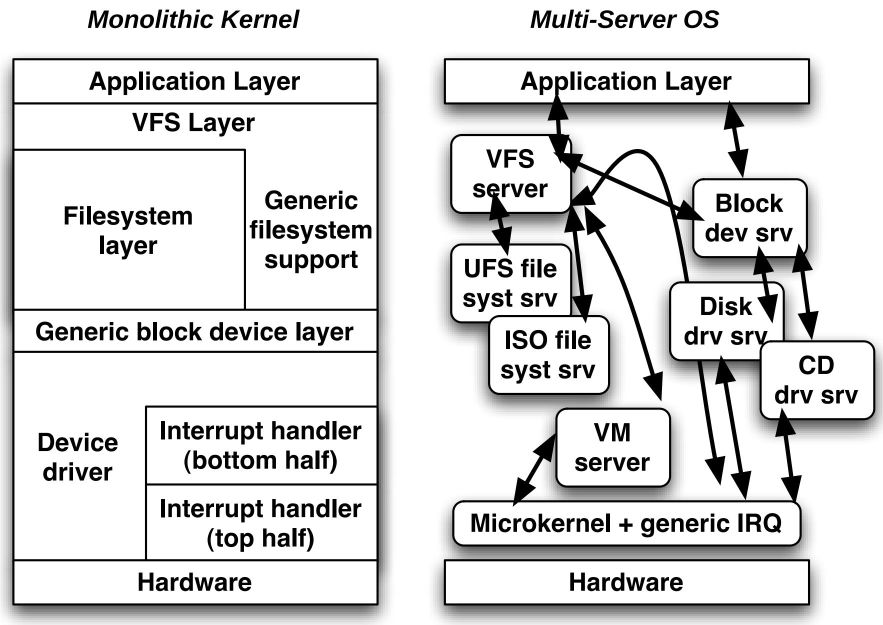


Figure 11.1: Multiserver OS structure

The original interfaces between the layers now have to be redesigned in order to use the communication primitives provided by the underlying microkernel. Refactoring methods and automatic stub code creation take care of this step, replacing affected function calls with calls to the provided IPC library.

Another important part of the system that has to be taken into account is code concerned with tasks that are in the responsibility of the microkernel in the new multiserver OS. These microkernel dependencies can also be detected by automatic source code analysis, supported by expert knowledge providing information on the affected parts. A further refactoring step takes care of the adaptation itself.

A major part of adapting and refactoring the system consists of rerouting and manipulation function calls in the reused code. Here, AOP can be helpful by providing *around* advice for implementing stub routines that change function calls into calls to the IPC library. Of special interest here is the employment of dynamic aspect orientation, as other system components can possibly be exchanged on the fly in a running system, requiring instantaneous adaptation of the stubs to new IPC message formats.

11.3.1 Domain-Specific Languages

Development of kernel code is a complex task. However, a significant part of the code to be implemented, like network protocols, file system operations and system calls, is written according to (more-or-less) formal specifications like internet RFCs.

Designing and implementing network protocols with conventional languages and tools is difficult. The protocols themselves are hard to design; implementing a protocol correctly is another challenge. Furthermore, protocol efficiency has become vital with the growing importance of networking, and the occasional need for protocol extensions complicates the issue. Unfortunately, these tensions work against one another. Many optimizations which make protocol code more efficient also tend to make it much harder to understand, and therefore harder to get right. Extensions affect deeply buried pieces of protocol code rarely identifiable a priori. Finally, the clearest organization of protocol code is often among the slowest. Domain-specific language (DSL) tools are a natural area to investigate for a solution to this software engineering problem.

Prolac [84] is a protocol language designed to ease the implementation of network code. It addresses all three issues in protocol implementation: correctness, efficiency, and extensions, implementing three specific goals: to implement protocol-specific optimizations, thus creating highperformance protocol implementations; to facilitate protocol extensions; and to make protocol implementations more tractable to human readers, and thus easier for people to reason about. To demonstrate the usabiliy of Prolac, an implementation of the TCP/IP (v4) protocol stack in Prolac was provided for Linux [85] that has a performance comparable to the original Linux implementation.

The BSD packet filter [97] follows a similar, yet less complex approach. Here, network packets can be processed, captured and rewritten using a simple in-kernel virtual machine that is able to intercept packet processing.

Another area in which domain-specific languages have been proposed is scheduler implementation. Emerging applications, such as multimedia applications and real-time applications, have increasingly specialized scheduling requirements. Nevertheless, developing a new scheduling policy and integrating it into an existing OS is complex, because it requires understanding (often implicit) OS conventions. One example is the Bossa framework for scheduler development [8]. Bossa is a kernel-level event-based framework to facilitate the implementation and integration of new scheduling policies. Bossa provides simplified scheduler implementation; the Bossa framework includes a domain-specific language that provides high-level scheduling abstractions that simplify the implementation and evolution of new scheduling policies. A dedicated compiler checks Bossa DSL code for compatibility with the target OS and translates the code into C.

Device drivers can also profit from using DSLs. Two examples of DSLs used for implementing drivers are NDL [35] and GAL [132]. NDL is a language for device driver development that provides high-level constructs for device programming, describing the driver in terms of its operational interface. Its declarations are designed to closely resemble the specification document for the device it controls. A NDL driver is typically composed of a set of register definitions, protocols for accessing those registers, and a collection of device functions. The compiler translates register definitions and access protocols into an abstract representation of the device interface. Device functions are then translated into a series of operations on that interface. GAL specializes in device drivers for graphics cards, providing an environment for controlling parameters like resolution, color depth, etc.

An extensive overview of domain-specific languages for various use cases is given in [38]. Domain-specific languages, integrated with a common basic virtual machine, can help to facilitate system-level code development and provide a more high-level semantics for pieces of code with which aspects can interact.

11.3.2 Language-Agnostic Aspects

The approach of using domain-specific languages in a broad variety in kernel code leads to a problem for deploying aspects. Currently, most aspect definitions are too much related to a specific programming language.

In order to work inside a multi-language environment, a defintion of aspects is required that abstracts from a specific programming language and concentrates on common syntactic and semantic properties of the code involved – so-called “language-agnostic aspects”. Approaches to implement such aspects can e.g. be found in the VEST toolkit [126] as well as Microsoft’s Phoenix and Compose toolkits [12].

Implementing a semantic view on aspects is no new idea. Approaches to semantic definition of aspects have been published e.g. in [57], [22], [98] and [125]. To apply a semantic approach to operating system code, especially if some of this code is implemented in a domain-specific language, a compherensive semantic description of a system is required – due to the complexity of this task, however, none exists so far. Semantic descriptions for well-defined functionality inside a kernel, however, has been specified. In [71], the semantics of timing behavior in kernels is analyzed, whereas [114] describes semaphore semantics.

Creating a more complete semantic description of kernel operations and possible deriving this information from existing source code or providing semantic-aware DSLs will be an interesting task that could redefine approaches to OS development.

11.3.3 Interrupt Handling

In microkernel-based systems, interrupt handling has to be handled by the microkernel itself in order to avoid crashing or blocking of the system by a kernel personality as well as to assure proper interrupt distribution over several personalities running concurrently. In addition, in a multiserver-based OS, the interrupts have to be routed to the correct component.

Fig. 11.2 shows interrupt handling in L4Linux running on top of L4 where device interrupts are received by the microkernel and in turn translated into IPC messages that are delivered to the monolithic Linux kernel personality’s (virtual) interrupt handlers that in turn employ the bottom halves of the appropriate device drivers.

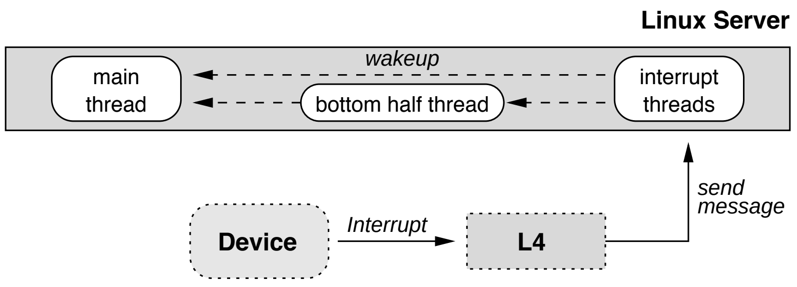


Figure 11.2: Interrupt handling in L4Linux

In a multi-server system, interrupt routing has to be more elaborate since interrupt handlers are now distributed over various device handling components. The most obvious crosscutting concern here is the enabling and disabling of interrupts. Here, the advice code has to synchronize enabling and disabling of varying sets of interrupts that are usually coordinated via the system-provided `spl`*x* functions. This is complicated in a microkernel-based multiserver system, since interrupt control is now distributed over various servers and has to be synchronized using IPC messages instead of controlling the hardware directly.

Fig. 11.3 shows an approach to solve the interrupt problem using TOSKANA's dynamic aspect handling capabilities, here running as a separate server component in the multi-server environment that has special rights to deploy advice code into other servers' address spaces.

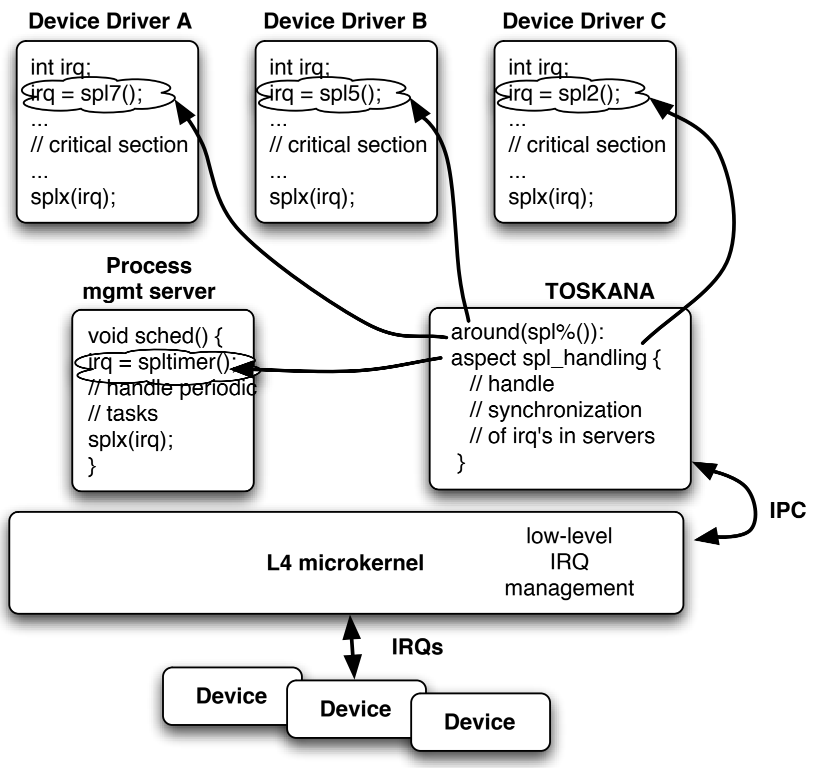


Figure 11.3: Interrupt handling using aspects

11.3.4 File System Access

In modern Unix systems, support for a large number of different file systems is available, ranging from simple on-disk file systems to distributed, redundant network file systems for cluster environments and extending to exotic solutions like virtual file systems that represent the dynamic changes in the devices attached to a system (*devfds*) or permit access to network connections using a pathname-based expression instead of socket operations (*portalfs*).

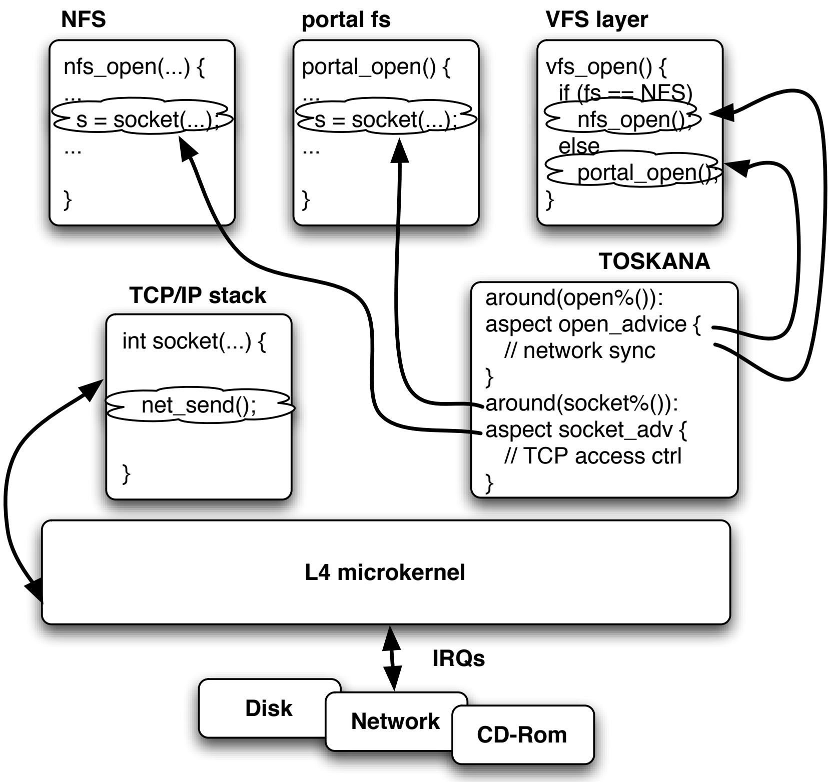


Figure 11.4: Interactions of file systems with the VFS server

When decomposing the intertwined operations of the virtual file system layer (VFS), the various file system implementations (e.g., the NFS network file system and portalfs), and the related TCP/IP stack, a new set of crosscutting concerns is generated, since some file systems contain common operations that are handled by the VFS, distributed file access control has to occur and network access of all file system servers using networking protocols has to be coordinated.

Fig. 11.4 shows the virtual file system as well as various individual file system server components of a multi-server OS. Here, aspects handle correct distribution of file system access from user mode to the various file system servers as well as synchronization for access to the network stack from the various file systems that generate networking activity.

11.3.5 Discussion

The aspect-oriented operating system structure described above deviates from mainstream operating system development methods in that the focus of system development is shifted towards improved language support, semantic analysis and using specialized languages for rapid and less error-prone code development. Security of system components can be improved by implementing the multi-server approach. Crosscutting problems created by this approach as well as the handling of dynamically changing requirements at system runtime can be handled by language-agnostic dynamic aspect tools.

It remains to be seen if this system approach is feasible. Among the problems to be solved here are inter-process and inter-server communication performance, handling the complexity of crosscutting concerns and defining approaches for language-agnostic aspects by shifting the definition of pointcuts from syntactic to semantic levels. This paradigm shift brings along improved introspection capabilities that allow aspects to analyze the current state of the system and its components.

However, in accordance with Rob Pike's ideas, such an approach should be tried, so that operating systems research can once again combine engineering and design and bring a small piece of the lost art of systems design back, represented by combining several "hot" technologies to form new concepts.

■

Bibliography

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation for UNIX Development. *Proceedings of the Usenix 1986 Summer Conference*, pages 93–112, July 1986.

[2] M. Allman. *RFC3465: TCP Congestion Control with Appropriate Byte Counting (ABC)*. IETF, 2003.

[3] G. Almasi, R. Bellofatto, C. Cascaval, J. G. Castanos, L. Ceze, P. Crumley, C. C. Erway, J. Gagliano, D. Lieber, J. E. Moreira, A. Sanomiya, and K. Strauss. An Overview of the Blue Gene/L System Software Organization. In *Proceedings of the International Conference on Parallel and Distributed Computing (Euro-Par), Klagenfurt, Austria*, pages 543–555, 2003.

[4] AOSD Wiki. http://aosd.net/.

[5] A. Appel and A. Felty. A Semantic Model of Types and Machine Instructions for Proof-Carrying Code. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 243–254. ACM Press, 2000.

[6] O. Arndt, B. Freisleben, T. Kielmann, and F. Thilo. A Comparative Study of Online Scheduling Algorithms for Networks of Workstations. In *Cluster Computing 3 (2000), Nr. 2*, pages 95–112. Kluwer Publishers, 2000.

[7] P. T. Barham, B. Dragovic, K. Fraser, S. Hand, T. L. Harris, A. Ho, and R. Neugebauer. Xen and the Art of Virtualization. *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 164–177, 2003.

[8] L. P. Barreto. Bossa: A DSL Framework for Application-Specific Scheduling Policies. In *HOTOS '01: Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, pages 161–164, Washington, DC, USA, 2001. IEEE Computer Society.

[9] A. Bauer and M. Pizka. The Contribution of Free Software to Software Evolution. In *Proceedings of the International Workshop on Principles of Software Evolution (IWPSE)*, pages 170–182, Helsinki, Finland, Sept. 2003. IEEE Computer Society.

[10] R. Baumgartl, M. Borriss, H. Härtig, C.-J. Hamann, M. Hohmuth, L. Reuther, S. Schönberg, and J. Wolter. Dresden Realtime Operating System. *Proceedings of the Workshop of System-Designed Automation (SDA'98)*, pages 14–21, 1998.

[11] F. Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Usenix 2005 Annual Technical Conference*, pages 41–46. Usenix, 2005.

[12] L. Bergmans. Compose *: Language-independent Aspects in .NET. Invited talk at Microsoft Academic Days in Stockholm, http://www.microsoft.com/sverige/events/academicdays.asp, 2004.

[13] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 267–283, New York, NY, USA, 1995. ACM Press.

[14] G. J. Bex, S. Maneth, and F. Neven. Expressive Power of XSLT, http://alpha.luc.ac.be/ fneven/.

[15] C. Bockisch, M. Haupt, M. Mezini, and K. Ostermann. Virtual Machine Support for Dynamic Join Points. In *Proceedings of the Third International Conference on Aspect-Oriented Software Development (AOSD'04)*, pages 83–92. ACM Press, 2004.

[16] R. Braden, T. Faber, and M. Handley. From Protocol Stack to Protocol Heap: Role-Based Architecture. *SIGCOMM Comput. Commun. Rev.*, 33(1):17–22, 2003.

[17] S. Bray, M. Yuen, Y. Coady, and M. E. Fiuczynski. Managing Variability in Systems: Oh What a Tangled OS We Weave. In *Proceedings of the Workshop on Managing Variabilities Consistently in Design and Code (MVCDC OOPSLA04)*, pages 6–7, 2004.

[18] S. Breu and J. Krinke. Aspect Mining Using Dynamic Analysis. In *GI-Softwaretechnik-Trends, Mitteilungen der Gesellschaft fur Informatik. Volume 23., Bad Honnef, Germany*, pages 21–22, 2003.

[19] M. Bruntink, A. van Deursen, and T. Tourwé. An Initial Experiment in Reverse Engineering Aspects from Existing Applications. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, pages 306–307. IEEE Computer Society, 2004.

[20] M. Bruntink, A. van Deursen, R. van Engelen, and T. Tourwé. An Evaluation of Clone Detection Techniques for Identifying Cross-Cutting Concerns. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*, pages 200–209. IEEE Computer Society, 2004.

[21] M. Burnet and B. Supnik. Preserving Computing's Past: Restoration and Simulation. *Digital Technical Journal Volume 8, Number 3*, pages 23–38, 1996.

[22] J. Cachopo. Separation of Concerns Through Semantic Annotations. In *OOPSLA '02: Companion of the 17th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 2–3, New York, NY, USA, 2002. ACM Press.

[23] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic Instrumentation of Production Systems. In *Proceedings of the Usenix Annual Technical Conference*, pages 15–28. Usenix, 2004.

[24] M. Ceccato, M. Marin, K. Mens, L. Moonen, P. Tonella, and T. Tourwe. A Qualitative Comparison of Three Aspect Mining Techniques. In *IWPC '05: Proceedings of the 13th International Workshop on Program Comprehension*, pages 13–22, Washington, DC, USA, 2005. IEEE Computer Society.

[25] D. Chamberlin, J. Clark, D. Florescu, J. Robie, J. Simon, and M. S. (Eds). XQuery 1.0: An XML Query Language. *W3C Working Draft, June 2001. http://www.w3.org/TR/xquery/*, 2001.

[26] A. Chernoff, M. Herdeg, R. Hookway, C. Reeve, N. Rubin, T. Tye, S. B. Yadavalli, and J. Yates. FX!32: A Profile-Directed Binary Translator. *IEEE Micro*, 18(2):56–64, Mar/Apr 1998.

[27] C. Cifuentes and M. V. Emmerik. UQBT: Adaptable Binary Translation at Low Cost. *Computer*, 33(3):60–66, 2000.

[28] C. Cifuentes, B. Lewis, and D. Ung. Walkabout - a Retargetable Dynamic Binary Translation Framework. In *Technical Report TR2002 -106, Sun Microsytems Laboratories*, pages 1–30, 2002.

[29] C. Cifuentes and S. Sendall. Specifying the Semantics of Machine Instructions. In *Proceedings of the 6th International Workshop on Program Comprehension - IWPC'98*, pages 126–133. IEEE Press, 1998.

[30] Y. Coady and G. Kiczales. Back to the Future: A Retroactive Study of Aspect Evolution in Operating System Code. In *Proceedings of the Second International Conference on Aspect-Oriented Software Development (AOSD'03)*, pages 50–59. ACM Press, 2003.

[31] Y. Coady, G. Kiczales, M. Feeley, N. Hutchinson, and J. S. Ong. Structuring System Aspects: Using AOP to Improve OS Structure Modularity. In *Communications of the ACM, Volume 44, Issue 10*, pages 79–82. ACM Press, 2001.

[32] Y. Coady, G. Kiczales, M. Feeley, N. Hutchinson, J. S. Ong, and S. Gudmundson. Exploring an Aspect-Oriented Approach to OS Code. In *Proceedings of the 4th Workshop on Object-Orientation and Operating Systems at the 15th European Conference on Object-Oriented Programming (ECOOP-OOOSW)*, pages 7–11. Universidad de Oviedo, 2001.

[33] Y. Coady, G. Kiczales, M. Feeley, N. Hutchinson, J. S. Ong, and S. Gudmundson. Position Summary: Aspect-Oriented System Structure. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HOTOS-VIII)*, page 166. IEEE Press, 2001.

[34] Y. Coady, G. Kiczales, M. Feeley, and G. Smolyn. Using AspectC to Improve the Modularity of Path-Specific Customization in Operating System Code. In *Proceedings of the Joint European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-9)*, pages 88–98. ACM Press, 2001.

[35] C. L. Conway and S. A. Edwards. NDL: a Domain-Specific Language for Device Drivers. In *LCTES '04: Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 30–36, New York, NY, USA, 2004. ACM Press.

[36] F. J. Corbató, M. M. Daggett, and R. C. Daley. An Experimental Time-Sharing System. In *Proceedings of the Spring Joint Computer Conference*, pages 117–137, 1962.

[37] C. Cranor and G. Parulkar. The UVM Virtual Memory System. In *Proceedings of the Usenix Annual Technical Conference*, pages 117–130. Usenix, 1999.

[38] A. v. Deursen, P. Klint, and J. Visser. Domain-Specific Languages: An Annotated Bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, June 2000.

[39] C. DiBona, S. Ockman, and M. Stone. *Open Sources - Voices from the Open Source Revolution*. O'Reilly, Sebastopol, 1999.

[40] E. W. Dijkstra. Go To Statement Considered Harmful. In *Communications of the ACM*, pages 147–148. ACM Press, 1968.

[41] E. W. Dijkstra. Hierarchical Ordering of Sequential Processes. *The Origin of Concurrent Programming: From Semaphores to Remote Procedure Calls*, pages 198–227, 2002.

[42] S. Dorward, R. Pike, D. Presotto, D. Ritchie, H. Trickey, and P. Winterbottom. The Design of the Inferno Virtual Machine. In *Proceedings of IEEE Compcon*, pages 241–244, 1997.

[43] S. Dorward, R. Pike, D. L. Presotto, D. Ritchie, H. Trickey, and P. Winterbottom. The Inferno Operating System. *Bell Labs Technical Journal*, 2(1):5–18, Winter 1997.

[44] S. Dorward, R. Pike, and P. Winterbottom. Programming in Limbo. In *Proceedings of the IEEE Compcon 97 Conference*, pages 245–250, San Jose, 1997.

[45] R. Douence, O. Motelet, and M. Südholt. A Formal Definition of Crosscuts. *Lecture Notes in Computer Science*, 2192:170–184, 2001.

[46] F. Douglis. Transparent Process Migration in the Sprite Operating System. Technical Report CSD-90-598, University of California, Berkeley, University of California, Berkeley, 1990.

[47] M. Eichberg. BAT2XML: XML-based Java Bytecode Representation. In *Proceedings of Bytecode' 05, Edinburgh, Scotland*. Elsevier, 2005.

[48] M. Engel and B. Freisleben. Wireless Ad-Hoc Network Emulation Using Microkernel-Based Virtual Linux Systems. *Proceedings of the Federation of European Simulation Societies Conference (EuroSIM) 2004*, pages 198–203, 2004.

[49] M. Engel and B. Freisleben. On Aspect Deployment into Native Code. In *Proceedings of IEEE EUROCON 05, Belgrade, Serbia*, pages 731–736. IEEE Press, 2005.

[50] M. Engel and B. Freisleben. Supporting Autonomic Computing Functionality via Dynamic Operating System Kernel Aspects. In *Proceedings of the Fourth International Conference on Aspect Oriented System Development*, pages 51–62. ACM Press, 2005.

[51] M. Engel and B. Freisleben. Using a Low-Level Virtual Machine to Improve Dynamic Aspect Support in Operating System Kernels. *Proceedings of the ACP4IS Workshop at the Fourth Conference on Aspect-Oriented Software Development (AOSD'05)*, pages 1–6, 2005.

[52] M. Engel, M. Mezini, and B. Freisleben. Creating a Component-Based Multi-Server OS From Existing Source Code Using Aspect-Oriented Programming. In *Proceedings of the International Conference on Communication, Computer and Power (ICCCP'05)*, pages 360–367. IEEE Press, 2005.

[53] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Symposium on Operating Systems Principles*, pages 251–266, 1995.

[54] R. Filman, T. Elrad, S. Clarke, and M. Aksit. *Aspect-Oriented Software Development.* Addison-Wesley, 2005.

[55] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, Summer 1997.

[56] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure.* Morgan-Kaufman Publishers, 1997.

[57] P. Fradet and M. Sudholt. AOP: Towards a Generic Framework Using Program Transformation and Analysis. In *ECOOP '98 Workshop Reader*, pages 394–397, 1998.

[58] T. Friese, M. Smith, and B. Freisleben. Hot Service Deployment in an Ad Hoc Grid Environment. In *Proceedings of the 2nd Int. Conference on Service-Oriented Computing (ICSOC'04), New York, USA*, pages 75–83. ACM Press, 2004.

[59] A. Gal, W. Schröder-Preikschat, and O. Spinczyk. On Minimal Overhead Operating Systems and Aspect-Oriented Programming. In *Proceedings of the 4th Workshop on Object-Orientation and Operating Systems at the 15th European Conference on Object-Oriented Programming (ECOOP-OOOSW).* Universidad de Oviedo, 2001.

[60] GCC Translation Framework. http://www.codetransform.com/gcc.html.

[61] D. P. Ghormley, D. Petrou, S. H. Rodrigues, and T. E. Anderson. SLIC: An Extensibility System for Commodity Operating Systems. In *Proceedings of the USENIX 1998 Annual Technical Conference*, pages 39–52. USENIX Association, June 15–19 1998.

[62] M. W. Godfrey and Q. Tu. Evolution in Open Source Software: A Case Study. In *ICSM*, pages 131–142, 2000.

[63] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[64] R. Grimm. xtc: Making C Safely Extensible. In *Proceedings of the Workshop on Domain-Specific Languages for Numerical Optimization, Argonne National Laboratory*, 2004.

[65] B. Griswold, E. Hilsdale, J. Hugunin, V. Ivanovic, M. Kersten, G. Kiczales, and J. Palm. Tutorial - Aspect-Oriented Programming with AspectJ, 2001.

[66] P. B. Hansen. The Nucleus of a Multiprogramming System. In *Comm. ACM, 13(4):238–250.* ACM Press, 1970.

[67] e. Harold Ossher and Gregor Kiczales. *AOSD '02: Proceedings of the 1st international conference on Aspect-Oriented Software Development*. ACM Press, New York, NY, USA, 2002.

[68] H. Härtig, M. Hohmuth, and J. Wolter. Taming Linux. In *Proceedings of the 5th Australasian Conference on Parallel and Real-Time Systems*. IEEE Press, September 1998.

[69] M. Haupt and M. Mezini. Micro-Measurements for Dynamic Aspect-Oriented Systems. In M. Weske and P. Liggesmeyer, editors, *Proceedings of Net.ObjectDays*, volume 3263 of *LNCS*, pages 81–96. Springer Press, 2004.

[70] J. Hill, R. Szewcyk, A. Woo, D. Culler, S. Hollar, and K. Pister. System Architecture Directions for Networked Sensors. In *Proceedings of the Seventh Conference on Architectural Support for Programming Languages and Operating Systems*, pages 93–104. ACM Press, 2000.

[71] V. Hirvisalo and S. Kiminki. Predictable Timing Behavior by using Compiler Controlled Operation. In *Proceedings of 4th International Workshop on Worst-case Execution Time Analysis (WCET'04)*, pages 3–6, 2004.

[72] U. Hölzle, L. Bak, S. Grarup, R. Griesemer, and S. Mitrovic. Java on Steroids: Sun's High-Performance Java Implementation. In *Proc. of HotChips IX* , pages 22–33. IEEE Press, August 1997.

[73] G. C. Hunt and J. R. Larus. Singularity Design Motivation. *Microsoft Technical Report TR-2004-105*, 2004.

[74] IBM Research. *Research Hypervisor Hackers Guide*, http://www.research.ibm.com/hypervisor/hackersguide.shtml.

[75] Intel. *IA-32 Intel Architecture Software Developer's Manual, Volume 2A/B: Instruction Set Reference*. Intel, 2004.

[76] Introduction to NetBSD Loadable Kernel Modules. http://www.home.unix-ag.org/bmeurer/netbsd/howto-lkm.html.

[77] ISO. The Open Systems Interconnection (OSI) Reference Model Standard.

[78] T. Jensen, D. Le Métayer, and T. Thorn. Security and Dynamic Class Loading in Java: A Formalisation. In *IEEE International Conference on Computer Languages*, pages 4–15, Chicago, Illinois, 1998.

[79] S. C. Johnson. Yacc: Yet Another Compiler Compiler. In *UNIX Programmer's Manual*, volume 2, pages 353–387. Holt, Rinehart, and Winston, New York, NY, USA, 1979.

[80] W. Jolitz and L. Jolitz. Porting UNIX to the 386: the Standalone System. *Dr. Dobb's Journal*, 16(3):38–88, March 1991.

[81] J. O. Kephart and D. M. Chess. The Vision of Autonomic Computing. In *IEEE Computer 01/2003*, pages 43–50. IEEE Press, 2003.

[82] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In M. Akşit and S. Matsuoka, editors, *Proceedings of the European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[83] S. Kochan. *Programming Objective C.* Sams Publishers, 2003.

[84] E. Kohler. Prolac: a Language for Protocol Compilation (Masters' Thesis, MIT), 1998.

[85] E. Kohler, M. F. Kaashoek, and D. R. Montgomery. A Readable TCP in the Prolac Protocol Language. In *ACM SIGCOMM – Journal of the Special Interest Group on Data Communications*, pages 3–13, 1999.

[86] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04), Palo Alto, California*, pages 75–84. ACM Press, March 2004.

[87] K. Lawton, B. Denney, and C. Bothamy. *Bochs Development Guide.* http://www.bochs.net, 2005.

[88] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quaterman. *The Design and Implementation of the 4.4 BSD UNIX Operating System.* Addison-Wesley, Reading, Mass., 1994.

[89] J. Liedtke. A Persistent System in Real Use: Experiences of the First 13 Years. In *Proceedings of the International Workshop on Object-Orientation in Operating Systems (IWOOOS)*, pages 27–36, 1993.

[90] J. Liedtke. μ-Kernels Must And Can Be Small. In *Proceedings of the 5th IEEE International Workshop on Object-Orientation in Operating Systems (IWOOOS), Seattle, WA*, pages 152–155. IEEE Press, October 1996.

[91] T. Lindholm and F. Yellin. *Java Virtual Machine Specification, Second Edition.* Addison-Wesley, Reading, Mass., 1999.

[92] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Programming Language Design and Implementation*, pages 190–200, Chicago, IL, June 2005.

[93] E. Mamas and K. Kontogiannis. Towards Portable Source Code Representation Using XML. In *Proceedings of the Seventh Working Conference on Reverse Engineering*, pages 172–182. IEEE Computer Society Press, 2000.

[94] M. K. McKusick and G. V. Neville-Neil. *The Design and Implementation of the FreeBSD Operating System.* Addison-Wesley, Reading, Mass., 2004.

[95] D. Milojicic, W. Zint, and A. Dangel. *Task Migration on Top of the Mach Microkernel – Design and Implementation – (Technical Report).* University of Kaiserslautern, 1992.

[96] J. G. Mitchell. JavaOS: Back to the Future. In *Proceedings of the 1996 Conference on Operating Systems Design and Implementation (OSDI)*, pages 1–6, 1996.

[97] J. Mogul, R. Rashid, and M. Accetta. The Packet Filter: An Efficient Mechanism for User-Level Network Code. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP)*, volume 21, pages 39–51, 1987.

[98] M. Mousavi, G. Russello, M. Chaudron, M. A. Reniers, T. Basten, A. Corsaro, S. Shukla, R. Gupta, and D. Schmid. Aspects + GAMMA = AspectGAMMA: A Formal Framework for Aspect-Oriented Specification. In *AOSD '02: Proceedings of the 1st International Conference on Aspect-oriented Software Development*, pages 69–75, 2002.

[99] R. A. Mullen. z/VM: The Value of zSeries Virtualization Technology for Linux. In *Proceedings of the IBM z/VM, VSE and Linux on zSeries Technical Conference, Miami, Florida*, pages 34–43. IBM, 2002.

[100] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of c programs. In *International Conference on Compiler Construction (CC '02)*, pages 213–228, 2002.

[101] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-Safe Retrofitting of Legacy Code. In *Symposium on Principles of Programming Languages*, pages 128–139, 2002.

[102] E. Organick. *The Multics System: An Examination of its Structure.* MIT Press, 1972.

[103] T. Parr and R. Quong. ANTLR: A Predicated-LL(k) Parser Generator. In *Journal of Software Practice and Experience, Vol. 25(7)*, pages 789–810. IEEE Press, 1995.

[104] D. J. Pearce, P. H. J. Kelly, T. Field, and U. Harder. GILK: A Dynamic Instrumentation Tool for the Linux Kernel. In *Computer Performance Evaluation / TOOLS*, pages 220–226, 2002.

[105] S. Pemberton and M. Daniels. *The P4 Compiler and Interpreter.* Halstead Press, 1983.

[106] R. Pike. Systems Software Research is Irrelevant. *freshmeat Editorial: http://freshmeat.net/articles/view/175/*, 2000.

[107] R. Pike, D. Presotto, S. Dorward, B. F. K. Thompson, H. Trickey, and P. Winterbottom. Plan 9 from Bell Labs. *Computing Systems*, 8(3):221–254, Summer 1995.

[108] I. Piumarta, B. Folliot, L. Seinturier, C. Baillarguet, and C. Khoury. Highly Configurable Operating Systems: the VVM Approach. In *Proceedings of the 3rd Workshop on Object-Orientation and Operating Systems at the 14th European Conference on Object-Oriented Programming (ECOOP-OOOSW)*. ECOOP, 2000.

[109] V. Prasad, W. Cohen, F. C. Eigler, M. Hunt, J. Keniston, and B. Chen. Locating System Problems Using Dynamic Instrumentation. In *Proceedings of the Ottawa Linux Symposium*, pages 49–64. Linux International, 2005.

[110] N. Provos. Improving Host Security with System Call Policies. In *Proceedings of the 12th Usenix Security Symposium*, pages 257–272. Usenix, 2003.

[111] D. Rémy. Using, Understanding, and Unraveling the OCaml Language. In G. Barthe, editor, *Applied Semantics. Advanced Lectures. LNCS 2395*, pages 413–537. Springer Verlag, 2002.

[112] D. M. Ritchie and K. Thompson. The UNIX Time-Sharing System. *The Bell System Technical Journal*, 57(6 (part 2)):1905+, 1978.

[113] O. Rohlik, A. Pasetti, P. Chevalley, and I. Birrer. An Aspect Weaver for Qualifiable Applications. In *Proceedings of the 2004 Conference on Data Systems in Aerospace (DASIA)*, pages 10–16, 2004.

[114] S. Rönn. Semantics of Semaphores. Research Report A53, Helsinki University of Technology, Digital Systems Laboratory, Espoo, Finland, 1998.

[115] P. Salus. *A Quarter Century of UNIX*. Addison-Wesley, New York, 1994.

[116] B. Scheuermann, C. Lochert, and M. Mauve. Implicit Hop-by-Hop Congestion Control in Wireless Multihop Networks. In *Internal Preprint*. University of Duesseldorf, 2005.

[117] S. Schonger, E. Pulvermüller, and S. Sarstedt. Aspect-Oriented Programming and Component Weaving: Using XML Representations of Abstract Syntax Trees. In *Proceedings of the 2nd German GI Workshop on Aspect-Oriented Software Development, Bonn, Germany, February 22 - 23, 2002*, pages 59 – 64, 2002.

[118] A. T. Schreiner. *Objekt-orientierte Programmierung mit ANSI-C*. Hanser Verlag, 1994.

[119] M. Segura-Devillechaise, J.-M. Menaud, G. Muller, and J. Lawall. Web Cache Prefetching as an Aspect: Towards a Dynamic-Weaving Based Solution. In *Proceedings of the 2nd International Conference on Aspect-oriented Software Development*, pages 110–119. ACM Press, 2003.

[120] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing With Disaster: Surviving Misbehaved Kernel Extensions. *SIGOPS Oper. Syst. Rev.*, 30(SI):213–227, 1996.

[121] M. Smith, T. Friese, and B. Freisleben. Towards a Service-Oriented Ad Hoc Grid. In *Proceedings of the 3rd International Symposium on Parallel and Distributed Computing, Cork, Ireland*, pages 201–208. IEEE Press, 2004.

[122] SourceForge Software Map. http://sourceforge.net/softwaremap/trove_list.php.

[123] O. Spinczyk, A. Gal, and W. Schröder-Preikschat. AspectC++: an Aspect-Oriented Extension to the C++ Programming Language. In *Proceedings of the Fortieth International Conference on Tools*, pages 53–60. Australian Computer Society, Inc., 2002.

[124] O. Spinczyk and D. Lohmann. Using AOP to Develop Architectural-Neutral Operating System Components. In *Proceedings of the 11th ACM SIGOPS European Workshop (SIGOPS04)*, pages 41–48. ACM Press, 2004.

[125] O. Stampflee, C. Gibbs, and Y. Coady. RADAR: Really Low-Level Aspects for Dynamic Analysis and Reasoning. In *Proceedings of the ECOOP Workshop on Programming Languages and Operating Systems*, pages 17–20. ECOOP, 2004.

[126] J. A. Stankovic. VEST — A Toolset for Constructing and Analyzing Component Based Embedded Systems. *Lecture Notes in Computer Science*, 2211:390–395, 2001.

[127] B. Stroustroup. *The C++ Programming Language and Reference*. Addison-Wesley Publishing Company, 2000.

[128] Sun Microsystems. *Java Native Interface Specification*. Sun, 1997.

[129] A. Tamches and B. P. Miller. Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels. In *Proceedings of Third Symposium on Operating Systems Design and Implementation (OSDI)*, pages 117–130. ACM Press, 1999.

[130] The Free Software Foundation. *Using the GNU Compiler Collection (GCC)*. FSF, 2004.

[131] The Tool Interface Standard Committee. *Executable and Linking Format (ELF) Specification Version 1.2*. TIS Committee, May 1995.

[132] S. Thibault, R. Marlet, and C. Consel. Domain-Specific Languages: From Design to Implementation Application to Video Device Drivers Generation. *Software Engineering*, 25(3):363–377, 1999.

[133] Thomas H. Cormen and Charles E. Leiserson and Ronald L. Rivest and Clifford Stein. *Introduction to Algorithms, Second Edition*. MIT Press, 2001.

[134] Transmeta. The Technology Behind Crusoe Processors. *White Paper*, 2003.

[135] TriBUG BSD User Group. http://www.tribug.org/img/bsd-family-tree.gif.

[136] V. Uhlig, J. LeVasseur, E. Skoglund, and U. Dannowski. Towards Scalable Multiprocessor Virtual Machines. In *Proceedings of the 3rd Virtual Machine Research & Technology Symposium (VM'04)*, pages 43–56. IEEE Press, May 2004.

[137] University of Karlsruhe. Afterburning and the Accomplishment of Virtualization. In *http://www.l4ka.org/projects/virtualization/afterburn/whitepaper.pdf*, 2005.

[138] A. C. Veitch and N. C. Hutchinson. Kea - A Dynamically Extensible and Configurable Operating System Kernel. In *Proceedings of the Third International Conference on Configurable Distributed Systems*, pages 123–129, 1996.

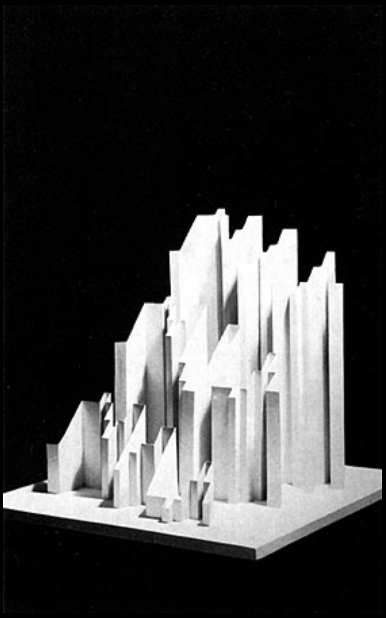
[139] VMware Whitepaper. *Accelerate Application Development, Testing, and Deployment*. VMware, Inc., 2005.

[140] L. Wall, T. Christiansen, and J. Orwant. *Programming Perl*. O'Reilly and Associates, 2000.

[141] M. Wand, G. Kiczales, and C. Dutchyn. A Semantics for Advice and Dynamic Join Points in Aspect-Oriented Programming. *Lecture Notes in Computer Science*, 2196:45–61, 2001.

[142] A. Whitaker, M. Shaw, and S. Gribble. Denali: Lightweight Virtual Machines for Distributed and Networked Applications. In *Proceedings of the USENIX Annual Technical Conference, Monterey, CA, June 2002. 10*, pages 32–47, 2002.

[143] C. Zetie. *Aspect-Oriented Programming Considered Harmful*. Forrester Group, 2005.



Colophon

The cover image is a photography of the sculpture “Fugue in es-minor” by Bauhaus artist Henri Nouveau (Henrik Neugeboren). This sculpture is a stereometric visual representation of four measures (52 to 55) of Johann Sebastian Bach’s “Fugue in es-Minor”. This work of art is a good match for visualizing the crosscutting concerns that form a central part of this thesis, as a fugue consists of three to five voices that perform according to strict rules, with melody lines crosscutting several voices while keeping to strict timing dependencies.

(Photography reproduced with friendly permission of the Bauhaus Archiv, Berlin)