Diplomarbeit

Request tracking in DROPS

Björn Döbel

30. Mai 2006

Technische Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur
Professur Betriebssysteme

Betreuender Hochschullehrer:	Prof. Dr. rer. nat. Hermann Härtig
Betreuende Mitarbeiter:	Dipl.-Inf. Martin Pohlack
	Dipl.-Inf. Ronald Aigner

Technische Universität Dresden
Fakultät Informatik

AUFGABENSTELLUNG FÜR DIE DIPLOMARBEIT

Name des Studenten:	**Björn Döbel**
Studiengang:	Informatik
Immatrikulationsnummer:	2860656

Thema: **IPC Monitoring in Mikrokern-Systemen**

Zielstellung:

Mikrokernsysteme bestehen üblicherweise aus einer Reihe von isolierten Komponenten die mittels "Interprocess Communication" (IPC) miteinander kommunizieren.

Ziel dieser Aufgabe ist es, zu untersuchen, wie diese Kommunikation im einzelnen erfolgt, welche Arten von IPC benutzt werden und wie schnell dies geschieht.

Mit Hilfe der gewonnenen Erkenntnisse sollen:

- vorhandene Protokolle überarbeitet und optimiert werden,
- alternative Protokolle vorgeschlagen und implementiert werden, die die gewonnenen Erkenntnisse sinnvoll nutzen,
- performancekritische Pfade im System bestimmt werden; wenn möglich sollen hier Lösungsvorschläge erarbeitet werden.

Im einzelnen sind dazu u.a. folgende Teilaufgaben zu lösen:

- Auswahl, Entwicklung bzw. Weiterentwicklung einer Laufzeit-Überwachungslösung, die Ergebnisse aus verschiedenen Teilsystemen integrieren kann (Kern, IPC-Stubs, L4 User-Programme, L4Linux Kern- und User-Programme)
- Anbindung der Infrastruktur an externe Evaluierungs- /Visualisierungsprogramme
- Online Auswertung einiger Leistungszahlen, um dynamisch zwischen verschiedenen Kommunikationsprotokollen umschalten zu können
- Anwendung der Infrastruktur auf einige Szenarien

verantwortlicher Hochschullehrer:	Prof. Dr. Hermann Härtig
Betreuer:	Dipl.-Inf. Martin Pohlack
Institut:	Systemarchitektur
Professur:	Betriebssysteme
Beginn:	01.12.2005
Einzureichen:	31.05.2006

[illegible]

Dresden, 01. 12. 2005

Unterschrift des verantwortlichen Hochschullehrers

Erklärung

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

Dresden, den 30. Mai 2006

Björn Döbel

Acknowledgements

I'm so happy, 'cause today I found my friends...
(Nirvana - Lithium)

I would like to thank Prof. Hermann Härtig for giving me the opportunity to work in the TU Dresden Operating Systems Group. Martin Pohlack and Ronald Aigner supported me with ideas, talks, comments and a lot of their time. Adam Lackorzynski helped me with his neverending knowledge about L4Linux and by making kProbes work on this system.

Carsten Weinhold, Sven Schneider, Thomas Friebel, Stephan Diestelhorst, and Christelle Braun colorized my life in the lab with their presence. Lars Iwer, Stefan Schiffner, and Christian Bitter proofread this thesis. Coffee and sweets were supported by the infamous "Club Dürerstraße" (R.I.P.).

My family highly influenced my life through the last 25 years and I wouldn't be what I am without them. I especially thank Christiane Berndt who gave me her support in terms of love and criticism whenever one of these was necessary.

Contents

List of Figures

List of Tables

1 Introduction

It's something unpredictable,
But in the end it's right...
(Green Day - Time of your life)

Overview

Donald Knuth once stated that "premature optimization is the root of all evil (or at least most of it) in programming." His observation was that programmers often try to write optimized code before their application is working at all. This is bad, because on the one hand these optimizations often increase the error rate in software, and on the other hand they are performed without even knowing where the performance problems are.

According to Knuth, the correct way of optimization is to start with a working but unoptimized application and perform only necessary improvements. With the evolution of software development, tools have been developed to support the task of tracking down performance issues.

These tools monitor the behavior of a single application or a whole system, collect data and visualize it for easier evaluation. My thesis aims at monitoring and evaluation of system behavior in the context of the Dresden Real-Time Operating System (DROPS). I reuse existing facilities and add new means as needed. Sensors and monitors are added to the system that support tracing of requests throughout system components. Data obtained by monitoring will be made available for offline as well as online evaluation. I show use cases for both scenarios.

L4Linux is a para-virtualized version of Linux running on top of DROPS. Monitoring events from L4Linux and its processes is another main topic of this thesis, because L4Linux is one of the most complex use cases for DROPS. I evaluate existing Linux tracing solutions to be ported to L4Linux and investigate ways of comparing L4Linux to native Linux.

With the help of the means introduced by this thesis, I am able to point out that system call overhead is one of the main reasons for L4Linux' performance problems. By implementing an optimization to L4Linux task management, I show that the cooperation between L4Linux and its environment can still be improved.

Outline

Section 2 introduces basic terms related to monitoring and reviews existing facilities in several operating systems. Thereafter I introduce means of code generation, because they can be used to automate the task of inserting sensors into applications. I introduce autonomic computing as another research area requiring efficient monitoring, and I give an overview of benchmarking as a means of evaluating performance.

In Section 3 I analyze the requirements needed for tracking requests within DROPS. I review existing tracing facilities and discuss how to instrument existing applications. I especially explain why and how to instrument the L4 Environment and L4Linux.

Section 4 explains the implementation of a trace plugin for the DROPS IDL Compiler that I used for automated instrumentation of DROPS applications. I describe the port of a TCP/IP stack to the ORe network switch to enable transfer of monitoring data through a network. Adam Lackorzynski's port of kProbes to L4Linux is introduced as well as my own work on the L4Linux task management. Thereafter I introduce a self-healing web server as an example for autonomic computing. Finally, my enhancements to the Magpie event postprocessor are described.

In Section 5 I evaluate the performance of DROPS and L4Linux using the means introduced in this thesis. Section 6 sums up my thesis and outlines ideas for future work.

2 Related work

The idea is good,
But the world isn't ready yet.
(Tocotronic)

In this section I will introduce basic terms necessary for the understanding of my thesis and present related work covering the discussed problems.

First, I will introduce the working environment of this thesis, which is the Dresden Real-Time Operating System (DROPS) in Section 2.1.

As my thesis covers tracing of events inside DROPS, I will thereafter give an introduction to monitoring in Section 2.2, followed by an overview of monitoring architectures and tool chains for the Linux, Windows, and DROPS systems. The section is concluded by an explanation of evaluation and visualization tools that support monitoring.

Tracking requests[1] in DROPS requires existing applications to be instrumented with event-generating sensors. Manual insertion of sensor code is time-consuming. Therefore I need to inspect techniques for automated sensor injection. I will introduce code generation as means of injecting sensor code into applications in Section 2.3.

Section 2.4 contains a description of autonomic computing, a concept that opens up new applications of monitoring. As a large part of my thesis will cover performance evaluation, I conclude this chapter with an overview of benchmarking.

2.1 The Dresden Real-Time Operating System

At first, I will introduce the environment in which my thesis is located. *Fiasco* is a microkernel implementing the L4 ABI [Lie96a]. It was implemented at TU Dresden in the context of the *Dresden Real-Time Operating System* (DROPS). DROPS consists of a set of cooperating *servers* running on top of the microkernel. *Inter-process communication* (IPC) is used by the servers to communicate. IPC performance is considered the main issue for microkernel-based operating systems [Lie93].

[1]In the context of my thesis, a *request* is a structured set of events.

The *L4 Environment* (L4Env) [TUDa] is a programming environment for DROPS. It provides a number of servers and libraries performing common tasks such as thread management, global naming, synchronization, and many more.

L4Linux [TUDb] is a para-virtualized version of Linux running as a DROPS server using the L4 Environment. Linux applications can run on top of it. They are handled as L4 tasks, system calls are translated into IPC to the L4Linux server. To enable native Linux binaries to run unmodified on L4Linux, *exception IPC* is introduced. A native binary issuing an `int80` to enter the kernel raises an exception in DROPS. The microkernel delegates this exception to an exception handling user-space application. In the case of L4Linux this is the L4Linux server, which thereby detects that a system call was issued and handles it.

2.2 Monitoring

My thesis aims at evaluation of performance and behavioral properties in the context of DROPS. To accomplish this task, I added sensors to basic system services and applications and obtained events from these sensors using the Ferret monitoring framework [Poh06]. One or more monitors filter this data and then transfer it to an external storage where it will be retained for later evaluation. Furthermore, I use tools to visualize and analyze the obtained data.

Being able to trace events inside an OS kernel as well as within applications running on top of the operating system helps developers finding bugs and detecting performance problems. Therefore, a lot of work has been invested to develop monitoring solutions for a wide range of operating systems. The following sections give an overview of monitoring and existing implementations.

2.2.1 Overview

In the field of system diagnostics several approaches exist for test and evaluation of systems, the most widespread approach being hardware and software debuggers. These tools enable developers to control their applications by setting breakpoints at arbitrary positions within their code. Reaching one of these breakpoints, the application is stopped. The developer may then inspect and change values of variables, so that he is able to find errors within the code or influence the application's further control flow.

Unfortunately, the debugging approach has several disadvantages rendering it useless in the following situations:

- Kopetz [Kop97] defines *real-time systems* as systems, “in which the correctness of their operation is defined not only in terms of functionality (what) but also in terms of timeliness (when).” Debugging timeliness of real-time applications is impossible using breakpoints, because they must not be stopped completely.
- Adding debugging code to an application may lead to situations, where the application runs fine with the debugging code enabled, but produces errors when the debugging code is removed, because the application’s timely behavior is changed.
- Often developers and analysts are not able to predict, which parts of an application are error prone or critical with respect to performance or when these parts of the application will be executed. Working with breakpoints then will require plenty of time and work. It is often not possible to put a human being in front of a long running system for all the time. Critical moments may thus be missed.
- Performance measurements and online evaluation are impossible with software debuggers.

System monitors [Tha00] provide another approach to diagnostics and performance analysis and remedy the disadvantages of debugging systems. Monitors automate the task of collecting data over a long period of time and consist of the following elements:

1. *Sensors* are inserted into applications at arbitrary positions and generate *events*, which may then be read from a monitoring application running in parallel to the monitored system on the same computer or even on a separate one. The sensor approach is basically similar to the breakpoints provided by debugging systems but sensors do not stop the whole application.

   Inserting sensors often leads to an increase of runtime, the so-called *probe effect*. It is caused by more code being executed in an instrumented application, an increased number of cache misses caused by accessing more code and data, and because obtained data needs to be sent to a persistent storage. The monitoring code developer needs to ensure that the *probe effect* is as small as possible, because it can influence application behavior as well as falsify measurements.

   While it is possible to step through an application line-by-line using breakpoints and check for unexpected behavior by checking the complete application state, users of monitoring systems need to know the data they are interested in before running the monitor.

This is because locations for sensor insertion need to be selected (e. g., by hard-coding the sensor into the application's source code or by using dynamic instrumentation as explained in Section 3.4).

2. *Efficient data storage* is needed so that storing events adds as little storage access overhead as possible. Furthermore, events must be available in a timely order for later evaluation. To meet this goal, most monitoring frameworks add a kind of timestamp to each event at the moment it is written to the sensor. [2]

3. *Triggers and Filters* are used to restrict the amount of events that need to be stored. Triggers are used to start and stop measuring at specific points in time or under specific circumstances, so that only relevant data is collected. Filters are applied to the collected events and classify them.

   An example: A system consists of a processor serving requests and a wait queue of length 10 to store incoming requests, which may not be served at the moment. The system administrator wants to check if the queue's length is large enough and therefore inserts a sensor into the system displaying the current number of elements within the queue. If this output is sent every second, the administrator receives 86,400 numbers a day needing to be checked.

   If the administrator knows that the system is under heavy load mainly from 11 a.m. to 3 p.m., she could insert a trigger into her monitor starting the collection of data at 10.50 a.m. and stopping it at 3.10 p.m. She then would receive only 15,600 numbers, which is still a lot. As the administrator wants to know if the queue is long enough, she is not interested in all the data telling her that the queue is long enough, she only needs to know whenever the queue is full. To obtain only these events, she can use a filter storing data only when the number of elements within the queue is equal to the maximum queue length. This will once again decrease the amount of data collected.

4. Monitors may produce an immense quantity of data. This data is used to

   - evaluate specific properties (for instance performance) of the monitored system,
   - obtain behavioral models of a system with respect to the events measured, and
   - detect abnormal system behavior by comparing the events to behavioral models.

   All the previously mentioned tasks are time consuming and error prone when performed manually. Therefore there exist *diagnostic tools* (such as Magpie introduced in Section 2.2.5) supporting the user. The most basic way of support is *visualization*, which presents the collected data in a more handsome way.

[2]Writing an event is sometimes called *committing*.

2.2.2 Monitoring in Linux

As Linux is maintained and improved by a large open-source community, several approaches to monitoring have been developed. Basic facilities come with programs such as *top* and *ps*, which use special files in the `/proc` file-system to gather data about which processes are running at the moment, and how much memory and CPU time they consume. However, they lack the possibility to determine dependencies and interaction between processes and therefore are inapplicable for complex monitoring tasks.

This section introduces three major tracing facilities for Linux in detail: the Linux Trace Toolkit (LTT), Dynamic Probes, and kProbes. I chose the latter facility for instrumenting L4Linux within the scope of this thesis.

The Linux Trace Toolkit

One approach directly focusing on complex interactions and dependencies is the *Linux Trace Toolkit* (LTT) [YD00]. LTT consists of several cooperating services, whose interaction is shown in Figure 2.1.

- A *kernel tracing facility* is added to the Linux kernel and enables its subsystems to produce events by static instrumentation,
- A *tracing kernel module* collects all events and provides them to user-space applications through a character device,
- A *daemon* in user-space reads out the data from the tracing device and stores it for later use, and
- A *visualization tool* can be used to present the collected events to the user.

LTT comes with several predefined sensors and events inside the Linux kernel. Measurements in [YD00] show that the execution overhead caused by LTT sensors is below 2.5%, which the authors consider to be negligible.

Recent work [Des06] resulted in the Next Generation Linux Trace Toolkit (LTTng), which supports more flexible event layouts, 100-nanosecond timing accuracy, and writing events to multiple traces. The project furthermore develops the LTT visualization part under the name LTTV. I did not find any new evaluation of the probe effect implied by newer versions of LTT.

Dynamic Probes

Moore and his group at the IBM Linux Technology Centre developed the *Dynamic Probes* mechanism — DProbes in short — which is described in [Moo01]. Originally, it was planned to

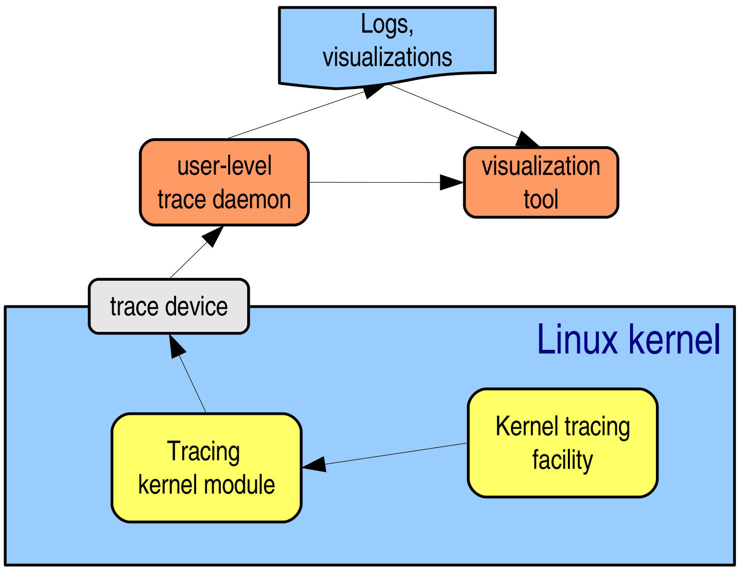

Figure 2.1: Linux Trace Toolkit component layout

be an automated kernel debugger for OS/2 and Linux. DProbes then were enhanced to cooperate with other existing commercial and non-commercial frameworks. One of these frameworks is the previously introduced LTT.

Dynamic Probes introduces the concept of *probe points*. A probe point is an arbitrary location within the source code of the kernel, a kernel module, or a user-space program. When a probe point is hit, a *probe handler* is executed. This handler is implemented in a low-level assembler-like language[3]. This language gives the implementer access to user and kernel memory as well as hardware registers. Probe points and handlers may be inserted into the code dynamically. This is achieved by replacing the original instruction with a trapping instruction — `int3` on the IA32 architecture — leading to a kernel entry where the trap is caught by the probe manager, which then executes the probe handler and returns to the trapping instruction at last.

To evaluate its overhead, DProbes' developers added probe points to certain locations within the OS/2 source code, their number ranging from only a few probes up to having a probe at every kernel API function. The results in [Moo01] show that the overhead is nearly unnoticeable even for large-scale instrumentation, as long as the overall system load is within a normal range. However, it can become noticeable if the machine is running under a workload causing high CPU utilization.

[3] An ANSI-C-to-DProbes compiler is also available.

kProbes

Kernel Probes (kProbes) are an enhancement of DProbes especially focused on the Linux kernel. They are Linux' default means for dynamic instrumentation since the 2.6 series of kernels. An introduction to the use of kernel probes is available with [Kri05] and [Coh05].

kProbes are implemented as Linux kernel modules and inserted into the running kernel. This makes them easier to use than the widespread habit of inserting `printk` statements allover the kernel, which is difficult for complex instrumentation tasks. Instrumentors do not need to recompile the kernel and reboot the machine every time an instrumentation is inserted, there is no instrumentation overhead with instrumentation turned off, and kProbes are a clean way of inserting and removing complex instrumentation code.

There are three variants of kernel probes available:

- *kProbes* are used to instrument an arbitrary address within the kernel. This address is linear within the kernel image and may be determined by either giving a pointer to a publicly exported kernel function or by inspecting the `System.map` file for private functions. The address is not restricted to function addresses, but the instrumentor may also specify a certain offset within the function to trap at a special instruction.

  An instrumentor may specify three function pointers to handlers within a kProbe, one for a pre-handler, one for a post-handler and one for an error handler that is executed if the normal probe execution fails. Conceptually, this is similar to aspect-oriented programming, which is introduced in Section 2.3, however AOP uses static instrumentation.

- As kProbes are related only to linear addresses, it is not easily possible to access the parameters of the currently executed function. If one wants to inspect such values, it is possible to write a *jProbe* [4] trapping at a function entry. The current register and stack context is saved and a handler function specified by the jProbe is called. This handler function needs to have the same signature as the instrumented one and provides easy access to the function parameters.

- Return probes (*kRetProbes*) can be registered to inspect the return value of a function. Upon registration of a return probe, kProbes inserts a probe at the function entry. When this probe is hit, the return address of the function is stored and replaced by a special trampoline instruction. Therefore, the function returns to this trampoline handler and kProbes is able to invoke the return handler before jumping to the real return address.

[4]jump probe

kProbes provide a powerful tool for instrumenting the Linux kernel, although there exist restrictions:

- Probe handlers are called with preemption disabled. Therefore using primitives such as semaphores that rely on preemption being enabled is not possible.
- Probe handlers cannot be specified for inline functions, because their code is inlined by the compiler and there is no easy way to detect occurrence of such functions within the running binary kernel image.
- The kProbes code itself cannot be instrumented. The kProbes subsystem rejects such instrumentation requests during registration of a probe[5].
- If a probed function is called inside a probe handler for the second time, no instrumentation is executed.
- Handling an exception caused by a kProbe and single-stepping the original instruction causes some overhead. I evaluate kProbes' overhead in Section 5.3.1.

Further tool chains

Hiramatsu describes and evaluates *Dynamic Jump Probes* (djProbes) in [Hir05]. This kProbes enhancement inserts a `jmp` instruction instead of an `int3` — the original instruction is overwritten by a simple jump instruction pointing to a handler function. To make sure that also the two-byte `jmp` instruction is written atomically, djProbes uses a kProbe to insert the opcode. djProbes are limited to the functionality of the previously explained jProbes. Their main benefit is that a `jmp` does not lead to an exception inside the kernel, thereby saving runtime overhead.

Eigler et al. describe a probe scripting tool chain called *Systemtap* in [FCE05]. They authors want to develop a script language that is able to generate probe code for arbitrary tracing APIs. Their first implementation aims at generating kProbes code from this script language and is motivated by the difficulties of getting own kProbes implemented correctly.

The Frysk project [fry05] develops an execution analysis tool interfacing all available trace tools to provide users with the information and possibilities they need to achieve their runtime monitoring task. Use cases stated on the project web page range from simple execution tracking up to logging all necessary data upon the crash of a certain application.

Sun's Solaris operating system includes DTrace [CSL04]. Like kProbes, DTrace does not have a probe effect, if tracing is switched off. In addition to kProbes it is able to trace user-space events similar to dynamic probes explained in Section 2.2.2. DTrace probes are written in

[5]This is achieved by placing the kProbes subsystem in an own section.

D, a high-level probe description language providing efficient filtering and event aggregation mechanisms. Cantrill and colleagues [CSL04] do not mention the probe effect for active tracing in their paper.

2.2.3 Monitoring in Windows

Event Tracing for Windows (ETW) [Mic06] is relevant for this thesis, because it inspired existing L4 monitoring frameworks. Furthermore, I use the Magpie tool chain for event visualization. This tool is described in Section 2.2.5 and was primarily designed to be used with traces obtained from ETW.

The Windows 2000 and XP operating systems support tracing of performance events. The operating system supports collection of the following kinds of events:

1. *Throughput* measures the number of requests serviced in a certain period in time and is used to characterize server applications. It can be obtained at several locations within the server and can be used to determine the weakest link in the chain of request processing. Throughput data is reported for disks and network devices.
2. *Queue lengths* can also be used to determine the load that is posed on a system component. It can be the cause of delays in request processing. Windows 2000 reports queue lengths for devices such as processors and disks.
3. *Response time* is the request processing time a client perceives at its side of a client-server environment.

The Windows monitoring infrastructure is provided by the operating system and can be used in two ways:

- *Sampling performance counters* can be read out from the system registry. This has a low overhead, but the measurements may be inaccurate, because sampling is performed periodically. Data within a period of time may be missed.
- ETW can be used to obtain accurate traces from the operating system, because it allows users to trace all operating system actions that are related to a request. However, it poses a larger probe effect on the system.

ETW comes with a number of *event providers* built into the operating system and some of the applications such as the Internet Explorer web browser and the IIS web server. These providers emit events, which then can be consumed by a monitoring application. [Pie04] states that the advantage of ETW being built into the system is that it has a considerably low overhead. However, I could not find any real numbers supporting this statement and did not perform measurements myself, because it was not necessary in the scope of this thesis. Events generated by ETW event providers consist of a common header describing the event with the global UID of its provider and an event type specifying the type of this special event. Furthermore the common header contains information that is obtained for every event, for instance in which process and on which CPU the event occurred. Additionally, each event contains its own private data. The layout of this data differs and monitors need to determine the event UID to interpret the information correctly. To review the path of a request through the system, several events have to be combined. They provide exact information about when the request triggered which action inside the operating system.

The Windows Management Instrumentation (WMI) contains a hierarchy of all events known to the operating system. It may be read out by the monitors and provides information about names and types of the events.

2.2.4 Monitoring in DROPS

As my thesis aims at tracing events in DROPS, it is necessary to investigate, which facilities already exist within this context. These include the Fiasco trace buffer for collecting kernel events and the evolution of a user-level monitoring framework from rt_mon through GRTMon to Ferret.

Tracing events in an L4 environment was first implemented as an addition to the microkernel. Andreas Weigand implemented the Fiasco trace buffer [Wei03], which trace buffer enables users to collect kernel-related events such as context switches, inter-process communication and page faults. Logging of such events can be switched on and off from the Fiasco kernel debugger. Logged events may be inspected using the debugger.

The trace buffer approach to monitoring is relatively coarse-grained because you can only select single event types to be collected. It is not possible to perform event filtering that is more complex than filtering out a single event producer. The lack of such filtering mechanisms leads to a high probe effect. Weigand measured the overhead caused by logging IPC and context switch events in [Wei03]. The overhead is about 100% for a short IPC. Tracing only IPC events still caused an overhead of about 60%. Note that these numbers give the pure system call overhead obtained using a synthetic micro-benchmark. Previously mentioned overheads for LTT specified influence the instrumentation had on the whole system using an application benchmark, therefore LTT overhead was lower. Because of the differences between these types of benchmarking explained in Section 2.5, they are not directly comparable.

The trace buffer also provides the option of writing events from user-space applications. This however has several drawbacks with respect to event tracing allover the system:

1. If monitoring is implemented inside the microkernel, every event needs to be posted using a system call which increases the probe effect by the system call overhead. Recent work on GRTMon [Rie05] and Ferret [Poh06] shows that user-space monitoring can be implemented without kernel support using a shared memory solution and a monitoring server.
2. A single trace buffer prevents us from having different sensors for different events and building sensor hierarchies from existing sensors as explained in Section 3.1.1.

rt_mon [Poh04] was the first user-space monitoring tool for DROPS. It consists of

- A server for managing all available sensors,
- A monitoring library used by event producers and monitors, and
- Monitoring applications reading the data produced.

rt_mon uses shared memory for data transfer between event producers and monitors. This has two major advantages: First, event creation is faster, because data is not copied into another buffer. Second, event generation and reception is typically not a synchronous task. Therefore asynchronous shared-memory protocols are more suitable than synchronous L4 IPC.

rt_mon supports different types of sensors:

- A *scalar* sensor provides a single value. It may be used for instance to implement software performance counters.
- A *list* sensor provides a fixed-sized ring buffer sensors. This is a general-purpose sensor type that can be used for arbitrary use cases.
- *Histograms* are a special sensor type that automatically generate a histogram view of generated data.

Torvald Riegel implemented a generalized version of rt_mon with the name *GRTMon* [Rie05]. GRTMon was written in C++ and only provides one kind of sensor: an event list. To create an event, producers need to *dequeue* a piece of memory from the list sensor. The retrieved event buffer can then be filled with arbitrary data. After this is done, the producer *commits* the event. At this point, a *timestamp* is added to the event, so that the event stream may be ordered later on. Monitors may now read the event from the sensor. Access to the shared-memory sensor is synchronized, so that multiple producers can write to the same sensor.

GRTMon's major contribution to real-time monitoring is guaranteed processing of events. To achieve this goal, the allocated event ring buffers need to be large enough to not wrap around before the oldest event has been read by a monitor. Riegel uses *jitter-constrained streams* [Ham97] to set up a ring buffer that is large enough.

Ferret [Poh06] is a re-implementation of a sensor framework making use of the lessons learned from rt_mon and GRTMon. Ferret sensors are identified by *major* and *minor numbers* in analogy to the identification of Unix devices. Furthermore, Ferret is able to handle several instances of a sensor (e.g., from multiple instances of L4Linux running in parallel) by adding an instance number to each sensor's identifier. Thereby it is able to manage hierarchies of sensors. Unlike GRTMon, Ferret is implemented in pure C, thereby making it easier to use in the L4 environment.

2.2.5 Evaluation and visualization tools

Evaluation and visualization tools help to analyse monitoring data. In this section I introduce Magpie, an event processing tool chain that was developed by Microsoft Research. It was designed to serve as event processor and visualization backend for ETW, which I introduced in Section 2.2.3. I use it for visualization and evaluation purposes throughout my thesis.

Magpie

As explained in Section 2.2.3, events in Windows 2000 and Windows XP are generated from a variety of producers allover the system services and applications running on top of it. Monitors register for a session and specify from which producers they want to obtain events. One or more events forming a *request* will not necessarily appear next to each other in the event stream, but interleaved with events from different requests. It is up to the analyst to figure out which information is needed for his purposes.

Magpie [BIMN03, BDIM04] targets this problem. Requests are extracted from an event stream according to a request description, called *schema*. A schema consists of the exact layout of events belonging to a request and a description of how events and resources like CPUs and threads interact.

Magpie first parses the event stream to extract valid events with respect to the currently applied schemata and drops those that do not match. Afterwards, Magpie combines extracted events to requests by binding them to *timelines*. A timeline is an attribute common to all events that are bound to it.

For example, in Unix the `open` operation returns a file descriptor, subsequent `read` and `write` operations use this descriptor and finally it is invalidated during `close`. All events regarding this file descriptor can be joined to form a file I/O request. This join needs to consider temporal properties, because after closing a file, a new `open` may return the same file descriptor while pointing to a completely different file. Therefore in Magpie's terminology it is called a *temporal join*.

With the preceding example it is clear, that different events influence a request in different ways. Opening a file starts up a new request and creates a new valid file descriptor. Reading and writing does not change it, and closing a file invalidates the descriptor. Furthermore, it might be of interest if another client accesses the same file while the observed process is using it. Such information does not directly belong to our file I/O request, but it may be useful for visualization.

To express the different types of events, a Magpie request schema binds events to a timeline with one of the following four types:

- A START binding shows that this event starts a new timeline section,
- A STOP binding ends a timeline section, and
- A BASIC binding marks this event belonging to the current current timeline section.

The main feature of timeline binding is that events may be bound to multiple timelines. An `open` event for a file I/O request can for instance be bound to the file descriptor it produces as well as to the ID of the thread issuing the operation. Thereby cooperating resources are connected.

An alternative to binding events to timelines is the assignment of *global request IDs*. Instead of binding file I/O events to the file descriptor as in the preceding example, it is also possible to generate a unique identifier at the beginning of each request. However, this would require the UID to be passed through the system and even over system boundaries when tracing requests spanning multiple computers. This approach is less flexible than Magpie's timeline approach.

The Magpie visualizer displays all timelines as horizontal lines. Events are marked at each timeline they are bound to. Requests are visualized starting from a *seed event*. A flood-fill algorithm is used to mark all timelines from this seed point until a STOP binding is reached.

2.3 Code generation

2.3.1 Overview

Code generation can help programmers to develop better code in shorter time by relieving them from time-consuming and repetitive tasks. In this section I will give a short overview of the topic in general and then introduce a code generator that is commonly used for DROPS development — the DROPS IDL Compiler (Dice). The section is concluded by an overview of aspect-oriented programming (AOP) which I consider to be helpful in the context of application instrumentation.

Code generation is used to transform a high-level specification of a problem or its solution into a lower-level implementation. It has two advantages:

1. The generated code usually contains less errors. A compiler translates templates from the source code into templates of the target language. As many developers use the compiler, errors in the translated code are found and removed faster.

2. Compiler designers usually have a better understanding of the target language than an application developer. Therefore compilers can perform optimizations with respect to an application's speed or binary size. Generated code is faster in many cases.

The evolution of software engineering took code generation to higher levels. Modern software technologies generate high-level source code from even more abstract descriptions of the program. *Model-driven architectures* (MDA) [Gro03] use a high-level description or graphical language to design a software system. A code generator then translates it into source code that with few or no modifications can be compiled into machine code. MDA is meant to speed up application development.

Other techniques target the problem, that in some cases generic programming languages are not the easiest or most efficient solution for a certain problem. A *domain-specific language* (DSL) [AvD00] "is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain." DSLs therefore on the one hand speed up development in their special domain and on the other hand the underlying code generator can apply better optimizations to the generated code resulting from domain-specific knowledge.

Aspect-oriented programming as described in Section 2.3.3 is used to manage different concerns of software separately to improve clarity of complex software.

2.3.2 The DROPS IDL Compiler

As explained in Section 2.1, DROPS consists of a set of servers providing services to each other. Fiasco provides IPC mechanisms to enable communication between servers. Communication between a client and a server always invokes the following steps:

1. The client packs its data into a buffer. This step is called *marshaling*.
2. IPC is issued using a Fiasco system call. Is is always *synchronous*. The client is blocked until the server gets ready or a timeout expires. Upon timeout expiration the IPC operation is canceled.
3. The server receives IPC using a Fiasco system call. It is blocked until a client actually sends data, or a timeout expires.
4. The server unpacks the data buffer according to the type of call the client issued. This step is called *unmarshaling*.
5. The server handles the request by executing a user-defined *component function*. Thereafter result data is once again marshaled and sent back to the client using IPC.

This task of writing IPC code can be automated. The DROPS IDL Compiler (Dice) [Aig01] does so by compiling a high-level interface description of a server into real IPC code performing all the steps mentioned above. For reasons of standardization the CORBA Interface Definition Language (IDL) is used for interface description. Most DROPS applications use Dice-generated communication code. As I will show in Section 3.2.2, this fact can be exploited for automated instrumentation by adding such means to Dice.

2.3.3 Aspect-oriented programming

Aspect-oriented programming is an interesting concept for my thesis, because I will need to insert event-generating code into the large amount of existing L4 user-space applications. It was invented by researches at Xerox PARC in the late 1990s as a concept orthogonal to the existing object-oriented approach to software development. The researches concluded that there exist software demands that cannot be expressed with objects and interfaces. An obvious example for this is logging: one can of course have a logger class to collect logged data. But the code creating log output needs to be spread allover the software system, which is complex and hard to maintain.

Concerns that cannot be solved with means of pure object-orientation are called *cross-cutting concerns*. The more of such concerns are found within a project, the more tangled the project's code becomes. Crosscutting concerns can even lead to interleaving of these concerns that may result in errors, as stated by Fiuczynski and colleagues in [FGCW05].

Furthermore, experience from operating systems implementation shows that at the system level a clear separation of components cannot always be established. Coady et al. [CKFS01] state that in low level software often the clear arrangement is sacrificed for the sake of optimization shortcuts. They argue that structured architectures often lead to performance decreases being unacceptable at the operating system level.

The solution to the problem of cross-cutting concerns is to keep these concerns, *aspects*, separate from the rest of the project. At compile time, the aspect code is inserted into the real code at certain *joinpoints*. Joinpoints can either be static, for instance function calls or variable declarations, or dynamic, for instance when an exception is raised.

Along with the aspects developers store information about how to apply these aspects to the joinpoints. The aspect information is called *advice*.

Advices can be applied

- *Before* certain code,
- *After* certain code,
- *Around* certain code, or
- *Instead* of certain code.

The process of applying advices to joinpoints is called *aspect weaving*.

Implementations providing aspect-oriented design are mostly targeted at a certain programming language and consists of a parser to process aspect files with differing grammars for each implementation, and an aspect weaver to insert aspect code into the actual software. Most existing implementations were developed for object-oriented programming languages, for instance AspectJ [Aspd] for Java. It is the reference implementation for so-called AspectX tools, where X can be the language of your choice, for instance Aspect# for the Microsoft .NET Framework and AspectS for Smalltalk.

Evangelists of aspect-oriented programming, like Matthews et al. [MSC+05], state that this approach to software design is also valuable for operating systems development, because the compile time overhead is low compared to the benefits resulting from better code written with AOP.

DROPS applications are mainly written in pure C, because they reuse Linux and FreeBSD device drivers (which in turn are written in C) and because a C environment is easier to implement than a full-featured C++ Standard Template Library, for instance because it does not need to support exceptions. I therefore unsucessfully searched for an AOP implementation supporting pure C code:

- AspectC++ [Aspc] implements AspectX for C/C++. The code generated by its aspect weaver unfortunately is C++ code. Therefore the current version of AspectC++ cannot easily be used for AOP in L4. Future versions might support a set of AOP bindings for pure C code, too.
- AspectC [Aspa] aimed at providing AOP for pure C. Unfortunately, there is no public aspect compiler and weaver available.
- The "Crosscutting C Compiler" [C4], developed at Princeton University is currently able to compile code with aspects woven in, but there is no aspect weaver at the moment. Mark Fiuczynski states in [C406] that this weaver will at earliest be available by the end of summer 2006.
- The Arachne project [DFL+05, Ara] aims at providing AOP for running Linux applications by dynamically patching binary code. It depends on a Linux application running along to the instrumented applications that patches running binaries. Using it in DROPS would require a port of this application which I do not consider to be in scope of this thesis.
- Aspicere [Aspb] is a C code weaver, developed at the University of Gent and used in a project re-extracting forgotten programmer knowledge from legacy C and Cobol applications. Unfortunately, it is part of a complex set of tools depending on each other and I could not get it to work within an acceptable amount of time.

As can be seen from the previous listing, there is currently no AOP implementation available that suits my needs. Therefore I did not use AOP for automated instrumentation.

2.4 Autonomic computing

With my thesis aiming at monitoring DROPS components, I will need to investigate areas in which monitoring can be applied. In addition to performance and behavioral evaluation, autonomic computing will be a major application domain.

Pervasive computing names software components running in parallel while being connected over a network, for instance the internet. These components interact to achieve a certain goal and therefore depend on each other. With the number of connected components rising, this interaction becomes more complex and will finally reach a point where a single programmer cannot cope with the complexity anymore. This situation is described as a *software complexity crisis* for instance in [IBM].

Since 2001 IBM propagate their solution to the complexity crisis under the term *autonomic computing*. The name derives from the autonomous nervous system that keeps a human's vital functions, for instance the heartbeat, running without direct brain activity. In analogy to this system, the authors of [KC03] propose that software systems should automatically monitor themselves and their environment to take the burden of managing complexity from the system designer or administrator and move it to the collaborating software components.

Autonomic computing therefore aims at providing software that possesses certain properties, called *self-* properties*:

- *Self-configuration*: Components should automatically discover hardware and software components in their environment. Each component propagates the services it provides and upon introduction of a new component, this component may discover the available service providers. As well, other components will perceive a new service provider and may adapt their configuration.
- *Self-repair*: Errors in complex software environments take time and manpower to discover. Autonomic components perform this discovery themselves by detecting error situations and then using available patches to correct the error or alert a human individual if there is no automatic solution.
- *Self-optimization*: In a system consisting of tens, hundreds, or thousands of separate software components, it is hard to tweak all the configuration options of a component so that it can provide its service in the most efficient way. Autonomic components will tune their runtime parameters and automatically find the best parameters for a certain situation.
- *Self-protection*: A system of autonomic components will be able to defend itself against failures caused by malicious software or by cascading failures that could not be corrected by self-repairing features.

The vision of autonomic computing is far from being reality and the authors of [KC03] expect it to take years of research until such components exist. To provide features as the self-* properties described in the preceding section, non-intrusive runtime monitoring will have to be a vital part of autonomic components.

2.5 Benchmarking

One goal of my thesis is to detect performance-critical paths inside DROPS components. Benchmarks are a common way of measuring performance and are therefore interesting in the context of my work.

Balsa [Bal97] defines a *benchmark* as a "documented procedure that will measure the time needed by a computer system to execute a well-defined computing task. It is assumed that this time is related to the performance of the computer system and that somehow the same procedure can be applied to other systems, so that comparisons can be made between different hardware/software configurations."

Benchmark results may be given with an arbitrary dimension, for instance "iterations per second." Furthermore, it is possible to compare the obtained values to a reference implementation by normalizing the obtained results to the ones measured from the reference component. The normalized results are called *indices*.

Balsa defines the *resolution* of a benchmark to be the minimum time interval that can be measured on the evaluated system. Furthermore, he defines the *precision* of a benchmark, which gives a measure about how much the obtained results vary. Variation may result from cache effects and other tasks being scheduled during a benchmark run.

Balsa distinguishes two types of benchmarks:

1. *Synthetic benchmarks* are designed to measure the best performance of a certain subsystem. Many benchmarks exist to evaluate performance of certain Linux subsystems. Tim Bray's Bonnie [Bra] is a synthetic benchmark for Unix file systems. David Niemi's Unixbench [Nie99] is a benchmark suite covering several parts of Linux/Unix' subsystems such as system calls, file systems, and hardware.

2. *Application benchmarks* measure the execution time of a commonly used application. For Linux systems, kernel compilation is often used as an application benchmark, because it tests several components of the system: file system operations, the compiler, the C library, and the hardware used.

One needs to be careful when comparing the results of synthetic and application benchmarks, because they are usually used to describe different things. A synthetic benchmark describes the behavior of one component with respect to a certain load. An application benchmark tests the behavior of a whole system without focusing a single part of it.

3 Design

The hands of time have brought you here
To make a change, to break down the walls.
(Hammerfall - Templars of Steel)

3.1 Defining goals

The goal of my thesis is to establish means for tracing requests in DROPS. With such means, it will be possible to

- Analyze and optimize existing client-server protocols,
- Compare multiple existing implementations of a protocol, and
- Determine performance-critical paths inside the system.

In this chapter I will explain design decisions that I made to achieve these goals. At first, it is necessary to define basic requirements for the tracing architecture. Thereafter I will inspect the points within DROPS that I need to take care of when tracing events. I will inspect and evaluate existing monitoring solutions, as well as discuss layout of events and use of instrumentation techniques. Furthermore, I will present thoughts about instrumentation of two special cases: the L4 Environment (L4Env) and L4Linux. Finally, I will inspect ways of storing, transferring, and post-processing the obtained data.

3.1.1 Basic requirements

The following list contains basic requirements for instrumentation and monitoring that influenced my design decisions.

- **Small probe effect:** Instrumenting applications leads to a certain probe effect. It needs to be as small as possible, because it influences the behavior of instrumented applications, which is especially critical when monitoring real-time systems.

- **Sensor hierarchies:** In a system with many event producers, a monitor needs to use efficient filtering techniques to keep the amount of processed data low. Having multiple sensors instead of one global sensor is a natural way of filtering. A monitor can choose to obtain and evaluate data from a subset of sensors.

  A monitor may aggregate information from several low-level sensors into higher-level events, which then are published through a new sensor. Thereby sensor hierarchies are established.

- **Automated instrumentation:** Because of the large amount of existing DROPS applications, manual instrumentation is costly. I will find ways of automating the task of adding sensors to applications.

- **Minimize the amount of data:** Data that is not needed to achieve a monitoring goal needs to be dropped as soon as possible. This reduces the amount of processed data during evaluation and therefore speeds up this process. Online monitors especially benefit from short processing times.

- **Flexibility:** While this thesis focuses on communication tracing, the means established for monitoring will be kept as flexible as possible to also fit future needs.

Achieving the previously defined goals and considering the basic requirements results in modifications to be made to DROPS and facilities to be created within the system. Figure 3.1.1 defines the points within DROPS that I need to take care of. I will discuss each point in the following list.

1. It is necessary to trace events from Fiasco, because some events such as context switches are only available from the kernel. [1]

   A kernel tracing facility exists with the Fiasco trace buffer. This buffer is made available to Ferret clients as a special sensor.

2. The basic services of the L4 Environment will be instrumented. This includes the existing servers as well as widely-used L4Env libraries such as the thread and semaphore libraries. Some instrumentation in this layer will be difficult, because those components that are used by the tracing framework itself (e. g., the `dm_phys` dataspace manager, and the global `names` service) need special attention.

3. A basic requirement is tracing communication between running DROPS applications. Part of this information (communication partners, timestamp, type of IPC, timeouts) is

[1] Unless we use a user-level scheduler, which is not available in DROPS.

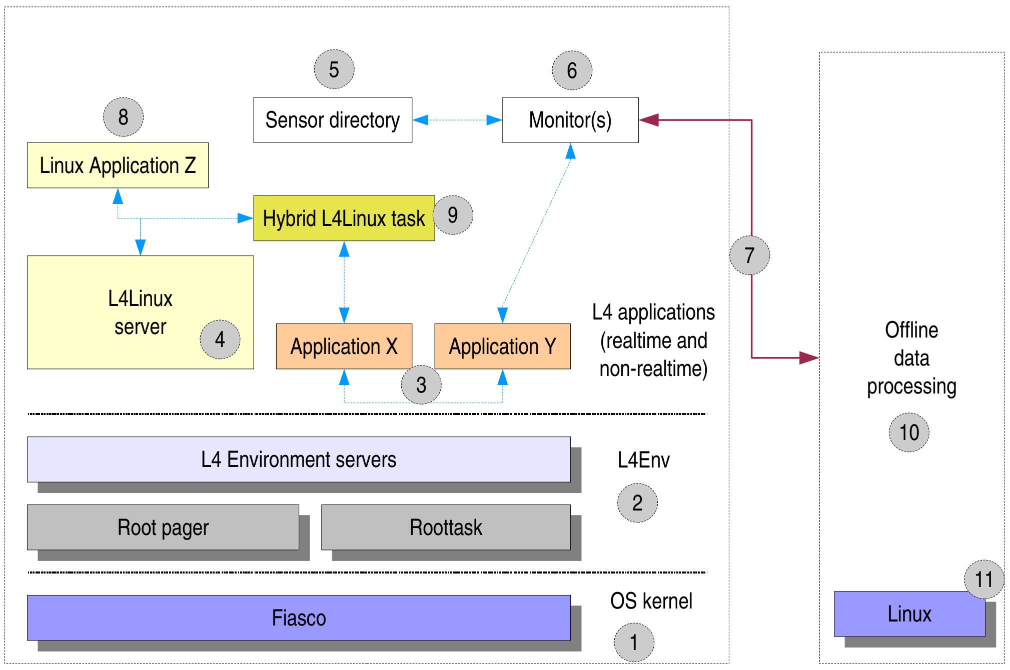

Figure 3.1: Instrumentation points within DROPS

available from the trace buffer. Additional information such as the message sizes can be extracted by instrumenting the IPC code generated by Dice, which was introduced in Section 2.3.2. Most DROPS developers use Dice to compile their IPC interface into code, therefore instrumenting Dice covers a wide range of L4 applications.

4. L4Linux is one of the most complex workloads for DROPS. Therefore I want to be able to trace events within the L4Linux kernel. Solutions for this exist for native Linux and it seems promising to adapt one of these implementations for L4Linux.

5. A system-wide sensor directory is needed, where clients can register their sensors and monitors can request sensor data. The Ferret monitoring framework comes with an implementation that is suitable for my purposes.

6. Monitors are needed to collect and process sensor data. I will have a look at what kinds of monitors are necessary and how these can be used.

7. When data is transferred from the L4 system to another computer for offline evaluation, means for efficient data transfer are needed.

8. Linux applications running in L4Linux shall be traced, because they are a part of the running system, even though they are not aware of running along DROPS applications. Connecting Linux programs to default L4 tracing facilities is possible, but in most cases I'd prefer not to make these applications L4Linux hybrid tasks just for monitoring. Therefore I will inspect other available means.

9. Hybrid L4Linux tasks should benefit from L4 tracing and Linux tracing.

10. Both online as well as offline evaluation need to be performed. *Online monitoring* can be used to detect and verify behavioral and runtime properties, but it is limited to a subset of available evaluation methods, because there is only a limited amount of execution time available. Furthermore online monitoring is a vital part of autonomic computing, which was introduced in Section 2.4.

    Offline monitoring can use more powerful and time-consuming methods of evaluation. It is used to obtain behavioral models of a system and compare these models to execution traces from abnormal situations.

11. Data obtained from instrumenting L4Linux shall be compared to data that can be collected by instrumenting the same points in a native Linux operating system. This may prove to be helpful for determining performance differences between both systems and for tuning L4Linux.

In the remainder of this chapter I will discuss the use and layout of events in the context of this thesis. I will explain and how existing applications can be instrumented for event generation. Furthermore, I will investigate efficient ways of instrumentation, so that the probe effect is kept low.

3.2 Event retrieval

3.2.1 Review of existing facilities

Before designing an event tracing system, it is necessary to investigate already existing means of tracing communication in DROPS. As introduced in Section 2.2.4, Andreas Weigand has implemented the Fiasco trace buffer. With the help of the trace buffer, user applications can trace kernel events. Measurements show that using the trace buffer for low-overhead tracing is insufficient. The amount of produced data is extremely high — a simple startup of DROPS and some basic servers produced more than 10,000 IPC events (about 700 kB of data) within a few seconds. For efficient processing, it is necessary to filter data early. There are basic means of filtering events inside the trace buffer, such as restricting the collected events to the ones originating from one special thread. However, these means are limited, because the trace buffer is part of a kernel that is targeted at real-time systems and it is not possible to use complex filter mechanisms in a bounded amount of time. The trace buffer's filtering is too inflexible for the need of a low-overhead event tracing system. The design of an own tracing facility should circumvent the drawbacks of the trace buffer approach and provide improvements such as

1. Low monitoring overhead,
2. Filtering of monitoring data, and
3. Use of arbitrary user-defined event types.

Generalized monitoring frameworks for DROPS exist with GRTMon and Ferret. These frameworks enable users to define whatever type of events they want to collect. They provide low monitoring overhead and even real-time guarantees for sensor producers and monitors as described in 2.2.4 and [Rie05].

Ferret provides means of data filtering, by using different sensor types which I introduced in Section 2.2.4. Within this thesis I will use this framework for monitoring.

Because of the high amount of data produced by the trace buffer and its lack of filtering mechanisms, I will not consider using the Fiasco trace buffer for general tracing. However, I will investigate its use in special cases, for instance when it provides data that cannot be obtained in a different way, such as scheduling information.

3.2.2 Communication tracing

As a next step I need to determine how communication inside DROPS can be traced. Manual instrumentation of all applications is not an option, because it requires a lot of work, whereas future changes to applications potentially render the current instrumentation useless.

Five solutions remain for consideration:

1. Fiasco developers recently introduced an *alien* state for L4 threads. If a thread is marked alien, every system call it performs raises an exception and an alien handler is called. This handler can allow or refuse the thread from performing this system call. This can be considered a mechanism for IPC tracing: all threads that shall be monitored are marked alien and the monitor becomes their alien handler. After storing the necessary data from an IPC system call, the monitor will allow this call. This solution's advantage is that it is transparent to the system and no source code modifications are necessary for IPC tracing. Unfortunately, alien handling is extremely slow, because it translates one IPC system call into four (exception IPC to the alien handler + exception reply + original IPC + result IPC to exception handler). Therefore it cannot be used for low-overhead tracing.

2. Most DROPS applications use the `l4sys` library for performing system calls. This library provides C wrappers for all calls, which can be instrumented to trace IPC system calls. Applications to be traced then need to be relinked against the new library version, no further changes are required. This kind of tracing enables users to perform event filtering at compile time by providing only events about instrumented applications. This is an improvement in comparison to Fiasco trace buffer events, but it is still too coarse-grained.

3. Jan Stöss [Stö05] uses dynamic instrumentation in his implementation of a user-level scheduler for L4Ka::Pistachio. To make the right scheduling decisions, he needs information from the OS kernel and as scheduling decisions need to be made often, this information must be retrieved with a low overhead. Stöss proposes to dynamically insert instrumentation at runtime, whenever it is needed. For this, he adjusted a compiler to insert a certain number of `noop` instructions at the instrumentation points. These instructions are replaced by the real instrumentation code, when instrumentation is switched on. This dynamic approach has the advantage that practically no monitoring overhead occurs with monitoring switched off and only those points suffer a probe effect that are really being traced at the moment. The drawback of Stöss' solution is the need for a special compiler that inserts `noop`s.

4. Aspect-oriented programming can be used to instrument application source code. Developers can write aspects fitting their instrumentation needs. In comparison to the trace buffer, this option is not restricted to special kernel events. As opposed to `l4sys` instrumentation this option is not restricted to system calls. Furthermore, developers can include high-level information into their application-specific IPC events, that is not available at the system call level. An example for such knowledge are the parameter values of an IPC call, which are easily available before marshaling, but not at the moment `ipc_call()` is executed. Using AOP therefore provides the highest degree of freedom for instrumentation.

   Unfortunately, as explained in Section 2.3.3, there is currently no AOP tool for pure C available and implementing one is out of the scope of this thesis. Therefore an AOP solution for automated instrumentation needs to be postponed until such an implementation exists. AspectC++ developers promise to have this in Q2/2006 [Asp06], C4 developers plan to have an implementation by the end of the same year [C406].

5. As explained in Section 2.3.2, most DROPS applications use the DROPS IDL Compiler (Dice) to generate their IPC code. The IPC code generated by Dice can be instrumented by extending Dice. Ronald Aigner [Aig] recently enabled Dice to use trace plugins. These plugins are user-provided libraries that can be used to generate tracing code at certain points within the Dice-generated code.

The advantage of tracing Dice code is that it provides the same degree of freedom with respect to event types and high-level information as an AOP solution. However, it is restricted to tracing of Dice-generated IPC code. In this thesis I will introduce an implementation of a Dice tracing plugin. For tracing applications that do not use Dice, such as L4Linux, other means of tracing need to be found.

3.2.3 Event layout

Just like in the Magpie request extraction tool introduced in Section 2.2.5, a request in the scope of this thesis is a structured set of events obtained from locations within the system. In this section I will discuss how these events are laid out and present rules that instrumentation should obey in order to reduce the probe effect.

The first constraints upon event layout derive from the Ferret monitoring framework and the Magpie event processor. Ferret classifies events with respect to a common event header consisting of at least:

- *Major* and *minor numbers* are used to identify a sensor and the events produced by this sensor globally. This concept is similar to the concept of major and minor numbers identifying devices in Linux.
- The monitoring framework adds an *instance* number to each event and each sensor so that several instances of a sensor may be distinguished. This may be the case if there are multiple instances of L4Linux running in parallel and monitors want to obtain events from all their kernel sensors.
- In order to be compatible with future SMP versions of L4Linux and DROPS, the framework adds a *CPU* field to each event.
- Each event is furthermore augmented with a *timestamp* so that a monitor can receive these events in a timely order.

The previously described fields define the *common event header*. Additionally, most events have a *data area*, whose structure is specific to the event type defined by the major and minor numbers. For the `dice_trace`-generated instrumentation, this data area differs in client-side and server-side events. Figure 3.2 shows the layout of both event types. Each event contains a producer ID as well as start and stop timestamps, which tell when the IPC call was sent and the answer was received by the client and vice versa for the server. Server-side events additionally contain timestamps storing timing information about the component function. At the client side additional data describes the size of sent and received data and the opcode with which the server is called.

Timestamp Major number Minor number Instance number CPU ID		*Event head*
Producer ID	Producer ID	*Additional data*
Start timestamp	Start timestamp	
Stop timestamp	Stop timestamp	
Send size	Component start	
Receive size		
Dice opcode	Component stop	
-		
Client-side	Server-side	

Figure 3.2: Event layout for `dice_trace` events

3.2.4 Instrumentation rules

In addition to the design of an event layout, I need to determine where instrumentation code is placed. During design and implementation of the `dice_trace` plugin as well as for other instrumentation purposes, the following rules proved to be helpful.

1. **Event packing:** GCC enables programmers to mark data structures as __`packed`__, which means that the single components are not aligned in memory. This makes access to single fields of the structure a bit slower, because on IA32 accessing addresses aligned to the size of a data type is faster. Compilers may therefore choose to align members of a data structure to a multiple of their size and leave padding bytes in between. This is difficult for two reasons:

   a) We do not always know if and how the compiler performs such an optimization. However, later post-processing needs such knowledge to unpack data correctly. Using the __`packed`__ attribute ensures a deterministic data layout.

   b) The padding introduced by the compiler contains no data at all and increases the size of an event by a few bytes. With typical event counts reaching tens or hundreds of thousands, saving this padding results in a perceivable reduction of consumed memory space.

2. **Use of large events:** When measuring calls to functions, it seems tempting to have one event before the function call and one event after return from the function. However, there are two reasons against this:

   a) The two single events are of no use at all when retrieved separately. Only a combination of a before- and an after-event make sense for later evaluation.[2]

   b) Calls to the monitoring framework cost time. Therefore it is better to pack all data into one event and pay the overhead only once instead of twice. This argument can further be extended, so that N events are packed into one monitoring event, thereby reducing the monitoring overhead to one dequeue and one commit for every N events.

   If event processing needs smaller events, for instance if a visualization later on needs a `before_call` and an `after_call` event, an event postprocessor can split large events into smaller ones.

   ```
   BAD:                             GOOD:
   e = get_event_buffer();          e = get_event_buffer();
   e.ts = rdtsc();                  e.start = rdtsc();
   e.commit();
   ret = func();                    ret = func();
   e2 = get_event_buffer();
   e2.ts = rdtsc();                 e.stop = rdtsc();
   e2.return_val = ret;             e.return_val = ret;
   e2.commit();                     e.commit();
   ```

   Note that this technique is only applicable if the complete function call is the event we are interested in. By putting a lot of data into one event, the monitoring overhead is reduced, but real-time capabilities such as online debugging are lost, because in this case it is necessary to have information available as soon as possible.

3. **Keep out monitoring overhead:** Instrumenting code falsifies performance measurements, because the overhead to retrieve a new event from Ferret, inserting data, and commiting the event to the sensor is added.

   An example for such falsification is the Dice instrumentation. At the server side a new event needs to be dequeued from Ferret every time a client issues a request. Therefore the measured server-side execution time is incorrect. Monitors and post-processors can remove the monitoring overhead from the measured data if the events contain enough

[2] This is a more philosophical reason.

information. To achieve this, I added not only a start and stop timestamp to each Dice server event, but also provided a start_dequeue timestamp. The timestamps are obtained as follows:

```
msg = ipc_reply_and_wait(old_mesg);

/* Instrumentation overhead */
rdtsc(start_dequeue);
event = ferret_dequeue();
fill_event(event);

/* Real processing */
rdtsc(start);
dice_handling(msg);
rdtsc(stop);

/* Instrumentation overhead */
ferret_commit(event);
```

This solution adds runtime overhead for taking another timestamp and storing it into the event. Measurements show that this overhead is approximately 50 CPU cycles on the test machine introduced in Section 5. Furthermore, this solution adds another 8 bytes to the event size, which is not always negligible.

4. **Hide behind latency:** As instrumentation has its costs, it is good to find instrumentation locations that are not performance-critical. Engler and colleagues [ECC01] developed a tool that uses static analysis of source code to find bugs in it. They introduce the concept of *beliefs* that can be derived from the source code. For instance, if a programmer dereferences a pointer p in her code, we can deduce that she believes that this pointer is never `NULL` at this point of execution. Static analysis can now detect all paths leading to this location and produce an error message if one or more of these paths may be executed without ensuring that the belief `p != NULL` is true.

   Instrumentation can also benefit from beliefs in the following way: At locations where an application performs an `ipc_send` with an infinite timeout, the programmer expects this operation to potentially last forever. This means that the instrumented application will not have problems if this operation takes more cycles caused by the monitoring overhead. Placing instrumentation code at such locations is less intrusive than at a point where `ipc_send` is called with zero timeout, where the programmer expects the system call to return as soon as possible.

3.2.5 Sensors and monitors

Varying needs for request tracking devise different requirements for the layout of sensors in a system as well as for the monitors. On the one hand, performance needs to be evaluated. Under these circumstances analysts only want to obtain as few data as possible, because this keeps the probe effect low. The higher the probe effect, the more it falsifies the obtained performance results. On the other hand, for applications such as protocol analysis we want to collect as much information as possible. The probe effect is less important for protocol analysis, unless the protocols themselves contain a timing effect such as IPC timeouts, where a low probe effect again becomes worthwhile.

Furthermore, we can distinguish global common sensors, where many producers insert events versus local specialized sensors that are used by a special application for its own purposes.

The same classification can be applied to monitors. When collecting lots of data that is simply handed over for offline evaluation, a global monitor merging all events from the sensors into one buffer its our needs. For online monitoring the monitor needs to perform its tasks fast. Therefore it should not be burdened with the task of filtering the required events from a large stream of unwanted ones. Table 3.1 shows the possible combinations of sensor and monitor types, as well as their implications.

	Common monitor	**Specialized monitor**
Common Sensor	suited for collecting data for offline evaluation	difficult: may lead to high load caused by filtering unnecessary events
Specialized Sensor	may be used for offline evaluation, bad performance for online monitoring	best solution for online monitoring

Table 3.1: Monitor-sensor combinations

3.3 Instrumenting the L4 Environment

The L4Env introduced in Section 2.1 is a set of servers and libraries that is used by many applications running on L4. Because it provides basic services that are used by most DROPS applications, it is interesting for instrumentation. To support self-monitoring capabilities for these programs, I propose to add one or more common sensors to the L4Env, so that applications can monitor their behavior at runtime. Such sensors can also be used for offline evaluation of application and L4Env performance. In the following section, I will discuss which parts of the L4Env services can be instrumented and for which purposes this instrumentation may be used.

The *L4 thread library* can be instrumented to obtain data about thread creation and shutdown as well as access to thread-local memory. The *region mapper library* can be instrumented to generate statistics about page faults. This information is local to an application and may be used to implement a local page replacement strategy. This approach is similar to the concept of application-level hints presented in [Döb05a]. I propose not to rely on page fault events from the Fiasco trace buffer here, because of its lack of filtering mechanisms.

Instrumentation in the *semaphore library* can be used for performance evaluation as well as for answering the following questions:

- It is interesting to determine the contention of semaphores. Recent discussion with Alexander Böttcher [Böt] showed that it interesting to know whether the semaphore library needs to be optimized for the contention or non-contention case.
- I can evaluate the length of critical sections. For the implementation of delayed preemption [SAG04] it is interesting to know exactly how short a short critical section is — this may then be used as a limit for the delay of preemption.

  Furthermore, this information can be used for application profiling, because if many threads are waiting while one is inside a very long critical section, this might be a design flaw leading to bad throughput.

Instrumentation of the *semaphore library* is not possible with only trace buffer events, because a call to `semaphore_down` does not necessarily lead to a perceivable IPC event. Manual instrumentation with a Ferret histogram sensor can be used to evaluate semaphore contention. Only the `semaphore_down()` function needs to be modified for that.

Evaluating the time spent in a critical section is more difficult because events come from two different functions: `semaphore_down()` and `semaphore_up()`. Further difficulties arise, because semaphores are used in at least three different manners:

1. *Mutexes* ensure that only one thread enters a critical section. For this case instrumentation is easy. We can simply add a timestamp field to the `l4_semaphore_t` struct and insert the start time of the critical section during `semaphore_down()`. When leaving the critical section, another timestamp is taken and the difference can be calculated.
2. *Counting semaphores* allow more than one threads to be inside a critical section. A single timestamp field in the semaphore struct is therefore not enough, but a list of thread-timestamp mappings is needed.
3. Semaphores are used to solve *producer-consumer problems*, where no critical section is involved at all. As in this case a thread calling `semaphore_down()` will never

call `semaphore_up()` again, the critical section time is not of interest. Instead, the time spent waiting for a semaphore can be measured by instrumenting `semaphore_-down()`.

The only generic solution covering all the previously introduced cases is to have a list sensor for semaphores, emitting

- producer thread ID, semaphore ID, wait queue length, and start time t_s for `semaphore_down()`, and
- producer thread ID and semaphore ID for `semaphore_up()`.

In addition, timestamps t_{down} and t_{up} are added to each event by the monitoring framework. We can therefore calculate:

$$\begin{aligned} t_{critical\_section} &= t_{up} - t_{down} \\ t_{wait} &= t_{down} - t_s \end{aligned}$$

Offline evaluation can furthermore detect if a semaphore is for a consumer-producer problem or for a critical section by having a look at the thread ID of the producers. Critical sections run in the same thread, whereas producers and consumers are different.

As the Ferret monitoring framework is a DROPS application, it also makes use of services provided by the L4Env. It uses the `names` server to register its name and the `dm_phys` server to create sensor dataspaces. Instrumenting these services is therefore difficult. `names` cannot register a sensor at Ferret, because this requires Ferret to be known at `names`. `dm_phys` cannot register a sensor, because Ferret then will call `dm_phys` in turn to create a sensor dataspace. This results in a deadlock, because the dataspace manager is single-threaded and the worker thread is still waiting for a reply from Ferret.

Possible solutions for these problems are

- Modification of Fiasco to map a fixed event buffer to a fixed location in memory. Clients can then rely on this buffer and write their events into it. Ferret can make this buffer available to monitors in the same way it does for the Fiasco trace buffer.
- Specialized solutions for the small amount of applications that suffer from the bootstrapping problem can be found.

Martin Pohlack [Poh] solved the `names` problem by modifying the server. `names` waits until Ferret registers its name and then registers its sensor at Ferret. Events from earlier points in time are stored locally and can be committed when the sensor has been established.

A similar solution can be used to instrument `dm_phys`. When registering a sensor at Ferret, the dataspace manager can hand in a proper dataspace for its sensor, so that Ferret does not need to contact `dm_phys` again.

3.4 Instrumenting L4Linux

L4Linux is of special interest for event tracing, because it is an elaborate application on top of DROPS. Incorporating event tracing into this complex environment is a major focus of this thesis, because it will help users and developers to understand their applications' behavior, detect behavioral differences to original Linux and its applications, and detect performance-critical paths.

As L4Linux is an adaption of the Linux operating system, it seems natural to use an existing Linux tracing solution for L4Linux. kProbes and the Linux Trace Toolkit are available in this context and have been introduced in Section 2.2.2.

The LTT is restricted to a fixed set of instrumentation points inside the Linux kernel. However, newer versions like LTTng also support user-defined events and instrumentation of user-space applications. kProbes support instrumentation of arbitrary locations inside the Linux kernel. User-level instrumentation is not yet available.

For my thesis, I chose to use kProbes for instrumentation inside the Linux kernel, because of its flexibility. Upon startup the L4Linux server registers a *kernel event sensor* with Ferret. This list sensor can then be used by kProbes to commit their events.

As I show in Section 5.3.1, due to different implementations, kProbes lead to a higher overhead in L4Linux than in native Linux and can therefore not directly be used for performance evaluation. However, the overhead is constant for L4Linux and native Linux. Therefore I can use kProbes for *differential analysis* and comparisons between both versions of Linux. To do so, I register kProbes at certain locations inside the Linux kernel and measure the relative amount of time Linux spends inside a piece of code in comparison to the total amount of time spent in all pieces. This can also be done for L4Linux and the results can be used to detect behavioral differences.

A minor requirement for differential analysis is that both Linux and L4Linux execute the same kProbe module. L4Linux kProbes use the internal Ferret kernel sensor to commit their events. Native Linux does not possess such a sensor. I chose to simulate the Ferret interface in native Linux using a kernel module. This module is introduced in Section 4.3.2.

Tracing user-space applications requires different means. For Linux there is a range of instrumentation applications available. ATOM [SE94] and the Flexible Instrumentation Toolkit (FIT) [BCS+04] use *static binary instrumentation*, which means that they instrument an application binary before running it. Pin [LCM+05] and Valgrind [NS03] use *dynamic binary instrumentation* by translating the application's binary code into a meta language that is then executed inside a virtual machine (VM). The VM supports injection of instrumentation code at runtime.

Binary instrumentation requires no source code access and therefore can handle every Linux binary. I tried to use static instrumentation with FIT, but this tool was developed for Intel CPUs and did not run on my test computer having an AMD Duron CPU. Dynamic instrumentation tools proved to be extremely slow, therefore I discarded this option.

The alternative to binary instrumentation is *source code instrumentation*, which is only applicable for open source applications. The easiest way to instrument applications running on top of L4Linux is to instrument them with Ferret sensors directly. This is possible, because L4Linux supports *hybrid tasks* that perform Linux system calls as well as Fiasco system calls. Unfortunately, registering a Ferret sensor returns a dataspace to the registering application and if this is a Linux application, it does not know where to map this dataspace. There are three possible solutions for this problem:

- The application can use `mmap` to map an anonymous memory area of the same size as the sensor. It is then safe to over-map this area with the dataspace retrieved from Ferret.
- The L4Linux server can be modified to register a Ferret user-space sensor at startup. The sensor can then be mapped to a reserverd address range. As L4Linux is the pager of all L4Linux applications, it will receive page faults whenever such applications access an address inside this address range and may map the user space sensor to applications if they trigger such faults. This approach is restricted to one sensor for L4Linux user applications.
- Linux applications can request mapping of a sensor area using a system call to a special Ferret kernel module in L4Linux. This has the advantage, that the kernel module can be more elaborate and map different sensors to one or more applications.

Martin Pohlack [Poh] implemented the second option in L4Linux. Source code instrumentation of Linux applications therefore consists of triggering a page fault to map the sensor into an application's address space and then committing events to it the same way as if it was a normal Ferret list sensor.

3.5 Storing data

With the design presented in the preceding sections, producers can generate and monitors can obtain events from allover DROPS. Online monitors can directly evaluate this data. For offline monitoring data needs to be stored permanently. Two alternatives seem possible:

1. Data can be transferred to another computer through a network. The amount of stored data is limited by the speed and bandwidth of the network connection, but this solution is flexible, because data can be sent to any computer on the internet.

2. Raw data can be stored on a hard disk and this disk may then be read out for offline evaluation. The amount of stored data here is only limited by the size and bandwidth of the hard disk. However, it is less flexible, because offline evaluation needs to physically access the disk.

Both alternatives are possible to implement. Network devices as well as hard disks may be accessed by DROPS applications through a network and a block device server. Sending data to these servers and to the hardware induces an overhead to the system. If possible, monitors should therefore perform measurements at first, temporarily save data in memory during monitoring and write it to permanent storage right after monitoring.

For my design I settled upon the first alternative for flexibility reasons. I ported a TCP/IP stack to the ORe network switch [Döb05b] as described in Section 4.2.

3.6 Data processing

As introduced in Section 2.2.5, I used the Magpie event processor for offline evaluation of event data. With its capabilities, it is able to parse and visualize request traces from an event stream. To be useful in the DROPS environment, an L4 request schema needs to be implemented.

Because Magpie schemata are written in Python, they can also be used for event processing not directly targeted at visualization. In Section 4.5, I will introduce several Magpie modules that perform offline evaluation tasks, such as generating statistics about an event stream, and producing output for different visualization backends.

4 Implementation

Welcome to the jungle,
We take it day by day.
(Guns'n'Roses - Welcome to the jungle)

In this chapter I will describe the implementation of facilities that were needed for this thesis. I will start with a description of my tracing plugin for Dice, which is used for automatic instrumentation of Dice-generated IPC code. Thereafter I will introduce my port of a TCP/IP stack to the ORe network switch. This section is followed by a description of implementation issues related to the instrumentation of Linux and L4Linux. I conclude this chapter by describing a sample application: a self-healing web server using on-line monitoring to improve its availability.

4.1 A tracing plugin for Dice

Overview

To gather IPC events from as many applications as possible one needs to insert event-generating code into these programs. The instrumentation code looks similar in many cases, so this task can easily be solved by automated code insertion. Aspect-oriented programming (AOP) — as introduced in Section 2.3 — can be a solution for automating this instrumentation. A large part of DROPS applications to be instrumented is written in C. Unfortunately, at the time of this writing no AOP implementation for C is available, therefore I needed to use other means for large-scale instrumentation.

As introduced in Section 2.3.2, most DROPS applications use the DROPS IDL Compiler (Dice) for generation of communication code from a simple interface definition. I extended Dice with a tracing plugin that generates instrumentation code.

A tracing plugin is compiled as a C++ library and must provide two functions:

- `void dice_tracing_init(int argc, char **argv)` This function is called during Dice initialization. It receives Dice's command line arguments and can be used to pass additional arguments to the plugin. My `dice_trace` implementation handles the following arguments:
  - `performance`: generate sensor code with main focus on performance, which means to collect as few data as possible to generate as few monitoring overhead as possible.
  - `cflow`: generate code which collects as many data as possible to support event visualization and later generation of statistics.
  - `eventname`, `sensorname`: set the sensor and event identifiers to be used by the generated code.
- `CBETrace *dice_tracing_new_class(void)` This function is called whenever a new trace class needs to be generated. A trace plugin will return one of its own tracing classes here, all of which are derived from Dice's tracing class `CBETrace`.

  Depending on the command line arguments `dice_trace` returns a performance trace class or a control flow trace class from this function.

Dice's basic tracing class defines a number of hooks, which are implemented by the plugin subclasses. Table 4.1 shows the hooks and describes the points during the code generation process at which they are called.

Instrumentation code

Instrumentation code needs to handle two tasks: setting up a Ferret sensor and producing events through this sensor. Sensor setup can be implemented in two ways:

1. Add a constructor function to setup a sensor for each instrumented application. Constructor functions are currently used by the semaphore and region manager libraries to startup their specific worker threads before any other application thread.
2. Setup sensors lazily. This requires insertion of sensor setup code into every Dice-function on the client side. This setup code checks, whether the sensor has already been set up and in the negative case performs sensor registration at the Ferret sensor directory.

Hook	**Action**
`DefaultIncludes()`	Called when the default include files are written to an implementation or header file.
`InitServer()`	Called before the Dice server initialization code is written.
`VariableDeclaration()`	Called at the beginning of each function to insert local variable declarations.
`BeforeCall()`	Called at client side before the `ipc_call()` code is written.
`AfterCall()`	Called at client side after the `ipc_call()` code was written.
`BeforeDispatch()`	Called at server side before the current request is handed over to the `dispatch()` function.
`AfterDispatch()`	Called at server side after return from the `dispatch()` function.
`BeforeReplyOnly()`	Special case at server side. Functions with the `allow_reply_only` attribute set have a special `reply()` function to be called for delayed response This hook is called at the beginning of this function.
`AfterReplyOnly()`	Called at server side at the end of a `reply()` function.
`BeforeReplyWait()`	Called at server side at the beginning of a general `reply_and_wait()` function.
`AfterReplyWait()`	Called at server side before return from a general `reply_and_wait()` function.
`BeforeComponent()`	Called at server side before calling the user-defined component function.
`AfterComponent()`	Called at server side after return from the component function.
`BeforeUnmarshal()`	Called before unmarshaling of incoming parameters at server and unmarshaling of return parameters at the client.
`AfterUnmarshal()`	Called after unmarshaling of incoming parameters at server and unmarshaling of return parameters at the client.
`BeforeMarshal()`	Called before marshaling of return parameters at server and marshaling of parameters at the client.
`AfterMarshal()`	Called after marshaling of return parameters at server and of the parameters at the client.

Figure 4.1: Hook functions that need to be implemented by Dice tracing plugins

For my implementation of `dice_trace` I chose the first option, because it results in less code being generated and reduces to probe effect. With the latter alternative, event producers have to check for a valid sensor before each access to the sensor, although only the first check will fail.

After sensors have been set up, the Dice-generated code is ready to produce events. We need to be aware that the instrumentation code needs to be thread-safe. The client-side code always needs to be thread-safe, because Dice does not limit the number of client threads using it in parallel. For servers, the code needs to be thread-safe as soon as more than one server thread runs a server loop for the instrumented Dice interface.

For the client case, at the begin of each call to the server, memory for an event is retrieved from the monitoring framework. Then event startup data is written and the normal IPC call is performed. Afterward, final data is written (e.g., the IPC's return value or error code) and the event is committed. All data used by the instrumentation code is local to the function and therefore resides on the thread's stack. This ensures thread-safeness by design.

Server-side code is more difficult for two reasons. The first problem is caused by the server code using more than one function. Therefore the instrumentation cannot simply store event data in local variables. Instead, a pointer to the currently processed event needs to be handed over between functions. [1]

The second difficulty arises, because servers may postpone replies to a client and serve another request in the meantime. Events belonging to such a postponed reply need to be stored until the request is finished. Unfortunately, the number of events to be stored is not known beforehand. It is only limited by the number of possible client threads. This problem is caused by an optimization in my instrumentation code. As explained in Section 3.2.4, data for a whole function call is stored into one event, so that the probe effect caused by calls to the monitoring framework is reduced. By sacrificing this optimization and accepting a higher probe effect caused by posting more events, this problem can be solved. However, a low probe effect is one of the main instrumentation requirements, therefore a different solution is needed.

Server-side Dice code calls a user-defined component function after unmarshaling data. This function returns with a `DICE_DEFERRED_REPLY` flag set, if the current client request shall not be answered immediately. In this case, the event is stored inside a list along with the corresponding client ID. The client ID is used as an index to find this event later on. Using this ID is sufficient, because the client thread at this time is sleeping in its `ipc_call` until the server answers and therefore will not issue another request.

Of course the client may perform this IPC with a timeout set and stops waiting after the timeout expires. This cannot be detected by the server directly, because the kernel does not notify it

[1] Aspect-oriented programming can be helpful here, because it enables to add arguments to functions. Thereby the currently handled event can be handed over between functions.

about expiration of a client timeout. The instrumentation code can detect this situation, if the client starts a new `ipc_call`. Then inserting a new event into the list will reveal that there already is such an event and we can deduce that it has timed out. Unfortunately, no information about the correct time of expiration is available at the server and if such information is needed, it has to be retrieved by instrumenting the client.

Using the afore mentioned list can also be used to circumvent the multiple-function problem. After memory for an event has been retrieved from the Ferret library, this buffer is stored in the list and further function calls inside the Dice-generated server code can retrieve the event buffer belonging to the currently served client.

Because the list is accessed often, its implementation needs to be fast in order to reduce the probe effect. I implemented the list as a linked list as well as a hash table using an $m^2 \bmod n$ hash algorithm[2]. Normally, one would expect the hash table implementation to perform better, but measurements show that the linked list implementation is faster, because of the following reasons:

- A hash table becomes fast, because the search effort is minimized by keeping many small lists instead of one large one. As long as the linked list itself remains small, searching it is as fast as for the hash table. The list used in my implementation typically stores less than 10 elements and measurements show that the hash table is faster only for larger element counts.
- Typically, the instrumentation code queries the list for the same element several times in a row. To support this, I implemented a cache for the last inserted or queried entry.

Sensor support library

The instrumentation code is supported by a sensor support library. This is currently a custom-tailored version for `dice_trace`, but it can easily be generalized. The library encapsulates common sensor tasks such as:

- Setting up a `dice_trace` sensor using a constructor,
- Getting a new event from Ferret and filling its header with default values,
- Committing an event to a sensor, and
- Managing the list of pending events.

[2] m is the checksum of the element to insert, n is the number of hash buckets. $m^2 \bmod n$ is calculated to determine the bucket an element is stored into. If multiple elements map to the same bucket, a linked list is used.

4.2 Network data transfer

To transfer monitored data to another computer for offline evaluation, I decided to use a local network connection. In DROPS the ORe network switch [Döb05b] handles multiplexing of hardware network interface cards for several clients. It works at the ethernet level and therefore does not provide TCP/IP communication, which is needed for efficient data transfer. This task is laid upon its clients. To provide a TCP stack to ORe clients I ported Adam Dunkels' micro IP stack (uIP) [Dun03] to ORe. Originally it was developed as a TCP stack implementation for small embedded systems such as 8-bit micro-controllers. Such systems need to cope with a limited amount of resources, therefore most of them will not be able to run a fully-fledged TCP stack.

uIP is an implementation that comes with all the necessary features to provide TCP communication between hosts, but it removes some functionality from the interface between application and TCP stack that are rarely used in small embedded devices, such as soft error reporting. Furthermore, uIP does not support segmentation of TCP frames, therefore the size of a packet is limited to the frame length of the underlying network protocol. Protocols supported on top of TCP are IP, ICMP, and ARP. UDP is not completely supported.

The advantage of porting uIP instead of a full implementation is simplicity. Porting it to ORe is easy, because it is only necessary to implement a device driver interfacing ORe. Then developers can write a uIP application loop invoking the uIP stack for incoming and outgoing packets. A user-defined callback function is called by the TCP stack whenever application interaction is necessary, for instance upon arrival of a packet or when a connection to a remote host has been established.

The last step in porting uIP was to improve its usability for DROPS applications. The original uIP implementation has several inconveniences, because it requires the uIP files to be compiled statically with the application. Furthermore, users need to implement the previously mentioned callback function, requiring knowledge about uIP internals.

To circumvent these problems, I implemented uIP as a library that can be linked to every application. The library provides a function taking care of communication with the uIP stack, once it has been set up with all information needed by the IP stack (IP address, port to listen on). Furthermore, during configuration pointers to callback functions are set, where each callback function is executed upon a certain TCP event:

- `recv`: called upon reception of a packet. The packet is provided to the callback function. After return from the callback, this packet is not valid anymore, so the user needs to make a copy of the packet if the obtained data is needed outside this callback.

- `ack`: called upon arrival of an acknowledgment. uIP can only handle one TCP segment traveling around at a time, so this is the point where users might want to send the next data packet.
- `rexmit`: called if a packet was not acknowledged for a certain amount of time. uIP does not store pending packets but relies on the user to re-send data here.
- `connect`: called when a connection is established, either because a connect request was issued by the application or a remote host connected to the application.
- `abort`: called when a connection has been aborted.
- `timeout`: called upon a connection timeout.
- `close`: called when a connection has been closed.

It is up to the library's user, whether the uIP thread is started in parallel to the client application.

With the uIP port, there is a working TCP stack available for clients using the ORe network switch. It is used by the monitoring applications implemented for my thesis to transfer obtained data to a remote host for further processing.

Unfortunately, uIP's limitations are a problem when monitoring data is produced at a high rate. The uIP implementation can only cope with one packet being on the way at a time, which leads to a low data transfer rate of up to 3 MB per second for local connections. This is only a quarter of what theoretically should be possible with a 100 Mbit ethernet network. To improve this situation, a more flexible TCP stack should be made available for ORe. Candidates are the L4 Flexible IP Stack (FLIPS) maintained by Christian Helmuth [Hel], and Adam Dunkels' Lightweight IP stack (lwIP) [Dun01].

4.3 Instrumenting L4Linux

4.3.1 Porting kProbes

Adam Lackorzynski [Lac] ported the Linux kProbes mechanism to L4Linux. The major problem was that Linux uses the `int3` instruction to patch instrumented code. This cannot be used for L4Linux, because `int3` is already used for entering the L4 kernel debugger. The x86 instruction set contains an undefined instruction `ud2` that causes a general protection fault. Unfortunately, this instruction is already used by Linux for the BUG macro. It was therefore necessary to find another one-byte instruction, that is able to cause an exception.

Lackorzynski chose to use `hlt` which triggers an exception, because L4Linux is running at privilege level 3, whereas `hlt` requires the privilege level to be 0. This implies that kProbes in L4Linux only work if L4Linux is not running in privileged mode.

4.3.2 Ferret emulation for Linux

As explained in Section 3.4, comparing native Linux and L4Linux is easier if the same kProbe modules may be run with each kernel. To support this task, I added a `ferretlx` kernel module to native Linux. This module simulates the L4Linux kernel sensor inside native Linux and provides access similar to the methods used to access Ferret list sensors.

Data stored by the simulated sensor is available to user space monitors through a special Ferret device. This approach is similar to the one taken by the Linux Trace Toolkit.

I also implemented a histogram sensor inside the Linux kernel. It was used for manual instrumentation besides the kProbes approach and is currently located inside the kernel. Future work will incorporate this sensor into `ferretlx` and add support for multiple list, scalar and histogram sensors.

To evaluate cache and TLB misses in both versions of Linux, I used hardware performance counters. In DROPS these counters can be programmed using the Fiasco kernel debugger and can be read out using `rdpmc()` functions provided by the L4util library. For native Linux, setting up the performance counters leads to modifications to the kernel, because programming the event selection registers is not allowed from user space. Reading the performance counters is possible from user space, if the kernel switches on the PCE bit in the CR4 register.

4.3.3 Improving L4Linux task management

As I explain in Section 5.3, two major performance problems showed up during my evaluation of L4Linux. The first problem arises from L4Linux running in a different context than its applications, so that a system call always leads to at least two context switches.

The second problem results from L4Linux depending on services provided by L4 servers running in parallel. This leads to IPC and context switches between the service providers and the L4Linux server. One example for this problem is task creation and deletion.

Figure 4.2 shows the steps that are necessary to create an L4 task that represents an L4Linux process:

1. An application calls `fork()`.
2. The L4Linux server receives the system call and contacts the L4 task server to allocate a new task. The task is not yet started.

3. The L4Linux server sets up the task's internal data structures and then calls the task server again to start the task.
4. The L4 task server only manages tasks that are owned by the Resource Manager (RMGR). The task server therefore needs to contact RMGR to start the task.
5. The Linux process runs in the L4 task.
6. The task calls `exit()` or is terminated by a signal.
7. The L4Linux server calls the task server to stop the task.
8. The task server terminates the task by calling RMGR.
9. The L4Linux server cleans up the internal task state and thereafter returns the task to the task server.

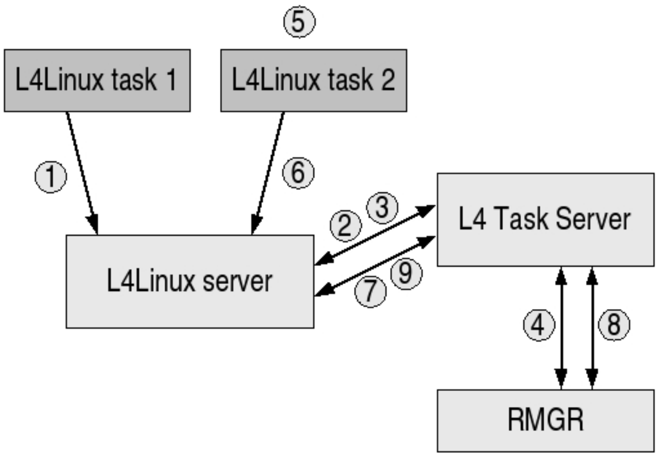

Figure 4.2: L4Linux task creation

There are four possible optimizations to this procedure:

1. *Remove the task server from the chain:* L4Linux can be modified to directly request tasks from RMGR. However, the task server allocates all tasks from RMGR during boot up. Therefore requesting tasks directly from RMGR may result in inconsistent task server state, when an application requests a task from it, while L4Linux is trying to get the same task from RMGR simultaneously.
2. *Unify RMGR and task server*: RMGR and the task server can be unified to reduce context switches. This is not a completely new idea, but has not yet been implemented for complexity reasons.

3. *Transfer task creation rights*: Starting and stopping tasks can be performed using the `l4_task_new` system call. Unfortunately, this is not possible for tasks that were obtained through the L4 task server, because in L4 only a task's *chief* is allowed to perform these operations and the task server does not transfer these rights. Adapting the task server to transfer chief rights to its clients will remove steps 3, 4, 7, and 8 from L4Linux task creation as shown in Figure 4.2.

4. *Task caching:* When a Linux user-space application terminates, L4Linux does not necessarily need to return the corresponding L4 task to the task server. It can keep this task and reuse it for another application instead, saving the overhead of returning and requesting a task several times in a row.

For my implementation I decided to pursue task caching. It soon became obvious that for a task cache to work, L4Linux needed to possess task creation rights. Therefore before implementing a task cache, I needed to provide a means to transfer task creation rights to the L4Linux server.

To pass task creation and deletion rights to arbitrary DROPS applications, I enhanced the task server interface by a `allocate_task_chief` call that allocates a task from RMGR and hands chief rights over to the requesting client. Thereafter, the client does not need to use the `task_create` and `task_kill` operations of the task server for starting and stopping a task. It can use the `l4_task_new` system call instead.

The L4Linux task cache consists of a static array of task entries. [3] When allocating a new task, L4Linux uses the `task_from_pool` function to get a cached task. If this does not succeed, the task server is queried for a new task. During task termination, L4Linux adds the now unused task to the cache using `task_to_pool` and only returns this task to the task server, if the task cache is already full.

I evaluate performance implications of my adaptions to task management and task caching in Section 5.3.3.

4.4 A self-healing web server

As introduced in Section 2.4, runtime monitoring is an essential part of autonomic computing. To show that the infrastructure developed for this thesis also supports self-monitoring in the sense of autonomic computing, I implemented a self-healing application. For this example I chose a small web server running with the uIP TCP stack.

The web server is a port of the `mini_http` web server originally developed for FLIPS. Runtime monitoring is achieved by running the server with two threads:

[3]The size is user-configurable. For my experiments I used a cache size of 5 entries.

1. One thread handles TCP/IP communication by running the uIP library thread and the callbacks specified by the web server.

2. Another thread monitors the communication caused by the uIP library thread, detects error situations in this thread and restarts it upon an error.

Errors can be detected by using internal knowledge about the uIP stack. The stack provides address resolution through the ARP protocol. ARP entries have a time-to-live and therefore the uIP stack needs to have a way of timing. This is achieved by calling ORe with a timeout of 500 ms. If this call times out, the ARP timer is updated. Therefore, an application using the uIP stack and ORe will produce at least two IPC events per second. A periodic monitor can obtain the IPC events generated by this communication and deduce a failure situation if no IPC event was generated during the last monitoring period.

Failures in the `mini_http` server can be recovered by *microrebooting* the server's worker thread by adding a runtime monitor to the application that restarts the worker thread whenever it detects a failure as described previously. Microreboots are introduced by Candea et al. in [CKF+04]. The authors state that many software failures can be solved, without knowing the correct error leading to a crash, by rebooting the system. They argue that rebooting a whole machine is time-consuming and introduce the concept of *microrebootable components*, where upon an error only a small software component is restarted. This takes much less time than a complete reboot and can already be a solution to the failure. If this is not the case, a complete reboot will be necessary.

Five preconditions must be fulfilled to support microrebooting:

1. *Fine-grained components*: To keep time for a microreboot low, software consists of small componentes with well-defined interfaces. The `mini_http` server is such a small component.

2. *State segregation*: Failing components must not corrupt any global state. Candea et al. [CKF+04] propose to manage software state in an external state storage so that it can be recovered after microrebooting. The `mini_http` server keeps no global state apart from a visitor counter. For my example I assume that this counter is not corrupted by the server's worker thread.

3. *Decoupling*: Components must not be tightly-coupled, for instance by storing a pointer to data in component A inside component B, because this pointer will be invalid after microrebooting component A. In my `mini_http` there are two components: the worker thread and the monitor. They are not directly coupled.

4. *Retryable requests*: The system must tolerate the inavailability of components. If a request fails because a component is not present at a moment, the caller must retry its request after a certain amount of time. Requests to `mini_http` are issued from a web browser using the HTTP protocol. HTTP assumes that clients retry their requests if necessary, no modifications to `mini_http` need to be made.

5. *Resource leases*: Resources used by software components need to be returned to the operating system even if the component fails. This is achieved by *leasing* resources. Components request a resource and need to request it again after their lease time expires. If the component fails, all its resources will automtically be returned to the resource managers upon expiration.

   The only resource used by the `mini_http` server is its connection to the ORe network switch. This connection needs to be cleaned up before restarting the worker thread.

To extend the web server with runtime monitoring capabilities, there exist several possible solutions. The first option is to use IPC events from the Fiasco trace buffer to detect errors. The trace buffer filter is able to filter events for exactly one task, therefore this alternative is applicable exactly for the web server example. However, it is not possible to support further self-healing applications in the same way without taking into account a large overhead caused by the need to filter a lot of unwanted events.

I chose a simpler and nearly non-intrusive solution by instrumenting the uIP library with a sensor that simply provides the number of calls to the ORe network switch. This Ferret sensor is a scalar, which means that the sensor overhead is smaller than for other sensors, such as a list sensor. It adds only 5 lines of code to the uIP library [4].

The monitoring code in the web server is also simple. After setting up the TCP stack and starting the uIP library thread, the web server's main thread starts running a periodic monitoring function:

```
/* - attach to Ferret sensor */
while (1)
{
    val = ferret_scalar_get(sensor);
    if (val == oldval) // ERROR!
        restart_worker_thread();
    /* sleep for some time */
    l4_sleep(4000);
}
```

[4]The uIP library consists of 600 lines of code.

These small enhancements are enough to provide the web server with self-healing capabilities, which lead to the following improvements for the web server:

- User-experienced downtime caused by server failures is decreased because failures are repaired within an amount of time that is smaller than the one needed to restart the whole machine.
- In the case of restart, the newly started worker may reuse the global server state. If user sessions contain more data than the simple counter used by `mini_http`, an external reliable storage needs to be used.

A drawback of this solution is that it uses manual instrumentation of the uIP library and the web server. Again, aspect-oriented programming will be helpful to automate this task, but is not available at the time of this writing.

I also considered using the `dice_trace` plugin introduced in Section 4.1. We can instrument the ORe client library with the help of this plugin and compile the web server with this instrumented library. I discarded this solution due to the following drawbacks:

1. `dice_trace` instruments all functions in an interface and all applications instrumented using this plugin commit their events into one single sensor. The web server's runtime monitoring code therefore has to filter out the data it needs from a potentially large amount of superfluous data. This drawback is caused by the layout of the tracing plugin. It can be circumvented by a custom-tailored trace plugin for this use case.
2. `dice_trace` uses a list sensor instead of a scalar, which leads to a considerably higher probe effect.

4.5 Magpie enhancements

Martin Pohlack [Poh] adapted Magpie to be used with events originating from DROPS applications and designed the Ferret monitoring framework to be compatible with Magpie. During my experiments I enhanced the L4 request schema to handle the events from my instrumentation. It soon became obvious, that the Magpie event parser can also be used for tasks beyond preparing data for visualization with the Magpie visualizer.

I adapted Magpie to work with multiple request schemata. This enables us to perform different types of evaluation of the event stream at once. Schemata are handed over to Magpie as modules from the command line. Each module needs to provide

- An `init` method, performing initializations,
- A `dispose` method for unloading the module and cleaning up, and
- Event handlers to collect data about single events.

Up to now, I implemented four modules in addition to the original L4 request schema. Two of them create visualization output by drawing a graph of communication relationships derived from `dice_trace` events. One of the modules targets the GraphViz [gra] graph visualizer, the other one produces output for the uDrawGraph [UB05] tool.

A third module collecs information from `dice_trace` events and turns them into overall statistics. It produces statistics about *connections*, which are unidirectional IPC relationships in the context of this module. A *usage rating* shows the number of IPCs sent through each connection. The *timeout hit ratio* shows the amount of IPCs that have been canceled because their timeout expired. Furthermore, *cumulative statistics* give information about how many connections made up N% of all communication with respect to the number of IPC messages sent and to the size of the sent data.

While instrumenting the L4 Environment with a semaphore sensor as described in Section 3.3, I also implemented a semaphore evaluation module for Magpie. This module takes data generated from the `l4semaphore_down()` and `l4semaphore_up()` methods and evaluates

- Average waiting time to enter a critical section,
- Average length of the wait queue, and
- Average time to execute the critical section.

5 Evaluation

Take your time, think a lot,
Think of everything you've got,
For you will still be here tomorrow,
But your dreams may not.
(Cat Stevens - Father and Son)

In this chapter I will discuss experiments that were carried out during my thesis. At first, I will evaluate DROPS' IPC performance. Thereafter I will evaluate L4Linux, especially in comparison to native Linux. I will point out the overhead of the different kProbes implementations in both systems and analyze overall and system call performance. Finally, I will evaluate the performance gains resulting from my implementation of L4Linux task caching which I described in Section 4.3.3.

5.1 Test setup

The experiments presented in this chapter were all run on the same test computer, if not stated otherwise:

- AMD Duron 800 MHz CPU,
- 256 MB RAM,
- 64 kB L1 Instruction Cache (2-way associative),
- 64 kB L1 Date Cache (2-way associative), and
- 64 kB L2 Universal Cache (8-way associative)

Fiasco was configured with the following performance-critical options:

- No assembler IPC shortcut, because L4Linux needs the exception IPC feature and this does not work with the IPC shortcut, and
- Fine-grained CPU time in order to get correct thread execution times.

Experiments comparing L4Linux and native Linux were run with Linux 2.6.16 versions of the kernels on the same test machine. If not stated otherwise, no hard disk was used and the systems were completely booted and operated from a 16 MB ramdisk. For the kernel compile benchmarks, a hard disk was mounted in both systems:

- Fujitsu MPG3204AT ATA
- 20 GB split into 6 logical partitions
- 512 kB disk Cache
- cached read throughput: 402.2 MB/s [1]
- buffered disk read throughput: 3.97 MB/s [2]

5.2 Analysis of DROPS

5.2.1 IPC sizes

A first experiment regarding IPC was the analysis of average message sizes. For this experiment I used the `dice_trace` plugin to instrument the following IDL interfaces:

- The ORe network switch,
- The L4 Virtual File System (L4VFS), and
- The DOpE Window Manager.

Using the generated sensors I collected IPC data for the following scenario: The DOpE Window Manager and the ORe network switch are running on top of the standard L4Env servers. The L4Loader is used to load

- The L4VFS virtual console server `vc_server`,

[1] `hdparm -T`
[2] `hdparm -t`

- The L4VFS `fstab` server,
- The L4VFS console test application,
- The DOpE logging console `dmon`,
- The DOpE `vscrtest`, and
- The ORe arping client

from the network. The binaries are retrieved using TFTP and ORe and then make extensive use of the instrumented interfaces during execution.

Figure 5.1 shows how IPC message sizes cumulate. The drawn through line shows the proportion of packets in relation to the complete number of packets obtained. We can see that about 85% of all IPC messages have a size below 90 bytes. The dotted line shows how the packets' message sizes add up to the complete amount of data sent. From this line we see that the 85% of the packets with message sizes below 90 bytes make up less than 10% of the total data sent. The reason for this observation is that only ORe and its clients perform data-intensive communication in my scenario.

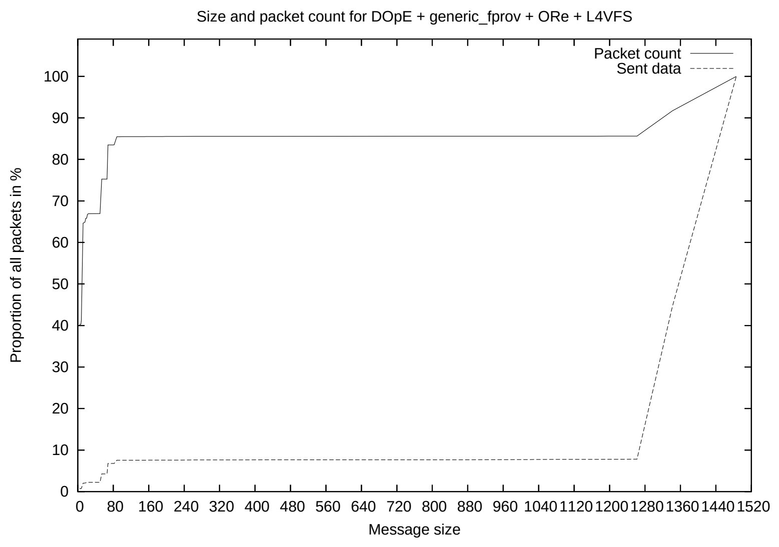

Figure 5.1: Proportional distribution of IPC sizes for the DOpE+L4VFS scenario

I then removed the ORe instrumentation and repeated the experiment. The results are completely different now and are shown in Figure 5.2. Now 98% of all messages have a message size below 30 bytes and only a few more messages are larger.

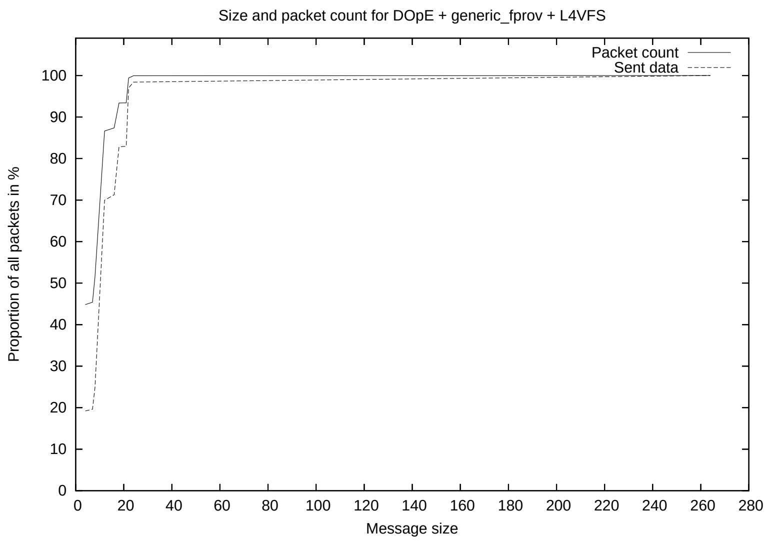

Figure 5.2: Proportional distribution of IPC sizes for the DOpE+L4VFS scenario without ORe

Lessons learned

It is not possible to determine correct averages for L4 IPC message sizes, because these numbers largely depend on the chosen scenario and the instrumented interface. The means developed with my thesis can be used to evaluate scenario-specific measurements.

One interesting fact from the results in Figures 5.1 and 5.2 is the large amount of short messages. In both cases more than 40% of all messages are at most 8 bytes large and can be sent using short IPC, which is reasonably faster than string IPC in DROPS. There are two main reasons for that:

1. Programmers know of the advantages of short IPC and optimize their interfaces to use it.
2. Many IPC calls are remote procedure calls (RPC). Typically, functions return at least an `int` value to signal success or failure. On 32-bit machines an `int` is four bytes large and can be sent with a short IPC.

5.2.2 String IPC throughput

The next experiment regarding IPC in DROPS covered indirect string IPC [Lie96a]. I sent data with sizes up to 16 MB from a client to a server application and used `dice_trace` instrumentation to measure the transmission times. As can be seen from Figure 5.3, transmission times for large strings grow linear to the string size. With this in mind, a naive approach to sending a fixed-size buffer is to send the buffer directly with one IPC call, because this leads to the lowest number of kernel entries and therefore reduces overall cost.

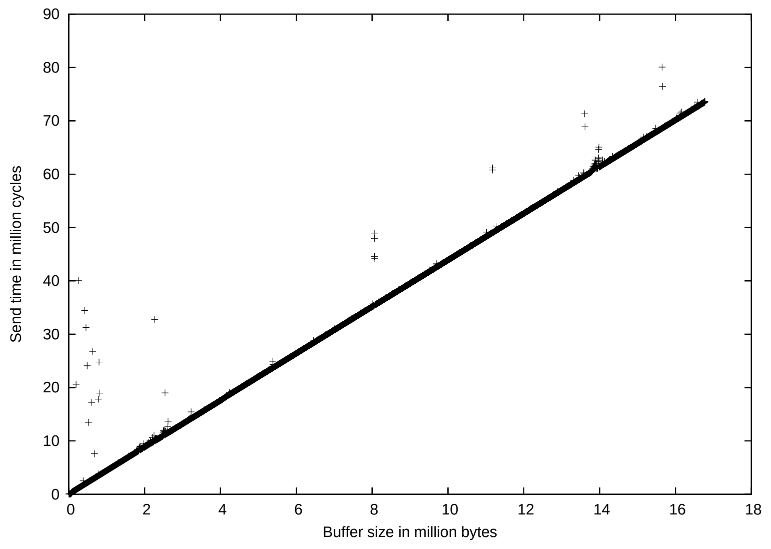

Figure 5.3: Send time for indirect strings

I tried to support this thesis by performing another benchmark: This time I transfered a 2 MB buffer between client and server and used IPC message sizes varying from 256 bytes up to 2 MB. The client sends a message and this message is written into a fixed-size receive buffer at the server side. The server does not touch the received data but only acknowledges it. In addition to my test computer, I performed the benchmark on two other machines:

- An Intel Celeron 900 MHz CPU with 16 kB L1 data and instruction caches, and 128 kB universal L2 cache, and
- An Intel Pentium 4 1.6 GHz CPU with 12 kB L1 instruction cache, 8 kB L1 data cache and 256 kB universal L2 cache.

Figure 5.4 shows, that the previously stated thesis is wrong. Surprisingly, sending the 2 MB data with send sizes ranging from 16 kB to 64 kB depending on the test machine is up to two times faster than sending the buffer at once.

Figure 5.5 gives hints about the reasons for this unexpected behavior. IPC with small send sizes wastes time by performing too many context switches between the IPC partners, which cause a

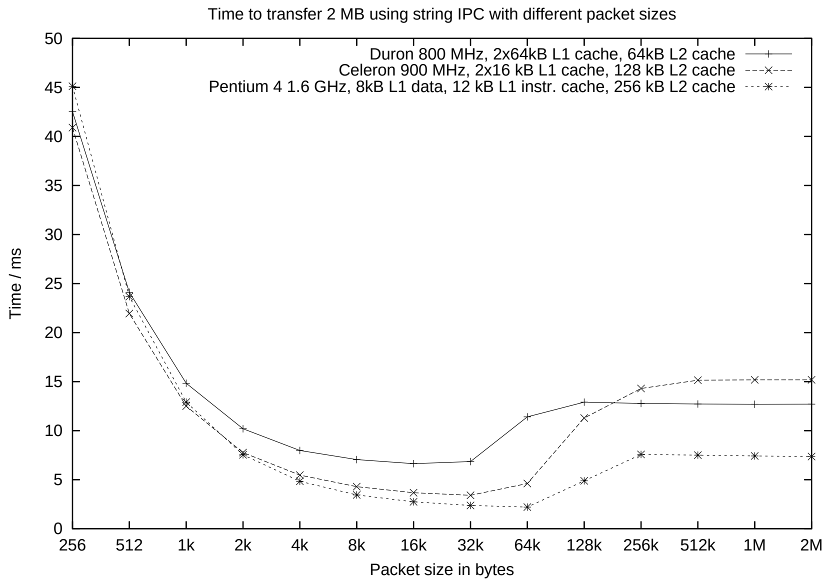

Figure 5.4: Sending a 2MB buffer with different send buffer sizes

high number of TLB misses, because TLBs are flushed at every context switch. For large send sizes, the number of context switches and TLB misses is smaller.

However, the number of cache misses during one 2 MB transfer rises from a certain point and increases transfer time once again. The results from Figure 5.4 imply that the optimal send size depends on the CPU's cache sizes. If the send size closes in on the cache size, it becomes more probable, that kernel data and instructions are thrown out of the cache during an IPC's copy operation. If the send size gets higher than the cache size, the cache is always completely thrashed by the IPC operation. Figure 5.5 correlates send times, cache misses, and TLB misses.

Future work shall investigate whether using cache coloring [Lie96b] to separate kernel and user-space cache usage can help to improve performance of indirect string IPC. Another option is to not cache the data copied during an IPC operation — this will reduce IPC overhead caused by cache misses, but it will lead to more cache misses later on, when the receiver accesses the received data.

Lessons learned

Because of cache effects copying data in smaller packets into a single receive buffer can be faster than copying a whole buffer at once. This is not always possible, but it can be an option

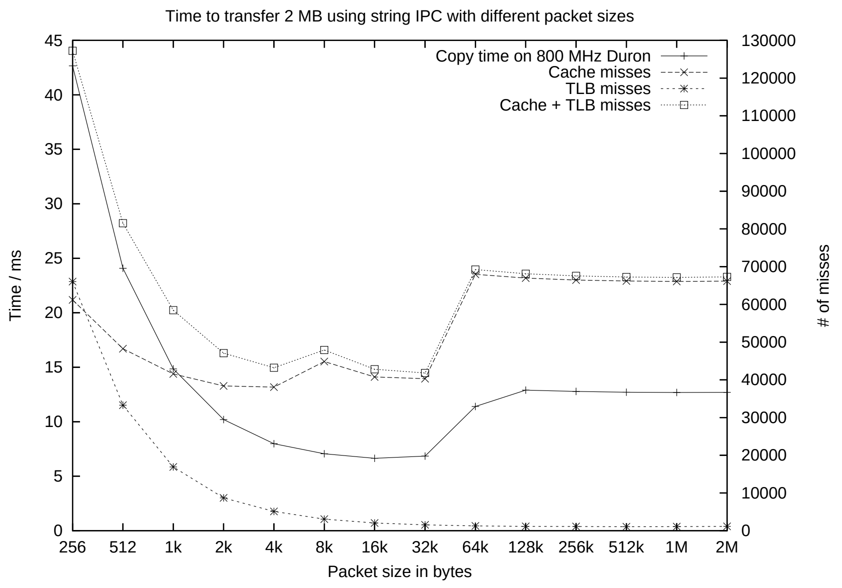

Figure 5.5: Cache and TLB misses for sending a 2 MB buffer using indirect string IPC

for multimedia-applications that process one piece of data at a time. Therefore this knowledge should be taken into account when designing DROPS servers.

5.2.3 String IPC vs. shared-memory communication

The DROPS Streaming Interface (DSI) [LHR] is an implementation of a shared-memory communication protocol for DROPS. It uses a producer-consumer protocol on a ring buffer and therefore only needs to use IPC when producers or consumers need to be woken up.

For my experiments, I tried to compare the performance of DSI and indirect string IPC for the scenario of transferring a 2 MB buffer with varying packet sizes. DSI cannot be instrumented using `dice_trace`, therefore I chose to perform manual instrumentation by taking global timestamps before and after sending the whole buffer. I furthermore evaluated DSI with respect to two different ways of sending data:

1. When performing a *zero-copy send* operation, the sender prepares all of its data directly inside the shared memory buffer. No copy operation is then needed to transfer data. DSI was designed for this use case.

2. A sender can also use DSI as a replacement for indirect string IPC. Then it is necessary to copy the data into the shared memory buffer before sending a packet through DSI. This

adds a whole copy operation, but might still be faster than string IPC because it leads to less kernel entries and exits.

Figure 5.6 shows the obtained results. Zero-copy outperforms both string IPC and manual copying — it takes not more than 4,000 CPU cycles when sending a single 2 MB packet. However, string IPC proves to be faster than manual copying. Interestingly, the cache effect described in Section 5.2.2 did not occur in the DSI experiment.

The difference between both experiments is the string IPC experiment copying all packets into the same target buffer, thereby touching only a small amount of memory. By contrast, the DSI experiment copies a packet from the target buffer to exactly the same offset in the shared-memory buffer. Thereby a larger address range is touched during the copy operation, leading to more cache misses.

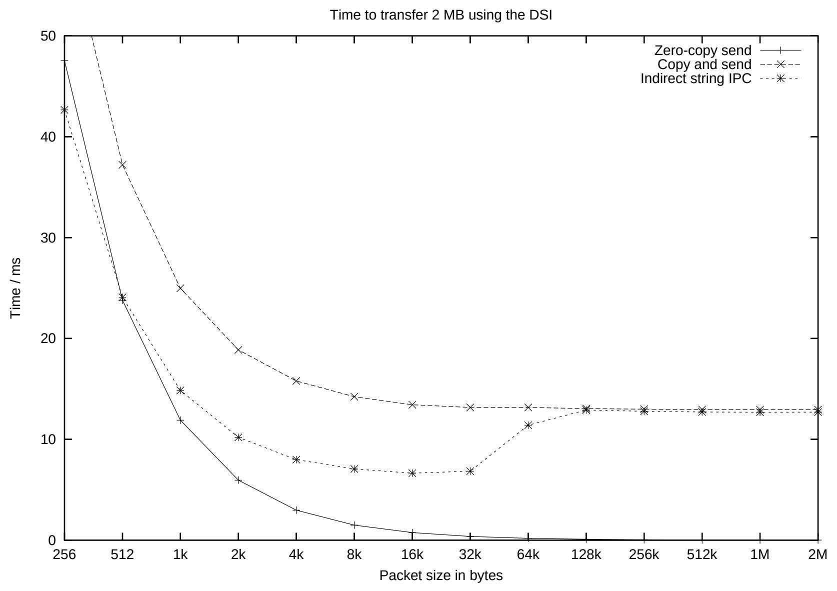

Figure 5.6: Comparing DSI and indirect string IPC

To prove this, I experimented with different configurations for the DSI connection by altering the maximum number of DSI packets. Figure 5.7 shows that the more packets are used with a DSI connection, the smaller the observed cache effect becomes. Using only a single DSI packet works the same way as indirect string IPC and shows the same cache effect. More packets increase parallelism between producer and consumer, but show more cache misses which in turn decrease performance.

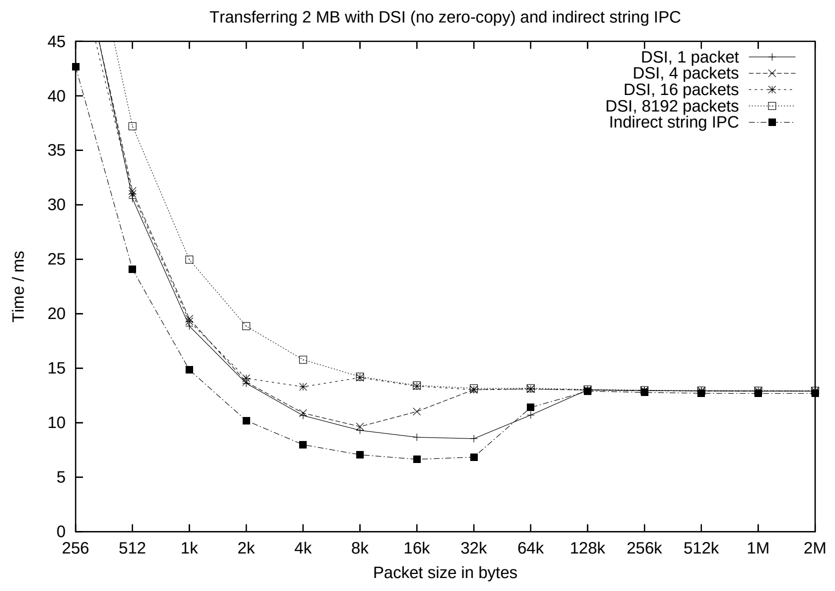

Figure 5.7: DSI w/o zero-copy, different packet configurations

Lessons learned

Shared-memory communication using the DSI is extremely fast when performing zero-copy send operations. If zero-copy is impossible for a scenario, indirect string IPC is faster than DSI with manual copying.

5.2.4 Semaphore usage

In Section 3.3 I described how the semaphore library can be instrumented with sensors that produce data for profiling applications. With the help of this instrumentation I tried to answer the question, whether a semaphore implementation for DROPS needs to be optimized for the case where no other threads are requesting the semaphore or for the case where multiple threads are waiting to enter a critical section.

I evaluated semaphore usage for a range of scenarios:

1. DOpE client and server semaphores while running the `vscrtest` application and the `dmon` log server.
2. ORe client and server semaphores while loading L4Linux via `tftp.ore`, and
3. Three instances of L4Linux loaded via `tftp.ore`, and performing `wget` each.

Lessons learned

For all three scenarios the vast majority of semaphore accesses was without a need to wait for the semaphore. Semaphore implementations therefore definitely need to be optimized for the non-contention case.

5.3 Analysis of L4Linux

5.3.1 kProbes Overhead

As a first experiment, I compared the overhead induced by using kProbes in L4Linux to the kProbes overhead in native Linux. I expected the kProbes overhead in L4Linux to be different from native Linux, because the implementations are different, too.

For my experiment I used a simple benchmark, performing 10,000 calls to the `fork()`, `vfork()`, and `execve()` functions provided by the Standard C Library and measuring their execution time. This was done without any kProbes in the system and with only one kProbe with a single pre-handler function registered on the first instruction of `do_fork()`. As kProbes overhead might vary depending on the instrumented location, I chose to perform the same test with the `getpid()` function and a kProbe registered upon the first instruction of `sys_-getpid()`.

All tests were run on my test computer running a (L4)Linux 2.6.16 kernel with a small ram disk containing my benchmark and the kProbes kernel module.

Table 5.1 shows the resulting execution times for native Linux, Table 5.2 shows the results for L4Linux. All values are CPU cycles.

	Without kProbes	With kProbes	Overhead
getpid	277	1,046	769 (*277.6 %*)
fork	90,798	92,061	1,263 (*1.4 %*)
vfork	21,407	22,164	757 (*3.5 %*)
execve	90,369	92,916	2,574 (*2.8 %*)

Table 5.1: kProbes overhead for native Linux system calls

	Without kProbes	With kProbes	Overhead
getpid	3,408	9,171	5,763 (*169.1 %*)
fork	329,818	345,556	15,738 (*4.8 %*)
vfork	44,564	52,818	8,254 (*18.5 %*)
execve	236,731	257,240	20,509 (*8.7 %*)

Table 5.2: kProbes overhead for L4Linux system calls

Lessons learned

The results show, that using kProbes definitely results in a performance overhead, which is higher in the L4Linux implementation than in the native one. While for long-running system calls such as `sys_fork` and `sys_exec`, the overhead becomes negligible, this is not the case for short-running calls such as `sys_getpid`.

5.3.2 Native Linux vs. L4Linux

Adam Lackorzynski adapted L4Linux to use the L4 Environment [Lac02]. According to his measurements [Lac04], L4Linux has a considerable overhead in comparison to native Linux. With the help of the tracing facilities presented in my thesis, I tried to find reasons for these performance problems.

A system benchmark

As a first step, I used the Unixbench system benchmark [Nie99] to get an overview of where performance problems are. Unixbench is a benchmark suite consisting mostly of synthetic benchmarks testing the following subsystems of Linux and the underlying hardware:

1. **Arithmetic operations:** Unixbench runs the Whetstone [CW76] floating point benchmark, the Dhrystone [Wei84] integer benchmark, and five synthetic benchmarks testing the arithmetic performance for arithmetic data types (`int`, `short`, `long`, `double`, and `float`).

   Results for the arithmetic benchmarks should not differ much between Linux and L4Linux, because they only measure hardware performance.

2. **File input–output:** Unixbench tests reading, writing, and copying a file with different buffer sizes (256 bytes, 1024 bytes, 4096 bytes).

3. **System performance:** System performance is tested with respect to process creation, system call overhead, pipe throughput and `execl` performance.

4. **Application benchmark:** Unixbench also performs an application benchmark by running a shell script alone, followed by 8 and 16 instances of the script running in parallel.

Table 5.3 shows the Unixbench results. They were obtained by running L4Linux 2.6.16 and Linux 2.6.16 on the same computer with the same ram disk setup.

The first section in Table 5.3 covers arithmetic benchmarks. As both versions of Linux ran on the same machine, the results are practically equal.

The second section contains results for the system call benchmarks. They show that major performance problems result from the system call overhead and from process creation. Native Linux can perform eight times more system calls in a period of time than L4Linux. Furthermore, process creation is five times faster in native Linux.

The results for file input–output in the third section and for the application benchmark in the fourth section show that there are further performance differences. However, because of the large penalty for system call overhead I cannot deduce the penalty for these areas without further investigation.

Test	Dimension	native Linux	L4Linux	relative difference L4Linux vs. native
Dhrystone	loops/second	1,810,146	1,819,542	+0.5%
Whetstone	MWIPS	558	557	-0.2%
Arithmetics (short)	loops/second	156,815	157,058	+0.2%
Arithmetics (int)	loops/second	163,164	163,470	+0.2%
Arithmetics (long)	loops/second	163,140	163,397	+0.2%
Arithmetics (float)	loops/second	311,530	311,436	0.0%
Arithmetics (double)	loops/second	310,705	311,366	+0.2%
Arith. Overhead	loops/second	3,687,234	3,693,022	+0.2%
Syscall overhead	loops/second	358,526	43,161	-88.0%
Pipe Throughput	loops/second	218,889	70,857	-67.6%
Pipe-based Context Switches	loops/second	68,960	31,275	-54.6%
Process creation	loops/second	4,426	845	-80.0%
Execl Throughput	loops/second	956	401	-58.1%
Read(1024/2000)	kB/second	238,209	115,824	-51.4%
Read(256/500)	kB/second	89,721	31,378	-65.0%
Write(1024/2000)	kB/second	171,200	86,086	-49.7%
Write(256/500)	kB/second	52,666	25,104	-52.3%
Copy(1024/2000)	kB/second	97,745	49,591	-49.3%
Copy(256/500)	kB/second	33,556	13,837	-58.8%
Shell scripts (1)	loops/minute	1,775	829	-53.3%
Shell scripts (8)	loops/minute	258	120	-53.5%
Shell scripts (16)	loops/minute	129	60	-53.5%

Table 5.3: Unixbench results. (Numbers in the file I/O section give the buffer size and the number of accesses, for shell scripts it is the number of scripts running in parallel.)

System call performance

As a next step I tried to inspect the runtime of certain system calls inside the kernel to find out whether there are performance penalties for some of them. Such an investigation is especially useful for frequently used system calls, because improving their performance can improve overall system performance most. Using the system call counter introduced in Section 4.3, I obtained statistics about the frequency of system calls for a run of the Unixbench benchmark and the kernel compile benchmark. This resulted in a list of the top 15 system calls used for these scenarios. I show these 15 system calls in Table 5.4.

`sys_read`	`sys_write`	`sys_open`
`sys_close`	`sys_mmap2`	`sys_oldmmap`
`sys_munmap`	`sys_llseek`	`sys_stat64`
`sys_fstat64`	`sys_rt_sigaction`	`sys_rt_sigprocmask`
`sys_getpid`	`sys_clone`	`sys_brk`

Table 5.4: Top 15 Linux system calls for Unixbench and kernel compile benchmark

As `sys_getpid` is one of the most short-running system calls, I skipped it for further evaluation, because its execution time depends mainly on the overall system call overhead. For the remaining 14 system calls, I measured their average execution times for five executions of the kernel compile benchmark on each system.

The results in Table 5.5 show, that for most of the evaluated system calls there is no extreme difference between native Linux and L4Linux. The `sys_read` and `sys_clone` calls are the ones differing most between the two versions. I therefore further investigated these calls' behavior.

Syscall	min. time Linux	avg. time Linux	max. time Linux	min. time L4Linux	avg. time L4Linux	max. time L4Linux
read	2,699,180	2,776,963	3,018,419	3,698,098	3,871,975	4,263,079
write	325,209	372,634	451,814	293,765	463,279	540,126
open	20,865	30,869	36,222	18,765	26,022	33,307
close	233	308	412	126	141	166
mmap2	10,962	11,772	12,654	12,520	13,968	15,221
oldmmap	5,787	6,089	6,466	8,522	8,941	10,159
munmap	5,302	5,400	5,646	10,239	10,656	11,610
llseek	952	994	1,597	931	998	1,195
stat64	9,510	13,604	17,071	3,821	4,193	4,748
fstat64	1,440	1,522	1,597	7,837	10,104	14,605
sigaction	827	1,137	2,298	1,484	1,550	1,754
sigprocmask	655	754	929	1,423	1,491	1,702
clone	6,607,959	7,115,684	7,417,643	143,032	149,739	169,353
brk	1,654	1,750	1,884	2,147	2,266	2,551

Table 5.5: Average system call execution times for L4Linux and native Linux. All values are clock cycles.

Runtime for `sys_read` varies between subsequent calls. The call returns fast if the read operation is asynchronous or the data to be read is already available from a cache. In contrast, blocking read operations to disk take a lot of time. Therefore, I first expected the differences between L4Linux and Linux to derive from unpredictable disk read overhead. However, the average execution times proved to be very stable. As a next step I tried to visualize the distribution of execution times, which is shown in Figure 5.8.

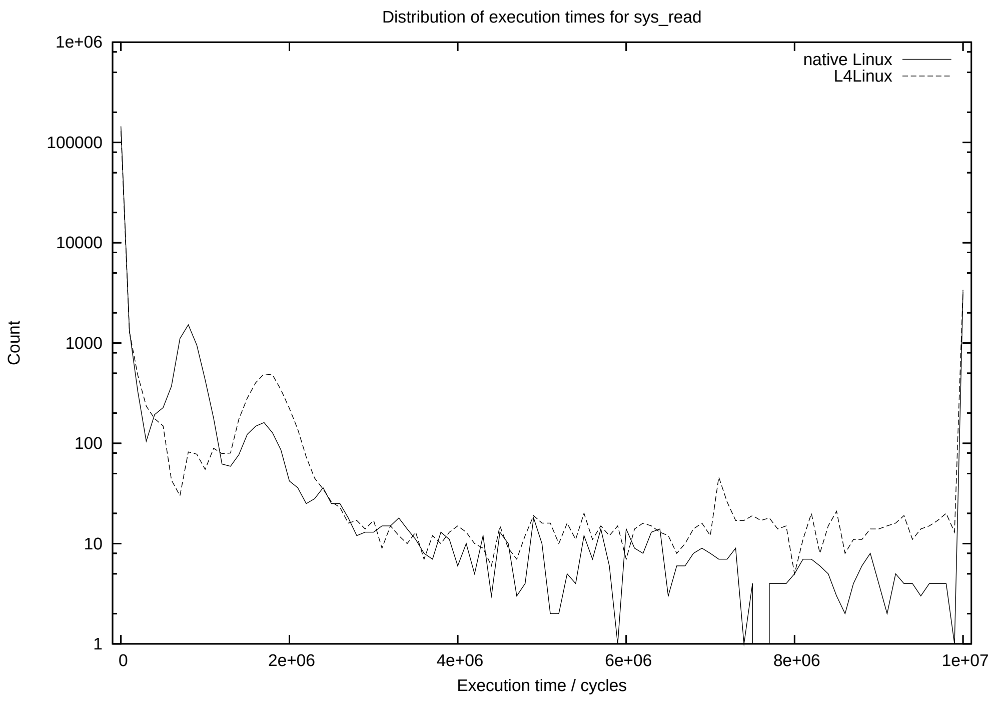

Figure 5.8: Execution times for the `sys_read` system call

The visualization shows high counts of short-running and extremely long-running read operations. In between there is one peak for each system[3]. Interestingly, the offset between the peaks is nearly as high as the offset between the two measurements' averages. I therefore deduced that this is the cause for the differing execution times. As the peaks are located nearer to the low values than to the high ones, disk overhead does not seem to be an explanation for this.

I decided to compare the average hardware cache and TLB misses for `sys_read`. The results shown in Table 5.6 can give hints about the cause for higher execution times in L4Linux. As L4Linux system calls always lead to one context switch from the application to the L4Linux server and one context switch back to the application, TLBs are flushed more often. This leads to a higher amount of cache and TLB misses and slows down operation. This especially affects `sys_read`, because it is one of the longest-running calls in the system, but it is also an explanation for the other system calls in Table 5.5 that have a slightly larger runtime in L4Linux.

[3]The peaks are at x=1,200,000 for native Linux and at x = 2,000,000 for L4Linux

	native Linux	L4Linux
Instr. cache misses	10,441	12,598
Data cache misses	7,677	8,337
ITLB misses	88	973
DTLB misses	676	1,715

Table 5.6: Average cache and TLB misses for one `read` system call

To support my thesis, that cache and TLB misses are a major cause of performance problems in L4Linux, I ran another synthetic benchmark. This time I executed `sys_getpid` for 100,000 times. This system call simply returns `current->tgid`, taking only about 14 cycles to execute inside the kernel. This means that when measuring this system call from user space, it will result in information about the real system call overhead of native and L4Linux. Table 5.7 shows the obtained results, which support my thesis.

	native Linux	L4Linux
User-space execution time in cycles	274	3,778
Instruction cache misses for 100,000 calls	544	1,703,332
Data Cache misses for 100,000 calls	286	402,398
ITLB misses for 100,000 calls	35	501,679
DTLB misses for 100,000 calls	141	1,052,520

Table 5.7: TLB and cache misses for sys_getpid

Lessons learned

System-call performance in L4Linux is decreased by the fact, that each system call leads to at least two context switches, which in turn lead to a much higher amount of TLB misses than for native Linux.

A solution for this problem needs to remove either the necessity of switching between different contexts for a system call, or the necessity of flushing TLBs during a context switch:

1. *Tagged TLBs* add an address space identifier to each TLB entry. Therefore it is not necessary to flush the TLB during a context switch.

2. *Cache coloring* [Lie96b] is a memory allocation strategy splitting the cache between applications, for instance between the L4Linux server and its applications. This results in less memory and cache being available for each application, but it ensures that application A accessing the cache does not trash a cache line that is used by application B.

3. *Small address spaces* [Lie95] split one address space into several smaller ones. Thereby the L4Linux server and its applications can run in the same context, but with less memory available. This leads to no context switch for system calls and thereby decreases the number of TLB flushes.

5.3.3 Task caching in L4Linux

As explained in Section 4.3.3, I tried to improve L4Linux' task management by

- Obtaining chief rights during task creation in order to start and stop tasks using the `l4_-task_new` system call, and
- Caching a fixed number of tasks instead of returning them to the task server, so that future task creations do not necessarily result in communication with the task server and RMGR. The task cache's size was set to 5 tasks for my experiment.

To evaluate these improvements, I reused the benchmark introduced in Section 5.3.1, this time only measuring execution times of `fork()`, `vfork()`, and `execve()`. Table 5.8 shows the results in CPU cycles.

	L4Linux unmodified	L4Linux with new task management	L4Linux with new task management and caching
`fork()` min.	330,425	292,857	255,087
`fork()` avg.	998,971	911,580	858,967
`vfork()` min.	45,502	44,274	45,510
`vfork()` avg.	56,556	55,565	56,185
`execve()` min.	237,014	243,574	207,376
`execve()` avg.	901,274	848,268	771,902

Table 5.8: Effects of improved L4Linux task management

Lessons learned

Execution time of `sys_fork` is sped up by 11.4% with the new task creation mechanism. Using task caching improves this even more, resulting in a 22.8% performance gain. `vfork` execution time remains unchanged, because this system call does not use task creation. Instead, task creation is postponed until the `vfork`ed application calls `execve`.

Interestingly, best-case performance for the `exec` system call is not improved by the new task management, while the average-case execution time is improved by 5.9%. Using task caching even shows up in best-case execution times, speeding up the system call by 12.5%.

6 Conclusion and outlook

On and on ’cause the road is never-ending.
At least we know, we’re on our way.
(Fiddler’s Green - On and on)

6.1 Conclusion

In my thesis I integrated means for tracing requests into DROPS. I reused existing DROPS solutions such as the kernel trace buffer and the Ferret monitoring framework. Furthermore, I defined general sensors as well as monitors to be used by tracing facilities.

I extended the DROPS IDL Compiler with a plugin that is able to automatically generate IPC tracing code and ported a small TCP/IP stack for the ORe network switch to send monitoring data through the network for offline evaluation. I used the Magpie event processing tool chain and extended it for event processing and visualization.

With the help of sensors inside the L4Linux kernel, I was able to instrument L4Linux as well as applications running on top of it. Using a device driver providing the Ferret monitoring interface, I could reuse L4Linux kProbes modules in native Linux and carry out comparative measurements.

Using the previously mentioned facilities, I evaluated and analyzed the behavior of DROPS components:

- I showed that the performance of indirect string IPC depends on a processor’s cache setup.
- I compared the DROPS Streaming Interface to indirect string IPC and showed that the communication using the DSI is faster only for zero-copy send operations.
- I evaluated L4Linux and pointed out that context switches for every system call are one main cause of performance penalties in comparison to native Linux. Solutions for these problems can be tagged TLBs, cache coloring, and use of small address spaces.
- I implemented a task cache in L4Linux, thereby improving task creation performance by up to 20%.

6.2 Future work

In this section I will outline ideas for future work based upon my thesis. These ideas came to my mind during my work and were not put into practice because there either was not enough time available for it, or the implementation was out of scope for this thesis.

1. *Generic tracing:* In order to make future versions of DROPS be usable in the context of autonomic computing which was introduced in Section 2.4, we need generic means of instrumentation, so that instrumentation becomes easier for application developers. The `dice_trace` plugin presented in Section 4.1 is an example for automated instrumentation. As mentioned earlier, using aspect-oriented programming can be another major step towards this goal.

   As proposed in Section 3.3, basic DROPS system services such as the L4 Environment shall be modified to provide default sensors to all their clients.

   The Systemtap project [FCE05] aims at providing a high-level scripting language that can then be translated into many different probe languages. It will be interesting to follow Systemtap's evolution and I consider it to become the language of choice for automated instrumentation in DROPS.

2. *Automating generation of Magpie schemata:* Writing a Magpie event schema requires some understanding of the underlying event processor and the Python programming language. Although a schema written in a real programming language has the advantage of being flexible, it may not always be the easiest solution. From my point of view, it is sufficient to describe the events and do simple bindings to timelines. For these scenarios, it will be interesting to have a high-level schema-description — for instance in an XML file — which then is compiled into a Magpie event description and an event schema by a code generator [1]

3. *Port Linux tracing tool chains:* Within my thesis I used kProbes to dynamically instrument the L4Linux kernel at runtime. Several other options can be considered for Linux tracing. At the one hand improvements to kProbes have been proposed, for instance djProbes [Hir05], which are used to speed up a subset of kProbes called jProbes. Recent discussion on the Linux kernel mailing list [Hir06] proposes further speedups for kProbes and kRetProbes. The corresponding patches have been merged into the Linux kernel and may be adapted for L4Linux as soon as it is moved to the next kernel version.

   I furthermore plan to port the Next Generation Linux Tracing Toolkit (LTTng) [Des06] to L4Linux. This will enhance usability of process tracing in L4Linux and the LTTng

[1] As an alternative, Magpie can be altered to parse the XML file directly instead of the event description.

visualizer can then be used with L4Linux applications. In addition to the LTT kernel module that makes kernel events available to a user-level daemon, L4Linux will enable us to have an interface between LTT and the Ferret monitoring framework. Thereby monitors can even collect LTT data from several instances of L4Linux running in parallel.

The `ferretlx` kernel module for native Linux will be extended to support multiple list, scalar and histogram sensors in a manner similar to Ferret running on top of DROPS, because this provides a handy way of instrumenting L4Linux and native Linux using the same kind of sensors.

4. *IPC patterns*: Future use of the means developed for my thesis will show whether there are specific patterns of communication between DROPS components. If such patterns are found to be essential, the Fiasco IPC path can be extended to support them more efficiently.

   One example for this is a *multicast notification pattern* I observed within the ORe network switch: When a multicast or broadcast ethernet packet arrives, this packet is sent to all clients by sending their worker threads a notification message. Currently I use one intra-task short IPC for each notification. An optimization might be to provide an `ipc_multicast` system call delivering the notifications to a set of threads at once.

5. *Model-carrying code* [SVB+03] is an approach to provide safe execution of applications. Each program comes with a state machine describing the type and sequence of system calls issued by the application. A local policy enforcer can then check if this specification fits its local security policy, and an execution monitor is able to verify that the application matches its specification at runtime.

   Model-carrying code can be implemented on top of DROPS using the instrumentation techniques developed during my thesis. It can be used to determine communication patterns between an application and certain service providers. An online monitor can then use instrumentation to verify that the application adheres to these patterns.

6. *Pinpoint* [CKF+02] is a framework that uses internal and external monitoring of a system to dynamically detect faulty software and hardware components. Its approach can be adapted to be used to improve dependability of DROPS modules.

Bibliography

[Aig] Ronald Aigner. http://www.tudos.org/~ra3.

[Aig01] Ronald Aigner. The development of an IDL-compiler for micro-kernel based components. Diploma thesis, Technische Universität Dresden, Lehrstuhl für Betriebssysteme, 2001.

[AMW⁺03] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance debugging for distributed systems of black boxes. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 74–89, New York, NY, USA, 2003. ACM Press.

[Ara] Arachne - dynamic aspect execution. http://www.emn.fr/x-info/arachne/index.html.

[ASAWM99] Ehab S. Al-Shaer, Hussein M. Abdel-Wahab, and Kurt Maly. HiFi: A new monitoring architecture for distributed systems management. In *International Conference on Distributed Computing Systems*, pages 171–178, 1999.

[Aspa] AspectC. http://www.cs.ubc.ca/labs/spl/projects/aspectc.html.

[Aspb] Aspicere C code weaver. http://users.ugent.be/~kdschutt/aspicere.

[Aspc] The AspectC++ project. http://www.aspectc.org.

[Aspd] AspectJ homepage. http://www.aspectj.org.

[Asp06] Personal communication with olaf spinczyk, February 2006.

[AvD00] Joost Visser Arie van Deursen, Paul Klint. Domain-specific languages: An annotated bibliography. http://homepages.cwi.nl/~arie/papers/dslbib/, 2000.

[Bal97] Andre D. Balsa. Linux benchmarking - article series in the Linux Gazette. http://linuxgazette.net/issue22/bench.html, 1997.

[Bat88] Peter Bates. Debugging heterogeneous distributed systems using event-based models of behavior. In *PADD '88: Proceedings of the 1988 ACM SIGPLAN and SIGOPS workshop on Parallel and distributed debugging*, pages 11–22, New York, NY, USA, 1988. ACM Press.

[BCS$^+$04] Bruno De Bus, Dominique Chanet, Bjorn De Sutter, Ludo Van Put, and Koen De Bosschere. The design and implementation of fit: a flexible instrumentation toolkit. In *PASTE '04: Proceedings of the ACM-SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 29–34, New York, NY, USA, 2004. ACM Press.

[BDIM04] Paul T. Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using magpie for request extraction and workload modelling. In *OSDI*, pages 259–272, 2004.

[BIMN03] Paul T. Barham, Rebecca Isaacs, Richard Mortier, and Dushyanth Narayanan. Magpie: Online modelling and performance-aware systems. In *HotOS*, pages 85–90, 2003.

[Bra] Tim Bray. Bonnie - a unix filesystem benchmark. http://www.textuality.com/bonnie/.

[Böt] Alexander Böttcher. http://www.tudos.org/~ab764283.

[C4] C4 - the CrossCutting C Compiler. http://c4.cs.princeton.edu.

[C406] Personal communication with marc e. fiuczynski, January 2006.

[CKF$^+$02] M. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic, internet services, 2002.

[CKF$^+$04] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot–a technique for cheap recovery, 2004.

[CKFS01] Yvonne Coady, Gregor Kiczales, Mike Feeley, and Greg Smolyn. Using AspectC to improve the modularity of path-specific customization in operating system code. In Volker Gruhn, editor, *ESEC'01*. ACM Press, 2001.

[Coh05] William Cohen. Gaining insight into the Linux kernel with kprobes. http://www.redhat.com/magazine/005mar05/features/kprobes/, 2005.

[CSL04] Bryan Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic instrumentation of production systems. In *USENIX Annual Technical Conference, General Track*, pages 15–28. USENIX, 2004.

[CW76] H. J. Curnow and Brian A. Wichmann. A synthetic benchmark. *Comput. J.*, 19(1):43–49, 1976.

[Des06] Mathieu Desnoyers. The LTTng usertrace package. http://ltt.polymtl.ca/svn/ltt-usertrace/README, 2006.

[DFL$^+$05] Rémi Douence, Thomas Fritz, Nicolas Loriant, Jean-Marc Menaud, Marc Ségura-Devillechaise, and Mario Südholt. An expressive aspect language for system applications with arachne. In *Proceedings of the 4th international conference on Aspect-oriented software development*, Chicago, USA, March 2005. ACM Press.

[Dun01] Adam Dunkels. Minimal tcp/ip implementation with proxy support. http://www.sics.se/~adam/thesis.pdf, 2001.

[Dun03] Adam Dunkels. Full TCP/IP for 8-bit architectures. In *Proceedings of The First International Conference on Mobile Systems, Applications, and Services (MOBISYS '03)*, San Francisco, California, May 2003.

[Döb05a] Björn Döbel. Improving system performance using application-level hints. Großer Beleg, Technische Universität Dresden, Lehrstuhl für Betriebssysteme, 2005.

[Döb05b] Björn Döbel. ORe - a software network switch for l4. not yet published, 2005.

[ECC01] Dawson R. Engler, David Yu Chen, and Andy Chou. Bugs as inconsistent behavior: A general approach to inferring errors in systems code. In *Symposium on Operating Systems Principles*, pages 57–72, 2001.

[FCE05] William Cohen Hien Nguyen Martin Hunt Jim Keniston Brad Chen F. Ch. Eigler, Vara Prasad. Architecture of systemtap: a Linux trace/probe tool. http://sourceware.org/systemtap/archpaper.pdf, 2005.

[FGCW05] Marc E. Fiuczynski, Robert Grimm, Yvonne Coady, and David Walker. patch (1) considered harmful. In *HotOS*, 2005.

[fry05] The Frysk execution analysis tool. http://sourceware.org/frysk, 2005.

[GLM05] Jan Glauber, Jochen Liedtke, and Frank Mehnert. Fiasco kernel debugger manual. http://os.inf.tu-dresden.de/~fm3/doc/fiasco/manual.pdf, 2005.

[gra] GraphViz - graph visualization software. http://www.graphviz.org.

[Gro03] Object Management Group. Model-driven architectures. http://www.omg.org/mda/, 2003.

[Ham97] C. Hamann. The quantitative specification of jitter constrained periodic streams, 1997.

[Hel] Christian Helmuth. http://www.tudos.org/~ch12.

[Hir05] Masami Hiramatsu. Overhead evaluation of kprobes and djprobe (direct jump probes). http://lkst.sourceforge.net/docs/probes-eval-report.pdf, 2005.

[Hir06] Masami Hiramatsu. kprobes boosting explained on the Linux kernel mailing list. http://www.ussg.iu.edu/hypermail/linux/kernel/0601.3/1975.html, 2006.

[IBM] IBM. The Autonomic Computing Manifesto. http://www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf.

[JLSU87] Jeffrey Joyce, Greg Lomow, Konrad Slind, and Brian Unger. Monitoring distributed systems. *ACM Trans. Comput. Syst.*, 5(2):121–150, 1987.

[KC03] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[Kop97] Hermann Kopetz. *Real-Time Systems, Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, 1997.

[Kri05] R. Krishnakumar. Kernel korner: kprobes-a kernel debugger. *Linux J.*, 2005(133):11, 2005.

[Lac] Adam Lackorzynski. http://www.tudos.org/~adam.

[Lac02] Adam Lackorzynski. L4Linux on L4Env. Großer Beleg, Technische Universität Dresden, Lehrstuhl für Betriebssysteme, 2002.

[Lac04] Adam Lackorzynski. L4Linux porting optimizations. Diploma thesis, Technische Universität Dresden, Lehrstuhl für Betriebssysteme, 2004.

[LCM$^+$05] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, New York, NY, USA, 2005. ACM Press.

[LHR] Jork Löser, Hermann Härtig, and Lars Reuther. A streaming interface for real-time interprocess communication.

[Lie93] J. Liedtke. Improving IPC by kernel design. In *Proceedings of the 14th ACM Symposium on Operating System Principles (SOSP)*, pages 175–188, Asheville, NC, December 1993.

[Lie95] Jochen Liedtke. Improved address-space switching on pentium processors by transparently multiplexing user address spaces. Technical Report 933, GMD, November 1995.

[Lie96a] J. Liedtke. L4 reference manual (486, Pentium, PPro). Arbeitspapiere der GMD No. 1021, GMD — German National Research Center for Information Technology, Sankt Augustin, September 1996. Also Research Report RC 20549, IBM T. J. Watson Research Center, Yorktown Heights, NY, September 1996.

[Lie96b] Jochen Liedtke. Colorable memory, 1996.

[Mic06] Microsoft Corporation. Windows 2000: Overview of performance monitoring. `http://www.microsoft.com/technet/prodtechnol/Windows2000Pro/reskit/part6/proch27.mspx`, 2006.

[Moo01] Richard J. Moore. A universal dynamic trace for Linux and other operating systems. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pages 297–308, Berkeley, CA, USA, 2001. USENIX Association.

[MSC$^+$05] Chris Matthews, Owen Stampflee, Yvonne Coady, Jonathan Appavoo, Marc E. Fiuczynski, and Robert Grimm. Hey ... you got your paradigm in my operating system! In *Proceedings of the 2nd ECOOP Workshop on Programming Languages and Operating Systems, Glasgow, UK*, 2005.

[Nie99] David C. Niemi. Unixbench - a unix system benchmark. `http://www.tux.org/pub/tux/benchmarks/System/unixbench/`, 1999.

[NS03] Nicholas Nethercote and Julian Seward. Valgrind: A program supervision framework. *Electronic Notes in Theoretical Computer Science*, 89(2), 2003.

[Pie04] Matt Pietrek. A series of blog entries related to work with event tracing for windows (etw). http://blogs.msdn.com/matt_pietrek/archive/2004/09/16/230700.aspx, 2004.

[Poh] Martin Pohlack. http://www.tudos.org/~mp26.

[Poh04] Martin Pohlack. The rt_mon monitoring framework. Implementation of a monitoring framework, 2004.

[Poh06] Martin Pohlack. The ferret monitoring framework. not yet published, 2006.

[Rie05] Torvald Riegel. A generalized approach to runtime monitoring for real-time systems. Diploma thesis, Technische Universität Dresden, Lehrstuhl für Betriebssysteme, 2005.

[SAG04] Universität Karlsruhe System Architecture Group, Dept. of Computer Science. L4 experimental kernel reference manual, version x.2. http://l4ka.org/projects/pistachio/l4-x2-r5.pdf, 2004.

[SE94] Amitabh Srivastava and Alan Eustace. Atom: a system for building customized program analysis tools. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 196–205, New York, NY, USA, 1994. ACM Press.

[SS97] M. I. Seltzer and C. Small. Self-monitoring and self-adapting operating systems. *Proceedings of the Sixth workshop on Hot Topics in Operating Systems*, 1997.

[Stö05] Jan Stöß. Using operating system instrumentation and event logging to support user-level multiprocessor schedulers. Diploma thesis, System Architecture Group, University of Karlsruhe, Germany, March 24 2005.

[SVB$^+$03] R. Sekar, V. Venkatakrishnan, S. Basu, S. Bhatkar, and D. DuVarney. Model-carrying code: A practical approach for safe execution of untrusted applications, 2003.

[Tha00] H. Thane. Monitoring, testing and debugging of distributed real-time systems, 2000.

[TUDa] Operating Systems Group Technische Universität Dresden. The l4 environment. www.tudos.org/l4env.

[TUDb] Operating Systems Group Technische Universität Dresden. L4Linux. http://www.tudos.org/L4/LinuxOnL4/.

[UB05] AG Programmiersprachen Übersetzer und Softwaretechnik Universität Bremen, Fachbereich Mathematik/Informatik. Homepage of the uDrawGraph visualization tool. `http://www.informatik.uni-bremen.de/uDrawGraph/`, 2005.

[Wei84] Reinhold P. Weicker. Dhrystone: a synthetic systems programming benchmark. *Commun. ACM*, 27(10):1013–1030, 1984.

[Wei03] Alexander Weigand. Tracing unter L4/Fiasco. Großer Beleg, Technische Universität Dresden, Lehrstuhl für Betriebssysteme, 2003.

[YD00] Karim Yaghmour and Michel Dagenais. Measuring and characterizing system behavior using kernel-level event logging. In *USENIX Annual Technical Conference, General Track*, pages 13–26, 2000.