An ATM Driver for DROPS

Uwe Dannowski
Uwe.Dannowski@inf.tu-dresden.de

Dresden University of Technology

July 14, 1998

Diese Arbeit entstand im Rahmen des Großen Belegs im Studiengang Informatik an der Technischen Universität Dresden.

Institut:	Betriebssysteme, Datenbanken und Rechnernetze
Lehrstuhl:	Betriebssysteme
Betreuender Hochschullehrer:	Prof. Dr. Hermann Härtig

Contents

List of Figures

1 Introduction

1.1 Motivation

The Dresden Real Time Operating System (Drops) project aims at providing applications with "Quality of Service" (QoS) support from the operating system. To this end, based on the L4 μkernel, a multi server environment is being developed. The Drops project attempts to develop design techniques for the construction of distributed real time systems where each component can guarantee a certain quality of service to applications.

To be able to provide QoS support even in networking environment, ATM is the network of choice for the basic communication.

All new operating systems suffer from missing applications and hardware support. Hardware and software manufacturers often support only few systems of commercial importance. There are mainly two approaches to overcome this problem: developing the needed components from scratch or porting existing code to the new system. The former could possibly lead to highly efficient code for the target system but means inventing the wheel once again. Porting existing code to a new system can be facilitated by means of a framework emulating the source system's functionality using the target system's resources. Once this framework exists - providing compatibility on either the binary or the source code level - a number of applications becomes available for the new system. When using this scheme for device drivers, the respective hardware is available, too.

With L4Linux Drops demonstrated that Linux as a whole can be ported to a new platform - the L4 μkernel - by (mainly) modifying only the architecture dependent part of the Linux kernel. L4Linux provides binary compatibility with Linux thus offering the ability to run existing Linux applications. The L4Linux system itself with its monolithic kernel is not a real time system. But it is possible to run real time tasks besides L4Linux on top of L4 - better to run L4Linux not influencing real time tasks. This way the Drops project combines QoS support for real time tasks with full Linux functionality.

As stated above, L4Linux and hence the device drivers in its monolithic kernel are unable to provide reliable services with guaranteed parameters for real time tasks. Therefore, efforts are spent on isolating some of Linux' device drivers into separate L4 tasks (servers). When isolated, these servers are potentially able to give guarantees in respect of bandwidth, memory or CPU utilization.

This work aims at porting the existing Linux driver for FORE Systems' ForeRunner PCA-200E ATM network board to L4. The result should be an L4 server providing a smart interface to access the PCA-200E's hardware and a Linux stub to be used with this server.

1.2 Synopsis

Section 2 gives an overview about related work - the two systems L4 and Linux with ATM-on-Linux. In addition, the PCA-200E Linux driver is described. Section 3 shows the design of the targeted L4 server with its IPC interface. In Section 4, some implementation details are pointed out and the IPC interface as well as the client library are introduced. Performance measurement results are presented in Section 5. Section 6 summarizes this work and discusses the scope for further work.

The reader is assumed to be familiar with basic ATM principles and vocabulary.

2 Related Work

This section briefly introduces the projects this work is based on: the L4 μkernel, Linux with ATM-on-Linux and - slightly more detailed - the PCA-200E Linux driver.

2.1 L4

L4 is a μkernel which has been developed by Jochen Liedtke at the National Research Center for Information Technology (GMD) and IBM Watson Research Center. The initial version of L4 runs on Intel's 80x86 - meanwhile there are implementations available for DEC-Alpha from the Dresden University of Technology and for MIPS from the University of New South Wales.

As the name "μkernel" implies, L4 offers only:

- fast, message based inter process communication (IPC)
- page based memory management
- priority based scheduling with hard priorities
- tasks as security domains

External pagers allow implementation of almost any desired memory management outside the kernel. This minimalistic design offers greatest flexibility at high speed.

The following definitions for L4 primitives are an excerpt of [Lie96]:

IPC Interprocess communication in L4 is message based and takes place between exactly two threads. For a message to be transferred both parties must agree to the transfer. Message transfers happen always synchronously. There are mainly two types of messages - short messages and long messages. A short message consists of words where (the size of a word and) the maximum number of words is architecturally dependent. Short messages are transported entirely in the processor's register set. Hence, short messages provide the fastest communication path. Long messages use a message descriptor (message dope) located in memory. By use of long messages any number of words can be copied between the IPC partners, either located in the message dope itself or referenced in the message dope. When a message is marked having Flexpages, the words of the message are interpreted as Flexpage descriptors.

Address spaces An address space is a mapping which associates each virtual page to a physical frame or marks it non-accessible. Address spaces can be manipulated by sending Flexpages in IPC messages. L4 supports recursive creation of address spaces outside the kernel. But, to prevent corruption of address spaces all changes must be controlled by the kernel.

Flexpages Flexpages are regions of the virtual address space, consisting of all pages mapped in this region. Sending a Flexpage by means of IPC adds all the pages currently mapped in this Flexpage to the destination's address space.

Threads A thread is an activity, being characterized by some kind of state information (registers, instruction and stack pointer, priority, ...) and an associated address space. L4's scheduling works on thread level.

Tasks A task is the entirety of an address space and all threads (active or inactive) executing in this address space. Moreover a task is also a protection domain for IPC.

Interrupts In L4 hardware interrupts are translated into an IPC message to a certain thread. A thread can register with a hardware interrupt to be notified when that interrupt occurs.

2.2 Linux

Linux is a freely-distributed UNIX-like operating system, originally created by Linus Torvalds. Developed under the GNU General Public License, the source code for Linux is available to everyone. With the freedom to create and adapt Linux for a wide variety of platforms, Linux has become quite popular worldwide for business and personal use. Lots of developers worldwide contribute to the Linux project by creating applications and extending the kernel (device drivers, protocol stacks, etc.).

From the system developers view Linux is an operating system with a monolithic kernel. According to [Sta96] in [Tan90] a monolithic kernel is defined to be a collection of procedures with each procedure being allowed to call any other procedure if needed. This results in a quite intricate structure making isolation of a single component more difficult. As there's no protection inside the Linux kernel itself, a failure in *one* procedure may influence others, and hence, the whole kernel might become instable due to a single failing component.

2.3 ATM on Linux

ATM-on-Linux adds ATM networking support to Linux and is being maintained by Werner Almesberger at Ecole polytechnique fédérale de Lausanne (EPFL). ATM-on-Linux offers support for PVCs, SVCs, Classical IP, LAN Emulation (emulates services of existing LANs across an ATM network) and Arequipa (experimental mechanism to create short-cut ATM SVCs for IP traffic). The API to access the ATM-related services is the slightly extended well-known Socket API (see [Alm96a] for details). The interface to ATM-on-Linux' device drivers is well structured and defined in [Alm96b].

Adding support for a new ATM network board mainly means providing a set of board-specific functions acting as handlers for protocol requests (`open`, `close`, `send`, etc.). If the network board uses interrupts, also an interrupt handler function must be provided.

2.4 PCA-200E Linux Driver

The Linux driver for FORE Systems' *ForeRunner PCA-200E* is designed to work with ATM-on-Linux (Fig. 1). Therefore its upper interface obeys the Linux ATM Device Driver Interface [Alm96b]. The driver accesses the hardware via the AALI [FOR97], which actually is a communication protocol to the firmware running on the board's control processor. Some of the operations provided by the AALI have semantics similar to the functions required by ATM-on-Linux.

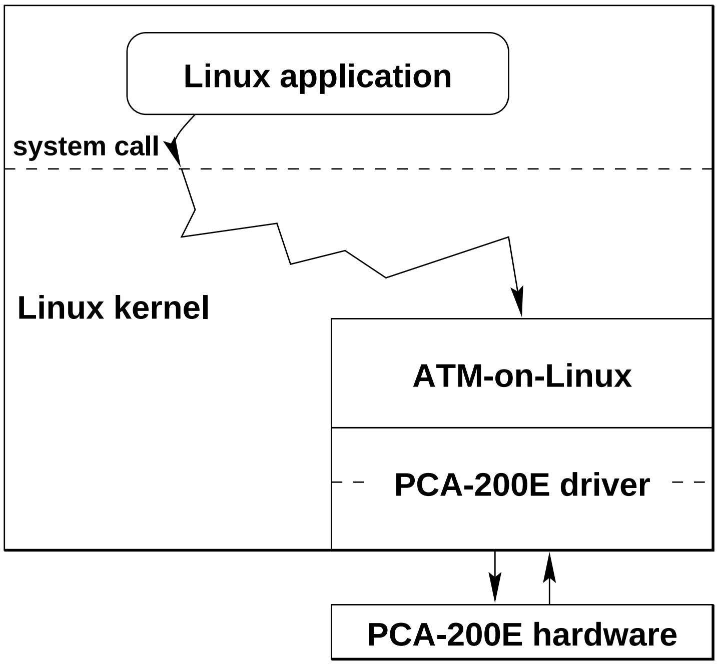

Figure 1: PCA-200E driver in Linux

2.4.1 High Level Functions

In ATM-on-Linux connections are represented by a `atm_vcc` structure (see Appendix 7.2). A physical ATM device is represented by an `atm_dev_t` structure. In Linux' networking code, a piece of data is referenced by a socket buffer structure (`skb`). Registering a physical device in ATM-on-Linux mainly means calling the function `atm_dev_register` with a structure containing function pointers (see Section 7.1).

In the PCA-200E driver only `open`, `close`, `send` and `ioctl` are implemented. `open` enables transmission/reception of cells on the specified VPI/VCI, `close` disables transmission/reception for the given VPI/VCI. `send` is called to actually send data referenced by the buffer descriptor `skb` over the connection described by the connection descriptor `vcc`. The `ioctl` function provides a way to monitor/control the board's behaviour. Reception of incoming data from the network is handled in the interrupt service function (which is not a member of the `ops` structure). Received data is put into an `skb` and then `push`, a member function of the corresponding `vcc`, is called to push the data upwards in the protocol stack.

2.4.2 The AALI

The ATM Adaption Layer Interface (AALI) is the programming interface for the *ForeRunner 200* series hardware [FOR97] and is implemented by the firmware running on the board. The interface consists mainly of several queue structures and a few special function registers, all located in a shared memory region provided by the network board.

Queues are simply an array of queue entries. Each queue forms a logical ring list by wrapping around after the last array element to the first. Queue lengths are configurable at initialization time. The queues in detail are:

- **Command queue**
  The command queue is used to send commands to the firmware:
  `activate_vci` (open), `deactivate_vci` (close), `request_stats`, `zero_stats`, etc.

- **Transmit queue**
  To send data, a Transmit PDU Descriptor (TPD, contains location/length of data in host memory, ATM cell header, ...) located in host memory is filled and then its address is written to the transmit queue.

- **Receive queue**
  The receive queue holds references to Receive PDU Descriptors (RPD). These RPDs are located in host memory and are written to by the firmware on reception of data from the network. An RPD holds references to buffers containing the received data in host memory.

- **Buffer queues**
  Via the buffer queues new receive buffer descriptors are supplied to the firmware. There are two buffer pools, each with two buffer sizes, giving four buffer queues.

3 Design

The main objective of this work is to port the Linux driver to L4 without major changes to the code. Therefore, Linux kernel functions used by the Linux driver need to be emulated on L4. Affected are memory and time management as well as interrupt handling. Emulation of the required functionality can be done by means of of a framework.

The existing Linux driver can be seen as being divided into two layers: low level functions directly dealing with hardware access and high level functions offering the service required by the ATM-on-Linux Device Driver Interface.

As the L4 driver should conceal hardware accesses from its clients, it offers access to the low level functions through an IPC interface. Figure 2 shows how the PCA-200E L4 driver is embedded in Drops. It will serve the ATM protocol component. When this work started, the ATM protocol component was not even designed.

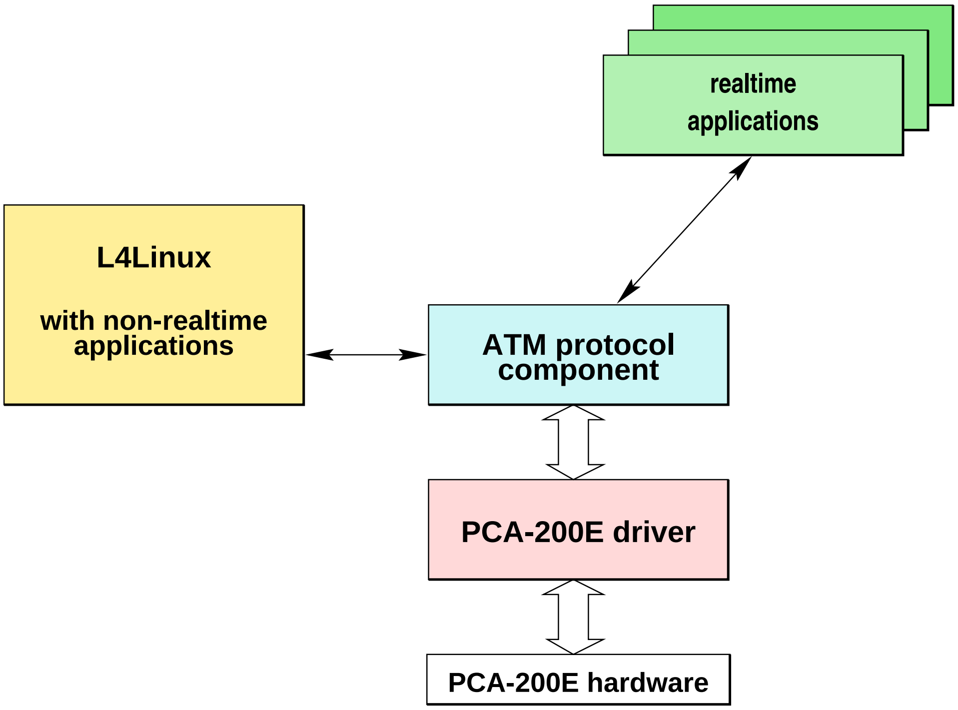

Figure 2: PCA-200E L4 driver in Drops

3.1 Framework

This section describes how the required Linux functionality is implemented using L4 resources.

3.1.1 Memory Management

The Linux driver makes use of the following Linux kernel functions: `vremap()`, `kmalloc()`, `kfree()`, `__get_dma_pages()` and `free_pages()`.

`vremap()` maps a region of memory into the Linux kernel's virtual address space, thus making it accessible for the driver. It returns the virtual address of the region. On L4 this operation can be easily performed by mapping the requested range of physical memory to an arbitrary position in the driver task's address space.

Linux' `kmalloc()` allocates a 4 byte aligned memory area of the requested size and returns that area's base address which equals its physical address.

`__get_dma_pages()` allocates contiguous pages (4KB) located in the first MB of physical memory (due to the ISA DMA limitation). The virtual address of each of these pages is equal to their physical addresses. In the Linux driver, `__get_dma_pages()` is called to allocate memory which is at least 32 bytes aligned. The same functionality can be achieved on L4 by using a suitable implementation of `kmalloc()`. `__get_dma_pages` is a macro for `__get_free_pages` with an option to request DMA-able pages.

Since all memory allocation takes place during initialization of the driver, freeing memory would be required only on L4 driver shutdown. Shutting down the driver can be done by flushing the whole driver task, which in turn frees all memory "allocated" by this task.

In summary, the memory management required by the L4 driver can be reduced to simple pointer arithmetics over a fixed size area of contiguous memory.

3.1.2 Time Management

The Linux driver does busy waiting during initialization, especially while it polls for completion of the INIT command. In order to avoid blocking of the whole Linux system in case of a firmware initialization failure, there is a upper limit for the duration of the INIT command. This limit is ensured by repeatedly polling the global variable `jiffies` which is incremented on each timer interrupt.

In the Linux driver, busy waiting is also used with the command queue. When a control command is sent to the board, the driver waits until the command has completed. Although it would be possible to trigger an interrupt when command execution has finished, this feature is not used. This is mainly due to historical reasons: during the early days of the Linux driver, repeatedly polling a memory location was a lot easier than waiting for an interrupt. Requesting generation of interrupt on command completion would require parsing of the command queue on each interrupt. Under normal circumstances, the time spent on waiting for command completion can be neglected.

The global variable `jiffies` in the Linux kernel can be represented by an L4 task global variable in the L4 driver. This variable could be incremented by a separate thread that repeatedly sleeps for a certain time using sleep-IPC.

3.1.3 Interrupt Handling

The Linux kernel provides a convenient way to install an interrupt handler by using the function `request_irq()`. As one of this function's parameters, a reference to the interrupt handler function is specified. The interrupt handler function is called each time an interrupt arrives on the requested interrupt line. Thus, on arrival of an interrupt, the currently executing kernel activity is suspended until the interrupt handler function returns.

L4 converts an interrupt occuring into an IPC message to a certain thread. The message is send to the thread that has previously successfully applied for being notified on arrival of the interrupt.

However, the behaviour of Linux can be emulated on L4 using a separate thread that associates itself to the interrupt. It then waits for the Interrupt message, and executes the interrupt handler function as usual in Linux.

Whether this interrupt thread becomes active or not depends on its priority. If there's any other thread in the driver task with a higher priority and its state is "ready", it can't be guaranteed that the interrupt handler will be executed as expected. To achieve a Linux-like behaviour, the interrupt thread should be the one with the highest priority in the driver task. Thus, all other threads working on "former kernel activities" can't become active until its work is done.

3.1.4 ATM-on-Linux Functions

In the Linux driver, there are several direct calls to functions being part of the ATM-on-Linux kernel patch, as `atm_dev_register` and `atm_dev_deregister` are. The former simply returns a pointer to a (newly allocated) `atm_dev` structure - the latter can be left empty, since shutting down the L4 driver is not planned.

Far more complex is the call structure in the receive part, especially in the function `pca200e_intrx`. This function parses the receive queue of the board and pushes received protocol data units (PDUs) upwards in the protocol stack. Firstly, the original Linux driver scans the list of connection descriptors (VCCs) of the respective device for the VPI/VCI of the received PDU. Maintaining a list of VCCs is not required in the L4 driver, since it will have one client only. Secondly, if the VCC is found, the VCC's member function `peek` is called with the size of the receive PDU as one of its parameters. `peek` is a function that allocates a socket buffer with at least the given size. In ATM-on-Linux it is a member function of the VCC, thus allowing easier adaption to the higher level protocols needs. The data chunks belonging to the received PDU are copied into the socket buffer. In the L4 driver, these two actions (requesting buffer, copying into the buffer) can be omitted; the PDU content is passed to the client through the IPC interface. Lastly, the function `pca200e_inttx` is called to free completed transmit queue entries and the associated socket buffers. On L4 only freeing of queue entries would be left over from the Linux version of `pca200e_inttx`, since VCC's are not part of the L4 driver.

To summarize this section, the receive function `pca200e_intrx` and the maintainance function `pca200e_inttx` must be changed, mainly by removing superfluous code related to VCCs and socket buffers.

3.2 IPC Interface

To offer its service to other system components the server provides an IPC interface. This interface must implement operations to access the underlying hardware.

The operations are:

```
detect
open
close
send
receive
```

The design of DROPS implies that there is only *one* client accessing the PCA-200E server. Hence, demultiplexing of received data and VPI/VCI ownership checks in `close` and `send` can be omitted. This is done in the ATM protocol component of DROPS.

detect (magic number) → (magic number, hwaddr) provides a way to check for correct initialization of the board and the driver and to obtain the board's hardware address (6 significant bytes). On initialization failure the operation might time out or deliver incorrect values. Therefore a magic number is used to verify its correct execution.

open (vpi,vci,aal) → (status) tries to open the specified VPI/VCI by enabling reassembly with the given AAL. The operation is expected to return an error if the VPI/VCI is already in use, the values for VPI or VCI are out of range or if the maximum number of connections the firmware can handle is exceeded.

close (vpi,vci) → (status) tries to close the specified VPI/VCI. The operation might return an error if the VPI/VCI has not been opened before.

send (vpi,vci,aal,data) → (status) builds an entry for the board's transmit queue and thus requests transmission of the referenced chunk of data via the given VPI/VCI and AAL. The operation reports an error in case of a filled-up transmit queue.

receive (vpi,vci,data) → (status) is not a client initiated operation as `detect`, `open`, `close` and `send` are. This is caused by the nature of networks, where reception of data (normally) is an externally triggered event. Since the PCA-200E can trigger an interrupt on arrival of data from the network, the occurring interrupt message wakes up the appropriate thread. This thread parses the receive queue and sends a message containing received data to the client.

3.3 Process Model

In this special case, there are mainly two design principles: single-threaded and multi-threaded server. This distinction depends on whether there is only one thread serving client requests *and* interrupts or more. Single-threaded constructions are most suitable when the server thread and the interrupt thread work on the same structures and thus need to be properly synchronized. Whereas multi-threaded solutions can be used if interrupt handling and client serving are not tightly coupled.

Due to the design of the PCA-200E's interface, no synchronization is required between the server thread and the interrupt thread. Hence, multi-threaded design suites best.

The PCA-200E driver for L4 is an L4 task with three threads (Fig. 3):

1. server thread
2. interrupt thread
3. timer thread

All threads run independent from each other. There is no direct interaction by means of IPC.

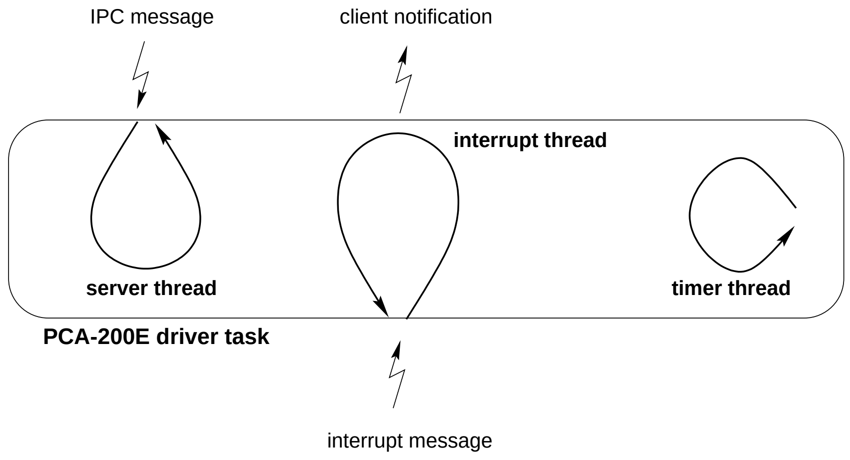

Figure 3: Threads in the L4 driver

The timer thread periodically waits using sleep-IPC (a receive operation with given receive timeout) and increments the `jiffies` variable on wakeup. It provides the timing basis for the driver.

The server thread waits for an IPC message from the “client”, where the client is the one thread that sent the first message to the server thread. The messages contain commands for the driver as described in Section 3.2. After execution of the requested operation, the server thread replies to the client with the result of the operation and waits for the next message.

The interrupt thread waits for a message from the L4 kernel, reflecting arrival of an interrupt. If such a message is received, the thread calls the interrupt handler function and waits again afterwards.

4 Implementation

4.1 Framework

The framework described in Section 3 implements the "missing" Linux functions used by the driver. This gives compatibility at the source code level. Hence, no modifications of the driver source code are required to use the Linux driver on L4. This section discusses selected implementation details of this framework.

4.1.1 Memory

The framework's memory management is initialized by a call to `mm_init(start,end)` to set up the local memory pool in the range [start,end]. `kmalloc(size)` returns the current beginning of the memory pool and increments the beginning by `size` (rounded up to the next multiple of 32 - see Section 3.1.1).

`vremap(offset, size)` establishes a mapping of the 4MB superpage containing the requested physical address `offset`. Therefore it sends a mapping request to the driver's pager which must have access to physical memory. The requested area is mapped to virtual address `offset - 1GB`.

`__get_dma_pages` tries to request the specified number of 4K-pages by repeatedly calling `get_page_from_l4`. `get_page_from_l4` requests a 4K-page from the pager.

4.1.2 Time

The framework provides the `jiffies` variable, as this is used by the driver to enforce bounding of polling delays during initialization. `jiffies` is being incremented by a dedicated thread, which is activated during `jiffies_init(lthreadno)`. The function `jiffies_thread` being executed by this thread is an endless loop - sleeping for 10ms and incrementing `jiffies`;

4.1.3 Interrupts

Interrupts from hardware are wrapped into a message to a thread by the L4 kernel. Thus, to get notified of an interrupt from the board, a thread must be waiting for an appropriate message from the kernel. In the framework thread 1 is used for that.

The framework's function `request_irq(irq,handler,flags,name,dev_id)` activates thread 1 (one) and starts the framework's interrupt handler `irq_thread(irq,handler, dev_id)`. The parameters of `irq_thread` are exactly the same as in `request_irq`, except from `flags` and `name`, which are of no use in the driver's context.

`irq_thread` attaches to the respective interrupt and waits for a message from the kernel. Upon arrival of a suitable message, the function that was passed to `irq_thread` as the `handler` parameter, is called. According to Linux conventions, its parameters are the interrupt number, the `dev_id` parameter and a pointer to the cpu register structure, where the latter one is NULL, as it is not used/needed.

For this interrupt handling to work as it does under Linux, the interrupt thread is required to have the highest priority in the driver task. This is necessary to ensure that no other action be performed in the task until the interrupt handler has finished its work.

4.1.4 ATM-on-Linux Functions

From the set of functions provided by ATM-on-Linux only `atm_dev_register` and `atm_dev_deregister` are used and implemented. The former allocates a new `atm_dev` structure, just to have one for all the driver's lowlevel functions. `atm_dev_deregister` is simply left empty, because it won't get called at all.

4.2 IPC Interface

From the client's point of view, there are two interface threads in the driver. The server thread accepts control messages and send messages, replying with code returning status. The interrupt thread sends receive messages to the client. This way, the send and receive directions are almost independent (Fig. 4).

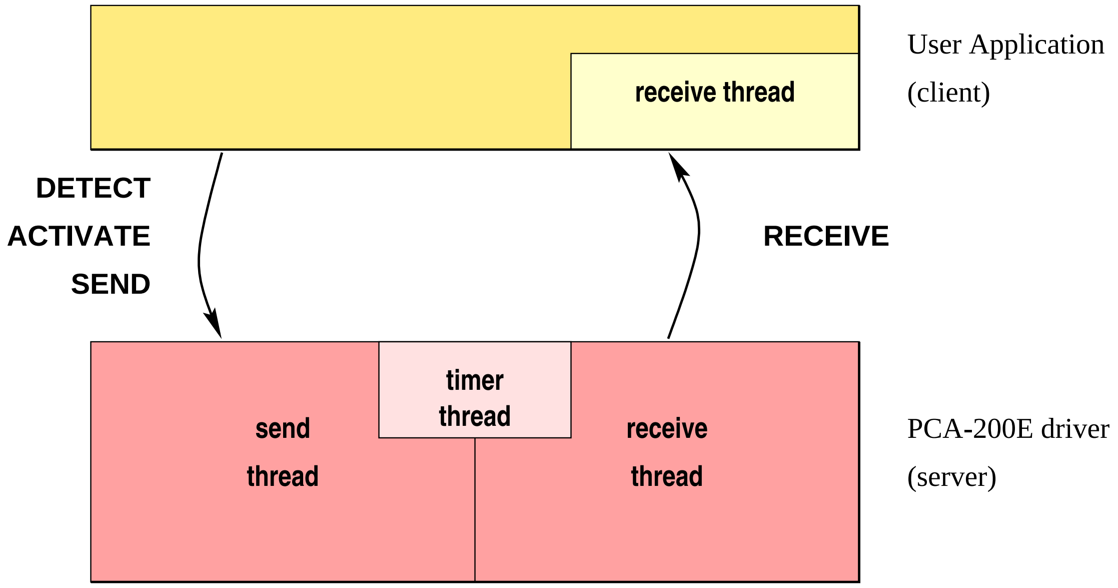

Figure 4: IPC interface

The current implementation of the interface uses "indirect strings", a feature (or curse ?) of L4's Intel-version. This was the choice for the first implementation, as indirect strings seemed to be easy to use, surely punished by a certain lack of performance (... allowing performance improvements in the future).

All messages from the client to the server have the same format. They consist of two dwords and one optional indirect string (Fig. 5):

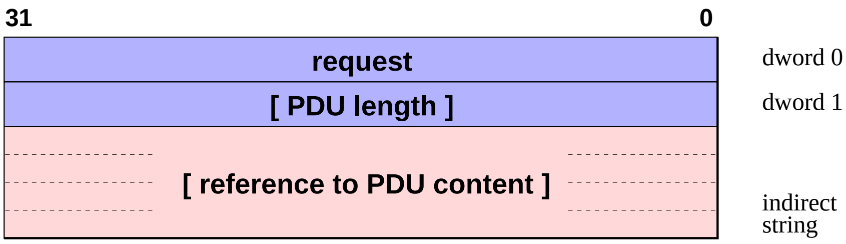

Figure 5: IPC message structure

If used, the second dword holds the length of the PDU copied in the indirect string, whereas the first dword contains the request (Fig. 6).

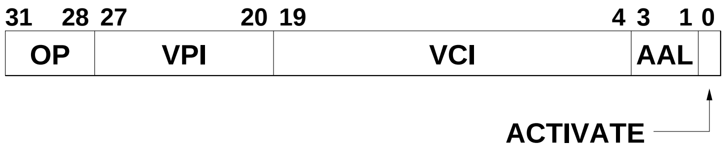

Figure 6: request member

In the following sections the current structure of the messages being passed via IPC will be discussed in detail.

4.2.1 DETECT

The DETECT request (Fig. 7) is sent to the server for two reasons. Firstly, to check if initialization of the board was successful, and secondly, to obtain the hardware address (ESI - end system identifier) of the board.

Figure 7: DETECT request

If initialization of the board was successful, the DETECT reply (Fig. 8) contains a magic number, 0x1014. The six significant bytes of the board's ESI fill up the remaining bytes.

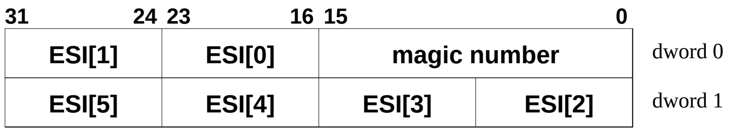

Figure 8: DETECT reply

4.2.2 ACTIVATE

The ACTIVATE request (Fig. 9) opens or closes a certain VCI. Whether reassembly for cells of this VCI is enabled or disabled, depends on the ACTIVATE flag in the request. Setting this to 1 opens the VCI by enabling reassembly of received cells with this VCI, using the AAL from the AAL field in the request. If the ACTIVATE flag is zero, a previously opened VCI is closed. Valid values for the AAL field are 0, 4 and 5. The maximum number of open connections supported by the board is 1024.

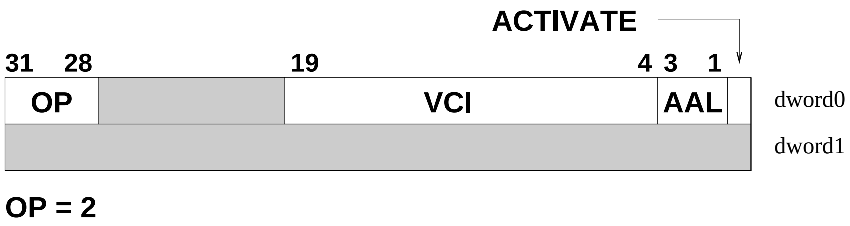

Figure 9: ACTIVATE request

The reply to an ACTIVATE message returns the result of the requested operation in the ST field (Fig. 10). A value of 1 indicates success, whereas 0 means, that the operation completed with an error. Common causes for errors are an invalid value in the AAL field or an already opened VCI.

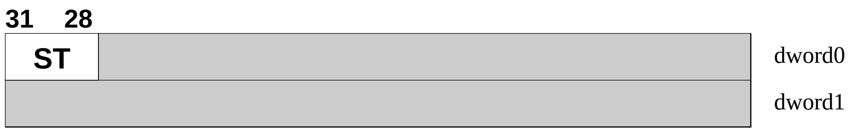

Figure 10: ACTIVATE reply

4.2.3 SEND

Using the SEND request, a client can cause the PDU content passed along with the request to be sent. The information from the fields VPI, VCI and AAL are used to build the ATM cell header. The PDU content is copied from the client's string buffer to the server. The PDU size is limited to 64KB (maximum AAL5 PDU). Owing to L4 requirements, the string buffers *must* be at least 4 bytes long. That's why the second dword of the message holds the exact PDU size in bytes (Fig. 11).

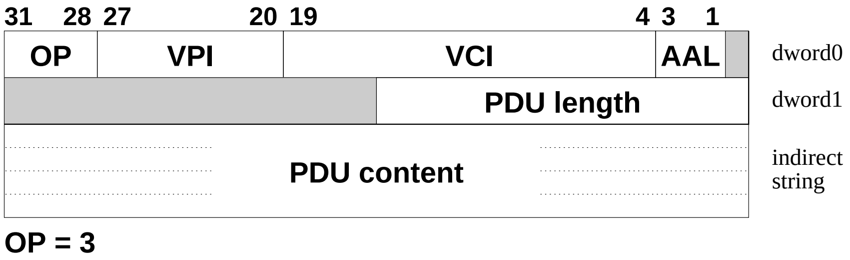

Figure 11: SEND request

The reply to the SEND request is sent back after complete emission of the PDU. The SEND reply message has the same structure and meaning as the ACTIVATE reply message.

4.2.4 RECEIVE

A RECEIVE message (Fig. 12) is sent from the server to its client, when reassembly of a PDU was successful. The first dword contains the ATM cell header provided by the firmware. Its structure is shown in Figure 13. The HEC field (the fifth byte of an ATM cell header) is missing; it is completely hidden by the firmware. Along with the exact PDU size in dword 1, all necessary information describing the PDU is transferred to the client. The PDU content is copied into the client's string buffer. The client's string buffer should have a size of at least 64 KB. This is the maximum PDU size that is copied with the RECEIVE message. It is the maximum PDU size for AAL5, too.

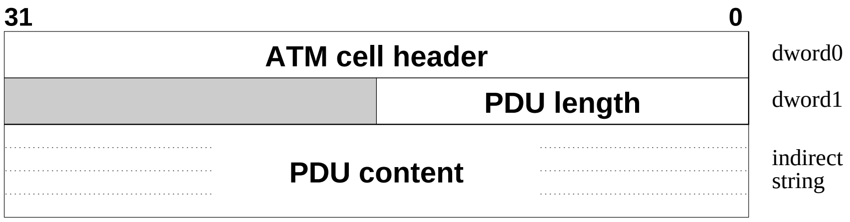

Figure 12: RECEIVE message

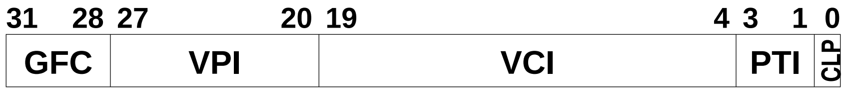

Figure 13: ATM cell header

There is no reply to a RECEIVE message; the IPC operation is performed with a send timeout of 0. In case of the client not being ready to receive the next message, this ensures nonblocking operation of the PCA-200E driver.

4.3 Client Library

The client library hides the IPC interface from the application programmer. It provides a convenient way to use the PCA-200E L4 driver's service through a C-language interface. The C functions are obviously made "compatible" with the lowlevel functions in the Linux PCA-200E driver. In the following these functions are discussed briefly.

4.3.1 pca200e_init

Declaration:

```
int pca200e_init(void (*rxfunction)(unsigned int vpi,
                                    unsigned int vci,
                                    void* p,
                                    unsigned int pdulen));
```

Parameters:

The one and only parameter `rxfunction` is a pointer to the receive function (see Section 4.3.4).

Description:

This function initializes the client library. Firstly, it tries to find the PCA-200E server task. Secondly, it sends the DETECT request, in order to register the client at the server and to obtain the board's ESI which is then stored in ESI[0]..ESI[5]. Lastly, a new thread (0x73) is activated, that waits for a RECEIVE message from the server. If this thread receives a RECEIVE message it calls the receive function.

Return value:

If initialization was successful, `pca200e_init` returns 0, otherwise the value returned is the error code returned by `rmgr_get_task_id`.

4.3.2 pca200e_activatevcin

Declaration:

```
int pca200e_activatevcin(unsigned int activate,
                         unsigned int aal,
                         unsigned int vpi,
                         unsigned int vci,
                         unsigned int mtu);
```

Parameters:

activate	0 or 1	enable or disable reassembly
aal	0, 4 or 5	the AAL to be used for reassembly
vpi	?	not used
vci	0..16383	VCI to open
mtu	?	not used

Description:

This function opens (`activate=1`) or closes (`activate=0`) the VCI `vci`. If `activate` is 1, reassembly of cells with this VCI is enabled using AAL `aal`. If `activate` is 0 a previously opened VCI is closed. Trying to reopen an already open VCI fails, except for VCIs opened for AAL0.

Return value:

If the operation completed successfully, the value returned is 1, otherwise zero.

4.3.3 pca200e_send

Declaration:

```
int pca200e_send(unsigned int vpi,
                 unsigned int vci,
                 unsigned int aal,
                 void* p,
                 unsigned int pdulen);
```

Parameters:

vpi	0 .. 255	
vci	0 .. 16383	
aal	0, 4 or 5	the AAL to be used
p		pointer to PDU content
pdulen	1 .. 65535	PDU length

Description:

The chunk of data that `p` points to is sent to the network via the given VPI/VCI, using the given AAL. `p` must point to a region of mapped memory. The region must be 4 byte aligned. The function returns *after* the PDU is emitted completely - this is is a synchronous operation.

Return value:

If the operation completed successfully, the value returned is 1, otherwise zero.

4.3.4 Receive Function Prototype

Declaration:

```
void rxfunction(unsigned int vpi,
                unsigned int vci,
                void* p,
                unsigned int pdulen);
```

Parameters:

vpi	0 .. 255	
vci	0 .. 16383	
p		pointer to PDU content
pdulen	1 .. 65535	PDU length

Description:

This is the prototype declaration for the receive function whose entry address is passed as an argument to `pca200e_init`.

Return value:

4.3.5 ESI

Declaration:

```
extern unsigned char ESI[6];
```

Description:

After a successful call to `pca200e_init`, ESI will hold the six lower bytes of the board's ESI. This information is mainly useful for address registration in the network.

4.4 Linux Driver Stub

Like the original PCA-200E Linux driver, the PCA-200E L4Linux driver is also available as a loadable module. It is used in combination with the PCA-200E L4 driver running as a stand-alone L4 task. The module was derived from the Linux driver's code by removing all lowlevel functions (these are in the L4 driver), adding the client library and a receive function. The PCA-200E L4Linux driver represents the high level part of the original PCA-200E Linux driver.

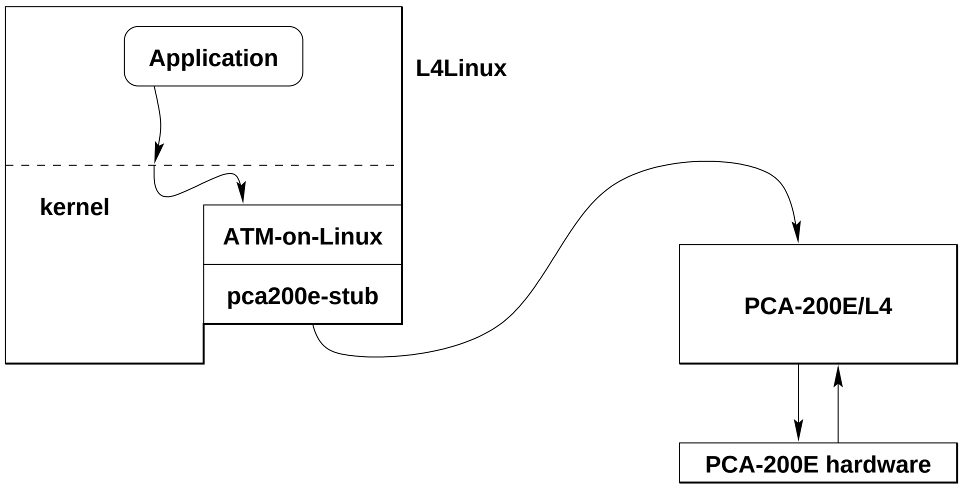

Figure 14: Architecture overview using the Linux driver stub

When an L4Linux applications is going to use the ATM services, the request finally arrives in the PCA-200E L4Linux driver. Here the request is sent via IPC (hidden in the client library) to the PCA-200E L4 server which in turn handles the request. Replies and receive messages travel along this path in the opposite direction (Fig. 14).

5 Performance

In the following sections, the performance of the components as well as some optimization steps are discussed.

5.1 Theory

Before evaluating any measurements, it is necessary to explore the achievable values.

Since the PCA-200E used in this work has an SONET STS-3c interface, it works with an STS-3c stream of 155.52 Mbps [ATM94]. With respect to SONET framing overhead, a continuous byte stream of 149.76 Mbps is available, which is tightly filled with cells of 53 bytes. Thus, the available cell rate is 353207.55 cells/s. Every cell has a header of 5 bytes, leaving 48 bytes per cell for payload, which gives a maximum bandwidth of 135.63 Mbps of user data.

The value $T_{PDU}(s)$ is the time it takes to emit an AAL5-PDU of size s. From that, the achievable throughput $B(s)$ can be calculated.

$$T_{PDU}(PDUsize) = \frac{cells\ per\ PDU}{line\ cellrate} = \frac{\left\lceil \frac{PDUsize+8}{48} \right\rceil cell}{353207.55 \frac{cell}{s}} \tag{1}$$

$$B(PDUsize) = \frac{PDUsize}{T_{PDU}(PDUsize)} \tag{2}$$

5.2 Measurement Setup

Measurements were done on a machine with the following hardware configuration:

- Intel Pentium 133 MHz
- Asus Mainboard P/I 55TP4N, 256KB Cache
- 32MB EDO RAM
- FORE Systems PCA-200E ATM Network Adapter, running ForeThought 4.1 firmware

The measurement software setup consists of a test application (*client*), the PCA-200E L4 driver (*server*) and the Resource Manager. The client has the client library (discussed in Section 4.3) statically linked with it.

5.3 Send Performance

The test application tries to send data as fast as possible using the `pca200e_send` function in the client library. The time spent in the send operation is measured by means of the `rdtsc` instruction.

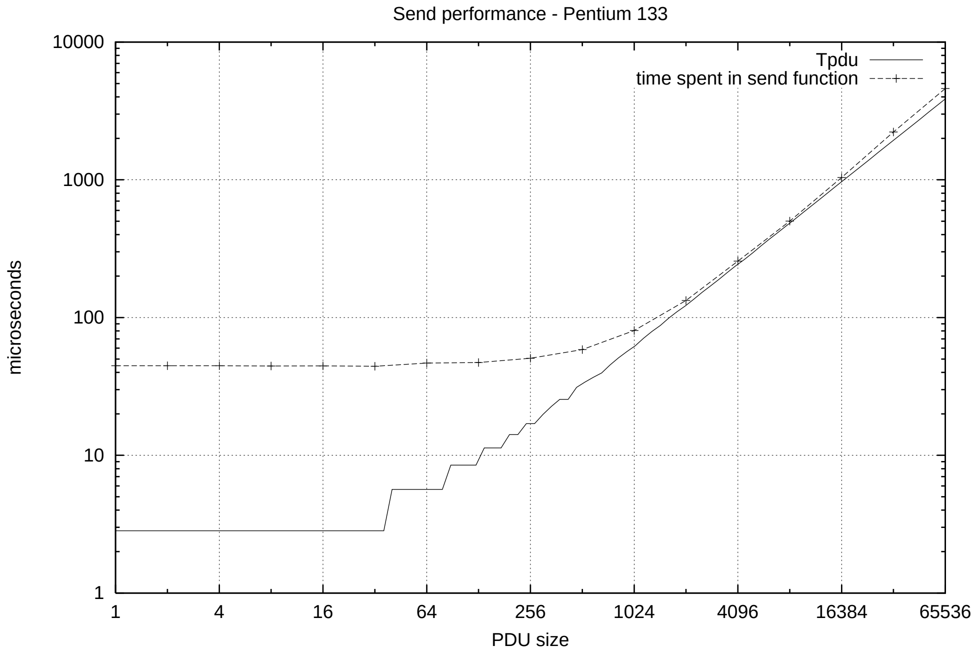

Figure 15: send call costs

In Figure 15, the contiuous line shows the time spent in the send function of the client library in order to send a PDU of the given size. This duration covers the time for the send request IPC, to send the PDU out through the AALI and for the reply IPC. As shown, the overhead is significant for small PDUs. It is almost neglectable for PDU sizes of about 8 Kbyte and grows for larger PDUs. This asymptotic slope results from the way the PDU content is transferred to the PCA-200E driver. Using indirect strings to copy the whole PDU content into the driver's address space consumes a remarkable amount of time, presumably introduced by cache influences. The resulting throughput is shown in Figure 16.

The throughput reduction for larger PDUs lead to two optimization steps. Due to the dynamic development of DROPS, several general decisions regarding the DROPS design have been made when this work was almost finished. So it can be assumed, that the client of the PCA-200E L4 driver has knowledge of the physical address of the buffer holding the PDU content. Thus, it is no longer necessary to copy the PDU contents into a buffer whose physical address is known to the PCA-200E L4 driver. It is sufficient to pass the address along with the length of the PDU to the driver.

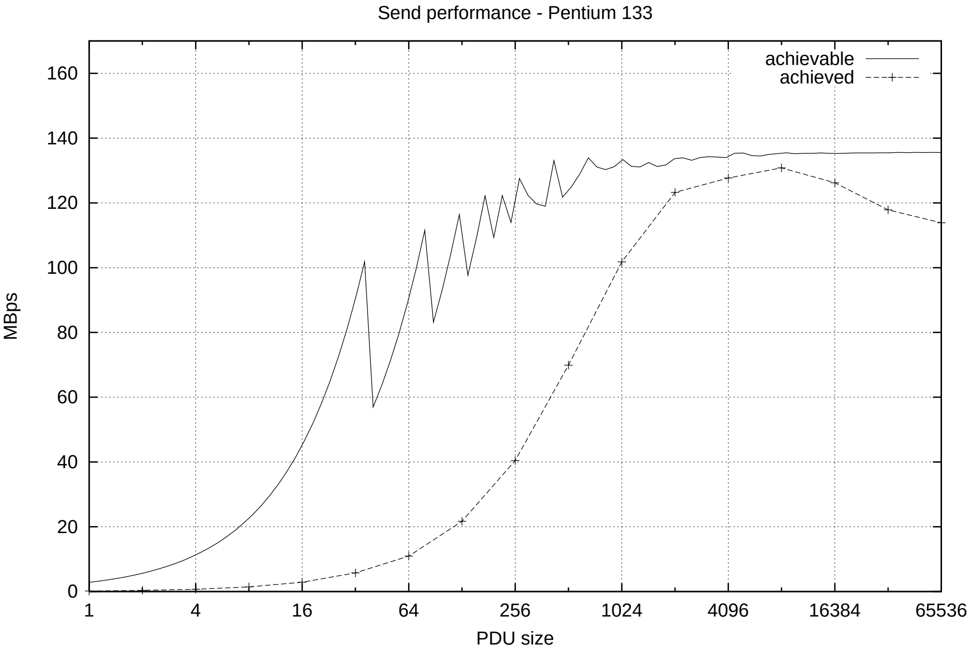

Figure 16: send throughput

As a first cautious step, instead of the PDU contents only the physical address was transferred in the string buffer. Thus, the time needed to pass the send message to the driver was constant. As a second optimization, the address of the PDU buffer was no longer copied in an indirect string, but in an additional dword. This lead to a remarkable reduction of the overhead.

The effects of these two optimization steps are shown as “Protocol 2” and “Protocol 3” in the following figures. “Protocol 2” eliminated the additional overhead for large PDUs whereas “Protocol 3” reduced the general SEND message overhead.

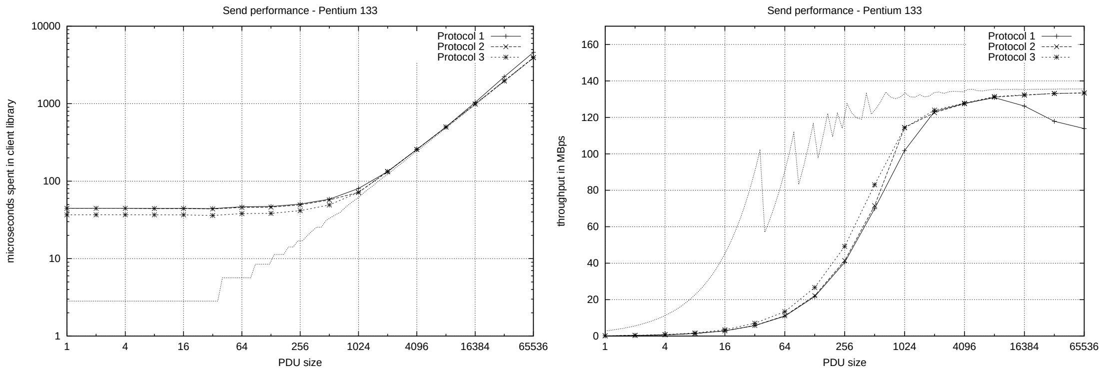

Figure 17: send performance - optimized

5.4 Receive Performance

Traffic is generated on a second machine connected via a cross-link cable, running a specialized version of the PCA-200E L4 driver. There the overhead of the IPC interface is eliminated by directly inserting multiple send requests into the transmit queue. That way, the highest send rates achievable with the PCA-200E are generated.

The test application tries to accept all the data from the network. The application-provided receive function being called from the client library each time a PDU has arrived counts the number of delivered PDUs.

Due to the design of the driver's IPC interface, an application failing to keep up with the incoming data stream is not critical, at least not for the driver. If the application (actually, the receive thread in the client library) is not ready to accept the next RECEIVE message from the driver, the driver simply drops that PDU because of an IPC timeout. To get the number of PDUs dropped at this point, a counter was added to the driver.

To summarize the measurements, at least 99.81 percent of the received PDUs really arrived at the client, using the protocol described in the implementation section.

This value could be increased to approximately 99.95 percent by changing the protocol the same way as described in Section 5.3. Thus, the protocol overhead is minimized by simply copying the physical address of the received data to the client instead of the data itself.

To have a reliable receive path, it was necessary to pay attention to the priorities of the participating threads, the interrupt thread in the driver and the receive thread in the client library. As there was no explicit rule for these thread's priority, the receive thread had a lower priority than the interrupt thread. This way, the receive thread could do its work only in the time slice donated by the interrupt thread through IPC. If the IPC took place just before the end of the interrupt thread's time slice, the donated time might not have been sufficient for the receive thread to become ready again. In this case, the next IPC operation would fail, leading to a dropped PDU.

In several discussions about this problem a possible solution was identifed: The receive thread runs with a higher priority than the interrupt thread. Thus, it always becomes ready again, before the interrupt thread can send the next RECEIVE message. Although in this scenario all PDUs arrive at the test application, there are some things to consider: The receive thread should do nothing more than accept the data, store it in the client's buffer and return to receive the next message. Otherwise, the interrupt thread would be blocked, possibly causing the board to drop PDUs itself due to a full receive queue or a lack of free receive buffers.

6 Summary

This work aimed at porting the PCA-200E Linux driver to L4, with the additional challenge of minimizing changes to the code. The result is a stand-alone L4 server for the PCA-200E, a client library for the client and an L4Linux driver stub which offers the services of the L4 server to L4Linux as an ATM-on-Linux device. The changes to the original Linux driver code have been minimized by means of a framework that emulates most of the Linux functions required by the Linux driver.

With the PCA-200E L4 server and the corresponding client library it is possible to generate high bandwidth data streams over the ATM network and to receive those streams, even on off-the-shelf standard PC machines.

The interface between the L4 server and its client is hardware independent. This allows implementation of an L4 server for another board type but using the same interface. The framework designed and implemented in this work serves well for ATM network boards. But, due to the somewhat special nature of ATM (connection-oriented, point-to-point), this framework doesn't perfectly fit for other kinds of network boards.

6.1 Future Work

There is still some room for optimization. The send path could be implemented to use the asynchronous capabilities of the board. The IPC protocol could be minimized in a way that all requests use fast short messages.

Acknowledgements

I would like to thank the members of the Operating Systems chair at the Dresden University of Technology who helped bringing this work to a successful end, in particular Prof. Hermann Härtig, Jean Wolter, Michael Hohmuth, Lars Reuther and Martin Borriss for helpful discussions, Sebastian Schönberg and Volkmar Uhlig for repeatedly reading and correcting this paper, and last but not least my friend for her understanding and her love during this busy time.

7 Appendix

7.1 The ATM Device Operations Structure atmdev_ops

```
struct atmdev_ops {
  int  (*open)(struct atm_vcc *vcc,short vpi,int vci);
  void (*close)(struct atm_vcc *vcc);
  int  (*ioctl)(struct atm_dev *dev,unsigned int cmd,unsigned long arg);
  int  (*getsockopt)(struct atm_vcc *vcc,int level,int optname,
                     char *optval,int *optlen);
  int  (*setsockopt)(struct atm_vcc *vcc,int level,int optname,
                     char *optval,int optlen);
  int  (*send)(struct atm_vcc *vcc,struct sk_buff *skb);
  int  (*sg_send)(struct atm_vcc *vcc,unsigned long start,
                  unsigned long size);
  void (*poll)(struct atm_vcc *vcc,int nonblock);
  int  (*send_oam)(struct atm_vcc *vcc,void *cell,int flags);
  void (*phy_put)(struct atm_dev *dev,unsigned char value,
                   unsigned long addr);
  unsigned char (*phy_get)(struct atm_dev *dev,unsigned long addr);
  void (*feedback)(struct atm_vcc *vcc,struct sk_buff *skb,
                    unsigned long start,unsigned long dest,int len);
  int  (*change_qos)(struct atm_vcc *vcc,struct atm_qos *qos);
  void (*free_rx_skb)(struct atm_vcc *vcc, struct sk_buff *skb);
};
```

7.2 The ATM VCC Structure atm_vcc

```
struct atm_vcc {
        unsigned short  flags;          /* VCC flags (ATM_VF_*) */
        unsigned char   family;         /* address family; 0 if unused */
        unsigned char   aal;            /* ATM Adaption Layer */
        short           vpi;            /* VPI and VCI (types must be equal */
                                        /* with sockaddr) */
        int             vci;
        unsigned long   aal_options;    /* AAL layer options */
        unsigned long   atm_options;    /* ATM layer options */
        struct atm_dev  *dev;           /* device back pointer */
        struct atm_qos  qos;            /* QOS */
        unsigned long   tx_quota,rx_quota; /* buffer quotas */
        atomic_t        tx_inuse,rx_inuse; /* buffer space in use */
        void (*push)(struct atm_vcc *vcc,struct sk_buff *skb);
        void (*pop)(struct atm_vcc *vcc,struct sk_buff *skb); /* optional */
        struct sk_buff *(*peek)(struct atm_vcc *vcc,unsigned long pdu_size,
            __u32 (*fetch)(struct atm_vcc *vcc,int i));
                                        /* super-efficient xfers; note that */
                                        /* PDU_SIZE may be rounded */
        struct sk_buff *(*alloc_tx)(struct atm_vcc *vcc,unsigned int size);
                                        /* TX allocation routine - can be */
                                        /* modified by protocol or by driver.*/
                                        /* NOTE: this interface will change */
        int (*push_oam)(struct atm_vcc *vcc,void *cell);
        void            *dev_data;      /* per-device data */
        void            *proto_data;    /* per-protocol data */
        struct timeval  timestamp;      /* AAL timestamps */
        struct sk_buff_head recvq;      /* receive queue */
        struct atm_aal_stats *stats;    /* pointer to AAL stats group */
        struct wait_queue *sleep;       /* if socket is busy */
        struct wait_queue *wsleep;      /* if waiting for write buffer space */
        struct atm_vcc  *prev,*next;
        /* SVC part — may move later */
        short           itf;            /* interface number */
        struct sockaddr_atmsvc local;
        struct sockaddr_atmsvc remote;
        void (*callback)(struct atm_vcc *vcc);
        struct sk_buff_head listenq;
        int             backlog_quota;  /* number of connection requests we */
                                        /* can still accept */
        int             reply;
        void            *user_back;     /* user backlink - not touched */
};
```

References

[Alm96a] Werner Almesberger. *Linux ATM API, Draft, version 0.4*. Laboratoire de Réseaux de Communication (LRC), Ecole polytechnique fédérale de Lausanne (EPFL), CH-1015 Lausanne, Switzerland, July 1996.

[Alm96b] Werner Almesberger. *Linux ATM device driver interface, Draft, version 0.1*. Laboratoire de Réseaux de Communication (LRC), Ecole polytechnique fédérale de Lausanne (EPFL), CH-1015 Lausanne, Switzerland, February 1996.

[ATM94] The ATM Forum. *ATM User-Network Interface Specification, Version 3.1*, September 1994.

[BDH97] M. Borriss, U. Dannowski, and H. Härtig. Performance von TCP over ATM unter Windows NT und Linux. Technical report, Dresden University of Technology, October 1997.

[Cav94] John David Cavanaugh. Protocol Overhead in IP/ATM Networks. Technical report, Minnesota Supercomputer Center, Inc., 1994.

[FML+97] Bryan Ford, Kevin Van Maren, Jay Lepreau, Stephen Clawson, Bart Robinson, and Jeff Turner. The Flux OS Toolkit: Reusable Components for OS Implementation. In *6th Workshop on Hot Topics in Operating Systems (HotOS)*, Cape Cod, MA, USA, May 1997.

[FOR97] FORE Systems, Inc., 1000 FORE Drive, Warrendale, PA 15086-7502. *Programmer's Reference Manual for AALI Interface*, May 1997.

[HHL+97] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The performance of μ-kernel-based systems. In *16th ACM Symposium on Operating System Principles (SOSP)*, Saint-Malo, France, October 1997.

[Lie96] 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, Sep 1996; available from URL: `ftp://borneo.gmd.de/pub/rs/L4/l4refx86.ps`.

[Sta96] René Stange. Systematische Übertragung von Gerätetreibern von einem monolithischen Betriebssystem auf eine mikrokernbasierte Architektur. Master's thesis, Dresden University of Technology, May 1996. available from `http://os.inf.tu-dresden.de/L4/stange-dipl.{html,ps.gz}`.

[Tan90] A.S. Tanenbaum. *Betriebssysteme - Entwurf und Realisierung*. Carl Hanser Verlag, München, 1990.