Policing Offloaded

Uwe Dannowski
Universität Karlsruhe
System Architecture Group
D-76128 Karlsruhe, Germany
Uwe.Dannowski@ira.uka.de

Hermann Härtig
Dresden University of Technology
Institute for System Architecture
D-01062 Dresden, Germany
haertig@os.inf.tu-dresden.de

Abstract

Policing of incoming packets can produce very high load in worst-case situations on a receiving computer. In real-time systems, resources must be allocated for such worst-case situations if guarantees are given to processes. This paper[1] *describes design and implementation of a policing function on an off-the-shelf network adapter board containing a relatively slow microprocessor. Performance measurements indicate that the host computer can be effectively shielded from misbehaving inbound connections. The load on the host CPU is confined to be proportional to the number of packets on admitted and conforming connections, even in the presence of very large numbers of incoming packets on not-admitted connections. As a side effect, zero-copy implementation of protocols is supported.*

1. Introduction

Increasingly, real-time and non real-time applications share computer systems and networks. In such scenarios, (hard or soft) guarantees must be given to the real-time applications. Some networks, e.g. ATM, support this scenario. However, end systems usually do not adequately support it, mainly due to a lack of support from current operating systems. This paper addresses one particular problem that arises in these mixed real-time and non real-time scenarios.

Let us assume a system effectively supporting reservation of resources for real-time applications while leaving the remaining resources available to other, non real-time applications [8]. Then, resources must also be allocated for demultiplexing incoming network traffic in order to decide whether to accept a packet and forward it or to dump it (this function is called policing). These resources must be allocated for the worst case, i.e. the case when an offensive application sends arbitrary large numbers of packets to the receiving station.

Some research operating systems employ early demultiplexing techniques for this purpose thus leaving the protection of real-time applications to general resource schedulers. But the resources that need to be dedicated to the demultiplexing function are still high. Measurements with a popular driver [7] on an 100 MHz Pentium system have indicated that about 32 μs per packet of 512 bytes are consumed. This implies that for a 1 MBit/s connection about 8 ms per second (i.e. 0.8 %) have to be dedicated to the processing of these packets if the demultiplexing is to be included in the driver. As such, policing a hostile connection of 125 MBit/s in 512 byte packets will require the reservation of 100 % of CPU time.

On the other hand, most network adapters include some processing power, usually small RISC processors. This situation made us investigate which type of processing power is needed in adapter boards (with their significantly simpler structure) to effectively shield host CPUs from offensive traffic. As a first experiment, we took an existing board with a 25 MHz RISC (i960), implemented policing and measured which type of traffic can be policed.

In this paper, we first describe the context of the experiments, i.e. the Dresden Real-Time Operating System[2] and the FORE PCA-200E network adapter. Then we describe the general design of the offloaded system. Next, we report the results of our measurements and draw some conclusions.

2. Related Work

This section briefly introduces two loosely related projects — the U-Net project [3] and the RIO Subsystem [9].

[1] The work described in this paper was done by Uwe Dannowski at Dresden University of Technology.

[2] The DROPS project is supported by DFG (Deutsche Forschungsgemeinschaft, SFB 358, Teilprojekt G2).

2.1. The U-Net Project

The U-Net architecture developed at Cornell University, provides low-latency and high-bandwidth communication over commodity networks for workstations and PCs. It achieves this by virtualizing the network interface such that every application can send and receive messages without operating system intervention. With U-Net, the operating system is no longer involved with the sending and receiving of messages. This allows communication protocols to be implemented at user-level where they can be integrated tightly with the application. In particular, the large buffering and copying costs found in typical in-kernel networking stacks can be avoided, and feed-back to the application about flow-control and packet loss is facilitated.

The key aspects of U-Net are:

- U-Net defines a virtual network interface for commodity networking hardware and operating systems.
- The U-Net virtual network interface preserves the traditional protection boundaries between processes. Multiple applications can use U-Net at the same time without interfering.
- U-Net uses commodity operating systems (Unix and Windows NT) and commodity networks (Fast Ethernet and ATM).

U-Net supports the Myrinet PCI and SBus interfaces, DECChip 21140 Fast Ethernet PCI interface, and the FORE Systems PCA-200/SBA-200 ATM interfaces. Supported operating systems include Linux, Windows NT, Sun OS 4.x, Solaris 2.x, and BSDI.

U-Net/MM is an extension to the U-Net user-level network architecture, allowing messages to be transferred directly to and from any part of an application's address space. This is achieved by integrating a translation look-aside buffer into the network interface and coordinating its operation with the operating system's virtual memory subsystem. This mechanism allows network buffer pages to be pinned and unpinned dynamically.

2.2. The RIO Subsystem

The RIO subsystem, a project at Washington University, enhances the Solaris kernel to enforce the QoS features of the The ACE ORB (TAO) end system and provide end-to-end QoS. This is achieved by using early demultiplexing and schedule-driven protocol processing.

The Solaris default network I/O subsystem processes all packets sequentially at the same priority, regardless of the destination user thread. This can lead to priority inversion easily, preventing high-priority connections to meet their QoS requirements. To overcome this problem, RIO supports priority-based queueing — instead of enforcing strict FIFO order, packets destined for high-priority applications are delivered ahead of low-priority packets. Connections are assigned a priority which is determined from the connection characteristics given with TAO's QoS specification. Priorities map to RIO queues, which are served by an associated in-kernel thread with respective scheduling priority. Incoming packets are inspected by a packet classifier in the network driver which decides whether processing takes place in interrupt context (for low-delay connections) or the packet is enqueued in the appropriate receive queue.

To summarize, the RIO subsystem tries to preserve end-to-end priorities by separating resources and applying priority-based protocol processing.

3. Environment

The experiments were done in the DROPS OS using a FORE PCA-200E adapter board.

3.1. The Dresden Real-Time OS and L^{4}ATM

The Dresden Real-Time Operating Systems Project [6] is a research project aiming at the support of applications with Quality of Service requirements.

Although much research has been done on networking support for continuous-media applications, very few projects tackle related operating system issues, such as scheduling and file system support for bounded response time. The DROPS project attempts to find design techniques for the construction of distributed real-time operating systems whose every component guarantees a certain level of service to applications.

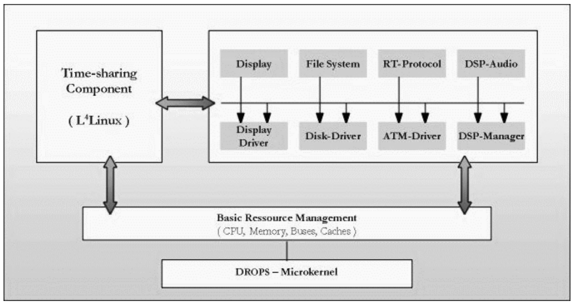

Figure 1. DROPS architecture

A key component is L^{4}Linux, the Linux server on top of the L4 μ-kernel; it serves standard Linux applications. In addition, separate real-time components — designed from scratch — provide deterministic service to real-time applications (Figure 1). At the moment, an ATM protocol component, a real time file system, and a presentation component are available. The real-time components of DROPS are connected via the DROPS real-time streaming interface, an interface designed for the transport of jitter constrained streams.

L^{4}ATM [2] is the ATM protocol component in DROPS. In its current state, it offers a narrow subset of the ATM-on-Linux API [1] — only PVCs are implemented. L^{4}ATM runs as a stand-alone L4 task, using the PCA-200E driver [4] to access the hardware. Clients can use L^{4}ATM through a client library hiding the complexity of the IPC protocol. The library provides the well-known BSD-style socket interface as well as functions for a "zero-copy" data path between the client and the ATM protocol server (`get_ncp_page()`, `ncp_read()`, `ncp_write()`). A stub driver exists for L^{4}Linux presenting the L^{4}ATM protocol component as an ATM network device.

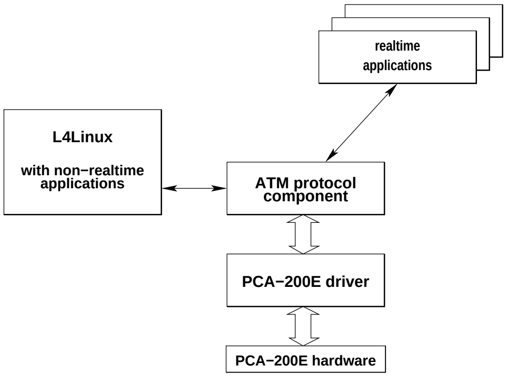

Figure 2. L^{4}ATM in DROPS

Two shared memory areas — one for transmit, the other for receive — are established between each client and the L^{4}ATM protocol server. The physical addresses of the mapped pages in the shared memory areas need to be known and fixed over the lifetime of the mapping. In other words this is pinned memory. The size of the shared memory areas is determined by L^{4}ATM from the QoS parameter set specified with a `setsockopt()` call after socket creation and prior to connection establishment.

At connection setup time, a dedicated worker thread per connection is created in L^{4}ATM. This thread handles transmit requests from the client and pushes received data to the client. On transmit, this thread may block to enforce the negotiated transmit rate (traffic shaping). If received data is available, it is pushed to the client at the negotiated receive rate.

A "pseudo interrupt thread" in L^{4}ATM waits for messages from the driver's interrupt thread and copies the received protocol data unit (PDU) into the buffer of the respective connection. If no free buffer space is available, the received PDU is dropped. At that point the PDU was already transfered into main memory and processed by the driver and by the ATM protocol. Since buffer sizes are determined from QoS parameters, this indicates a non conforming traffic source. With the current implementation of L^{4}ATM and the PCA-200E device driver this path is very CPU intensive, which is even worse in an overload situation.

In short, L^{4}ATM manages exclusive receive buffers and a dedicated worker thread per virtual channel (VC). The receive buffer size is determined from the QoS parameter set of the connection.

3.2. FORE PCA-200E

The PCA-200E is the PCI based member of FORE's *FORERunner 200E* 155 Mbps ATM network adapter series. Figure 3 shows a schematic diagram of the PCA-200E.

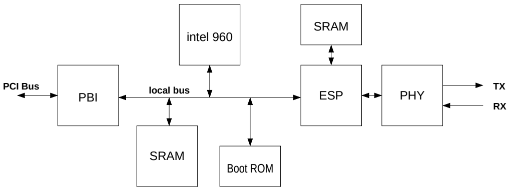

Figure 3. PCA-200E Structure

All members of the 200E series share a common part and a host-bus specific interface. The common part consists of Intel's i960-CA processor, 256 KB RAM local to the i960, an ESP-ASIC (Enhanced SAR Processor) with 128 KB RAM, and PMC Sierra's SUNI-155/Lite User Network Interface (UNI). Either an UTP interface or a fiber-optical interface is attached to the UNI. Host side interfaces are available for SBus, EISA Bus, GIO Bus, PCI Bus, VME bus and Micro Channel Bus.

During host driver initialization, the firmware executing on the i960 is to be loaded into the i960's local RAM.

4. General Design

4.1. Design Goals

The design goals can be summarized as follows:

- **per virtual channel (VC) buffering**
  Current firmware as used in the Linux system provides only two distinct receive buffer pools. Receive buffers are allocated in a hardly predictable order forcing the system to do copy/map operations. Per VC buffering allows a zero copy receive path up to the application. The effects of buffer overruns due to non conforming connections would be limited to the single connection.

- **data routing decisions in firmware**
  Having per VC buffering, it is possible to locate receive buffers either in host memory or in PCI attached memory, depending on the further processing steps.

- **drop unwanted cells earlier**
  Transferring knowledge of connection characteristics (traffic parameters) to the firmware moves the point where non conforming data can be discarded from the device driver to the firmware. This saves valuable PCI/host memory bandwidth and host CPU cycles.

- **reduced and bounded host resource usage**
  By reducing and bounding the usage of the host's resources like memory bandwidth and CPU cycles, predictability can be improved.

- **priorities**
  Real-time connections should have priority over non real-time connections. That way, both connection types can be used concurrently without much effort.

4.2. Interface

The adapter firmware maintains and operates transmit and receive connections. Each receive connection has its own buffer and queue of received PDUs.

Requests to initially configure the adapter, to open and close connections are maintained in a command queue in the adapter's local RAM. The command queue is a list of physical addresses of command descriptors. The firmware polls the current entry of the command queue. Since the queue is in the local RAM, no PCI transfers are required for that. An entry can be written by the host driver in a single PCI transfer.

Results are communicated to the host (the driver) by inserting one of the following events into a host-resident event queue:

- transmit request completed
- control request completed
- PDU received
- receive queue full
- receive queue over threshold

Events can be **silent** or **interruptive**. Whenever an interruptive event is written to the event queue, an interrupt is generated on the PCI bus. Furthermore, an interrupt is generated when a receive queue gets filled up (error condition due to a non conforming connection) or the number of silent events has reached a threshold value. As such, connections with relaxed timing requirements (i.e. best-effort connections) can receive or transmit data without causing an interrupt for each single PDU. Hence, silent events are the means of reducing the interrupt frequency.

Buffer management is done by firmware. It manages the list of received PDUs and exports it to the host driver. The firmware reassembles received PDUs directly into per-VC receive buffers. Due to hardware limitations, a PDU always starts at an address that is a multiple of four. In the case of insufficient space at the end of a buffer the PDU continues at the start of the buffer (wrap around). With this buffer model, a PDU can happen to be split into two segments. Neither the API of L^{4}ATM nor the ATM-on-Linux API support scattered buffers. To overcome this, the start of the buffer could be mapped behind the end of the buffer — assuming the buffer is aligned to a page boundary and a multiple of pages in size. This way a split PDU can be accessed as a contiguous block of virtual memory.

In addition, a receive queue is maintained by the firmware for each connection. The firmware writes information about the received PDUs to the queue. The dedicated worker thread in L^{4}ATM pulls entries from the receive queue, optionally applies traffic management algorithms, pushes the PDU to the application, and adjusts the tail pointer for the buffer and the receive queue afterwards.

Using this model, connections can be set up with dedicated buffers. Connections which are non conforming simply run out of buffer space and the firmware discards further PDUs without causing load on the host CPU.

4.3. Implementation Structure

The implementation essentially can be described by this pseudo-code:

```
while (true)
{
  transmit();
  receive();
  handle_commands();
}
```

Execution time of each function is minimized and bound to reduce overall latency and to avoid data loss due to FIFO overflows. Whenever a function would block in a certain operation, it saves its current state and returns to its caller. The current state is stored in a set of descriptors. Per-VC receive descriptors hold the state of the receive function, per-PDU transmit descriptors are used for the transmit function. The system could be characterized as a very simple cooperative process manager. In the following we will therefore refer to the functions as processes.

The implementation of the **transmit process** is straight forward. Its only interesting property is the usage of the four available hardware FIFOs. We use three FIFOS for real-time connections and the one remaining FIFO for best-effort connections. This has one drawback: even though the rate of a FIFO's token generator can be changed anytime, the rate must not be changed when the FIFO holds transmit data, since this would change the rate of the currently processing PDU. However, as long as the number of concurrent real-time connections does not exceed the number of high-priority FIFOs, these FIFOs can be used exclusively by a certain connection. Other scenarios with sharing of FIFOs for several connections are discussed in [5].

The implementation of the **receive process** we describe in somewhat more detail. Reception of data from the network involves several stages: identify incoming cells, either drop cells or apply AAL processing, transfer data into host buffers, and notify the driver.

Incoming cells are put in one of the four receive FIFOs, no matter if there is an open connection for that VPI/VCI pair. If any of the four FIFOs contains more than a configurable number of cells, a flag is set in a control register (An interrupt could be requested too). For the sake of low latency the cells should be pulled from the ESP's receive FIFO as soon as possible. Cells for which a connection is registered are processed by the respective receive function for reassembly. All other cells are dropped. The receive function implements AAL processing and reassembly. Obviously, the receive function is the best point to apply early demultiplexing. Receive buffers for a VC are installed at connection setup. Buffer space is provided by the protocol implementation; the buffer size is determined from the QoS parameter set. The receive function transfers the reassembled data directly from the ESP's receive FIFO into the per VC buffers in host memory. This way a zero-copy receive data path can be implemented.

A major point of interest is the identification of incoming cells. First of all, a decision has to be made on whether to discard the cell(s) or not. Only cells of open inbound connections are important. All other cells should be removed from the ESP's receive FIFO as soon as possible.

The receive hardware provides four different receive FIFOs. When the use of all four FIFOs is not disabled, the receive hardware selects the appropriate FIFO (0..3) by inspecting the two least significant bits of the VCI field in the cell header. Depending on the way the FIFOs are served, cells with different VCI values could outstrip each other. This is legitimate, as long as cells of the same VC are kept in order. The same could also happen in an ATM switch featuring per VC queuing.

The firmware must maintain information on all open VCs. When the cell header is pulled from the receive cell header FIFO, a connection descriptor — if a connection exists for this VPI/VCI pair — must be looked up. This operation should be fair in that it is equally expensive for all open connections. Furthermore, the operation should be very cheap when no associated connection can be found. Implementation details are discussed in [5].

The result of the lookup operation is a pointer to an internal receive descriptor holding all information for the reassembly process. If incoming cells do not belong to an open connection, the pointer is invalid and the cells are discarded immediately.

5. Some Implementation Details

5.1. Header Coalescing

Transmission of a cell consists of two parts — updating the cell header register and transferring the cell payload. Cell headers in an AAL5 PDU are common for all but the last cell. Opposed to writing a cell header for each cell, using the header coalescing feature only two updates to the cell header register are required. Hence, the number of accesses to the ESP can be reduced which frees CPU cycles for other tasks. Figure 4 shows the ratio of required number of accesses with and without this feature enabled.

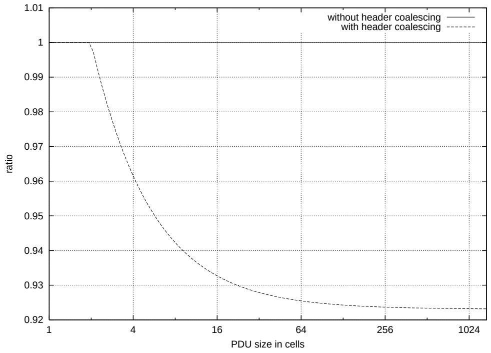

Figure 4. Effects of Header Coalescing

For a PDU size of 9180 bytes, header coalescing reduces the required number of accesses to the ESP to 92.3 %. Hence, this feature reduces the transmit costs by nearly 8 %. A similar mechanism exists for the receive path, too. Another important implication of header coalescing is, that multiple consecutive cells can be transfered in one block.

5.2. Identification of Incoming Cells

During the receive process, a cell header is pulled from the receive FIFO. Solely with the cell header available a decision must be made on whether the cells belong to an open connection or not. One can think of several approaches of how to determine that:

tree-based search: Using this method, the whole range of VPIs and VCIs can be covered. The tree contains pointers to dynamically allocated connection descriptors. The number of connections is limited by the available memory for connection descriptors. The time it takes to lookup a certain connection depends on the number of connections. By the use of self optimizing trees the seek time for "high-traffic" connections could be minimized while incrementing the time for other connections. But, this would induce a certain level of unfairness. Furthermore, it would be necessary to investigate if inserting a node (opening a connection is not a time-critical operation unlike the lookup on cell arrival) could break the hard time limits for searching the tree.

hash-based search with overflow buckets: A commonly used approach would be to calculate a hash value from VCI and VPI as an index into a hash table with overflow buckets. Although this method could reduce the mean lookup time, it can result in heavily different lookup times. The lookup costs for a non-present entry can be enormous.

VPI/VCI as table index: By reducing the significant bits in the VPI and VCI field to a total of, say 10, the value gained from merging the significant bits is used as an index into an array of descriptors. This reduces the number of VPIs and VCIs the firmware can cope with. The number of connections is limited by the table size (which in turn is limited by the available memory). Using an array of pointers to dynamically allocated connection descriptors, the table size (and thus the VPI/VCI space) could be slightly enlarged while incrementing the costs for a lookup operation by an additional level of indirection.

With respect to the criteria listed in Section 4.3, the method described last looks most suitable for the given problem. It takes minimum and constant lookup time — no matter whether an open connection exists or not — at the price of a reduced VPI/VCI space.

5.3. Cell Discard and Packet Discard

Incoming cells are inserted into one of four receive FIFOs in the ESP, even if no connection has been opened for those cells or a connection's buffer is filled up. In both of these situations the firmware should remove the cells from the receive FIFO. Therefore, the firmware reads the cells from the receive FIFO without further processing. Even in a hand-optimized loop this takes at least twelve cycles per cell.

A solution that significantly reduces the overhead associated with discarding cells could be to adjust the memory pointers of the ESP's FIFO logic. That way, a short address calculation and a single write access to an ESP register would be enough to get rid of multiple cells. This method is most efficient for large blocks of cells. But, even though the pointers are accessible from the i960, no stable mechanism was found by the time of writing. However, independent of the method used for discarding, cells classified as to be discarded produce no extra load on the host CPU.

One of the firmware's jobs is to reduce the work for the host CPU. Therefore, it implements a packet discard mechanism for AAL5 PDUs. Whenever a connection's receive buffer has insufficient free space for handling the current PDU, the connection is marked as disabled and all future cells except for the End-of-PDU cell are discarded. After arrival of the last cell, the connection is enabled again. Since the length of an AAL5 PDU is encoded in the *last* cell, there is no way to determine at arrival of a PDU's first cell whether the entire PDU will fit into the available buffer space. Hence, for connections with variable PDU sizes this packet discarding scheme cannot completely avoid unnecessary PCI-Bus usage, whereas with a committed fixed PDU size the firmware can begin discarding of cells even at the start of a PDU. Nevertheless, it unburdens the handling of dropped packets from the host CPU in overload situations, which is a major advantage over the previous combination of the PCA-200E driver and L^4ATM.

5.4. FIFO Selection Scheme

The transmit FIFO selection scheme described in Section 4.3 works fine for up to three concurrent real-time connections. Scenarios with more than three concurrent real-time connections can be handled similarly as long as they can be implemented by PDU interleaving (VBR connections). For the remaining cases, two FIFOs should be associated with the two connections with the highest rate, while a cell-level scheduler multiplexes the remaining real-time connections onto the third high-priority FIFO. But, that cell-level scheduling approach may be limited to quite low data rates because of the PCA-200E's architectural constraints. In [10] a feasibility study of a software-based cell-level scheduler is presented: based on a Pentium Pro 200 MHz up to 1000 concurrent connections can be shaped simultaneously. But, in contrast to the i960 on the PCA-200E, the CPU used in this experiment is *not* responsible for actual data movement. It runs algorithms for the scheduling of DMA transfers and cell transmits. Using this reasoning and the estimates from Section 6.2, software-based cell-level scheduling at high rates is likely to fail on the PCA-200E hardware.

6. Performance

6.1. Memory Considerations

The PCA-200E board has 256 KB of fast SRAM installed for storing the firmware code and data. The first KB is shadowed by the i960's internal RAM. The mon960 debugger is of much help during debugging (breakpoints, single stepping, etc.) — which takes about 20 KB for its private data structures. So, there are about 234 KB available for the firmware. This space has to be shared for code and data. 32 KB would be a very pessimistic estimate of the code size, leaving 200 KB for data related to connection management. Targeting at a minimum of 1024 connections, this would result in nearly 200 bytes per-connection data (including receive descriptors, buffer descriptors, transmit descriptors, etc.) Taking into consideration that no data is to be buffered in this SRAM, that seems plenty of space.

6.2. CPU Cycles

Estimates

The i960 CPU on the PCA-200E board is clocked at 25 MHz. A saturated full-duplex 155 Mbps link transfers 706414 cells/s. The PCA-200E hardware allows to transfer cells as twelve 32-bit words in a burst operation (taking one external cycle per word), leaving an average of 23 cycles per cell for overall processing overhead. Even with the i960 being able to execute about two instructions per cycle, this value seems insufficient. This underlines the importance of the header coalescing feature that allows to avoid treating each cell separately.

Call Costs

The modular design presented in Section 4.3 is strongly supported by the i960 architecture. The combination of two features, a sophisticated call-and-return mechanism saving procedure-local registers and a saved register cache with a maximum depth of sixteen, makes function calls very cheap. In a typical program, procedure calls and returns cause procedure depth to oscillate a few levels around a median call depth. Unless oscillation is larger than the number of cache-able register sets, no cache flush is required. A **call** or **return** instruction involves transfer of sixteen 32-bit registers which consumes only four clock cycles. By the use of clever instruction scheduling, up to two instructions prior to the call/return can be executed in parallel with the **call** or **return** instruction.

6.3. Execution Timings

Table 1 lists execution times for certain tasks[3]. Due to the lack of a timestamp counter in the i960, timings were taken by generating an interrupt after a certain number of iterations of the task. The host then calculated the duration between two successive interrupts by use of the Pentium's timestamp counter. From that the cycles for a single iteration were derived. Obviously, the values gained reflect a minimum — the i960's instruction cache combined with its 16-word prefetch buffer eventually may have reduced execution time during measurements in a way that is not achievable under normal conditions. On the other hand, running the i960 with cache disabled would increase execution times far beyond the norm.

task	cycles
test if cells are in the any of the receive FIFOs	5
parse empty TX FIFO queues	28
enqueue a TPD to a FIFO's empty queue	21
enqueue a TPD to a FIFO's queue	38
dequeue a TPD from a FIFO's queue	34

Table 1. Execution Times

Related to the estimated 23 cycles per cell these values underline the motivation of header coalescing. Header coalescing enables transfer of multiple cells in a block operation, grouping data transfer cycles together. That way "long running" operations (taking more than 23 cycles) become feasible.

6.4. Maximum Bandwidth

Though we were interested in latency and throughput, latency was not measured due to lacking suitable ATM measurement tools. The numbers presented in this section were obtained either by the use of interrupts and the host's timestamp counter or from a network management/monitoring tool whose precision is more than questionable.

Figure 5 shows the maximum achievable transmit bandwidth in relation to the PDU size. The dashed line shows the values for a connection with reduced transmit rate. The degradation on small PDU sizes relates to the processing overhead in the firmware. The transmit FIFO drains off before the firmware is ready to refill the FIFO again. For a connection at link speed the overhead becomes significant for PDU sizes of less than 1024 octets. But, a connection with approximately 100000 cells/s can achieve its requested bandwidth even with a PDU size of three cells.

[3] TPD — internal transmit PDU descriptor

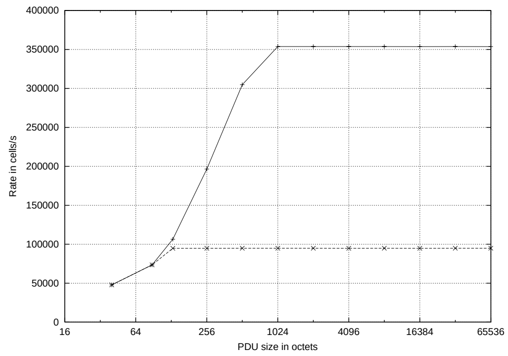

Figure 5. Maximum Transmit Rate

6.5. Concurrency

One of the problems addressed by DROPS is the isolation of concurrent activities. Regarding ATM connections, this means reducing mutual influences of concurrent connections. In this section the behavior and effects of concurrent connections are investigated.

Concurrent Transmitters

With the ESP's transmit FIFOs the firmware is able to shape up to three concurrent outgoing real-time connections while offering the remaining bandwidth for best-effort connections. Given that there is a feasible cell schedule[4] for these connections, no mutual influences should occur. The graphs in Figure 6 show cell rates of four concurrent outgoing connections and their sum: C2 with 24000 cells/s, C3 with 48000 cells/s, C1 whose rate is changed and the best-effort connection C4.

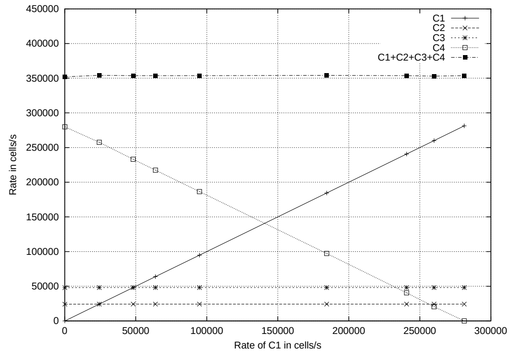

Figure 6. Concurrency of Outbound Connections

During the experiment, the rate of C1 was changed. The rate of C2 and C3 remained stable as expected whereas the rate of C4, the best-effort connection, adapted to the remaining rate.

Concurrent Receivers

Two central ideas of this work are the isolation of inbound connections and the protection of the host system from overload situations. Connections exceeding their negotiated rate are to be policed by discarding their packets. The host's CPU utilization should not be influenced by those connections.

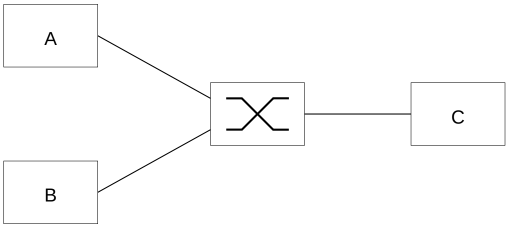

Figure 7. Measurement Setup

The setup depicted in Figure 7 was used for the measurements: Three machines (A, B and C) ran DROPS with an application to control the firmware. Additionally, an "idler" ran on C to measure the available CPU time. Host A generated two AAL5 streams of 8192 byte PDUs, both with a rate of 20 MBit/s. Host B generated an AAL5 stream with varying PDU size and rate. The switch was configured with three VCs leading to the port C was connected to. The "idler" on C repeatedly measures the number of iterations in a tiny loop in a certain number of CPU cycles. Taking the value for an unloaded CPU as reference, this allows estimation of remaining CPU time. Furthermore, the firmware was configured to request an interrupt for every PDU that was received successfully.

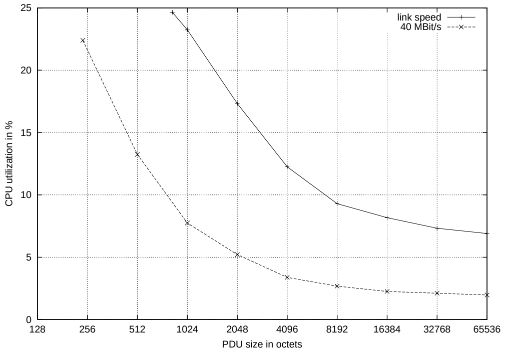

Figure 8. Host CPU Utilization

As expected, without any open connection the host CPU

[4] It is left to L^{4}ATM to determine if a cell schedule exists for a set of real-time connections. This decision should be made during the admission control phase at connection setup.

does not even notice the incoming data stream. Figure 8 shows utilization of the host's CPU time for a connection at link speed and for a 40 MBit/s connection, both with varying PDU size. Here, the interrupt handler on the host immediately acknowledges the PDU by advancing the tail pointer of both the receive buffer and the receive queue. Values for smaller PDU sizes are not available because of massive CRC errors, probably induced by the firmware's overhead. The influences of misbehaving inbound connections on the host CPU are shown in Figure 9. Here, two connections of 20 MBit/s each and a third connection of 40 MBit/s were opened. Host A sends on the two connections at their negotiated rate, whereas the rate of the other connection, the "hazard", originating at B is changed.

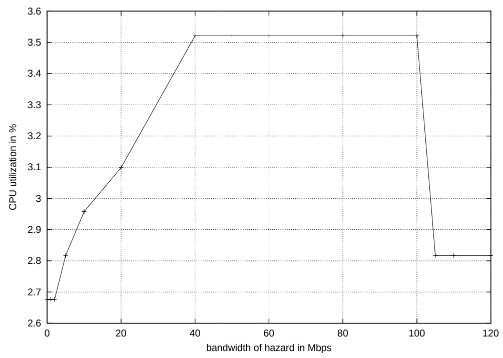

Figure 9. Host CPU Utilization with "hazard"

Up to a rate of 2 MBit/s no significant load can be detected. Starting with 5 MBit/s, an almost linear increase in CPU utilization can be monitored, up to the point where the actual rate exceeds the negotiated rate. Since the receive buffer is drained with a rate of max. 40 MBit/s, additional PDUs are discarded by the firmware — the host does not get an interrupt for that. Hence the CPU utilization remains constant. When the rate of the connection exceeds 100 MBit/s, a remarkable decrease in CPU utilization can be seen. This is obvious: all three data streams are joined on C's switch port, which gets saturated at about (20 + 20 + 100) MBit/s. Due to cell discard strategies in the switch, cells of the hazardous connection are dropped. This leads to damaged PDUs, which in turn pose no load to the C's CPU.

7. Summary and Discussion of Performance Results

This work has been done to study the feasibility of offloading policing functions to a slow dedicated CPU. We first summarize the results given in section 6 and then discuss their relevance.

7.1. Summary of Results

The obtained data indicate:

- Header coalescing is an important feature to avoid cell level scheduling decisions for firmware.
- For packet sizes of above 1 KB transmitting data is possible at full speed on a 25 MHz CPU (such as i960) with specialized cell assembly support available.
- Within certain limitations that depend on the hardware (mainly number of hardware FIFOs) concurrent writing is possible at full speed.
- Load on host CPU depends linearly on the actual bandwidth of conforming connections. Thus the host CPU can be effectively shielded from non conforming traffic.

This indicates that a small, dedicated, and inexpensive network CPU can very effectively be used to improve worst-case behavior in the presence of non conforming network connections.

7.2. Discussion of Results

The firmware supports traffic shaping on the transmit path, differentiates between real-time and best-effort connections, and allows implementation of zero-copy transmit and receive paths. It uses a simple but powerful policing mechanism to protect the host from misbehaving inbound connections. As shown in Section 6, it provides effective methods to minimize the work for the host CPU. Paired with L^4 ATM this firmware offers resource-saving real-time communication via ATM.

However, it may be (and has been) argued that *Offloading Policing* solves a nonexistent problem, since servers commonly handle very little incoming traffic while clients handle very little traffic at all. Even if that might be correct in general, the argument ignores requirements of real-time systems. In real-time systems, resources need to be allocated for the worst case, and worst-case requirements limit the number of admissible applications. Hence, offloading policing to a cheap CPU to confine the resources needed for worst-case scenarios is a very economic solution.

Supporting arguments arise in the context of denial of service attacks. Using *Offloaded Policing*, second class packets from non admitted or non conforming offensive sources can be prevented from overwhelming the processing of admitted and conforming requests. Since denial of service attacks cannot in general be attributed to certain network sources (addresses), more practical work on more sophisticated policing techniques needs to be done to substantiate that argument.

It may be (and has been) argued that tomorrow's CPUs will be powerful enough to do centralized policing and that experiments with 100 MHz Pentium and 25 MHz i960 are completely irrelevant. While we have to admit that 155 Mbps ATM and the used CPUs are not exactly the latest technology, we still claim the principal validity of our results for several reasons:

- Faster CPUs will have to handle faster networks as well.
- CPU speed essentially increases with respect to clock rate and the execution of instructions within the chip. A corresponding speedup of I/O and memory accesses is generally not expected.
- Memory speed will remain a limiting factor that especially will make early dropping of packets very desirable.

7.3. Limitations

More work remains to be done, especially in two areas:

- From the current status of experiments it is not clear yet which transmitting bandwidth can be guaranteed in the presence of non conforming inbound connections (worst case). In other words, it is not clear yet up to which transmission load can be loaded to the dedicated CPU concurrently with policing an offensive connection.
- There is no interface so far that allows the operating system on the host CPU to tell the dedicated CPU what to do with non conforming traffic other than dropping. Hence, short term adjustment on temporary low load situations is not possible.

Although the design was aligned to L^{4}ATM's requirements, the general structure allows easy adaption to different host systems. This covers changes to the buffer mechanism as well as different strategies for host notification via interrupts.

8. Conclusions

The emphasis of the work described in this paper has been to provide empirical data of whether or not it makes sense to include some general purpose computing power in network interface cards to offload policing. The results we have obtained using an off-the-shelf adapter card (FORE PCA-200E) let us conclude that the inclusion of a small fraction of a central CPU's processing power on the network interface card suffices to enable offloaded policing. This can, at least for the coverage of worst-case scenarios, relieve a large percentage of load off the main CPU.

Acknowledgements

We'd like to thank Espen Skoglund, Volkmar Uhlig, and all anonymous reviewers for their helpful hints.

References

[1] 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.

[2] Martin Borriss. *Operating Systems Support for Predictable High-Speed Communication*. PhD thesis, Dresden University of Technology, 1999.

[3] Cornell University. U-Net — A User-Level Network Interface Architecture. `http://www2.cs.cornell.edu/U-Net/Default.html`, 1996.

[4] Uwe Dannowski. An ATM Driver for DROPS. Großer Beleg, Dresden University of Technology, June 1998.

[5] Uwe Dannowski. ATM Firmware for DROPS. Master's thesis, Dresden University of Technology, July 1999.

[6] Dresden University of Technology. DROPS — Dresden Realtime Operating System. `http://os.inf.tu-dresden.de/drops/`, 1996 – 1999.

[7] Dresden University of Technology. PCA-200E Linux Driver. `http://os.inf.tu-dresden.de/pca200e/`, 1997 – 1999.

[8] H. Härtig, R. Baumgartl, M. Borriss, Cl.-J. Hamann, M. Hohmuth, F. Mehnert, L. Reuther, S. Schönberg, and J. Wolter. DROPS: OS support for distributed multimedia applications. In *Proceedings of the Eighth ACM SIGOPS European Workshop*, Sintra, Portugal, September 1998.

[9] Fred Kuhns, Douglas C. Schmidt, and David L. Levine. The Design and Performance of a Real-time I/O Subsystem. In *Fifth Real-time Technology and Applications Symposium*, Vancouver, British Columbia, Canada, June 1999.

[10] J. Schiller and P. Gunningberg. Feasibility of a Software-based ATM cell-level scheduler with advanc ed shaping. In *Broadband Communications '98*, Stuttgart, Germany, April 1998.