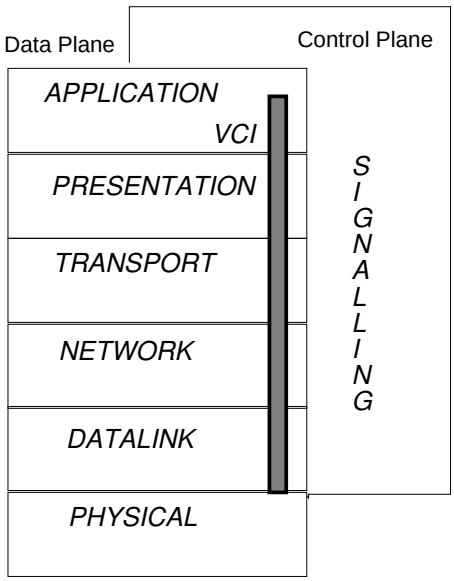
Native-mode ATM in FreeBSD: Experience and Performance

A. Jain and S. Keshav

AT&T Research, Murray Hill

Motivation for a Native Mode Stack



Most current implementations run TCP/IP over ATM

TCP checksums data, not needed with AAL5

Also, neither obeys ABR signals, nor is leaky-bucket bounded

Datagram (IP) layer over connection-oriented ATM may multiplex incompatible connections

For end-to-end QoS native-mode ATM is necessary

Features

No multiplexing of VCIs above physical level

Targeting to ATM

Minimal functionality in the critical path

Portability ensured by separating OS-specific and OS-independent portions

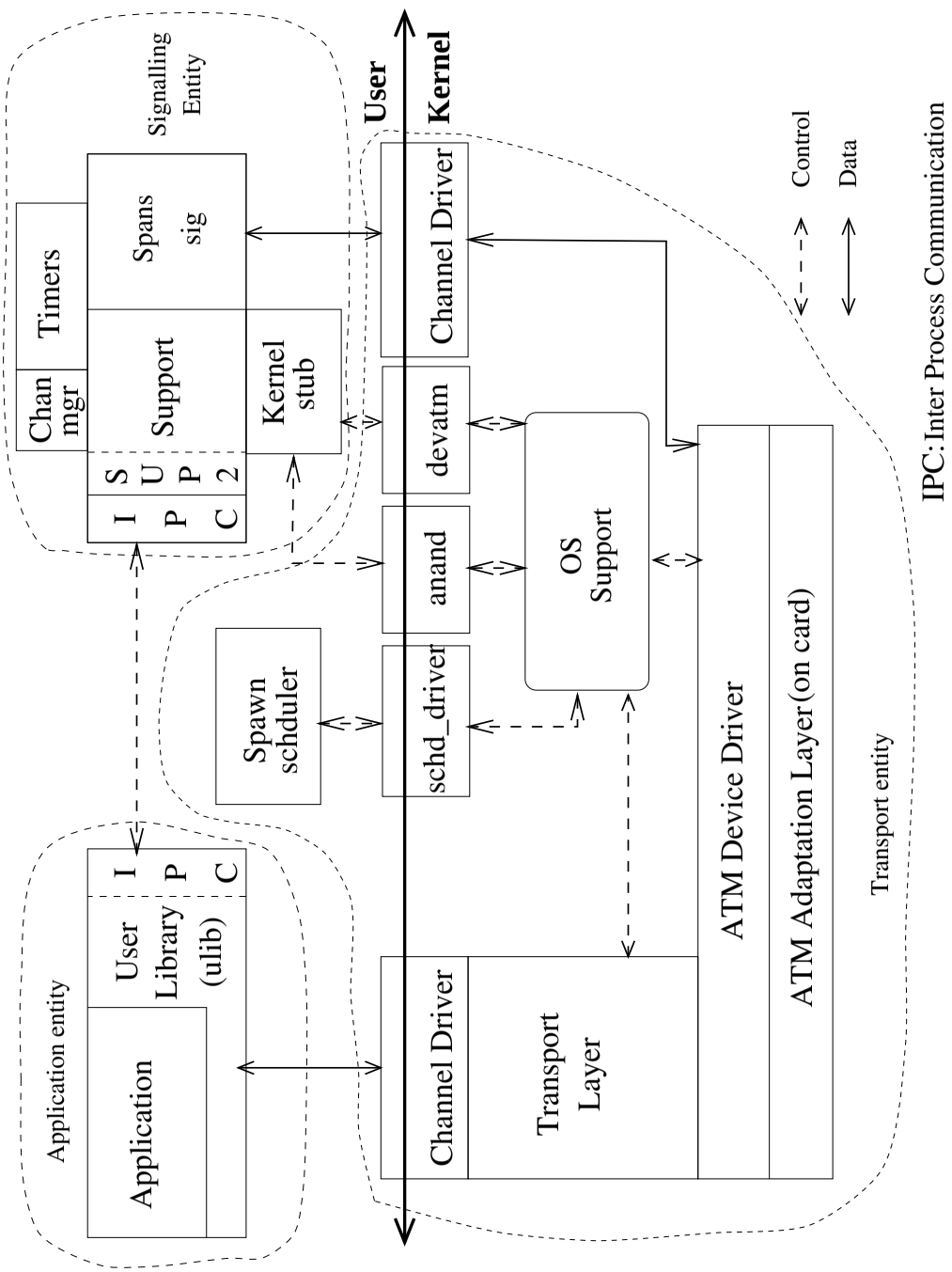
QOS

- Applications specify quality of service (QOS) parameters to signalling
- QOS information is bound to VCI, and used by each data transport layer for scheduling
- We do not assume any specific QOS specification, at the moment

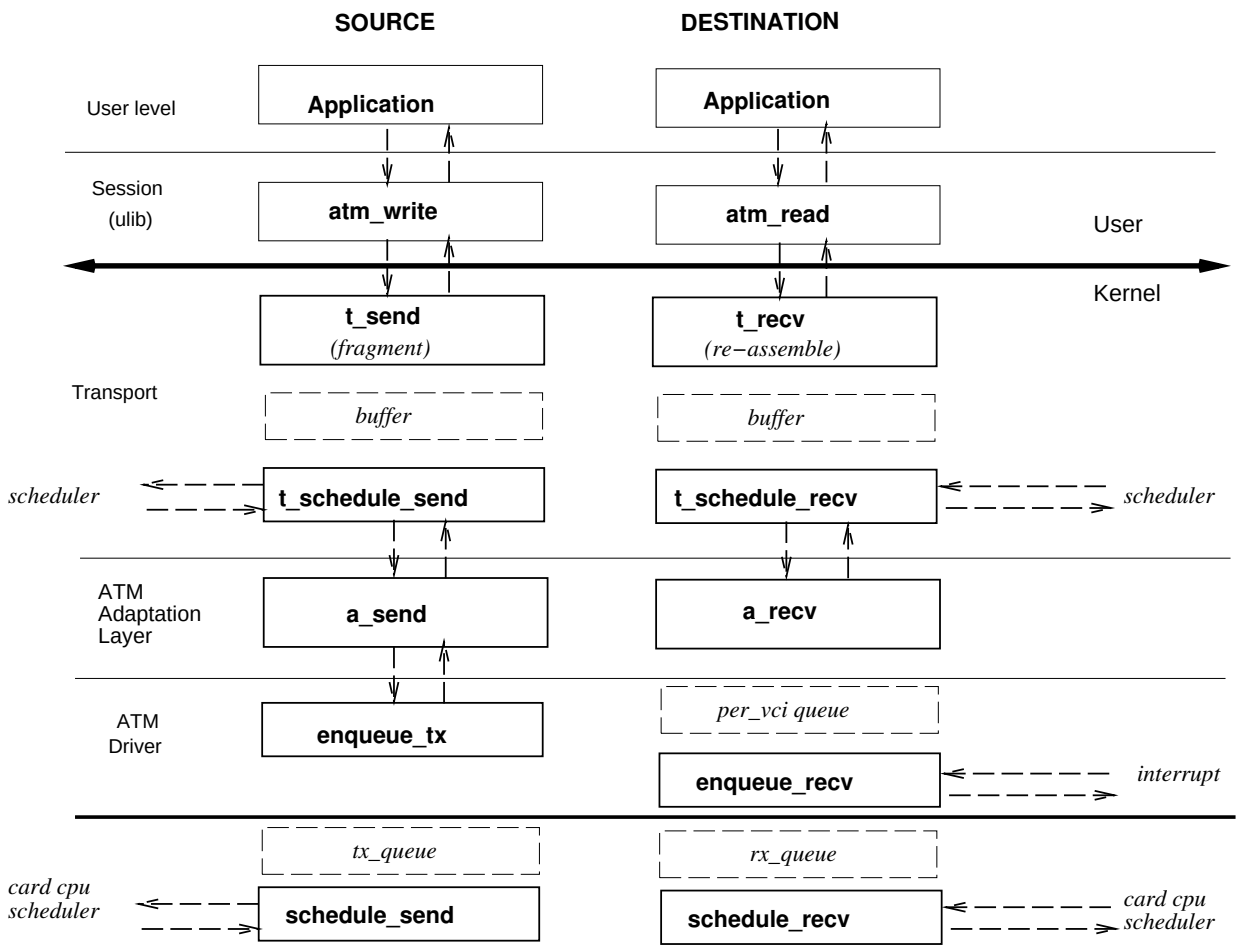
Platform: PC

- Implemented on PCs: high performance at low cost
- FreeBSD OS is freely available and has standard protocol stack
- Off-the-shelf components: easy to replicate

Features


Signalling Entity
Timers
Spans sig
Chan mgr
Support
Kernel stub
SUP2
IPC
User
Kernel
Channel Driver
devatm
anand
schd_driver
Spawn schduler
OS Support
ATM Device Driver
ATM Adaptation Layer(on card)
Transport Layer
Channel Driver
Application entity
Application
User Library (ulib)
IPC
Transport entity
Control
Data
IPC: Inter Process Communication

Overall Data Flow



Porting requirements

IPC

Channel driver (socket interface)

Task scheduler

Locking

Abnormal termination

Page size issues

Buffer pools

Packet- and cell-level smoothing

Testbed

2 486/66 PCs with FORE HPA-200 EISA host-adaptors and FORE ASX-100 switch

2-year old technology, now bottom-of-the-line!

FreeBSD Operating system

Initial version of code came from Indian Institute of Technology, Delhi, India

All code also runs in a packet-level simulator (REAL)

In the process of upgrading to Pentium/133 and PCI-based ATM cards

Performance

Round-trip time : averaged over 10,000 pings

User-to-user latency of 650 microseconds for 64-byte messages

Measured throughput by sending 10,000 messages, results averaged over 50 reps

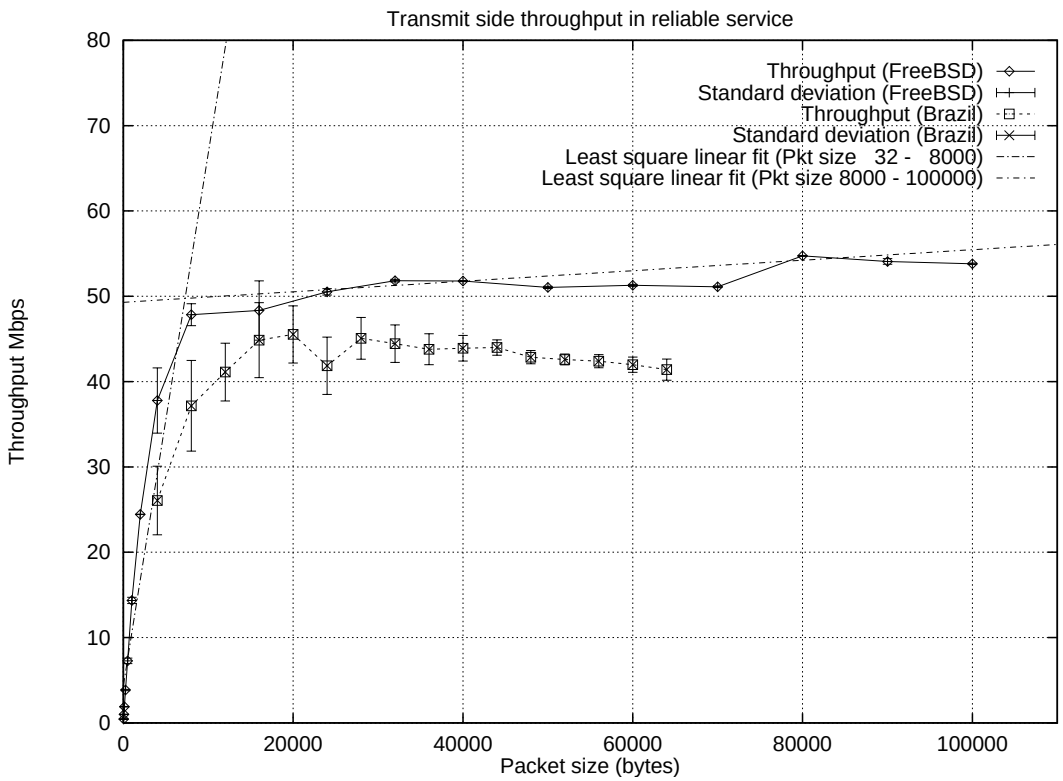
User-to-user speed of 56 Mbps for unreliable transport and 54 Mbps for reliable transport connections

Throughput is half the peak value for 2200 byte messages

Only a slight degradation due to reliable service

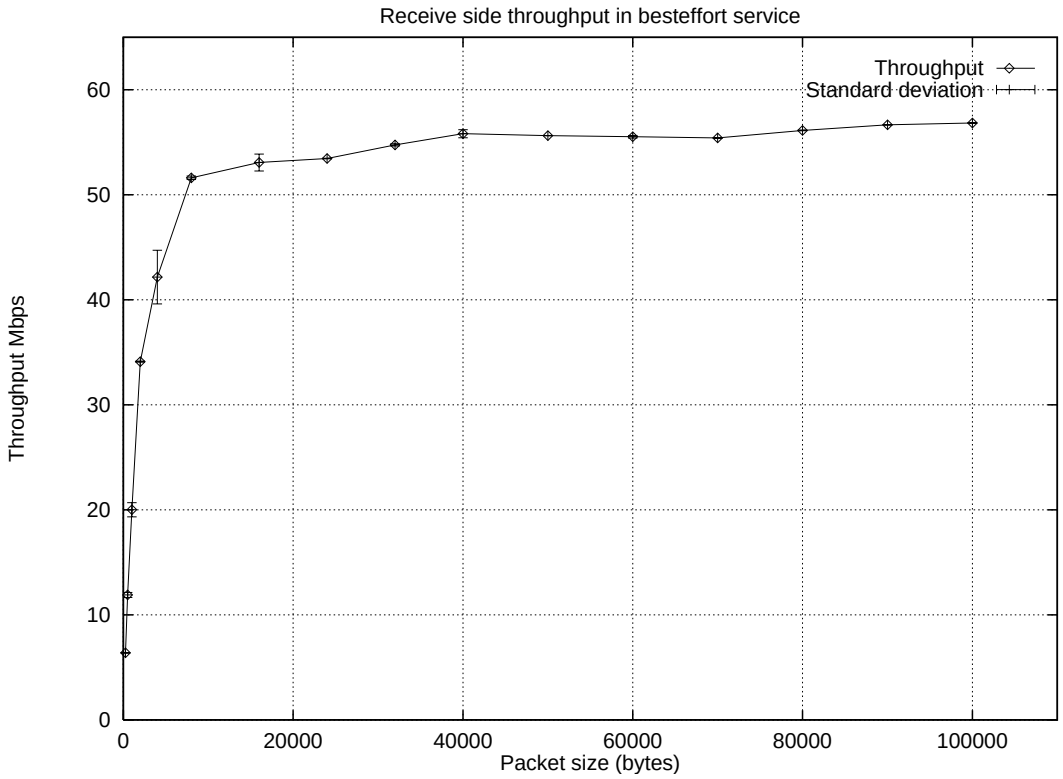
Better than Brazil!

Reliable transmission (using timeouts and retransmissions)



Performance

Unreliable transmission (UDP-like)



Observations

Most of the delay is due to waiting for task scheduler => use scheduler only when pre-emptability is critical

Bottleneck in the system is the receiver's CPU

Dropping data in the interrupt service routing leads to livelock

- Frames arrive too rapidly to be processed and are dropped
- Processing partial frames wastes time

Dropping data in the card removes livelock, but can cause unfairness

Per-VC queueing in the card is essential for fairness

It would also be nice to have a way to turn on and off AAL5 CRC checking, and interrupts

Need modifications to CPU scheduler for providing realtime QOS guarantees

Implementation experience

Reverse-engineering device drivers is a pain, even with source code

Careful memory management is necessary: we manage all headers and mbufs ourselves

Locking (and, particularly, releasing locks on sleeping) is hard to get right

For good performance, every instruction counts

PCs are excellent for development, particularly with FreeBSD/Linux

University release

Testbed can be replicated easily: every component is off-the-shelf

Source code for entire stack available free to universities

Contact: keshav@research.att.com with name of PI and postal address