INTEGRATING ATM NETWORKING INTO BSD

Charles D. Cranor

WUCS-98-??

August 6, 1998

Department of Computer Science
Campus Box 1045
Washington University
One Brookings Drive
St. Louis, MO 63130-4899

Abstract

New computer applications in areas such as multimedia, imaging, and distributed computing demand high levels of performance from computer networks. ATM-based networking solutions provide one possible alternative to meeting these performance needs. However, the complexity of ATM over traditional networks such as Ethernet has proven to be a barrier to its being used. In this paper we present the design and implementation of BSD ATM, a light-weight and efficient ATM software layer for BSD-based operating systems that requires minimal changes to the operating system. BSD ATM can be used both for IP-based networking traffic and for "native" ATM traffic.

INTEGRATING ATM NETWORKING INTO BSD

Charles D. Cranor
chuck@ccrc.wustl.edu

1. INTRODUCTION

New computer applications in areas such as multimedia, imaging, and distributed computing demand high levels of performance from computer networks. In order to meet the performance demands of these applications researchers have been designing new types of networks such as ATM that have higher bandwidths and support performance guarantees. Unfortunately, ATM standards are both large and complex. The cost and complexity of ATM has proven to be a barrier to its integration into traditional operating systems, thus making it difficult to perform experimental research using ATM host-network interfaces. This has become even more of a problem now that new ATM-based multi-media devices such as Washington University's Multimedia Explorer (MMX) have been developed and deployed [11].

To address this problem, we created BSD ATM. BSD ATM provides support for ATM networking under a traditional BSD-based operating system. Features of BSD ATM include:

- BSD ATM is cleanly integrated into the BSD kernel with only minimal changes required to existing networking code (the ATM protocol layer must be registered with the kernel). No new user-level programs are required to use ATM.
- BSD ATM supports both IP-based protocols (using PVCs) and a "native" ATM protocol layer that provides programs with access to ATM virtual circuits at either the AAL0 or AAL5 layer.
- BSD ATM provides good performance. We have observed bandwidths of up to 133 Mbps using IP-based protocols on a 155 Mbps ATM link.
- BSD ATM's device-independent layer is both small and light weight. This allows it to be used as a building block for larger and more complex ATM protocols, and also allowed us to focus our efforts on providing a stable and robust ATM device driver that can handle high-speed multimedia streams from a number of sources.
- BSD ATM compiles and runs under FreeBSD, NetBSD, and OpenBSD, and it is now a standard part of all three of those operating systems.

In this paper we present the design and implementation of BSD ATM. Section 2 presents a brief overview of BSD and ATM networking. Section 3 describes the device-independent parts of BSD ATM. Section 4 describes the operation of the "Midway" ATM device driver that is used to control ATM cards from Efficient Networks [9] and Adaptec [1]. We summarize the results of this research in Section 5.

2. OVERVIEW

In this section we present a brief overview of ATM networking and of the networking subsystem of the BSD kernel.

2.1. ATM NETWORKING

Asynchronous transfer mode (ATM) networks typically consist of a set of hosts connected by ATM links to an ATM switch. An ATM switch receives data from the hosts connected to it and forwards the data to the destination. The destination host can either be an end-point, or it can be an intermediate ATM switch on the path from the data's source to the eventual destination.

Data is transmitted over an ATM network in "ATM cells." A cell is a fixed-sized 53 byte data structure that contains 48 bytes of data and 5 bytes of control information. Each cell's control information includes a "virtual circuit" number. This number is used by ATM switches to determine where the cell should be sent, and it is used by receiving hosts to determine which process' buffers should receive the data (thus allowing inbound network data to be demultiplexed in hardware).

The virtual circuit number is composed of two numbers: the virtual channel identifier (VCI) and the virtual path identifier (VPI). All data sent over an ATM network is associated with a virtual circuit. There are two types of virtual circuits: permanent virtual circuits (PVCs) and switched virtual circuits (SVCs). Permanent virtual circuits are usually set up in an ATM switch by a network administrator. Switched virtual circuits are connections that are established "on demand" through the use of complex signaling protocols [2, 3].

The 48 byte data area of an ATM cell is quite small when compared to the data area of an ethernet or FDDI packet. To address this problem, ATM includes a number of "ATM adaptation layers" (or AALs). There are two AALs of interest: AAL0 and AAL5. AAL0 allows a host to send and receive individual ATM cells. AAL5 allows a host to send and receive frames up to 64K in size. When a host sends a large AAL5 frame, the ATM host-network interface segments it up into ATM cells. When the cells arrive at the receiving host, these cells are reassembled into a frame by the receiving machine's ATM host-network interface. AAL5 allows hosts to send and receive frames and not have to worry about small ATM cells.

One advantage of ATM is its use of virtual circuits makes it easier to provide network performance guarantees to applications. Each active virtual circuit on an ATM network can be allocated a fixed portion of the network's bandwidth. If a host attempts to exceed the allocated bandwidth for a virtual circuit, then the ATM switch may drop its cells rather than allowing the host to congest the network and effect other circuits. Many ATM host-network interfaces support "pacing" on transmission. Pacing allows a host to throttle the transmission rate of its ATM host-network interface so that it does not exceed the bandwidth allocated to it. An example ATM network is shown in Figure 1.

2.2. THE BSD NETWORKING SUBSYSTEM

The networking subsystem of the BSD kernel is composed of three layers: the socket layer, the protocol layer, and the network interface layer [10]. Transmitted data travels from an application through the socket and protocol layers to the network interface layer. Received data arrives at the network interface and is passed up towards the socket layer. All data in the networking subsystem is stored in a data structure called an "mbuf." There are two basic types of mbufs: small mbufs and large mbufs. Small mbufs contain about one hundred bytes of data and are used for small data or packet headers. Large mbufs typically contain either 2K or 4K of data. Mbuf structures can be linked together to form an "mbuf chain." Mbuf chains are used to

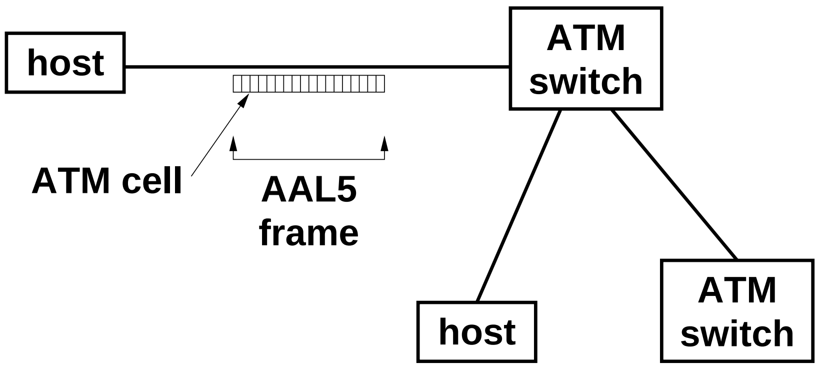

Figure 1: An ATM network. Data is being transmitted over virtual circuits in AAL5 frames composed of ATM cells.

store packets. Data can be added to either end of a packet by adding mbufs to the chain. Mbuf chains can be linked together to form a packet queue.

The socket layer has two main roles. First, it is responsible for transferring data between a user's address space and mbufs. Second, it is responsible for the queuing of data between the user and the kernel. If a process tries to transmit too much data and its socket buffer becomes full, then the socket layer will put the process to sleep until more room is available. Likewise, if a process tries to read queued data and none is available, then the socket layer may put the process to sleep until more data arrives.

All network protocol processing is done at the protocol layer. For example, TCP, UDP, and IP are all implemented in the BSD networking subsystem's protocol layer. When transmitting data the protocol layer receives data from the socket layer, adds the necessary protocol headers, and passes a packet for transmission to the network interface layer. When receiving data the protocol layer dequeues packets from its input queue, and determines the destination of each packet. If the packet is to be forwarded to another host, then the protocol passes it back to the network interface layer. If the packet is bound for a local process, then the protocol layer enqueues the packet on the receiving process' socket's receive buffer and notifies the socket layer that new data are available.

The network interface layer transfers packets between networking hardware and the protocol layer. When transmitting data the network interface layer receives packets through its interface queue and transmits them on the network. When receiving data the network interface layer determines which protocol to pass the inbound packet to, enqueues the packet for the protocol, and then schedules a software interrupt to service the protocol. The BSD network stack is shown in Figure 2.

2.3. INTEGRATING ATM NETWORKING INTO BSD

Integrating ATM networking into the BSD kernel required us to design a device-independent ATM networking layer and a device-specific driver for the Midway-based ATM cards. Key design features of our device-independent ATM networking layer include the following:

- In 4.4BSD, the ARP table was generalized and merged into the routing table. We took advantage of this change to allow ATM IP PVCs to be established using the new routing table architecture and the standard `route` command. Thus, our device-independent ATM layer does not have to emulate a traditional network interface such as ethernet and does not require any additional ATM-specific user-level administrative programs.

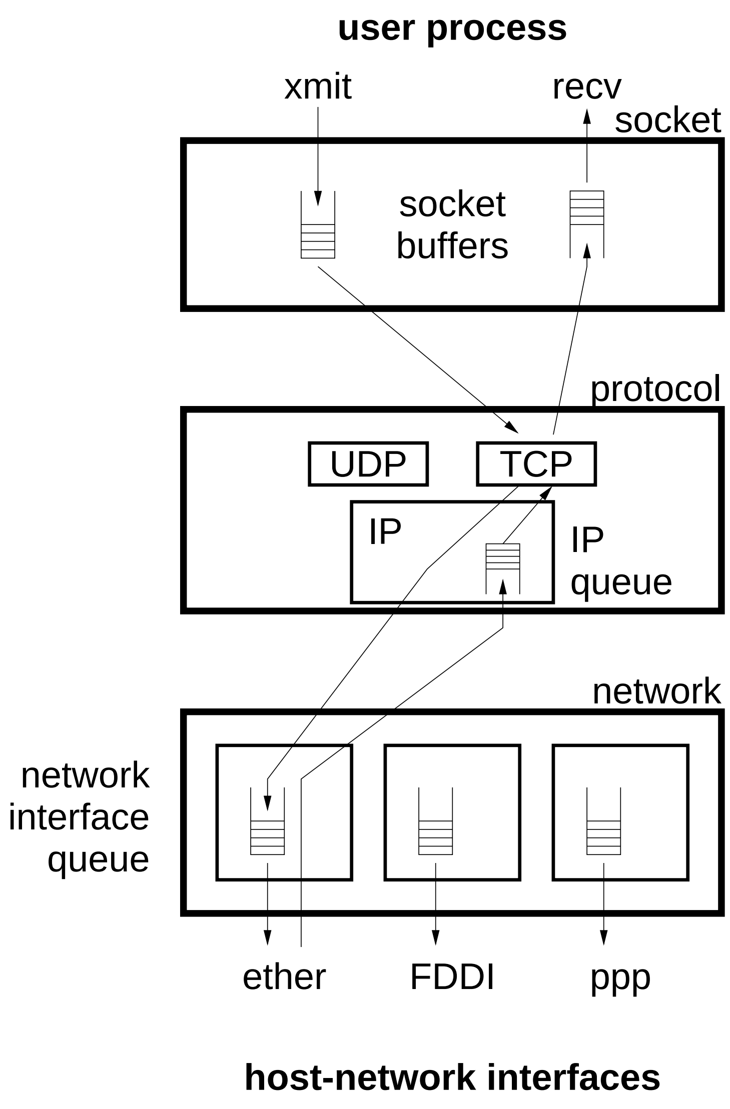

Figure 2: The BSD networking subsystem

- The design of our machine-independent ATM layer allows network protocols to take advantage of ATM's hardware-based early demultiplexing capability to avoid software-based connection lookups. This feature is enabled only if requested. This allows ATM to be used with existing protocol layers without requiring modifications to those protocol layers.
- Our design allows an ATM network interface to be used both for IP (over PVCs) and for direct socket-based I/O to ATM virtual circuits. Our native-mode ATM sockets allow non-privileged applications to directly access ATM virtual circuits using either AAL5 or AAL0. Note that we do not allow native-mode ATM sockets to be connected to virtual circuits that are already being used by IP. This prevents non-privileged processes from violating network security by taking over IP PVCs.

In addition to supporting these features, our device-specific Midway driver has been designed to use a transmit queuing structure that allows the networking system to take advantage of its pacing hardware so that virtual

Byte	Name	Usage
0	atm flags	0x01 - use AAL5 0x02 - use LLC/SNAP if AAL5 0x04,0x08,0x10,0x20 - unused 0x40,0x80 - reserved for device driver
1	VPI	the 8 bit VPI
2	VCI	the 8 high bits of the 16 bit VCI
3	VCI	the 8 low bits of the 16 bit VCI

Table 1: The ATM pseudo header structure

circuits that have different quality of services settings can be used. This queuing structure was designed so that the basic BSD protocol/network-interface software layering structure did not have to be changed.

3. DEVICE-INDEPENDENT ATM LAYER

In this section we present the design and implementation of our device-independent BSD ATM layer. This device-independent layer of BSD ATM provides support for using IP on ATM networks through PVCs and also provides support for “native” mode ATM sockets that give an application the power to send and receive either ATM cells or AAL5 frames. First we will describe an important data structure used throughout BSD ATM — the ATM pseudo header structure. Then we will describe how BSD ATM cleanly integrates support for IP through PVCs into the BSD kernel. Finally, we will describe the implementation and usage of native mode ATM sockets.

3.1. ATM PSEUDO HEADER STRUCTURE

We designed an ATM pseudo header structure (`atm_pseudohdr`) that is used to route ATM packets through the BSD networking subsystem. This four-byte header consists of the virtual circuit number (VPI and VCI) and a set of flag bits. Of the eight flag bits, two are used by the device-independent code, two are reserved for the device driver, and the rest are unused. The first device-independent flag bit indicates if AAL0 or AAL5 is being used. The other device-independent flag is valid only if AAL5 is being used. If this bit is set, then LLC/SNAP headers are used. LLC/SNAP headers allow us to include an “ethernet type” field in our packets so that multiple protocols can be multiplexed over the same virtual circuit.

The ATM header structure is a pseudo header rather than a normal header because it is only used within the BSD kernel to tag ATM packets. The header is removed from the packet before it is transmitted or passed to the user. The encoding of the ATM pseudo header is shown in Table 1. The four-byte ATM pseudo header can be displayed as an ordered list of four hexadecimal numbers. For example, an AAL5 virtual circuit on VCI 2, VPI 1 would be displayed as: `1.1.0.2`.

3.2. BSD ATM AND IP

In this section we discuss the method for establishing IP permanent virtual circuits and the flow of data through the device-independent layer of BSD ATM.

3.2.1. Establishing PVCs. Before IP can be used with BSD ATM, a PVC must be established in the ATM switch between the host and an end-point. This permanent circuit can be thought of as a point-to-point link. Once a PVC has been established in the network, then the host must be configured to use it.

The first step in configuring a host to use a PVC is to initialize the ATM host-network interface with the local host's IP address. This is done with the `ifconfig` command. For example, to initialize network interface "`en0`" one could use the command:

```
ifconfig en0 128.252.200.1 netmask 0xffffff00 up
```

This causes a `SIOCSIFADDR` `ioctl` to be passed down to the ATM host-network interface's device driver. The device driver will reset and initialize the card.

Once an ATM network interface has been initialized, PVCs must be added to the host's routing table. Each PVC is associated with the IP address of the host on the other end of the circuit. This is done with the `route` command. For example, if a virtual circuit with a VCI of 12 and a VPI of zero is established to host 128.252.200.18 using AAL5, then the following `route` command would be used to add the circuit to the routing table:

```
route add -iface 128.252.200.18 -link en0:1.0.0.c
```

Note that the first address is the IP address of the remote end of the virtual circuit and the second address is an ATM host-network interface and an ATM pseudo header structure. Once the route command has been issued, the virtual circuit can be used. Note that it is possible to display a list of active IP PVCs using the "`netstat -r`" command.

Internally, the `route` command causes a `RTM_ADD` request to be issued to the ATM route request function (`atm_rtrequest`). This function sanity checks the routing command, ensures that the requested virtual circuit is not in use by a native mode ATM socket, and then issues a `SIOCATMENA` `ioctl` to the ATM device driver. This `ioctl` causes the driver to enable the requested virtual circuit for receiving and transmitting data. Once the circuit is enabled the ATM routing request function completes the route and data can now be transfered over it.

IP PVCs can be shut down using a corresponding `route` delete command. This causes an `RTM_DELETE` request to be issued to the ATM route request function. The route request function issues a `SIOCATMDIS` `ioctl` to the ATM device driver to tell it that the specified virtual circuit will no longer be used. Once that is done the system removes the PVC route from the routing table and the connection is broken.

Note that all the actions required to establish and remove IP PVCs can be done with the standard BSD routing table functions, and the standard BSD `route` command. No new programs or changes to the BSD routing layer are required to support ATM. Furthermore, the ATM device does not have to masquerade as a traditional network device such as an ethernet.

3.2.2. Transmitting and Receiving IP Data. An application can transmit IP data over an ATM PVC using standard system calls such as `write` or `send`. This causes the socket layer to load the data into mbufs and pass it to the TCP protocol. The TCP protocol passes the data to the IP protocol and the IP protocol looks up the destination IP address in the routing table. In the case of an ATM PVC, the routing table lookup will return a route to an ATM device to the IP layer. The route contains a pointer to the ATM host-network interface's network interface structure (`ifnet`). IP transmits data by calling the network interface's "output" function. All ATM network interfaces share the same output function: `atm_output`.

The ATM output function checks the destination address to see what type of address it is. If it is an IP address, then the `atmresolve` function is called. The `atmresolve` function takes a route, a packet, and a destination IP address and returns the ATM pseudo header structure that corresponds to the target IP address. If a valid pseudo header cannot be found, then `atmresolve` returns an error code[1]. Once an ATM pseudo header has been found, then it is prepended to the front of the packet by `atm_output`. Then the packet is enqueued on the network interface queue of the ATM device and the device's "start" routine is called if the device is not already active.

When an ATM host-network interface receives data, it passes the received packet (in an mbuf chain) to the `atm_input` function. Note that the ATM pseudo header is passed as a separate argument to the `atm_input` function. The ATM input function determines which protocol the packet should be given to. It then enqueues the packet on that protocol's queue and schedules a software interrupt so that the protocol can process its new data. For IP data, the `atm_input` places the packet on the IP queue. At that point all ATM-specific processing on the packet has been completed and the IP protocol can treat the packet the same way as it treats packets from other network interfaces such as ethernet or ppp.

3.3. NATIVE MODE ATM SOCKETS

In addition to supporting IP over PVCs, BSD ATM also supports native mode ATM sockets, or NATM. NATM allows a process to create a socket that is bound directly to an ATM virtual circuit. Either AAL0 or AAL5 can be used. There are no protocols between the socket and the network. This allows direct access to ATM. NATM sockets can be used to prototype signaling software or interact directly with ATM devices. For example, NATM can be used to program the configuration of virtual circuits in an ATM switch by passing control cells to a special port on the switch. NATM can also be used for ATM-based multimedia devices. For example, Washington University's MMX device can send and receive audio and video data streams using AAL0. BSD ATM has been used with the MMX to build a networked video server [5, 6]. In this section we present the NATM API, and the design and implementation of BSD ATM's NATM protocol layer.

3.3.1. NATM API.

NATM is accessed using standard BSD socket-based system calls. NATM has its own address family (`AF_NATM`). NATM socket addresses are specified using this family and the `sockaddr_natm` data structure. The NATM socket address structure is defined as:

```
struct sockaddr_natm {
  u_int8_t      snatm_len;              /* length */
  u_int8_t      snatm_family;           /* AF_NATM */
  char          snatm_if[IFNAMSIZ];     /* interface name */
  u_int16_t     snatm_vci;              /* vci */
  u_int8_t      snatm_vpi;              /* vpi */
};
```

To create an AAL5 connection on network interface "`en0`" to a virtual circuit with a VPI of zero and a VCI of 201 the following code would be used:

```
struct sockaddr_natm snatm;
int s, r;
s = socket(AF_NATM, SOCK_STREAM, PROTO_NATMAAL5);
                      /* note: PROTO_NATMAAL0 is AAL0 */
if (s < 0) { perror("socket"); exit(1); }
bzero(&snatm, sizeof(snatm));
snatm.snatm_len = sizeof(snatm);
```

[1]To provide support for SVCs, `atmresolve` could be modified to perform ATM ARP.

Name	**Usage**
`pcblist`	pointers used to maintain the list of NATM PCBs
`npcb_inq`	the number of inbound packets for this connection waiting to be read; used when closing an NATM socket to determine when the final packet has been read and the PCB can be freed
`npcb_socket`	back-pointer to the PCB's socket structure (if any); PCBs for IP connections are not associated with any sockets
`npcb_ifp`	pointer to the network interface associated with this PCB
`ipaddr`	the IP address of the remote end of the circuit (IP only)
`npcb_vci`	the VCI
`npcb_vpi`	the VPI
`npcb_flags`	flags (free, connected, IP, drain, raw)

Table 2: NATM protocol control block

```
snatm.snatm_family = AF_NATM;
sprintf(snatm.snatm_if, "en0");
snatm.snatm_vci = 201;
snatm.snatm_vpi = 0;
r = connect(s, (struct sockaddr *)&snatm, sizeof(snatm));
if (r < 0) { perror("connect"); exit(1); }
/* s now connected to ATM! */
```

The call to `socket` creates an unconnected NATM socket. To associate the socket with a virtual circuit, the `connect` system call is used. The NATM socket address structure contains the interface name and virtual circuit number that the application wishes to connect to. Once the `connect` call has returned successfully, the application may read and write on the socket file descriptor to perform ATM I/O. Note that for AAL0 NATM sockets, the application should read and write using the proper cell data size of 48 bytes.

3.3.2. NATM Implementation. NATM is implemented as a lightweight "protocol" in the BSD protocol layer. NATM manages a linked list of all active ATM virtual circuits on the system. Each active circuit has its own protocol control block structure (`natmpcb`). The content of the protocol control block structure is shown in Table 2. Not only do these protocol control blocks store state information for a connection, but they are also used to prevent NATM and IP over PVCs from attempting to use the same virtual circuit.

When an NATM socket is created with the `socket` system call, a new PCB is allocated and installed on the list of active PCBs. The new PCB remains in an unconnected state (i.e. not associated with any circuit) until a `connect` call is issued.

When a `connect` call is issued on a NATM socket, the NATM protocol layer first verifies that the specified virtual circuit is both valid and available. If so, the PCB is put in a connected state and the circuit on the corresponding ATM network device is enabled (using the `SIOCATMENA` `ioctl`).

When data is sent through the NATM protocol layer, it prepends a ATM pseudo header to the mbuf chain containing the data and then passes the packet on to the `atm_output` function. The `atm_output` function can easily detect packets from the NATM layer and knows that it does not need to add an ATM pseudo header itself.

An NATM socket is discarded by closing all file descriptors that refer to it. This causes the NATM protocol layer to issue a `SIOCATMDIS` `ioctl` to the ATM network device to disable the virtual circuit. If an NATM socket is discarded while there are still inbound packets queued for it, then the PCB is placed in the "draining" state. Otherwise it is freed immediately. PCBs remain in the draining state until all their inbound packets are removed (and discarded) from the NATM protocol queue. Once the packets are discarded, the draining PCB can be freed. The number of inbound packets in the protocol queue is counted in the PCB's `npcb_inq` variable.

3.3.3. NATM Special Features. The NATM protocol layer supports two special features that are implemented within the framework of the BSD networking subsystem. These features are fast PCB lookup and "raw" mode.

When data arrives on a virtual circuit for a NATM socket the NATM protocol layer must determine which socket's buffer to place the data in. A pointer to the needed socket is stored in that circuit's PCB. One way to find the PCB is to perform a linear search of the list of active PCBs searching for a match on the virtual circuit and device pointer. A linear search could be wasteful if the number of active circuits is large, but fortunately it is not necessary. When a virtual circuit is enabled for receiving data with the `SIOCATMENA` `ioctl`, the NATM protocol layer passes the address of the PCB down into the device driver as a "receive handle." When inbound data arrives, the ATM hardware automatically demultiplexes it into separate buffers for each virtual circuit. The device driver can take advantage of this to easily lookup the receive handle associated with a virtual circuit and pass it back up to to the NATM protocol layer. Since the NATM receive handle is the address of the PCB, no lookup is required.

The NATM "raw" mode is a special feature used to avoid the overhead of using AAL0 to receive high-bandwidth video data. Our multimedia MMX device transmits real-time video data using a stream AAL0 cells. Each cell received generates an interrupt. If we were to process each cell of a high-bandwidth video stream individually our processor would get swamped. The "raw" mode is used to overcome this problem. When "raw" mode is enabled, the driver allows a cluster of AAL0 frames to accumulate before attempting to pass them up to the socket layer. The advantage of this is that we can use raw mode to safely record real-time video data to disk. The disadvantage of raw mode is that it leaves the AAL0 cell header information in the data stream, and thus the data must be cleaned up before it can be played back. Raw mode is enabled with the `SIOCRAWATM` `ioctl`. For example:

```
int size = 4000; /* bytes */
ret = ioctl(s, SIOCRAWATM, (caddr_t)&size);
```

We hope that the next generation of multimedia devices uses AAL5 so that raw mode will no longer be necessary.

Note that although the low-level BSD ATM driver supports setting the maximum bandwidth of a connection, the NATM API does not currently export this functionality to the application. We have already implemented a test API to do this, and we expect to add this feature to BSD ATM sometime in the future.

4. DEVICE-DEPENDENT ATM LAYER

In this section we examine the internal workings of the device-dependent aspects of BSD ATM. Currently, the only ATM cards supported by BSD ATM are the ones based on the Efficient Networks "Midway" ATM chipset. Such cards are manufactured by Efficient and Adaptec.

4.1. DRIVER STRUCTURE

The Midway driver is divided into two layers: a bus-specific layer and a generic chip-level layer. This allows the same driver to be used by Midway-based devices on different bus types. For example, there is both an SBUS and a PCI bus-specific layer for the Midway chip. All chip-level registers are accessed using the "bus_dma" interface developed for NetBSD [13]. This allows a generic chip-level driver to compile and run on multiple systems including the i386 and DEC Alpha. The structure of the bus_dma interface allows us to easily emulate it on systems that do not directly support it.

The bus-specific layer of the driver is usually stored in a file named `if_en_pci.c` (for PCI), and the generic chip-level layer is stored in `midway.c`. In addition to these two files, there are two header files that are used. The `midwayreg.h` file contains definitions related to the hardware, while the `midwayvar.h` file contains software-related definitions.

Note that all Midway state information is stored in the per-device softc structure (`en_softc`).

4.2. CHIP INITIALIZATION

At system initialization time, the Midway device driver first determines how much on-board memory the card has. It then puts the card through a rigorous set of DMA test to ensure that the card is functioning properly and to determine what level of DMA the host system supports. For example, some systems only support small DMA transfer burst sizes and some architectures support DMA only if the buffer is aligned in user memory. After performing the DMA tests, the driver will report the results:

```
en0: maximum DMA burst length = 64 bytes
```

We have found that most of our i386 systems support 64 byte bursts on any alignment. However, we have also had reports of some older i386 PCI chipsets being unable to handle all sizes properly[2].

Once the card's DMA system has been tested, then the on-board memory is divided into circular buffers for transmit and receive. The number and size of these buffers are controlled by defines in `midwayvar.h` and can be overridden through defines in the kernel config file. The driver prints the number and size of the transmit and receive buffers after it prints out the maximum DMA burst length. Note that the number of active virtual circuits is limited by the number of receive buffers. The transmit buffers are shared among all circuits. If pacing (traffic shaping) is not being used, then only one transmit buffer is needed (i.e. it is safe to set `EN_NTX` to one).

4.3. TRANSMITTING DATA

To transmit data, the protocol layer enqueues an mbuf chain on the network device's input queue and calls the device start routine. Normal network interfaces process this queue in FIFO order, leaving mbufs on the queue until they are transmitted. However, since the Midway driver supports pacing, it cannot do this. The Midway chip can send multiple frames at the same time at different rates. If the driver were to process the queue in FIFO order, then a packet at the front of the queue bound for a low-bandwidth connection could block other connections from being serviced.

To avoid this problem, the Midway driver maintains its own set of queues for transmitting data. Each transmit channel has its own queue. Data that is not being paced is always routed to transmit channel zero. This channel is the lowest priority and gets whatever bandwidth is left over from the other channels. Note

[2]This can be detected when the driver prints warnings about DMA timeouts when booting, but then functions properly after the DMA burst length has been configured.

that if pacing is not being used, then all transmitted data is routed through channel zero. Note that the driver having its own transmit queues defeats the "full queue" checks used by the protocol layer when queuing data for a network device. In order to avoid having the ATM device use too much memory, the queue full checks are duplicated for the device driver's private queues.

When the driver's start routine is called it immediately removes the outbound packet from the network interface queue and inspects the packet's ATM pseudo header to determine which transmit channel to enqueue the packet on. Next the mbuf chain's data areas are checked for proper alignment (based on the DMA test performed at system startup). If the mbuf is not properly aligned, then the driver must align it by allocating new mbufs and copying the data. Well behaved protocols should usually generate properly aligned mbufs.

Once the mbuf's alignment has been checked, then the driver attempts to insert a transmit buffer descriptor (TBD) at the front of the packet. The TBD is read by the hardware to determine the size and destination of the packet. If there is no room for a TBD in the mbuf, then we will add it to the packet after it has been DMA'd to the card's on-board memory (but it will cost us a few extra DMA steps). Likewise, the driver also attempts to append a trailer to the data so that it is the proper length. At this point the packet is enqueued on a transmit channel's input queue and the `en_txdma` function is called on that channel.

The `en_txdma` function first checks to see if there is room in the transmit buffer in on-board memory for the packet. If there is not enough room, then nothing is done until more memory becomes available. If there is room, then the `en_txdma` function examines the mbuf chain to determine how many transmit DMA descriptors (DTQs) will be needed to DMA the packet into on-board memory. If there are not enough free DTQs, then nothing is done until more DTQs become available. If there are enough DTQs, then the packet is removed from the input queue and passed to the `en_txlaunch` function.

The `en_txlaunch` function programs the Midway chip to start DMAing the packet from host memory to the card's memory and then places the packet on the "indma" queue. The `en_txlaunch` can also be configured to use data copying rather than DMA if necessary. At this point the transmit process is complete, and the packet will be transmitted as soon as it is completely in the card's memory. Note that the `en_txlaunch` function does *not* DMA the ATM pseudo header into the card's memory.

Once the packet has been completely DMA'd into the card's memory the card generates a "transmit complete" interrupt for the channel on which the data was transmitted. To service this interrupt, the card removes the mbuf chain from the "indma" queue and frees it. It then checks to see if any packets are waiting for memory to become available so that they can be transmitted. If so, the driver calls `en_txdma` to start processing on the next packet.

4.4. RECEIVING DATA

When a Midway card receives a complete AAL5 frame or an AAL0 cell into its on-board memory it puts the virtual circuit on a hardware managed "service list" and generates a "receive" interrupt. The driver's interrupt handler responds to this by taking the virtual circuit off the hardware service list and placing it on a software managed service list.

The software service list is needed in case there is a shortage of memory resources. In that case the driver must defer processing until more memory is available by leaving the virtual circuit on the software service list. Note that it is not possible to put a virtual circuit back on the hardware service list once it has been removed. It is also not possible to "peek" at the first virtual circuit on the hardware service list without removing it because it would create a race condition. If the driver peeks at the first virtual circuit on the hardware service list and decides that it is finished with the virtual circuit, new data could arrive between the time the driver decides it is done with the virtual circuit and the time it removes the virtual circuit from the hardware service list. This will result in the driver losing an interrupt and delaying the delivery of the data.

Flag name	Flag value	Usage
`STATS`	0x01	dump statistic counters
`MREGS`	0x02	dump Midway registers
`TX`	0x04	dump transmit state
`RX`	0x08	dump receive state
`DTQ`	0x10	dump DTQ DMA state
`DRQ`	0x20	dump DRQ DMA state
`SWSL`	0x40	dump software service list

Table 3: Dump function flags

Once the driver has placed a virtual circuit on the software service list, it calls `en_service` to process it. The `en_service` function traverses the software service list allocating mbuf chains for each frame and then programming the Midway card to DMA the data from on-board memory to host memory. Like the `en_txdma` function, the `en_service` function must handle memory shortages. The system may be out of mbuf memory or the Midway card may have a shortage of receive DMA descriptors (DRQs). If the system is out of mbuf memory then the `en_service` function attempts to drop the packet by setting up a DRQ that discards the data. However, if there are also no DRQs available, then the driver is stuck and leaves the virtual circuit on the software service list for later processing. If the driver is able to allocate an mbuf chain for the received data but is short on DRQs, then the allocated mbuf chain is stored on a queue until DRQs become available.

On the other hand, if the driver allocates an mbuf chain and has enough DRQs to DMA the data into it, then it programs the Midway chip to start DMAing data from on-board memory to host memory. The mbuf chain receiving the data is placed on the receive “indma” queue. If all inbound packets for a virtual circuit are processed, then the driver removes that circuit from the software service list. At this point the service interrupt has been processed.

Once the receive DMA operation completes the Midway chip will generate a “receive DMA complete” interrupt. This causes the driver to walk down the list of completed DRQs searching for the final DRQ for a packet. When detected, the packet is pulled off the “indma” queue for the receive virtual circuit and passed to `atm_input`. Note that the driver passes the “receive handle” up to `atm_input` so that fast PCB lookups can be used for NATM sockets. Once packets have been passed upward, the driver checks to see if there were any received packets waiting for DRQs to become free. If so, it starts processing on those packets.

4.5. TROUBLESHOOTING

The Midway driver has a diagnostic function called `en_dump`. The `en_dump` function takes two arguments: a unit number and a set of flags. If the unit number is zero, then it will dump stats on “`en0`.” A unit number of minus one causes `en_dump` to dump stats for all Midway devices on the system. The flag bits are defined in Table 3. Note that the `en_dump` function was designed to be called from DDB. The statistic counters are useful in finding out where packets are being dropped.

5. CONCLUSION

In this paper we have presented the design and implementation of BSD ATM. We have shown how ATM can be integrated cleanly into the BSD kernel by taking advantage of the flexibility of the 4.4BSD routing table interface. Our design does not force the ATM interface to emulate an ethernet interface, requires minimal changes to existing networking code, and requires no new user-level programs. In addition to supporting IP-based protocols over PVCs, BSD ATM also supports native mode ATM sockets. This allows researchers to use normal socket-based programs to directly access an the ATM network.

Our Midway ATM device driver is robust and fault tolerant. It gracefully handles error conditions such as running out of interface memory, running out of mbuf memory, and running out of DMA descriptors. The Midway driver avoids data copying by DMAing data directly into and out of mbufs. We have extensively tested the Midway driver both with IP-based protocols and with our MMX multimedia device.

BSD ATM performs well. We have observed bandwidths of up to 133 Mbps with IP-based protocols on a 155 Mbps ATM link [4]. BSD ATM is currently being used in a number of projects around the world. For example, BSD ATM is the basis for the ATM software used in the Collaborative Advanced Interagency Research Network (CARIN) testbed [12]. BSD ATM is also one of the building blocks used in Sony's alternate queuing framework (ALTQ) research [8]. At Washington University we are using BSD ATM in our network video server project [5, 6]. We are also using BSD ATM in the control processor of our next generation ATM switch (WUGS) [7]. Rather than having a special purpose control processor, the WUGS switch will be controlled by standard PC running BSD ATM. The PC will be connected to a special port on the switch via an ATM link, and it will communicate and configure the switch through a NATM socket-based control program. The WUGS switch, along with BSD ATM, will be distributed as part of the Nation Science Foundation's (NSF) Gigabit Network Kit program [14]. BSD ATM is easy to install, and is now a standard part of the FreeBSD, NetBSD, and OpenBSD operating systems.

ACKNOWLEDGMENTS

I would like to thank Kenjiro Cho for assisting with the FreeBSD port of BSD ATM. I would also like to thank Anne Hutton and Martin Brundage for assisting with porting the Midway driver to Adaptec cards.

References

[1] Adaptec, Inc. ANA-59x0 PCI ATM cards. See `http://www.adaptec.com` for more information.

[2] ATM Forum. *The ATM Forum Technical Committee Private Network-Network Interface (PNNI) Specification Version 1.0*. The ATM Forum, 1996.

[3] ATM Forum. *The ATM Forum Technical Committee User-Network Interface (UNI) Specification Version 4.0*. The ATM Forum, 1996.

[4] ISI Atomic2 Project. Adaptec ATM measurements. `http://www.isi.edu/div7/atomic2/adaptec-measure.html`.

[5] M. Buddhikot. *Project MARS: Scalable, High Performance, Web based Multimedia-On-Demand (MOD) Services and Servers*. PhD thesis, Washington University, August 1998.

[6] M. Buddhikot, X. Chen, D. Wu, and G. Parulkar. Enhancements to 4.4 BSD UNIX for networked multimedia in project MARS. In *Proceedings of IEEE Multimedia Systems'98*, June 1998.

[7] T. Chaney, A. Fingerhut, M. Flucke, and J. Turner. Design of a gigabit atm switch. Technical Report WUCS-96-07, Washington University, 1996.

[8] K. Chao. A framework for alternate queueing: Towards traffic management by PC-UNIX based routers. In *Proceedings of USENIX Conference*. USENIX, 1998.

[9] Efficient Networks, Inc. ENI-155 PCI ATM cards. See `http://www.efficient.com` for more information.

[10] M. McKusick, K. Bostic, M. Karels, and J. Quarterman. *The Design and Implementation of the 4.4BSD Operating System*. Addison Wesley, 1996.

[11] W. Richard, J. Cox, A Engebretson, J. Fritts, B. Gottlieb, and C. Horn. The Washington University multimedia explorer. Technical Report WUCS-93-40, Washington University, 1993.

[12] The CARIN Testbed. Carin: Contributed software. `http://www.isi.edu/div7/atomic2/adaptec-measure.html`.

[13] J. Thorpe. A machine-independent DMA framework for NetBSD. In *Proceedings of USENIX Conference (FREENIX track)*. USENIX, 1998.

[14] Washington University and the National Science Foundation. Gigabit network kits. `http://www.arl.wustl.edu/ jst/gigatech/cfp.html`.

A. CONFIGURATION

To configure BSD ATM into a kernel, add one of the following lines to the kernel's config file:

```
en*     at sbus? slot ? offset ?        # NetBSD SBUS
en*     at pci? dev ? function ?        # NetBSD, OpenBSD PCI
device en0                              # FreeBSD PCI
```

To configure NATM into a kernel, add the following like to the kernel's config file:

```
options NATM
```

If all goes well the kernel should print something like this during autoconfig:

On NetBSD or OpenBSD with a PCI device:

```
en0 at pci0 dev 14 function 0
en0: interrupting at irq 3
en0: ATM midway v0, board IDs 6.0, Utopia (pipelined), 512KB on-board RAM
en0: maximum DMA burst length = 64 bytes
en0: 7 32KB receive buffers, 8 32KB transmit buffers allocated
```

NetBSD with an SBUS device:

```
en0 at sbus0 slot 2 offset 0x0
en0: claims to be at the following IPLs: 16 1 2 3 5 7 9 13
en0: we choose IPL 5
en0: midway v0, board IDs 0.2, SUNI, 512KB on-board RAM
en0: maximum DMA burst length = 4 bytes
en0: 7 32KB receive buffers, 8 32KB transmit buffers allocated
```

FreeBSD with a PCI device:

```
en0 <Efficient Networks ENI-155p> rev 0 int a irq 5 on pci0:16
en0: ATM midway v0, board IDs 6.0, Utopia (pipelined), 512KB on-board RAM
en0: maximum DMA burst length = 64 bytes
en0: 7 32KB receive buffers, 8 32KB transmit buffers allocated
```