Linux Multilayer Switching with LiSA

Radu Rendec, Ioan Nicu and Octavian Purdila

***Abstract* — LiSA stands for Linux Switching Appliance. It's an entirely open-source project that aims at delivering a cheap and efficient solution to small-sized networks. LiSA implements both Layer 2 and Layer 3 packet switching. At the same time, it has advanced configuration and monitoring features. Based on the Linux Operating System, packet switching is entirely accomplished in software. As opposed to a hardware implementation, LiSA aims at providing advanced features rather than high bandwidth and low latency. Being based on the Linux Kernel, LiSA can run on any architecture supported by this kernel.**

***Keywords* — Linux Kernel, Multilayer Switching, Operating Systems .**

I. Introduction

THIS paper contains a brief presentation of LiSA's currently implemented features, a comparison between LiSA and existing Linux Kernel modules that aim at similar functionality, a detailed description of LiSA's architecture and kernel integration and results of our own performance tests.

II. LiSA features overview

LiSA currently implements Layer 2 and Layer 3 packet switching. The central point for control and management is a user space command line interface (CLI). It was designed to resemble as close as possible Cisco's IOS CLI.

Full 802.1q support is implemented, including a VLAN database, port trunks and a list of allowed VLANs. Because it was meant as an IOS clone, the IOS approach to VLAN assignment is used:

- VLANs are assigned to ports, using a port-centric configuration;
- ports have two VLAN-operating modes: access (untagged) and trunk (802.1q tagged);
- each trunk port has a list of allowed VLANs.

VLAN management was adapted from newer versions of IOS: it's handled through the config/vlan prompt rather than an entirely separate configuration area called "VLAN Database".

At Layer 2, no support for STP, CDP and IGMP snooping is currently available (although we plan to implement these during the next months).

At Layer 3, only simple packet routing (i.e. using the existing kernel routing code) is supported, with no routing management available through the command line interface.

Access lists, policy maps and routing protocols are not implemented (although further development, which should include such features, is expected). Similar to the IOS, VLAN-specific virtual interfaces can be dynamically created. Layer 3 addresses can be assigned to these virtual interfaces, and thus inter-VLAN routing occurs naturally.

III. LiSA vs. Bridge + 8021q

Judging by its currently implemented features, one might say that LiSA is useless, since similar functionality can be achieved with two existing kernel modules, if they are cleverly used together. The two modules are *bridge* and *8021q*.

The *bridge* module combines several network devices into a classic switch. It also provides a virtual interface, which can be assigned an ip address. In terms of the bridge, the virtual interface is just another port to it. In terms of the host, the virtual interface behaves like a physical interface connected to one of the switch's ports [1].

The *8021q* module turns a network device into a VLAN-aware interface. One virtual interface is created for each VLAN. Each virtual interface sends and receives untagged packets. When these packets are sent or received through the network device, 8021q tags are appropriately processed, so that packets appear to be untagged on the virtual interfaces [1].

IV. Port Trunk with Bridge + 8021q Example

As stated earlier, similar functionality can be achieved with two existing kernel modules, if they are cleverly used together. Here's a short, eloquent example:

Suppose we had a machine with 4 network interfaces, and wanted to configure these interfaces as follows:

- eth0 in trunk mode, allowing access to VLANs 1 and 2;
- eth1 in trunk mode, allowing access to VLANs 1 and 3;
- eth2 in access mode, in VLAN 2;
- eth3 in access mode, in VLAN 3.

Additionally, routing must be accomplished between VLANs 2 and 3, using address 192.168.2.254/24 on

Radu Rendec was with the Faculty of Computers and Automatic Control, "Politehnica" University of Bucharest, Romania. He is now with iNES Group SRL, Virgil Madgearu 2-6, 2322112 Bucharest, Romania; (e-mail: radu.rendec@ines.ro).

Ioan Nicu was with the Faculty of Computers and Automatic Control, "Politehnica" University of Bucharest, Romania. He is now with iNES Group SRL, Virgil Madgearu 2-6, 2322112 Bucharest, Romania (e-mail: ioan.nicu@ines.ro).

Octavian Purdila is with the Faculty of Computers and Automatic Control, "Politehnica" University of Bucharest, Romania (e-mail: tavi@cs.pub.ro).

VLAN 2 and address 192.168.3.254/24 on VLAN 3.

The command sequence that builds up this configuration is:

```
modprobe bridge
modprobe 8021q

vconfig add eth0 1
vconfig add eth0 2
vconfig add eth1 1
vconfig add eth1 3

brctl addbr br1
brctl addif br1 eth0.1
brctl addif br1 eth1.1

brctl addbr br2
brctl addif br2 eth0.2
brctl addif br2 eth2

brctl addbr br3
brctl addif br3 eth1.3
brctl addif br3 eth3

ifconfig br2 192.168.2.254 netmask 255.255.255.0
ifconfig br3 192.168.3.254 netmask 255.255.255.0

ifconfig eth0 up
ifconfig eth1 up
ifconfig eth2 up
ifconfig eth3 up
```

This approach has two major drawbacks:

- One virtual interface is necessary for each VLAN of each trunk mode port. If all 4094 VLANs must be switched between two trunk ports, a total of 8188 virtual interfaces are needed for VLAN support only.
- When a packet is flooded / multicast and both trunk mode and access mode ports are involved, the same tagging / untagging operations take place several times. This also implies additional overhead with socket buffer operations.

LiSA uses its own kernel module to accomplish these tasks. Since it was meant for an intelligent Layer 2 switch right from the beginning, several optimizations have been made to mitigate these drawbacks:

- LiSA needs no virtual interfaces at all to handle per-VLAN packet switching.
- When a packet is flooded / multicast, ports are "re-ordered" so that a minimum number of socket buffer operations are performed.

Virtual interfaces are still used to provide compatibility with existing Layer 3 code and thus allow for easy inter-VLAN routing. However, these interfaces were designed in such a manner that they have minimum overhead (basically they just add a few more stack levels).

V. User Space Architecture

Right from the beginning, LiSA was designed bearing dedicated systems in mind. Ideally, embedded systems should be used, but this does not mean it cannot run on standard PC architectures. Because of the dedicated systems idea, several flavors of CLI were designed:

The *swcon* binary is meant to be indirectly spawn by init (by the means of an *agetty* or similar program). Ideally, it would be spawn on a serial port to resemble a

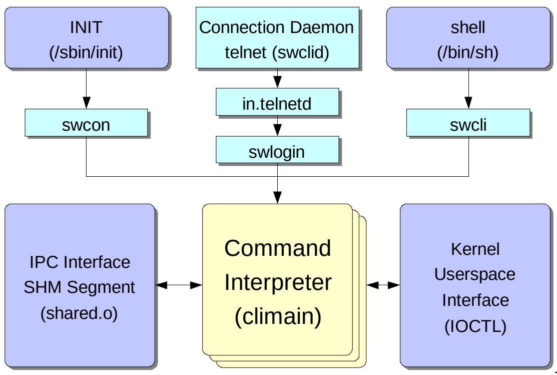

Figure 1. LiSA User Space Architecture

dedicated switch's serial console.

The *swclid* daemon is a very light version of the famous inetd daemon. It only listens on a single TCP port (usually telnet) and indirectly (by the means of in.telnetd) spawns the *swlogin* binary.

The *swcli* binary directly opens a highest privilege level CLI. It's mostly meant for testing purposes or shared (i.e. not dedicated) hosts.

Once authentication is completed, all binaries must provide the same functionality. Therefore, all CLI functionality is actually implemented in a common library (object) rather than the binaries themselves.

Because several management sessions (CLI instances) can run simultaneously, configuration must be shared between them. All switching control and configuration data is stored in kernel space, and thus it's shared by default. However, some data (such as the user and enable passwords) are not fitted to be stored in kernel space. Instead, IPC mechanisms are used to serve this purpose. A shared memory segment is used to store shared information in userspace. Consistent access to the shared memory segment is accomplished by means of a semaphore.

VI. Kernel Space Architecture

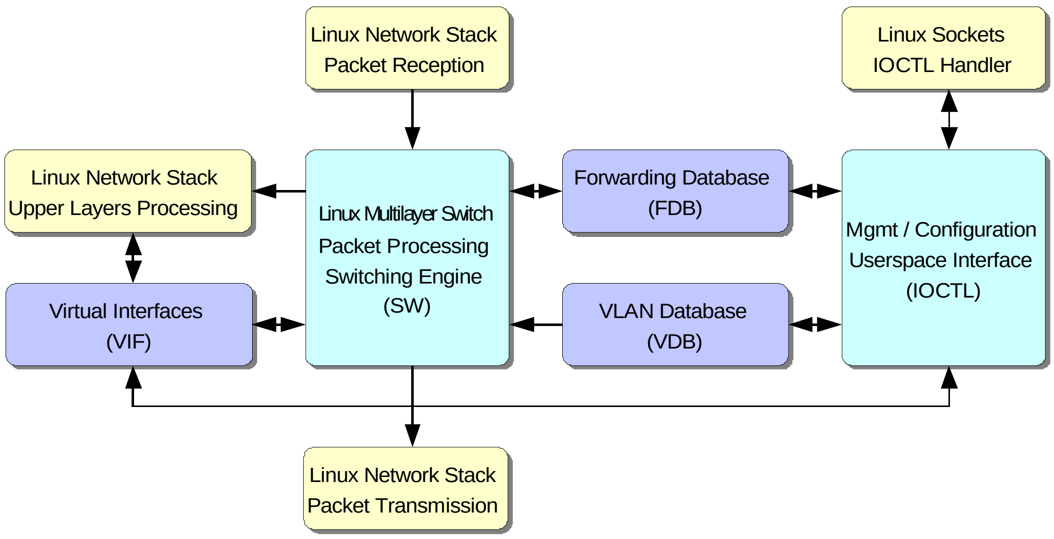
Linux Network Stack Packet Reception
Linux Sockets IOCTL Handler
Linux Network Stack Upper Layers Processing
Linux Multilayer Switch Packet Processing Switching Engine (SW)
Forwarding Database (FDB)
Mgmt / Configuration Userspace Interface (IOCTL)
Virtual Interfaces (VIF)
VLAN Database (VDB)
Linux Network Stack Packet Transmission

Figure 2. LiSA Kernel Space Architecture

When we first thought of implementing a software switch, we wanted to write our own dedicated kernel. Soon we realized this was a very bad idea and quickly moved on to using the linux kernel [2]-[3]. This approach proved to be much better because:

- we no longer needed to write a scheduler, a memory manager and a synchronization framework;
- device drivers existed for a variety of network chipsets and many system architectures were already supported by the core kernel components – much better portability, for short;
- the linux kernel NAPI and socket buffers are a very clever solution to efficiently dealing with network packets [4] – [5];
- the IP stack and routing were already implemented;
- the LiSA system no longer needed to be dedicated; any PC would just do, and it could also be used for other tasks at the same time.

We tried to use as much as possible from the linux kernel, without affecting switching performance. As we have already shown, existing modules have severe performance issues when it comes to trunk ports.

The LiSA kernel module is made up of five parts:

- Switching Engine – the very heart of the whole thing, which actually deals with packets;
- Forwarding Database – information on where packets should be forwarded based on their MAC address;
- VLAN Database – list of all configured VLANs;
- Virtual Interfaces – a clever trick to reuse the existing linux kernel code when packets must be "switched" at Layer 3;
- Userspace Interface – implements all the IOCTL calls.

Just like the bridge module, LiSA hooks into the NAPI packet reception code [5]. The hook was added between generic packet handler processing[1] and protocol-specific packet handler (upper layer) processing.

When a packet is received by the network driver and passed to the kernel for processing, it is first handed to all registered generic handlers. Then it is passed to the Switching Engine. If the incoming interface is a switch member, the packet will be "absorbed" by the Switching Engine, and it will never reach the rest of the linux packet reception code. On the other hand, if the incoming interface is not a switch member, the packet will be handed off to the rest of the kernel reception routine.

The Switching Engine examines the Forwarding Database and the VLAN Database and decides the fate of the packet. It can either be:

- discarded, if it's not legitimate according to the current configuration;
- forwarded to one or more ports; or
- forwarded to a Virtual Interface.

When a packet is forwarded to a port, the socket buffer is placed directly into the interface's outgoing queue. This is right, because the Switching Engine always places an exclusive copy of the original socket buffer[2] into the queue, and the driver will eventually free the socket buffer after successful transmission.

When a packet is forwarded to a Virtual Interface, the NAPI packet reception routine is explicitly called and the packet is handed off to it as if it were received on a physical interface. The packet will eventually end up in upper layer processing. Because of this approach, Virtual Interfaces have no RX queue, no interrupt handlers and no poll method.

The packet transmission method of Virtual Interfaces

[1] Generic packet handlers are called for all incoming packets. All libpcap based traffic analysis tools (such as tcpdump) use such a handler to capture incoming packets.

[2] Actually it could be either the original socket buffer, a clone of it (the data segment is shared with the original socket buffer) or a full copy (both the sk_buff structure and the packet data are copied).

does nothing but inject the packet (the socket buffer) in the Switching Engine. This is perfectly safe because the switching decision adds very little overhead and the socket buffer quickly ends up in a physical interface outgoing queue.

VII. Minimum Data Copy Optimization

At the best, no generic packet handler is registered and the Switching Engine needs to send the packet to only one physical interface. In this case the socket buffer could be sent right away, or freely modified if the tagging on the incoming interface differs from the tagging on the outgoing interface[3].

Although this is what happens most frequently, it's not the only possible case. For instance, a socket buffer may have already been passed to one or more generic packet handlers when it reaches the Switching Engine. And it also must be sent to several ports, of which some are tagged and some untagged. This is quite a difficult task to accomplish in an efficient manner.

Socket buffers are fragile and delicate things. For the sake of efficiency, the kernel makes no sanity checks when you modify them. It's the programmer's task to wisely modify socket buffers without breaking anything. Basically, whenever a socket buffer needs to be touched, the programmer must make sure it's an exclusive copy. If it's not, then it is the programmer who must explicitly create an exclusive copy [4].

When a packet is sent to several ports, each port must be sent a different socket buffer. No one can guarantee that all network chips will have finished sending the packet before the first driver frees the socket buffer. Moreover, drivers may need to change some fields in the sk_buff structure, so they need an exclusive copy. Fortunately, the packet data can be shared and only the sk_buff structure itself needs to be copied.

We've worked hard to design an efficient algorithm. There are several rules that led us to the current implementation:

- First send the packet to all the ports that have the same tagging as the incoming port and then change the packet and send it to the other ports. This way the packet is tagged/untagged only once.
- If a packet needs to be tagged/untagged, copy it only if the socket buffer is used by someone else (it's not an exclusive copy).
- If a socket buffer is copied for tagging/untagging purposes, it doesn't need to be cloned again before sending it to the next outgoing port (unless there are more ports to send it to).

Using these rules, we designed an algorithm that makes the minimum number of operations: tagging/untagging, cloning (copying only the sk_buff structure) and copying (both the sk_buff structure and the packet data).

[3] This means that either the incoming interface is in trunk mode (uses 802.1q tags) and the outgoing interface is in access mode (uses untagged packets) or vice-versa.

VIII. Packet "Post-Processing"

The rules above inevitably led us to a "post-processing" approach. Basically this means that, when you traverse a list of outgoing ports, you must postpone packet transmission to the current port (list element) until you process the next list element. This is so because you can't know what you have to do with the socket buffer until you analyze the next list element.

To better illustrate this concept, let's take a simple example. Suppose the Switching Engine gets an exclusive copy of a socket buffer (which must eventually be freed) and the packet needs to be sent to three ports. Because the copy is exclusive, the sk_buff structure must be copied only twice. If the packet must be sent to N ports, then exactly N - 1 copies need to be made. Before you send the packet to a port you must clone the socket buffer first, but *only if* there are more ports to send it to. You cannot however clone it after you've sent it because it will lead to a race condition.

The sane approach to this problem is the following:

```
while(more ports in list) {
    if(there is a "previous port") {
        clone the socket buffer;
        send the socket buffer to "previous port";
    }
    the current port becomes "previous port";
}
if(there is a "previous port") {
    send the socket buffer to "previous port";
} else {
    discard the socket buffer;
}
```

We call this post-processing because we always send to the "previous port". The key to this approach is that the last port is processed outside the loop, where no clone operation occurs. Thus the number of clones is one less than the number of ports, which is exactly what we needed.

After the loop, we could just send the packet to the "previous port" but the extra check for an existing "previous port" prevents the algorithm from messing things up when there's no port to send to (the port list is empty).

IX. Performance

Because all packet switching is done in software, most performance issues arise with a large number of small-sized packets. In this case, the CPU spends most of its time with packet processing and actual data throughput is very low.

Under normal conditions, the overall performance is better, because packets carry more usable data and thus less packets are switched at the same data throughput.

Our first performance test was run using 64 bytes (headers included) packets. We'll show the test network diagram and the test results.

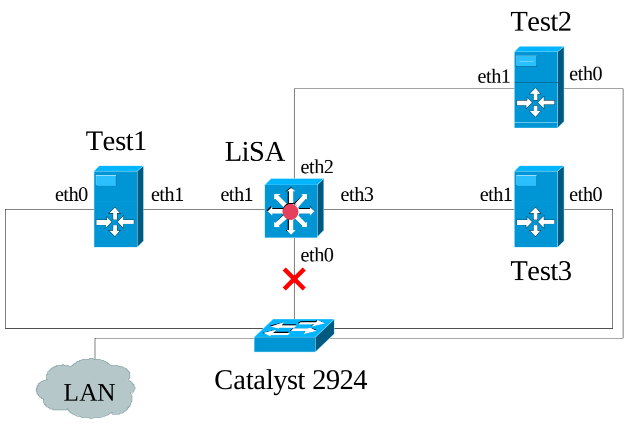

Figure 3. Switched Packet Rate Testing Setup

Test1, *Test2* and *Test3* are all Dual Xeon / 2.8 GHz, with Intel 6300 ESB chipset and two BCM5721 (Broadcom Tigon 3) network adapters. *LiSA* is a Commel Systems LE-564 embedded system (please refer to http://www.commell-sys.com/Product/SBC/LE-564.htm for further details).

On Test1 we ran the Linux kernel embedded packet generator, *pktgen* (please refer to Documentation/networking/pktgen.txt in the 2.6 Linux

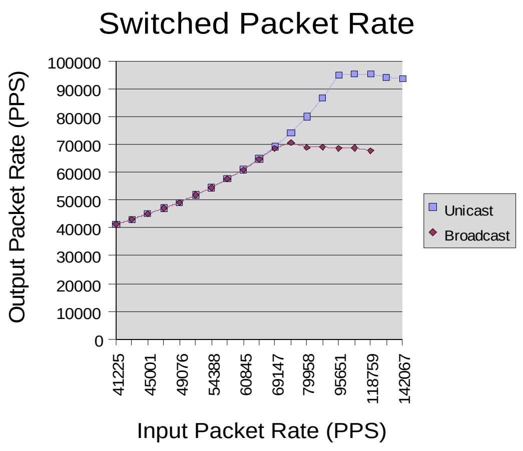

Figure 4. Switched Packet Rate Results

kernel source). On Test2 and Test3 we ran *tcpdump* in logging mode.

We ran several tests with 64-bytes packets, but only two of them are really important: unicast and broadcast, with packet tagging. Both tests were run with 1,000,000 packets samples and packet rates up to 140,000 pps.

For the unicast test, the source port was configured in access mode and the destination port was configured in trunk mode. For the broadcast test, the source port was configured in access mode and the destination ports were configured in both access and trunk mode.

The output rate scales with the input rate up to a certain point, called *cut point*. With higher input rates, the output rate remains constant. This behaviour can be fully explained through the NAPI design theory [5]. Because of the interrupt mitigation technique, the system will process as many packets as it can, no matter what the input packet rate is. As the input packet rate increases, the system will use more and more of its resources. Eventually (when the cut point is reached), it will use all its resources and the output packet rate will remain constant even if the input packet rate increases more.

Obviously, the cut point is reached faster with broadcast packets, because broadcast packets involve more switching overhead than unicast packets.

NAPI interrupt mitigation works by disabling hardware

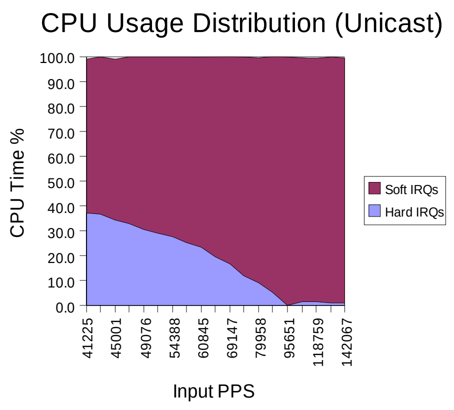

Figure 5. CPU Usage Distribution (Unicast)

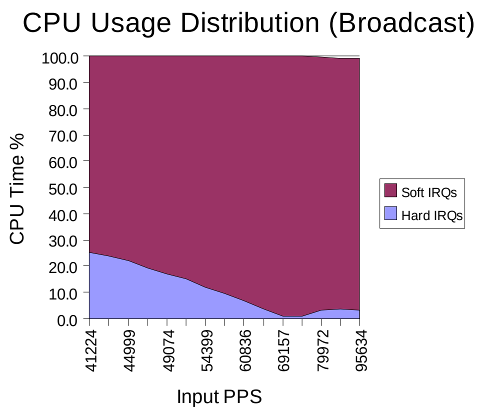

Figure 6. CPU Usage Distribution (Multicast)

interrupts and polling the network chip when the input packet rate is high enough. This way the system is freed from handling hardware interrupts and performs much better. Hardware polling and packet processing is done in software interrupt context. The following charts show the CPU time distribution during the tests.

These tests were run with lab generated traffic. However, this is the worst case (because the switching overhead is the highest) but it is unlikely to happen in real networks (except for packet flood traffic, which is not usual traffic anyway). Therefore, we ran additional tests and we generated traffic that closer resembles real-world networks.

For the bandwidth test, we slightly changed the network topology.

We used the *nc* tool on all machines. It was indirected

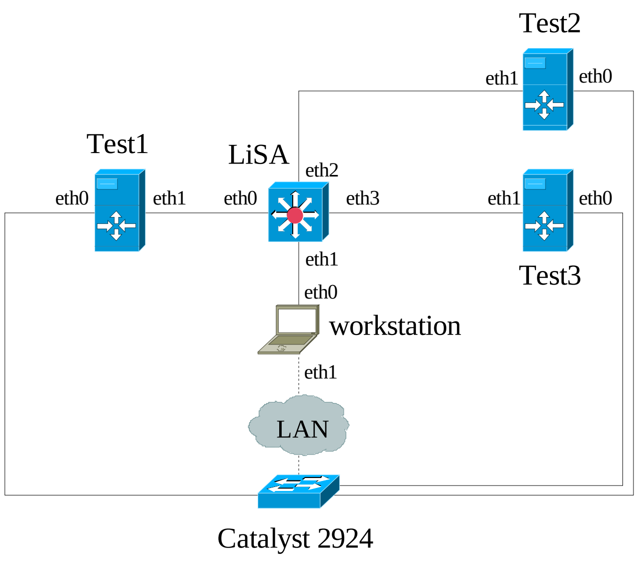

Figure 7. Bandwidth Test Setup

from /dev/zero on server machines and redirected to /dev/null on client machines. The effective bandwidth was measured with the *iptraf* tool. Because the eth0 interface on the LiSA system is gigabit, we made asymmetric transfers: all transfers had the Test1 system at one end.

Two separate tests were run. For the first one we used bidirectional transfers, and the second we used unidirectional transfers, with Test1 as source. The following table summarizes all test results:

	Test1	***Test2***	***Test3***	***Laptop***
Unidir In (Mb/s)	199.59	0.15	1.51	1.41
Unidir Out (Mb/s)	4.43	68.39	68.44	63.24
Unidir Total (Mb/s)	**204.02**	**68.54**	**69.95**	**64.65**
Bidir In (Mb/s)	148.29	20.37	18.58	23.1
Bidir Out (Mb/s)	69.8	50.66	50.73	49.29
Bidir Total (Mb/s)	**218.09**	**71.03**	**69.31**	**72.39**

As you can see, the total transfer speed in both cases is almost the same. This can be explained because the transfer speed is actually limited by the LiSA system's PCI bus bandwidth. Each packet crosses the PCI bus twice (once from the input interface to the central memory and once again from the central memory to the output interface). Thus, the PCI bus speed is double the transfer bandwith, approximately 400 ... 450 MBit/s.

If the PCI bus runs at 33 MHz and transfers 32 bits in each cycle, its total bandwidth is 1056 MBit/s. As one can easily see, this is not what actually happens. There are several reasons why the actual transfer bandwidth is only half the expected (theoretical) bandwidth. All transfers are performed through DMA, and performance penalties are related to the DMA transfers rather than the PCI bus itself:

- The PCI bus has no separate address and command lines, so additional bus cycles are required to transfer the address and the command (DMA transfer, for instance).
- DMA transfers are performed in bus master mode – the network chip must become bus master before the DMA transfer can start. Additional wait cycles are required before a specific device can become bus master.
- The main memory is shared between the CPU and DMA transfers. Additional wait cycles are required when the main memory is not available for DMA.
- When transmitting packets, each packet (or a small group of packets) must be acknowledged by the CPU before another packet can be sent. Thus packets are not transmitted right away: additional CPU work is required and so there may be bus idle times.

References

[1] Klaus Wehrle, Frank Pahlke, Hartmut Ritter, Daniel Muller, Marc Bechler; *"The Linux Networking Architecture"*, Prentice Hall, 2005.

[2] Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman; *"Linux Device Drivers "*, O'Reilly and Associates, 2000.

[3] Octavian Purdila; *"Curs de Siteme de Operare"*.

[4] Alan Cox; *"Network Buffers and Memory Management"*; Linux Journal 29th edition, September 1996.

[5] Alexey Kuznetzov, Robert Olson, Jamal Hadi Salim; *"Beyond Softnet"*, Usenix Paper, 2004.