Lincoln Tunnel L3 Software

Application Note

July 2009

Intel Confidential

Reference Number: 426941-0.5

Contents

Figures

Tables

Revision History

Date	Revision	Description
July 2009	0.5	Initial release

1.0 Overview

Lincoln Tunnel software modifies an existing Ethernet* driver to implement a flow processing pipeline to efficiently classify and process network traffic at very high rates. Intel® architecture is well suited for this type of operation due to its multi-core architecture and large caches, as long as packets are delivered to the processing core efficiently.

The software comprises two parts:

- **Driver Core**
  The driver core contains a modified version of the e1000e driver (version 0.5.20). Modifications were made to e1000.h, netdev.c, param.c, and e1000_82571.c files and two new files were added for flow processing (flows.c and flows.h). Refer to Appendix A, "e1000e-0.5.20 Modifications" for additional information.
- **Flows Setup**
  The flows setup is a user space tool for rules configuration. It parses a policies file and builds a policy table based on the "HiCuts" algorithm. HiCuts is a decision tree-based packet classification algorithm. Refer to the "Packet classification using hierarchical intelligent cuttings" whitepaper[1] for additional information. The policy table is copied to kernel space for policy lookups.

Lincoln Tunnel software has the following characteristics:

- Leverages existing e1000e driver:
  - Reuse of NIC configuration/communication code
  - Reuse of interrupt handling code
- Use of Linux* kernel for interrupts/NAPI and memory management:
- Bypass of Linux kernel for packet processing:
  - Flow lookup/processing
  - Multiple per-CPU, per-interface lockless transmit queues
  - Descriptor bunching

1.1 Motivation

One basic motivation for implementing Lincoln Tunnel software was the excessive time spent waiting for locks. When the RX interface receives a packet, the kernel locks the RX queue on the incoming interface. The kernel also locks the corresponding TX queue on the outgoing interface.

The other basic motivation was the lack of TX descriptor bunching. Without descriptor bunching, the driver updates the TX descriptor ring tail pointer for every packet, which requires a costly PCI configuration space write on each packet.

Lincoln Tunnel software solves these issues by:

- Using separate TX queues
- Bypass the Linux stack (complex "One size fits all" solution)
- Doing RX and TX descriptor bunching

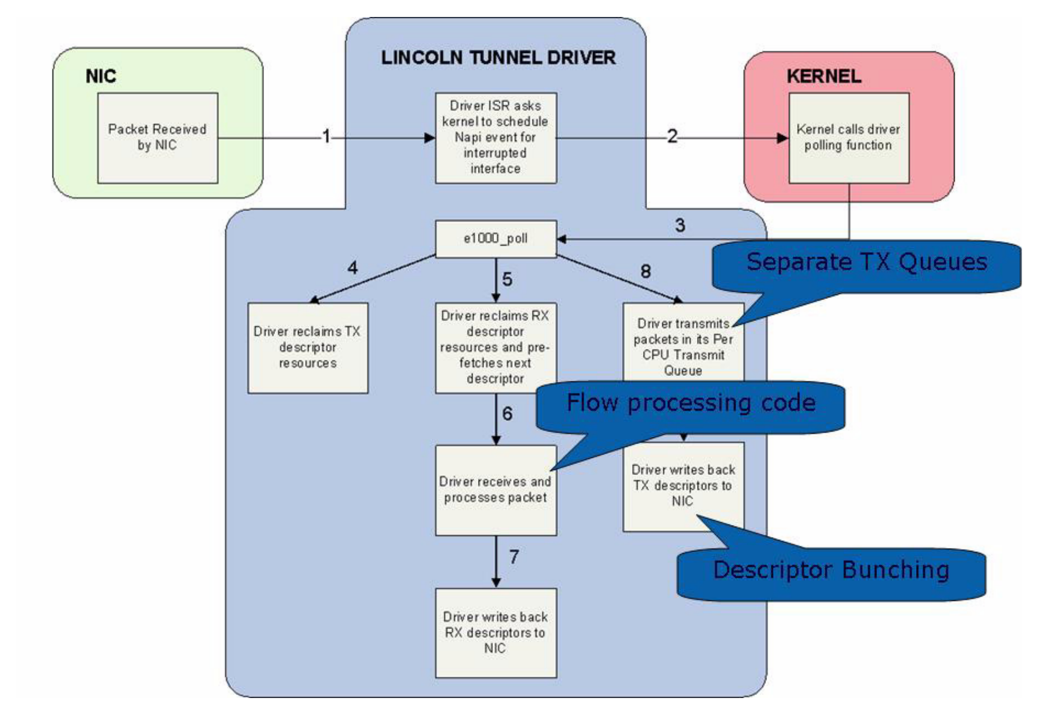
---

[1] HiCuts: P. Gupta and N. McKeown, "Packet classification using hierarchical intelligent cuttings", in Proc. Hot Interconnects, 1999.

1.2 Packet Flow using Lincoln Tunnel

Figure 1 shows the packet flow used by the Lincoln Tunnel software. The packet flow is desribed as a sequence of steps following the figure.

Figure 1. Packet Flow


NIC
Packet Received by NIC
LINCOLN TUNNEL DRIVER
Driver ISR asks kernel to schedule Napi event for interrupted interface
KERNEL
Kernel calls driver polling function
1
2
3
e1000_poll
4
5
8
Separate TX Queues
Driver reclaims TX descriptor resources
Driver reclaims RX descriptor resources and pre-fetches next descriptor
Driver transmits packets in its Per CPU Transmit Queue
Flow processing code
6
Driver receives and processes packet
Driver writes back TX descriptors to NIC
7
Descriptor Bunching
Driver writes back RX descriptors to NIC

NIC

1. A packet is received by the NIC.

Lincoln Tunnel Driver

2. The driver Interrupt Service Routine (ISR) asks the kernel to schedule the NAPI event for an interrupted interface.

Kernel

3. The kernel calls the driver polling function.

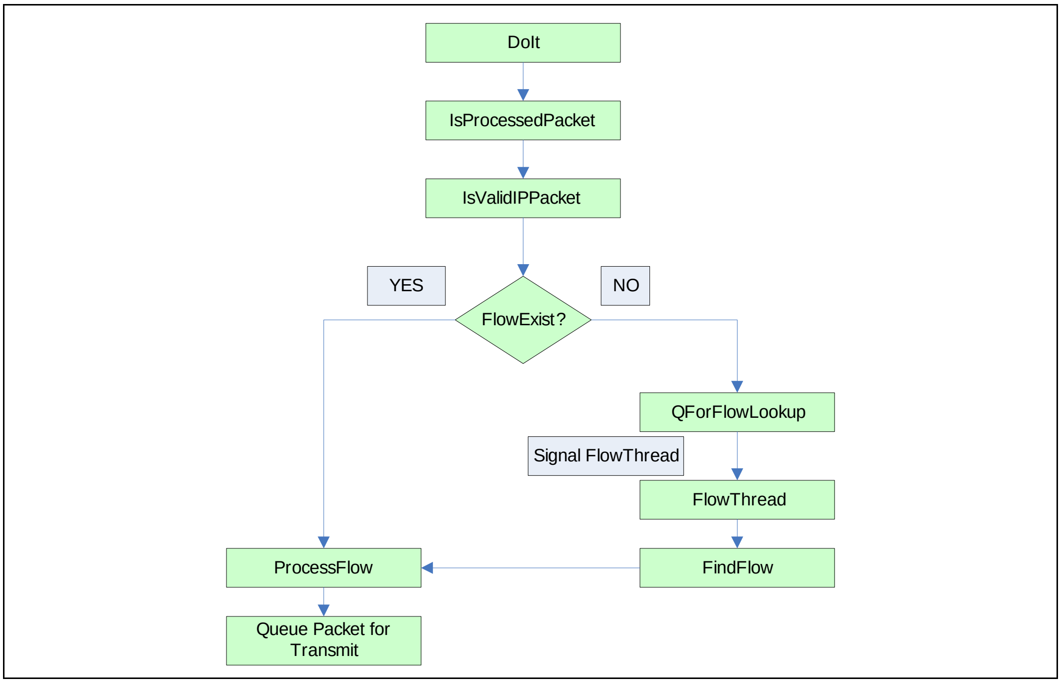
Lincoln Tunnel Driver

4. The driver reclaims the TX descriptor resources.
5. The driver reclaims the RX descriptor resources and pre-fetches the next descriptor.
6. The driver receives and processes the packet. This is the flow processing code that is described in more detail in Section 2.0, “Flow Processing Code” on page 7.
7. The driver writes back RX descriptors to the NIC.
8. The driver transmits packets in its per-CPU transmit queue.
9. The driver writes back TX descriptors to the NIC. Descriptor bunching is used to optimize packet flow. See Section 2.2 for additional information.

2.0 Flow Processing Code

Figure 2 is a flowchart of the flow processing code. The flow processing code is desribed as a sequence of steps following the figure.

Figure 2. Flow Processing Code



1. The DoIt function is called by e1000_receive_skb
2. IsProcessedPacket checks to ensure the packet is a valid IP packet and extracts the 5tupple for hash computation. The 5tupple is based on: SRC/DST IP, SRC/DST Port, and protocol (TCP, UDP and so on).
3. FlowCheck first checks to see if there is a cached flow associated with the RX interface. If a cached flow is found, the flow exists. If no cached flow is found, the hash value is computed and a flow lookup in the flow table is performed. If flow is found in flow table, the flow exists. If it is not found, the flow does not exist in the flow table and it is not cached. If flow is found, proceed to ProcessFlow (Step 6.)

Note: The flow does not exist in the flow table and it is not cached.

4. QForFlowLookup signals a separate kernel thread (FlowThread) to do the FlowLookup.
5. FlowThread calls FindFlow to look up the flow in the policies HiCuts tree. If found, the flow is added to the flows table and the receiving interface's cache. ProcessFlow is called.

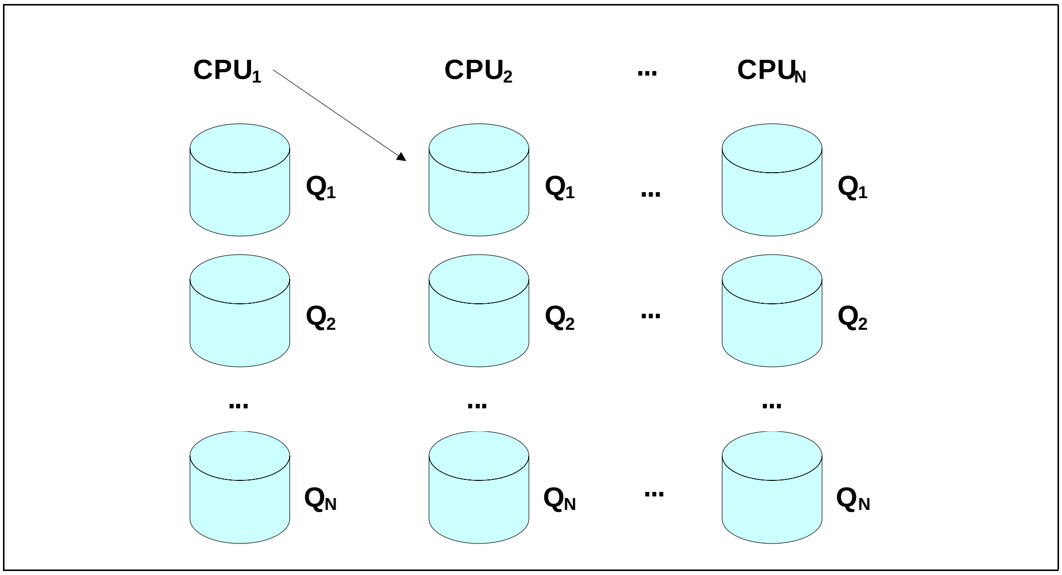
Note: The flow exists in the flow table.

6. ProcessFlow performs actions on the packet based on the rules specified in the policies file. The policies file is created by user and loaded using the user-space tool.
7. ProcessFlow adds the packet to be forwarded to the per-CPU transmit queue. Transmit queues are desribed in Section 2.1.

2.1 Transmit Queues Example

Figure 3. shows an example of a transmit queue.

Figure 3. Transmit Queues Example



Consider the flow from eth1 -> eth2. eth1 was assigned to CPU_1 using IRQ affinity and receives a packet. eth1 determines the output device for eth2 using policies specified by the user. CPU_1 writes the packet to Q_1 belonging to CPU_2.

Meanwhile, CPU_2 is polling all of its TX queues. CPU_2 sees that CPU_1 and transmits the packet.

2.2 Descriptor Bunching

The standard e1000e driver writes TX descriptors back to the NIC after transmitting every individual packet. This requires a costly PCI config space write.

To avoid this, the Lincoln Tunnel software uses a modified version of the driver to wait until many packets have been transmitted before updating the TX descriptor tail pointer on the NIC. (TODO: How many packets? Is this configurable?) This has the advantage of balancing the overhead of many packets and reduces the per-packet load.

Note: The standard e1000e driver performs RX descriptor bunching already, therefore the modification was implemented for TX descriptors only.

3.0 Performance

Do we want to include any performance information?

If so can we use contents from Jeff's presentation?

Do these results need legal review?

Appendix A e1000e-0.5.20 Modifications

A number of changes have been made to the standard e1000e driver to support the Lincoln Tunnel softare. The following is a list of files that were created/modified and a brief description of the changes.

- Files Changed:
  - — e1000_82571.c – Register configuration required for descriptor bunching
  - — e1000.h – Added members to e1000_adapter structure
  - — netdev.c – Added code to use Lincoln Tunnel TX queues, descriptor bunching, and NUMA awareness
  - — param.c - Added code to setup NUMA node assignment
- Files Added:
  - — flows.c – Flow searching/processing/queuing and condition checking
  - — flows.h – Defines structures for in flow processing

§ §