Optimizing Network Performance

Rajanarayana Priyanka Marigi
Andreas Ireståal

Department of Electrical and Information Technology
Lund University
and
Axis Communications AB

Advisor: Mats Cedervall, Lund University
Co-Advisor: Mikael Starvik, Axis Communications AB

January 13, 2012

Printed in Sweden
E-huset, Lund, 2012

Abstract

This thesis investigates and improves hardware and software architectural aspects that directly influence the TCP network transmission in order to reduce CPU utilization. The main areas of investigation have been Linux kernel network implementation handling video data before handing over to device driver, cache hierarchy, DMA transmit rings, and specialized network offloading hardware. The final results of the thesis yielded an overall improvement of 10% in throughput while the CPU usage is reduced by approximately 60%.

Acknowledgement

We would like to take this opportunity to thank our supervisor, *Mikael Starvik* for his guidance and valuable comments. We would also like to thank *Axis Communications AB* for giving us the opportunity to work with this master's thesis at their office.

We would also thank *Dr. Mats Cedervall*, our supervisor at the *Department of Electrical and Information Technology* of *Lund University*, for providing us with valuable suggestions, guidance and constant supervision.

Finally, we would like to specially thank our families and friends for their support and encouragement.

Lund, Sweden
January 20, 2012

Rajanarayana Priyanka Marigi
and Andreas Ireståll

Contents

List of Figures

List of Tables

Chapter 1
Introduction

This master's thesis investigates computer architecture aspects influencing performance of Central Processing Unit (CPU), structure of network protocol processing offload hardware engine to decrease CPU usage, and transmit data handling for optimal performance of TCP/IP network stack in Linux kernel. The optimizations are evaluated for the Axis ARTPEC-4 architecture which is specialized for video transmission over network. This work can be beneficial to those with an interest in architectural impacts on CPU performance and transmission efficiency.

The thesis was carried out at the Department of Electrical and Information Technology at the Faculty of Engineering of Lund University in cooperation with Axis Communications AB, a company located in Lund.

1.1 Background

The TCP/IP protocol suite is the most widely used technology for networking. Such a technology where per-byte and per-packet overhead dominates consumes significant amount of CPU processing time. Transmission Control Protocol, TCP is one of the important connection oriented protocols used to achieve a reliable transmission. In an Axis System on Chip (SoC) video surveillance system data is input from a video camera, encoded to digital data using special hardware, and transmitted to a remote destination over network without data being modified by CPU. In such a system having a hardware accelerator to perform network processing improves performance and reduces CPU usage. Also, it is Axis software's goal to keep the CPU relatively free by identifying bottlenecks in each chip generation and solve them through hardware accelerators. This allows the CPU to handle future workloads that is necessary in the event of enhancing features. The Axis ARTPEC-4 system offloads stateless processing of network stack to a Network Processor (NETPROC) accelerator. However, at times such optimizations to solve a particular issue shifts the area of problem to other aspects, and the full expected improvement won't be achieved. In the ARTPEC-4 SoC system, tests have shown that though performance is good for a single client scenario it drops rapidly for multiple connected clients. Thus, this master's project was proposed to investigate into these issues, provide architectural suggestions and/or improvements to optimize network performance by decreasing CPU usage.

1.2 Problem description

The purpose of the ARTPEC-4 SoC is to transmit video surveillance images to several connected clients. With several clients, processing of TCP/IP packets in software consumes a high amount of the CPU resources. This means that other useful system or application processing like audio compression algorithms don't get enough resources. Such applications also reduce network throughput because the CPU has to perform the same tasks in a repeated orderly manner on data (like copying, segmentation, checksumming) to generate packets to be sent over the network.

In most cases the ARTPEC-4 SoC acts as a server system with traffic mainly flowing in egress direction, and without the packet data being touched by the CPU for any kind of processing. In such scenario CPU consuming tasks can be offloaded to a network processing hardware block so the system instead can use the CPU cycles for computation intensive applications, and meanwhile also achieve better performance. The ARTPEC-4 SoC implements a Network Processor towards this end which offloads the TCP segmentation and protocol checksumming tasks from the main CPU on the transmit path. AXIS software also utilizes the zero-copy concept supported by the Linux kernel in version 2.6 to avoid all copying of data, with the entire image data being handled using pointers. These pointers are given to the Network Processor to be transmitted over the Ethernet interface by the Direct Memory Access (DMA) without the data going through the CPU. Performance tests have shown that this system architecture works well for a single client, but with multiple clients the performance drops significantly.

1.3 Problem analysis

One suspected cause for the surprisingly low performance of Axis's offloaded system with several clients connected is the computer architecture of the system. As the ARTPEC-4 SoC is transmitting video data most of the time, and spends little time on the receiving path which only contains connection requests and acknowledgement traffic, the first logical assumption is to look into architectural aspects on the transmission path. Since the Linux kernel does not support a complete offload of TCP/IP processing, and probably never will since there is strong opposition against implementing it due to maintainability concerns [5] [13], the kernel only supports segmentation and checksum offloading. Even with the zero-copy implemented in the kernel, the network stack can be modified in a way such that the CPU processing of TCP/IP data would be reduced.

1.4 Thesis scope

The main goal of this thesis is to improve the performance in the multiple connected clients scenario of the ARTPEC-4 SoC. In other words this means decreasing the CPU usage. Since many aspects can contribute to the factor of CPU usage, the scope of interest has been limited to investigation and/or improvement in three areas:

- Investigation of possible additions or changes to the hardware architecture and/or the firmware functionality in the Network Processor to decrease CPU usage.
- Investigation of possible computer architectural changes in the ARTPEC-4 SoC that would influence the CPU usage. The main components influencing the data in memory on transmit path are the CPU architecture, the cache hierarchy and related policies, the DMA unit, the NETPROC, and the Ethernet interface. The custom hardware accelerator, called video subsystem in ARTPEC-4 reads the data from the video inputs, compresses the data, and then writes the compressed image data back to memory. As such the investigation is for the data path from memory to Ethernet interface.
- Investigation of different approaches to transmit video surveillance data more efficiently from the memory to the NETPROC by minimizing the amount of CPU support being required in the process.

The main protocol of investigation is TCP as this is the main protocol used by the clients to receive surveillance video data.

1.5 Thesis outline

The remaining chapters in this thesis are organised as follows:

Chapter 2 presents basic theory for the thesis. This includes TCP/IP protocol suite and basic computer architecture of ARTPEC-4.

Chapter 3 describes the tools used throughout the thesis work.

Chapter 4 presents main investigations and implementations carried out in order to achieve the goal of the thesis. It also gives the account of all measurements made and results obtained.

Chapter 5 presents the final results, discusses several considerations, and options for future implementation.

Chapter 2
Theory

This chapter describes the background theory required for basic understanding of the thesis, decisions made, and work carried out. It describes the basic TCP/IP network stack protocols and ARTPEC-4 architecture. It is recommended to read the referenced articles in the bibliography section on the topic if the reader finds the topic interesting and wants to learn more about it.

2.1 The OSI Model and the networking protocols

The Open Systems Interconnection model, or OSI model, is a standardization of communication systems. It defines 7 different layers as shown in Figure 2.1, but does not specify any specific standard protocols for those layers to use. This model is widely accepted, as the protocols tends to vary over time with different vendors demanding different features and short-comings being discovered over time. It exists, however, a de facto standard in the Internet infrastructure of using Ethernet in the data link layer, IPv4 (and lately IPv6) in the network layer, and TCP or UDP in the transport layer. By simplifying, it could be said that each layer only contains necessary information for processing on it's own level. The data output of one layer is used as data input for the immediate adjacent layer. For instance, an IP packet (level 3) contains IP relevant information (source and destination addresses, Time-To-Live, etc) and carries a layer 4 packet (TCP, for example) in it's payload. This hierarchy creates a lot of traffic overhead in terms of header data, but does also provide durability of communitations infrastructure. The three lower layers are the media layers, and defines ways of transmitting data between hosts. The upper 4 layers are the host layers used in the hosts upon data arrival. Therefore, large networks could be created by using routers or switches only implementing the bottom three layers. But for useful host-to-host communication it is required that both end nodes implements layers 4-7.

2.1.1 Ethernet standard

Ethernet is a family of hardware standards for network communication where the standards differ in speeds, actual hardware used, and other features. The very first standardized version of Ethernet was released by the Institute of Electrical and Electronics Engineers (IEEE) in 1983 with the IEEE 802.3 standard. It's

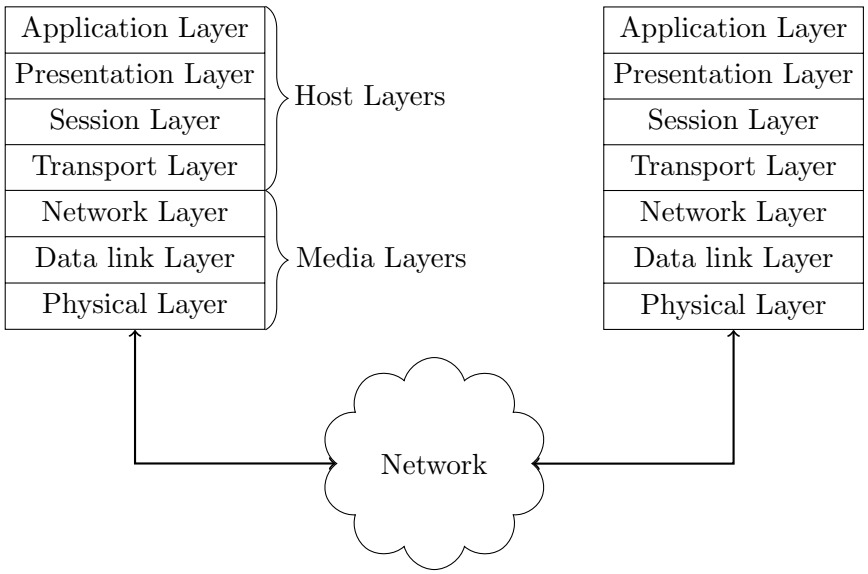


Figure 2.1: The OSI model illustrated

specifications were to use the speed of 10 Mbit/s using thick coax cables. Later standards have specified speeds up to 100 Gbit/s, and several different hardware links like optical fibre and RJ-45 connectors over copper. The most commonly used speed for consumer wire connected devices today is the Gigabit Ethernet running at 1000 Mbit/s or 1 Gbit/s superseeding the somewhat older but still quite popular Fast Ethernet Standard running at 100 Mbit/s.

One Ethernet standard was made in 2003 which added the ability to power devices using only an Ethernet cable. This standard, known as Power over Ethernet (PoE), has become very popular in Axis products since it can save customers the trouble of installing additional power cords for the cameras. The first version of this standard specifies a maximum power supply of 15.4W using a 44-57V power source which is enough for most of the Axis products.

Even with a lot of different Ethernet standards and additional features being evolved in the last 30 years the structure of the Ethernet frame[1] has seen very small or no modifications up to date, and is fully described in Section 2.2.

2.2 Common Network Protocols

To achieve any form of usable communication between two connected devices a protocol has to be agreed upon. A protocol defines how information should be encoded and interpreted. Hence, a poorly developed implementation of a protocol could render a device useless. The best way to describe a protocol is to describe it as a language in electronic form. In order for two devices to understand each other

[1] A data packet consisting of protocol header and payload sent over the Ethernet Line.

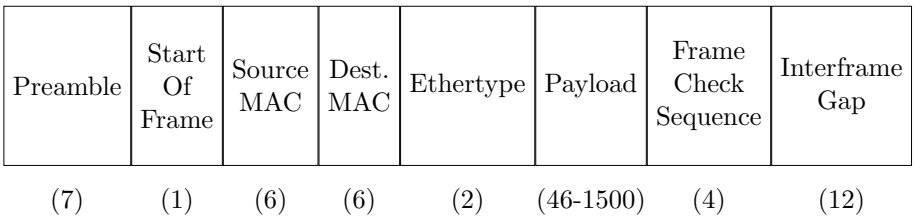


Figure 2.2: An Ethernet frame with different fields and their sizes in bytes

they have to speak the same language. The following sections will discover the most widespread network protocols used currently in almost all network infrastructure and server systems today.

Ethernet Frames

The 802.3 Ethernet frame, shown in Figure 2.2, defines the way data is sent over the Ethernet interface. The MAC addresses used for source and destination fields are unique for each Ethernet device. These addresses are used to make sure the transmitted data will be delivered to the device with the corresponding address. The Ethertype field tells the receiving Ethernet interface which protocol it can find embedded in the payload. Since the Ethernet frame standard is at the second level of the OSI model it is used for carrying data and forwarding it to the upper layers. The Ethernet frame also contains a checksum field to confirm data integrity. Preambles are used to detect new incoming packets and locate the start of packets. Interframe gaps also have a somewhat similar purpose by leaving room for the receiving hardware to process it's data. Since the preamble, interframe gap, and Ethernet header use some of the available capacity of the Ethernet channel the amount of useful available data traffic on a Fast Ethernet channel is at best

$$\frac{1500\text{ B}}{1538\text{ B}} \cdot 100\text{ Mbit/s} = 97.529\text{ Mbit/s}$$

when only MTU[2] packets are sent. At the worst possible case, the utilization can be at most

$$\frac{46\text{ B}}{84\text{ B}} \cdot 100\text{ Mbit/s} = 54.762\text{ Mbit/s}$$

Internet Protocol

Internet Protocol (IP) [21] exists in several different versions, with version 4 being widely used. It has gained a lot of attention lately in various computer magazines and other media due to it's limited 32-bit address space which could cause troubles

[2]Maximum Transmission Unit. Largest data unit possible in payload. 1500 Bytes (B) for Ethernet.

bit offset	0-3	4-7	8-13	14-15	16-18	19-31
0	Version	Header Length	Differentiated Services Code Point	Explicit Congestion Notification	Total Length	
32	Identification				Flags	Fragment Offset
64	Time To Live		Protocol		Header Checksum	
96	Source IP Address					
128	Destination IP Address					
160	Options (if Header Length > 5)					
160 or 192+	Data					

Figure 2.3: The IP header for protocol version 4

in a near future. A lot of effort is being put into encouraging Internet Service Providers to upgrade their infrastructure to support IPv6.

The IP protocol header for version 4 is shown in Figure 2.3 and contains many different fields, but the interesting ones are the source and destination addresses as well as the protocol field. The addresses are, as mentioned earlier, 32-bit values and are used to define unique host numbers. The IP packets are used for inter-network communication, and carries protocol data as specified by the protocol field. The payload consist of a packet from/for the transport layer protocol usually TCP or UDP.

Internet Control Message Protocol

The Internet Control Message Protocol (ICMP) [20] is a protocol used for diagnostic and routing purposes. One of the most common utilizations of the ICMP protocol is the ability to determine whether a system is online or not by sending an ICMP 'Echo Request', also known as a 'Ping' message. The receiver of this message replies by sending an ICMP 'Echo Reply' message with the identical payload of the received 'Echo Request' message. The ability to add a certain amount of payload data could be very useful for determining transmission errors, and also as seen in Section 3.3 for debugging purposes.

Transmission Control Protocol

Operating at the transport layer, the Transmission Control Protocol (TCP) [22] is used for end-to-end communication of applications. TCP is a connection based protocol, and uses several different states to set up and tear down connections, and reliably communicate data. The structure of the TCP header is illustrated in Figure 2.4 and important fields are discussed below.

The flags are used for TCP state handling and will be described below. The ones used in all basic TCP implementations and the ones accounted for in Fig-

<table>
<tr><td colspan="3">Source Port(16 bits)</td><td>Destination Port(16)</td></tr>
<tr><td colspan="4">Sequence Number(32)</td></tr>
<tr><td colspan="4">Acknowledgement Number(32)</td></tr>
<tr><td>Data Offset (4)</td><td>Reserved (6)</td><td>Flags (6)</td><td>Window size(16)</td></tr>
<tr><td colspan="3">Checksum(16)</td><td>Urgent(16)</td></tr>
<tr><td colspan="4">Options and Padding</td></tr>
<tr><td colspan="4">Data</td></tr>
</table>

Figure 2.4: The TCP header and it's fields

ure 2.4 are Acknowledge(ACK) flag, Synchronize(SYN), Finish(FIN), Reset(RST), Push(PSH), and Urgent(URG).

The source and destination ports are used to determine which application running on the host or client system the packet originates or should be delivered to. For server applications there exists some standard port numbers like web-servers running on port 80 and FTP servers running at port 21.

The sequence and acknowledgement numbers are used to ensure ordered delivery. TCP data is seen as a stream of bytes where every byte is numbered. The sequence number field tells the number of the first byte in the payload. By using this approach packets received out of order could be re-organized at the receiver side, and hence ordered delivery can be achieved. The acknowledgement number is used to acknowledge data up to but not including the acknowledgement number. To enable this the ACK flag in the flags field has to be set, and this is usually the case since it doesn't cost anything extra in terms of traffic overhead. The mechanism of sequence and acknowledgement numbers therefore helps the detection of corrupted or lost packets at the sender end. By reading the acknowledgement numbers and comparing them to what has been sent the lost and corrupted segments could be retransmitted after a timeout has occurred. This timeout is basically a timer waiting for acknowledgement of the last sent data segment.

The window size field is used to tell a sender how much data it can send unacknowledged. Furthermore checksumming is implemented to ensure data integrity.

Connection setup and tear down

The flags are used for maintaining TCP state data and for connection setup and tear down. TCP uses what is called a '3-way handshake' as illustrated in Figure 2.5. To establish a TCP connection a SYN flag is first sent along with a randomly generated sequence number. The node at the other end replies by sending a SYN message along with another randomly generated sequence number. The ACK flag is also set and the acknowledgement number is set to that of the initial received SYN packet incremented by one. Finally an ACK with the response sequence number incremented by one is sent by the initiator and the connection is considered established. It could now be used for streaming data in any direction.

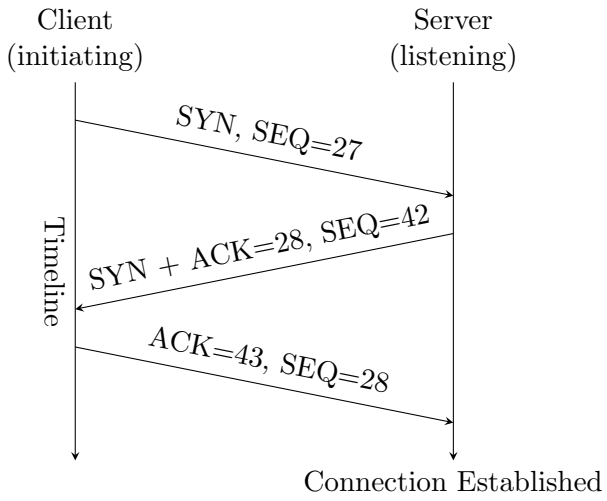


Figure 2.5: The typical procedure of a TCP connection setup

The tear down of a TCP connection is very similar to the procedure of a TCP connection setup since it also involves the three way handshake. See Figure 2.6. A TCP packet with the FIN flag is initially sent by the host requesting tear down. The other end sends a packet with the ACK flag set, acnowledging the tear down request. It then sends a packet with FIN and waits for the acknowledgement of the initiating node. After this procedure the connection is closed and all used system resources by the connection are freed.

User Datagram Protocol

User Datagram Protocol (UDP) [19] is the other common transport layer protocol that has seen a widespread use. It is much simpler than the TCP protocol, and neither support in order data delivery nor retransmission of lost data. It has an advantage though when it comes to response times and processing overhead. This makes UDP ideal for cases where fast response times and low server processing overheads are required. Video streaming and online gaming are two examples where UDP makes a strong case, and in such scenarios a lost segment is more tolerable than delayed packets.

Even some of the upper layer protocols requiring ordered delivery sometimes uses UDP. The network file system, NFS, is one such example and the responsibility of implementing features of reliable connection are instead moved up to the application layer as speed is prioritized. The header is shown in Figure 2.7 and as can be seen is simpler than that of TCP. As can be seen the port numbers are used as well, with a 16-bit port range.

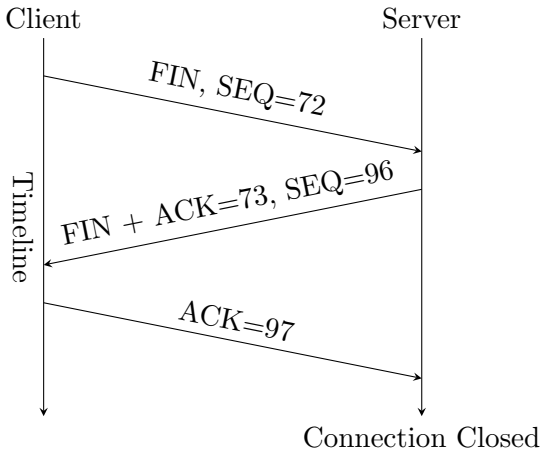


Figure 2.6: The typical procedure of a TCP connection teardown

offset(bits)	0 — 15	16 — 31
0	Source Port Number	Destination Port Number
32	Length	Checksum
64+	Data	

Figure 2.7: The UDP header and it's fields

Real Time Streaming Protocol

The Real Time Streaming Protocol (RTSP) [23] is an application-level protocol that is widely used for media streaming. This protocol can use UDP or TCP for transport layer. The video server software run on Axis cameras use this protocol to stream the video data to a client.

Hyper Text Transfer Protocol

The Hyper Text Transfer Protocol (HTTP) [10] was invented 1990 by Tim Berners-Lee and started the WWW revolution. It does not specify a specific header with fixed length fields, but rather specifies a series of different commands to be sent over a TCP stream for interacting with an HTTP server. The typical use case is a user asking for a specific resource on the server. The server then replies, if the user is authorized to get the information, by acquiring the requested resource on the system, and then transferring it to the client.

2.3 Linux TCP/IP protocol handling of video streaming

AXIS software uses the TCP/IP network stack with the TCP transport layer protocol to transmit video data. The TCP header is extended with a 10 Bytes timestamp option to calculate better Round Trip Times (RTT) for transmitted packets. This results in a total size of 32 Bytes for the TCP header and 20 Bytes for the IPv4 header adding up to a total of 52 Bytes of header overhead.

When the media server has data to send it makes a system call to the kernel. Depending on the system call and available hardware support the kernel performs necessary processing, and makes the data available to the Ethernet device driver to be sent over network. A high level account of the data processing from the socket later to device processing in Linux kernel is given in [9]. [17] and [18] also give a good explanation on socket buffers and TCP handling of socket data respectively.

2.4 Commercial TOE solutions

There exists several different solutions on the market today for network offloading. The most commonly used and widespread is the Transport Segmentation Offloading (TSO) solution where dedicated hardware is used for checksum calculation and segmentation. The solutions that involves hardware handling of TCP parameters are called TCP Offload Engines (TOE) and are used in high-end server cards.

Through the years various TOE solutions have existed. Recently, such solutions have been popular in 10 Gbit Ethernet interfaces where the high traffic requires lots of CPU power. One major contributor in this market is Chelsio Communications which has implemented TOE in their high-end network cards targeted for server users.

Major server operating systems supporting TOE devices are Windows server 2003 and onwards, as well as FreeBSD. As mentioned earlier, there is strong opposition against complete TOE solution amongst Linux Kernel developers, and such a feature would not see implementation in a foreseeable future. Chelsio has, however, published a kernel interface for TOE as well as a few patches. These patches were rejected, so instead their customers have to build their own patched kernel with support from Chelsio in order to enable TOE. Similar solutions exists from other vendors as well, with all requiring third party patches of some kind.

TOE is extremely rare amongst consumer network interfaces since modern computers could handle the traffic generated from Fast Ethernet or Gigabit Ethernet interfaces without any problems. As a rule of thumb, 1 bit/s of TCP traffic require 1 Hz of CPU frequency, and amongst consumer Ethernet cards the benefit of TSO reduces enough system resource usage for a Gigabit or Fast Ethernet interface so TOE is considered unnecessary.

2.5 ARTPEC-4 architecture

ARTPEC-4 is the latest generation SoC in Axis ARTPEC SoC series. It is a system optimized for high performance image processing required for a network camera or video server on a single chip. The SoC supports four video input ports, one video output port, a powerful 400 MHz little endian MIPS 34Kc CPU, 128 KB of internal RAM, 16 KB of internal ROM, DDR2 DRAM and NAND flash PROM, DMA unit, two level cache hierarchy, and 10/100/1000 Mbps full duplex Ethernet MAC with support for TCP/IP stack offloading using a hardware accelerator called Network Processor. Figure 2.8 shows a simplified block diagram of the ARTPEC-4 architecture where only the hardware blocks of interest for this master's thesis are shown.

2.5.1 Cache Hierarchy

ARTPEC-4 implements a two level hierarchy consisting of the caches L1 and L2. An internal RAM memory block of 128 KB can be configured as plain memory, L2 cache or both. A cache is a fast memory used to hold frequently accessed data. This reduces the effective access time to data in main memory, and reduces stalls caused by waiting for data to become available.

L1 cache is an immediate fast memory next to the registers in the main CPU, called CPU cache sometimes. CPU first checks whether a copy of data intended to be written or read is in the L1 cache. If so, the CPU performs the operation on the cached data which is much faster than accessing the L2 cache or main memory. There is a 32 KB of instruction cache and 32 KB of data cache between CPU and L2 cache.

The L2 cache is a system level cache that caches data from main memory for the CPU and the DMA clients. The internal RAM stores both cached data and tag data (124 KB for data and 4 KB for tag data). Presently, the internal RAM is completely configured as L2 cache and does not act as on-chip RAM. The L2 cache is 8-way set associative, uses 128 Bytes cache lines, and write-back write

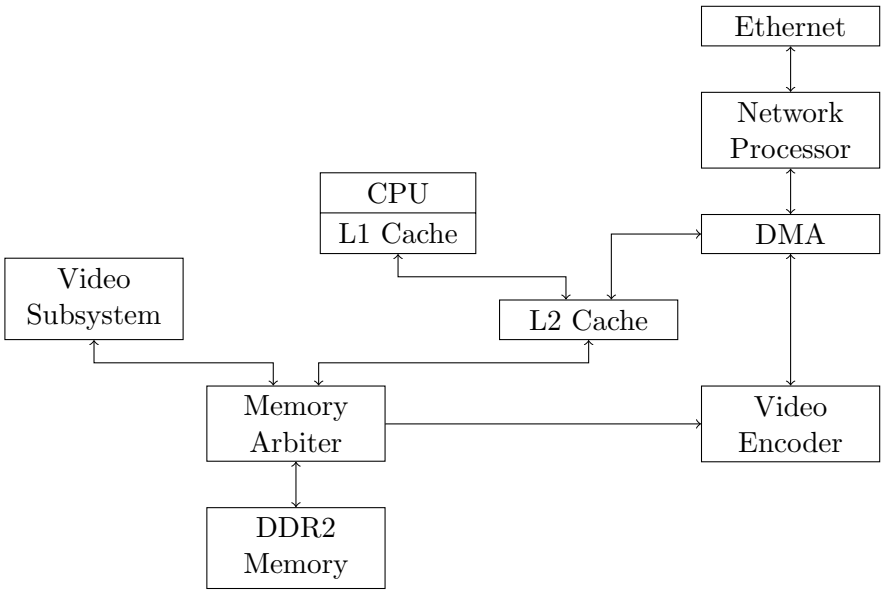


Figure 2.8: ARTPEC-4 architecture with blocks of interest

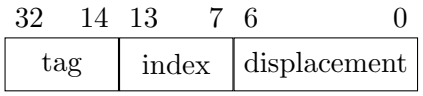


Figure 2.9: Effective memory address split to access cache line of interest

strategy. This means a cache line consists of 128 Bytes of continuous block of data, a set consists of 8 such cache lines. A cache way is all the cache lines at the same position within each set. Write-back strategy means that only the cache line is modified and not written to main memory. Modified cache lines are written to main memory only when they get replaced. The L2 cache uses only physical addresses which means indexes and tags both uses physical addresses, and not virtual addresses. The effective physical address can be divided into three fields as shown in Figure 2.9.

The displacement field selects a byte or word that is requested. The index field selects a set that the data has been put in. The tag field is compared with the tag field of the tag data to verify for a particular physical address.

The L2 cache is non coherent with the L1 cache and other subsystems accessing main memory directly. For instance the video subsystem writes uncompressed video data directly to the main memory without going through L2 cache. This data then goes to an encoder without passing L2. The encoder stores the compressed image data to main memory via L2 using DMA. After processing, compressed

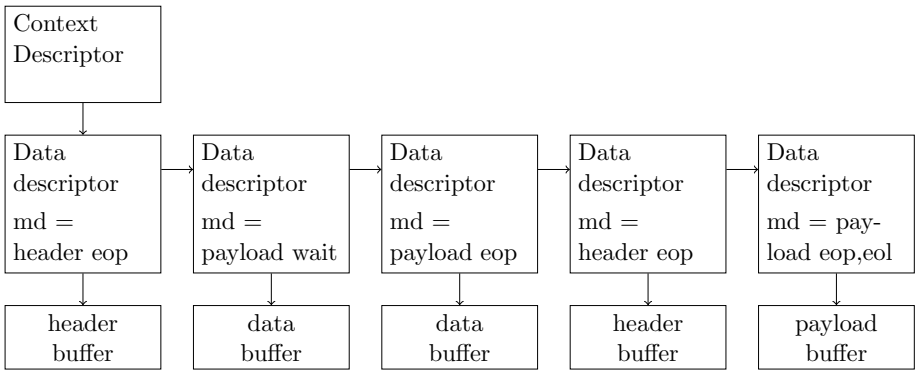


Figure 2.10: Example of two packets in DMA transmit ring

data is sent out on the Ethernet interface using DMA. Because of the absence of automatic cache coherency, relevant parts of the L1 cache needs to be manually flushed to maintain coherency with the L2. The interface between the L2 cache and memory is 256 Bytes wide with a capacity of 1.6 GB/s.

2.5.2 Direct Memory Access

Direct Memory Access (DMA) unit is used to transfer data between a given memory location and any peripheral attached to DMA as client, or between any two memory locations without involving CPU except for initiation of the transfer in some cases. This unit offloads CPU of routine and inefficient task of copying data, and precious CPU cycles can be used to do more computation intensive tasks increasing its efficiency. DMA interface to L2 cache is 32 Bytes wide. Ethernet client attached to DMA via Network Processor is of particular interest in this thesis.

All the control data required to complete transactions is stored in data structures called descriptors. Each descriptor pointing to a buffer in memory is called a data descriptor. A linked list of these data descriptors is managed by a context descriptor data structure. All descriptors have a meta data field for communication between the DMA client and the software. The software interface uses a set of C structs to represent the DMA list descriptors as shown in Figure 2.10.

If several data descriptors are used to describe one packet, the wait flag is set in all but the last data descriptor. The last data descriptor should flag an end of packet bit (EOP) indicating the end of a logical block of data. A list of data descriptors may contain several packets and the last data descriptor of the list is always marked with end of list flag (EOL).

2.5.3 Main MIPS CPU

The main CPU is multi-threaded which behaves as two virtual processors. What this means is that it looks like two CPUs from the view point of software, but these virtual CPUs share the same execution hardware with two different sets of

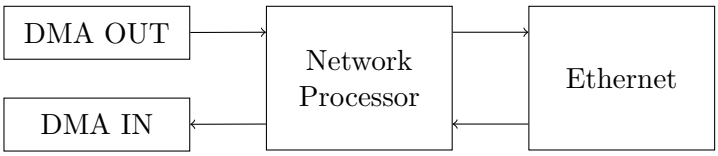


Figure 2.11: Network Processor's architectural position

program registers. The effective performance is about 120% that of single threaded CPU. In the events of stalls from memory reads or pipeline dependencies the other thread runs instead of the first thread waiting. The interrupts, for instance the receive packet interrupt, goes to the primary virtual CPU termed CPU0.

2.5.4 Ethernet Interface

The Ethernet interface is connected to two DMA channels, one in transmit direction and the other in receive direction. When receiving and transmitting packets the Ethernet MAC layer keeps track that packets are communicated correctly. Before the Ethernet interface is ready to transmit or receive data over a network it should be configured with a MAC address, half or full duplex, etc. It should also setup and activate a context descriptor with linked DMA data descriptors for each of the DMA channels.

When transmitting, the interface generates a preamble and start of frame delimiter (SOF) and sends it over the wire. After that it reads data from the memory via the DMA out channel until the end of packet (EOP) flag is reached. The data is read starting at the destination and source MAC addresses followed by the type field, and continuing with the payload part. When EOP is reached the interface automatically appends the frame's Cyclic Redundancy Check (CRC) resulting in the Ethernet frame as seen in Figure 2.2.

On the receive path, the interface starts by sensing the preamble and SOF which are used to lock on an incoming bit stream. Frames passing initial checks on the header field are accepted reading the payload part. In parallel to receiving a frame the interface calculates the CRC and compares with the CRC field in the frame. In the case of a match (should always be the case except for transmission errors and faulty hardware), the frame along with CRC is written to memory via DMA. If not, DMA is commanded to discard the packet and clean up.

2.5.5 Network Processor

Network Processor (NETPROC) is a hardware accelerator that offloads the main CPU from network processing tasks. This increases the network bandwidth and decreases the CPU usage. NETPROC is placed between Ethernet interface and DMA IN and OUT channels as shown in Figure 2.11.

NETPROC uses DMA lists in automatic mode where there is no need to issue DMA commands for every packet DMA handles. NETPROC has three main parts:

- Transmit path: reads packet data from memory via DMA OUT channel and sends the processed packet frame to Ethernet interface.
- Receive path: reads packet from Ethernet interface and stores the processed data in memory via DMA IN channel.
- Network CPU: handles the necessary processing of the packet like segmentation for TCP protocol.

The transmit path acts as a network interface that can receive large chunks of data, segment it into packet frames of underlying network MTU, generate header and checksum for each frame, and send the resulting frame to the destination with very little or no assistance from the network stack of the kernel. The transmit path is also capable of sending same data to multiple destinations and optionally, at a specific rate.

The NETPROC transmit path has two meta data memories each of 256 Bytes and two payload memories of 1.5 KB each. Packet buffers on the transmit path could contain either header or payload data. The meta data field of the DMA data descriptor is used to distinguish between the two. Typically, header data is received from the DMA OUT channel and stored in one of the free meta data memories. The NETPROC CPU process this header information and generates the header for the first frame of the large segment to be sent. In parallel NETPROC continues to read payload data and store it in the corresponding payload memory. While the payload is being stored checksum is generated by the checksumming hardware unit. When one MTU sized frame or end of segment is reached, the header and payload data is transferred from the meta data and payload memory respectively to Ethernet interface. While this packet frame is being transmitted NETPROC receives another packet data via DMA OUT to the second meta data and payload memory. Thus, the transmit path handles two packets simultaneously. A simple block diagram of the transmit path in the NETPROC can be seen in Figure 2.12. The copy unit copies header data for packet n+1 to the other meta data memory while NETPROC CPU generates header for packet n, if the next packet ready for transmission belongs to the same segment. This is valid as major part of the header remains constant for all packets of a segment and changing fields are generated by NETPROC CPU. The control packets like ARP, ICMP, TCP SYN, etc are, on the contrary, transparent to the NETPROC. All these operations are automatically carried out by NETPROC. If required, the NETPROC CPU can also be operated in manual mode.

The receive path has a 512 Bytes memory to act as FIFO buffer for the data received from Ethernet interface before being stored into memory via the DMA IN channel. A block diagram of the receive path is shown in Figure 2.13. The pattern matching unit can be programmed to accept or discard a packet by comparing the header fields to desired values, but presently this unit is not used in the current system configuration. The checksum unit calculates the folded 16-bit checksum in little endian while the data is being sent to memory by DMA. The checksum is written to the meta data field of the data descriptor to be used by the kernel. Again the receive path is operated in automatic mode but can be set to manual control.

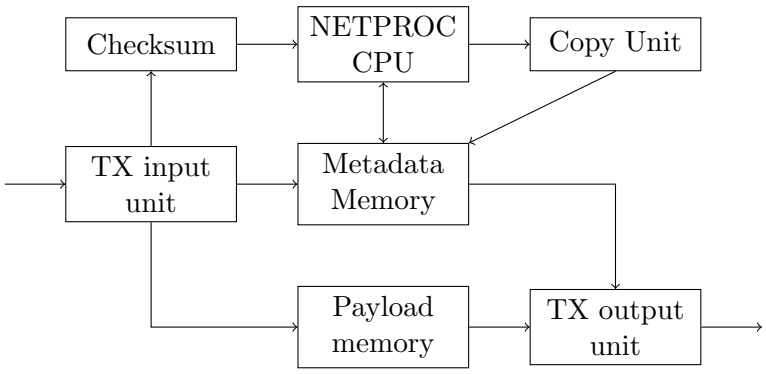


Figure 2.12: Overview of transmit path of the NETPROC

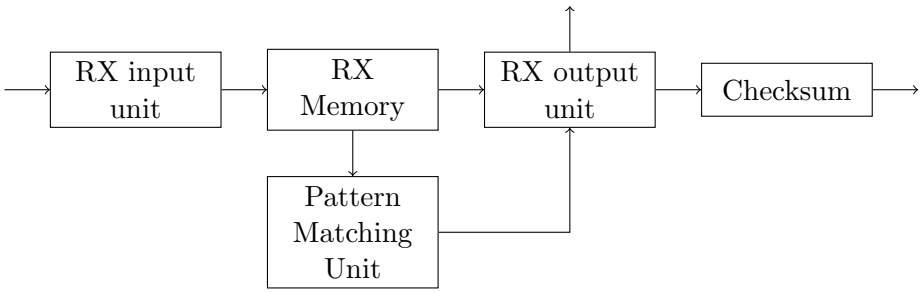


Figure 2.13: Overview of receive path of the NETPROC

The NETPROC CPU is a CRIS[3]v10 CPU with no user mode and multiplication. It has an instruction and data memory of 4 KB. NETPROC has a programmable timer that can be used to schedule events for NETPROC CPU, measure performance etc.

[3]A Computer architecture developed by Axis.

Chapter 3
Tools

Multiple tools, both hardware and software, have been used in order to accomplish the goals of the thesis. The most important ones required to understand the workflow will be described in this chapter.

3.1 P7214 Video Encoder and Test Setup

The test setup used throughout the master's project consisted of:

- P7214 Video Encoder
- DVD Player
- Network switch with PoE
- PCs for connecting to P7214 Video Encoder

The P7214 Video encoder is a convenient board for owners of analogue surveillance equipment to enhance their current system with the Axis network functionality. It can manage up to 4 independent video sources, and has the full network functionality of Axis products by using the ARTPEC-4 SoC platform.

To reduce the amount of cables needed to connect the P7124 Video Encoder, the system is powered by the Ethernet interface itself. Furthermore a DVD player generating a 25 Frames Per Second (FPS) PAL[1] video signal is used as the video input source. A video showing a train arriving at a train station following a large crowd leaving the train is used as the test motion picture. This is ideal due to it's richness of colour and motion, since it will put heavy load on hardware resources inside the ARTPEC-4 SoC. The test setup is shown in Figure 3.1.

3.2 Profiling

In order to know where the software bottlenecks are located, profiling software needs to be used. A profiling software samples the program counter of the CPU and looks up the address in a symbol table to get the symbol currently running. Every symbol has a counter for how many hits it had so far in the profiling. When

[1] European video signal standard

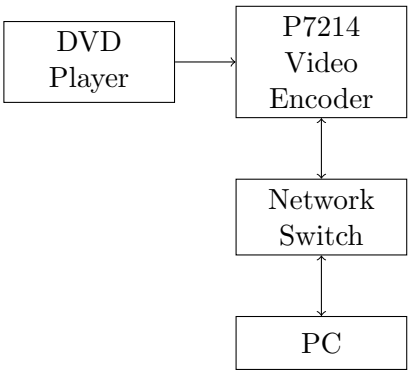


Figure 3.1: Test Setup

the profiling is done a table of results is printed. This table shows how many hits each label had (usually it shows only the 20 most used or so) in both actual numbers and as percentage out of total samples.

3.2.1 AXIS profiling software

The preferred way of profiling an Axis system is to do it over network. To achieve this Mikael Starvik[2] at Axis has developed a software for profiling over network in an automagic manner. To ensure proper execution Telnet has to be enabled on the profiled system. The profiled system will profile itself and send the results back to the system requested the profiling data over FTP. This reduces the network overhead involved as the program counter requests are made internally from the profiled system.

The user could at any time after setup press enter on his keyboard to print the profiling results collected until that time. This method of profiling has been the most commonly used method of profiling throughout the course of the thesis.

3.2.2 FS^2 probing hardware

For some measurements the absense of interrupt-disabled execution paths could lead to inaccurate results. If the user wants to include all possible code paths, and also achieve zero overhead profiling another way of measuring has to be used. This is possible by using the FS^2 hardware probing system.

By connecting to the target's system with a Joint Test Action Group (JTAG) cable and reading the Program Counter, the profiled system gets no overhead or interrupts, and do not know about the profiling at all. The results achieved this way are accurate, but since this way of profiling involves lot of extra work in terms of buggy software, extra cables, and sensitive hardware, it has been of limited use.

[2]Also, one of the supervisors of this thesis

3.3 NETPROC debugging

Since the NETPROC used in ARTPEC-4 is very limited, a set of tools had to be used separately for debugging the software run on the NETPROC CPU. There is a possibility of adding printing libraries to the NETPROC firmware itself. But this would add an extra 1.5 KB to the firmware size which is unacceptable when the memory is 4 KB, and advanced functionality requiring memory area for code are implemented.

To still be able to read parameters from the NETPROC itself a custom solution was designed. By using the Wireshark software for packet analysis and modifying the payload of ICMP replies, any variable or parameter could be printed and read from Wireshark. This way, debugging code was reduced to a few 100 Bytes, but the modification also introduced a lot of extra work.

Chapter 4
Implementation

Efficient data transmission from the source (video input) to the destination (network output) is affected by many factors such as address space of data, actual communication of data between different abstraction layers of the kernel and the network protocol layers, DMA support, special hardware engines like video encoder, cache policies, receive path offload, and so on. This chapter will explore different but not limited to, architectural (both software and hardware) factors impacting the data flow. It also discusses how this affects the kernel load and the resulting network throughput.

As the design space of a SoC architecture is huge the investigation is limited to the set of hardware components involved in the network transmit path. For ARTPEC-4 these components are the cache, the DMA transmit ring, and the network processor. The Linux kernel itself supports transmission of large data segments with minimal overhead. The implementation and working of this software functionality are explored as well. The resultant potential improvements are discussed in each track of investigation.

4.1 Data transmission mechanisms over network

Linux kernel has several I/O-functions which could be used over a network, over a file system or between any file descriptors since anything could be a file in a Unix system. A natural consequence of these I/O-functions are that the implementation is done in a very generic manner to increase maintainability. By not considering requirement specific aspects performance may not reach its full potential.

Initial investigations were to examine how different system calls moved data in different ways between the user space application and the device driver for network transmissions. This section discusses two such system calls which affect the throughput and CPU utilization significantly. The reader is to note that these are not the only system calls that support network transmission, but rather show the extreme cases considering version 2.6 of the Linux kernel.

4.1.1 `write()`

One classic way to do network transmissions is to use the `write()` system call. This function copies the data from a user space buffer to a kernel space buffer

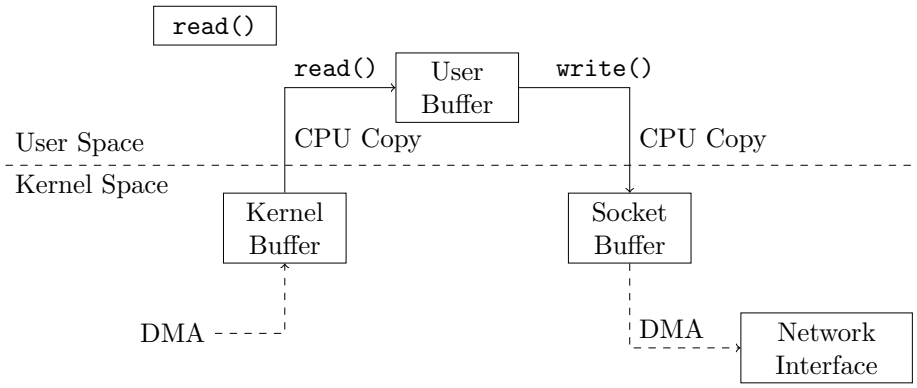


Figure 4.1: Data path using `read()` and `write()`

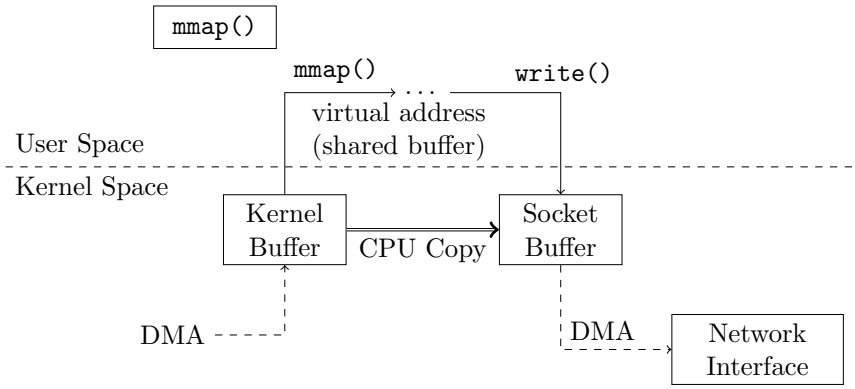


Figure 4.2: Data path using `mmap()` and `write()`

associated specifically with the currently used socket, which is then DMA'ed by the device driver to the network interface. If the requested data is not currently available in user space it has to be read first. One way is to use `read()` which reads data from the kernel space buffer to the user space buffer. This results in four context switches and two data duplications. Another way which avoids the data duplication involved in the `read()` mechanism is to memory map the kernel data to a virtual address of the user space application using `mmap()`, thus sharing the kernel buffer with user space. If the data is always read from the same source, a video file for instance, the overhead of context switches involved in reading data could be avoided as well. It is important to note that if the data is not found in the kernel space it will be DMA'ed to kernel space first. The data copies and system calls involved in both these cases are shown in Figure 4.1 and 4.2 [24]. The data is also checksummed while copied in these cases since the overhead added by checksumming is negligible when all the data is processed anyway.

4.1.2 `sendfile()`

Another system call that has been used more recently to improve network performance significantly is `sendfile()`. Its functionality is to copy data from a source file descriptor to a destination file descriptor without the need of going through user space. This eliminates the need of extra work for setting up and tearing down virtual mappings associated with the data. In Linux versions more recent than and including 2.6.17, this system call internally calls another function named `do_splice_direct()` to achieve this functionality. The 'Splice' mechanism is used to achieve what is called a 'Zero copy transfer' during network transmissions and other file copy operations. When this mechanism is used data is not even copied between the kernel buffer and the socket.

The basic data unit used in the Linux for memory management is a page (or part of a page). The Linux kernel maintains a page cache which is a cache of the page-sized data blocks of files. This is mainly useful in systems where data can be stored in an external memory like a hard drive where data access time often can be very long. If the page containing the requested data is not already in the cache a new entry is added to the cache and filled with the data read from the external memory. Same principle applies for the data being written out [7].

At the heart, the splice mechanism uses pipes to move the data around. Here a pipe is an in-kernel buffer which is implemented as a set of reference-counted pipe buffers as shown in Figure 4.3. This means that there are no physical copies involved but still the data can be copied around many times. A pipe buffer is a descriptor to a single page of data, and the number of pipe buffers are hard coded to 16 in current Linux kernel versions. This allows for a maximum of 64 KB data to be piped at a time. Internally these pipe buffers are implemented as circular buffers. [3] [4] A pipe is created and cached in the kernel when transferring data between two file descriptors when neither of them is a pipe. [27]

When using the `splice()` call to transfer data from one file descriptor to another the following happens:

1. Splice data from the source file to a pipe, `splice_to_pipe()`. This means the page buffers pointing to the memory pages containing the file data are added to the pipe. It is to be noted that data is not copied but the reference count of the pages used are incremented.
2. Splice data from pipe to destination file, `splice_from_pipe()`. This is similar to the above except data is moved from a pipe to the file.

For network transmissions the destination file descriptor is a socket. Pages from the pipe buffers are moved to the socket as shown in Figure 4.4. In the socket, socket buffers (`sk_buff`) are created by taking the available hardware support into consideration. These buffers are then added to the output queue or sent out immediately. As shown in Figure 4.3, `sk_buff` is capable of storing chunked pages of data which are organized in an array in the shared socket buffer area. Since the data and the header can be scattered in memory instead of existing in a contiguous memory area, data gathering (scatter/gather(SG)) support is required if copying needs to be avoided. The Linux API has support for this functionality. As the DMA controller in the ARTPEC-4 platform supports SG functionality a

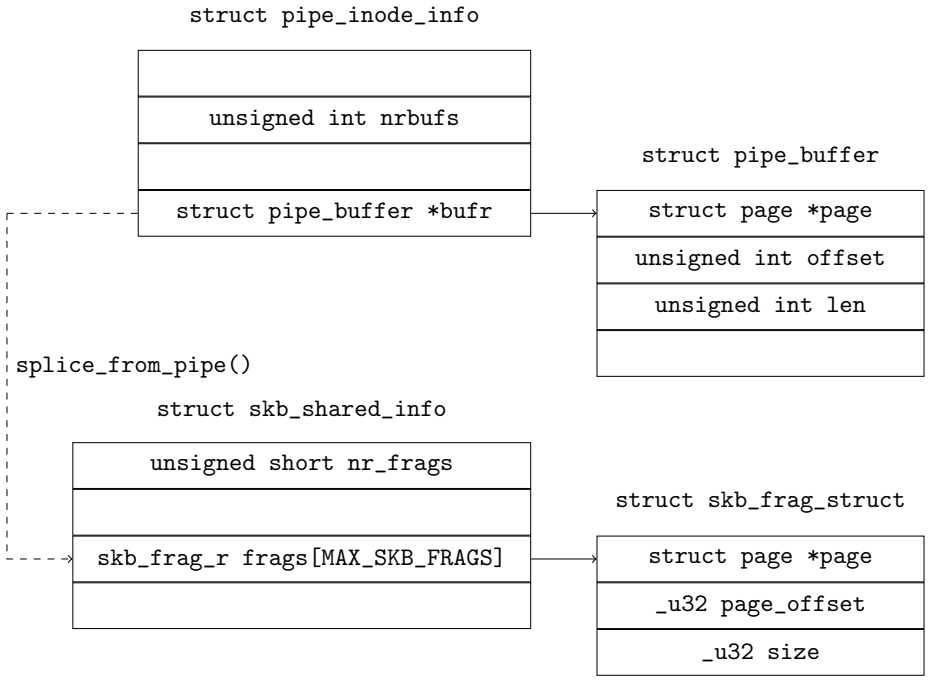


Figure 4.3: Class map of the pipe structure used in the `splice()` call

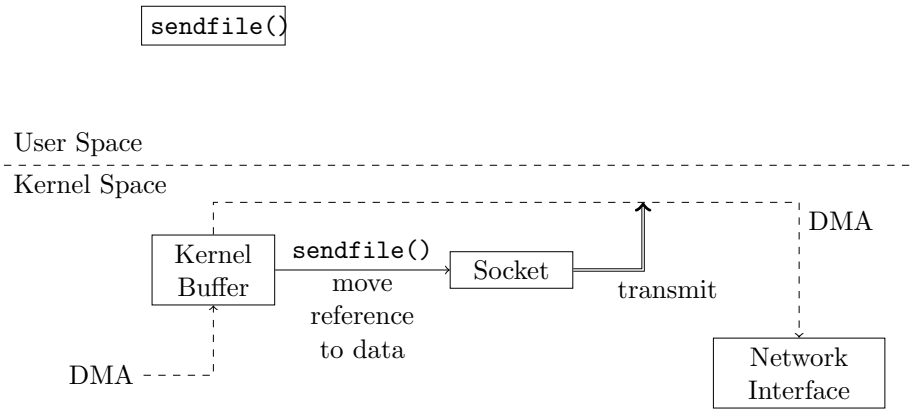


Figure 4.4: An illustration of Splice mechanism.

true zero copy is achieved. This splice mechanism is very useful in Axis cameras as they mainly behave as server systems just moving data without looking at it. Presently software used in the cameras utilizes this call to deliver video data to the clients.[15][14][26]

4.1.3 Network traffic analysis

In order to measure the performance of these system calls for Axis camera a generic client-server suite of programs was written [1]. These programs supported configuration parameters for connection setup; protocol, data block size, number of clients, transfer mechanism, and number of data transmissions. A test file of size 100858 Bytes was used to generate an arbitrary amount of network traffic. The setup included an Axis camera, a computer, and a Fast Ethernet switch. The server program was run on the camera and all clients were run on the PC. This program was only used in the initial stages of the project, and was later discarded since real-life scenarios were used instead.

Initial analysis of the network packets revealed some interesting aspects that affected throughput and CPU utilization. Only TCP is discussed as it was the main focus for improvement. Some early observations were that:

- With the Network Processor enabled the outgoing Ethernet full frames were only 1498 Bytes long, 16 Bytes short of MTU (1514 Bytes) for standard Ethernet frames. These 16 Bytes of data appeared as separate small frames on the line. The small frames carried a payload with the length that was an integral multiple, 'n', of 16, where 'n' is consecutive frames of 1498 Bytes length. This behavior was common to both `write()` and `sendfile()` system calls. The same applied to UDP traffic as well. Figure 4.5 illustrates the problems experienced. Since the network line had lots of these small packets, the resulting overhead for the network affected the network throughput significantly. It did also utilize the DMA channels, the network processor,

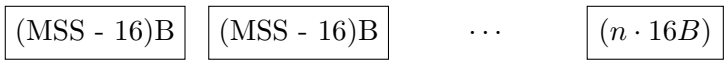


Figure 4.5: An illustration of non-standard MTU packets generated.

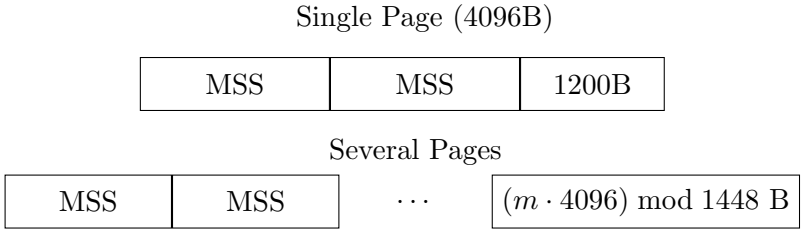


Figure 4.6: An illustration of the segmentation observed in `sendfile()`

and the Linux kernel itself in a very inefficient way.

- With the Network Processor disabled standard full frames of 1514 Bytes were transferred. The two behaviors in this case that was found interesting are:

  - `sendfile()` always transmitted data in a way such that small sized frames appeared after a number of full frames. The frames between two small packets and including the latter small packet formed an integral multiple, 'm', of pages of 4 KB. What this means is that the outgoing traffic consisted of $\lfloor \frac{m \cdot 4096}{1448} \rfloor$ of full frames followed by a small frame of size $(m \cdot 4096)$ mod 1448. Analysis of the Linux implementation of this call stack revealed that this function's basic operational data unit is a memory page. It sends one memory page at a time to the socket using `tcp_sendpage()`. The non-MTU packet is generated when switching to a non-adjacent page in memory and that page is not ready. Thus, data that occupies multiple non-contiguous pages results in a non-MTU packet for every hole, increasing overhead, and lowering throughput. See Figure 4.6.

  - Using the `write()` system call resulted almost always in full frames. Since the investigated protocol is TCP, this function would subsequently call `tcp_sendmsg()`. As DMA supports SG, the implementation of this function utilizes this feature, and uses pages as basic unit of data handling.

It should also be noted that both functions, `tcp_sendpage()` and `tcp_sendmsg()`, handle the segmentation process of TCP protocol.

The client server suite of programs were mainly used to measure performance initially and for debugging throughout. Other methods as discussed in Chapter 3.

Method	Connections	Throughput(MB/s)
TCP write	1	11.78
TCP write	4	11.79
TCP write	16	11.77
TCP write	32	11.75
TCP sendfile	1	11.64
TCP sendfile	4	11.77
TCP sendfile	16	11.76
TCP sendfile	32	11.76

Table 4.1: Throughput figures with Network Processor disabled

Method	Connections	Throughput(MB/s)
TCP write	1	11.77
TCP write	4	11.75
TCP write	16	11.73
TCP write	32	11.74
TCP sendfile	1	11.07
TCP sendfile	4	11.13
TCP sendfile	16	11.55
TCP sendfile	32	11.56

Table 4.2: Throughput figures with Network Processor enabled

4.1.4 Measurements

The measured TCP throughput for different test cases are tabulated in Table 4.1 and 4.2.

It can be observed that the throughput is close to the 100 Mbit/s line speed used, and both `write()` and `sendfile()` perform almost the same. In the test case with one client, `write()` is a few hundred KB/s faster than `sendfile()`. This is possible because, to have a fair measurement the data is read into the user buffer for `write()` before initializing socket in the server application, and hence the data pages are already in the page cache whereas for `sendfile()` pages are fetched continuously during the transmission. This is also the reason why `write()` has slightly better performance measures than `sendfile()` for other cases as well. Over time and with many clients, data is distributed all over the available memory areas. This was also evident when packets on the line were analysed for pages being sent more non-contiguous.

ETH	ETH	IPv4	TCP	Payload
(14)	(16)	(20)	(32)	(1432)

Figure 4.7: Calculated header (in Bytes) before correction applied

Method	Connections	Throughput(MB/s)
TCP write	1	11.79
TCP write	4	11.79
TCP write	16	11.79
TCP write	32	11.80
TCP sendfile	1	11.78
TCP sendfile	4	11.79
TCP sendfile	16	11.79
TCP sendfile	32	11.80

Table 4.3: Throughput figures with Network Processor enabled and driver MTU correction

4.1.5 Small packet problem

In order to understand the problem with non-MTU sized frames for the scenarios where the Network Processor was enabled, the network device driver and Network Processor were investigated further with focus on packet handling and MTU calculation. Very soon a bug was discovered in the network driver when calculating the current MTU size. When the Linux kernel calculates the size of header it considers TCP, IPv4, and Ethernet protocols. The network driver, after having received header size information from kernel, treated this calculated header size as if the size of the Ethernet protocol header was excluded, and accounted for this header explicitly in MTU calculation. This resulted in a miscalculation since the Ethernet protocol header size was being considered twice. After masking to achieve proper alignment, the payload MTU became 16 Bytes lesser which was delivered to the network processor for packet handling. The header size calculated in the network driver is shown in Figure 4.7. After this bug was corrected in the driver by adding in an extra header throughput improved tens to hundreds of KB/s as can be seen in Table 4.3.

4.1.6 Splice Modification

Linux kernel implements a function, `do_tcp_sendpages()` which was designed to send an array of pages to the socket instead of going through the entire network stack for each page sent. In the kernel implementation `tcp_sendpage()` actually calls `do_tcp_sendpages()`, but with a single page array. It is not clear why it was

Streams				Throughput (KB/s)			CPU
TCP	UDP	Audio	Changes	TCP	UDP	FPS	Load
10	0	2	None	951.34	—	—	66.15%
5	5	2	None	912.76	867.68	21.4	58.36%
10	0	2	Splice	1038.17	—	—	52.56%
5	5	2	Splice	984.77	915.06	23.4	54.69%

Table 4.4: Performance results from modifications to splice implementation.

decided to only send one page at a time instead of sending all the pages in the pipe using `do_tcp_sendpages()` with a page array as argument. In order to observe how kernel utilization improved by using `do_tcp_sendpages()` to send all pages in the pipe at once to the socket, changes to the kernel `sendfile()` call stack was made.

`splice_from_pipe()` calls `splice_from_pipe_feed()` which eventually invoked the function `tcp_sendpage()`. To make changes to the Linux kernel, the function implementation of `splice_from_pipe_feed()` was modified to call directly `do_tcp_sendpages()` with an array of pages built with the pipe buffer pages in the pipe. In order to keep the implementation simple this modification assumed that the destination file was a socket.

4.1.7 Validation

To verify for the improvements in reducing CPU utilization, the kernel run on the Axis camera was profiled, and results for different test cases are tabulated in Table 4.4. All different test cases have added computation demanding audio traffic to put some extra load on the CPU, since the network interface itself would become the bottleneck when measuring only with video clients. In a TCP traffic only scenario CPU utilization decreased by 20.54% and throughput increased by 9.13%. For the mixed traffic scenario CPU utilization decreased by 6.29%, TCP throughput increased by 7.89%, and UDP throughput increased by 5.46%. The difference in CPU utilization is because UDP is a datagram protocol and the data is handled by the application layer. The mixed traffic case used in these measurements was mainly to measure the average FPS rate improvement, and it can be observed that FPS increased by roughly 2 frames. Since all the pages are confirmed before being sent to `do_tcp_sendpages()`, no small packets were observed on the wire, hence increasing efficiency further. An overall observation is that the modifications made to the network call stack improved both the CPU efficiency and network throughput.

4.2 Cache architecture

As already described in Section 2.5, ARTPEC-4 supports a two level cache hierarchy with L1 and L2 caches. The L2 cache is a system level cache that caches data from main memory for the CPU and the DMA channels. Each DMA access has a width of 32 Bytes, whereas the L2-DDR2 interface width is 128 Bytes. The L1 cache lies between the L2 cache and the CPU, and is not coherent with L2. Hence, everytime a DMA transaction is initiated by software manual blasting of relevant pages in the L1 data cache.

Since the camera acts as a server moving video data from memory to the network interface without requiring the CPU having to look at the data, blasting of the L1 data cache can significantly affect CPU performance. Another potential problem related to caches is the video subsystem that moves data to memory in a way that it bypasses the L2 cache. As a result of these data transactions, L2 is non-coherent with memory, and every DMA transaction invalidates L2 cache lines so that they are fetched from memory before being read by DMA. Hence, for the given hierarchy of memory with cache and the 2 threaded MIPS architecture it is of interest to measure their effect on CPU utilization.

Profiling using JTAG showed approximately 4% of the total execution time under load (4 TCP video streams, 4 UDP video streams, and 2 UDP audio streams) was being spent in cache flushing functionality. This is much less than the earlier predicted value of about 20% that comes from earlier chip generations. If the L1 cache is made coherent with the L2 cache this would potentially increase CPU hit rates, hence its efficiency. After these discoveries this investigation track was considered a dead end, and no further analysis was carried out.

4.3 Ethernet driver queues

In a multi-core, multi-threaded system, locks are used to serialize access to shared resources avoiding race conditions. As the number of threads and cores increase, these locks can easily become bottleneck of the system since every thread and core wanting to use the shared resource must wait for it to be free. One such lock that would affect the network performance is located in the network driver, where the DMA transmit queue used for queueing network data uses a lock, as it is a system wide shared resource.

Before proceeding to make any changes to the transmit queue the utilization of the lock was measured using timers to find out how much of the time the lock is held between two cores. The percentage of time for which the lock is held during queueing, and rescheduling when egress is busy is shown in Table 4.5. As can be observed the lock is not held for more than 2% of total execution time, and rescheduling happens less than 0.1%. This accounted for CPU waiting for the lock approximately 8 times per second.

To make sure this track was not of potential interest the total percentage of time spent in transmitting, (until packet is output if transmit queue is available or rescheduled for later transmission if egress is busy, and returned) `do_sendfile()`, was measured and found to be no more than 5% under high loads. Thus, this track for potential improvements did not prove promising. Since the workload required

Measured part	Utilization
Ethernet queue lock	1.269%
Egress queue rescheduling	0.046%
Entire TX Path	4.269%

Table 4.5: Results from timer measurements in the network transmit path.

to change this architecture did go beyond the scope of the project compared to the potential improvement, it was decided that this track was not a target for further investigation.

4.4 TOE in network processor

In a server system dense network traffic would consume almost all of the CPU resources leaving little or no room for other tasks. A lot of these CPU resources are spent on checksum calculations and segmentation of transmit data. With General Segmentation offload (GSO) [11] implemented in ARTPEC-4, the Network Processor is used to offload the main CPU from this kind of network stack processing. As a result total overhead for the transmit path takes less than 5% of the CPU resources (see Figure 4.5). The main idea here is to offload the most demanding parts of the network stack processing. The most demanding parts are the most commonly executed code as well called the common path. For TCP protocol the common path occurs when the connection has reached the ESTABLISHED state. The current implementation of the network processor partially offloads the transmit part of the common path. This includes TCP segmentation, TCP header processing, and checksum calculation for TCP/UDP/IP protocols. Flow control and TCP timers management are still handled by the kernel software.

With this fast processing of egress traffic handling of the acknowledgements on the receive path becomes a bottleneck. This is a little unexpected, since the throughput on the transmit path is several orders of magnitude higher. In ARTPEC-4, the receive path of the Network Processor engine is transparent to received packets. All the received packets are, therefore, handled by the main CPU with the only exception being checksum calculation since dedicated hardware exists. Profiling results, both while profiling with the FS^2 hardware probe and over network, have shown a high CPU usage for the receive path in the Linux kernel. Traffic analysis have shown that TCP packets are received continuously in the RX path to update TCP parameters. Since TCP is a stateful protocol the Linux kernel must update and keep track of several connection parameters for flow control, congestion control, and retransmission timers based on the received packets.

Every time a network packet is received, the CPU is interrupted to process the packet. In a high-speed network this can create thousands of interrupts per second. Each of these interrupts involve wasting CPU cycles due to context switching overhead, repeated execution of the interrupt handling code, and replacing cache

contents. The network driver for ARTPEC-4 is NAPI-compliant [12], which mitigates interrupts and packet trottling by disbaling interrupts and switching to polling for high traffic scenarios. Linux 2.6 also implements a Large Receive Offload (LRO) [6] and [16] feature in software for TCP traffic. LRO combines the received TCP packets to a single larger packet and then pass the packet up the network stack, reducing per-packet overhead at higher levels. With both NAPI and LRO features assisting, receive path profiling results still showed $\approx 13\%$ of kernel utilization for TCP traffic.

Considering the traffic scenario for Axis cameras where traffic is one-way and the only received packets are just TCP ACK's carrying no data, this was considered being a very high utilization number. If the CPU gets interrupted for every other incoming TCP ACK packet as is the typical case for TCP traffic, a 100 Mbit/s data bandwidth produces

$$\frac{100.0 \text{ Mbit/s}}{2 \cdot 8 \cdot \text{MTU}} = \frac{100.0 \text{ Mbit/s}}{24224} = 4128$$

interrupts just for TCP ACK traffic control at the worst case, which could explain the somewhat high utilization.

Packet analysis revealed that the TCP video streams were mainly used in one direction as suspected. The receive path for the camera had almost no utilization, and all incoming traffic were just TCP overhead with no data except for TCP state data along with acknowledgement number. Since measurements have shown, compared to the amount of traffic, a very high CPU utilization ($\approx 13\%$) it was considered critical to reduce the amount of traffic reaching the Linux kernel itself. Thus, because of the high receive path utilization, a proof of concept for TCP acknowledgement offloading was developed for the ARTPEC-4 Network Processor. This could be achieved by only developing a new firmware, so hardware changes could be avoided. To achieve a highly significant improvement in CPU utilization reduction small hardware changes in the Network Processor would have been preferable.

4.4.1 TOE Algorithm

Even though the Network Processor is capable of handling any protocol, it was decided only to implement TCP acknowledgements handling with a simple algorithm due to the limited instruction and data memory inside the Network Processor (4 KB total). Retransmission timers handling, flow control, and congestion control algorithm were decided to still be handled by the kernel on main CPU as for the transmit path. The algorithm mitigates the number of interrupts generated for the main CPU on the receive path.

The receive path for the Network Processor is configured to manual mode so the ingress packets could be processed in the Network Processor. Upon reception of every packet the Network Processor is interrupted, and the control is passed to an interrupt handler to process the received packet. Only TCP traffic is processed in the interrupt handler and all other protocol packets are transparently passed up the network stack.

Contexts about established connections are registered in a small connection table inside the memory of the Network Processor. SYN in an outbound packet

adds an entry to the established connections table, while FIN and RESET flags are used to remove a connection from table. Due to the memory limitation only 16 simultaneous TCP connections are allowed in the table. It should be noted, however, that this is a configurable parameter that is set to 16 due to the memory constraints. Also, the networking hardware running at 100 Mbit/s speed is limited in capability to support roughly 12 TCP video connections at full speed (25 FPS in our measurements corresponded to approximately 1 MB/s throughput). Connections that are not registered are treated transparently on the receive path.

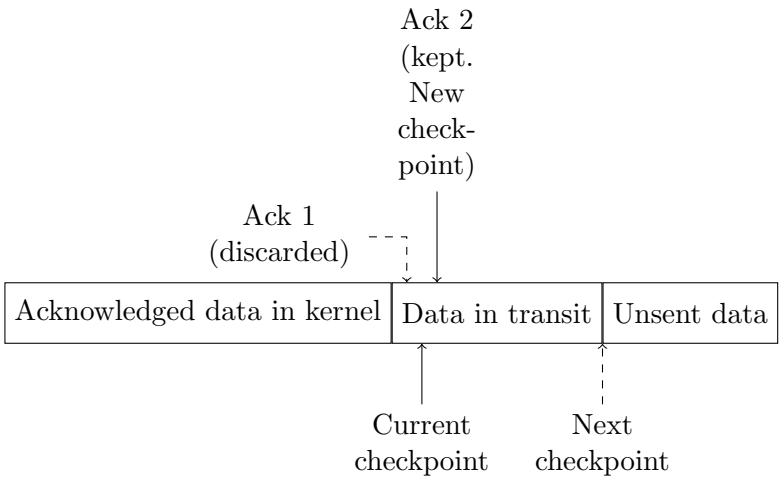
Whenever data is sent on a registered connection, the last sent data byte of the TCP segment is remembered as 'unacked' by saving its last sequence number. The 'unacked' state variable is initialised with the sequence number received in the initial TCP SYN packet. Another state variable called 'checkpoint', initialized to 'unacked' during connection registration, is used to act as point at which acknowledgement has to be sent to the network stack. The 'checkpoint' parameter is updated to 'unacked' to form a new checkpoint when ever the received acknowledgement passes the 'checkpoint', or a new 'unacked' value is calculated and 'checkpoint' is lagging behind the received ACK window.

Incoming acknowledgements with an ACK number smaller than the 'checkpoint' or greater than the lagging 'checkpoint' are discarded if not being a duplicate ACK or carrying TCP data. This allows the kernel to still manage the TCP connections. This is because the kernel offloads large TCP segments to the Network Processor, and hence has large retransmission timers. As a result, all ACK packets of a segment can be processed in the network processor without passing on to the network stack. The exception is the last ACK of a certain segment (the kernel must get a final acknowledgement). Received packets containing data would pass through, and the implementation of the duplicate ACK feature [2] could co-exist with the network processor by doing a simple equality check with the highest received ACK number so far and the incoming packet's ACK number. An illustration of how the algorithm works is shown in Figure 4.8.

Because of the small receive memory (512 Bytes) received ACK packets cannot be stored inside the Network Processor's memory. In the case of duplicate ACKs, the first ACK is discarded (but the number is remembered) while the second ACK is forwarded to kernel. The Linux kernel treats this as a normal TCP ACK, and as such the sender of duplicate ACKs would only be able to reach the server with 3 out of its total 4 duplicate ACKs. It is not until the third ACK packet arrives the kernel would notice a duplicate ACK.

The case of sequence numbers wrapping around their 32 bit value range complicates the comparison of TCP sequence numbers. This problem was solved by using modular arithmetic macros for sequence number arithmetic as explained in [28, p. 810]. Also, TCP acknowledgements received out of order would be no problem with this algorithm either because it would just discard the acknowledgement with earlier number since it's already acknowledged by the the later sequence number.

One area the algorithm affects, however, is the TCP window updates. Since the discarding of packets also meant discarding of TCP parameter updates, this leads to the client's TCP receive window not being updated in the Linux kernel between checkpoints. In the project setup, the window sizes of the client and server



After Checkpoint Update:

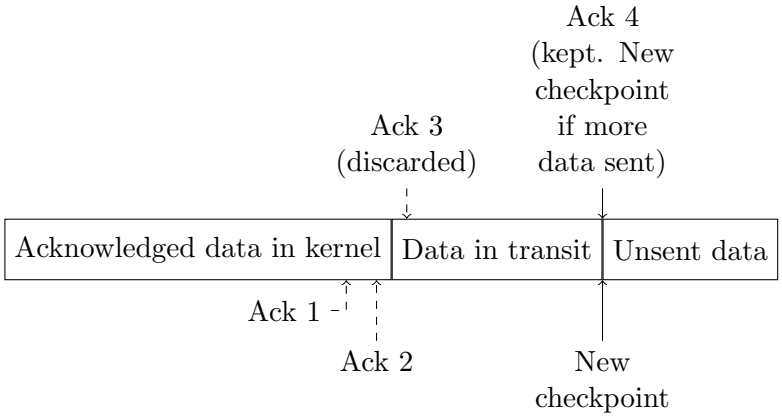


Figure 4.8: Illustration of the TCP ACK filtering algorithm in action.

Streams				Throughput (KB/s)			CPU
TCP	UDP	Audio	Changes	TCP	UDP	FPS	Load
10	0	2	None	951.34	—	—	66.15%
5	5	2	None	912.76	867.68	21.4	58.36%
10	0	2	NP	1025.58	—	—	58.21%
5	5	2	NP	951.98	889.85	22.3	57.22%

Table 4.6: Results from offloading TCP ACK processing

was large enough to not suffer any performance penalties from the discarding of in-between checkpoint window updates. At the worst case, the receive window size at the client side dropped from ≈130 KB to ≈90 KB.

4.4.2 Validation

By offloading TCP ACK processing to Network Processor the ACK packets passed up the network stack are reduced by up to 97.8% with a measured typical case of about 85-90% for a larger data segment offloads. The results of profiling are tabulated in Table 4.6 where it can be observed that the kernel load is reduced by 12% for TCP only video traffic, meanwhile throughput is increased by 7.8%. Improvements in CPU utilization reduction and throughput increase are 2% and 4% respectively for mixed traffic. This number is quite small because UDP is a connectionless protocol.

Chapter 5
Conclusions

The measurement results in Chapter 4 showed that CPU utilization decreased significantly for the ARTPEC-4 SoC when system optimizations were made for network traffic. It also concluded that these optimizations improved the throughput for TCP traffic as well.

One approach that is used to efficiently transmit data from the file system memory to the network interface is by optimizing the transmit path. The data is not copied around in software before being sent over DMA to Network Processor memory, but instead references to the pages of data are used throughout the call stack. Instead of handling a page at a time all the pages of the data in a pipe are handled at once to be transmitted over DMA. This way the network call stack is executed once instead of one time for every page of data. Also, the network driver does not have to program every page of data, but instead program it once for the entire data for the transmit interface. It is important to note that the actual amount of data programmed is dependent of the receiver window size to allow flow control. For TCP traffic CPU utilization is lowered by 20%. Also, the throughput increased by 9%.

As initially suspected the transmit path did not prove to be using up a significant amount of CPU resource, but instead the receive path turned out to be the bottleneck. Another major optimization was done by offloading acknowledgements on the receive path to be handled by Network Processor instead of the kernel. As the NETPROC supports segmentation offloading, the kernel need to receive an ACK only for the segment itself and not for all the constituting frames sent over the network. By handling all the 'unnecessary' ACKs in the NETPROC and not interrupting CPU for every ACK packet, the CPU utilization is decreased by 12% for TCP traffic. By having more free CPU resources available the transmit path throughput increased by 7%.

Having both the optimizations act together for TCP traffic CPU utilization decreased by approximately 60% and throughput improved by 10%. The video stream reached it's maximum of 25 FPS. Table 5.1 shows the profiling results for various test scenarios with different optimizations. Thus, by evaluation, optimizing kernel for Axis application and improving the NETPROC hardware, an even highly significant improvement can be achieved in lowering the CPU utilization.

Even though cache hierarchy and transmit rings tracks did not prove candidates for potential improvements they can be improved to decrease CPU utilization

Streams				Through (KB/s)			CPU
TCP	UDP	Audio	Changes	TCP	UDP	FPS	Load
10	0	2	None	951.34	—	—	66.15%
5	5	2	None	912.76	867.68	21.4	58.36%
10	0	2	Splice	1038.17	—	—	52.56%
5	5	2	Splice	984.77	915.06	23.4	54.69%
10	0	2	NP	1025.58	—	—	58.21%
5	5	2	NP	951.98	889.85	22.3	57.22%
10	0	2	All	1047.65	—	—	23.56%
5	5	2	All	1044.70	953.15	24.8	41.36%

Table 5.1: Results from all changes

further as discussed in the Section 5.1.

As shown, the stated goal to optimize transmission by improving network processor and transmit call stack has been met.

5.1 Discussion

Several ideas had appeared throughout the project. Many of them disappeared as more knowledge about the ARTPEC-4 platform and the Linux Kernel were obtained. However, there still exists several suggestions for improvements in several fields of the ARTPEC-4 SoC and the software running.

Pipe buffers in the Linux kernel

In Linux kernel 2.6 the number of pipe buffers are hard coded to 16, and hence a maximum of 64 KB of data can be piped out to the driver at a time. Also, the number of page fragments the socket buffer supports is 16. For data amounts larger than 64 KB movement of data through the pipe happens in chunks of 64 KB. By increasing the number of pipe buffers the performance could be further increased, but this will also increase kernel memory usage. Great care has to be taken when deciding the number of pipe buffers in the Linux kernel, since it does not only affect network performance. Also, without the modifications made to the splice mechanism in this project, the number of pipe buffers is irrelevant since the entire call stack is executed for each buffer. It should also be noted that, in theory, the performance gain of changing from 16 buffers to 32 would reduce the number of system calls in the same magnitude as changing from 8 to 16 would do. Hence, the performance gain from incrementing the number of buffers decreases as the number of buffers increase.

Cache coherency

Cache hierarchy and the L2 controller can be optimized for the AXIS application by making the L1 cache coherent with the L2 cache. This would prevent the unnecessary blasting of data in the L1 cache and put less load on the CPU, since it saves the overhead involved with manual blasting. Also, the CPU could use the cached data more frequently and efficiently. To avoid this full cache coherency could be implemented so this inefficient use of L1 would no longer be a problem.

L2 cache controller

As DMA fetches the data from memory through the L2 cache, and a compressed video frame is large enough to fill significant number of cache lines in L2 and replace some of the data necessary for CPU, fair share of L2 to CPU is necessary for its efficient utilization. This is important so that CPU doesn't spend hundreds of cycle waiting for data to be fetched from memory. L2 controller can be designed such that it is possible to configure L2 to be divided between the DMA and the CPU. This is useful, as video data read by DMA is never seen by CPU and the data necessary for CPU is never replaced by cache lines involved in DMA transactions. One possible way to do this is by assigning a few ways in a set to DMA and the rest of the ways to CPU. The hardware changes required to make these modification would be minimal. The decision on number of ways to be assigned should be made based on profiling results.

L2 controller in ARTPEC-4 doesn't implement any performance registers. These registers which keep track of number of hits and misses can help tune L2 to achieve a high performance for both the DMA and the CPU.

MIPS cache prefetching

MIPS specification supports prefetching of data to cache. By implementing basic prefetching support in cache controller the performance can be improved significantly. DMA can initiate prefetch of next line of data (cache line size) to L2 in parallel to delivering video data from L2 to Network Processor memory. This improves the effective bandwidth, and will significantly improve the throughput and efficiency of DMA and Network Processor.

Multiple locks for the Network DMA TX Ring

ARTPEC-4 implements a hyperthreaded architecture, where 2 threads execute in parallel and a single transmit queue is shared between threads. One possible improvement here is, as investigated, to introduce multiple DMA transmit queues and design an algorithm to control access to these queues (array of context descriptors) to further minimize waiting on the resource. This option is viable, as each DMA physical channel is able to handle an unlimited number of virtual channels, theoretically. Also, DMA supports multiple context descriptor with a list of data descriptors each. Multiple queues increase the probability of getting a free lock. Another possible solution is to design the transmit ring to mimic a single reader and multiple writer queue without using spinlocks. This requires using MIPS

instructions like load linked (LL) and store conditional (SC). As already investigated in the project, this lock currently does not cause any significant performance penalty. But, since the number of cores in CPUs are expected to increase in the future this could pose future problems. When increasing the number of cores this should be taken into consideration again, since it may be a major contributor to performance penalties.

Utilizing multiple destination feature in NETPROC

The Network Processor supports sending the same payload to multiple destinations. For the Axis cameras this can be very useful, since it is the four same video images that are sent to the connected clients. As discussed before, the kernel only passes references to the data down to the device driver. By making changes to the driver to generate one context descriptor with a list of data descriptors for data and one context descriptor each for a TCP segmentation offload header, the Network Processor can be utilized to its full potential. Also, the DMA and bus utilization is greatly reduced as the number of times the data is read from memory to the network processor memory by DMA is reduced significantly.

RX interrupt handling in NETPROC

Every received packet generates an interrupt for the NETPROC CPU, transferring the control to the interrupt handler which processes the received packet. The NETPRO CPU operates at a clock frequency of 100 MHz, and with 100 Mbit/s line speed the network CPU has to wait about 400-500 clock cycles before processing of the received packet header could begin. The interrupt system can, therefore, be improved so that it generates an interrupt to the network CPU after a configurable number of Bytes. This way the interrupt handler can ensure it's looking at the most recent header without having to stall the transmitting path, like a 500 cycle would do. It will also, further, improve the transmission capability and programmability of the Network Processor.

Receive memory in NETPROC

Another interesting topic throughout the thesis has been the single receive memory of 512 Bytes in the Network Processor. The decision, whether to keep or discard an incoming packet has to be made before these 512 Bytes are filled up, or the next packet arrives overwriting the previous one. With a Gigabit Ethernet interface, this memory can fill up very fast and the network processor CPU may not have enough time to make decisions based on header data. To achieve the same performance of incoming packet processing in a Gigabit Ethernet interface as a Fast Ethernet interface, the clock frequency of the Network Processor needs to be increased tenfold! This is not a viable option for power reasons, and some other way of maintaining decision times for the NETPROC CPU has to be created.

One way of doing this could be to divide the receive memory into two parts. This way, one memory could be processed when the other is being written to. Since most of the incoming traffic in an Axis system, transferring video data over TCP, is TCP state data packets would fit even within these small 256 Bytes memories.

Packets that exceed the RX memory size would have to be accepted, but since they usually carry data this wouldn't make any difference with the algorithm implemented in this project. Just increasing the RX memory size wouldn't be beneficial from a TCP offloading perspective, since a high packet frequency would be equally bad with one RX memory regardless of memory size.

One area that, however, could benefit from increasing the RX memory size though is hardware checksumming. Since the hardware checksum unit in the receive path calculates the checksum, the packet is transferred to memory and processed up the network stack, and then if checksum verification fails the packet is discarded. By increasing the size of the receive memory to fit at least one full packet the checksum comparison can be done in the network processor instead, and discard corrupted packets there without the kernel seeing them. This also increases the DMA utilization as the corrupt packet is not written to memory just to be discarded later by network stack. Since CRC errors are rare in wired Ethernet devices today, this may show little or no benefit, but for WLAN connected cameras this could be beneficial.

Memory constraints in NETPROC

The offloading functionality on the receive path is kept to minimal in the NETPROC CPU because of the low instruction and data memory (4 KB). Also, the Network Processor was mainly designed to offload the transmit path in the network stack. Since the TCP Offloading must share memory area with the already existing Axis processing code, the available 4 KB quickly drains. Work is being carried out in order to optimize the existing Axis code and remove unneccessary processing. When this is fixed more functionality can be offloaded to the network processor, as memory areas used by code will be freed up. Even if it is decided to implement more functionality in the Network Processor, a memory of 4 KB would still be too small and in this case the memory size should be considered before doing such decisions.

TCP offloading in NETPROC

Implementing a full TCP offload engine in the NETPROC block would require major hardware changes and also increase the software complexity in the NETPROC firmware, since the NETPROC initially was not designed for such a complex task. However, as have been proven, some parts of the TCP processing could be implemented in the NETPROC block. One thing to keep in mind is that the Linux kernel must be more aware of this offloading than it is at present, because the implementation in the project could affect TCP performance of the kernel since ACKs are used for congestion control, retransmission timers, and flow control. Missing a few of those ACKs could potentially affect the connection in some cases if the kernel is unaware of packets being discarded due to performance optimizations.

One way to implement TOE would be to implement a Common Path offload as discussed by an earlier thesis [9]. This way the kernel would initiate connections and hand them over to the NETPROC, but it will still require the NETPROC to implement TCP congestion control algorithms and retransmission timers.

Another suggestion is to keep the filtering algorithm and tune the Linux kernel to be adjusted for the filter. It was seen in the project that some manual changes of Linux kernel TCP parameters were required (mainly buffer sizes) in certain cases. These cases involved transmission of uncompressed video data (roughly 5.0 MB/s) where the filter actually decreased performance, but after some tuning increased it again as expected. This option would reduce the NETPROC complexity, but instead needs more hacks and workarounds to be implemented.

Regardless, any of these two above mentioned options would require changes in the Linux kernel, where the driver needs to be altered and the network stack also would need modifications. The main difference is the possible performance gain and the required workload to achieve such a gain. It should be investigated further as to what option is the most suitable, as it is unclear today which of the options would be the better one in terms of price/performance.

UDP performance

In this thesis TCP has been the protocol of interest. It should be noted that UDP seems to be more resource hungry compared to TCP. This shouldn't be the case and there are several potential improvements in UDP performance available. Segmentation and checksumming could be done in the Network Processor, where UDP then could be fully offloaded with the current available infrastructure since it is a stateless protocol. One way to do it is to implement a network driver implementing a file system and also change the NETPROC firmware. The small firmware memory would be the bottleneck here, since it is unclear how much memory is needed for this code. For TCP this is much harder since TCP implements congestion control and is a stateful protocol.

Current status

The work carried out in this project is still at an experimental stage. However, it could be used as is but with no guarantee that it would benefit all scenarios. As already mentioned earlier, the problem with using the implemented code as is would be the TCP congestion control algorithms.

For the splice modifications very minor changes would be required to implement it in a more proper way. In the current implementation some function pointers are bypassed and some shortcuts have been taken in the call stack, but by adding a few multiple-page functions to the kernel these shortcuts could be implemented in a clean manner.

The Network Processor firmware is also usable as is, but to work proper in all conditions it would require major kernel changes that would never be accepted by the Linux developers. A small patch set for the Axis platforms could be maintained provided it's considered worth the performance gain.

5.2 Summary

The report covers the thesis work done in order to reduce CPU utilization in network transmission of Axis camera video images. It also shows the throughput

enhancements achieved along the way. Main focus of optimization is directed to the TCP protocol where various architectural changes impacting CPU utilization are investigated and discussed. Of these, two investigation tracks proved efficient for potential CPU improvements. First, by reducing repetitive processing of data by network stack a 20% improvement is achieved in CPU utilization reduction under high network traffic. Second, offloading of receive ACK processing to Network Processor yields an improvement of about 12%. A combined optimization results in an approximately 60% improvement in CPU utilization. The resultant throughput improvement is 10%. With some architectural improvements and network driver modifications, a much significantly higher optimization in CPU utilization can be achieved as well.

Bibliography

[1] Sockets tutorial. http://www.linuxhowtos.org/C_C++/socket.htm.

[2] M. Allman, V. Paxson, and W. Richard Stevens. TCP congestion control. RFC 2581, Internet Engineering Task Force, April 1999.

[3] Jonathan Corbet. Circular pipes. http://lwn.net/Articles/118750/, January 2005.

[4] Jonathan Corbet. The evolution of pipe buffers. http://lwn.net/Articles/119682/, January 2005.

[5] Jonathan Corbet. Linux and TCP offload engines. http://lwn.net/Articles/148697/, August 2005.

[6] Jonathan Corbet. Large receive offload. http://lwn.net/Articles/243949/, August 2007.

[7] Gustavo Duarte. Page cache, the affair between memory and files. http://duartes.org/gustavo/blog/post/page-cache-the-affair-between-memory-and-files, February 2009.

[8] Jon Eibertzon and Sebastian Hultqvist. Acceleration of network protocol processing for system-on-chip. Master's thesis, Department of Electrical and Information Technology, Lund University, January 2006.

[9] Gustaf Engquist and Magnus Nilsson. TCP/IP offload engine for an embedded system. Master's thesis, Department of Electrical and Information Technology, Lund University, May 2004.

[10] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Updated by RFCs 2817, 5785, 6266.

[11] The Linux Foundation. GSO. http://www.linuxfoundation.org/collaborate/workgroups/networking/gso, November 2009.

[12] The Linux Foundation. NAPI. http://www.linuxfoundation.org/collaborate/workgroups/networking/napi, November 2009.

[13] The Linux Foundation. TOE. http://www.linuxfoundation.org/collaborate/workgroups/networking/toe, November 2009.

[14] Grzegorz Kulewski et al. Linus Torvalds, Diego Calleja. Linux: Explaining splice() and tee(). http://kerneltrap.org/node/6505, April 2006.

[15] Larry McVoy. Splice. http://lwn.net/2001/0125/a/splice.php3, January 2001.

[16] Aravind Menon and Willy Zwaenepoel. Optimizing TCP Receive Performance. http://www.usenix.org/event/usenix08/tech/full_papers/menon/menon_html/paper.html#lrp, April 2008.

[17] David S. Miller. How SKBs work. http://vger.kernel.org/~davem/skb_data.html, July 2005.

[18] David S. Miller. How the linux TCP output engine works. http://vger.kernel.org/~davem/tcp_output.html, August 2005.

[19] J. Postel. User Datagram Protocol. RFC 768 (Standard), August 1980.

[20] J. Postel. Internet Control Message Protocol. RFC 792 (Standard), September 1981. Updated by RFCs 950, 4884.

[21] J. Postel. Internet Protocol. RFC 791 (Standard), September 1981. Updated by RFC 1349.

[22] J. Postel. Transmission Control Protocol. RFC 793 (Standard), September 1981. Updated by RFCs 1122, 3168, 6093.

[23] H. Schulzrinne, A. Rao, and R. Lanphier. Real Time Streaming Protocol (RTSP). RFC 2326 (Proposed Standard), April 1998.

[24] Dragan Stancevic. Zero copy i: User-mode prespective. http://www.linuxjournal.com/article/6345, January 2003.

[25] W. Richard Stevens. *TCP/IP Illustrated, Volume I: The Protocols*. Addison-Wesley, Reading, MA, 1994.

[26] Linus Torvalds. Splice(). http://yarchive.net/comp/linux/splice.html.

[27] Linus Torvalds. Making pipe data structure be a circular list of pages, rather than. http://lwn.net/Articles/118760/, January 2005.

[28] Gary R. Wright and W. Richard Stevens. *TCP/IP Illustrated, Volume 2: The Implementation*. Addison-Wesley Publishing Company, Reading, MA, 1995.