TR03-17

December, 2002

Applying Network Processors to Header Compression

Siva Kollipara

Supervisor: Dr. Rajiv Gupta

Applying Network Processors to Header Compression

Siva Kollipara (siva@cs.arizona.edu)

Abstract

Network Processors (NPs) are highly specialized processors optimized to handle protocol processing, with possibly including many other services like QoS, Statistics, Compression-Decompression etc, at highest speeds possible. Many vendors like Intel, IBM etc., have released powerful silicon for this market. NPs represent the Holy Grail – the panacea of network bottlenecks. Even though the architectural design of these various NPs varies significantly, the architectural principles/goals/ideas/requirements are almost similar – to achieve network processing at highest speeds possible with low latency. The programmability and parallel nature of these processor chips make them an ideal choice when high performance and ability to quickly adapt to new network standards are the requirements.

This report briefly describes the need for header compression in the Internet today and provides an analysis of the evolution of CRTP header compression design, using the IXP2X00. This was a particularly difficult challenge due to the fact that the order of packets is important. Reordering is not admissible since the compression (decompression) of each valid packet makes changes to the context, which is used for the compression (decompression) of the next packet. We also have time constraints to consider based on bus width, bus clock speed, processor clock speed, pipelining, network data rate (keeping in mind the link layer, physical layer overhead) and packet processing. Say some 'x' bus ops/pkt and 'y' CPU ops/pkt. At this rate, for each packet, buffers must be allocated, the packet must be received, stored in DRAM, header verified, classified, evaluated as to whether it should be dropped, compressed (decompressed), context updated, previous header stripped, new compressed (decompressed) header prepended, statistic counters updated, (multi-field and destination lookups performed for destination,) enqueued for transmit, transmitted, and buffers freed. This report chronicles the issues encountered and solutions devised to achieve header compression. The necessary concepts of context/functional pipeline, critical sections, signaling and ring buffers are defined with references. The CRTP pipe stages are described.

Introduction

I plan to explore how network programs could be developed on this chip by using CRTP as a proof-of-concept application. General ideas in header compression are provided where appropriate. I assume the reader has some idea about Network Processors [PPT-SIVA] and some familiarity with IXP 2X00 [ngINP].

Overview Of Header Compression/Decompression (HCD)

As RFC 2507 puts it, header compression deals with how to compress multiple IP headers and TCP and UDP headers per hop over point-to-point links. The methods can be applied to of IPv6 base and extension headers, IPv4 headers, TCP and UDP headers, and encapsulated IPv6 and IPv4 headers. Headers of typical UDP or TCP packets can be compressed down to 4-7 octets including the 2 octet UDP or TCP checksum. This largely removes the

negative impact of large IP headers and allows efficient use of bandwidth on low and medium speed links. The compression algorithms are specifically designed to work well over links with nontrivial packet-loss rates. Several wireless and modem technologies result in such links.

There are various header compression algorithms in use today. The following are some to mention:
VJHC – Jacobson, "TCP/IP Compression for Low-Speed Serial Links", RFC 1144
IPHC – Degermark, M., Nordgren, B. and S. Pink, "Header Compression for IPv6", RFC 2507
CRTP – Casner, V. Jacobson, "Compressing IP/UDP/RTP Headers for Low-Speed Serial Links", RFC 2508.
ROHC – RObust Header Compression, RFC 3095

These differ on the efficiency, robustness of compression algorithm and also on the underlying link.

HCD Outline

Since my example application is CRTP, I provide a brief description of CRTP (other algorithms develop on similar ideas, but have better performance and/or features), followed by its algorithms:
There has been growing interest in using the Real-time Transport Protocol [RFC 1889] as one step to achieve interoperability among different implementations of network audio/video applications. However, there is also concern that the 12-byte RTP header is too large an overhead for 20-byte payloads when operating over low speed lines such as dial-up modems at 14.4 or 28.8 kb/s. RTP header compression (CRTP) [RFC 2508] was designed to reduce the header overhead of IP/UDP/RTP datagrams by compressing the three headers. Half of the bytes in the IP and TCP headers remain constant over the life of the connection. The first-order difference between packets is usually varying or constant. By maintaining both the uncompressed header and the first-order differences in a session state shared between the compressor and de-compressor, and by communicating the first-order difference (if any), the de-compressor can reconstruct the original header without any loss of information simply by adding the first-order differences to the saved uncompressed header as each compressed packet is received. The IP/UDP/RTP headers are compressed to 2-4 bytes most of the time. The RFC references provide more detailed descriptions.

CRTP Flowchart

Negotiate: { negotiate cid space, delta codecs etc } /* assume this is done beforehand */

Compress:

1. Receive Input Packet
2. Classify Flow (as 8/16 bit cid) {
   Classify packet as {
   Repair (perform context repair)
   Invalid (non ip, non udp, fragment, hdr_length>MAX, stream in neg_cache) -> sent as it is
   Doubtful (SSRC) -> update per_stream_count
   IP Compressible (udp checksum present toggle) -> CNTCP
   UDP Compressible (PT change, TS diff>MAX, X toggle) -> CUDP, update per_stream_count
   RTP Compressible (CSRC change, M toggle) -> CRTP
   Compressible (new flow (ip options/intermediate headers, ttl, tos) change) -> FH
   if (per_stream_count>LIMIT) enter stream in neg cache

```
        }
    }
    3. Control per flow {
        Prioritization, flow control, bandwidth allocation, congestion management, statistics, repair
        based on above change type of packet to send
    }
    4. Task {
        FH: initialize context, update pkt headers,
        CNTCP,CUDP,CRTP: form header, encode delta values, update context
        Enqueue
    }
    5. Transmit {
        transmit scheduler, queue manager
        dequeue, transmit data
    }
```

Decompress:

```
1. Receive Input Packet
2. Decompress {
        parse packet, update context, form new packet
        on errors signal  compressor
    }
3. Transmit {
        transmit scheduler, queue manager
        dequeue, transmit data
    }
```

Suggestions

- How to do this fast, compressed header can't be formed in place, how to deal with this?
- Read/process packet headers and data in two different data paths. How to identify which payload to compressed header?
- Can we bring in ideas from L3 switching??
- How about 2 context structures per context, now we update (secondary structure) and encode simulateously, then swap primary & secondary. Is this helpful?

CRTP Parameters:

Memory Requirements for x86:

ROM Footprint : ~22 KB

RAM Footprint (assuming 32 flows):

Compressor : ~ 5-12KB

Decompressor : ~ 3-10KB

Processing Requirements:

Full Header Compression – ~650 CPU-cycles.

Full Header De-Compression – ~1500 CPU-cycles.

Compressed Header Compression – ~555 CPU-cycles.

Compressed Header De-Compression – ~955 CPU-cycles.

Memory Requirements for ARM Thumb:

ROM Footprint : ~19 KB

RAM Footprint (assuming 32 flows):

Compressor : ~ 5-12KB

Decompressor : ~ 3-10KB

Processing Requirements:

Full Header Compression – ~250 CPU-cycles.

Full Header De-Compression – ~500 CPU-cycles.

Compressed Header Compression – ~205 CPU-cycles.

Compressed Header De-Compression – ~355 CPU-cycles.

Moving From x86 to ARM hosts, there is huge performance increase for CRTP. In general, these numbers illustrate the need to migrate network applications from x86 hosts to ARM hosts. Network processors based on ARM/XScale cores promise to be faster than ARM/Xscale/x86. Thus huge performance improvements can be expected by using NPs.

ARCHITECTURAL REQUIREMENTS

- Word alignment
- Seeking/Parsing using pointers
- (Modulo) Arithmetic addition/multiplication
- Rotates/Shifts
- Bit operations/Masking
- Memory operations
- Substitutions (delta codec)
- Hashing
- Internet Checksum
- Timer
- Cache with LRU policy

BRIEF OVERVIEW OF IXP2400/2800

IXP2X00 has the following major internal units:

- Media/Switch Fabric
- Bus system
- 8/16 Microengine(ME)s
- DRAM, SRAM (atomic read-modify-write, link-list, ring operations)
- Cryptography, Hash, Scratch Unit
- Xscale Control Processor
- PCI Unit

Each IXP2XXX Microengine has the following features:

- 8 hardware threads
- 640 words Local Memory (LM)
- { 32 GP, 16 Next Neighbor (NN), 64 Data Transfer, 2 Local Memory (LM) } Registers per thread
- 4K words Control Store
- CRC Block, CSRs, Random Unit, Timer Block,
- 16 entry, 32 bit CAM, LRU logic
- multiply, add, shift, logical, ALU

For more details refer to [npINP].

CHALLENGES AND SOLUTIONS

Several rules have to be followed for header compression. Some obstacles have to be overcome to achieve the required performance. The following sections describe some challenges and a corresponding solution. [INTEL-POS] describes some of these in more detail. Some of them are reproduced here for completeness.

Some rules of the game are:

- Packet Order: packets are exiting the compressor (decompressor) in the same order in which they entered.
- Atomic Access: for each packet, atomic access is needed to the corresponding context structures.

Some inherent obstacles are:

- Based on the type of pipeline, we need to identify critical sections (and synch sections) provide atomic access (and sequencing),
- Atomic access involves fetching data from slower SRAM/DRAM,
- Ensuring in-order delivery/processing of packets involves some kind of signaling between threads (and MEs)

Per Packet Budget

First the budget (cycles) per packet processing has to be determined. If it exceeds the inter-packet arrival time, software pipeline architecture can be deployed to achieve cycle requirements.

For example, a 133MHz processor processing 40-byte IP packets using POS (Packet over SONET) at OC48 speeds (2.4Gbps) sees a packet arrive every 21.7 clock cycles.
(Sonet has an overhead of 3.5% and Point to Point Protocol (PPP) has an overhead of at least 9 bytes per packet, resulting in a throughput of ((2.488e9 * .965) / 8) / (40 + 4 + 4 + 1) = 6.125e6 packets/s. At 133Mhz, that's 21.7 clocks/packet)

To handle a 1Gbps stream at 100 B/pkt, 64bit 50MHz bus, 200MHz 1op/cycle CPU and 1 Gbps in/out requires a budget of 15 bus ops/pkt and 160 CPU ops/pkt!!
(incoming_pkt_rate= nw_bit_rate / (bytes/packet * 8) = 1 Gbps / 800 = 1.25 M Packets/ sec
available_bus = bus_bw – ingress_bw – egress_bw = (64*50M)-1 Gbps-1Gbps = 1.2 Gbps
Max Bus ops/pkt = available_bus/bus_width / incoming_pkt_rate = 1.2G/64/1.25M = 15 ops /pkt
Max CPU ops/pkt = CPU speed * ops/cycle / incoming_pkt_rate = 200M * 1 / 1.25M = 160 ops/pkt)

Based on the experimental results provided for CRTP, this budget is not sufficient. Moreover CRTP is the simplest of those HCD algorithms. In general HCD goes along with some other protocols (see Applications section), thus this available budget is certainly not sufficient. Thus, software pipelining must be used.

Context Pipe Stages

Context pipelining and functional pipelining are two possible candidates. [ngINP] gives more details on these techniques and some examples of situations where to use context or functional or a combination of both. Briefly,

in context pipelining, the context moves along the pipeline; in functional pipelining, the function moves along the pipeline. In a context pipeline, each ME performs only one particular function. In a functional pipeline, the pipeline is split into stages each with different functionality; a ME, which supports all these different functions, implements each stage. In both as time progresses, effectively different functions are performed on the packet. But they offer different cycle budgets. If 'y' is the number of threads in a ME and 'x' is the number of stages in a functional pipeline then – while context pipeline gives 'y' times inter-packet arrival time period, functional pipeline gives 'x'*'y' times inter-packet arrival time period. Based on the pipelining strategy, the implementation of critical section differs. Using ring-buffers the advantages of both these strategies can be exploited. Now more concepts of producer-consumer concepts come into picture.

Ring Buffers

Also known as elasticity buffers, these buffers can be used, in mixed pipelines configurations, when a pipeline transition occurs. This buffer is written by the producer and consumed by the producer. Depending on the number of producers and consumers, either SRAM or NN registers can be used to implement this.

Synch Section (SS)

Since packet order is important for our applications, each thread has to write to the memory in sync. Code sections where 2 or more threads try to establish a new context, enqueue or dequeue packets; update statistics are potential synch sections. (Code portions where the order is important, the previous thread's value is important globally, but to this thread that current value is irrelevant and can proceed without reading that value, but write to that should use the latest values)

Critical Section (CS)

Also known as mutual exclusion sections, these identify regions of code where only a single thread has exclusive access to the memory. Context update, context tear down, flow control functions are some critical code sections. The rules of CSs state that only one function (only one ME or stage) modify it. Since in-order packet processing is required, and each processing begins with accessing the CS, longer delays be incurred while each thread, to access SRAM/DRAM, waits for the previous thread to exit CS. Folding is a technique where intra-ME threads can read from the local cache rather than from SRAM/DRAM. This technique is used along with CAM support. CAM search results in {miss, lru index}, {hit, index, 4bit-state}. This 4-bit state information can be used to tell other threads about the current usage status of the CAM entry (described in Design). The context can be split into static context and dynamic context so that the CS range can be reduced and functions that work with static context can still proceed.

Signaling

Intra-ME signaling is achieved by writing the local CSR register; inter-ME signaling is done by writing the CSR proxy. Critical/Synch section signaling are the traditional "passing the baton" problems.

Some other obstacles are:

- Execution Bubbles

Execution bubbles occur when a thread accesses a resource, such as memory or a coprocessor, and then has to wait significant time—say, 300 cycles—for a response. All efforts must be made to remove execution bubbles by re-arranging code or switching threads. Since we write CRTP targeted for IXP (which is a multithreaded-multiprocessor), we may have these bubbles. To optimize such code is difficult since there are inter-thread and inter-ME dependencies to resolve. We can re-arrange code, but that would deteriorate code linearity; moreover we can only do some limited amount of code re-arrangement since most of the packet processing can proceed only after the previous packet has been processed. (Can we exploit this time bubble to move payload in the other path?)

- Thread Balancing

  Thread Balancing deals with how to split the task(s) among the threads. With asymmetric balancing we can take advantage of staggered symmetry to optimize resource usage and eliminate bubbles. But now we have to develop and optimize two different code segments. On the other hand, with symmetric balancing we have to develop and optimize a single code segment.

- Thread Loading

  Thread Loading is the decision making process of thread allocation to functions.

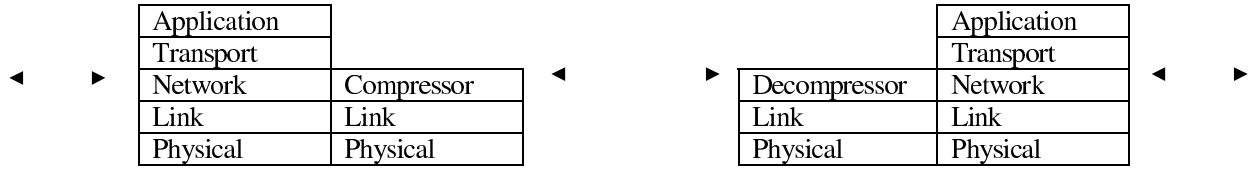
These 3 issues are to some extent handled by the target NP specific compiler. But still hand optimization may be required.

Apart from these, there are some programming challenges also:

- Not all data types are supported (this limits lots of stuff)
- Data types size/range differ from normal PC architectures.
- No stack support, different parameter passing techniques (this limits the nesting levels)
- Local Memory and Control store size. Based on CRTP values, these sizes are not sufficient!

Proof Of Concept Design

HCD goes along with other protocols (like wireless protocols, point-to-point protocols). Consider an example scenario, a mobile cell phone and a RNC. If both have compressor and decompressor, then the traffic between them can be compressed – the sender compressing before sending and the receiver decompressing it after receiving it. The general stack diagram would be:



The following tables identify typical compressor and decompressor functions.

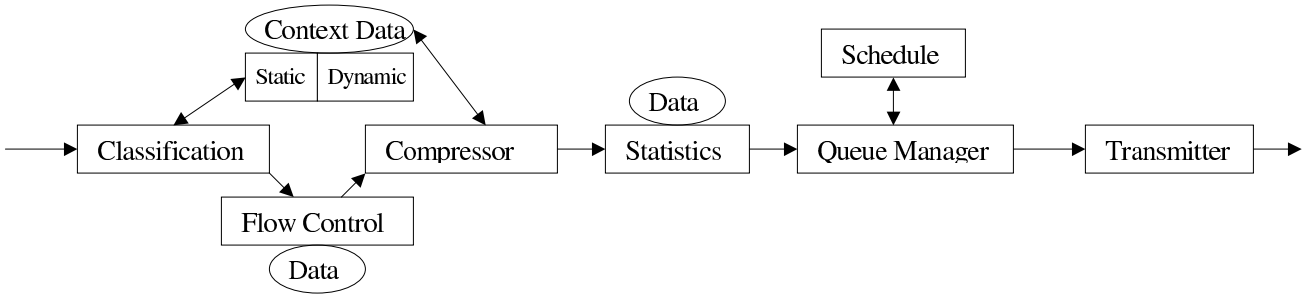
Compressor Side Functions

Function	Pipeline	Description
Classification	Functional	Classify the incoming packet based on {src_ip, dest_ip, src_port, dest_port, ttl/hl, tos/tc, rtp_ssrc} (and some more fields in the subheaders) Refer to [IPHC] for some information on classifying packets. This stage would read-access the static context (dynamic context may be). Information such as {context identifier, potential compressible types for this packet, pointers to some headers} can be provided as output.
Flow Control	Context	Based on situations like – slow start, context repair decide the compression type for this packet. The type can be tagged to the packet to be used by successive stages.
Compression	Context	This is a critical section. Based on the type of compression requested, the compressor would create a new header and update its fields and also update the dynamic context (or create a new (static) context)
Statistics	Context	Update statistics information
Scheduler	Context	Schedule the packets transmission
Queue Manager	Context	Queue per channel. Handles enqueue requests from compression/statistics stage. And dequeue requests from Scheduler. Hand the packet to the output transmitter
Transmitter	Context	Encapsulates the compressed packet with a new link/physical header and sends it through the output channel

According to CS requirements, only one function (a ME or a Pipe Stage) can modify the critical data. Thus classification is merely read-accessing the static context, whereas compression would have both read-write access on the (static and dynamic) context. Static context modification comes into play when a Full Header is sent either to establish a new context flow or to refresh the decompressor state or in response to a 'context repair' request by the decompressor.

Some puritans would argue that classification is not a part of compression. It is provided here for completeness.

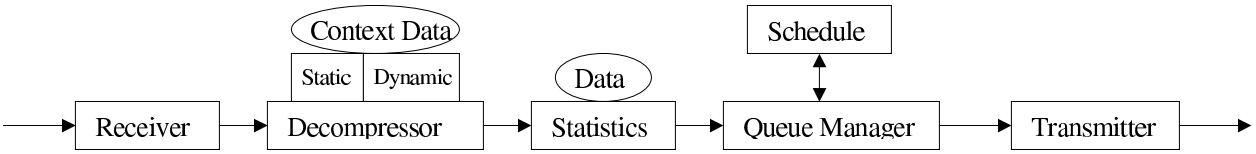
The link and physical layer in the stack diagram do the transmitter stage.



Decompressor Side Functions

Function	Pipeline	Description
Receiver	Context	Decapsulates link/physical header and sends the compressed packet to decompressor
Decompression	Context	This is a critical section. Based on the type of compressed packet as indicated by the link header, the decompressor would create a new header and update its fields and also update the dynamic context (or create a new (static) context), if errors occur prepare context repair information
Statistics	Context	Update statistics information, based on error count, time since last error queue up a context repair packet.
Scheduler	Context	Schedule the packets transmission
Queue Manager	Context	Queue per channel. Handles enqueue requests from decompression/statistics stage. And dequeue requests from Scheduler. Hand the packet to the next protocol Hand repair packet to transmitter.
Transmitter	Context	Encapsulates the packet with a new link/physical header and sends it through the output channel

The link and physical layer in the stack diagram do the receiver/transmitter stage.



Some hardware assists like Hash units, CRC unit and scratch unit can be used during these compressor/decompressor stages. An Internet checksum unit is also needed, but IXP doesn't provide this unit.

Folding

Making compression and decompression as context pipeline rather then functional pipeline has some advantages. Since according to CS rules, only one function (ME or stage) can modify data, only this context pipe stage ME will modify the context information. If the number of cycles to perform compression exceeds the 'y'*packet arrival rate (y is #threads=8), then we can switch to functional pipeline. Since a context pipe is the only ME that uses the critical data, by using CAM, context data caching in Local Memory can be exploited. Whereas with functional pipeline, multiple MEs can perform the same function (by definition not simultaneously) and thus the local data has to be evicted to external memory after every stage. Thus a thread after using the data will 'fold' to the next thread.

Algorithm to Implement Compression/Decompression using Folding

CAM search will result in {Miss, lru index} or {Hit, index, 4 bit state}. This 4 bit state information can be used to implement mutex (1 bit) or 3 bits can be used for other purposes like thread counter for number of threads interested in the same entry (max 8 threads) or for holding the current thread's (the one who has lock) number or

the first requesting thread's number (the first thread who is also interested in the same entry after the current thread). This can be used for providing flow priorities.

Incoming packets are assigned to threads in order. A brief algorithm follows:

```
START:
        /*
        do a cam lookup with the cid as tag.
        If miss, CAM will flush the lru index's data in Local memory to external memory.
        Return {flag, index} along with lock information.
        If not locked, then lock it
        */
        Flag, index = CAM_LOOKUP(cid, &lock)

        /*
        next is next modulo #threads
        so 0 → 1→ 2 → 3 → 4 → 5 → 6 → 7 → 0
        */
        if (hit, lock=0) { // this thread owns lock
                yield to/signal next thread
                modify/update context, form new header,
                unlock
                yield to/signal next thread
        } else if (hit, lock=1) { // someone else updating context, retry later
                yield to/signal next thread
                goto START
        } else if (miss, lock=0) { // this thread has to fetch data from external mem
                async_read(ext.mem)
                // while (data not in LM) { yield, sleep }
                yield to/signal next thread
                sleep;
                modify/update context, form new header
                unlock lock
                yield to/signal next thread
        }  else if (miss, lock=1) { // someone else initiated data fetch. Retry later
                yield to/signal next thread
                goto START
        }
```

Fairness to all threads is ensured.

The decompressor algorithm can be implemented in a similar fashion.

Applications

[INTEL-NP-Wireless] discusses using Intel IXP2400 Network Processors for building an all IP wireless networks. It considers using Network Processors at RNC whose functions, among others, includes Ipv6 routing, Ipv4 routing, tunneling, QoS and header compression and decompression.

[IBM-NP-Wireless] features building All-IP wireless networks by using IBM PowerNP 4GS3 for BTS/Node-B, BSC/RNC, SGSN, GGSN and MSC data-plane applications.

Conclusion

Header Compression / Decompression is a necessary requirement for wireless links. With powerful network processors this task can be integrated along with other protocol processing functions to achieve efficient use of wireless spectrum and bandwidth while maintaining link rate. Careful design decisions and coding decisions have to be made to fit in within the cycle budget and avoiding inefficiencies. Based on the node processing requirements an appropriate NP can be chosen. This report is one step forward in realizing such a goal. Somu and I plan to implement this task in the upcoming semesters.

Acknowledgements

I would like to thank Dr. Rajiv Gupta, Dr. Stephen Pink, Dr. Mikael Degermark and Somu for their inputs.

References

[VJHC] Jacobson, V., "Compressing TCP/IP Headers for Low-Speed Serial Links", RFC 1144, February 1990.

[IPHC] Degermark, M., Nordgren, B. and S. Pink, "IP Header Compression", RFC 2507, February 1999.

[CRTP] Casner, Jacobson, "Compressing IP/UDP/RTP Headers for Low-Speed Serial Links", RFC 2508, February 1999.

[PPT-SIVA] Siva, Presentation Slides on “Network Processors – An Overview”, University of Arizona.

[ngINP] Matthew Adiletta et. Al., The Next Generation of Intel IXP Network Processors” Intel Technology Journal (ITJ)

[ROHC] RObust Header Compression, RFC 3095

[INTEL-NP-Wireless] Network Processor Building Blocks for All-IP Wireless Networks. ITJ

[IBM-NP-Wireless] 2.5G/3G Wireless Networks and the Application of Network Processors.

[INTEL-POS] POS: Achieving 10Gbps Packet Processing with an IXP2800, ITJ