Fast Incremental CRC Updates for IP over ATM Networks

Florian Braun Marcel Waldvogel

Department of Computer Science
Washington University in St. Louis
<{florian,mwa}@arl.wustl.edu>

***Abstract*—In response to the increasing network speeds, many operations in IP routers and similar devices are being made more efficient. With the advances in other areas of packet processing, the verification and regeneration of cyclic redundancy check (CRC) codes of the data link layer is likely to become a bottleneck in the near future. In this paper, we present a mechanism to defer CRC verification without compromising reliability. This opens the possibility of incremental updates of the CRC. We introduce a new high-speed technique and present efficient implementations, speeding up CRC processing by a factor of 15. Although the paper and analysis focuses on IP over ATM, the scheme applies to a much wider set of network protocols.**

I. Introduction

The Internet is growing rapidly in terms of number of users and amount of bandwidth used, requiring size, speed, and network equipment to concur. Besides the transmission and switching speeds, the per-packet operations necessary for Internet Protocol (IP) packet forwarding are the current limiting factors. As transmission speeds are continually increasing, thanks to advances in optical technology, switching speeds and, to a greater extent, IP packet processing overheads have become the main bottlenecks.

Often, IP packets are encapsulated in Data Link layer frames protected by a cyclic redundancy check (CRC) code, as used when IP runs over Ethernet or ATM's AAL5. Since IP packets need to be modified at every router, the classical approach has been to check and then recreate these CRCs at every hop. At speeds achievable soon, this calculation also tends to become a bottleneck. This paper addresses improving the speed of this forwarding operation by eliminating most of the duplicate CRC calculations. We use AAL5 as our example, but the method can be generalized to upcoming multi-gigabit/s Ethernets. It applies to both the current version of the Internet Protocol, IPv4, as well as to the upcoming IPv6.

II. IP Processing

In the Internet, each packet passing through a router is subject to at least the following operations at the network layer[1]:

1. verifying the remaining packet lifetime (Time-to-Live in IPv4, Hop Count in IPv6, "TTL"),
2. updating (decrementing) the TTL/Hop Count,
3. updating the IP header checksum (IPv4 only),
4. selecting an appropriate next router or ultimate destination ("forwarding decision"),
5. forwarding the packet to this next hop.

Each of these operations had the potential to become a bottleneck. Most of these are no longer a threat:

Verifying the remaining TTL and decrementing it are simple, straightforward operations. Updating the IPv4 header checksum is necessary, since it covers the TTL field, which has just been changed. Originally, IP routers verified the header checksum and recreated a new header checksum. As wire speeds increased, the two checksum operations on $20 \ldots 64$ bytes became too expensive: an incremental update mechanism became necessary. At the fact that only the third Internet RFC [2] finally got the procedure right in all possible cases, it can be seen that this operation is non-trivial.

For a long time, the speed at which forwarding decisions could be made used to be a major limitation of router speeds. Recently, two papers [3, 4] initiated a flurry of activity, which resulted in a significant improvement of the speed of IP forwarding decisions. Switching fabrics used for forwarding the packet to the output port have also been able to increase performance at adequate rates. The emergence of purely optical switching technologies promises a further quantum leap in this area. Several methods try to avoid per-packet routing lookups in parts of the network, such as label switching techniques [5, 6]. Many routers are still required to perform full packet processing, including CRC updates, at very high speeds.

In ATM networks, IP packets are encapsulated in AAL5 frames, since the basic transmission unit only has a payload size of 48 octets. At the end of a frame, a cyclic redundancy check (CRC) code is used to guarantee the integrity of the data which is spread over several ATM cells. If you change the header of an IP packet, you invalidate the CRC and have to update this as well. This document deals with updating the CRC of AAL5 frames in a highly efficient way. Its primary intention is to use this method in router hardware such as the Field-programmable Port eXtender (FPX) [7], which is used to turn Washington University Gigabit Switch (WUGS) [8] into an Internet and Active Networks router.

III. CRC Operation

A detailed discussion of cyclic codes and reliability in error cases is given in [9]. An easier to understand introduction can be found in [10, 11]. For sake of completeness, a short overview will be given here. To put it simple, a CRC is a glorified version of the old "nines check," where a check digit is added, representing the value of the number modulo 9.

In CRCs, a message is considered to be a polynomial $M(x)$ in an unknown variable x, with the ith bit of the message having a factor of x^i. The least significant bit of the message is numbered 0, and has thus an associated factor of $x^0 = 1$. All other polynomials used are formed similarly from their binary counterparts. $G(x)$ is the predefined generator polynomial (divisor) of degree r. The polynomial division takes place in GF(2), the Galois field of size two, indicating that all operations are modulo 2.

The check value $C(x)$ is calculated as

$$C(x) = x^r M(x) \bmod G(x).^1$$

Subtracting this remainder from the dividend ($x^r M(x)$) yields a polynomial $T(x) = x^r M(x) - C(x)$, which is evenly divisible by $G(x)$. $T(x)$ is then transmitted to the receiver, which will divide the (potentially corrupted) $T'(x)$ by $G(x)$. A non-zero remainder indicates corruption of the message.

Implementation is not as hard as it sounds: The polynomial division in GF(2) can be implemented using simple operations, with addition and subtraction replaced both by exclusive-or, and multiplication or division by powers of two using left or right shifts, respectively.

Let us consider a short example. Given the divider polynomial $x^3 + 1$, which is obviously of degree 3, and given a 4-bit data stream 1101. First we construct the dividend by adding 3 zeros to the data stream, resulting in 1101000. The division in the following example is done almost as we know it from school, with subtraction replaced by XOR.

```
1 1 0 1 0 0 0 / 1 0 0 1 = 1 1 0 0
1 0 0 1
---------
  1 0 0 0
  1 0 0 1
  -------
      1 0 0
```

The remainder we get is 100. The resulting data stream, consisting of the concatenation of the original message and the CRC, will thus be 1101100.

This technique is very robust against most known error sources. Let $E(x)$ be the error during transmission, i.e., each factor in $E(x)$ indicates a flipped bit in the data stream. The remainder will be $[T(x)+E(x)] \bmod G(x)$ and the error will only be undetected if $E(x)$ is divisible by $G(x)$. If you choose a suitable polynomial, all bit errors with an odd number of flipped bits, two bit errors, burst errors with less than or equal to s bits and most with greater than s bits are detected, where s is the number of factors in $G(x)$. In many applications (including the AAL5 frames), the CRC-32 is used which uses the divider polynomial

$$\begin{aligned} G(x) \quad = \quad & x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} \\ & + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1. \end{aligned}$$

The basic CRC algorithm still has a weakness: leading zeros of a message are always ignored. The CRC becomes non-zero the first time, when a '1'-bit is processed in the message. Thus additional or lost zeros at the beginning of a message cannot be detected by the basic algorithm. Therefore the common CRC algorithms start with an initial, non-zero remainder value C_0. Additional or missing zeros do now affect the result. The CRC-32 uses 0xFFFFFFFF as an initial remainder. How does this affect our basic formula? Starting with a non-zero value is equivalent to prepending a carefully crafted header to the message, where this header gives an (intermediate) result equal to the initial value. Let $h(x)$ be the header, which has the remainder C_0, i.e., $x^r h(x) \bmod G(x) = C_0$. To prepend this header to the message, it has to be shifted by the message length, i.e., $H(M(x)) = x^m h(x)$, whereas m indicates the message length, or the degree of $M(x)$, respectively. The new improved CRC formula is now

$$C_{imp}(x) = x^r [H(M(x)) + M(x)] \bmod G(x).$$

[1] Multiplying by x^r is for simplification of the verification process only.

```
/* implicit first 1 (x^32) */
#define POLYNOMIAL 0x04C11DB7L

/* lookup table */
unsigned long crc_tab[256];

/* generate lookup table */
void gen_crc_tab(void)
{
  unsigned long i, j, crc_accum;

  for (i = 0; i < 256; i++) {
    crc_accum = i << 24;
    for (j = 0; j < 8; j++) {
      if (crc_accum & 0x80000000L)
        crc_accum =
          (crc_accum << 1) ^ POLYNOMIAL;
      else
        crc_accum = crc_accum << 1;
    }
    crc_tab[i] = crc_accum;
  }
}
```

Listing 1: Computing the update table

IV. Faster CRC Algorithms

The algorithm we used above is not very fast. In each iteration only one bit of the data stream is handled. There are several approaches for a faster calculation of the CRC.

A. Table Lookup

As you can see above the divisor is subtracted from the data stream, if the first bit of the stream is '1'. Since we use exclusive-or for the arithmetic, there are no carries and we can easily determine the next result at this point. This result is a function of the two topmost bits right now, but it can be extended to have an arbitrary number of input bits. A very suitable number is 8 [12], which gives us a table with 256 32-bit entries (for the commonly used CRC-32). The table has a reasonable size and 8-bit values can be handled very comfortable with modern computers. This approach is widely used in software CRC implementations. Listing 1 [13] computes the update table in C.

To calculate the CRC the first byte of the (remaining) message and of the current remainder are xor'ed and used as the index to the table. The table entry is exclusive-ored with the

```
/* compute CRC on data block */
unsigned long
update_crc(unsigned long crc_accum,
           const char *data, int size)
{
  unsigned long i, j;
  for (j = 0; j < size; j++) {
    i = ((crc_accum>>24) ^ *data++) & 0xFF;
    crc_accum = (crc_accum<<8) ^ crc_tab[i];
  }
  return crc_accum;
}
```

Listing 2: Updating a CRC with a message

current remainder shifted by 8 bits. This gives the new remainder or the CRC at the end, respectively. The message is also shifted by one byte. Listing 2 implements an update to the CRC with a message.

B. Hardwired Update Function

The algorithm above is well suited for software implementations. If the implementation is to be done in hardware, another approach can be used. Recall that the pre-computed table is nothing more than a function of 8 input values returning a 32-bit value. As stated above, any number of input values can be used. Instead of writing the results to a lookup table, the function can also be represented in a hardware structure of XOR gates [14]. The number of gates along the critical path determines the resulting operating speed. On the Xilinx VirtexE series, this accepts 32-bit words at a rate of up to 100 MHz, when several optimization tricks are used.

C. Better Polynomials

The main factor affecting the speed of hardware implementations is the number of inputs to the gates and the length of the critical path. The CRC polynomials are not well suited under these criteria. [15] describes an approach where the division is done by a simpler polynomial than the CRC polynomial, which we will refer to as $G_{simp}(x)$. "Simpler" means that it contains fewer terms. Obviously the result will be different, so some additional constraints must be met. If $G_{simp}(x)$ is a $P(x)$-fold multiple of the original polynomial $G(x)$, then a final division by $G(x)$ will correct the result, as $T(x) \bmod G(x) \equiv (T(x) \bmod (P(x) \times G(x))) \bmod G(x)$. [15] proposes the polynomial $G_{simp}(x) = x^{123}+x^{111}+x^{92}+x^{84}+x^{64}+x^{46}+x^{23}+1$. The powers of this term are at least 8 bits apart from each other, so each cycle, 8 bits can be updated, even though the update logic only needs 2-input XOR gates. The final division is more complicated and cannot be computed in the same time as the update cycle. But it only needs to be computed once per total message, amortizing its cost.

V. Incremental CRC Update

The methods described in section IV all work well at data rates up to a few gigabits per second. Higher data rates seem hard to achieve in the near future. We therefore propose to apply a mechanism analog to the incremental IP header checksum update algorithm described in [2]. Not only is an incremental update faster than calculating the checksum from scratch, it is no longer necessary to check the sum at each and every routing node. The low error probabilities of modern high-speed communication lines combined with the policy of only modifying and not rewriting an unchecked CRC makes sure that errors introduced anywhere in the path will be detected by the receiver.

A. Mathematics

As incremental updates of the IP header checksum are widely accepted, this raises the question of applying incremental updates to the CRC. Recall the basic CRC formula, $C(x) = x^r M(x) \bmod G(x)$. Assume that we make a change to the message, and the difference is $I(x)$. The new message $M^*(x)$ will be $M^*(x) = M(x) + I(x)$ (recall that $+$ is exclusive-or in GF(2)). The new checksum will be

$$\begin{aligned} C^*(x) &= x^r[M(x) + I(x)] \bmod G(x) \\ &= \underbrace{x^r M(x) \bmod G(x)}_{C(x)} + \underbrace{x^r I(x) \bmod G(x)}_{C_I(x)} \end{aligned}$$

Obviously it is possible to just calculate a CRC of the changes and "add" this to the CRC supplied by the message frame.

Recall that the real CRC-32 uses a non-zero initial value for the CRC, which we model as $H(M(x))$:

$$\begin{aligned} C^*_{imp}(x) &= x^r[H(M(x)) + M(x) + I(x)] \bmod G(x) \\ &= \underbrace{x^r[H(M(x)) + M(x)] \bmod G(x)}_{C_{imp}(x)} \\ &\quad + \underbrace{x^r I(x) \bmod G(x)}_{C_I(x)} \end{aligned}$$

As you can see the incremental CRC is not affected by either an additional header or an initial update value, so we can ignore the initial value for the incremental update.

So far there is only the advantage that we don't have to check the old CRC, but still the incremental update is as expensive to compute as any other CRC. Fortunately, an IP router changes only a few bytes of the packet. For decrementing the TTL field, it is only necessary to change two fields: the TTL field itself (8 bits) and the header checksum (16 bit). The message update $I(x)$ thus contains mostly zeros. Only at the fixed positions of these changed fields, the update is nonzero.

While the offset of the header fields *from the beginning of the message* is well-known and constant, the effect on the CRC depends on the number of message bits *following the modified fields.* Since ATM cells come in one-size-fits-all (48 bytes), and the position of the fields in the first cell are known, the offset from the end is also known (modulo 48).

B. Implementation

Remember that updating the IP header will result in modifications to 3 bytes. We can treat these bytes as three independent updates and combine the resulting CRC updated at the end. So we create three lookup tables for each of the updated bytes. The result will be the CRC as if it were at the end

```
/* generate an incremental
   lookup table */
void
gen_inc_tab (unsigned long* tab,
             int offs)
{
  unsigned i, j, update;

  for (i=0; i<256; i++) {
    update = i << 24;
    for (j=0; j<offs; j++) {
      update = (update << 8)
        ^ crc_tab [(update>>24)&0xFF];
    }
    tab[i] = update;
  }
}

/* update tables */
unsigned long crc_ttl_tab[256], ...;

void gen_tabs()
{
  gen_crc_tab();

  /* ttl update table */
  gen_inc_tab (crc_ttl_tab, 36);
  /* ... other tables */
}
```

Listing 3: Computing incremental update tables

of the first ATM cell. So far, we can immediately update all IP packets which are smaller than or equal to 40 bytes (recall that 8 bytes in the last cell are used for AAL5 control bytes and CRC). The resulting CRC update is again a 4 byte update of the message. So we precompute another 4 lookup tables for these bytes which give us the CRC update for the next ATM cell, imitating the effect of appending 48 bytes of zeroes to the message. Note that an IP router makes no changes to any cell other than the first and that the update message of the second and all further cells is therefore a constant zero.[2] With this approach we can update the CRC of an AAL5 frame with 3 or 4 lookups and one XOR operation with 3 or 4 inputs for each ATM cell.

Listing 3 generates a table to update the CRC if the TTL field of an IPv4 header is changed. Note that the field is 36 bytes ahead of the end of the cell. Tables for other fields can be computed in a similar way.

To get the update value for the first cell, one lookup for each changed field is required, in our example three. For the succeeding cells, four lookups are necessary to update the CRC field.

Listing 4 demonstrates the use of the update tables. Note that only one function call is necessary per cell. Also note that the parameters to `first_cell()` contain the difference between the old and the new value, i.e., *old* XOR *new*.

The resulting 7 tables at 256 32-bit words each require only 7 KB, easy to store in first-level cache of a CPU or static RAM.

[2] Generalization of our approach to changes in other cells is straightforward.

```
unsigned long
first_cell (int ttldiff,
            int checksumdiff)
{
  return crc_ttl_tab[ttldiff]
      ^ crc_hcshi_tab[checksumdiff>>8]
      ^ crc_hcslo_tab[checksumdiff&0xFF];
}

unsigned long
update_cell (unsigned long update)
{
  return update
      ^ crc_inc_tab0[ update     &0xFF]
      ^ crc_inc_tab1[(update>> 8)&0xFF]
      ^ crc_inc_tab2[(update>>16)&0xFF]
      ^ crc_inc_tab3[(update>>24)&0xFF];
}
```

Listing 4: Updating a CRC cell-by-cell

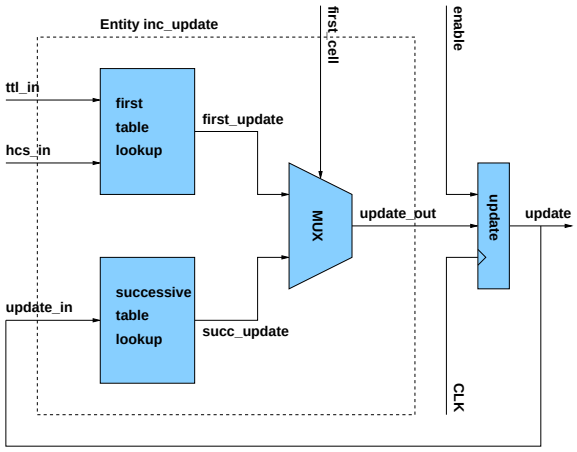

Fig. 1. *Schematic view of an incremental update entity*

C. Updating in Hardware

If the CRC update is to be done in hardware, the lookups can be done in parallel, because the tables are all independent of each other. A schematic with a register to buffer the value can be seen in Fig. 1. The synthesized code runs with 110 MHz on a Xilinx VirtexE, while utilizing only 123 CLB slices of logic, occupying about 2% of a Virtex 1000E. The resulting 7 tables use an additional $7 \times 256 \times 32bits = 56kbits$ of on-chip memory, about 15% of the Virtex 1000E's block memory. Although the operating frequency is comparable to that achieved by a full CRC (see section IV-B), it processes an entire ATM cell in one clock cycle, not just 4 bytes, resulting in a speedup of 13. An ATM cell can be handled in less than 10 ns, which corresponds to a line speed of 43 Gbps. Unfortunately, other routing jobs as IP lookup can't operate at this speed (on the same hardware), but the CRC update is now no longer the bottleneck in the system.

Is it possible to avoid using table lookup ROM? As the lookup is nothing more than a function with 8 input bits and 32 output bits, it must be possible to get the same result from a logic function as from a table. A program transforming the lookup tables into logic functions, i.e., generating VHDL

0 31		
..1..1.1.1.1..11.1..11..111...1.	14	31
.11.11111111.1.111.1.1.1..1..11.	20	
11111.1.1.111...111..11.1.1.111.	20	
1111.1.1.111...111..11.1.1.111.1	20	
11..11111.11....11.1.11..1.11...	17	
1.111.1...11..1.111......1.1..11	15	
.111.1...11..1.111......1.1..11.	14	
11..11.11..11...11..11.11.1.1111	19	
1.11111..11...1.11.1.1111.1111..	20	
.11111..11...1.11.1.1111.1111...	19	
11.111..11.11......1..1....1..1.	13	
1..111..111...11.11.1...11...111	17	
...111..1..1.1.11..111.1.11.11..	16	
..111..1..1.1.11..111.1.11.11..1	17	
.111..1..1.1.11..111.1.11.11..1.	17	
111..1..1.1.11..111.1.11.11..1..	17	
111.11......1.1.1..11.1...1.1.1.	14	
11.11......1.1.1..11.1...1.1.1..	13	
1.11......1.1.1..11.1...1.1.1...	12	
.11......1.1.1..11.1...1.1.1...1	12	
11......1.1.1..11.1...1.1.1...11	13	
1......1.1.1..11.1...1.1.1...111	13	
..1..1111111.1.111...11..11.11..	18	
.11.1.1.1.111...11........111.1.	14	
11.1.1.1.111...11........111.1..	14	
1.1.1.1.111...11........111.1...	13	
.111....1..1.1.1.1..11.1..11..11	15	
111....1..1.1.1.1..11.1..11..111	16	
11....1..1.1.1.1..11.1..11..111.	15	
1....1..1.1.1.1..11.1..11..111..	14	
....1..1.1.1.1..11.1..11..111...	13	
...1..1.1.1.1..11.1..11..111...1	14	0

TABLE I
XOR WIRING TABLE FOR UPDATING CRC FIELD

code, is straightforward. The schematic above (Fig. 1) still applies, since we only replace the ROM lookup by these logic functions. A hardwired CRC update like this uses 173 CLB slices and performs at 120 MHz. It uses 40% more CLB space, but eliminates the need for lookup memory. In addition, it is slightly faster, resulting in a theoretical line speed of 47 Gbps.

Now instead of computing the table for whole words, we can also do this for a single bit. Each change of a bit will result in a unique change of the CRC. Again the values from all bit positions have to be exclusive-or-ed. This is shown in Table I. Each column represents the effect of one single bit position on the CRC. The ones in each row indicate the bit positions, which have to be wired together to compute a single update bit of the CRC. In the second column you can see the number of XOR gates necessary for this structure. The speed stays almost the same, but the circuit is reduced to 118 CLB slices now. The reason for this improvement is that the Xilinx configurable logic blocks use lookup tables instead of wiring gates. Therefore, XOR functions are as cheap as other gates. For this application, XOR is more suitable than AND and OR gates. Only the number of logic levels limits the speed of this circuit.

VI. CONCLUSIONS

This paper describes a new method of calculating CRCs after modifying ATM AAL5 frames using incremental updates. The paper also discusses hardware implementation techniques, resulting in a speedup of almost 15, compared to efficient implementations of other current approaches. As a result, network messages can be updated at very high speeds, especially if only a few fields of the data have to be changed. This holds true for decrementing the TTL-field in IP headers and updating the IPv4 header checksum. CRC calculations are thus no longer at risk to become the next bottleneck for IP routers.

REFERENCES

[1] Fred Baker. Requirements for IP version 4 routers. Internet RFC 1812, June 1995.

[2] Computation of the Internet checksum via incremental update. Internet RFC 1624, May 1994.

[3] Mikael Degermark, Andrej Brodnik, Svante Carlsson, and Stephen Pink. Small forwarding tables for fast routing lookups. In *Proceedings of ACM SIGCOMM '97*, pages 3–14, September 1997.

[4] Marcel Waldvogel, George Varghese, Jon Turner, and Bernhard Plattner. Scalable high speed IP routing table lookups. In *Proceedings of ACM SIGCOMM '97*, pages 25–36, September 1997.

[5] Christopher Metz. Ingredients for better routing? Read the label. *IEEE Internet Computing*, 2(5):10–15, September–October 1998.

[6] Eric C. Rosen, Arun Viswanathan, and Ross Callon. Multiprotocol label switching architecture. Internet RFC 3031, January 2001.

[7] John W. Lockwood, Jon S. Turner, and David E. Taylor. Field programmable port extender (FPX) for distributed routing and queuing. In *ACM International Symposium on Field Programmable Gate Arrays (FPGA'2000)*, pages 137–144, Monterey, CA, USA, February 2000.

[8] Tom Chaney, J. Andrew Fingerhut, Margaret Flucke, and Jonathan S. Turner. Design of a gigabit ATM switch. Technical Report WU-CS-96-07, Washington University in St. Louis, 1996.

[9] W. Wesley Peterson and E. J. Weldon, Jr. *Error- correcting codes*. MIT Press, 2nd edition, 1972.

[10] Andrew S. Tanenbaum. *Computer Networks*. Prentice Hall, 3rd edition, 1996.

[11] Ross N. Williams. A painless guide to CRC error detection algorithms. ftp://ftp.rocksoft.com/papers/crc_v3.txt, 1996.

[12] Aram Perez. Byte-wise CRC calculations. *IEEE Micro*, 3(3):40–50, June 1983.

[13] Charles Michael Heard. Charles Michael Heard's CRC-32 code. http://cell-relay.indiana.edu/cell-relay/publications/software/CRC/32bitCRC.c.

[14] R. J. Glaise and X. Jacquart. Fast CRC calculation. In *Proceedings of the IEEE International Conference on Computer Design (ICCD)*, pages 602–605, Boston, MA, USA, 1993.

[15] R. J. Glaise. A two-step computation of cyclic redundancy code CRC-32 for ATM networks. *IBM Journal of Research and Development*, 41, November 1997.