Brief Contributions

Out of Order Incremental CRC Computation

Julian Satran, *Senior Member*, *IEEE*,
Dafna Sheinwald, and Ilan Shimony

Abstract—We consider a communication protocol where the sender breaks a message, comprised of an information block and a corresponding CRC, into small segments and transmits these segments separately, possibly via different routes, to the receiver. Traditionally, reversing the sender operations, the receiver first assembles all the segments that make up the message, then computes a CRC based on the information part of the message and verifies it against the arriving CRC, and, finally, delivers the information part on to the upper layer protocol (ULP). We present an incremental CRC computation whereby each arriving segment contributes its share to the message's CRC upon arrival, independently of other segment arrivals, and can thus proceed immediately to the ULP. We impose no constraint on the order of segment arrivals nor on their sizes. Yet, in its time complexity, our scheme does not exceed the traditional computation, which assembles all the segments first, and it uses only a fixed and very small amount of extra memory. Our scheme is beneficial when the ULP can process the individual segments without reading the entire message first and can revert, if needed, to the state it had prior to the processing of any segment of that message. One practical example application is the evolving protocol for Remote Direct Memory Access (RDMA) over TCP, where an overall CRC is added to a concatenation of data segments.

Index Terms—CRC, cyclic redundancy codes, out-of-order computation, RDMA.

1 Introduction

Braun and Waldvogel [2] consider packet CRC (Cyclic Redundancy Code) recreation at network routers which modify information at the packet header. The traditional approach of verifying the CRC (error detection) and then creating a new CRC for the modified packet can create a bottleneck at high network speeds. Braun and Waldvogel propose eliminating most of the duplicate calculations, namely, the CRC calculation that is based on the nonmodified part of the frame, and only computing the change in the CRC due to the actual modification done. In [1], the authors consider embedding their technique into a strictly layered modular protocol stack.

We consider another type of incremental computation that speeds up CRC calculation in a different manner. In our case, the message, comprised of a large data block and an associated CRC, is broken up by the sender into small segments which travel independently to the receiver and arrive there in any order. Traditionally, the receiver first assembles all the segments composing the message and then it computes a CRC based on the data part and verifies it against the CRC part. This scheme is simple and it allows us to avoid passing erroneous messages on to the upper layer protocol (ULP). On the other hand, it suffers from long latency, needs extra memory space for the temporary message store, and requires extra bandwidth to and from this extra storage space. Our scheme addresses these drawbacks.

We suggest computing the message CRC incrementally, accumulating individual contributions of the arriving segments. This will allow an immediate delivery of an arriving segment on to the ULP, having calculated its share in the message's CRC. Total computation time consumed by our incremental scheme does not exceed the time consumed by the traditional "in order" computation and the extra memory used is very small, of fixed size, independent of the message length or number of segments.

Our scheme is beneficial when the ULP can start processing individual segments without reading the entire message first. We assume, to this end, that, once the receiver detects an error on the message received, it can retrieve all the segments of that message it has already forwarded on to the ULP.

In Section 2, we briefly review Cyclic Redundancy Codes and, in Section 3, we present our incremental CRC computation scheme. In Section 4, we show an actual application, related to Remote Direct Memory Access (RDMA), where the segment conveys its target address that specifies where the ULP is to store it within a very large address space. Our incremental computation of the message CRC allows the receiver to forward an arriving segment to the ULP and have it stored, immediately following the segment arrival, without first reassembling all the segments.

2 Cyclic Codes—A Brief Overview

Detailed discussions of cyclic codes can be found in text books (e.g., [3] or [4]). For the sake of completeness, we present a short overview.

A codeword c of a binary (n, k) cyclic code consists of k data, or information bits, followed by $n - k = r$ parity bits, also called check bits or CRC bits, which are computed from the block of data bits. Typically, n is in the tens or hundreds or even thousands, but smaller than 2^r, and r is 16 or 32 or 64, negligible with respect to n. Viewing codewords as polynomials $c(x)$, with the ith bit of the codeword being the coefficient of x^i (the leading coefficients lead the codeword transmission and the 0th bit concludes it), the code is associated with a predetermined generator polynomial $g(x)$ of degree r that evenly divides all the codewords $c(x)$. Operations on polynomial coefficients take place in $GF(2)$, the Galois Field of size two, namely, with additions and subtractions modulo 2. We refer interchangeably to messages or codewords and the polynomials that correspond to them.

With a binary (n, k) cyclic code generated by $g(x)$, the encoding process goes as follows: Given a binary information block f of length $|f| = k$ bits, r parity bits are computed such that the message composed by concatenating f with these r bits corresponds to a polynomial evenly divisible by $g(x)$ and is thus a codeword. These r bits are the coefficients of the polynomial $C(f(x))$ computed from the polynomial $f(x)$ as follows:

$$C(f(x)) = crc(f(x)) = (x^r f(x)) \bmod g(x). \quad (1)$$

In the case where the degree of $C(f(x))$ is smaller than $r - 1$, we extend it with leading 0 coefficients and, if the information block f we started with is shorter than k, we assume it had leading 0s, which do not change its CRC.

The message m consisting of f appended with the computed CRC can be expressed as

$$m(x) = x^r f(x) - C(f(x))$$

and is thus readily seen to be evenly divisible by $g(x)$ and, hence, a codeword. m is sent to the receiver. Due to errors in transmission, the message m' arriving at the receiver might be corrupted, different from m. We assume substitution errors only (a "0" sent is received as "1" or vice versa) and, hence, m' is of the same length

• *The authors are with IBM Labs, Haifa University, Mount Carmel, Haifa 31905, Israel. E-mail: {dafna, Julian_Satran, ishimony}@il.ibm.com.*

Manuscript received 27 Sept. 2004; accepted 20 Apr. 2005; published online 15 July 2005.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-0307-0904.

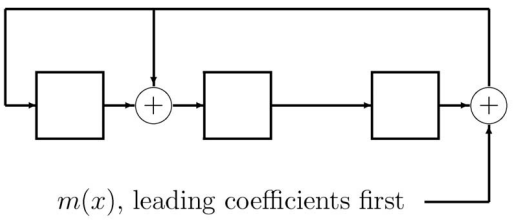

Fig. 1. A Linear Feedback Shift Register. When fed with polynomial $m(x)$, it presents $(x^3\, m(x)) \bmod (x^3 + x + 1)$, leading coefficient in the right bit register.

as m. The receiver detects transmission errors employing the maximum likelihood principle, taking into account the very small probability for distortion at any specific bit position. It tests m' for being a codeword. If it is not, a transmission error is clearly detected. If it is a codeword, then the codeword most likely sent is m' itself and the receiver assumes no transmission error occurred. Testing $m'(x)$ for being a codeword (i.e., evenly divisible by $g(x)$) can be done by computing r CRC bits from the leading $|m'| - r$ bits of m', as in (1), and comparing them against the trailing r bits of m'. A mismatch indicates that m' is not a codeword. Alternatively, $m'(x)$ is evenly divisible by $g(x)$ if and only if feeding the whole of it into the computation of (1) yields the 0 remainder polynomial.

The classic hardware implementation of CRC computation and verification uses a binary linear feedback shift register (LFSR) of length r, wired according to the coefficients of $g(x)$. It takes $|m|$ clock cycles to feed message m through the LFSR, after which the contents of the LFSR are the r CRC bits for m. Hence, the time complexity of CRC computation and verification is linear in the length of the message. See Fig. 1 for an example of an LFSR wired according to the coefficients of $x^3 + x + 1$.

3 Incremental CRC Calculation

The breaking of message m into s segments S_i of respective lengths $|S_i|$ can be expressed

$$m(x) = \sum_{i=1}^{s} x^{l_i} S_i(x) \text{ with } l_i = \sum_{j=1}^{i-1} |S_j|. \tag{2}$$

l_i is the offset of segment S_i within the message m, measured by the number of bits from the least significant bit of m to the least significant bit of S_i. Note that no constraint is imposed on the segment lengths $|S_i|$ and they need not be equal. By (1) and the linearity of the mod function (namely, $(p(x) + q(x)) \bmod g(x) = p(x) \bmod g(x) + q(x) \bmod g(x)$), we have:

$$\begin{aligned} C(m(x)) &= (x^r m(x)) \bmod g(x) = \left(x^r \sum_{i=1}^{s} x^{l_i} S_i(x) \right) \bmod g(x) \\ &= \sum_{i=1}^{s} \left\{ \left(x^r x^{l_i} S_i(x) \right) \bmod g(x) \right\} = \sum_{i=1}^{s} C(x^{l_i} S_i(x)). \end{aligned} \tag{3}$$

That is, the CRC of $m(x)$ equals the sum of s CRCs, a sum to which segment $S_i(x)$ contributes the CRC of $x^{l_i} S_i(x)$. We assume that, when the receiver processes segment S_i, it knows l_i (e.g., the offset l_i is delivered in the segment header). In Section 4, we consider a different scenario. Direct calculation of (3) thus calls for feeding the arriving segment S_i, followed by a tail of l_i 0s, into an LFSR wired according to $g(x)$. This will take $|S_i| + l_i$ clock cycles and, thus, has two drawbacks: 1) In cases where the number of segments is proportional to the message length, e.g., when their length is limited by a fixed predetermined number t, the total number of clock cycles for the processing of all the segments is $O(\frac{t+|m|}{2}\frac{|m|}{t}) = O(|m|^2)$, rather than $O(|m|)$, as in the traditional computation, and 2) the differences in processing times demanded by the segments, times which are not linear in their lengths, has a bad impact on system design.

We will now calculate, in $O(r)$ clock cycles only, the effect of the feeding of each tail of zeros. In doing so, we reduce the time it takes to process segment S_i down to $O(|S_i| + r)$ clock cycles. This accumulates to $O(|m| + s\, r)$ clock cycles for the entire message. Bearing in mind that r is usually 32 or 64 and is fixed in advance independently of the message or segment length, we have $O(|S_i|)$ clock cycles for the processing of segment S_i and $O(|m|)$ clock cycles for the processing of the entire message m, thus removing the above two drawbacks associated with feeding, bit by bit, the whole tail of 0s.

Given an LFSR of r bit registers, let $\mathbf{T}$ be the $r \times r$ binary transformation matrix in which $T_{i,j} = 1$ iff bit register j directly feeds bit register i in the LFSR. Namely, there is a wire leading from bit register j to bit register i, possibly going through adders, but not through other bit registers. For the LFSR in Fig. 1, as an example, enumerating the bit registers from left to right, the corresponding transformation matrix is

$$\mathbf{T} = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}.$$

Let the column vector $\mathbf{p}$ denote the current contents of the bit registers, with the leftmost bit register at the top of the vector. If the input to the LFSR consists of 0s only, then, after one clock cycle, the contents of the bit registers will be $\mathbf{T}\,\mathbf{p}$, where matrix operations carried out in $GF(2)$ (i.e., modulo 2). Consequently, if $\mathbf{p}$ denotes the current contents of the LFSR and the input consists of only 0s, then, after two clock cycles, the contents of the LFSR will be $\mathbf{T}\,\mathbf{T}\,\mathbf{p}$ and, after l clock cycles, the contents will be $\mathbf{T}^l\,\mathbf{p}$.

Transformation $\mathbf{T}$ is directly derived from the generator polynomial $g(x)$ and we can compute in advance the r matrices $\mathbf{T}^{2^i}$ for $i = 0, 1, \ldots, r-1$ and store them in a space of size $O(r^3)$. For a number of clock cycles l, whose binary representation is $(b_{r-1} \ldots b_1\, b_0)$, $b_i \in \{0, 1\}$, i.e., $l = \sum_{i=0}^{r-1} b_i 2^i$, we then have

$$\mathbf{T}^l = \mathbf{T}^{\sum_{i=0}^{r-1} b_i 2^i} = \mathbf{T}^{\sum_{b_i \neq 0} 2^i} = \prod_{b_i \neq 0} \mathbf{T}^{2^i}. \tag{4}$$

Upon the arrival of a segment S, whose offset within its including message is l, we first feed S into the LFSR, in $|S|$ clock cycles. The LFSR thus takes the value of $C(S(x))$. Then, denoting these LFSRs' contents by the column vector $\mathbf{p} = \mathbf{p}_0$ and the binary representation of l by $(b_{r-1} \ldots b_1\, b_0)$, we update iteratively as follows:

$$\text{for each } b_i \neq 0, \quad \mathbf{p} \leftarrow \mathbf{T}^{2^i}\,\mathbf{p}. \tag{5}$$

By (4), we end up with $\mathbf{p} = \mathbf{T}^l \mathbf{p}_0$, which would have been the LFSR's contents had it been fed with S followed by a tail of l zeros.

Altogether, we compute the contribution of the arriving segment S to the message CRC, according to (3), in $|S|$ clock cycles, followed by at most r updates.

We now turn to investigating the time it takes to make each of these r updates. For an $r \times r$ binary matrix $\mathbf{A}$, the multiplication $\mathbf{q} \leftarrow \mathbf{A}\,\mathbf{p}$, with $\mathbf{p}$ and $\mathbf{q}$ being dimension r vectors, can be carried out in one clock cycle. See, for example, Fig. 2, where the horizontal buses are of width r: The first bus carries $A_{1,1}, A_{1,2}, \ldots, A_{1,r}$, with $A_{1,1}$ participating in the first **AND** gate on the left, $A_{1,2}$ participating in the second **AND** from left, etc. The vertical wires are of width 1, with the jth line carrying p_j. Formally, there are $r \times r$ AND gates $N_{i,j}$ making up the first layer and r XOR gates X_i making up the second layer. The inputs to $N_{i,j}$ are $A_{i,j}$ and p_j,

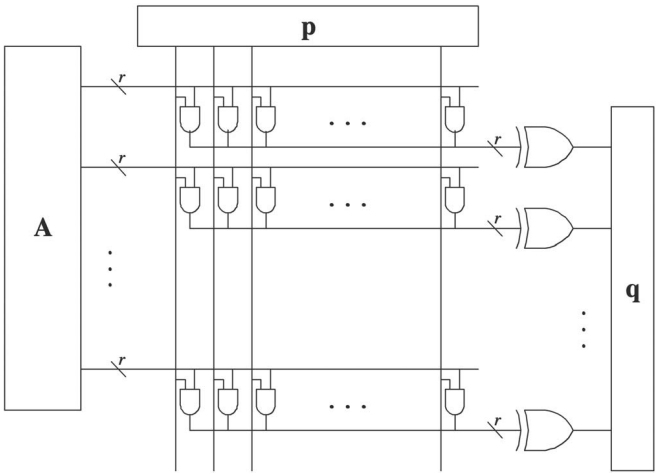

Fig. 2. Parallel calculation of $\mathbf{q} \leftarrow \mathbf{Ap}$.

while the inputs of X_i are the outputs of $N_{i,1}, N_{i,2}, \ldots, N_{i,r}$ and its output is q_i.

Using this hardware, and the r matrices $\mathbf{T}^{2^i}$, which we prepare in advance, we can perform the iterative process (5) in r clock cycles.

Altogether, by precomputing r matrices $\mathbf{T}^{2^i}$, which depend only on $g(x)$, and using a rather standard hardware, whose size depends only on r, we can compute the CRC of a message m, by processing its segments, in a total time of $|m| + rs$ clock cycles. Because r is only 32 or 64 and s cannot exceed m, the total computation time is $O(|m|)$. That is, the time complexity of our incremental, out of order, CRC computation does not exceed the complexity of the original one pass computation, the processing time of each segment is linear in the segment length, and the extra memory needed is a small function of r.

Our iterative updating (5) consumes r clock cycles. With additional hardware, whose size, too, is a function of r, the number of clock cycles for the updates needed to feed the zero tail can be reduced down to one as follows: Upon the arrival of a segment S, whose offset within its including message is l, two computing processes are launched and run concurrently. The first process feeds S into the LFSR in order to obtain $C(S(x)) = \mathbf{p}_0$ and the second obtains $\mathbf{T}^l$ by (4). The latter can be completed in r clock cycles and is thus assumed to complete first, by a simple extension of the scheme in Fig. 2. Upon completion of the two concurrent processes, $\mathbf{T}^l$ and $\mathbf{p}_0$ are fed into the hardware of Fig. 2 and, in one more clock cycle, the contribution of the arriving segment S to the message CRC is finally determined.

4 Incremental CRC Computation with Relative Segment Offsets

In this section, we consider an application, derived from the evolving RDMA (Remote Direct Memory Access) protocol, where the ULP can process each segment upon its arrival. As such, this application can benefit greatly from our scheme. The segments in this application do not convey their relative offset within their message, but the receiver can tell when all segments comprising a message have arrived. The ULP at the receiver side posts an arriving segment in a large address space, called *the reassembly address space*, which is much larger than the size of a message. In its header, each segment conveys the address where it is to be posted, called its *target_address*, and we assume that this information arrives intact. That is, when a segment arrives, the receiver cannot tell its offset within its including message, but this information becomes available when all the segments comprising the message have arrived and not earlier.

Viewed as a polynomial, the leading bit of the segment corresponds to its leading coefficient. The segment's *target_address* is a bit address, specifying where that first bit of the segment should be stored, with the rest of the bits to be stored in consecutive increasing addresses. We assume the segments do not overlap when stored in the memory.

Here is how we adjust the solution devised in the previous section to these conditions. While the segments are arriving, the receiver updates an *end_of_message* variable, which holds the largest bit address within the *reassembly address space*, where any of the arriving segments are stored. When all segments have arrived, *end_of_message* holds the bit address of the end of the message, where the least significant coefficient of the polynomial corresponding to the message is stored. The lengths l_i of the zero tails, referred to in the previous section, relate to this address. In the current problem setting, we update the lengths l_i as the segments arrive.

The receiver also maintains the r-bit variable $crc(x)$, which accumulates the contributions of the arriving segments to the message CRC. At any given moment, *crc* reflects the CRC value calculated for a message which spans from the smallest *target_address* received thus far down to the current *end_of_message*, with holes of missing data considered as zeros. Upon arrival of a new segment, *crc* is updated to reflect the new contribution as follows:

Upon arrival of the first segment S of message m, whose *target_address* is $ta(S)$ and whose length is $|S|$, the largest bit address it is thus going to occupy is $end\_of\_segment(S) = ta(S) + |S| - 1$, the receiver sets

$$crc(x) \leftarrow C(S(x)), \text{and } end\_of\_message \leftarrow end\_of\_segment(S),$$

where $C(S(x))$ is the block of r CRC bits computed for S, as in (1).

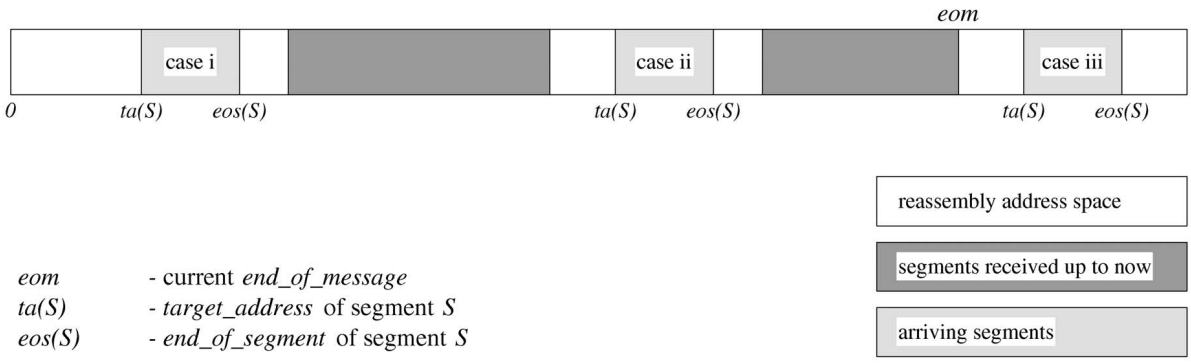

Fig. 3. Segment arrival example.

For every other segment S that arrives, the receiver updates its variables as follows:

If $ta(S) > end\_of\_message$ (Fig. 3, case iii), the message spans into larger addresses than thought thus far and we need to increase the offsets l_i of all segments that arrived so far by $d = eos(S) - end\_of\_message$. By (3), this means the updating

$$crc(x) \leftarrow (x^d\, crc(x)) \bmod g(x),$$

which, as described in Section 3, can be carried out in r clock cycles. The contribution of S itself is then calculated in $|S|$ clock cycles:

$$crc(x) \leftarrow crc(x) + C(S(x)).$$

Finally, the receiver updates

$$end\_of\_message \leftarrow end\_of\_segment(S).$$

If $ta(S) < end\_of\_message$ (Fig. 3, cases i and ii), which, by our nonoverlapping assumption, also means $end\_of\_segment(S) < end\_of\_message$, there is no reason to update the largest address of the message nor the offsets of the segments that arrived thus far. The contribution of S is added in $|S| + r$ clock cycles as in (3), with $d = end\_of\_message - end\_of\_segment(S)$ being the offset from the current *end_of_message*:

$$crc(x) \leftarrow crc(x) + \left(x^d\, C(S(x))\right) \bmod g(x).$$

Once all the segments have arrived and have been processed, their contributions reflect their correct offsets within the message (which is now clear) and, by (3), *crc* holds the message's CRC. Altogether, the total time for processing all the segments comprising a message m of length $|m|$, which is broken up into s segments, is $O(|m| + r\,s) = O(|m|)$, as with the traditional, one pass computation.

5 Conclusion

We presented a scheme by which a message CRC is computed at the receiver side by the accumulated individual contributions of its constituent segments. The segments arrive at the receiver side independently, in any order. The processing time of an arriving segment is linear in the segment length, independent of its offset within the message, and the total computation, for the entire message, is linear in the message length, as is the case with the traditional computation, which first assembles all the segments and then makes one pass on their concatenation. The extra memory and logic needed is fixed, independent of the message or the segment length. It is a function of r, the degree of the generator polynomial of the cyclic redundancy code used.

In terms of [1], our scheme does lend itself to a strictly layered modular protocol stack. The layer that processes the individual segments can forward to the upper layer, the layer that processes whole messages, along with each segment, the contribution of the segment to the message CRC. The upper layer will accumulate these contributions, thus computing the message CRC.

Our scheme can also be employed in order to parallelize the CRC computation of a given whole message by breaking the latter into segments which are then processed in parallel and, finally, their contributions are accumulated.

References

[1] F. Braun, J. Lockwood, and M. Waldvogel, "Protocol Wrappers for Layered Network Packet Processing in Reconfigurable Networks," *IEEE Micro,* vol. 22, no. 1, pp. 66-74, Jan./Feb. 2002.

[2] F. Braun and M. Waldvogel, "Fast Incremental CRC Updates for IP over ATM Networks," *Proc. 2001 IEEE Workshop High Performance Switching and Routing*, May 2001.

[3] F.J. McWilliams and N.J. Sloane, *The Theory of Error-Correcting Codes.* North Publishing Co., 1983.

[4] R.E. Blahut, *Algebraic Codes for Data Transmission.* Cambridge Univ. Press, 2003.