A Systematic Approach to Building High Performance, Software-based, CRC Generators

Michael E. Kounavis and Frank L. Berry
Intel Research and Development, Hillsboro, OR, 97124, USA

***Abstract*—A framework for designing a family of novel fast CRC generation algorithms is presented. Our algorithms can ideally read arbitrarily large amounts of data at a time, while optimizing their memory requirement to meet the constraints of specific computer architectures. In addition, our algorithms can be implemented in software using commodity processors instead of specialized parallel circuits. We use this framework to design two efficient algorithms that run in the popular Intel IA32 processor architecture. First, a 'slicing-by-4' algorithm doubles the performance of existing software-based, table-driven CRC implementations based on the Sarwate [12] algorithm while using a 4K cache footprint. Second, a 'slicing-by-8' algorithm triples the performance of existing software-based CRC implementations while using an 8K cache footprint. Whereas well-known software- based CRC implementations compute the current CRC value from a bit-stream reading 8 bits at a time, our algorithms read 32 and 64 bits at a time respectively[1].**

I. INTRODUCTION

CYCLIC redundancy codes (CRC) are used for detecting the corruption of digital content during its production, transmission, processing or storage. CRC algorithms treat each bit stream as a binary polynomial and calculate the remainder from the division of the stream with a standard generator polynomial. The binary words corresponding to the remainder are transmitted together with the bit stream. At the receiver side CRC algorithms verify that the correct remainder has been received. Long division is performed using modulo-2 arithmetic.

In this paper we investigate the implementation of CRC generation algorithms in software. The reason why software-based CRC generation is important is because many commercial host, network and server chipsets which are in use today do not include specialized CRC generation circuits. In addition a number of recently proposed Internet protocols (e.g., datacenter protocols such as MPA[4] or iSCSI[13]) require that data integrity checks are performed above the transport layer using CRC at very high speeds (e.g., 10 Gbps).

To accelerate the CRC generation process, a number of software-based algorithms have been proposed in the past [6-12, 15, 16]. Among these algorithms the most commonly used today is the algorithm proposed by Dilip V. Sarwate [12]. The Sarwate algorithm reads 8 bits at a time from a stream and calculates the stream's CRC value by performing lookups on a table of 256 32-bit entries. The Sarwate algorithm was designed at a time when most computer architectures allowed XOR operations between 8 bit quantities. Since then, computer architecture technology has progressed to the point where arithmetic operations can be performed efficiently between 32 or 64 bit quantities. In addition modern computer architectures comprise large on-chip cache memory units which can be accessed in a few clock cycle time and support mechanisms for preventing pollution.

In this paper we argue that recent advances in computer architecture technology and the need for efficient CRC generation above the transport layer at very high speeds call for re-examination of the mathematical principles behind software-based CRC generation. Most existing CRC generation algorithms are based on the assumption that the amount of bits read at a time from a stream should be smaller than the degree of the generator polynomial. In this paper we relax this assumption and propose algorithms that can ideally read arbitrarily large amounts of data at a time, while optimizing their memory requirement to meet the constraints of specific computer architectures.

We use our framework to design two efficient algorithms that run in the popular Intel IA32 processor architecture. First, we propose the design of a novel 'slicing-by-4' algorithm which doubles the performance of existing CRC implementations based on the Sarwate [12] algorithm. The 'slicing-by-4' algorithm uses a 4K cache footprint. Second, we propose the design of a 'slicing-by-8' algorithm which triples the performance of existing CRC implementations, using an 8K cache footprint. Whereas well-known table-driven CRC implementations compute the current CRC value from a bit-stream reading 8 bits at a time, our algorithms read 32 and 64 bits at a time, respectively. To accelerate the long division process, we apply the technique of parallel table lookups first used in [3, 8] to the generation of CRC values over long bit streams. Our algorithms avoid the memory explosion problem associated with creating a table of 2^{32} and 2^{64} entries by expressing the current remainder from a long division step as well as the next data bits read from the bit stream as sums of smaller terms. In this way our algorithms can compute the next remainder by performing parallel lookups into smaller tables using as indexes the slices from the previous remainder and data bits.

The paper is structured as follows: In section II we present related work. In Section III we provide an overview of the CRC generation process. In Section IV we present our framework. In Section V we evaluate our algorithms whereas in Section VI we provide some concluding remarks.

[1] This is an extended version of a paper that appeared at the 10th IEEE International Symposium on Computers and Communications (ISCC 2005) in Cartagena, Spain, June, 2005.

II. RELATED WORK

Efficient implementation of the CRC generation process has been the subject of substantial amount of research [1,3,5,6-12,14-16]. Software-based CRC generation has been investigated in [6-12, 15, 16]. Among these algorithms the tea-leaf reader algorithm introduced by Griffiths and Stones [7] supports the generation of 'CRC32' codes using five 256-byte table lookups, five XOR and four shift operations for each byte of a bit stream. By CRC32 we mean a CRC generation algorithm where the degree of the generator polynomial is 32. Sarwate [12] optimized the teal leaf reader algorithm reducing the cost of CRC32 generation to a single table lookup, two XOR operations, a shift and an AND operation per byte. The algorithm proposed by Sarwate uses a single table of 256 32-bit entries. Feldmeier [6] motivated by the fact that table-driven solutions are subject to cache pollution presented a software technique that avoids the use of lookup tables. In our work we do use lookup tables because modern computer architectures support large cache units and mechanisms for preventing the eviction of table entries from these cache units. Our algorithms are distinguished from [4, 6-12, 15, 16] by the fact they can ideally read large amounts of data at a time.

The concept of parallel table lookups which we use in our framework also appears in early CRC5 implementations [8] and in the work done by Braun and Waldvogel [3] on performing incremental CRC updates for IP over ATM networks. The CRC5 implementations reported in [8] have been used for validating the header fields of ATM packets and cannot be easily used in long bit streams. The reason why is because these CRC5 implementations associate each slice of a stream with a separate table. As a result long bit streams would need as many tables as the slices constituting each stream. On the other hand, if the contribution of each slice to the final CRC value is computed using the square and multiply technique as in the work by Doering and Waldvogel [5], the processing cost may be too high in software. Our work is distinguished from [3, 8] in that our algorithms reuse the same lookup tables in each iteration, thus keeping the memory requirement of CRC generation at reasonable level.

Our algorithms bear some resemblance with a recent scheme published by Joshi, Dubey and Kaplan [9]. Like our algorithms the Joshi-Dubey-Kaplan scheme calculates the remainders from multiple slices of a stream in parallel. The Joshi-Dubey-Kaplan scheme has been designed to take advantage of the 128-bit instruction set extensions to IBM's PowerPC architecture [9]. In our contrast our algorithms do not make any assumptions about the instruction set used. Moreover, the Joshi-Dubey-Kaplan scheme uses a single table and hence requires more complex calculations for finding the total remainder from the slices of streams than the algorithms of our framework, as explained in Section V.

Finally, references [1, 14] describe techniques for designing a family of hardware-based CRC generators. Reference [14] describes CRC generators which can perform the long division on a bit-by-bit basis using parallel circuits. The circuits of reference [14] split an input bit stream into multiple constituent bit streams. The original stream is reconstructed by interleaving the bits of the constituent bit streams. Reference [14] describes how circuits can be designed for processing these bit streams in parallel. In this paper we also describe a systematic methodology for building CRC generators, but our focus is on software implementations.

III. THE CRC GENERATION PROCESS

A. Description

CRC algorithms augment bit streams with functions of the content of the streams. In this way it is easier for CRC algorithms to detect errors. To avoid self-failures the functions used by CRC algorithms need to be as 'close' to 1:1 as possible. CRC algorithms treat each bit stream as a binary polynomial $B(x)$ and calculate the remainder $R(x)$ from the division of $B(x)$ with a standard 'generator' polynomial $G(x)$. The binary words corresponding to $R(x)$ are transmitted together with the bit stream associated with $B(x)$. The length of $R(x)$ in bits is equal to the length of $G(x)$ minus one. At the receiver side CRC algorithms verify that $R(x)$ is the correct remainder. Long division is performed using modulo-2 arithmetic. Additions and subtractions in module-2 arithmetic are 'carry-less'. In this way additions and subtractions are equal to the exclusive OR (XOR) logical operation. Table 1 shows how additions and subtractions are performed in modulo-2 arithmetic.

0+0 = 0-0 = 0
0+1 = 0-1 = 1
1+0 = 1-0 = 1
1+1 = 1-1 = 0

Table 1: Modulo-2 Arithmetic

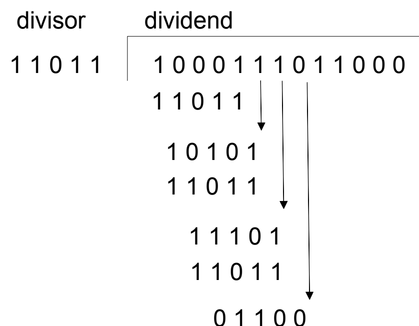


Figure 1: Long Division Using Modulo-2 Arithmetic

Figure 1 shows a long division example. In the example, the divisor is equal to '11011' whereas the dividend is equal to '1000111011000'. The long division process begins by placing the 5 bits of the divisor below the 5 most significant bits of the dividend. The next step in the long division process is to find how many times the divisor '11011' 'goes' into the 5 most significant bits of the dividend '10001'. In ordinary arithmetic 11011 goes zero times into 10001 because the second number is smaller than the first. In modulo-2 arithmetic, however, the number 11011 goes exactly one time into 10001. To decide how many times a binary number goes

into another in modulo-2 arithmetic, a check is being made on the most significant bits of the two numbers. If both are equal to '1' and the numbers have the same length, then the first number goes exactly one time into the second number, otherwise zero times. Next, the divisor 11011 is subtracted from the most significant bits of the dividend 10001 by performing an XOR logical operation. The next bit of the dividend, which is '1', is then marked and appended to the remainder '1010'. The process is repeated until all the bits of the dividend are marked. The remainder that results from such long division process is often called CRC or CRC 'checksum' (although CRC is not literally a checksum).

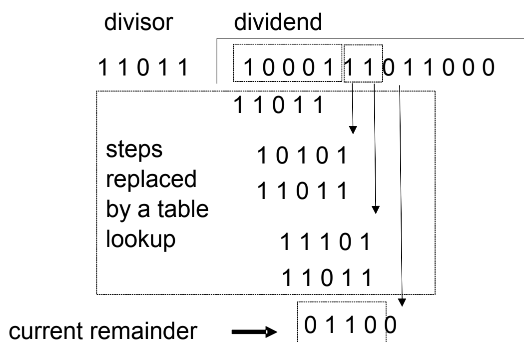


Figure 2: Accelerating the Long Division Using Table Lookups

B. *Software Approaches*

The purpose of software-based CRC generation algorithms is to perform the long division quicker than the bit-by-bit marking process described above. The long division process is a compute-intensive operation because it requires in the worst case one shift operation and one XOR logical operation for every bit of a bit stream. One commonly used technique for accelerating the long division process is to pre-compute the current remainder that results from a group of bits and place the result in a table. Before the beginning of the long division process all possible remainders which result from groups of bits are pre-computed and placed into a lookup table. In this way, several long division steps can be replaced by a single table lookup step.

The main idea behind this technique is shown in Figure 2. In the example of Figure 2, the remainder '0110', which is formed in the third step of the long division process is a function of the five most significant bits of the dividend '10001' and the next two bits '11'. Since these bits are known, the remainder 0110 can be calculated in advance. As a result, 3 long division steps can be replaced by a single table lookup. Additional table lookups can further replace subsequent long division steps. To avoid using large tables, table-driven CRC acceleration algorithms typically read no more than 8 bits at a time. 8-bit strides result in moderate CRC generation speeds using commercial general purpose processors. For example, we measured that the performance of the popular Sarwate [12] algorithm is 7 clock cycles per byte when running on an Intel 'Pentium® M' processor.

C. *The Sarwate Algorithm*

In what follows we describe the algorithm proposed by Dilip V. Sarwate, which is one of the fastest software-based, table-driven CRC generation algorithms used today. The Sarwate algorithm is shown in Figure 3. The length of the CRC value generated by the algorithm of Figure 3 is 32 bits. The Sarwate algorithm is more complicated than the straightforward lookup process of Figure 2 because the amount of bits read at a time (8) is smaller than the degree of the generator polynomial.

Initially, the CRC value is set to a given number (i.e., `INIT_VALUE` in Figure 3) which depends on the standard implemented (e.g., this number is 0xFFFFFFFF for CRC32c). For every byte of an input stream the algorithm performs the following steps: First, the algorithm performs an XOR operation between the least significant byte of the current CRC value and the byte from the stream which is read. The 8-bit number which is produced by this XOR operation is used as an index for accessing a 256 entry table. The value returned from the table lookup is then XOR-ed with the 24 most significant bits of the CRC value, shifted by 8 bit positions to the right. The result from this last XOR operation is the CRC value used in the next iteration of the algorithm's main loop. The iteration stops when all bits of the input stream have been taken into account. The bits of the input stream are considered to be reflected inside their respective bytes in the description of Figure 3.

```
crc = INIT_VALUE;
while(p_buf < p_end )
    crc = table[(crc ^ *p_buf++) & 0x000000FF] ^ (crc >> 8);
return crc^FINAL_VALUE;
```

Figure 3: The Sarwate Algorithm

By performing an XOR operation between a new byte from an input stream and the least significant byte of the current CRC value, and by performing a table lookup, the Sarwate algorithm determines how the current CRC value is modified when a new byte from an input stream is taken into account. The lookup table used by the Sarwate algorithm stores the remainders from the division of all possible 8-bit numbers shifted by 32 bits to the left with the generator polynomial. Detailed justification and proof of correctness of the Sarwate algorithm is beyond the scope of this paper. The reader can learn more about the Sarwate algorithm in [12] and [16].

D. *Toward New Schemes*

The main disadvantage of the Sarwate algorithm and other existing table-driven CRC algorithms is their memory requirement when reading a large number of bits at a time. For example, to achieve acceleration by reading 32 bits at a time, table-driven algorithms require to store pre-computed remainders in a table of 2^{32} = 4G entries. In this paper we suggest that the current remainder which is formed after the execution of a group of long division steps as well as the next data bits read from a stream can be expressed as sums of smaller terms. In this way, new algorithms can be designed that compute the next remainder by performing parallel lookups into smaller tables using as indexes the slices produced in the previous iteration. Our approach results in algorithms that use reasonable cache footprints (i.e., 4K and 8K bytes) as opposed of 16G bytes while accelerating the speed of CRC generation by a factor of 2-4, over the Sarwate

algorithm. Our approach is generic and can be used in long division strides of different sizes.

The main idea behind our approach is shown in Figure 4. To calculate the remainder from the division of '1000111' with '11011' it is suffice to split the dividend 1000111 into three 'slices', i.e., '10', '001' and '11'. The binary number 1000111 can be written as the sum of three terms. The first term is equal to the slice '10' shifted by 5 bit positions to the left. The second term is equal to the slice '001' shifted by 2 bit positions to the left. Finally the third slice is equal to the slice '11'. The remainder from the division of 1000111 with 11011 can be found as the result of an XOR operation between the remainders returned from the division of the three constituent terms '1000000', '00100' and '11' of '1000111' with the divisor '11011'. These remainders can be computed in advance.

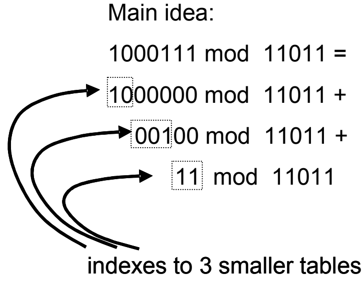


Figure 4: Remainder Slicing

IV. Building High Performance CRC Generators

A. Algorithmic Framework

In what follows we describe the algorithms of our framework. The design of our algorithms is based on two principles associated with modulo-2 arithmetic. The first principle called 'bit slicing' principle is the one discussed in the previous section and suggests that if a binary number is sliced into two or more constituent terms the CRC value associated with the binary number can be calculated as a function of the CRC values of its constituent terms. The other principle called 'bit replacement' principle suggests that amounts of bits from bit streams can be replaced by potentially much smaller in length binary numbers producing the same CRC values. The first step of our algorithms differs from all subsequent steps.

1) First Step

Let P be the initial p most significant (i.e., initially transmitted) bits of an input bit stream B. Let $l > p$ be the length of B in bits. Let also g, $g < l$ be the length of the generator polynomial $G(x)$ used in the generation of the CRC value. We consider that the l-g+1 most significant bits of the input stream B are the information bits which are being encoded, whereas the g-1 least significant bits of B are equal to zero as required by typical CRC generation algorithms. For the binary numbers P and B we write:

$$P = [b_1 \ b_2 \ \ldots \ b_p] \ , \ B = [b_1 \ b_2 \ \ldots \ b_l] \qquad (1)$$

where b_1 is the most significant bit of P and B.

The binary number P is sliced into m slices, which we symbolize as $P_1, P_2, \ldots, P_m$, of lengths $p_1, p_2, \ldots, p_m$ such that $P = [P_1{:}P_2{:} \ldots{:}P_m]$ and $p = \sum_i p_i$ for every $i \in [1, m]$. As mentioned before, the binary number P is sliced in order for our algorithms to be able to read potentially large amount of data without having to access a lookup table of 2^p entries.

For each of the slices, a table lookup is performed. Each lookup takes place using a separate table. For the table lookups which take place during the first step, m different tables $T_1, T_2, \ldots, T_m$ are used of sizes equal to $2^{p_1}, 2^{p_2}, \ldots, 2^{p_m}$ entries respectively. Each table T_i contains the remainders from the long division of all possible values of slice P_i shifted by an offset o_i. The divisor used is the generator polynomial. The offset o_i used in the calculation of the entries of table T_i is given by Eq.2 below:

$$o_i = \sum_{j=i+1}^{m} p_j \qquad (2)$$

Let $R_1^{(1)}, R_2^{(1)}, \ldots, R_m^{(1)}$ be the values returned from the table lookups during the first step.

$$R_i^{(1)} = P_i \cdot 2^{o_i} \mod G \qquad (3)$$

We define:

$$R^{(1)} = \bigoplus_{i=1}^{m} R_i^{(1)} \qquad (4)$$

where by '$\oplus$' we mean the XOR logical operation.
Let also:

$$S^{(1)} = [R^{(1)} : Q^{(1)}] = R^{(1)} \cdot 2^q \oplus Q^{(1)} \qquad (5)$$

where $Q^{(1)}$ is the set of the next q bits of the bit stream, which are positioned after the initial p bits:

$$Q^{(1)} = [b_{p+1} \ b_{p+2} \ldots \ b_{p+q}] \qquad (6)$$

The value q, $q>p$ is the amount of bits which are read during all subsequent steps of the algorithms of our framework.

The first step of the algorithms of our framework ends with the derivation of the binary number $S^{(1)}$. In each subsequent step k the algorithms operate on a binary number $S^{(k-1)}$ produced during the previous step k-1.

2) Step k

In the beginning of step k, the binary number $S^{(k-1)}$ produced at step k-1 is sliced into n slices, which we symbolize as $S_1^{(k-1)}, S_2^{(k-1)}, \ldots, S_n^{(k-1)}$, of slice lengths $s_1, s_2, \ldots, s_n$ such that $S^{(k-1)} = [S_1^{(k-1)} : S_2^{(k-1)} : \ldots : S_n^{(k-1)}]$ and $s = \sum_i s_i$ for every $i \in [1, n]$. The length of the binary number $S^{(k-1)}$ is s bits and is the same for every subsequent step k of the algorithmic framework.

For each of the slices a table lookup is performed. Each lookup takes place using a separate table. Each step k, k>1 uses the same set of tables in order for the space

requirement of the algorithmic framework to be reduced. The tables used in each step k, k>1 of the algorithmic framework are not necessarily the same as the tables used in the first step. The reason why our algorithmic framework distinguishes between the first and all subsequent steps is because the length of the input stream l may not be a multiple of the number of bits which are read at a time q. During the first step the number of bits read p may be different than q and hence a different set of tables may be needed.

We use n tables $T_1^{'}$, $T_2^{'}$,..., $T_n^{'}$ of sizes equal to 2^{s_1}, 2^{s_2},..., 2^{s_n} entries respectively. Each table $T_i^{'}$ contains the remainders from the long division of all possible values of slice $S_i^{(k-1)}$ shifted by an offset f_i. The divisor used is the generator polynomial. The offset f_i used in the calculation of the entries of the table $T_i^{'}$ is given by Eq. 7 below:

$$f_i = \sum_{j=i+1}^{n} s_j \qquad (7)$$

Let $R_1^{(k)}$, $R_2^{(k)}$,..., $R_n^{(k)}$ be the values returned from the table lookups during step k.

$$R_i^{(k)} = S_i^{(k-1)} \cdot 2^{f_i} \mod G \qquad (8)$$

The values returned from the table lookups are added in modulo-2 arithmetic (i.e., XOR-ed) producing a new binary number $R^{(k)}$:

$$R^{(k)} = \bigoplus_{i=1}^{n} R_i^{(k)} \qquad (9)$$

Subsequently a new number $S^{(k)}$ is calculated from $R^{(k)}$ and the next q bits of the bit stream:

$$S^{(k)} = [R^{(k)} : Q^{(k)}] = R^{(k)} \cdot 2^q \oplus Q^{(k)} \qquad (10)$$

where $Q^{(k)}$ is the set of the next q bits of the bit stream which are positioned after the initial $p + (k\text{-}1)\cdot q$ bits:

$$Q^{(k)} = [b_{p+(k-1)\cdot q+1} \; b_{p+(k-1)\cdot q+2} \ldots \; b_{p+k\cdot q}] \qquad (11)$$

The step k of our algorithmic framework ends with the derivation of the binary number $S^{(k)}$. In subsequent iterations of the algorithmic framework the same procedure of bit number slicing and parallel table lookups is repeated until all the bits of a bit stream are taken into account. The total number of steps N which are required for the calculation of a CRC value, assuming that $p \neq q$, is equal to:

$$N = \left\lfloor \frac{l}{q} \right\rfloor + 1 \qquad (12)$$

In Section IV-B we prove that the value $R^{(N)}$ produced in the last step of this algorithmic framework is the remainder from the division of the input stream B with the generator polynomial using modulo-2 arithmetic. In other words, $R^{(N)}$ is the desired CRC value. In the last step of this algorithmic framework no binary number $S^{(N)}$ needs to be derived since all bits of an input stream have been taken into account. The last step ends with the calculation of the binary number $R^{(N)}$ using Eq. 9 for $k = N$.

B. *Correctness*

To prove the correctness of our algorithmic framework we need to show that the value $R^{(N)}$ which is produced in the last step of our framework is indeed the remainder from the division of the input stream B with the generator polynomial using modulo-2 arithmetic. This can be shown using the following theorem:

Theorem 1: The value $R^{(k)}$ which is produced at step k of the algorithmic framework is equal to the remainder from the division of the most significant $p + (k\text{-}1)\cdot q$ bits of an input bit stream with the generator polynomial:

$$R^{(k)} = P^{(k)} \mod G \qquad (13)$$

where 'mod' is the remainder operator in modulo-2 arithmetic, G is the binary number corresponding to the generator polynomial $G(x)$ and:

$$P^{(k)} = [b_1 \; b_2 \; \ldots \; b_{p+(k-1)\cdot q}] \qquad (14)$$

One can see that if Eq. 13 is true then the algorithmic framework of Section IV-A does return the correct CRC value since $P^{(N)} = B$. To prove Theorem 1 we first show the correctness of two useful lemmas:

Lemma 1 (bit replacement principle): Let's assume that U_1 and U_2 are two binary numbers of lengths u_1 and u_2 in bits respectively. Then the following is true:

$$[U_1 : U_2] \mod G = [R_1 : U_2] \mod G \qquad (15)$$

where:

$$R_1 = U_1 \mod G \qquad (16)$$

Lemma 1 tells us that if we replace a number of consecutive most significant bits of a bit stream with an appropriately selected binary number we can still get the correct CRC value after the division with the generator polynomial. More specifically, the u_1 most significant bits of the binary number $[U_1 : U_2]$ in Eq. 15 can be replaced by the remainder R_1 from the division of U_1 with G.

Lemma 1 also indicates that arbitrarily large amounts of bits from bit streams can be replaced by potentially much smaller in length binary numbers when deriving the CRC value. It is this bit replacement principle expressed in Lemma 1 which we take advantage of in the design of our algorithmic framework in order to read arbitrarily large amount of data at a time.

Proof of Lemma 1: Lemma 1 can be directly proven as shown below. In the equations below the symbol Q_1 represents the quotient associated with the division of number U_1 with G.

$$[U_1 : U_2] \bmod G = (U_1 \cdot 2^{u_2} \oplus U_2) \bmod G$$
$$= (Q_1 \cdot G \cdot 2^{u_2} \oplus R_1 \cdot 2^{u_2} \oplus U_2) \bmod G$$
$$= (Q_1 \cdot G \cdot 2^{u_2} \bmod G) \oplus ([R_1 : U_2] \bmod G)$$
$$= [R_1 : U_2] \bmod G$$

where in the third step of the proof shown above we used the distributive property of the mod function in module-2 arithmetic:

$$(a \oplus b) \bmod G = (a \bmod G) \oplus (b \bmod G) \qquad (17)$$

Apart from the bit replacement principle our framework takes advantage of a 'bit slicing' principle expressed in Lemma 2 below.

Lemma 2 (bit slicing principle): Let U_1, U_2, …, U_n are n binary numbers of lengths u_1, u_2, …, u_n respectively, where $n > 1$. Then the following is true:

$$[U_1 : U_2 : \ ...: U_n] \bmod G = \bigoplus_{i=1}^{n} R_i \qquad (18)$$

where $R_i = U_i \cdot 2^{o_i} \bmod G$ and $o_i = \sum_{j=i+1}^{n} u_j$

Proof of Lemma 2: Lemma 2 can also be proven in a similar manner as Lemma 1:

$$[U_1 : U_2 : \ ...: U_n] \bmod G = (\bigoplus_{i=1}^{n} U_i \cdot 2^{u_{i+1}+u_{i+2}+...+u_n}) \bmod G$$
$$= (\bigoplus_{i=1}^{n} U_i \cdot 2^{o_i}) \bmod G = \bigoplus_{i=1}^{n} R_i$$

Lemma 2 tells us that if we slice a binary number into two or more component terms the CRC value associated with the binary number can be calculated as a function of the CRC values of its component terms. The bit slicing principle is applied in our framework in order to reduce the space requirement of CRC generation algorithms.

Now that we have stated and proved lemmas 1 and 2 we can prove Theorem 1.

Proof of Theorem 1: We prove theorem 1 by induction. Fist we show that Eq. 13 holds for $k = 1$. Then we show that if Eq. 13 holds for some value $k = k^*$, it also holds for $k = k^*+1$, where $k^* \leq N\text{-}1$.

For $k = 1$ Eq. 13 can be shown to be true using Lemma 2.

$$R^{(1)} = \bigoplus_{i=1}^{m} R_i^{(1)} = \bigoplus_{i=1}^{m} (P_i \cdot 2^{o_i} \bmod G)$$
$$= \text{(by lemma 2) } [P_1 : P_2 : ... : P_m] \bmod G = P^{(1)} \bmod G$$

where o_i is given by Eq. 2.

Next, assuming that Eq. 13 holds for $k = k^*$, we can show that Eq. 13 holds for $k = k^*+1$ using both Lemmas 1 and 2:

$$R^{(k^*+1)} = \bigoplus_{i=1}^{n} R_i^{(k^*+1)} = \bigoplus_{i=1}^{n} (S_i^{(k^*)} \cdot 2^{f_i} \bmod G)$$
$$= \text{(by lemma 2) } [S_1^{(k^*)} : S_2^{(k^*)} : ... : S_n^{(k^*)}] \bmod G$$
$$= S^{(k^*)} \bmod G = [R^{(k^*)} : Q^{(k^*)}] \bmod G$$
$$= \text{(by assumption and lemma 1) } [P^{(k^*)} : Q^{(k^*)}] \bmod G$$
$$= P^{(k^*+1)} \bmod G$$

where f_i is given by Eq. 7.

C. Space and Time Requirements

Our algorithmic framework requires N steps to execute, where N is given by Eq. 12. In the first step, m slices are created and m table lookups are performed. The creation of each slice requires in the worst case one shift operation and one AND logical operation. In addition, m-1 XOR operations are required for the derivation of $R^{(1)}$. The total number of operations required for the execution of the first step of our framework including, shift, AND, XOR and table lookup operations is:

$$O^{(1)} = 4 \cdot m - 1 \qquad (19)$$

Since table lookups take place in parallel, one can count all lookups as a single operation. In this case, the total number of operations executed during the first step is:

$$O_h^{(1)} = 3 \cdot m \qquad (20)$$

Each subsequent step k of our framework requires the creation of n slices and the execution of n parallel table lookups. Thus, the total number of operations required by step k is:

$$O^{(k)} = 4 \cdot n - 1 \qquad (21)$$

Counting all table lookups as a single operation the total number of operations becomes:

$$O_h^{(k)} = 3 \cdot n \qquad (22)$$

From the values of $O^{(1)}$ and $O^{(k)}$ we can calculate the total number of operations required for the execution of our algorithmic framework.

$$O = \sum_{i=1}^{N} O^{(i)} = (N-1) \cdot (4 \cdot n - 1) + (4 \cdot m - 1) \qquad (23)$$

The total number of operations, counting all table lookups as one is:

$$O_h = \sum_{i=1}^{N} O_h^{(i)} = 3 \cdot n \cdot (N-1) + 3 \cdot m \qquad (24)$$

The space required for storing the tables used by the first step of our algorithmic framework expressed as number of table entries is:

$$E^{(1)} = \sum_{i=1}^{m} 2^{p_i} \quad (25)$$

Similarly, the space required for storing the tables used by every step k, $k > 1$of our algorithmic framework is:

$$E^{(k)} = \sum_{i=1}^{n} 2^{s_i} \quad (26)$$

All steps k, k>1 use the same tables. Hence, the total space required for the execution of our algorithmic framework is:

$$E = E^{(1)} + E^{(k)} = \sum_{i=1}^{m} 2^{p_i} + \sum_{i=1}^{n} 2^{s_i} \quad (27)$$

We define the space reduction factor r characterizing our algorithmic framework as the ratio between the space required by the algorithms of our framework and the space required by same the algorithms without using slicing (i.e., by algorithms with the same p and q but with $m = n = 1$).

$$r = \frac{\sum_{i=1}^{m} 2^{p_i} + \sum_{i=1}^{n} 2^{s_i}}{2^{\sum_{i=1}^{m} p_i} + 2^{\sum_{i=1}^{n} s_i}} \quad (28)$$

```
crc = INIT_VALUE;
while(p_buf < p_end ) {
    crc ^= *(uint32_t *)p_buf;
    term1 = table_56[crc & 0x000000FF] ^
            table_48[(crc >> 8) & 0x000000FF];
    term2 = crc >> 16;
    crc = term1 ^
          table_40[term2 & 0x000000FF] ^
          table_32[(term2 >> 8) & 0x000000FF];
    p_buf += 4;
}
return crc^FINAL_VALUE;
```

Figure 5: The 'Slicing-by-4' Algorithm

D. The Slicing-by-4 and 8 Algorithms

We have used the algorithmic framework presented above to accelerate the performance of CRC32c implementations. CRC32c is data integrity standard specified as part of many well known systems and such as MPA[4] and iSCSI[13]. The length of the CRC value in CRC32c is 32 bits whereas the length of the generator polynomial is 33 bits. The generator polynomial used by the CRC32c has coefficients equal to 0x11EDC6F41. The initial CRC value used is 0xFFFFFFFF.

A 'slicing-by-4' algorithm is shown in Figure 5. The slicing-by-4 algorithm reads 32 bits at a time. The length of the input stream is considered to be multiple of 32. In this case, $p = q = 32$ bits. The length of the value $R^{(k)}$ which is produced by step k of this algorithm is 32 bits. Since the algorithm reads 32 bits at a time, the length of the value $S^{(k)}$ which is produced by step k of the algorithm is 64 bits. During the first step of the algorithm, the first 32 bits of the input stream are not grouped into slices (i.e., $m = 1$) since the number of bits read p is smaller than the degree of the generator polynomial. Each value $S^{(k)}$ produced by step k is split into $n = 5$ slices. The first 4 slices s_1, s_2, s_3 and s_4 have equal lengths, i.e., $s_i = 8$ bits for every $i \in [1, 4]$. The last slice s_5 (i.e., the one associated with zero offset) has length equal to 32 bits.

Because of the fact that slice s_5 is associated with zero offset and its length is equal to 32 bits, the remainder from the division of s_5 with the generator polynomial is s_5 itself. Hence, no table lookup is required for slice s_5. For the slices s_1, s_2, s_3 and s_4, we use four lookup tables of 256 32-bit entries each. Our algorithm is called 'slicing-by-4' because it performs 4 table lookups in parallel, although the number of slices produced in each step is 5. For the first step, we do not use lookup tables since the total number of bits which are read (32) is smaller than the degree of the generator polynomial. Hence, the slicing-by-4 algorithm requires only four lookup tables to execute. The total space requirement of the slicing-by-4 algorithm is 4K bytes.

Figure 5 illustrates an optimized implementation of the slicing-by-4 algorithm in C. The names of the tables follow the convention '`table_`*offset*'. The offset values used for the generation of the tables are 56, 48, 40 and 32 bits, respectively. As in the description of the Sarwate algorithm in Figure 3, the bits of the input stream are considered to be reflected inside their respective bytes.

```
crc = INIT_VALUE;
while(p_buf < p_end ) {
    crc ^= *(uint32_t *)p_buf;
    p_buf += 4;
    term1 = table_88[crc & 0x000000FF] ^
            table_80[(crc >> 8) & 0x000000FF];
    term2 = crc >> 16;
    crc = term1 ^
          table_72[term2 & 0x000000FF] ^
          table_64[(term2 >> 8) & 0x000000FF];
    term1 = table_56[(*(uint32_t *)p_buf) & 0x000000FF] ^
            table_48[((*(uint32_t *)p_buf) >> 8) & 0x000000FF];
    term2 = (*(uint32_t *)p_buf) >> 16;
    crc = crc ^
          term1 ^
          table_40[term2  & 0x000000FF] ^
          table_32[(term2 >> 8) & 0x000000FF];
    p_buf += 4;
}
return crc^FINAL_VALUE;
```

Figure 6: The 'Slicing-by-8' Algorithm

Another algorithm designed using our framework called 'slicing-by-8' reads 64 bytes at a time. The slicing-by-8 algorithm is shown in Figure 6. In the implementation of Figure 6, $p = 32$ bits and $q = 64$ bits. The length of the value $R^{(k)}$ which is produced by step k of this algorithm is 32 bits. Since the algorithm reads 64 bits at a time, the length of the value $S^{(k)}$ which is produced by step k of the algorithm is 96 bits. As in the slicing-by-4 algorithm, the first 32 bits of the input stream are not grouped into slices (i.e., $m = 1$). Each value $S^{(k)}$ produced by step k is split into $n = 9$ slices. The first 8 slices have equal lengths, i.e., $s_i = 8$ bits for every $i \in [1, 8]$. The last slice s_9 has length equal to 32 bits.

Because of the fact that slice s_9 is associated with zero offset and its length is equal to 32 bits, no table lookup is required for this slice. For the slices s_1-s_8 we use eight lookup tables of 256 32-bit entries each. As in the slicing-by-4 algorithm, the first step does not involve table

lookups. The total space requirement of the slicing-by-8 algorithm is 8K bytes, which is two times the space requirement of slicing-by-4. Figure 6 illustrates an optimized implementation of the slicing-by-8 algorithm in C. The offset values used for the generation of the tables are 88, 80, 72, 64, 56, 48, 40 and 32 bits respectively.

```
            mov ecx, p_buf
            mov esi, buf_length
            cmp esi, 0
            jz SHORT end
            add esi, ecx
            or eax, -1
            push edi
top_of_loop:
            xor edx, edx
            mov dl, BYTE PTR [ecx]
            xor edx, eax
            and edx, 255
            shr eax, 8
            mov edi, DWORD PTR table[edx*4]
            xor eax, edi
            inc ecx
            xor edx, edx
            mov dl, BYTE PTR [ecx]
            xor edx, eax
            and edx, 255
            shr eax, 8
            mov edi, DWORD PTR table[edx*4]
            xor eax, edi
            inc ecx
            xor edx, edx
            mov dl, BYTE PTR [ecx]
            xor edx, eax
            and edx, 255
            shr eax, 8
            mov edi, DWORD PTR table[edx*4]
            xor eax, edi
            inc ecx
            xor edx, edx
            mov dl, BYTE PTR [ecx]
            xor edx, eax
            and edx, 255
            shr eax, 8
            mov edi, DWORD PTR table[edx*4]
            xor eax, edi
            inc ecx
            cmp ecx, esi
            jb SHORT top_of_loop
            pop edi
end:
            not eax
```

Figure 7: Assembly Implementation of the Sarwate Algorithm

V. EVALUATION

A. Qualitative Evaluation

We observe that a trade-off exists between the number of operations involved in the execution of a CRC generation algorithm and the space requirement of the algorithm. From Eq. 19-22 it is evident that the number of operations involved in the execution of a CRC generation algorithm is minimized when the number of slices used is equal to one, i.e., $m = n = 1$. On the other hand, when $m = n = 1$ the space reduction factor r is maximized as indicated by Eq. 28. The maximum value for r is 1.

The benefit from slicing comes from the fact that modern processor architectures comprise large cache units. These cache units are capable of storing moderate size tables (e.g., 4KB and 8KB tables as required by the slicing-by-4 and slicing-by-8 algorithms) but not sufficient for storing tables associated with significantly larger strides (e.g., 16BG tables associated with 32-bit strides). If tables are stored in an external memory unit, the latency associated with accessing these tables may be significantly higher than when tables are stored in a cache unit. For example, a DRAM memory access requires several hundreds of clock cycles to complete by a Pentium® M processor, whereas an access to a first level cache memory unit requires less than five clock cycles to complete. The processing cost associated with slicing, which is observed in Eq. 19-22, is typically insignificant when compared to the cost of accessing off-chip memory units.

```
            mov ecx, p_buf
            mov edx, buf_length
            cmp edx, 0
            jz SHORT end
            add edx, ecx
            or eax, -1
            push esi
            push edi
top_of_loop:
            xor eax, DWORD PTR [ecx]
            add ecx, 4
            movzx esi, al
            mov edi, DWORD PTR table_o56[esi*4]
            movzx esi, ah
            xor edi, DWORD PTR table_o48[esi*4]
            bswap eax
            movzx esi, ah
            xor edi, DWORD PTR table_o40[esi*4]
            movzx esi, al
            xor edi, DWORD PTR table_o32[esi*4]
            mov eax, edi
            cmp ecx, edx
            jb SHORT top_of_loop
            pop edi
            pop esi
end:
            not eax
```

Figure 8: Assembly Implementation of the Slicing-by-4 Algorithm

Slicing is also important because it reduces the number of operations performed for each byte of an input stream when compared to other techniques used in the state of the art. For example, the slicing-by-4 and slicing-by-8 algorithms are faster than the Sarwate algorithm. This happens because the Sarwate algorithm calculates updates on the CRC values of streams on a byte-by-byte basis. In contrast, the slicing-by-4 and slicing-by-8 algorithms calculate updates on CRC values reading 32 bit and 64 bit amounts at a time. To further demonstrate this, we show the instructions required for executing the Sarwate and slicing-by-4 algorithms over 32 bits of data using Intel's IA32 processor architecture. Instructions are shown in Figures 7 and 8 respectively. For fair comparison, the loop of the Sarwate algorithm is unrolled over four iterations. The loop of the Sarwate algorithm consists of 35 IA32 instructions, whereas the loop of the slicing-by-4 algorithm consists of 16 instructions. For each byte of an input stream the Sarwate algorithm performs the following: (i) an XOR operation between a byte read and the least significant byte of the current CRC value; (ii) a table lookup; (iii) a shift operation on the current CRC value; and (iv) an XOR operation between the shifted CRC value and the word read

from the table. In contrast, for every byte of an input stream the slicing-by-4-algorithm performs only a table lookup and an XOR operation. This is the reason why the slicing-by-4 algorithm is faster than the Sarwate algorithm. Since the Sarwate algorithm requires 35 instructions to execute over 32 bits and the slicing-by-4 algorithm requires 16 instructions only, one can expect that the slicing-by-4 algorithm is approximately 2.2 times faster than the Sarwate algorithm provided that data are placed in a cache memory unit. Similarly, we observe that the slicing-by-8 algorithm requires 27 instructions to execute over 64 bits. As a result, one can expect that the slicing-by-8 algorithm is approximately 2.6 times faster than the Sarwate algorithm provided that data are placed in a cache memory unit.

	minimum processing cost (cycles/byte)	average processing cost for warm data and warm tables (cycles/byte)	average processing cost for cold data and warm tables (cycles/byte)
Sarwate	6.10	6.66	6.67
Joshi-Dubey-Kaplan	5.18	5.65	5.67
Slicing-by-4	2.75	3.29	3.31
Slicing-by-8	2.19	2.39	2.41

Table 2: Minimum and average processing costs

We observe that our slicing-by-4 and slicing-by-8 implementations do not suffer from cache pollution significantly. By 'cache pollution' we mean the undesired and uncontrolled eviction of application data structures from cache memory. In the case of CRC generation algorithms, cache pollution is caused by the fact that packets are fetched into cache units at very high speeds. The extent of cache pollution depends on the size and quantity of the source buffers over which CRC algorithms operate. We argue that the impact of cache pollution on the performance of CRC generation algorithms is not significant because on-chip cache memory units are typically large, capable of storing many packet buffers simultaneously. In addition, current processors employ a number of custom solutions for avoiding cache pollution. For example, cache lines can be set to an 'invalid' state after they are being used. In this way, the next packets fetched from memory can be placed in the same cache lines as the previous packets without causing undesired evictions. The impact of cache pollution on the performance of the slicing-by-4 and slicing-by-8 algorithms is quantified in the next section.

The implementations of the slicing-by-4 and slicing-by-8 algorithms presented in this paper can be further optimized by performing table lookups in parallel as opposed to sequentially using different general purpose registers in each lookup. Another way to accelerate the slicing-by-4 and slicing-by-8 algorithms is by employing multiple processing units for parallel packet processing. The algorithms of our framework can ideally read arbitrarily large amounts of data at a time and also create an arbitrary number of slices in each step of their execution. As a result, the slices produced in each step of the execution of these algorithms can be processed by different processors. The speed of CRC generation algorithms is multiplied by the number of processors used, in this case. The detailed description and evaluation of parallelized CRC generation algorithms, however, is beyond the scope of this paper.

B. *Quantitative Evaluation*

1) *Minimum and Average Processing Costs*

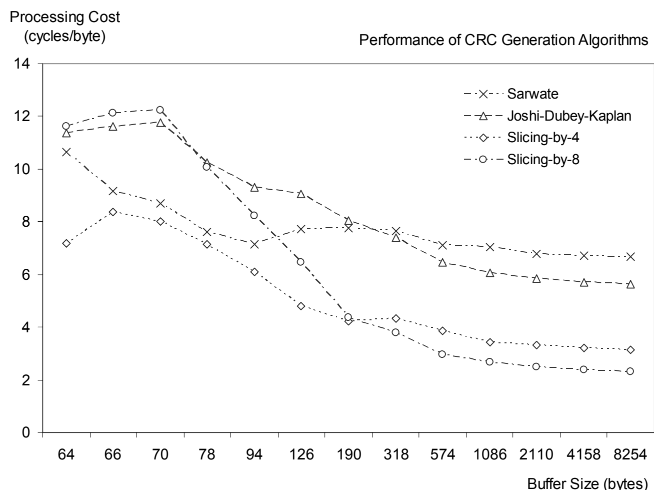
We compare the performance of the slicing-by-4 and slicing-by-8 algorithms with the state of the art. The minimum and average processing costs associated with the Sarwate [12], Joshi-Dubey-Kaplan [9], slicing-by-4 and slicing-by-8 algorithms are shown in Table 2. By 'minimum' cost we mean the processing cost of these algorithms when packets and tables are placed in the fastest cache memory unit and no other activities such as operating system interruptions occur. The minimum processing cost is independent of the values of packet sizes. The minimum cost values presented in Table 2 reflect the performance of CRC algorithms under ideal operating conditions. By 'average' cost we mean the average processing cost of algorithms when algorithms run under more realistic operating conditions. By 'realistic operating conditions' we mean conditions when operating system interruptions or other system activities can occur during the generation of CRC values. Each cost number presented in the third and fourth column of Table 2 represents an average value measured over 200 experiments. In these experiments, values for the size, content, and initial address alignment of packets are chosen by a pseudo-random number generator. Our pseudo-random number generator implements the uniform probability distribution. The processor used for producing the results of Table 2 is a 'Dothan' Pentium® M processor. This processor includes a 32KB first level (L1) cache unit and a 2MB second level (L2) cache unit. In what follows we use the term 'warm' to refer to any memory entry placed in a cache memory unit. Similarly, we use the term 'cold' to refer to any memory entry stored in an external memory unit.

Table 2 shows that the slicing-by-8 algorithm is the fastest among the four algorithms compared. The minimum cost of slicing-by-8 is 2.19 cycles per byte, which is 2.79 times smaller than the cost of the Sarwate algorithm and 2.37 times smaller than the cost of the Joshi-Dubey-Kaplan algorithm. We also observe that the minimum processing cost of the slicing-by-4 algorithm is similar to the cost of slicing-by-8 and equal to 2.75 cycles per byte. Slicing-by-4 is 2.22 times faster than the Sarwate algorithm and 1.88 times faster than the Joshi-Dubey-Kaplan algorithm.

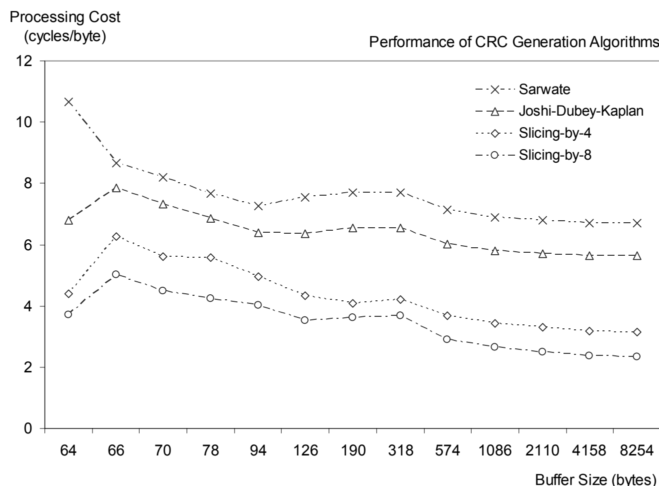
We observe that the averaging processing costs of the algorithms of Table 2 are higher than their minimum costs. For example, the minimum cost of the slicing-by-8 algorithm is 2.19 cycles per byte, whereas its average processing cost when data and tables are initially warm is 2.39 cycles per byte. The difference in the cost values observed is due to operating system interruptions or other activities which evict memory entries from the cache memory. However, the impact of cache pollution on the performance of the algorithms of Table 2 is not significant as discussed above.

Another observation we make is that the average processing cost of the algorithms of Table 2 is higher when packets are initially cold. For example we observe that the cost of the slicing-by-8 algorithm 2.41 cycles per byte when packets are

initially cold whereas the same cost value is 2.39 cycles per byte when packets are initially warm. The reason why such difference is not significant is because a large part of the content of packets is prefetched into the cache memory by the processor. As a result only a few cache misses typically occur during the CRC generation process. These cache misses usually occur during the beginning of CRC generation process. The larger the size of a packet buffer is the smaller increase in the processing cost of CRC generation is likely to be observed.



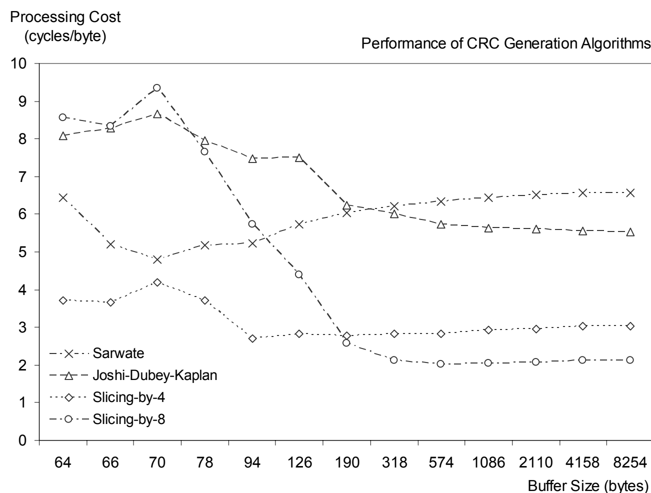
(a) initial state: cold data, cold tables



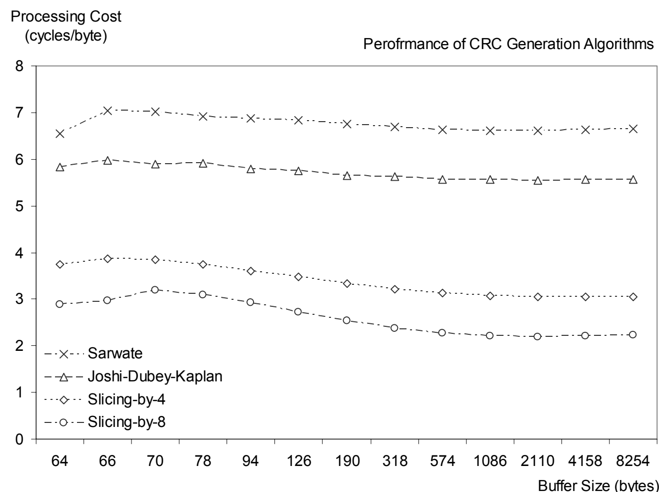
(b) initial state: cold data, warm tables

Figure 9: The impact of warming the tables on the performance of CRC generation algorithms

The implementation of the Joshi-Dubey-Kaplan algorithm which we use in our experiments is different from the one described in reference [9] in that our implementation does not take advantage of the instruction set of IBM's PowerPC architecture. Comparing algorithm implementations is difficult since the performance of implementations varies depending on how implementations are optimized. In this paper we report how our scheme compares with the state of the art for the sake of qualitative comparison only. The reason why the slicing-by-4 and slicing-by-8 algorithms perform better than the Joshi-Dubey-Kaplan algorithm is because the latter uses a single table only. As a result, the values returned from each table lookup need to be XOR-ed with each other at different offsets creating a 'CRC tail' every time the algorithm's main loop is executed. In contrast, the slicing-by-4 and slicing-by-8 algorithms use different tables. In this way, the slicing-by-4 and slicing-by-8 algorithms reduce the number of operations needed for producing the final CRC value and avoid the creation of CRC tails.



initial state: warm data, cold tables



initial state: warm data, warm tables

Figure 10: The impact of warming the data on the performance of CRC generation algorithms

2) Impact of warming the tables

In Figure 9 we study the impact of warming the tables on the performance of CRC generation algorithms. Figures 9a and 9b show the processing cost of the algorithms of Table 2 for different packet sizes when tables are initially cold and warm. Each point in the plots of Figures 9a and 9b represents an average value from 200 experiments. Figure 9 shows that it is important for lookup tables to be placed in some cache memory unit. This is true especially when packet sizes are small. For small packet sizes (e.g., between 64 bytes and 574

bytes) we observe significant difference in the performance of CRC generation algorithms which is as high as 7 cycles per byte for the slicing-by-8 algorithm. Such difference is explained by the fact that the latency associated with accessing table entries is significant (e.g., a few hundred clock cycles) when tables are stored in an external memory unit. For large packet sizes, the impact of the initial state of tables is not that high because after some bytes are processed most table entries can be accessed locally.

3) Impact of warming the data

In Figure 10 we study the impact of warming the data on the performance of CRC generation algorithms. Figures 10a and 10b show the processing cost of the algorithms of Table 2 for different packet sizes when data is warm and tables are initially cold and warm. Figure 10 demonstrates the fact that the processing cost of CRC generation algorithms is smaller when data is initially warm than when data is cold as expected. However the impact of the warming the data is not as significant as the impact of warming the tables on the performance of CRC generation algorithms. For example the performance of the slicing-by-8 algorithm is increased only by 2 cycles per byte when data is initially cold and tables are warm. The reason why is because data entries are accessed sequentially, and hence data access patterns can be 'recognized' by the hardware prefetchers of processors. In contrast table entries are accessed in a random manner.

VI. CONCLUDING REMARKS

We presented a framework for designing a family of novel fast CRC generation algorithms. Our algorithms can ideally read arbitrarily large amounts of data at a time, while optimizing their memory requirement to meet the constraints of specific computer architectures. In addition, our algorithms can be implemented in software using commodity processors instead of specialized parallel circuits. We used this framework to design two efficient algorithms that run in the popular Intel IA32 processor architecture outperforming the popular Sarwate algorithm by factors almost equal to 2 and 3.

The most significant contribution of our work is that our algorithms can ideally read arbitrarily large amounts of data at a time and also create an arbitrary number of slices in each step of their execution. As a result, the slices produced in each step of the execution of these algorithms can be processed by different processors. In this case, the speed of CRC generation algorithms is multiplied by the number of processors used and may potentially become arbitrarily large, limited only by the bus speed, the cache footprint and the number of processors used. Further investigation on the design and performance of such parallel algorithms will be the subject of future work.

REFERENCES

[1] G. Albertengo and R. Sisto, "Parallel CRC Generation", *IEEE Micro*, October 1990.

[2] "Architectural Specifications for RDMA over TCP/IP", *RDMA Consortium Web Site*, available at http://www.rdmaconsortium.org

[3] F. Braun and M. Waldvoger, "Fast Incremental CRC Updates for IP over ATM Networks", *High Performance Switching and Routing (HPSR),* 2001.

[4] P. Culley, U. Elzur, R. Recio, S. Bailey and J. Carrier, "Marker MPU Aligned Framing for TCP Specification", *Work in Progress*, Internet Draft, July 2004, expires January, 2005.

[5] A. Doering and M. Waldvogel, "Fast and flexible CRC calculation", *Electronics Letters*, January 2004.

[6] D. Feldmeier, "Fast Software Implementation of Error Correcting Codes", *IEEE Transactions on Networking*, December, 1995.

[7] G. Griffiths and G. C. Stones, "The Tea-leaf Reader Algorithm: An Efficient Implementation of CRC-16 and CRC-32", *Communications of the ACM*, Vol. 30, No. 7, pp. 617-620, Jualy 1987.

[8] C. M. Heard, "AAL2 CPS-PH HEC calculations using table lookups", ftp://ftp.vvnet.com/aal2_hec/crc5.c

[9] S. M. Joshi, P. K. Dubey and M. A. Kaplan, "A New Parallel Algorithm for CRC Generation", *proceedings of the International Conference on Communications*, 2000.

[10] M. C. Nielson, "Method for High Speed CRC computation", *IBM Technical Disclosure Bulletin*, Vol. 27, No. 6, pp. 3572-3576, November 1984.

[11] T. V. Ramabadran and S. V. Gaitonde, "A Tutorial on CRC Computations", *IEEE Micro*, Vol. 8, No. 4, pp. 62-75, August 1988.

[12] D. V. Sarwate, "Computation of Cyclic Redundancy Checks via Table Lookup", *Communications of the ACM*, Vol. 31, No 8, pp.1008-1013, August 1988.

[13] J. Satran, K. Methm C. Sapuntzakis, M. Chadalapaka and E. Zeidner, "Internet Small Computer Systems Interface (iSCSI)", *Request for Comments*, RFC 3720, April 2004.

[14] M. D. Shieh, M. H. Sheu, C. H. Chen and H. F. Lo, "A systematic Approach for Parallel CRC Computations", *Journal of Information Science and Engineering*, Vol. 17, pp. 445-461, 2001

[15] A. Perez, "Byte-wise CRC Calculations", *IEEE Micro*, Vol. 3, No. 3, pp. 40-50, June 1983.

[16] R. N. Williams, "A Painless Guide to CRC Error Detection Algorithms", *Technical Report*, available at: http://www.ross.net/crc, August 1993.

Copyright © 2005 Intel Corporation. All Rights Reserved.

THIS DOCUMENT IS PROVIDED "AS IS" WITH NO WARRANTIES WHATSOEVER, INCLUDING ANY WARRANTY OF MERCHANTABILITY, NONINFRINGEMENT, FITNESS FOR ANY PARTICULAR PURPOSE, OR ANY WARRANTY OTHERWISE ARISING OUT OF ANY PROPOSAL, SPECIFICATION OR SAMPLE. The information in this document is furnished for informational use only, and Intel Corporation assumes no responsibility or liability for any errors or inaccuracies that may appear herein or in any software that may be provided in association with this document. Intel Corporation disclaims all liability, including liability for infringement of any proprietary rights, relating to use of information herein.