Everything we know about CRC but afraid to forget

Andrew Kadatch[1] and Bob Jenkins[2]

[1]Google Inc.
[2]Microsoft Corporation

September 3, 2010

Abstract

This paper describes a novel interleaved, parallelizeable word-by-word CRC computation algorithm which computes N-bit CRC ($N \leq 64$) on modern Intel and AMD processors in 1.2 CPU cycles per byte, improving state of the art over word-by-word 32-bit and 64-bit CRCs (2.1 CPU cycles/byte) and classic byte-by-byte CRC computation (6-7 CPU cycles/byte). It computes 128-bit CRC in 1.7 CPU cycles/byte.

CRC implementations are heavily optimized and hard to understand. This paper describes CRC algorithms as they evolved over time, splitting complex optimizations into a sequence of natural improvements.

This paper also presents a collection of CRC "tricks" that we found handy on many occassions.

Contents

1 Definition of CRC **2**

2 Related work **3**

3 CRC tricks and tips **4**
3.1 Incremental CRC computation 4
3.2 Changing initial CRC value 4
3.3 Concatenation of CRCs 5
3.4 In-place modification of CRC-ed message 5
3.5 Storing CRC value after the message 6

4 Efficient software implementation **7**
4.1 Mapping bitstreams to hardware registers 7
4.2 Multiplication of D-normalized polynomials 7
4.3 Multiplication of unnormalized polynomial 7

4.4 Computing powers of x 8
4.5 Simplified CRC 9
4.6 Computing a CRC byte by byte 9
4.7 Rolling CRC 11
4.8 Reading multiple bytes at a time 12
4.9 Computing a CRC word by word 12
4.10 Processing non-overlapping blocks in parallel 13
4.11 Interleaved word-by-word CRC 16
4.11.1 Parallelizing CRC computation 16
4.11.2 Combining individual CRCs 18
4.11.3 Efficient computation of individual CRCs 19

5 Experimental results **22**
5.1 Testing methology 22
5.2 Compiler comparison 22
5.3 Choice of interleave level 23
5.4 Performance of CRC algorithms 23

1 Definition of CRC

Cyclic Redundancy Check (CRC) is a well-known technique that allows the recipient of a message transmitted over a noisy channel to detect whether the message has been corrupted.

A message $M = m_0 \dots m_{N-1}$ comprised of $N = |M|$ bits ($m_k \in \{0, 1\}$) may be viewed either as a numeric value

$$M = \sum_{k=0}^{N-1} m_k 2^{N-1-k}$$

or as a polynomial of a single variable of degree $(N - 1)$

$$M(x) = \sum_{k=0}^{N-1} m_k x^{N-1-k}$$

where $m_k \in GF(2) = \{0, 1\}$ and all arithmetic operations on coefficients are performed modulo 2. For example,

Addition: $(x^3 + x^2 + x + 1) + (x^2 + x + 1) = x^3 + 2x^2 + 2x + 2 = x^3$,

Subtraction: $(x^3 + x + 1) - (x^2 + x) = x^3 - x^2 + 1 = x^3 + x^2 + 1$,

Multiplication: $(x + 1)(x + 1) = x^2 + 2x + 1 = x^2 + 1$.

For a given polynomial $P(x)$ of degree $D = \deg\big(P(x)\big)$, $\mathrm{CRC}_u\big(M(x), v(x)\big)$ is the reminder from division of $\big(M(x) \cdot x^D\big)$ by $P(x)$. In practice, a more

complex formula is used:

$$\mathrm{CRC}_u\big(M(x), v(x)\big) = \Big(\big(v(x) - u(x)\big) \cdot x^{|M|} + M(x) \cdot x^D + u(x)\Big) \bmod P(x), \tag{1}$$

where polynomial $P(x)$ of degree D and polynomial $u(x)$ of degree less than D are fixed.

The use of the non-zero value of $u(x)$ guarantees that the CRC of a sequence of zeroes is different from zero. That allows detection of insertion of zeroes in the beginning of a message and replacement of both content of the message and its CRC value with zeroes. Typically,

$$u(x) = \sum_{k=0}^{D-1} x^k. \tag{2}$$

The use of auxilary parameter $v(x)$ allows incremental CRC computation as shown in section 3.1.

2 Related work

Cyclic Redundancy Checks (CRCs) were proposed by Peterson and Brown [PB61] in 1961. An efficient table-driven software implementation which reads and processes data byte by byte was described by Hill [Hil79] in 1979, Perez [Per83] in 1983. The "classic" byte-by-byte CRC algorithm described in section 4.6 was published by Sarwate [Sar88] in 1988.

In 1993, Black [Bla93] published a method that reads data by words (described in section 4.8); however, it still computes the CRC byte by byte in strong sequential order.

In 2001, Braun and Waldvogel [BW01] briefly outlined a specialized variant of a CRC that could read input data by words and process them byte by byte – but, thanks to the use of multiple tables, different bytes from the input word could be processed in parallel. In 2002, Ji and Killian [JK02] provided detailed description and analysis of a nearly identical scheme. Both solutions were targeted for hardware implementation. In 2005, Kouvanis and Berry [KB05] demonstrated clear performance benefits of this scheme even when it is implemented in software. A generalized version of this approach is described in section 4.9.

Surprisingly, until [GGO+10] we have not seen prior art describing or utilizing a method of computing a CRC by processing in parallel (in an interleaved manner to utilize multiple ALUs) multiple input streams belonging to non-overlapping sections of input data, desribed in section 4.10.

A novel method of CRC computation that processes in parallel multiple words belonging to overlapping sections of input data is described in section 4.11. A special case restricted to the use of 64-bit tables, 64-bit reads, and 32

or 64-bit generating polynomials was implemented by the authors in February-March 2007 and was used by a couple of Microsoft products. In 2009, the algorithm was generalized and these limitations were removed.

The fact that the CRC of a message followed by its CRC is a constant value which does not depend on the message, described in section 3.5, is well known and has been widely used in the telecommunication industry for long time.

A method of storing a carefully chosen sequence of bits after a message so that the CRC of a message and the sequence of bits appended to the message produces predefined result, described in 3.5, was implemented in 1990 by Zemtsov [Zem90].

A method for recomputing a known CRC using a new initial CRC value, described in section 3.2, and the method of computing a CRC of the concatenation of messages having known CRC values without touching the actual data, described in section 3.3, were implemented by one of the authors in 2005 but were not published.

3 CRC tricks and tips

3.1 Incremental CRC computation

The use of an arbitrary initial CRC value $v(x)$ allows computation of a CRC incrementally. If a message $M(x) = M_1(x) \cdot x^{|M_2|} + M_2(x)$ is a concatenation of messages M_1 and M_2, its CRC may be computed piece by piece because

$$\mathrm{CRC}_u\big(M(x), v(x)\big) = \mathrm{CRC}_u\Big(M_2(x), \mathrm{CRC}_u\big(M_1(x), v(x)\big)\Big). \tag{3}$$

Indeed,

$$\begin{aligned}
\mathrm{CRC}_u(M, v) &= \big((v-u)x^{|M|} + Mx^D + u\big) \bmod P = \\
&= \big((v-u)x^{|M_1|+|M_2|} + (M_1x^{|M_2|} + M_2)x^D + u\big) \bmod P = \\
&= \Big(\big((v-u)x^{|M_1|} + M_1x^D\big)x^{|M_2|} + M_2x^D + u\Big) \bmod P = \\
&= \big(\mathrm{CRC}_u(M_1, v)x^{|M_2|} + M_2x^D + u\big) \bmod P = \\
&= \mathrm{CRC}_u\big(M_2, \mathrm{CRC}_u(M_1, v)\big)
\end{aligned}$$

3.2 Changing initial CRC value

If $\mathrm{CRC}_u\big(M(x), v(x)\big)$ for some initial value $v(x)$ is known, it is possible to compute $\mathrm{CRC}_u\big(M(x), v'(x)\big)$ for different initial value $v'(x)$ without touching the value of $M(x)$:

$$\mathrm{CRC}_u(M, v') = \mathrm{CRC}_u(M, v) + \Big((v'-v)x^{|M|}\Big) \bmod P. \tag{4}$$

Proof:

$$\begin{aligned}
\mathrm{CRC}_u(M, v') &= \big((v'-u)x^{|M|} + Mx^D + u\big) \bmod P = \\
&= \Big(\big((v'-u) + (v-v)\big)x^{|M|} + Mx^D + u\Big) \bmod P = \\
&= \Big(\big((v-u) + (v'-v)\big)x^{|M|} + Mx^D + u\Big) \bmod P = \\
&= \Big(\big((v-u)x^{|M|} + Mx^D + u\big) + (v'-v)x^{|M|}\Big) \bmod P = \\
&= \mathrm{CRC}_u(M, v) + \Big((v'-v)x^{|M|} \bmod P\Big).
\end{aligned}$$

3.3 Concatenation of CRCs

If a message $M(x) = M_1(x) \cdot x^{|M_2|} + M_2(x)$ is a concatenation of messages M_1 and M_2, and CRCs of M_1, M_2 (computed with some initial values $v_1(x)$, $v_2(x)$ respectively) are known, $\mathrm{CRC}_u\big(M(x), v(x)\big)$ may be computed without touching contents of the message M:

1. Using formula (4), the value of $v_1' = \mathrm{CRC}_u(M_1, v)$ may be computed from the known $\mathrm{CRC}_u(M_1, v_1)$ without touching the contents of M_1.
2. Then, $v_2' = \mathrm{CRC}_u(M_2, v_1')$ may be computed from known $\mathrm{CRC}_u(M_2, v_2)$ without touching the contents of M_2.

According to (3), $\mathrm{CRC}_u(M, v) = v_2'$.

3.4 In-place modification of CRC-ed message

Sometimes it is necessary to replace a part of message $M(x)$ in-place and recompute CRC of modified message $M(x)$ efficiently.

If a message $M = ABC$ is a concatenation of messages A, B, and C, and $B'(x)$ is new message of the same length as $B(x)$, $\mathrm{CRC}_u(M')$ of message $M' = AB'C$ may be computed from known $\mathrm{CRC}_u(M)$. Indeed,

$$\begin{aligned}
M(x) &= A(x) \cdot x^{|B|+|C|} + B(x) \cdot x^{|C|} + C(x), \\
M'(x) &= A(x) \cdot x^{|B|+|C|} + B'(x) \cdot x^{|C|} + C(x) = \\
&= M(x) + \big(B'(x) - B(x)\big) \cdot x^{|C|},
\end{aligned}$$

therefore

$$\begin{aligned}
&\mathrm{CRC}_u\big(M'(x), v(x)\big) = \\
&= \mathrm{CRC}_u\Big(M(x) + \big(B'(x) - B(x)\big) \cdot x^{|C|}\Big) = \\
&= \Big(\big(v(x) - u(x)\big)x^{|M|} + M(x)x^D + \big(B'(x) - B(x)\big)x^{|C|+D} + u(x)\Big) \bmod P(x) \\
&= \Big(\mathrm{CRC}_u\big(M(x), v(x)\big) + \big(B'(x) - B(x)\big)x^{|C|+D}\Big) \bmod P(x) = \\
&= \mathrm{CRC}_u\big(M(x), v(x)\big) + \Big(\big(B'(x) - B(x)\big)x^{|C|+D} \bmod P(x)\Big).
\end{aligned}$$

It is easy to see that

$$\begin{aligned}&\mathrm{CRC}_u\big(B'(x), v(x)\big) - \mathrm{CRC}_u\big(B(x), v(x)\big) = \\ &= \big(B'(x) - B(x)\big)x^D \bmod P(x),\end{aligned}$$

so

$$\mathrm{CRC}_u\big(M'(x), v(x)\big) = \mathrm{CRC}_u\big(M(x), v(x)\big) + \Delta$$

where

$$\Delta = \Big(\mathrm{CRC}_u\big(B'(x), v(x)\big) - \mathrm{CRC}_u\big(B(x), v(x)\big)\Big)x^{|C|} \bmod P(x).$$

3.5 Storing CRC value after the message

Often $Q(x) = \mathrm{CRC}_u\big(M(x), v(x)\big)$ is padded with zero bits until the nearest byte or word boundary and is transmitted as a sequence of W bits ($W \geq D$) right after the message $M(x)$. This way, the transmitted message $T(x)$ is the concatenation of $M(x)$ and $Q(x)$ followed by $(W - D)$ zeroes, and is equal to

$$T(x) = M(x) \cdot x^W + Q(x) \cdot x^{W-D}.$$

According to (1), (3) and taking into account that $Q(x) + Q(x) = 0$ since polynomial coefficient are from $GF(2)$, $\mathrm{CRC}_u\big(T(x), v(x)\big)$ is a constant value which does not depend on the contents of the message and is equal to

$$\begin{aligned}&\mathrm{CRC}_u\big(T(x), v(x)\big) = \\ &= \mathrm{CRC}_u\Big(Q(x) \cdot x^{W-D}, CRC\big(M(x), v(x)\big)\Big) = \\ &= \mathrm{CRC}_u\big(Q(x) \cdot x^{W-D}, Q(x)\big) = \\ &= \Big(\big(Q(x) - u(x)\big) \cdot x^W + Q(x) \cdot x^{W-D} \cdot x^D + u(x)\Big) \bmod P(x) = \\ &= \Big(u(x)\left(1 - x^W\right)\Big) \bmod P(x).\end{aligned}$$

A more generic solution is to store a W-bit long value after the message such that the CRC of the transmitted message is equal to a predefined value $R(x)$ (typically $R(x) = 0$). The D-bit value followed by $(W - D)$ zero bits that should be stored after $M(x)$ is

$$\hat{q}\big(Q(x)\big) = \Big(\big(R(x) - u(x)\big)x^{-W} - \big(Q(x) - u(x)\big)\Big) \bmod P(x)$$

where x^{-W} is the multiplicative inverse of $x^W \bmod P(x)$ which exists if $P(x)$ is not divisble by x and may be found by the extended Euclidean algorithm

[Has01]:

$$\begin{aligned}
&\mathrm{CRC}_u\Big(\hat{q}\big(Q(x)\big)x^{W-D}, CRC\big(M(x), v(x)\big)\Big) = \\
&= \mathrm{CRC}_u\Big(\hat{q}\big(Q(x)\big)x^{W-D}, Q(x)\Big) = \\
&= \Big(\big(Q(x) - u(x)\big) \cdot x^W + \hat{q}\big(Q(x)\big) \cdot x^{W-D} \cdot x^D + u(x)\Big) \bmod P(x) = \\
&= R(x).
\end{aligned}$$

4 Efficient software implementation

4.1 Mapping bitstreams to hardware registers

For little-endian machines (assumed from now on), the result of loading of a D-bit word from memory into hardware register matches the expectations: the 0-th bit of the 0-th byte becomes the 0-th (least significant) bit of the word corresponding to $x^{(D-1)}$.

For example, the 32-bit sequence of 4 bytes 0x01, 0x02, 0x03, 0x04 (0x04030201 when loaded into a 32-bit hardware register) corresponds to the polynomial

$$\left(x^{31} + x^{22} + x^{15} + x^{14} + x^5\right).$$

Addition and subtraction of polymonials with coefficients from $GF(2)$ is the bitwise XOR of their coefficients. Multiplication of a polynomial by x is achieved by logical right shift of register contents by 1 bit. If a shift operation causes a carryover, the resulting polynomial has degree D.

Polynomials of degree less than D whose coefficients are recorded using exactly D bits irrespective of actual degree of the polynomial will be called *D-normalized*.

Whenever possible – and unless mentioned explicitly – all polynomials will be represented in D-normalized form.

Since the generating polynomial $P(x)$ is of degree D and has $(D + 1)$ coefficients, it does not fit into the D-bit register. However, its most significant coefficient is guaranteed to be 1 and may be implied implicitly.

4.2 Multiplication of D-normalized polynomials

Multiplication of two D-normalized polynomials may be accomplished by traditional bit-by-bit, shift-and-add multiplication. This is adequate if performance is not a concern. Sample code is given in listing 1.

4.3 Multiplication of unnormalized polynomial

During initialization of CRC tables it may be necessary to multiply d-normalized polynomial $v(x)$ of a degree $d \neq D$ by a D-normalized polynomial. It may be accomplished by representing the operand as a sum of weighted polynomials of

```
// "a" and "b" occupy D least significant bits.
Crc Multiply(Crc a, Crc b) {
  Crc product = 0;
  Crc bPowX[D];  // bPowX[k] = (b * x**k) mod P
  bPowX[0] = b;
  for (int k = 0; k < D; ++k) {
    // If "a" has non-zero coefficient at x**k,
    // add ((b * x**k) mod P) to the result.
    if (((a & (1 << (D-k)) != 0) product ^= bPowX[k];

    // Compute bPowX[k+1] = (b ** x**(k+1)) mod P.
    if (bPowX[k] & 1) {
      // If degree of (bPowX[k] * x) is D, then
      // degree of (bPowX[k] * x - P) is less than D.
      bPowX[k+1] = (bPowX[k] >> 1) ^ P;
    } else {
      bPowX[k+1] = bPowX[k] >> 1;
    }
  }
  return product;
}
```

Listing 1: Multiplication of normalized polynomials

degree of no more than $(D-1)$, then calling $Multiply()$ function repeatedly as shown in listing 2.

4.4 Computing powers of x

Often (see sections 3.2, 3.3, 3.5) it is necessary to compute $x^N \bmod P(x)$ for very large values of N. This may be accomplished in $O\big(\log(N)\big)$ time.

Consider the binary representation of N:

$$N = \sum_{k=0}^{K} n_k 2^k$$

where $n_k \in \{0, 1\}$. Then

$$x^N = x^{\sum n_k 2^k} = \prod_{k=0}^{K} x^{n_k 2^k} = \prod_{n_k!=0} x^{2^k} \tag{5}$$

and may be computed using no more than $(\lfloor \log_2(N) \rfloor + 1)$ multiplications of polynomials of degree less than D provided known values of

$$Pow2k(k) = x^{2^k} \bmod P(x). \tag{6}$$

Values of $Pow2k(k)$ may be computed iteratively using one multiplication

```
// "v" occupies "d" least signficant bits.
// "m" occupies D least significant bits.
Crc MultiplyUnnormalized(Crc v, int d, Crc m) {
  Crc result = 0;
  while (d > D) {
    Crc temp = v & ((1 << D) - 1);
    v >>= D;
    d -= D;
    // XpowN returns (x**N mod P(x)).
    result ^= Multiply(temp, Multiply(m, XpowN(d)));
  }
  result ^= Multiply(v << (D - d), m);
  return result;
}
```

Listing 2: Multiplication of unnormalized polynomial

$\bmod P(x)$ per iteration:

$$\begin{aligned} Pow2k(0) &= 0, \\ Pow2k(k+1) &= x^{2^{k+1}} \bmod P(x) = \\ &= x^{2\cdot 2^k} \bmod P(x) = \\ &= \left(x^{2^k}\right)^2 \bmod P(x) = \\ &= \Big(Pow2k(k-1)\Big)^2 \bmod P(x). \end{aligned}$$

4.5 Simplified CRC

It is sufficient to be able to compute

$$\mathrm{CRC}_0\big(M(x), v(x)\big) = \Big(v(x)\cdot x^{|M|} + M(x)\cdot x^D\Big) \bmod P(x), \tag{7}$$

since

$$\mathrm{CRC}_u\big(M(x), v(x)\big) = \mathrm{CRC}_0\big(M(x), v(x) - u(x)\big) + u(x),$$

$\mathrm{CRC}_u\big(M(x), v(x)\big)$ of message $M = M_1 \ldots M_K$ may be computed incrementally using CRC_0 instead of CRC_u:

$$\begin{aligned} v_0(x) &= v(x) - u(x), \\ v_k(x) &= \mathrm{CRC}_0\big(M_k(x), v_{k-1}(x)\big), \\ \mathrm{CRC}_u(M(x), v(x)) &= v_K + u(x). \end{aligned}$$

4.6 Computing a CRC byte by byte

If $M(x)$ is W-bit value (typically, $W = 8$) and $\deg\big(v(x)\big) < D$, by definition (7)

$$\mathrm{CRC}_0\big(M(x), v(x)\big) = \Big(v(x)\cdot x^W + M(x)\cdot x^D\Big) \bmod P(x).$$

When $D \leq W$,

$$\begin{aligned}\mathrm{CRC}_0\big(M(x), v(x)\big) &= \Big(v(x) \cdot x^W + M(x) \cdot x^D\Big) \bmod P(x) = \\ &= \Big(\big(v(x) \cdot x^{W-D} + M(x)\big) \cdot x^D\Big) \bmod P(x), \qquad (8)\end{aligned}$$

which may be obtained via single lookup into precomputed table T of size 2^W such that $T[i] = \big(i(x) \cdot x^D)\big) \bmod P(x)$ since $\deg\big(v(x) \cdot x^{W-D} + M(x)\big) < W$.

D-normalized representation of $v(x)$ occupies D least significant bits and is equal to $\big(v(x) \cdot x^{W-D}\big)$ when viewed as W-normalized representation which is required to form W-bit index into a table of 2^W entries. Therefore, explicit multiplication of $v(x)$ by x^{W-D} in formula (8) is not required.

When $D \geq W$, $v(x)$ may be represented as

$$v(x) = v_L(x) + v_H(x) \cdot x^{D-W}$$

where

$$\begin{aligned} v_H(x) &= \left\lfloor \frac{v(x)}{x^{D-W}} \right\rfloor, & \deg\big(v_H(x)\big) &< W, \\ v_L(x) &= v(x) \bmod x^{D-W}, & \deg\big(v_L(x)\big) &< D - W. \end{aligned}$$

Since $\deg\big(v_L(x) \cdot x^W\big) < D$, $\Big(v_L(x) \cdot x^W\Big) \bmod P(x) = v_L(x) \cdot x^W$. Therefore,

$$\begin{aligned}&\mathrm{CRC}_0\big(M(x), v(x)\big) = \\ &= \Big(v(x) \cdot x^W + M(x) \cdot x^D\Big) \bmod P(x) = \\ &= \Big(\big(v_L(x) + v_H(x) \cdot x^{D-W}\big) \cdot x^W + M(x) \cdot x^D\Big) \bmod P(x) = \\ &= \Big(v_L(x) \cdot x^W + \big(v_H(x) + M(x)\big) \cdot x^D\Big) \bmod P(x) = \\ &= \Big(v_L(x) \cdot x^W + \big(v_H(x) + M(x)\big) \cdot x^D\Big) \bmod P(x) = \\ &= \Big(v_L(x) \cdot x^W\Big) \bmod P(x) + \Big(\big(v_H(x) + M(x)\big) \cdot x^D\Big) \bmod P(x) = \\ &= \Big(v_L(x) \cdot x^W\Big) + \mathrm{MulByXpowD}\big(v_H(x) + M(x)\big), \qquad (9)\end{aligned}$$

where

$$\mathrm{MulByXpowD}\big(a(x)\big) = \big(a(x) \cdot x^D\big) \bmod P(x). \qquad (10)$$

The value of $\big(v_L(x) \cdot x^W\big)$ may be computed by shifting $v(x)$ by W bits and discarding W carry-over zero bits.

Since $\deg\big(v_H(x) + M(x)\big) < W$, the value of $\mathrm{MulByXpowD}\big(v_H(x) + M(x)\big)$ may be obtained using precomputed table containing 2^W entries.

The classic table-driven, byte-by-byte CRC computation [Per83, Sar88] implementing formulas (1), (3), (8), (9), and (10) for $W = 8$ is given in listing 3.

```
Crc CrcByte(Byte value) {
  return MulByXpowD[value];
}
Crc CrcByteByByte(Byte *data, int n, Crc v, Crc u) {
  Crc crc = v ^ u;
  for (int i = 0; i < n; ++i) {
    Crc ByteCrc = CrcByte(crc ^ data[i]);
    crc >>= 8;
    crc ^= ByteCrc;
  }
  return (crc ^ u);
}
void InitByteTable() {
  for (int i = 0; i < 256; ++i) {
    MulByXPowD[i] = MultiplyUnnormalized(i, 8, XpowN(D));
  }
}
```

Listing 3: Computing CRC byte by byte

Experience shows that computing CRC byte by byte is rather slow and, depending on a compiler and input data size, takes $6-8$ CPU cycles per byte on modern 64-bit CPU for $D <= 64$. There are two reasons for it:

1. Reading data 8 bits at a time is not the most efficient data access method on 64-bit CPU.

2. Modern CPUs have multiple ALUs and may execute 3-4 instructions per CPU cycles provided the instructions handle independent data flows. However, byte-by-byte CRC contains only one data flow. Futhermore, most instructions use the result from the previous instruction, leading to CPU stalls because of result propagation delays.

4.7 Rolling CRC

Given a set of messages $M_k = m_k \dots m_{k+N-1}$ where m_k are W-bit symbols and N is fixed (i.e. each next message is obtained by removing first symbol and appending new one), $C_{k+1} = \mathrm{CRC}_u(M_{k+1}, v)$ may be obtained from known $C_k = \mathrm{CRC}_u(M_k, v)$ and symbols m_k and m_{k+N} only, without the need to compute CRC of entire message M_{k+1}. This property may be utilized to efficiently compute a set of rolling Rabin fingerpints.

Since $M_{k+1}(x) = M_k(x)x^W - m_k(x)x^{NW} + m_{k+N}(x)$,

$$
\begin{aligned}
&C_{k+1}(x) = \mathrm{CRC}_u\big(M_{k+1}(x), v(x)\big) = \\
&= \left(\big(v(x) - u(x)\big)x^{NW} + u(x) + \sum_{n=0}^{N-1} m_{k+1+n}(x)x^{D+W(N-1-n)} \right) \bmod P(x) = \\
&= F\big(C_k(x), m_{k+N}(x)\big) + G\big(m_k(x)\big),
\end{aligned}
$$

where

$$F\big(C_k(x), m_{k+N}(x)\big) = \Big(C_k(x)x^W + m_{k+N}(x)x^D\Big) \bmod P,$$

$$G\big(m_k(x)\big) = \Big(\big((v(x) - u(x))x^{NW} + u\big)(1 - x^W) - m_k(x)x^{D+NW}\Big) \bmod P$$

are polynomials of degree less than D.

$G\big(m_{k-1}(x)\big)$ may be computed easily via a single lookup in a table of 2^W entries indexed by m_k.

Computation of $F\big(C_k(x), m_{k+N}(x)\big)$ may be implemented as described in section 4.6 and requires one bitwise shift, one bitwise XOR, and one lookup into a precomputed table containing 2^W entries.

4.8 Reading multiple bytes at a time

One straightforward way to speed up byte-by-byte CRC computation is to read $W > 8$ bits at once. Unfortunately, this is the path of very rapidly diminishing return as the size of the MulByPowD table increases with W exponentially. From practical perspective, it is extremely desirable to ensure that the MulByPowD table fits into the L1 cache (32-64KB), otherwise table entry access latency sharply increases from 3-4 CPU cycles (L1 cache) to 15-20 CPU (L2 cache).

The value of $\mathrm{MulByXpowD}\big(v(x)\big)$ may be computed iteratively using a smaller table because

$$\mathrm{MulByXpowD}\big(v(x)\big) = v(x) \cdot x^D \bmod P(x) = \mathrm{CRC}_0\big(v(x), 0\big) \tag{11}$$

and therefore may be computed using formulas (3) and (9) for smaller values of W'.

[Bla93] provided the implementation for $W = 32$ and $W' = 8$. Our more general implementation was faster than byte-by-byte CRC but not substentially: the improvement was in 20-25% range. However, the result is still important – it demonstrates that reading input data per se is not a bottleneck.

4.9 Computing a CRC word by word

The value of $\mathrm{MulByXpowD}\big(v(x)\big)$ may be computed using multiple smaller tables instead of one table. Given that $\deg\big(v(x)\big) < W$, $v(x)$ may be represented as a weighted sum of polynomials $v_k(x)$ such that $\deg\big(v_k(x)\big) < B$:

$$v(x) = \sum_{k=0}^{K-1} v_k(x) \cdot x^{(K-1-k)B},$$

where $K = \lceil W/B \rceil$ and

$$v_k(x) = \left\lfloor \frac{v(x)}{x^{(K-1-k)B}} \right\rfloor \bmod x^B.$$

Consequently,

$$\begin{aligned}\mathrm{MulByXpowD}\big(v(x)\big) &= v(x)\cdot x^D \bmod P(x) = \\ &= \left(\sum_{k=0}^{K-1} v_k(x)\cdot x^{(K-1-k)B}\right)\cdot x^D \bmod P(x) = \\ &= \sum_{k=0}^{K-1}\Big(v_k(x)\cdot x^{(K-1-k)B+D} \bmod P(x)\Big) = \\ &= \sum_{k=0}^{K-1}\mathrm{MulWordByXpowD}\big(k, v_k(x)\big), \end{aligned} \tag{12}$$

where the values of

$$\mathrm{MulWordByXpowD}\big(k, v_k(x))\big) = v_k(x)\cdot x^{(K-1-k)B+D} \bmod P(x) \tag{13}$$

may be obtained using K precomputed tables. Given that $\deg\big(v_k(x)\big) < B$, each table should contain 2^B entries.

A sample implementation of formulas (1), (3), (12), and (13) is given in listing 4 using $B = 8$ and assuming that W is a multiple of 8.

CrcWordByWord[1] with $W = 64$ uses only 2.1-2.2 CPU cycles/byte on modern 64-bit CPUs (our implementation is somewhat faster than the one described in [KB05]). It solves the problem with data access and, to lesser degree, allows instruction level parallelism: in the middle of the unrolled main loop of CrcOfWord function the CPU may process multiple bytes in parallel.

However, this solution is still imperfect – the beginning of computation contends for a single source of data (variable *value*), and the end of computation contends for a single destination (variable *result*). Further improvement requires processing of multiple independent data streams in interleaved manner so that when computation of one data flow path is stalled the CPU may proceed with another one.

4.10 Processing non-overlapping blocks in parallel

Straighforward pipepiling may be achieved by spliting the input message $M(x) = M_0(x)\ldots M_{N-1}(x)$ into N blocks $M_k(x)$ of approximately the same size and computing CRC of each block in an interleaved manner, concatenating CRCs of individual blocks in the end. A sample implementation is given in listing 5.

A tuned implementation of $CrcWordByWordBlocks$ is capable of processing data at 1.3-1.4 CPU cycles/byte on sufficiently large (64KB and more) inputs, which is noticeably better that 2.1-2.2 CPU cycles/byte delivered by word by word CRC computation. It is a good sign that it is a move in right direction.

[1] The variant presented in this paper is more general than "slicing" described in [KB05]. Sample implementation given in listing 4 does not include one subtle optimization implemented in [KB05] as it was found to be counter-productive.

```
Crc CrcWord(Word value) {
  Crc result = 0;
  // Unroll this loop or let compiler do it.
  for (int byte = 0; byte < sizeof(Word) / 8; ++byte) {
    result ^= MulWordByXpowD[byte][(Byte) value];
    value >>= 8;
  }
  return result;
}
Crc CrcWordByWord(Word *data, int n, Crc v, Crc u)
  Crc crc = v ^ u;
  for (int i = 0; i < n; ++i) {
    Crc WordCrc = CrcWord(crc ^ data[i]);
    if (sizeof(Crc) <= sizeof(Word)) {
      crc = WordCrc;
    } else {
      crc >>= 8;
      crc ^= WordCrc;
    }
  }
  return (crc ^ u);
}
void InitWordTables() {
  for (int byte = 0; byte < sizeof(Word) / 8; ++byte) {
    // (K-1-k)*B + D = (W/8-1-byte)*8 + D = D - 8 + W - 8*byte.
    Crc m = XpowN(D - 8 + sizeof(Word)*8 - 8*byte);
    for (int i = 0; i < 256; ++i) {
      MulWordByXpowD[byte][i] =MultiplyUnnormalized(i, 8, m);
    }
  }
}
```

Listing 4: Computing CRC word by word

```
// Processes N stripes of StripeWidth words each
// word by word, in an interleaved manner.
Crc CrcWordByWordBlocks(Word *data, Crc v, Crc u) {
  assert(n % (N * StripeWidth) == 0);
  // Use N local variables instead of the array.
  Crc crc[N];
  // Initialize the CRC value for each stripe.
  crc[0] = v ^ u;
  for (int stripe = 1; stripe < N; ++stripe)
    crc[i] = 0 ^ u;
  // Compute each stripe's CRC.
  for (int i = 0; i < StripeWidth; ++i) {
    // Compute multiple CRCs in interleaved manner.
    Word buf[N];
    for (int stripe = 0; stripe < N; ++stripe) {
      buf[i] =
        crc[stripe] ^ data[i + stripe * StripeWidth];
      if (D > sizeof(Word) * 8) {
        crc[stripe] >>= D - sizeof(Word) * 8;
      } else {
        crc[stripe] = 0;
      }
    }
    for (int byte = 0; byte < sizeof(Word) / 8; ++byte) {
      for (int stripe = 0; stripe < N; ++stripe) {
        crc[stripe] ^=
          MulWordByXpowD[byte][(Byte) buf[stripe]];
        buf[stripe] >>= 8;
      }
    }
  }
  // Combine stripe CRCs.
  for (int stripe = 1; stripe < N; ++stripe) {
    crc[0] = ChangeStartingValue(
                 crc[stripe], StripeWidth, 0, crc[0]);
  }
  return (crc[0] ^ u);
}
```

Listing 5: Processing non-overlapping blocks in parallel

The drawbacks of this approach are obvious: it does not work well with small inputs – the cost of CRC concatentation becomes a bottleneck, – and it may be susceptible to false cache collisions caused by cache line aliasing.

If the cost of CRC concatenation was not a problem, cache pressure could be mitigated with the use of very narrow stripes. The code in question, lines 33-37 of listing 5 which combine CRCs of individual stripes, iteratively computes

$$\mathrm{crc}_0(x) = \mathrm{crc}_k(x) + \Big(\mathrm{crc}_0 \cdot x^{8S} \bmod P(x)\Big)$$

for $k = 1, \ldots, N-1$ where N and S are the number and the width of the stripes respectively. It may be rearranged as

$$\mathrm{crc}_0(x) = \sum_{k=0}^{N-1} \Big(\mathrm{crc}_{K-1-k} \cdot x^{8kS} \bmod P(x)\Big).$$

Explicit multiplication by x^{8kS} may be avoided by moving it into preset tables

$$\mathrm{MulWordByXPowD}_k(n) = \mathrm{MulWordByXPowD}(n) \cdot x^{kS} \bmod P(x).$$

that are used to compute $crc'_k(x) = \mathrm{crc}_k(x) \cdot x^{8kS}$, so that

$$\mathrm{crc}_0(x) = \sum_{k=0}^{N-1} crc'_k.$$

Unfortunately, this approach alone does not help because

1. It increases the memory footprint of MulWordByXPowD by factor of N. Once the cumulative size of $\mathrm{MulWordByXPowD}_k$ tables exceeds the size of L1 cache (32-64KB), the cost of memory access to multiplication table data increases from 3-4 CPU cycles to 15-20, eliminating all performance gains achieved by reducing the number of table operations.

2. It is still necessary to combine all N values of crc_k into crc_0 at the end of the CRC computation.

4.11 Interleaved word-by-word CRC

4.11.1 Parallelizing CRC computation

Assume that input message M is the concatenation of K groups g_k, and each group g_k is concatenation of N W-bit long words:

$$M(x) = \sum_{k=0}^{K-1} g_k(x) \cdot x^{(K-1-k)NW},$$

$$g_k(x) = \sum_{n=0}^{N-1} m_{k,n} \cdot x^{(N-1-n)W}.$$

Input message $M(x)$ may be represented as

$$\begin{aligned}
M(x) &= \sum_{k=0}^{K-1} g_k(x) \cdot x^{(K-1-k)NW} = \\
&= \sum_{k=0}^{K-1} \left(\sum_{n=0}^{N-1} m_{k,n} \cdot x^{(N-1-n)W} \right) \cdot x^{(K-1-k)NW} = \\
&= \sum_{n=0}^{N-1} \left(\sum_{k=0}^{K-1} m_{k,n} \cdot x^{(K-1-k)NW} \right) \cdot x^{(N-1-n)W} = \\
&= \sum_{n=0}^{N-1} M_n(x) \cdot x^{(N-1-n)W} \qquad (14)
\end{aligned}$$

where

$$M_n(x) = \sum_{k=0}^{K-1} m_{k,n} \cdot x^{(K-1-k)NW}.$$

In other words, M_n is concatenation of n-th W-bit word from g_0 followed by $(N-1)W$ zero bits, then n-th word from g_1 followed by $(N-1)W$ zero bits, etc., ending up with n-th word from last group g_{K-1}.

Appending $(N-1)W$ zero bits to M_n yields $M'_n(x) = M_n(x) \cdot x^{(N-1)W}$ which may be viewed as the concatenation of K NW-bit groups f_k:

$$\begin{aligned}
M'_n(x) &= M_n(x) \cdot x^{(N-1)W} = \sum_{k=0}^{K-1} f_{k,n} \cdot x^{(K-1-k)NW}, \\
f_{k,n}(x) &= m_{k,n}(x) \cdot x^{(N-1)W},
\end{aligned}$$

so

$$\begin{aligned}
M(x) &= \sum_{n=0}^{N-1} M_n(x) \cdot x^{(N-1-n)W} \\
&= \sum_{n=0}^{N-1} M'_n(x) \cdot x^{-(N-1)W} \cdot x^{(N-1-n)W} \\
&= \sum_{n=0}^{N-1} M'_n(x) \cdot x^{-nW}. \qquad (15)
\end{aligned}$$

According to (3), $v_{K,n}(x) = \mathrm{CRC}_0\big(M'_n(x), v_{0,n}(x)\big)$ may be computed incre-

mentally:

$$
\begin{aligned}
v_{k+1,n}(x) &= \mathrm{CRC}_0\big(f_{k,n}(x), v_{k,n}(x)\big) = \\
&= \mathrm{CRC}_0\big(m_{k,n}(x) \cdot x^{(N-1)W}, v_{k,n}(x)\big) = \\
&= \Big(v_{k,n}(x) \cdot x^{NW} + m_{k,n}(x) \cdot x^{(N-1)W} \cdot x^{D}\Big) \bmod P(x) = \\
&= \Big(v_{k,n}(x) \cdot x^{W} + m_{k,n}(x) \cdot x^{D}\Big) \cdot x^{(N-1)W} \bmod P(x) = \qquad (16) \\
&= \mathrm{CrcWordN}\big(m_{k,n}(x), v_{k,n}(x)\big). \qquad (17)
\end{aligned}
$$

This approach:

1. Creates N independent data flows: computation of $v_{k,0}, \ldots, v_{k,N-1}$ may be performed truly in parallel. There are no contentions on a single data source or destination like those the word-by-word CRC computation described in section 4.9 suffered from.

2. Input data is accessed sequentially. Therefore, the load on cache subsystem and false cache collisions are minimal. Thus, the performance bottlenecks of approach described in 4.10 are eliminated.

4.11.2 Combining individual CRCs

Once $v_{K,n}(x) = \mathrm{CRC}_0\big(M'_n(x), v_{0,n}(x)\big)$ are computed starting with

$$
\begin{aligned}
v_{0,0} &= v(x), \\
v_{0,n} &= 0, n \geq 1,
\end{aligned}
$$

by definition (7) of CRC_0 and relationship (15),

$$
\begin{aligned}
\mathrm{CRC}_0\big(M(x), v(x)\big) &= \mathrm{CRC}_0\left(\sum_{n=0}^{N-1} M'_n(x) \cdot x^{-nW}, v(x)\right) = \\
&= \sum_{n=0}^{N-1} \mathrm{CRC}_0\big(M'_n(x) \cdot x^{-nW}, v_{0,n}(x)\big) = \\
&= \sum_{n=0}^{N-1} \mathrm{CRC}_0\big(M'_n(x), v_{0,n}(x)\big) \cdot x^{-nW} = \\
&= \sum_{n=0}^{N-1} v_{K,n}(x) \cdot x^{-nW}. \qquad (18)
\end{aligned}
$$

Even though this step is performed only once per input message, it still requires $(N-1)$ non-trivial multiplications modulo $P(x)$ negatively affecting the performance on small input messages. Also, (18) uses the multiplicative inverse of x^{nW} modulo $P(x)$ which does not exists when $P(x) \bmod x = 0$.

There is more efficient and elegant solution. Assume that $M(x)$ is followed by one more group $g_K(x)$. Then

$$
\begin{aligned}
&\mathrm{CRC}_0\big(M(x)\cdot x^{NW}+g_K(x),v(x)\big)=\\
&=\mathrm{CRC}_0\Big(g_K(x),\mathrm{CRC}_0\big(M(x),v(x)\big)\Big)=\\
&=\Big(\mathrm{CRC}_0\big(M(x),v(x)\big)\cdot x^{NW}+g_K(x)\cdot x^D\Big)\bmod P(x)=\\
&=\left(x^{NW}\sum_{n=0}^{N-1}v_{K,n}(x)\cdot x^{-nW}+x^D\sum_{n=0}^{N-1}m_{K,n}(x)\cdot x^{(N-1-n)W}\right)\bmod P(x)=\\
&=\left(x^{W}\sum_{n=0}^{N-1}v_{K,n}(x)\cdot x^{(N-1-n)W}+x^D\sum_{n=0}^{N-1}m_{K,n}(x)\cdot x^{(N-1-n)W}\right)\bmod P(x)\\
&=\sum_{n=0}^{N-1}\Big(v_{K,n}(x)\cdot x^{W}+m_{K,n}(x)\cdot x^{D}\Big)\cdot x^{(N-1-n)W}\bmod P(x)= \qquad (19)\\
&=\sum_{n=0}^{N-1}\mathrm{CRC}_0\big(m_{K,n}(x),v_{K,n}(x)\big)\cdot x^{(N-1-n)W}\bmod P(x). \qquad (20)
\end{aligned}
$$

(20) may be implemented using formula (13) by setting $v'_0=0$, and then for $n=0,\ldots,N-1$ computing

$$
\begin{aligned}
v'_{n+1}(x)&=\Big(\big(v'_n(x)+v_{K,n}\big)\cdot x^W+m_{K,n}\cdot x^D\Big)\bmod P(x)\\
&=\mathrm{CRC}_0\big(m_{K,n},v'_n(x)+v_{K,n}\big).
\end{aligned}
$$

Alternatively, this step may be performed using the less efficient technique described in section 4.8.

4.11.3 Efficient computation of individual CRCs

Given $v(x)$, $\deg\big(v(x)\big)<D$ and $m(x)$, $\deg\big(m(x)\big)<W$,

$$
\mathrm{CrcWordN}\big(m(x),v(x)\big)=\Big(v(x)\cdot x^W+m(x)\cdot x^D\Big)\cdot x^{(N-1)W}\bmod P(x)
$$

may be implemented efficiently utilizing the techniques described in sections 4.6, 4.8, and 4.9. When $D\le W$,

$$
\begin{aligned}
\mathrm{CrcWordN}\big(m(x),v(x)\big)&=\Big(v(x)\cdot x^W+m(x)\cdot x^D\Big)\cdot x^{(N-1)W}\bmod P(x)=\\
&=\Big(v(x)\cdot x^{W-D}+m(x)\Big)\cdot x^{(N-1)W+D}\bmod P(x),
\end{aligned}
$$

and may be implemented using the table-driven multiplication as described in (13) except that the operand is multiplied by $x^{(N-1)W+D}$ instead of x^D. Like in (8), explicit multiplication of $v(x)$ by x^{W-D} is not required since D-normalized

representation of $v(x)$, viewed as a W-normalized representation, is equal to $\big(v(x) \cdot x^{W-D}\big)$.

Using the same technique as in formula (9), for $D \geq W$ let

$$v_H(x) = \left\lfloor \frac{v(x)}{x^{D-W}} \right\rfloor, \qquad \deg\big(v_H(x)\big) < W,$$
$$v_L(x) = v(x) \bmod x^{D-W}, \qquad \deg\big(v_L(x)\big) < D - W,$$

so that $v(x) = v_L(x) + v_H(x) \cdot x^{D-W}$. Then,

$$\begin{aligned}
&\mathrm{CrcWordN}\big(m(x), v(x)\big) = \\
&= \Big(v(x) \cdot x^W + m(x) \cdot x^D\Big) \cdot x^{(N-1)W} \bmod P(x) = \\
&= \Big(\big(v_L(x) + v_H(x) \cdot x^{D-W}\big) \cdot x^W + m(x) \cdot x^D\Big) \cdot x^{(N-1)W} \bmod P(x) = \\
&= \Big(v_L(x) \cdot x^W + \big(v_H(x) + m(x)\big) \cdot x^D\Big) \cdot x^{(N-1)W} \bmod P(x) = \\
&= \Big(\big(v_H(x) + m(x)\big) \cdot x^{(N-1)W+D} \bmod P(x)\Big) + \\
&+ \Big(\big(v_L(x) \cdot x^W\big) \cdot x^{(N-1)W} \bmod P(x)\Big). \qquad (21)
\end{aligned}$$

Since $\deg\big(v_H(x)+m(x)\big) < W$, the first summand of $\mathrm{CrcWordN}\big(m(x), v(x)\big)$,

$$\Big(\big(v_H(x) + m(x)\big) \cdot x^{(N-1)W+D} \bmod P(x)\Big),$$

may be computed using the table-driven multiplication technique described in (13) except that the operand is multiplied by $x^{D+(N-1)W}$ instead of x^D.

Computation of the second summand of $\mathrm{CrcWordN}\big(m(x), v(x)\big)$,

$$\Big(\big(v_L(x) \cdot x^W\big) \cdot x^{(N-1)W} \bmod P(x)\Big),$$

is somewhat less intuitive. Since $\deg\big(v_L(x)\big) < D - W$,

$$\big(v_L(x) \cdot x^W\big) \bmod P(x) = \big(v_L(x) \cdot x^W\big),$$

and may be computed by shifting $v_L(x)$ by W bits. Additional multiplication by $x^{(N-1)W}$ is accomplished by adding $\big(v_L(x) \cdot x^W\big)$, produced at step $n < N-1$ of the algorithm described by formula (17), to the value of $v_{k,n+1}(x)$ which will be additionally multiplied by $x^{(N-1)W}$ as shown in formula (16).

For $n = N - 1$, the value of $\big(v_L(x) \cdot x^W\big)$ should be added to the value of $v_{k+1,n'}(x)$ where $n' = 0$. For $k < K$, it will be multiplied by $x^{(N-1)W}$ during next round of parallel computation as shown in (16). For $k = K$, $v_{k+1,n'}(x)$ will be multiplied by $x^{(N-1)W}$ during CRC concatenation as shown in (19) since $n' = 0$.

```
Crc CrcInterleavedWordByWord(
    Word *data, int blocks, Crc v, Crc u) {
  Crc crc[N+1] = {0};
  crc[0] = v ^ u;
  for (int i = 0; i < N*(blocks - 1); i += N) {
    Word buffer[N];
    // Load next N words and move overflow
    // bits into "next" word.
    for (int n = 0; n < N; ++n) {
      buffer[N] = crc[n] ^ data[i + n];
      if (D > sizeof(Word) * 8)
        crc[n+1] ^= crc[n] >> (sizeof(Word) * 8);
      crc[n] = 0;
    }
    // Compute interleaved word-by-word CRC.
    for (int byte = 0; byte < sizeof(Word); ++byte) {
      for (int n = 0; n < N; ++n) {
        crc[n] ^=
            MulInterleavedWordByXpowD[byte][(Byte) buffer[n]];
        buffer[n] >>= 8;
      }
    }
    // Combine crc[0] with delayed overflow bits.
    crc[0] ^= crc[N];
    crc[N] = 0;
  }
  // Process the last N bytes and combine CRCs.
  for (int n = 0; n < N; ++n) {
    if (n != 0) crc[0] ^= crc[n];
    Crc WordCrc = CrcOfWord(crc[0] ^ data[i + n]);
    if (D > sizeof(Word) * 8) {
      crc[0] >>= D - sizeof(Word) * 8;
      crc[0] ^= WordCrc;
    } else {
      crc[0] = WordCrc;
    }
  }
  return (crc[0] ^ u);
}
void InitInterleavedWordTables(void) {
  for (int byte = 0; byte < sizeof(Word); ++byte) {
    Crc m = XpowN(D - 8 + N*sizeof(Word)*8 - 8*byte);
    for (int i = 0; i < 256; ++i) {
      MulInterleavedWordByXpowD[byte][i] =
          MultiplyUnnormalized(i, 8, m);
    }
  }
}
```

Listing 6: Interleaved, word by word CRC computation

5 Experimental results

The tests were performed using Intel Q9650 3.0GHz CPU, DDR2-800 memory with 4-4-4-12 timing, and a motherboard with an Intel P45 chipset.

5.1 Testing methology

All tests were performed using random input data over various block sizes. The code for all evaluated algorithms was heavily optimized. Tests were performed on both aligned and non-aligned input data to ensure that misaligned inputs do not carry performance penalty. CRC tables were aligned on 256-byte boundary.

Tests were performed with warm data and warm CRC tables: as shown in [KB05], the footprint of CRC tables – as long as they fit into L1 cache – is not a major contributor to the performance.

Performance was measured in number of CPU cycles per byte of input data: apparently, performance of CRC computation is bounded by performance of CPU and its L1 cache latency. Spot testing of few other Intel and AMD CPU models showed little variation in performance measured in CPU cycles per byte despite substential differences in CPU clock frequencies.

To minimize performance variations caused by interference with OS and other applications (context switches, CPU migrations, CPU cache flushes, memory bus interference from other processes, etc.), the test applications were run at high priority, each test was executed multiple times, and the minimum time was measured. That allowed the tests to achieve repeatability within ±1%.

5.2 Compiler comparison

Despite CRC code being rather straightforward, there were surprises (see tables 5 and 4).

On 64-bit AMD64 platform, Microsoft CL compiler (version 15.00.30729) consistently and noticeably generated the fastest code that used general-purpose integer arithmetics. For instance, CRC-64 and CRC-32 code generated by CL was 1.24 times faster than the code generated by Intel's ICL 11.10.051, and 1.74 times faster than the code generated by GCC 4.5.0. A tuned, hand-written inline assembler code for CRC-32 and CRC-64 for GCC was as fast as the code generated by CL.

When it comes to arithmetics with the use of SSE2 intrinsic functions on 64-bit AMD64 platform, the code generated by GCC 4.5.0 consistenly outperformed the code generated by Microsoft and Intel compilers – by factor of 1.15 and 1.30 respectively. However, earlier versions of GCC did not produce efficient SSE2 code either. For that reason, pre-4.5.0 versions of GCC use hand-written inline assember code which was as fast as the code generated by GCC 4.5.0.

On 32-bit bit X86 platform, neither compiler was able to generate efficient code (most likely because because the compilers could not overcome scarsity of general-purpose registers). Performance of the code that used MMX intrinsic

functions was better but still not as good as hand-written assember versions, which were provided for all compilers.

The fastest code for 128-bit CRC on X86 platform was generated by GCC 4.5.0.

5.3 Choice of interleave level

Number of data streams processed by interleaved, word-by-word CRC computation described in section 4.11 should matter. Too few means underutilization of available ALUs. Too many will increase the length of the main loop and stress instruction decoders, and may cause splilling of registers containing hot data (interleaved processing of N words of data uses at least $(2N + 2)$ registers).

As table 3 shows, the optimal number of interleaved data streams on modern Intel and AMD CPUs for integer arithmetics is either 3 or 4 (likely because they all have exactly 3 ALUs). However, for SSE2 arithmetics on AMD64 platform the optimal number of streams is 6 (3 on X86), which is quite counter-intuitive result as it does not correlate with the number of available ALUs. Good old performance mantra "you need to measure" still applies.

5.4 Performance of CRC algorithms

Average performance of best variants of CRC algorithms for 64-bit AMD64 and 32-bit X86 platforms processing 1KB, 2KB, ..., 1MB inputs is given in tables 1 and 1 respectively. Proposed interleaved multiword CRC algorithm is 1.7-2.0 times faster that current state of the art "slicing".

As demonstrated in tables 7 and 6, interleaved word-by-word CRC described in section 4.11, running at 1.2 CPU cycles/byte, is 1.8 times faster than 2.1 CPU cycles/byte achieved by current state of the art word-by-word CRC algorithm ("slicing") described in [KB05].

On 64-bit AMD64 platform, the best performance was achieved using 64-bit reads and 64-bit tables for all variants of N-bit CRC for $N \leq 64$. In particular, tables 7 and 6 clearly show that performance of 32-bit and 64-bit CRCs is nearly identical. Consequently, there is no reason to favor CRC-32 over CRC-64 for performance reasons.

The use of MMX on the 32-bit X86 platform allowed to utilize 64-bit tables and 64-bit reads achieving 1.3 CPU cyles/byte. Neither compiler generated efficient code using MMX intrinsic functions, so inline assembler was used.

With the use of SSE2 intrinsics on AMD64 architecture, 128-bit CRC may be computed takes at 1.7 CPU cycles/byte using the new algorithm (see table 9), compared with 2.9 CPU cycles/byte achieved by word-by-word CRC computation (see table 8). On the 32-bit X86 architecture, the use of SSE2 intrinsics and GCC 4.5.0 allowed the computation of 128-bit CRC at 2.1 CPU cycles/byte, compared with 4.2 CPU cycles/byte delivered by word-by-word algorithm.

Given that MD5 computation takes 6.8-7.1 CPU cycles/byte and SHA-1 takes 7.6-7.9 CPU cycles per byte, CRCs are still the algorithm of choice for data corruption detection.

References

[Bla93] Richard Black. Fast CRC32 in software. `http://www.cl.cam.ac.uk/research/srg/bluebook/21/crc/crc.html`, 1993.

[BW01] Florian Braun and Marcel Waldvogel. Fast incremental CRC updates for IP over ATM networks. Technical Report WUCS-01-08, Washington University in St. Louis, April 2001. Available at `http://marcel.wanda.ch/Publications/braun01fast-techreport.pdf`.

[GGO+10] Vinodh Gopal, Jim Guilford, Erdinc Ozturk, Gil Wolrich, Wajdi Feghali, Martin Dixon, and Deniz Karakoyunlu. Fast CRC computation for iSCSI polynomial using CRC32 instruction. Intel White Paper 323405, February 2010. Available at `http://download.intel.com/design/intarch/papers/323405.pdf`.

[Has01] M. A. Hasan. Efficient computation of multiplicative inverses for cryptographic applications. In *ARITH '01: Proceedings of the 15th IEEE Symposium on Computer Arithmetic*, page 66, Washington, DC, USA, 2001. IEEE Computer Society.

[Hil79] John R. Hill. A table driven approach to cyclic redundancy check calculations. *SIGCOMM Comput. Commun. Rev.*, 9(2):40–60, 1979.

[JK02] H. Michael Ji and Eeal Killian. Fast parallel crc algorithm and implementation on a configurable processor. In *ICC*, volume 3, pages 1813–1817, April 2002.

[KB05] Michael E. Kounavis and Frank L. Berry. A systematic approach to building high performance, software-based, CRC generators. `http://www.intel.com/technology/comms/perfnet/download/CRC_generators.pdf`, 2005.

[PB61] W.W. Peterson and D.T. Brown. Cyclic codes for error detection. In *IRE(1)*, volume 49, pages 228–235, January 1961.

[Per83] Aram Perez. Byte-wise CRC calculations. *IEEE Micro*, 3(3):40–50, 1983.

[Sar88] Dilip V. Sarwate. Computation of cyclic redundancy checks via table look-up. *Commun. ACM*, 31(8):1008–1013, 1988.

[Zem90] Pavel Zemtsov. Proprietary copy protection system. Personal communication, August 1990.

Table 1: CRC performance, AMD64 platform

Method	Slicing[1]	Multiword[2]	Improvement
CRC-32	2.08[3]	1.16[4,5]	1.79
CRC-64	2.09[3]	1.16[4,5]	1.79
CRC-128	2.91[4]	1.68[4,6]	1.73

Table 2: CRC performance, X86 platform

Method	Slicing[1]	Multiword[2]	Improvement
CRC-32	2.52[3]	1.29[3,7]	1.96
CRC-64	3.28[3]	1.29[3,7]	2.55
CRC-128	4.17[4]	2.10[4,8]	1.98

Average number of CPU cycles per byte processing 1KB, 2KB, . . . , 1MB inputs. Warm data, warm tables.
[1] *"Slicing"* implements the algorithm described in section 4.9.
[2] *"Multiword/N"* implements algorithm described in section 4.11 processing N data streams in parallel in interleaved manner.
[3] Microsoft CL 15.00.30729 compiler, "-O2" flag.
[4] GCC 4.5.0 compiler, "-O3" flag.
[5] Multiword/$N = 4$, hand-written inline assembler.
[6] Multiword/$N = 6$, C++.
[7] Multiword/$N = 4$, hand-written MMX inline assember.
[8] Multiword/$N = 3$, C++.

Table 3: Interleaved multiword CRC: choosing the number of stripes N

CRC	Platform	N=2	N=3	N=4	N=5	N=6	N=7	N=8
CRC-64[9]	AMD64	1.42	1.23	**1.17**	1.46	2.08	2.59	2.73
CRC-128[10]	AMD64	2.07	1.84	1.76	1.70	**1.68**	1.75	1.79
CRC-128[10]	X86	2.56	**2.10**	2.46	2.61	2.52	2.62	2.57

Average number of CPU cycles per byte processing 1KB, 2KB, . . . , 1MB inputs. Interleaved word-by-word CRC computation as described in section 4.11. Warm data, warm tables.
[9] Microsoft CL 15.00.30729 compiler, AMD64 platform, C++ code.
[10] GCC 4.5.0 compiler, AMD64 platform, C++ code.

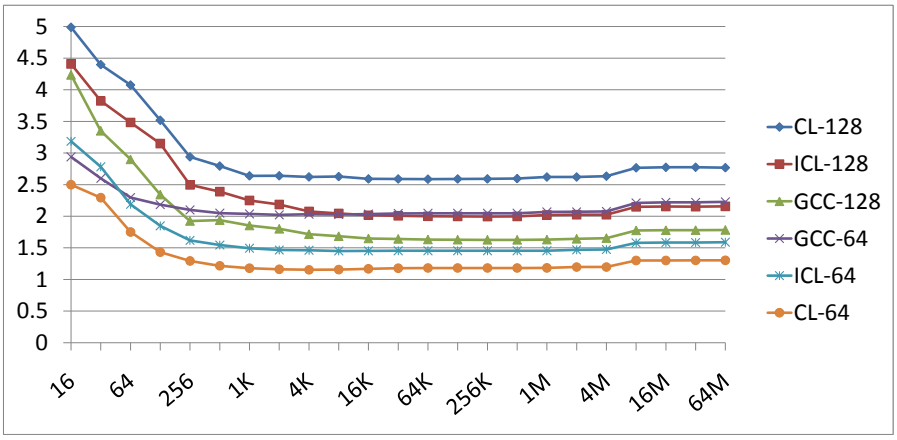
Table 4: Compiler comparison: Multiword/4 64-bit CRC

Input size	64	256	1K	4K	16K	64K	256K	1M
GCC/C++	2.30	2.10	2.04	2.03	2.03	2.05	2.05	2.07
ICL	2.19	1.62	1.49	1.46	1.45	1.45	1.46	1.46
CL	1.75	1.29	1.18	1.15	1.17	1.18	1.18	1.18
GCC/ASM	1.65	1.26	1.17	1.15	1.16	1.17	1.17	1.17

Table 5: Compiler comparison: Multiword/6 128-bit CRC

Input size	64	256	1K	4K	16K	64K	256K	1M
CL	4.08	2.94	2.64	2.62	2.59	2.59	2.59	2.62
ICL	3.48	2.50	2.25	2.08	2.02	2.00	2.00	2.02
GCC	2.90	1.93	1.85	1.72	1.65	1.63	1.63	1.63

Number of CPU cycles per byte. 128-bit CRC (CRC-128/IEEE polynomial) and 64-bit CRC (CRC-64-ECMA-182 polynomial) respectively. 64-bit platform, 64-bit reads. Warm data, warm tables.
Microsoft CL 15.00.30729 compiler was used with "-O2" flag. Intel ICL 11.10.051 and GCC 4.5.0 were used with "-O3" flag.

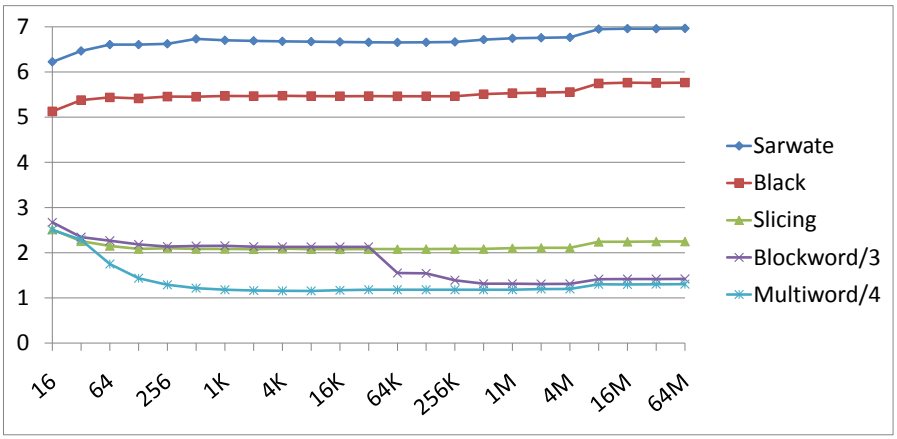


"Multiword/N" implements algorithm described in section 4.11 processing N data streams in parallel in interleaved manner.

Table 6: CRC-32 performance

Input size	64	256	1K	4K	16K	64K	256K	1M
Sarwate	6.61	6.62	6.70	6.68	6.67	6.66	6.67	6.75
Black	5.44	5.46	5.47	5.48	5.47	5.46	5.47	5.53
Slicing	2.15	2.10	2.09	2.09	2.08	2.08	2.08	2.10
Blockword/3	2.27	2.14	2.15	2.13	2.13	1.55	1.39	1.31
Multiword/4	1.75	1.29	1.18	1.16	1.17	1.18	1.18	1.18

Number of CPU cycles per byte. 32-bit CRC (CRC-32C polynomial), 64-bit platform, 64-bit tables, 64-bit reads (except Sarwate). Microsoft CL 15.00.30729 compiler. Warm data, warm tables.

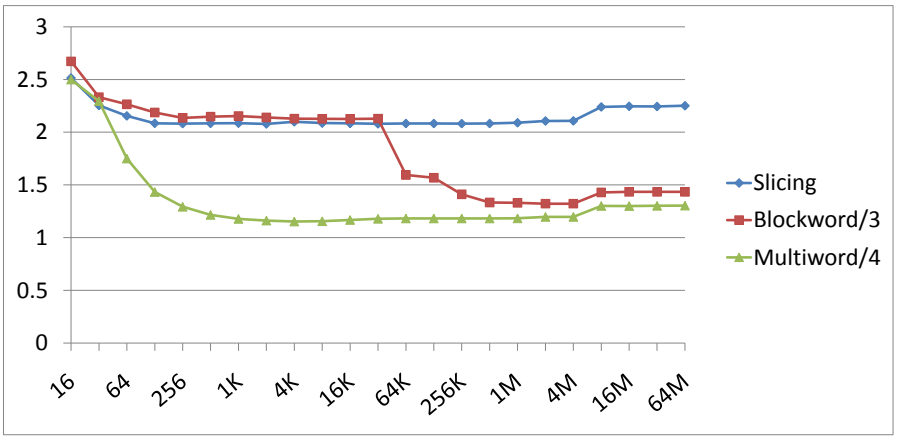


"Sarwate" implements the algorithm described in section 4.6.
"Black" implements the algorithm described in section 4.8.
"Slicing" implements the algorithm described in section 4.9.
"Blockword/3" implements the algorithm described in section 4.10 with 3 stripes of 15,376 bytes each.
"Multiword/4" implements the algorithm described in section 4.11 processing 4 data streams in parallel in interleaved manner.

Table 7: CRC-64 performance

Input size	64	256	1K	4K	16K	64K	256K	1M
Sarwate	6.61	6.62	6.70	6.68	6.67	6.65	6.66	6.75
Black	5.44	5.46	5.47	5.47	5.47	5.47	5.47	5.53
Slicing	2.16	2.08	2.09	2.10	2.08	2.08	2.08	2.09
Blockword/3	2.27	2.14	2.15	2.13	2.13	1.59	1.41	1.33
Multiword/4	1.75	1.29	1.18	1.15	1.17	1.18	1.18	1.18

Number of CPU cycles per byte. 64-bit CRC (CRC-64-ECMA-182 polynomial), 64-bit platform, 64-bit tables, 64-bit reads (except Sarwate). Microsoft CL 15.00.30729 compiler. Warm data, warm tables.



"Sarwate" implements the algorithm described in section 4.6.
"Black" implements the algorithm described in section 4.8.
"Slicing" implements the algorithm described in section 4.9.
"Blockword/3" implements the algorithm described in section 4.10 with 3 stripes of 15,376 bytes each.
"Multiword/4" implements the algorithm described in section 4.11 processing 4 data streams in parallel in interleaved manner.

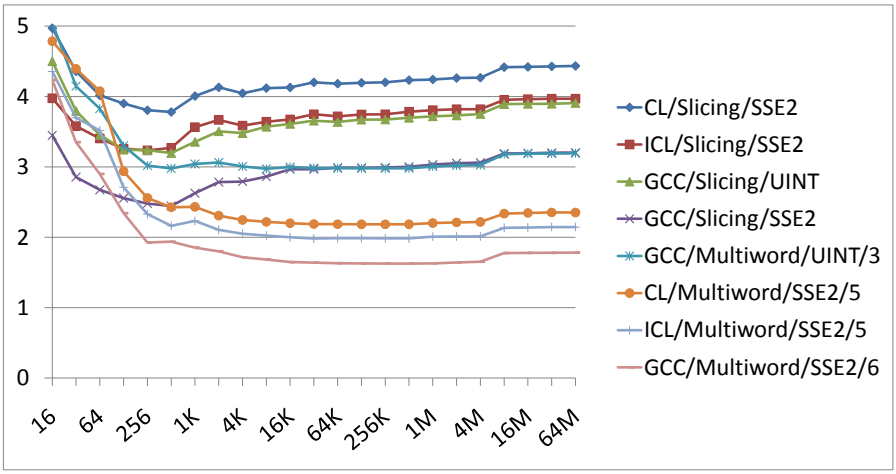
Table 8: CRC-128 performance: Slicing CRC

Input size	64	256	1K	4K	16K	64K	256K	1M
CL/SSE2	4.02	3.81	4.01	4.05	4.13	4.18	4.20	4.24
ICL/SSE2	3.40	3.24	3.57	3.59	3.68	3.72	3.75	3.81
GCC/UINT	3.45	3.24	3.36	3.48	3.61	3.64	3.67	3.72
GCC/SSE2	2.67	2.48	2.63	2.79	2.97	2.99	2.99	3.03

Table 9: CRC-128 performance: Multiword CRC

Input size	64	256	1K	4K	16K	64K	256K	1M
GCC/UINT/3	3.83	3.02	3.04	3.01	3.00	2.98	2.98	3.00
CL/SSE2/5	4.08	2.56	2.43	2.25	2.20	2.19	2.18	2.20
ICL/SSE2/5	3.52	2.33	2.23	2.05	2.00	1.99	1.99	2.01
GCC/SSE2/6	2.90	1.93	1.85	1.72	1.65	1.63	1.63	1.63

Number of CPU cycles per byte. 128-bit CRC (CRC-128/IEEE polynomial), 64-bit platform, 128-bit tables, 64-bit reads. Warm data, warm tables.
All compilers were tested using SSE2 intrinsics (/SSE2 variants). GCC was also tested using 128-bit integers provided by the compiler (GCC/UINT).



"Slicing" implements algorithm described in section 4.9.
"Multiword/N" implements algorithm described in section 4.11 processing N data streams in parallel in interleaved manner. The optimal (for given compiler) value of N was used.