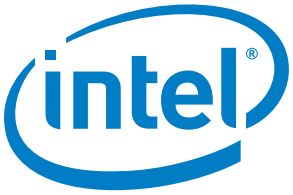


White Paper

Vinodh Gopal
Erdinc Ozturk
James Guilford
Wajdi Feghali

IA Architects
Intel Corporation

Choosing a CRC polynomial and associated method for Fast CRC Computation on Intel® Processors

August 2012

327889-001

Executive Summary

Cyclic Redundancy Check (CRC) codes are widely used for integrity checking of data in fields such as storage and networking. Fast and efficient methods of computing CRC on Intel® processors have been proposed for the fixed (degree-32) iSCSI polynomial, using the CRC32 instruction introduced in the Intel® Core™ i7 Processors. In addition, the PCLMULQDQ instruction can be used for fast CRC computation defined with any generic polynomial over the field GF(2) of degree smaller than 64. The performance of the PCLMULQDQ methods is independent of the polynomial chosen.

In this paper, we compare the iSCSI polynomial with other generic polynomials from a performance perspective and provide some guidelines on choosing the best polynomial for an application. We show that by using the iSCSI polynomial, we can achieve over 7X better CRC performance compared to any other polynomial.

Contents

Overview

A Cyclic Redundancy Check (CRC) is the remainder, or residue, of binary division of a potentially long message, by a CRC polynomial typically defined over the GF(2) field [7]. There is an ever-increasing need for very high-speed CRC computations on processors for end-to-end integrity checks [8][10][11]. Intel® processors can accelerate computation of the iSCSI CRC (that is, the degree 32 CRC polynomial 0x11EDC6F41) using the CRC32 instruction [1] introduced in the Intel® Core™ i7 Processor.

In the Intel® Core™ i7 Processors, the CRC32 instruction is implemented with a latency of 3 cycles and a throughput of 1 cycle. This implies that a traditional method to compute CRC of a buffer serially will achieve about a third of the optimal performance, whereas faster parallel implementations/methods have been proposed in [2], [4], [5] and [6].

The PCLMULQDQ instruction can be used for fast CRC computation defined over any generic polynomial over the field GF(2). The PCLMULQDQ methods [3] result in the same performance for any generic polynomial of degree smaller than 64.

Both PCLMULQDQ and CRC32 significantly increase performance over the best-known software implementations using a slice-by-8 table-lookup method [12].

This paper includes a brief description of the computations involved in computing an iSCSI CRC residue, and computing CRC residue using a different polynomial, followed by a performance analysis comparing the two polynomials. We target usages such as those typically found in Storage applications, where data block sizes are 512 bytes or larger powers-of-2, and compare functions that work on data buffers of these sizes.

Computing an iSCSI CRC

We can compute the iSCSI CRC of a data buffer with a simple serial approach using the CRC32 instruction; however this would not yield the optimal single-thread performance due to the latency of the instruction. This method however has advantages in code size and data-segment size over other methods, and gives good performance especially in the context of running two threads of execution on a single core. This method is also very efficient for small buffers. Figure 1 shows the main processing loop of the simple serial method.

Figure 1. Pseudo-code for simple serial approach using the CRC32 instruction

```
        mov     rax, crc_init       ;; rax = crc_init;

crc_loop:                           ;; while(len >0) {
        crc32   rax, [bufp]         ;;    crc32 of 64-bit data
        add     bufp, 8             ;;    buf +=8; (next quadword)
        sub     len, 8              ;;    len -= 8;
        jne     crc_loop            ;; }
```

More complex methods to parallelize the computation are derived from and explained in detail in the patents and pending applications [4], [5], [6] as well as the white-paper [2]. If we divide the buffer into three non-overlapping parts, we can perform three CRC calculations in parallel. Since the calculation for each part is independent of the calculations for the other parts, each set of three CRC32 instructions is no longer dependent and can execute at the throughput rate. Figure 2 shows the main processing loop of a 3-way (denoted x3) parallel approach. These parallel approaches become more efficient for larger buffers.

Figure 2. Pseudo-code for 3-way parallel (x3) approach using CRC32 instruction

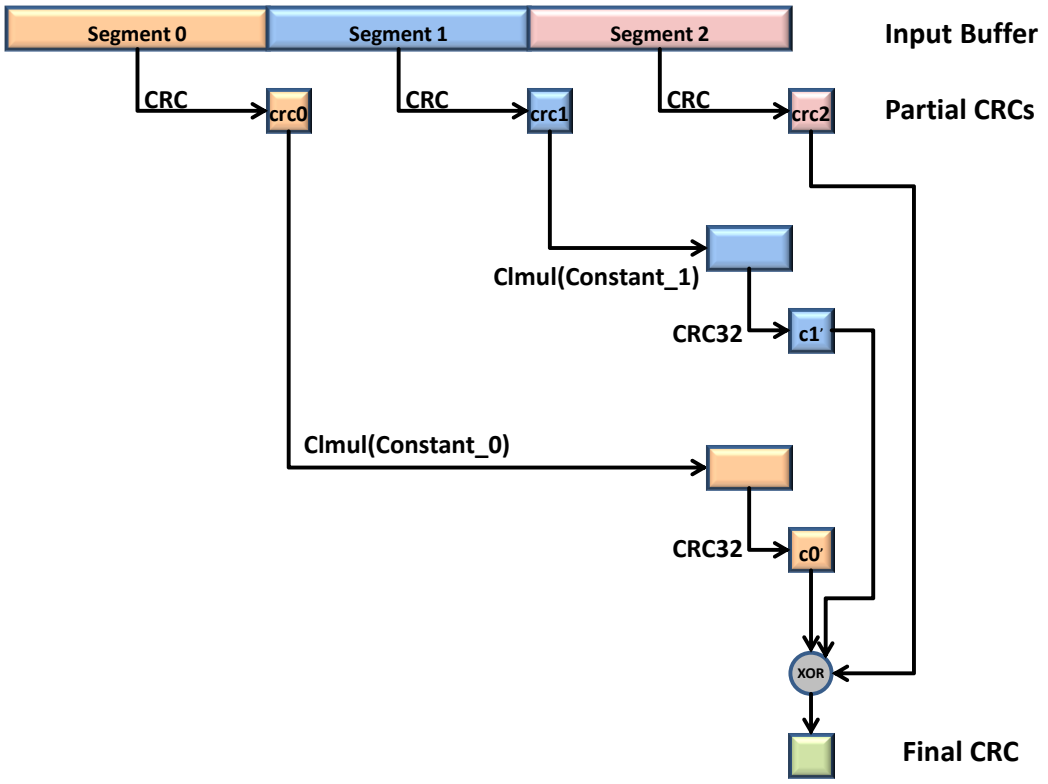
```
        mov     r12, BLOCKSIZE/8

_main_loop:
        crc32   rdx, [r9 + 0*BLOCKSIZE]     ; crc0
        crc32   rbx, [r9 + 1*BLOCKSIZE]     ; crc1
        crc32   rax, [r9 + 2*BLOCKSIZE]     ; crc2
        add     r9, 8
        dec     r12
        jne     _main_loop
```

At the end of this process, we have three separate CRC values. These need to be merged into a single value. This is the recombination "overhead" that is needed in order to process the buffer in parallel.

The basic idea is illustrated in Figure 3.

Figure 3. Recombining 3 CRC values in the parallel approach



The input data is divided into three segments, and the CRC is computed in parallel for each segment. Call the three values (from left to right) crc0, crc1, and crc2. The value crc2 is in the proper bit location, but the other two values need to be shifted into the correct bit position by multiplying crc0 and crc1 by appropriate constants. This multiplication "shifts" the value towards the right, but it also increases the size to 64 bits. For this reason, we cannot simply use the multiplication to "shift" crc0 and crc1 into the same bit position as crc2. Instead, we shift crc0 and crc1 into a position 32 bits to the left of crc2. Then we take the CRC of each of these shifted values. We can then XOR these two new CRC values to crc2 to get the final CRC value.

The multiplication described here is a carry-less one, and can be done using the PCLMULQDQ instruction if available. However, if there is no support for PCLMULQDQ, then the same computations can be achieved using pre-computed lookup tables. These techniques and implementations are further described in [2].

Computing a CRC for an Arbitrary Polynomial using PCLMULQDQ

The methods to compute a CRC for an arbitrary polynomial using PCLMULQDQ are described in detail in [3]. A brief overview is presented in this section. The method is based on folding most-significant portions of the data buffer by carry-less multiplication with pre-computed constants until a small number of bytes remain. At this point, a more irregular method of reduction is used to derive the final CRC, which is a somewhat fixed overhead cost incurred on buffers of any size, and therefore starts to become negligible when amortized over larger buffers.

Given a data buffer whose length >= 2*128 bits, we can reduce it by 128 bits iteratively. In this case, we simply work with two adjacent 128-bit blocks, using constants $\mathbf{K_1 = [x^{(128+64)} \bmod P(x)]}$ and $\mathbf{K_2 = [x^{128} \bmod P(x)]}$. The pseudo-code for folding one 128-bit block is shown in Figure 4.

Figure 4. Pseudo-code for Single Folding

```
Fold_by_1_loop:                    ; for(i=rax-1; i>0 ; i--) {
   movdqa xmm2, xmm1               ; xmm2 = xmm1;
   add rcx, 16                     ; buf += 16;
   movdqa xmm0, [rcx]              ; xmm0 = [buf];
   pshufb xmm0, [SHUF_MASK]        ; byte reflection if required
   pclmulqdq xmm1,  xmm3, 0x00     ; xmm1  = clmul(xmm1 , K2);
   pclmulqdq xmm2,  xmm3, 0x11     ; xmm2  = clmul(xmm2 , K1);
   pxor xmm0, xmm1                 ; xmm0 = xmm0 ⊕ xmm1;
   pxor xmm0, xmm2                 ; xmm0 = xmm0 ⊕ xmm2;
   movdqa xmm1, xmm0               ; xmm1 = xmm0;
   sub rax, 1
   jne Fold_by_1_loop              ; }

SHUF_MASK : DDQ 000102030405060708090A0B0C0D0E0Fh
xmm3      : K1|K2
```

The folding operation allows efficient utilization of the PCLMULQDQ instruction, which computes the carry-less multiplication of two 64-bit operands. Each folding requires 2 PCLMULQDQ instructions and 2 PXOR instructions as illustrated in the above pseudo-code.

In order to further maximize the efficiency of folding, it can be applied on different chunks of the data in a parallel manner. Thus for buffers that are suitably large (length >= 2*(8*128) bits), we can iteratively reduce by 8 folding operations for maximal efficiency, until we have a congruent buffer

that is much smaller. Each of the 8 folds is identical – we work on 128-bit data chunks using the constants $\mathbf{K_1 = [x^{(1024+64)} \bmod P(x)]}$ and $\mathbf{K_2 = [x^{1024} \bmod P(x)]}$, where the exponent is based on a separation of 8*128 = 1024.

Note that the code-size will increase due to a replication of the code sequence shown in the above pseudo-code to support 8 folds, if we want to optimize the performance on large buffers. This is due to the fact that all the data ideally needs to be in registers during the parallel folding operations, making it difficult to roll the code into an inner loop of count 8. In this study, we optimize for performance, and therefore have an unrolled (by 8) sequence of the above pseudo-code.

Furthermore, increasing parallelism of fold leaves a larger portion of the data buffer to be processed after the main loop of processing completes – e.g. for the case of 8 folds, we are left with 128 bytes. One can choose to reduce this data using a series of fold-by-1 steps iteratively, or use a progression of fold-by-4, fold-by-2, and fold-by-1 steps. These are then followed by a final reduction to 32 bits. In our implementation, we chose to optimize for performance at the expense of larger code size. However, since this processing occurs once at the end of the processing of a data buffer, if the application deals with large buffers, we could choose to have an implementation with compact code size for the final processing. These techniques and implementations are further described in [3].

Performance

Configuration and Testing

The performance was measured on the 2nd generation Intel® Core™ i7 Processor and represents the performance on a single core. For each run, we compute the CRC of the same data buffer 350,000 times. The RDTSC instruction is used to sample the processors time stamp clock before and after the run, to get the number of timestamps per run. We then perform 256 runs, discarding the 64 fastest and 64 slowest times, and use the mean of the remaining 128 values. This number is then adjusted to reflect that the time stamp clock may not run at the same rate as the processor core clock, due to Intel® Turbo Boost Technology and/or power saving features.

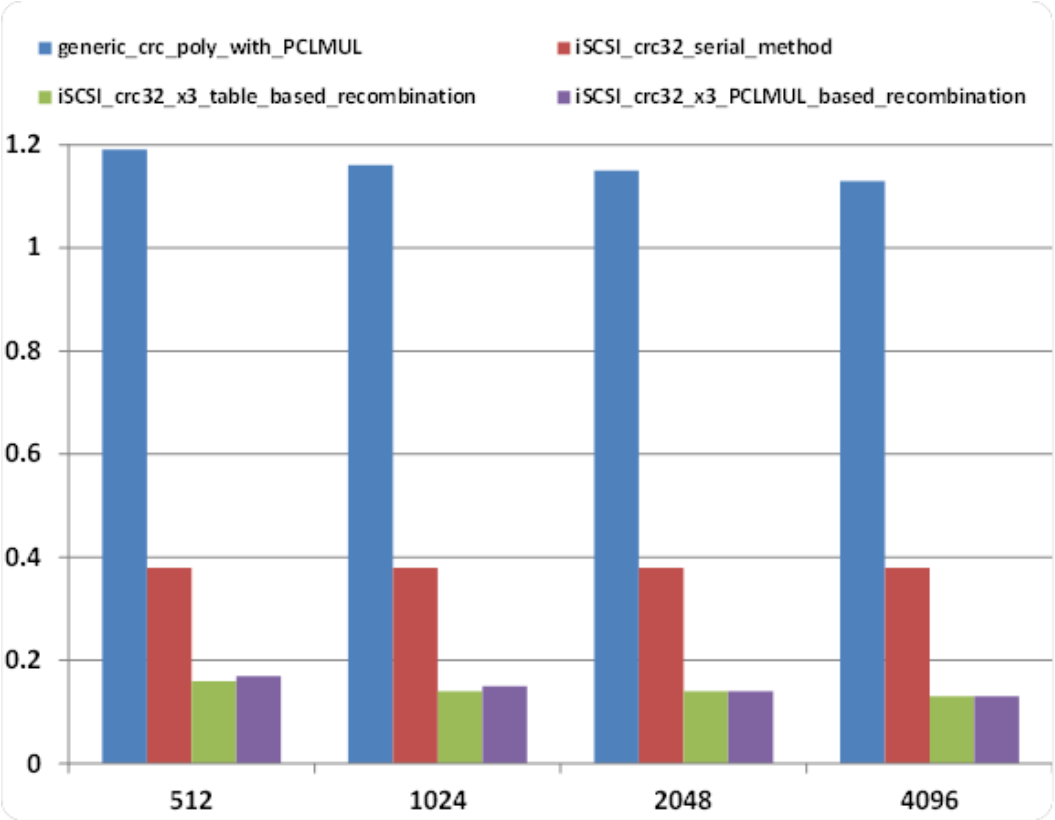
We show the performance of 4 different functions, each working on input data buffer sizes of {512, 1024, 2048, 4096} bytes in Figure 5. For each function, we also show the data segment size and the code size, which could become important for applications that are very sensitive to the usage of 1st level Instruction/Data Caches. Figure 6 is a graphic representation of the single-thread performance.

Figure 5. Performance characteristics of CRC functions[1]

	Input size (bytes)	Performance (Cycles/byte)		Data Segment Size (Bytes)	Code Size (Bytes)
		Single Thread	HT		
generic_crc_poly_with_PCLMUL	512	1.19	0.98	192	997
	1024	1.16	0.94		
	2048	1.15	0.93		
	4096	1.13	0.92		
iSCSI_crc32_serial_method	512	0.38	0.21	0	20
	1024	0.38	0.19		
	2048	0.38	0.19		
	4096	0.38	0.19		
iSCSI_crc32_x3_table_based_recombination	512	0.16	0.14	2,048	211
	1024	0.14	0.13		
	2048	0.14	0.13		
	4096	0.13	0.13		
iSCSI_crc32_x3_PCLMUL_based_recombination	512	0.17	0.15	8	140
	1024	0.15	0.14		
	2048	0.14	0.13		
	4096	0.13	0.13		

[1] Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.
For configuration and tests used, refer to the Performance section, paragraph 1. For more information go to: http://www.intel.com/performance.

Figure 6. CRC function performance (cycles/byte) for varying input size (bytes)[2]



It can be seen that iSCSI-based CRC functions have much better performance (over 7X[2]) than the function for generic polynomials computed using PCLMULQDQ. Note that the function for generic polynomials computed using PCLMULQDQ, is significantly faster than the best-known lookup-table based implementations in [12]. The table-based recombination for iSCSI CRC has the largest data segment size but is also a little faster than PCLMULQDQ-based recombination for smaller sizes. Overall, if PCLMULQDQ is available, it should be used for the recombination rather than tables.

The generic polynomial function using PCLMULQDQ has a larger code size relative to the iSCSI-based functions, but this can be reduced by ~25% with a small loss of performance for smaller buffers. It should be noted that even without any further code-size optimizations, the code-size is smaller than

[2] Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.
For configuration and tests used, refer to the Performance section, paragraph 1. For more information go to: http://www.intel.com/performance.

1024 bytes which is negligible compared to the size of the first-level instruction caches of these processors.

Different Data Buffer Sizes

We can extend the functions to handle data buffers of an arbitrary size with some increase in the code size. An interesting practical usage of processing irregular data sizes is where the CRC of a block of data is updated with a small amount of data at a later time in the application processing flow. For these usages, the iSCSI polynomial is easier to compute since we can apply the CRC32 instruction a few times using the new data bytes (e.g. 1 CRC32 instruction for each new quadword of data). The update process is more complex when we have a generic CRC polynomial and a small number of extra bytes to process; each quadword of data will require more than 1 associated PCLMULQDQ instruction, which will be more expensive than 1 CRC32 instruction. In very small updates with generic polynomials, it may even be preferable to use a table lookup method over PCLMULQDQ, depending on the application sensitivity to Data Cache usage.

Conclusions

We show that the CRC32 instruction-based iSCSI CRC functions have much better performance than the PCLMULQDQ instruction-based function for generic polynomials. We also see that the iSCSI CRC computation can achieve better performance while using a smaller footprint in terms of code and data segment sizes by using the PCLMULQDQ instruction for recombination. If an application/standard designer were to consider choosing between the iSCSI polynomial and another CRC polynomial, from a performance perspective on Intel® Processors, we would recommend the iSCSI CRC polynomial so that the CRC32 instruction can be used.

Acknowledgements

We thank Sean Gulley and Gilbert Wolrich for their substantial contributions. We also thank Robert Elliott from HP for his excellent feedback and suggestions.

References

[1] Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2A: Instruction Set Reference, A-M
http://www.intel.com/products/processor/manuals/

[2] Fast CRC Computation for iSCSI Polynomial Using CRC32 Instruction
http://download.intel.com/design/intarch/papers/323405.pdf

[3] Fast CRC Computation for generic polynomials using PCLMULQDQ Instruction http://download.intel.com/design/intarch/papers/323102.pdf

[4] Determining a Message Residue, Gopal et al. United States Patent 7,886,214

[5] Determining a Message Residue Gueron et al. United States Patent Application 20090019342

[6] Determining a Message Residue Gopal et al. United States Patent Application 20090158132

[7] "A Tutorial on CRC Computations", Ramabadran et al., Micro, IEEE, IEEE Computer Society, Aug. 1988, pp. 62-75.

[8] "High-Speed CRC Design for 10 Gbps applications", Lin et al., ISCAS 2006, IEEE, pp. 3177-3180.

[9] "Cyclic Redundancy Code (CRC) Polynomial Selection for Embedded Networks", Koopman et al., The International Conference on Dependable Systems and Networks, DSN- 2004, pp. 1-10.

[10] "Parallel CRC Realization", Campobello et al., IEEE Transactions on Computers, vol. 52, No. 10, Oct. 2003, Published by the IEEE Computer Society; pp. 1312-1319.

[11] "A Systematic Approach to Building High Performance Software- based CRC Generators", Kounavis et al., Proceedings of the 10th IEEE Symposium on Computers and Communications, ISCC 2005; pp. 855-862.

[12] "High Octane CRC Generation with the Intel Slicing-by-8 Algorithm", http://download.intel.com/technology/comms/perfnet/download/slicing-by-8.pdf

Authors

Vinodh Gopal, Erdinc Ozturk, Wajdi Feghali and James Guilford are IA Architects with the IAG Group at Intel Corporation.

Acronyms

IA Intel® Architecture