IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received December 25, 2018, accepted March 2, 2019, date of publication March 6, 2019, date of current version March 26, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2903304

Exploring Various Levels of Parallelism in High-Performance CRC Algorithms

MUCONG CHI, DAZHONG HE, AND JUN LIU, (Member, IEEE)
[1]Center for Data Science, Beijing University of Posts and Telecommunications, Beijing 100876, China
[2]School of Information and Communication Engineering, Beijing University of Posts and Telecommunications, Beijing 100876, China
[3]Beijing Laboratory of Advanced Information Networks, Beijing University of Posts and Telecommunications, Beijing 100876, China
[4]Beijing Key Laboratory of Network System Architecture and Convergence, Beijing University of Posts and Telecommunications, Beijing 100876, China

Corresponding author: Mucong Chi (chimucong@bupt.edu.cn)

This work was supported by the 111 Project of China under Grant B08004 and Grant B17007.

ABSTRACT Modern processors have increased the capabilities of instruction-level parallelism (ILP) and thread-level parallelism (TLP). These resources, however, typically exhibit poor utilization on conventional cyclic redundancy check (CRC) algorithms. In this paper, various levels of parallelism in high-performance CRC algorithms are investigated. The main idea of the proposed algorithms is to make full utilization of modern processors, from the perspective of both instruction-level and thread-level parallelism. First, a fine-grained algorithm executes the CRC computation in an interleaved manner, so that multiple independent data flows can be processed simultaneously. This algorithm allows instruction-level parallelism, which triples and doubles the performance of the existing slicing-by-4 and slicing-by-8 algorithms, respectively. Second, a coarse-grained algorithm can ideally deal with data in a parallel way by parallelizing a family of serial CRC generating algorithms. Therefore, this algorithm allows thread-level parallelism, which can make full use of multi-core computing capability. As a result, it achieves a speedup that is almost equal to the number of threads used. In addition, both fine-grained and coarse-grained algorithms can be applied together to achieve high throughput further. (This is an extended version of a paper that appeared at the 28th Annual International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC) in Montreal, QC, Canada, in 2017.)

INDEX TERMS Cyclic redundancy check (CRC), fault detection, parallel algorithms.

I. INTRODUCTION

Cyclic Redundancy Check (CRC) codes are widely used in digital networks and storage systems to detect the accidental alterations of data during its transmission or storage. For instance, the CRC-32 is employed by IEEE 802.3 (Ethernet) network standard [2] and compression programs, such as Bzip2 and Gzip, for data integrity checks. Besides, Stream Control Transmission Protocol (SCTP) and Internet Small Computer Systems Interface (iSCSI) adopt CRC-32C to detect errors. Due to Gaussian noise in transmission, hard drive malfunctions, and some other reasons, data may be modified accidentally. CRC demonstrates a good Hamming distance and has a good error detecting performance, which makes it suitable for error detection.

In this paper, we study the implementation of the CRC algorithm in software. While CRC generation can be efficiently implemented in hardware, software-based CRC generation is important as well. This is because many commercial servers that are widely used do not include dedicated CRC generation circuits. Moreover, many applications require the CRC generation to be implemented as effectively as possible.

In order to improve the performance of the CRC generation process, a number of software-based algorithms have been proposed in the past. Among these algorithms, the most commonly used today are table lookup based CRC [3], [4], parallel table lookup based CRC [5]–[7], symbolic simplification based CRC [8] and multi-processor based parallel CRC [9]–[11].

In recent years, computer architecture technology has progressed dramatically. Modern processors have increased the capabilities of instruction-level parallelism (ILP) and thread-level parallelism (TLP). However, most existing CRC generation algorithms do not make full use of these resources. Therefore, the purpose of this paper is to employ these new techniques to accelerate the process of well-known CRC

The associate editor coordinating the review of this manuscript and approving it for publication was Antonio J. Plaza.

codes that are widely used today. From the perspective of instruction-level and thread-level parallelism, two types of algorithms are proposed to make full use of modern processors. The contribution of this paper is summarized as follows:

- A fine-grained CRC algorithm is proposed, which divides the CRC computation into multiple independent data flows and performs the CRC process in an interleaved manner. This algorithm makes the existing table-based CRC algorithms suitable for processing multiple independent data flows simultaneously, which allows instruction-level parallelism. We apply this fine-grained interleaved method to the well-known Slicing-by-4 and Slicing-by-8 algorithms. As a result, it triples and doubles the performance of Slicing-by-4 and Slicing-by-8 algorithms, respectively.
- A coarse-grained CRC algorithm is proposed for efficient multi-core processor computation. This algorithm can ideally parallelize conventional CRC algorithms, which allows thread-level parallelism. The evaluation results demonstrate that the speedup achieved by this algorithm is almost equal to the number of threads used.
- Since these two algorithms optimize the CRC generation from different perspectives, they can be combined to improve the performance further.

The most significant innovation of our work is that the proposed algorithms can be achieved with a variable number of interleaved data flows for instruction-level parallelism, and also a variable number of threads for thread-level parallelism.

The rest of this paper is organized as follows: we present related work in Section II. An overview of the CRC generation is provided in Section III. We present our fine-grained and coarse-grained algorithms in Sections IV and V, respectively. In Section VI, we provide experiments and results. And then we conclude in Section VII.

II. RELATED WORK

The CRC algorithm was first proposed by Peterson and Brown [12]. Traditionally, CRC calculation is achieved with a linear-feedback shift register (LFSR). In the original LFSR method, a simple shift register is used, and one input bit is processed at a time [13]. For an l-bit message, we need l times of operations to perform the CRC calculation. However, this scheme significantly limits the throughput of the CRC generation. To improve the performance, a number of previous works have focused on efficient implementation of CRC computation from a software or hardware perspective.

Among these algorithms, the tea-leaf reader algorithm [3] is one of the first to use lookup tables. In order to support the generation of CRC-32, five 256-byte tables are employed. For each byte of the input, this algorithm performs five table lookups, five exclusive OR (XOR) operations, and four shift operations to generate the CRC-32 code of the message. Compared to the original bit-wise approach aforementioned, this byte-wise algorithm is 19 times faster. To optimize the tea-leaf reader algorithm, Sarwate [4] proposed a classic table-based CRC algorithm, which reads 8 bits at a time and a single table of 256 32-bit entries is used to calculate the CRC value. This algorithm performs a single table lookup, two XOR operations, a shift, and an AND operation for each byte of the message. Table lookup based CRC can be viewed as a parallel method of LFSR by combining several bit-wise iterative steps into one, obtaining high throughput.

The Sarwate algorithm was proposed at a time when XOR operations of most processors could only be performed between 8-bit bytes. Since then, computer architecture technology has made significant progress, and arithmetic operations between 32- or 64-bit words can be performed efficiently. At the same time, processors are equipped with large cache units that can be accessed in a few clock cycles. Moderate-size tables (such as 1 KB) are suitable to be stored in cache units, which significantly reduce the access latency. In order to improve the performance further, Kounavis and Berry [7] proposed a novel lookup-based framework. In this framework, the number of bits that read at a time can be arbitrarily large. Based on this framework, two algorithms are designed to run on the Intel IA32 processor. First, the Slicing-by-4 doubles the performance of the Sarwate algorithm. It reads 32 bits at a time, and four 1-KB lookup tables are required. Second, the Slicing-by-8 triples the performance, using eight 1-KB lookup tables. By slicing the bits that read each time into smaller terms, table lookups can be simultaneously performed on these terms to accelerate the process. In this paper, the slicing technology and lookup tables are adopted as well. But unlike other table-based algorithms, our fine-grained parallel CRC algorithm performs table lookups in an interleaved manner, which can improve the performance further.

From another point of view, symbolic evaluation can be leveraged to simplify the CRC generation. Based on this concept, a CRC generation algorithm without lookup tables was proposed by Engdahl and Chung [8]. No table lookups are needed in this proposed methodology, which reduces the memory accesses. In some cases, their algorithm is more efficient than conventional table lookup methods. However, in other cases, this solution is even slower and provides no benefit. Besides, for some complex polynomials, such as CRC-32K, it is hard to simplify.

The algorithms as mentioned above are all fine-grained parallel methods. In recent years, coarse-grained parallel CRC algorithms based on multi-processor architectures are proposed [9], [11]. These algorithms can be easily combined with fine-grained CRC algorithms to improve performance further. Do *et al.* [11] proposed a parallel CRC algorithm called "N-byte RCC (Repetition of Computation and Combination)." In this algorithm, $n \times N$ bytes of the message are processed at each iteration. If the message is not a multiple of $n \times N$, extra zero bits are inserted at the front of itself. At each iteration, n numbers of N-byte blocks are computed by n processors simultaneously. The Slicing-by-4 algorithm [7] is leveraged to generate partial CRC values of each N-byte block. When CRC computations by n processors

are completed, a combination operation, which is through table lookups, is performed to combine the partial CRCs from n processors. The CRC value combined previously is XORed with the following $n \times N$ bytes of the message, and the iteration is repeated until there are no more bytes left. Notice that, this algorithm induces folk-join overhead for every $n \times N$ bytes. In order to minimize the synchronization overhead, Cho *et al.* [9] proposed a block-interleaving based parallel CRC algorithm. This algorithm is similar to [11], except that a propagation step, instead of the combination step mentioned in [11], is performed so that the dependency problem is resolved. In this manner, partial CRC values can be combined only at the last stage. Since algorithms proposed in [9] and [11] split the message into blocks of fixed size, more synchronization or propagation steps are required as the size of the message grows. Different from these two algorithms, our coarse-grained parallel algorithm divides the message into only n blocks. Furthermore, no synchronization or propagation steps are required during the CRC computation. As a result, the overhead of thread synchronization is minimized.

From a hardware perspective, some kinds of parallel LFSR architectures for efficient CRC generation have been proposed. A tree structure of gates has been designed in [14] to optimize the cascaded logic parts. In order to reduce the complexity of parallel CRC circuits, a state-space transformation has been proposed in [15] and [16]. A novel parallel LFSR based on IIR filter topology [17]–[19] uses a pipeline technique to improve the hardware efficiency. Besides, table-based algorithms have been explained in detail in [20]–[22]. Recently, a new method to construct the transformation matrix used in state-space transformation has been proposed in [23]. Besides, a novel approach of applying iterative decoding technology has been introduced in [24]. Abdulnabi and Ahmed [25] have presented an FPGA-based implementation of Slicing-by-16.

Since software-based CRC algorithms can be widely applied to commercial servers, this paper mainly focuses on the software implementation of parallel CRC algorithms.

III. THE CRC GENERATION PROCESS

A. DESCRIPTION

Cyclic Redundancy Check (CRC) is a well-known error-detecting code, which has the capability to detect whether the message transmitted over a noisy channel has been corrupted. This technique can also be applied to storage systems to check the integrity of the data stored on the hard drive.

For a message M with $N = |M|$ bits, it can be denoted in binary form

$$M = [m_{n-1} m_{n-2} \cdots m_0] \tag{1}$$

where $m_i \in \{0, 1\}$. In particular, m_{n-1} and m_0 are referred to as the most significant bit (MSB) and the least significant bit (LSB), which are corresponding to the first and the last bits of the message M, respectively. Mathematically, the message

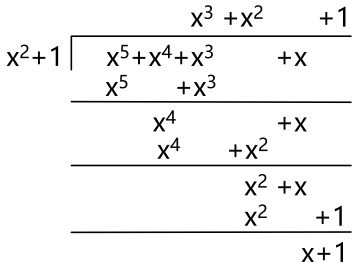


FIGURE 1. Division of polynomial using modulo-2 arithmetic.

M can be viewed as a polynomial of degree $(N-1)$

$$M(x) = \sum_{i=0}^{N-1} m_i x^i \tag{2}$$

For a specific CRC, there is a predetermined polynomial $G(x)$ of degree $J = \deg(G(x))$, which is called a generator polynomial. The CRC value of the message M is defined as the reminder of division $M(x) \cdot x^J$ by $G(x)$. Namely, it is

$$CRC[M(x)] = M(x) \cdot x^J \bmod G(x) \tag{3}$$

Since coefficients of polynomials are defined in $GF(2)$, all arithmetic operations on polynomials are performed using modulo-2 arithmetic. For example, addition:

$$(x^2 + x + 1) + (x + 1) = x^2 + 2x + 2 = x^2$$

subtraction:

$$(x^2 + 1) - (x + 1) = x^2 - x = x^2 + x$$

multiplication:

$$(x + 1) \cdot (x + 1) = x^2 + 2x + 1 = x^2 + 1$$

In this way, additions and subtractions of coefficients are equal to the XOR logical operation.

The division operation of the polynomial is a little bit complicated. Figure 1 demonstrates a division example. In this example, $x^2 + 1$ is the divisor, whereas $x^5 + x^4 + x^3 + x$ is the dividend. At the first step, the polynomial $(x^2+1) \cdot x^3 = x^5 + x^3$ is found, which is a multiple of the divisor $x^2 + 1$ and shares the same degree of the dividend $x^5 + x^4 + x^3 + x$. It is subtracted from $x^5 + x^4 + x^3 + x$, obtaining $x^4 + x$. The next step goes in the same way, which subtracts $(x^2 + 1) \cdot x^2 = x^4 + x^2$ from $x^4 + x$, and $x^2 + x$ is obtained. The process is repeated until the degree of the difference is less than 2 (the degree of $x^2 + 1$). At last, we get the quotient $x^3 + x^2 + 1$ and remainder $x + 1$.

B. THE SARWATE ALGORITHM

The Sarwate algorithm is one of the most well-known CRC algorithms, which is based on a lookup table. This algorithm generates a 32-bit CRC value, which is shown in Figure 2. At first, the variable *crc* is set to a specific number (e.g., 0xFFFFFFFF for CRC32c). In each iteration, this algorithm reads one byte (8 bits) from the data buffer and performs an

```
1 crc = INIT_VALUE;
2 while(p_buf < p_end){
3   crc = table[(crc ^ *p_buf++) & 0x000000FF] ^
        (crc>>8);
4 }
5 return crc ^ FINAL_VALUE;
```

FIGURE 2. The Sarwate algorithm.

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

FIGURE 3. The Slicing-by-4 algorithm.

XOR operation between this byte and the least significant byte of the variable *crc*. Then, a table lookup operation is performed, which uses the number produced in the previous step as an index. The most significant bytes of the *crc*, which are obtained by shifting the *crc* by 8 bits to the right, are XORed with the value returned from the table lookup. The *crc* is updated to the result from this last XOR operation and is used in the next iteration. The loop continues until all bits of the input buffer are processed. Finally, the *crc* is XORed with a given number that depends on the standard (e.g., 0xFFFFFFFF for CRC32c), and this value is returned as the CRC value of the original input buffer.

This algorithm is considered to be rather slow nowadays: First of all, reading data byte by byte is not an efficient data access approach on modern processors. Secondly, modern processors are equipped with multiple ALUs, and 3-4 instructions may be executed simultaneously to handle independent data flows. However, the Sarwate algorithm processes data byte by byte and contains only one data flow. Most instructions take the result from the previous instruction as inputs, which may lead to CPU stalls owing to the result propagation delays (e.g., accessing the L1 cache requires $\sim$ 4 CPU cycles of latency).

C. THE SLICING-BY-4 ALGORITHM

In order to speed up the Sarwate algorithm, a straightforward approach is to read multiple bytes at a time. However, the size of the lookup table increases exponentially with the bits read at a time, which may make the lookup table unable to fit into the L1 cache (e.g., 32 KB). For example, a table of 2^{16} entries is required to achieve acceleration by reading 16 bits at a time.

The Slicing-by-4 algorithm [7] reads 32 bits at a time, while the memory requirement is optimized to occupy reasonable cache footprints. Figure 3 demonstrates the implementation of the Slicing-by-4 algorithm. In each iteration,

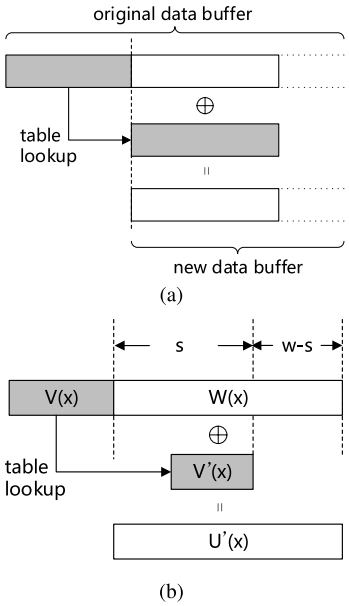


FIGURE 4. Demonstration of the folding method. (a) Folding the most significant chunk into an adjacent chunk of the data buffer, (b) A general method of folding a data chunk across *s* bits.

a 32-bit chunk is read from the data buffer and is split into four slices. Then four table lookups are performed, which use these four slices as indexes. This algorithm is called "Slicing-by-4" because it can perform four table lookups at a time.

IV. FINE-GRAINED PARALLEL CRC ALGORITHM

The Slicing-by-4 algorithm reduces the number of data access operations and allows instruction-level parallelism: multiple table lookup operations may be processed in parallel. As a result, it doubles the performance of the Sarwate algorithm.

However, there is still room for further performance improvement. At the beginning of each iteration, all table lookup operations contend for a single source of data (variable *crc* in Figure 3). At the end of each iteration, the value XORed from all results of the lookup operations is written to a single destination (variable *crc* again). Further improvement may be achieved by processing multiple independent data flows in an interleaved manner. The benefit of an interleaved approach is that when the execution of one data flow path is stalled, the processor may switch execution to another one so that the CPU utilization is improved. In this section, we propose an interleaved Slicing-by-4 algorithm to improve the performance further.

A. THE FOLDING METHOD

Commonly, in order to perform the CRC computation, a few constants are precomputed, which are stored in a lookup table. And then these constants are applied to fold the most significant chunks of the data buffer repeatedly. In each iteration, a new buffer is created, which is small in length but congruent to the original one. Figure 4 (a) demonstrates a simple example, where the most significant chunk of the data buffer is folded into the adjacent chunk.

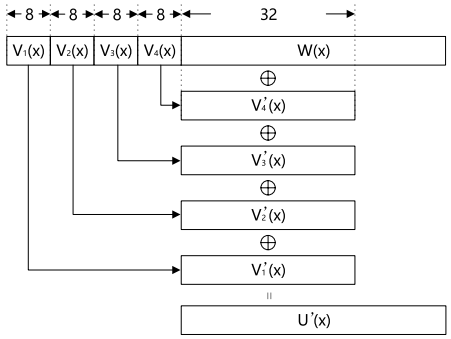


FIGURE 5. Folding method of the Slicing-by-4 algorithm.

A more generalized method is introduced in this paper, which folds the most significant chunk to an arbitrary position in the data buffer. This method reduces the data buffer into a smaller one, keeping the CRC value of the buffer unchanged. For a message U, we write

$$U = [V : W] \tag{4}$$

where V and W are the most significant v-bit and remaining w-bit chunks of U, respectively. Mathematically, it can be expressed as

$$U(x) = V(x) \cdot x^w \oplus W(x) \tag{5}$$

In Figure 4 (b), we show the folding method. In order to fold the V into the position that is s-bit ($s \leq w$) after itself, we compute

$$V'(x) = V(x) \cdot x^s \bmod G(x) \tag{6}$$

and XOR it with $W(x)$ at that position. Therefore, the reduced data buffer is

$$\begin{aligned} U'(x) &= V'(x) \cdot x^{w-s} \oplus W(x) \\ &= \big(V(x) \cdot x^s \bmod G(x)\big) \cdot x^{w-s} \oplus W(x) \end{aligned} \tag{7}$$

We compute the remainder of U' divided by G

$$\begin{aligned} &U'(x) \bmod G(x) \\ &= \big(\big(V(x) \cdot x^s \bmod G(x)\big) \cdot x^{w-s} \oplus W(x)\big) \bmod G(x) \\ &= \big(V(x) \cdot x^s \cdot x^{w-s} \oplus W(x)\big) \bmod G(x) \\ &= \big(V(x) \cdot x^w \oplus W(x)\big) \bmod G(x) \end{aligned} \tag{8}$$

which is equal to

$$U(x) \bmod G(x) = \big(V(x) \cdot x^w \oplus W(x)\big) \bmod G(x) \tag{9}$$

At this point, we have proven that the general folding method generates a new data buffer that is congruent to the original one but smaller in length.

The Slicing-by-4 algorithm can also be viewed as performing four folding operations at a time, and Figure 5 demonstrates the details. In each iteration, four bytes are folded, which are denoted as V_1, V_2, V_3 and V_4. Since that these bytes are in the different positions, different offset values are used to fold them, which are 56, 48, 40 and 32 bits, respectively. The value V'_i ($1 \leq i \leq 4$) to be XORed with $W(x)$ is given by

$$V'_i(x) = V_i(x) \cdot x^{32+(4-i)\times 8} \bmod G(x) \tag{10}$$

As a whole, a 32-bit chunk of data is folded into an adjacent chunk of the same size. Notice that the length of each data chunk V_i ($1 \leq i \leq 4$) is 8 bits. Therefore four lookup tables of 256 32-bit entries each are required to compute V'_i.

To illustrate the main points, our discussions are focused on 32-bit CRCs in the following sections of the paper. However, these points can also be applied to CRCs of other sizes.

B. THE INTERLEAVED SLICING-BY-4 ALGORITHM

Suppose that the input message M consists of T groups M_t, and each group M_t consists of N W-bit words, that is,

$$M(x) = \sum_{t=0}^{T-1} M_t(x) \cdot x^{tNW} \tag{11}$$

$$M_t(x) = \sum_{n=0}^{N-1} m_{t,n}(x) \cdot x^{nW} \tag{12}$$

where $m_{t,n}$ is the $(N-n)$-th W-bit word of the M_t. The message M can be represented as

$$\begin{aligned} M(x) &= \sum_{t=0}^{T-1} M_t(x) \cdot x^{tNW} \\ &= \sum_{t=0}^{T-1} \left(\sum_{n=0}^{N-1} m_{t,n}(x) \cdot x^{nW} \right) \cdot x^{tNW} \\ &= \sum_{n=0}^{N-1} \left(\sum_{t=0}^{T-1} m_{t,n}(x) \cdot x^{tNW} \right) \cdot x^{nW} \\ &= \sum_{n=0}^{N-1} I_n(x) \cdot x^{nW} \end{aligned} \tag{13}$$

where

$$I_n(x) = \sum_{t=0}^{T-1} m_{t,n}(x) \cdot x^{tNW} \tag{14}$$

It means that, I_n is composed of a W-bit word $m_{T-1,n}$ followed by $(N-1)W$ zero bits, then a W-bit word $m_{T-2,n}$ followed by $(N-1)W$ zero bits, ..., ending up with $m_{0,n}$. Figure 6 shows the relationship between M and I_n.

The CRC value of the message M can be represented as

$$\begin{aligned} \mathrm{CRC}[M(x)] &= \mathrm{CRC}\left[\sum_{n=0}^{N-1} I_n(x) \cdot x^{nW} \right] \\ &= \sum_{n=0}^{N-1} \mathrm{CRC}[I_n(x) \cdot x^{nW}] \end{aligned} \tag{15}$$

which means that the CRC computation of message M can be split into N independent data flows. Therefore, the folding method can be applied to compute CRC values of each $I_n(x) \cdot$

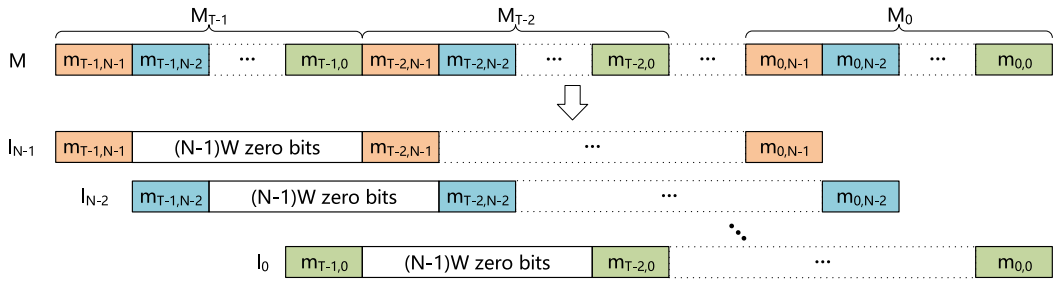


FIGURE 6. Splitting a message in an interleaved manner.

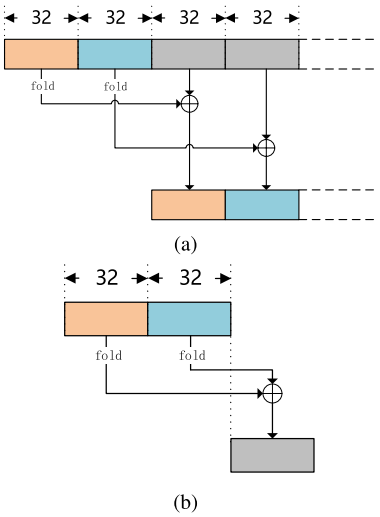


FIGURE 7. Folding scheme of the interleaved Slicing-by-4 algorithm. (a) Folding two 32-bit buffer chunks at a time, (b) Final reduction.

x^{nW} in an interleaved manner, and the contentions on a single data source and destination are eliminated.

We take the implementation of the interleaved Slicing-by-4 algorithm with two independent data flows as an example. As Figure 7 (a) shows, when the message buffer is large (length $\geq 4 \times 32$ bits), it is iteratively reduced by folding operations in an interleaved manner. For each data flow, the operation of folding 32 bits is similar to the Slicing-by-4 algorithm, except that different lookup tables are adopted. This folding operation continues until there are only 64 bits remaining in the buffer. At last, a final reduction is performed to combine the results of two data flows into one, which is demonstrated in Figure 7 (b). Notice that, the length of the message is assumed to be divisible by 64 (8 bytes). In other cases, several steps of the Sarwate algorithm are firstly performed to make the length divisible.

For a more specific description, the C implementation of the interleaved Slicing-by-4 algorithm is shown in Figure 8. In the folding loop (lines 3-18), table lookup and XOR operations are performed in an interleaved manner. Unlike the Slicing-by-4 algorithm, which folds 32 bits into the adjacent one, this algorithm folds each chunk across 32 bits. Therefore, the offset values used for the lookup tables are 88, 80, 72 and

```
1  crc1 = INIT_VALUE;
2  crc2 = 0;
3  while (length > 8) {
4    crc1 ^= *(uint32_t *) p_buf;
5    crc2 ^= *(uint32_t *) (p_buf + 4);
6    x1 = table_88[crc1 & 0x000000FF];
7    x2 = table_80[(crc1 >> 8) & 0x000000FF];
8    x3 = table_72[(crc1 >> 16) & 0x000000FF];
9    x4 = table_64[(crc1 >> 24) & 0x000000FF];
10   y1 = table_88[crc2 & 0x000000FF];
11   y2 = table_80[(crc2 >> 8) & 0x000000FF];
12   y3 = table_72[(crc2 >> 16) & 0x000000FF];
13   y4 = table_64[(crc2 >> 24) & 0x000000FF];
14   crc1 = x1 ^ x2 ^ x3 ^ x4;
15   crc2 = y1 ^ y2 ^ y3 ^ y4;
16   p_buf += 8;
17   length -= 8;
18 }
   // Final Reduction
19 crc1 ^= *(uint32_t *) p_buf;
20 crc2 ^= *(uint32_t *) (p_buf + 4);
21 x1 = table_88[crc1 & 0x000000FF];
22 x2 = table_80[(crc1 >> 8) & 0x000000FF];
23 x3 = table_72[(crc1 >> 16) & 0x000000FF];
24 x4 = table_64[(crc1 >> 24) & 0x000000FF];
25 y1 = table_56[crc2 & 0x000000FF];
26 y2 = table_48[(crc2 >> 8) & 0x000000FF];
27 y3 = table_40[(crc2 >> 16) & 0x000000FF];
28 y4 = table_32[(crc2 >> 24) & 0x000000FF];
29 return x1 ^ x2 ^ x3 ^ x4 ^
          y1 ^ y2 ^ y3 ^ y4 ^ FINAL_VALUE;
```

FIGURE 8. The interleaved Slicing-by-4 algorithm.

64 bits, respectively. The loop repeats until there are 64 bits (8 bytes) left. In the last step (lines 19-29), a final reduction is performed, combining two CRC values into one.

V. COARSE-GRAINED PARALLEL CRC ALGORITHM

In this section, we introduce a coarse-grained parallel CRC algorithm that is suitable for multi-core processors. This algorithm can extend the parallelism to thread level.

For a given parallel factor P, we split the input message M into P blocks of approximately the same size, that is

$$M = [M_{P-1} : M_{P-2} : \cdots : M_0] \tag{16}$$

Mathematically,

$$M(x) = \sum_{p=0}^{P-1} M_p(x) \cdot x^{K_p} \tag{17}$$

$$K_p = \begin{cases} \sum_{i=0}^{p-1} L_i, & 1 \le i \le p-1 \\ 0, & i = 0 \end{cases} \tag{18}$$

where L_i is the length of M_i. Therefore,

$$\begin{aligned} &CRC[M(x)] \\ &= CRC\left[\sum_{p=0}^{P-1} M_p(x) \cdot x^{K_p}\right] \\ &= \sum_{p=0}^{P-1} CRC[M_p(x) \cdot x^{K_p}] \\ &= \sum_{p=0}^{P-1} CRC[M_p(x)] \cdot (x^{K_p} \bmod G(x)) \bmod G(x) \end{aligned} \tag{19}$$

For the sake of conciseness, we define coefficients by

$$\beta_p = x^{K_p} \bmod G(x), 0 \le p \le P-1 \tag{20}$$

and modular multiplication

$$A(x) \otimes B(x) = A(x) \cdot B(x) \bmod G(x) \tag{21}$$

Equation (19) can be simplified to

$$CRC[M(x)] = \sum_{p=0}^{P-1} CRC[M_p(x)] \otimes \beta_p \tag{22}$$

Therefore, we design the coarse-grained parallel CRC algorithm as follows:

1) Split the input message M into P blocks.
2) Compute the CRC values of these P blocks in parallel by P threads. The computation parts may be carried on existing fine-grained CRC algorithms.
3) The CRC values are modular multiplied by the corresponding coefficients β_p respectively and XOR all the products to obtain the CRC value of the original message.

In what follows we describe the implementation of modular multiplication and computation of coefficients β_p in detail.

A. MODULAR MULTIPLICATION

Modular multiplication takes A and B as inputs, which are 32-bit words. A can be represented as

$$A = [a_{31}a_{30} \cdots a_0] \tag{23}$$

where $a_i \in \{0, 1\}$. Therefore

$$A(x) = \sum_{i=0}^{31} a_i x^i \tag{24}$$

and

$$\begin{aligned} A(x) \otimes B(x) &= A(x) \cdot B(x) \bmod G(x) \\ &= \left(\sum_{i=0}^{31} a_i x^i\right) \cdot B(x) \bmod G(x) \\ &= \left(\sum_{i=0}^{31} a_i x^i \cdot B(x)\right) \bmod G(x) \\ &= \sum_{i=0}^{31} a_i x^i \cdot B(x) \bmod G(x) \\ &= \sum_{i \in R} x^i \cdot B(x) \bmod G(x) \end{aligned} \tag{25}$$

where $R = \{i \mid a_i = 1,\ 0 \le i \le 31\}$.

```
1  product = 0;
2  for (i = 0; i < 32; ++i) {
3    if ((A & 0x00000001 << (31 - i)) != 0) {
4      product = product ^ B;
5    }
6    if ((B & 0x00000001) != 0) {
7      B = (B >> 1) ^ POLY;
8    } else {
9      B = B >> 1;
10   }
11 }
12 return product;
```

FIGURE 9. The modular multiplication algorithm.

The modular multiplication algorithm is shown in Figure 9, which is analogous to the Galois Field multiplication. During each iteration of the loop (lines 2-11), the variable i traverses from 0 to 31, and the variable B corresponds to $x^i \cdot B(x) \bmod G(x)$ in Equation (25). For each a_i that equals to 1 (i.e., the test expression in line 3 is true), the corresponding $x^i \cdot B(x) \bmod G(x)$ is XORed with *product*, and *produce* is updated with the new value (line 4). When the loop ends, the variable *product* is exactly the modular product of A and B. Notice that, the variable POLY (line 7) is the binary form of the CRC generator (e.g., 0x82F63B78 for CRC32c).

B. EFFECTIVE COMPUTATION OF COEFFICIENTS

In order to compute the coefficients effectively, a precomputed lookup table LUT is defined by

$$LUT[i] = x^{2^i} \bmod G(x),\ 0 \le i \le 63 \tag{26}$$

Since that

$$\begin{aligned} &LUT[i+1] \\ &= x^{2^{i+1}} \bmod G(x) \\ &= x^{2^i} \cdot x^{2^i} \bmod G(x) \\ &= \left(x^{2^i} \bmod G(x)\right) \cdot \left(x^{2^i} \bmod G(x)\right) \bmod G(x) \\ &= \left(x^{2^i} \bmod G(x)\right) \otimes \left(x^{2^i} \bmod G(x)\right) \\ &= LUT[i] \otimes LUT[i] \end{aligned} \tag{27}$$

elements of LUT can be calculated iteratively based on the previous value.

We define the modular exponentiation as

$$\text{MOD\_EXP}(L) = x^L \bmod G(x) \tag{28}$$

```
1 acc = 0x00000001;
2 for (i = 0; i < 64; ++i) {
3   if ((L & 0x00000001 << i) != 0) {
4     acc = MOD_MUL(LUT[i], acc);
5   }
6 }
7 return acc;
```

FIGURE 10. The modular exponentiation algorithm.

Often it is necessary for very large values of L. Therefore, it is represented as a 64-bit word, that is,

$$L = \sum_{i=0}^{63} l_i \cdot 2^i \tag{29}$$

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

$$\begin{aligned}
\text{MOD_EXP}(L) &= x^{\sum_{i=0}^{63} l_i \cdot 2^i} \bmod G(x) \\
&= \left(\prod_{i=0}^{63} x^{l_i \cdot 2^i} \right) \bmod G(x) \\
&= \left(\prod_{i \in S} x^{2^i} \right) \bmod G(x) \\
&= \left(\prod_{i \in S} x^{2^i} \bmod G(x) \right) \bmod G(x) \\
&= \left(\prod_{i \in S} LUT[i] \right) \bmod G(x) \\
&= \bigotimes_{i \in S} LUT[i]
\end{aligned} \tag{30}$$

where $S = \{i \mid l_i = 1,\ 0 \le l \le 63\}$. Similar to the Knuth's square-and-multiply algorithm [26], which is a general method for fast computation of large positive integer power of a number, the MOD_EXP is used for computation of large positive modular power of a polynomial. Equation (30) implies that the modular exponentiation can be calculated by performing a series of modular multiplication, the multipliers of which are elements of LUT.

The implementation of modular exponentiation is demonstrated in Figure 10. Initially, the accumulator *acc* is set to 0x00000001 (corresponding to polynomial 1, which is the multiplicative identity element). During each iteration of the loop (lines 2-6), the variable i traverses from 0 to 63. For each l_i that equals to 1 (i.e., the test expression in line 3 is true), the corresponding $LUT[i]$ is modularly multiplied by *acc*, and *acc* is set to the product (line 4, MOD_MUL stands for modular multiplication). When the loop ends, variable *acc* is exactly the result of modular exponentiation. The coefficients β_p can be calculated by the modular exponentiation algorithm:

$$\beta_p = \text{MOD\_EXP}(K_p)$$

VI. EXPERIMENTS AND RESULTS

In order to evaluate the performance, experiments both in instruction and thread levels are carried out. The experiments are performed on an Intel Xeon E5-2680 processor, which is equipped with 12 cores operating at 2.5 GHz. This processor has a 32-KB first-level (L1) data cache per core and a 256-KB second-level (L2) cache per core. The operating system is Linux running with a 3.10.0 kernel, and the GNU C Compiler 4.8.5 is used. The CPUID and RDTSC instructions are used to measure the consumed clock cycles.

All evaluations are performed using random input data over various message lengths. Lookup tables are aligned on a 256-byte boundary to eliminate the non-aligned penalty. In order to minimize performance variations, we perform 2000 runs for each message length, discarding the 500 fastest and 500 slowest times, and the mean of the remaining 1000 values is used. Besides, the evaluations are performed at high priority to minimize the interference of the operating system and other applications.

In what follows, we present our experiments of fine-grained and coarse-grained parallelism respectively.

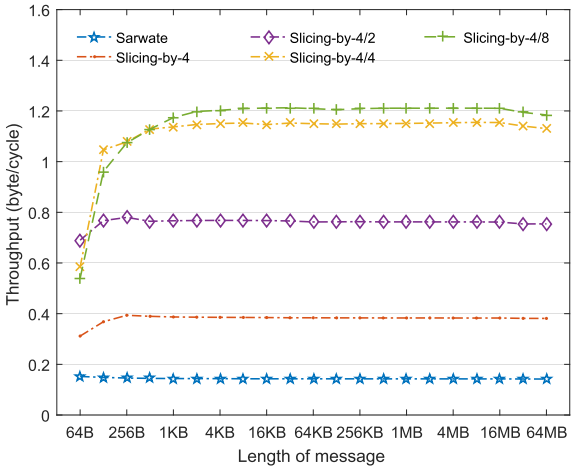
A. FINE-GRAINED PARALLELISM

We implement N-way interleaved versions of Slicing-by-4 and Slicing-by-8, which are denoted as Slicing-by-4/N and Slicing-by-8/N respectively. We compare the performance of our algorithms with the Slicing-by-4 and Slicing-by-8 algorithms, and the Sarwate algorithm is used as a baseline.

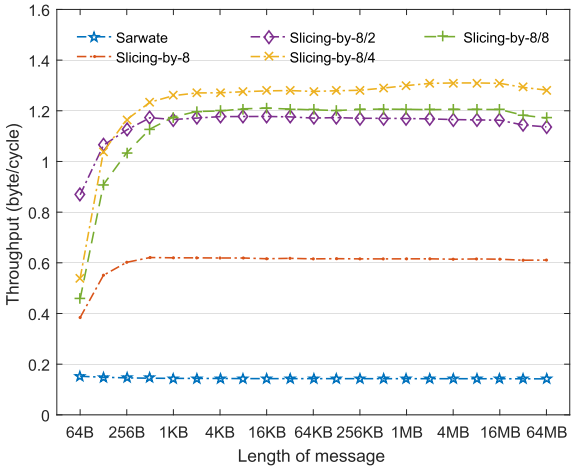
Figure 11 (a) shows the performance of the Sarwate, Slicing-by-4, and interleaved versions of Slicing-by-4 algorithm. We observe that, for table-based algorithms, it is important to place lookup tables in the cache unit, especially when the message length is small. For example, the performance of the Slicing-by-4 algorithm is 0.31 byte/cycle when the message length is 64 bytes. For large message length, the performance of the Slicing-by-4 algorithm is stable at 0.38 byte/cycle. It is explained by the fact that when the lookup tables are first accessed, the latency is significant (commonly a few hundred CPU cycles) since tables are fetched from the main memory. As the size of the message grows, the impact of the access latency is not that high because after some bytes of the message are processed, lookup tables may be found in the cache unit and only a few CPU cycles are needed. This phenomenon is more obvious on Slicing-by-4/4 and Slicing-by-4/8 algorithms than other algorithms since more lookup tables are fetched from the main memory during the computation. In contrast, there is no lookup table needed for the Sarwate algorithm. As a result, the performance of it is independent of the values of message lengths.

When the message length is larger than 128 bytes, the proposed algorithms achieve good performance. For example, the performance of the Slicing-by-4/2 is 0.76 byte/cycle, which achieves 1.98× speed-up compared to the Slicing-by-4 algorithm. Furthermore, the Slicing-by-4/4 and Slicing-by-4/8 algorithms achieve 2.99× and 3.15× speed-up respectively.

Another observation we make is that the performance difference between the Slicing-by-4/4 and Slicing-by-4/8 is not



(a)



(b)

FIGURE 11. Performance of the fine-grained parallel CRC algorithms. (a) Slicing-by-4, (b) Slicing-by-8. The *N*-way interleaved versions of Slicing-by-4 and Slicing-by-8 are denoted as Slicing-by-4/*N* and Slicing-by-8/*N* respectively. In each plot, the performance of the Sarwate algorithm is used as a baseline.

as noticeable as expected. Since that each core of CPU has a fixed number of ALUs, the performance does not increase linearly with the number of interleaved data flows.

In Figure 11 (b), we study the performance of Slicing-by-8 algorithm [7] and its interleaved implementations. The Slicing-by-8 algorithm is similar to the Slicing-by-4 algorithm, except that the former reads 8 bytes at a time, instead of reading 4 bytes like the latter. This algorithm uses 8 lookup tables, and the total space required is 8 KB, which is two times the space requirement of Slicing-by-4. As shown in Figure 11 (b), the interleaved versions of the Slicing-by-8 algorithm improve the performance significantly. The speed-ups are 1.91×, 2.10×, and 1.96× for Slicing-by-8/2, Slicing-by-8/4, and Slicing-by-8/8, respectively.

To our surprise, the Slicing-by-8/4 achieves better performance than Slicing-by-8/8 does. In the main loop of the Slicing-by-8/*N* algorithm, 2*N* 32-bit words are needs to hold the folding results. Therefore, the Slicing-by-8/*N* algorithm

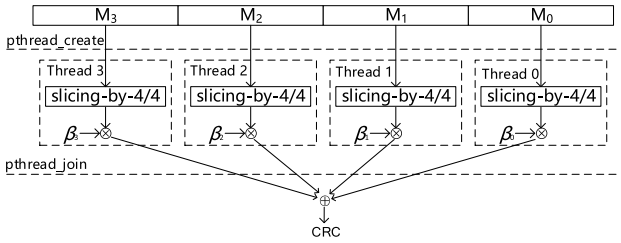


FIGURE 12. The Slicing-by-4/4×4 algorithm.

uses at least 2*N* variables. Too many variables may cause spilling of registers [27] since processors have a limited number of registers and not all variables can be assigned to registers. Commonly, a spilled variable is stored in stack space, which has a much slower accessing speed than a variable in a register. Therefore, the number of interleaved data flows should matter: a) too few may cause underutilization of instruction-level parallelism of the processor; b) too many may cause spilling of registers, which leads to performance degradation instead.

B. COARSE-GRAINED PARALLELISM

We implement the coarse-grained parallelism based on POSIX Threads, which is a parallel execution model for multi-core systems. Four algorithms, Slicing-by-4, Slicing-by-8, Slicing-by-4/4, and Slicing-by-8/4 are chosen to be parallelized. The corresponding coarse-grained parallel algorithms are denoted as Slicing-by-4×*P*, Slicing-by-8×*P*, Slicing-by-4/4×*P*, and Slicing-by-8/4×*P*, respectively, where *P* is the parallel factor. Three different parallel factors, 2, 4, and 8 are achieved for each algorithm. For a given parallel factor *P*, the message is split into *P* blocks, and the pthread_create function is invoked to create *P* threads. For each thread, the corresponding algorithm is employed for partial CRC computation. Then the pthread_join function is invoked in the main thread to wait for partial CRC computations on these *P* blocks to finish. At last, all partial CRC values are recombined to obtain the CRC value of the original message. For example, Figure 12 illustrates the Slicing-by-4/4×4 algorithm which executes four Slicing-by-4/4 processes simultaneously.

Figure 13 demonstrates the performance of the coarse-grained parallel algorithm proposed in this paper. For a message with small length, the overhead of thread folk-join and CRC recombination accounts for the majority, which makes the coarse-grained parallel algorithm not as efficient as expected, and even worse than the sequential algorithm in some cases. Fortunately, the efficiency of the proposed algorithm is significantly improved when the message reaches a certain size.

We take parallelizing Slicing-by-4 as an example. As shown in Figure 13 (a), when the message length is smaller than 32 KB, the performance of Slicing-by-4×2 and Slicing-by-4×4 is worse than that of Slicing-by-4, while the performance of Slicing-by-4×8 does not exceed that of

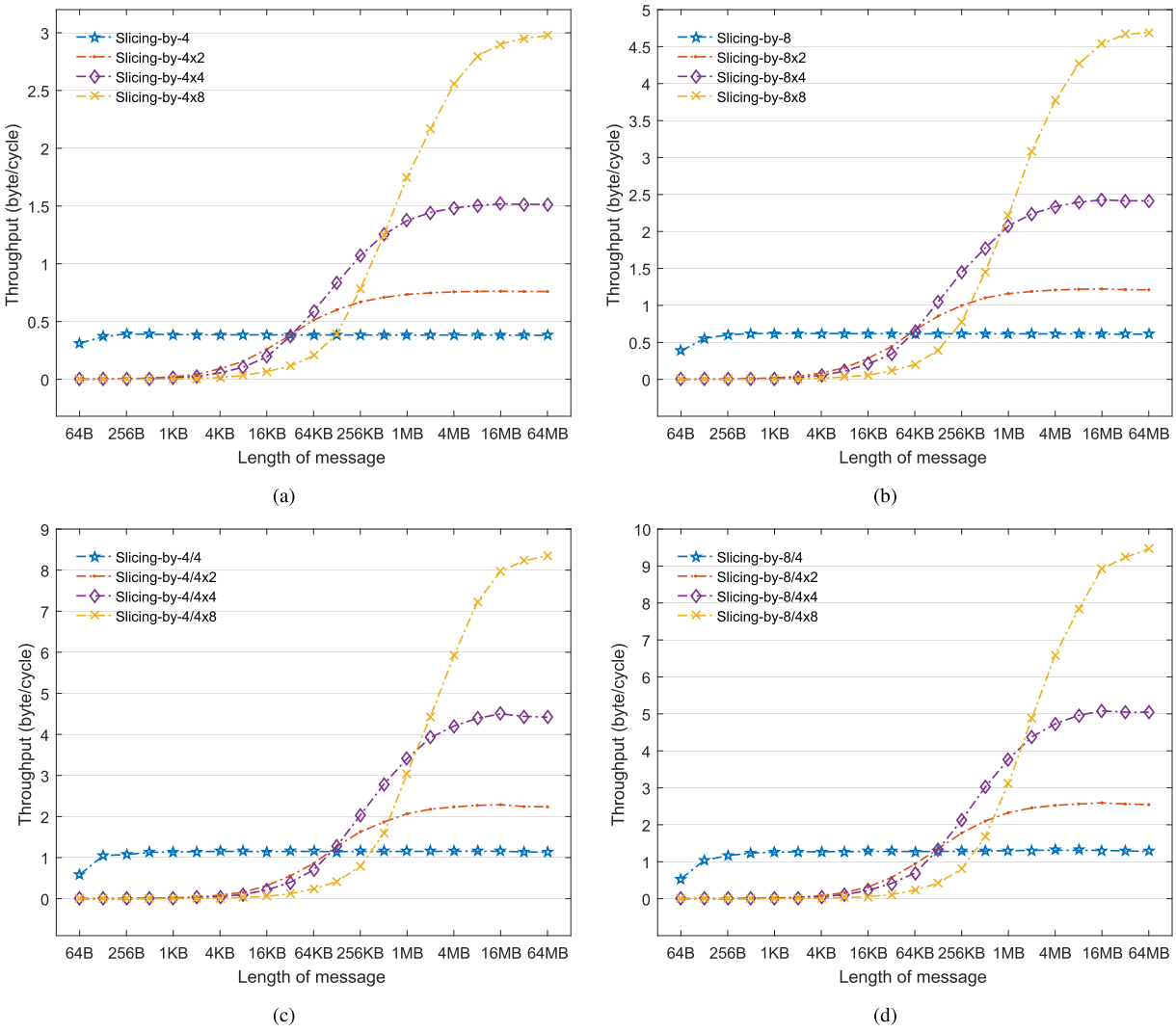


FIGURE 13. Performance of the coarse-grained parallel CRC algorithms. (a) Slicing-by-4, (b) Slicing-by-8, (c) Slicing-by-4/4, (d) Slicing-by-8/4. The corresponding coarse-grained parallel algorithms are denoted as Slicing-by-4×*P*, Slicing-by-8×*P*, Slicing-by-4/4×*P*, and Slicing-by-8/4×*P*, respectively, where *P* is the parallel factor.

Slicing-by-4 until the message length is greater than 128 KB. It demonstrates the fact that, for a message with small length, the number of CPU cycles spent doing CRC computation is small and there is a fairly significant overhead involved with creating and managing multiple threads. However, the efficiency of the coarse-grained parallel algorithm increases when the message length grows. As a result, for long enough messages, it achieves 1.98×, 3.97× and 7.80× speed-ups for parallel factors of 2, 4 and 8, respectively. Figures 13 (b), (c), and (d) show the performance of parallelizing Slicing-by-8, Slicing-by-4/4, and Slicing-by-8/4, respectively. Similar conclusions can be drawn from the results of these experiments.

Table 1 shows the thread synchronization overhead under various parallel factors. For each parallel factor P, P threads are created, and these threads terminate immediately without performing CRC computations. The main thread waits until all these P threads complete their executions. The number of CPU cycles consumed by this process is measured as the overhead of thread synchronization.

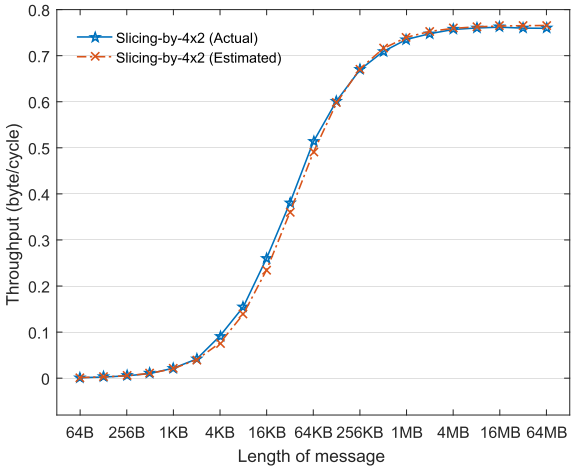
TABLE 1. Overhead of thread synchronization under various parallel factors.

Parallel factor	2	4	8
Overhead (cycles)	48,234	78,106	300,776

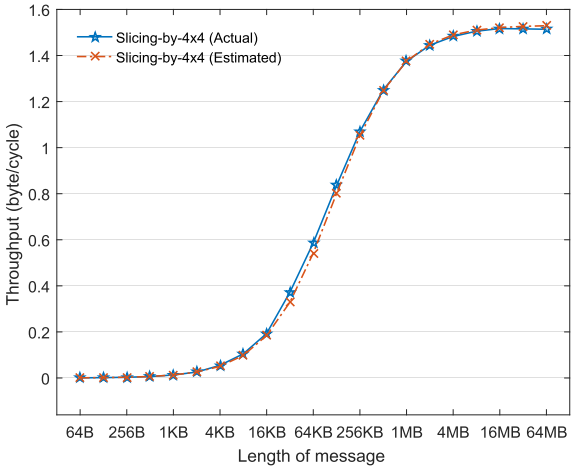
It is assumed that the thread synchronization overhead is constant, regardless of the length of the message. Therefore, the throughput of the coarse-grained parallel algorithm can be estimated by the following equation

$$\widehat{T_P} = \frac{L}{\frac{L}{T \cdot P} + O_P} \tag{31}$$

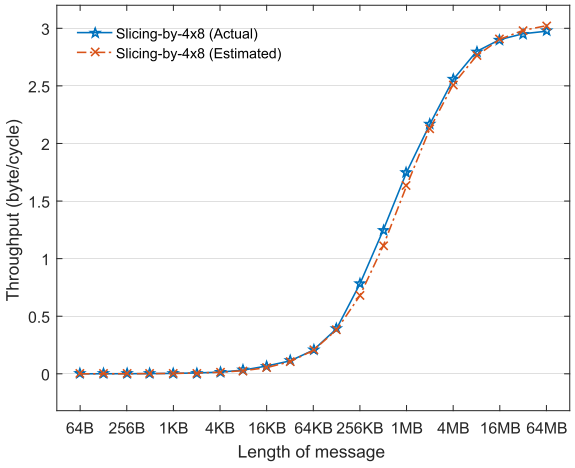
where $\widehat{T_P}$ is the estimated throughput of the parallel algorithm under parallel factor P, L is the length of the message, T is the



(a)



(b)



(c)

FIGURE 14. Actual and estimated performance of Slicing-by-4×*P*. (a) Slicing-by-4×2, (b) Slicing-by-4×4, (c) Slicing-by-4×8.

throughput of the corresponding single-thread algorithm, and O_P is the synchronization overhead under parallel factor P. Due to space limitations, we only show the result of Slicing-by-4×P in Figure 14. We observe that the estimated curve and the actual curve are highly coincident in each plot. The estimation results of other coarse-grained are similar to Slicing-by-4×P. From the experimental result, we have confirmed that the thread synchronization overhead is the main factor that causes the coarse-grained parallel algorithm to have low performance when the message length is small.

VII. CONCLUSION

In this paper, we proposed two algorithms to parallelizing CRC computation at both fine-grained and coarse-grained levels. First, the fine-grained parallel algorithm, which utilizes the instruction-level parallelism, triples the performance of the Slicing-by-4 algorithm and doubles the performance of the Slicing-by-8 algorithm. Second, in order to implement thread-level parallelism, the coarse-grained parallel algorithm is designed, which achieves a speedup almost equal to the number of threads employed. Furthermore, both fine-grained and coarse-grained algorithm can be applied together to obtain a high throughput. Compared to the Slicing-by-4 and Slicing-by-8 algorithms, the Slicing-by-4/4×8 and Slicing-by-8/4×8 algorithms achieve 21.53× and 15.66× speed-ups, respectively. The most important contribution of this paper is that the proposed algorithms can be achieved with an arbitrary number of interleaved data flows and also an arbitrary number of threads. As a result, the instruction-level and thread-level parallelism of the modern processor can be fully utilized.

The coarse-grained parallel algorithm achieves high performance for long enough messages. However, in cases where the message length is small, the performance drops dramatically. This situation will be improved in our future work.

REFERENCES

[1] M. Chi and J. Liu, "VACA: A high-performance variable-length adaptive CRC algorithm," in *Proc. IEEE 28th Ann. Int. Symp. Pers., Indoor, Mobile Radio Commun. (PIMRC)*, Oct. 2017, pp. 1–6.

[2] *IEEE Standards for Local Area Networks: Carrier Sense Multiple Access With Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications*, IEEE Standards 802.3-1985, 1985.

[3] G. Griffiths and G. C. Stones, "The tea-leaf reader algorithm: An efficient implementation of CRC-16 and CRC-32," *Commun. ACM*, vol. 30, no. 7, pp. 617–620, Jul. 1987.

[4] D. V. Sarwate, "Computation of cyclic redundancy checks via table look-up," *Commun. ACM*, vol. 31, no. 8, pp. 1008–1014, Aug. 1988.

[5] F. Braun and M. Waldvogel, "Fast incremental CRC updates for IP over ATM networks," in *Proc. IEEE Workshop High Perform. Switching Routing*, May 2001, pp. 48–52.

[6] Y. Sun and M. S. Kim, "A pipelined CRC calculation using lookup tables," in *Proc. 7th IEEE Consum. Commun. Netw. Conf.*, Jan. 2010, pp. 1–2.

[7] M. E. Kounavis and F. L. Berry, "Novel table lookup-based algorithms for high-performance CRC generation," *IEEE Trans. Comput.*, vol. 57, no. 11, pp. 1550–1560, Nov. 2008.

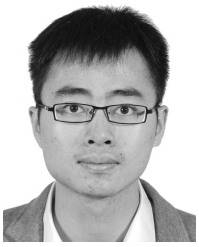
[8] J. R. Engdahl and D. Chung, "Fast parallel CRC implementation in software," in *Proc. 14th Int. Conf. Control, Autom. Syst. (ICCAS)*, Oct. 2014, pp. 546–550.

[9] J. Cho, B. Sung, and W. Sung, "Block-interleaving based parallel CRC computation for multi-processor systems," in *Proc. IEEE Workshop Signal Process. Syst.*, Oct. 2010, pp. 311–316.

[10] H. M. Ji and E. Killian, "Fast parallel CRC algorithm and implementation on a configurable processor," in *Proc. IEEE Int. Conf. Commun. Conf. (ICC)*, Apr./May 2002, pp. 1813–1817.

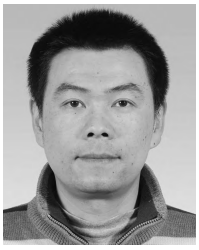
[11] Y. Do, S.-R. Yoon, T. Kim, K. E. Pyun, and S.-C. Park, "High-speed parallel architecture for software-based CRC," in *Proc. 5th IEEE Consum. Commun. Netw. Conf.*, Jan. 2008, pp. 74–78.

[12] W. W. Peterson and D. T. Brown, "Cyclic codes for error detection," *Proc. IRE*, vol. 49, no. 1, pp. 228–235, Jan. 1961.
[13] T. V. Ramabadran and S. S. Gaitonde, "A tutorial on CRC computations," *IEEE Micro*, vol. 8, no. 4, pp. 62–75, Aug. 1988.
[14] T. Pei and C. Zukowski, "High-speed parallel CRC circuits in VLSI," *IEEE Trans. Commun.*, vol. 40, no. 4, pp. 653–657, Apr. 1992.
[15] J. H. Derby, "High-speed CRC computation using state-space transformations," in *Proc. IEEE Global Telecommun. Conf. (GLOBECOM)*, Nov. 2001, pp. 166–170.
[16] C. Kennedy and A. Reyhani-Masoleh, "High-speed CRC computations using improved state-space transformations," in *Proc. IEEE Int. Conf. Electro/Inf. Technol.*, Jun. 2009, pp. 9–14.
[17] M. Ayinala and K. K. Parhi, "High-speed parallel architectures for linear feedback shift registers," *IEEE Trans. Signal Process.*, vol. 59, no. 9, pp. 4459–4469, Sep. 2011.
[18] J. Jung, H. Yoo, Y. Lee, and I.-C. Park, "Efficient parallel architecture for linear feedback shift registers," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 62, no. 11, pp. 1068–1072, Nov. 2015.
[19] R. A. C. Varma and Y. V. Apparao, "High-Throughput VLSI Architectures for CRC-16 computation in VLSI signal processing," in *Microelectronics, Electromagnetics and Telecommunications*, J. Anguera, S. C. Satapathy, V. Bhateja, and K. Sunitha, Eds. Singapore: Springer, 2018, pp. 23–32.
[20] Y. Sun and M. S. Kim, "A table-based algorithm for pipelined CRC calculation," in *Proc. IEEE Int. Conf. Commun.*, May 2010, pp. 1–5.
[21] Y. Huo, X. Li, W. Wang, and D. Liu, "High performance table-based architecture for parallel CRC calculation," in *Proc. 21st IEEE Int. Workshop Local Metropolitan Area Netw.*, Apr. 2015, pp. 1–6.
[22] Bajarangbali and P. A. Anand, "Design of high speed CRC algorithm for ethernet on FPGA using reduced lookup table algorithm," in *Proc. IEEE Annu. India Conf. (INDICON)*, Dec. 2016, pp. 1–6.
[23] G. Hu, J. Sha, and Z. Wang, "High-speed parallel LFSR architectures based on improved state-space transformations," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 25, no. 3, pp. 1159–1163, Mar. 2017.
[24] E. Tsimbalo, X. Fafoutis, and R. J. Piechocki, "CRC error correction in IoT applications," *IEEE Trans. Ind. Informat.*, vol. 13, no. 1, pp. 361–369, Feb. 2017.
[25] M. S. Abdulnabi and H. Ahmed, "Design of efficient cyclic redundancy check-32 using FPGA," in *Proc. Int. Conf. Comput., Control, Elect., Electron. Eng. (ICCCEEE)*, Aug. 2018, pp. 1–5.
[26] D. E. Knuth, "The art of computer programming," *Seminumerical Algorithms*, vol. 2, 3rd ed. Reading, MA, USA: Addison-Wesley, 1997.
[27] C. W. Fraser and D. R. Hanson, "Simple register spilling in a retargetable compiler," *Softw., Pract. Exper.*, vol. 22, no. 1, pp. 85–99, 1992.



MUCONG CHI received the B.E. degree from Beijing Jiao Tong University, in 2014. He is currently pursuing the Ph.D. degree with the Beijing University of Posts and Telecommunications (BUPT). His research interests include big data analysis and big data storage.

DAZHONG HE received the B.E. and M.E. degrees in information engineering from the Beijing University of Posts and Telecommunications (BUPT), in 1994 and 1997, respectively. Since 1997, he has been with BUPT. His current research interests include networks' traffic research, big data analysis, and hardware-based deep learning.



JUN LIU received the B.E. and Ph.D. degrees from the Department of Information Engineering, Beijing University of Posts and Telecommunications (BUPT), in 1998 and 2003, respectively. He is currently the Director of the Center for Data Science, BUPT. His research interests include networks' traffic monitoring, telecom big data analysis, and streaming data algorithm.