Low-Cost and Programmable CRC Implementation Based on FPGA

Huan Liu, Zhiliang Qiu, Weitao Pan, *Member, IEEE*, Jun Li, Ling Zheng, and Ya Gao

***Abstract*—Cyclic redundancy check (CRC) is a well-known error detection code that is widely used in Ethernet, PCIe, and other transmission protocols. The existing FPGA-based implementation solutions encounter the problem of excessive resource utilization in high-performance scenarios. The padding zeros problem and the introduction of programmability further exacerbate this problem. In this brief, the stride-by-5 algorithm is proposed to achieve the optimal utilization of FPGA resources. The pipelining go back algorithm is proposed to solve the padding zeros problem. The method of reprogramming by HWICAP is proposed to realize programmability with small and constant resource utilization. The experimental results show that the resource utilization of the proposed non-segmented architecture is 80.7%-87.5% and 25.1%-46.2% lower than that of two state-of-the-art FPGA-based CRC implementations, and the proposed segmented architecture has lower resource utilization, by 81.7%-85.9% and 2.9%-20.8%, than two state-of-the-art architectures. Furthermore, throughput and programmability are guaranteed. The source code has been made available on GitHub.**

***Index Terms*—Cyclic redundancy check, FPGA, low cost, programmable, HWICAP.**

I. Introduction

AS THE throughput of networks is on a constant rise, increasingly more packet processing tasks are being offloaded to the FPGA-based SmartNIC, including the generation and verification of cyclic redundancy check (CRC). Technologies such as 400G and the coming multi-terabit Ethernet demand faster CRC calculations [5], and the implementation of high-performance CRC calculations based on FPGAs must meet three requirements: 1) *Reduce the parallelization cost.* The end of Dennard scaling [2] results in a bottleneck for improving the frequency of integrated circuits, and higher throughput means wider buses in chips. The slicing-by-4 and slicing-by-8 algorithms are proposed for parallel processing in [3] and are suitable for CPUs but not optimal for FPGAs [4]. 2) *Solve the padding zeros problem.* Parallelization means that the final word of a transaction is composed of valid bytes along with padding zeros. The number of padding zeros is uncertain, and CRC calculations using the complete final word would cause an erroneous result, which is called the padding zeros problem. Reference [5] illustrate a state-of-the-art scheme for solving this problem. The tables for the final word are organized in the manner of a pipeline, and each pipeline step corresponds to one layer of a binary search tree. An $O(n)$ resource utilization is introduced. 3) *Maintain programmability.* A programmable implementation of the CRC algorithm can achieve better reusability; thus, a wide range of applications can be supported without circuit modification. The demand can be found in iSCSI [6] and P4 [7]. A specific circuit architecture is used to guarantee programmability [8], but it is not suitable for FPGAs. Reference [4] is a state-of-the-art scheme that is suitable for FPGAs, but it requires a complex configuration circuit that leads to a large increase in resource utilization with increasing bus width.

All three of the aforementioned requirements lead to considerable resource utilization. Although slicing [3], [4], aggressive strides, simultaneous processing of multiple streams [5] and many other principles that underlie CRC acceleration are well known, they cannot achieve low cost, high performance and programmability at the same time. A multi-core, multi-socket system with Intel's CRC instruction [9] can achieve high throughput, but it suffers from high latency and high power consumption in packet processing applications. In this brief, two algorithms and a method corresponding to the three requirements are proposed to reduce the resource utilization with guaranteed throughput and programmability. First, the stride-by-5 algorithm is proposed, which can reduce the resource utilization by 79.69%-79.98% compared with the slicing-by-4 and slicing-by-8 algorithms. Second, the pipelining go back algorithm is proposed to solve the padding zeros problem, which will introduce an $O(\log_2 n)$ resource utilization. Finally, a hardware internal configuration access port (HWICAP) is used to realize dynamic programmability, and it leads to small and constant resource utilization regardless of the bus width.

The remainder of this brief is organized as follows. Section II presents preliminaries to our proposals. Section III

Manuscript received May 1, 2020; revised June 19, 2020; accepted July 5, 2020. Date of publication July 13, 2020; date of current version December 21, 2020. This work was supported in part by the National Key Laboratory Foundation of China under Grant HTKJ2019KL504012, in part by the National Natural Science Foundation of China under Grant 61502204, and in part by the National Joint Foundation of China under Grant 6141B06301. This brief was recommended by Associate Editor H.-G. Yang. *(Corresponding author: Weitao Pan.)*

Huan Liu, Zhiliang Qiu, and Weitao Pan are with the State Key Laboratory of Integrated Service Networks, Xidian University, Xi'an 710071, China (e-mail: wtpan@mail.xidian.edu.cn).

Jun Li is with the National Key Laboratory of Science and Technology on Space Microwave, Xi'an Institute of Space Radio Technology, Xi'an 710100, China.

Ling Zheng is with the School of Communication and Information Engineering, Xi'an University of Posts and Telecommunications, Xi'an 710121, China.

Ya Gao is with the School of Internet of Things Technology, Wuxi Institute of Technology, Wuxi 214121, China.

Color versions of one or more of the figures in this article are available online at https://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCSII.2020.3008932

discusses the system architecture and the three proposals. Section IV shows the synthesis results. Section V concludes this brief.

II. Preliminaries

A. Parallel CRC Algorithms

The parallel CRC algorithm can process multiple data input bits simultaneously [10]. The number of bits processed in parallel is n, which is also the width of the inner bus in the remainder of this brief. The parallel input data are $B_n = [b_0, b_1, \ldots, b_{n-1}]^T$. The value of the CRC register is $C^{(k)}$ before B_n enters. The relationship between $C^{(n+k)}$ and $C^{(k)}$ is

$$\begin{aligned} C^{(n+k)} &= TC^{(n+k-1)} + Sb_{n-1} \\ &= T^nC^{(k)} + T^{n-1}Sb_0 + T^{n-2}Sb_1 + \cdots + Sb_{n-1} \\ &= T^nC^{(k)} + W_{ln}B_n \end{aligned} \tag{1}$$

where W_{ln} is a matrix of size $l \times n$ and

$$W_{ln} = \left[T^{n-1}S, T^{n-2}S, \ldots, TS, S\right] \tag{2}$$

T is a matrix of size $l \times n$, and S is a column vector of size l. Both T and S can be derived from an LFSR-based serial CRC algorithm, and l is the size of the LFSR. The serial CRC algorithm can be found in the extended version of this brief [11].

B. Programmability and HWICAP

T^n and W_{ln} are generally stored inside LUTs for the FPGA-based implementation of CRC algorithms, and a programmable implementation requires the ability to modify the content of the LUTs at runtime. HWICAP is a Xilinx IP core that can afford users with access to ICAP primitives using the AXI4-Lite protocol. It can modify the content of the LUTs dynamically. The resource utilization of HWICAP is as low as 186 LUTs, and it will not increase with increasing inner bus width. For the Intel/Altera FPGAs, a similar function can be achieved by using PR-IP [12].

III. Proposed Work

A. Non-Segmented System Architecture

The proposed non-segmented system architecture is shown in Fig. 1. In a non-segmented system architecture, there should be one frame in a single word, and a segmented system architecture can process multiple frames at the same time [13]. Regions 1 and 2 correspond to the computation of $W_{ln}B_n$ in (1). Region 1 consumes most of the LUTs, and the number of consumed LUTs linearly depends on the size of W_{ln}. The stride-by-5 algorithm, which is discussed in Section III-B, is proposed to reduce the LUT consumption of Region 1. Region 2 is implemented by means of an xor tree instead of a one-stage xor function to achieve higher performance. Region 3 completes the computation of (1). It consumes few LUTs for the small size of $\boldsymbol{T}^n$. The padding zeros problem is solved by Region 4, and the pipelining go back algorithm, which results in an $O(\log_2 n)$ resource utilization, is proposed

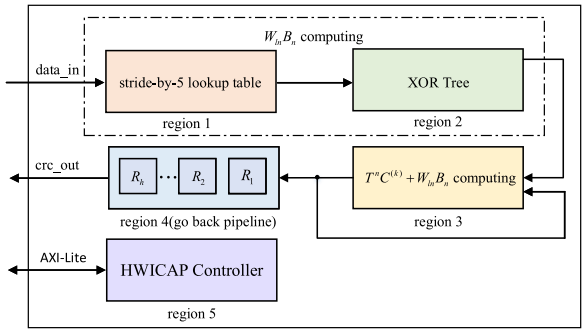


Fig. 1. Proposed non-segmented system architecture.

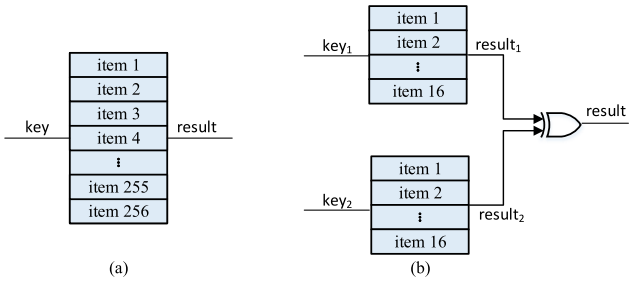


Fig. 2. Function implementation with (a) stride-by-8 and (b) stride-by-4.

and discussed in Section III-C. Region 5 is an HWICAP controller that can modify the content of the LUTs dynamically. The operation procedure is discussed in Section III-D. A segmented system architecture is proposed in Section III-E. Implementation details of the abovementioned proposals can be accessed at [1].

B. Stride-by-5 Algorithm

In this section, the model of the resource utilization is established, the stride-by-5 is proved to be the best stride for various bus widths, and the stride-by-5 algorithm is described in Algorithm 1.

Stride, as its name implies, refers to the number of bits processed by a single logical table. The logical table can be realized using FPGA LUTs, and it can load the truth table of a function. For example, an eight-input function is defined as

$$y = x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + x_8 \tag{3}$$

which can be transformed equivalently as

$$\begin{cases} y_1 = x_1 + x_2 + x_3 + x_4 \\ y_2 = x_5 + x_6 + x_7 + x_8 \\ y = y_1 + y_2 \end{cases} \tag{4}$$

Equations (3) and (4), whose strides are 8 and 4, can be implemented as shown in Fig. 2(a) and Fig. 2(b), respectively. A smaller stride means that a smaller logical table can be realized by a single LUT or cascaded LUTs. Can stride-by-1 be considered the best stride for FPGA implementation? We will establish the resource utilization model and determine the answer.

l equations with the same n inputs are required to realize the computation of $W_{ln}B_n$ in (1). n is also the bus width, and

$$n = ms + r \tag{5}$$

Algorithm 1 Stride-by-5 Algorithm

Input: The bus width n. The stride s. The input vector $B[n]$. The computing matrix $W[l][n]$.
Output: The meta matrix $MD[l][m+1]$, which is the input of Region 2.

```
Initialize s to 5;
Initialize m to ⌊n/s⌋;
Initialize MD[l][m + 1] to the null matrix;
for i = 0 to l − 1 do
    for j = 0 to m − 1 do
        for k = 0 to s − 1 do
            MD[i][j] = MD[i][j] ⊕ (B[j × s + k] · W[i][j × s + k]);
        end for
    end for
end for
for i = 0 to l − 1 do
    for j = s · m to n − 1 do
        MD[i][m] = MD[i][m] ⊕ (B[j] · W[i][j]);
    end for
end for
```

in which s is the stride. m equals $\lfloor n/s \rfloor$. r is the remainder, which equals $n \bmod s$.

The function $A(x)$ is defined as

$$A(x) = \begin{cases} 0 & x = 0 \\ 1 & x > 0 \end{cases} \tag{6}$$

and the resource utilization function is defined as

$$K(n, s, l) = \begin{cases} (l/2) \cdot (\lfloor n/s \rfloor + A(n \bmod s) & \\ \qquad\qquad s \le 5 & \\ (l/2) \cdot (\lfloor n/s \rfloor \cdot 2^{s-5} + A(n \bmod s)) & \\ \qquad\qquad s > 5, n \bmod s \le 5 & \\ (l/2) \cdot (\lfloor n/s \rfloor \cdot 2^{s-5} + A(n \bmod s) & \\ \quad \times\ 2^{(n \bmod s)-5}) \qquad s > 5, n \bmod s > 5 & \end{cases} \tag{7}$$

There are three equations corresponding to different s. A single LUT is required to realize a logical table when s is less than five, and cascaded LUTs are needed to realize a logical table when s is greater than five. This is because a single LUT has five inputs. l is divided by 2 for the two outputs of a single LUT.

The stride-by-5 algorithm is optimal for the 5-input LUTs in FPGAs. Stride-by-5 reduces the resource utilization by 79.69%-79.98% compared with stride-by-8, which is used in the slicing-by-4 and slicing-by-8 algorithms. For FPGAs with non-5-input LUTs (prior to Xilinx Virtex-5 or Altera Stratix II), the stride defined by the number of LUT inputs should be used, and the LUT sharing mechanism should be exploited. The stride-by-5 algorithm is described in Algorithm 1; it processes the computation in Region 1 here, but the algorithm can also be used in Regions 3 and 4.

C. Pipelining Go Back Algorithm

In this section, the pipelining go back algorithm is proposed with an $O(\log_2 n)$ resource utilization, and the derivation and description of the algorithm are presented.

The padding zeros problem is discussed in Section I. p is used to represent the number of valid bits in the final word. q is used to represent the number of padding zeros. The data vector of the final word is $B_{p+q} = [\mathrm{b}_0, \ldots, \mathrm{b}_{p-1}, 0, \ldots, 0]^T$. Substitute B_{p+q} into (1), and then

$$\begin{aligned} C^{(p+q+k)} &= T^{p+q} C^{(k)} + W_{(p+q)n} B_{p+q} \\ &= T^q \left(T^p C^{(k)} + T^{p-1} Sb_0 + \cdots + Sb_{p-1} \right) \\ &= T^q C^{(p+k)} \end{aligned} \tag{8}$$

The relationship between $C^{(p+k)}$ and $C^{(p+q+k)}$ is

$$C^{(p+k)} = T^{-q} C^{(p+q+k)} \tag{9}$$

There will be an $O(1)$ resource utilization to realize the computation of T^{-q} because the size of T^{-q} is $l \times l$ and has no relation to n. However, q varies and $0 \le q < n$, and if we use the n table corresponding to every possible q, there will be an $O(n)$ resource utilization. We introduce a pipeline to reduce the resource utilization to $O(\log_2 n)$.

Inspired by the binary representation, q is represented as

$$q = 8 \cdot \left(x_{h-1} \cdot 2^{h-1} + x_{h-2} \cdot 2^{h-2} + \cdots + x_1 \cdot 2 + x_0 \right) \tag{10}$$

q and n are multiples of 8 because the data transfer is in bytes. The value of x can be 0 or 1, and h is the number of pipeline stages, which can be represented as

$$h = \log_2(n/8) \tag{11}$$

Equations (10) and (11) can be used to convert (9) to

$$\begin{aligned} C^{(p+k)} &= \left(\left(T^{-8 \cdot 2^{n-1}} \right)^{x_{n-1}} \cdots \left(T^{-8} \right)^{x_0} \right) C^{(p+q+k)} \\ &= \left(R_1^{x_{n-1}} \cdot R_2^{x_{n-1}} \cdots R_h^{x_0} \right) C^{(p+q+k)} \end{aligned} \tag{12}$$

where $[R_1, R_2, \ldots, R_h]$ is the h matrices for the h-stage pipeline, and the size of each matrix is $l \times l$. $[R_1, R_2, \ldots, R_h]$ can be used to convert $C^{(p+q+k)}$ to $C^{(p+k)}$. The stride-by-5 algorithm can be used to convert $[R_1, R_2, \ldots, R_h]$ to the content of the LUTs. Using the resource utilization function in (7), the resource utilization of the pipeline is $K_{R_4} = h \cdot K(l, s, l)$, where R_4 indicates Region 4 in Fig. 1. K_{R_4} can be represented as

$$K_{R_4} = \begin{cases} \log_2(n/8) \cdot (l/2) \cdot (\lfloor l/s \rfloor + A(l \bmod s)) & \\ \qquad\qquad s \le 5 & \\ \log_2(n/8) \cdot (l/2) \cdot (\lfloor l/s \rfloor \cdot 2^{s-5} + A(l \bmod s)) & \\ \qquad\qquad s > 5, n \bmod s \le 5 & \\ \log_2(n/8) \cdot (l/2) \cdot (\lfloor l/s \rfloor \cdot 2^{s-5} + A(l \bmod s) & \\ \cdot 2^{(l \bmod s)-5}) \qquad s > 5, n \bmod s > 5 & \end{cases} \tag{13}$$

As shown in (13), we can achieve an $O(\log_2 n)$ resource utilization using the pipelining go back algorithm. The algorithm is described in Algorithm 2.

D. Reprogramming by HWICAP

Region 5 in Fig. 1 represents an HWICAP IP core, which can dynamically modify the content of the LUTs. It consumes 186 LUTs for any bus width. In contrast, the configuration logic realized by logic resources leads to several thousand LUTs being consumed when $n \ge 1024$ [4], and the resource utilization increases with increasing bus width.

The operation procedure of reprogramming using the HWICAP IP core is described as follows:

1) Complete the initial design, generate the bitstream using Vivado, and download the bitstream into the FPGA;
2) Extract the locations of the LUTs used;

Algorithm 2 Pipelining Go Back Algorithm

Input: The temporary CRC value $C^{(p+q+k)}$, the bus width n, the number of padding zeros q, the computing matrix T.
Output: The desired CRC value $C^{(p+k)}$.

```
Initialize h to log2(n/8), q to q/8;
Initialize matrix R to the null matrix, matrix C^(p+k) to C^(p+q+k);
for i = h − 1 to 0 do
    if q ≥ 2^i then
        R = T^(−8·2^i);
        C^(p+k) = R C^(p+k);
        q = q − 2^i;
    else
        continue;
    end if
end for
```

3) When reprogramming is needed, compute the new content of the LUTs using (1) and (12);
4) Map the content of the LUTs to the initial value of the LUTs;
5) Write the initial value to the LUTs using the AXI Lite interface of the HWICAP IP core.

The method of reprogramming by HWICAP is useful in engineering. Our contributions are as follows:

1) We verify the feasibility of reprogramming the FPGA implementation of the CRC algorithm using the HWICAP IP core. It leads to small and constant resource utilization regardless of the bus width;
2) The proposed method can change the CRC polynomial directly, without re-coding and re-synthesizing;
3) The code of the above procedure can be accessed in [1], as a part of the entire project. To the best of our knowledge, this is the first open-source code covering the whole procedure described above.

E. Segmented System Architecture

The non-segmented system architecture cannot process multiple frames in one word (clock), which reduces the throughput of short or misaligned frames. This is called the bus efficiency problem. A segmented system architecture is proposed to solve the problem. The bus format is the same as that in [5], and the *block* in [5] is another name for the *segment* in [13]. For example, a 4096-bit bus can process eight complete frames at the same time; hence, the bus can be divided into eight *regions* [5]. The number of regions depends only on the bus width. Different segment widths are feasible, and if a 64-bit segment width is chosen, one region can be divided into eight segments (blocks). The proposed segmented system architecture is shown in Fig. 3. Compared with the proposed non-segmented system architecture, the proposed segmented system architecture has a slightly more complex Region 1 and Region 2 and multiple duplicates of Region 3 and Region 4. The number of duplicates is just the maximum number of frames processed in a single word.

The comparison between the proposed segmented system architecture and the proposed non-segmented system architecture can be found in Fig. 4. The red cuboids represent the non-segmented system architecture. The blue cuboids represent the increment between the proposed segmented

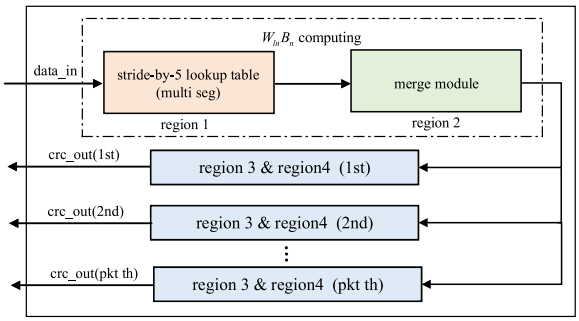
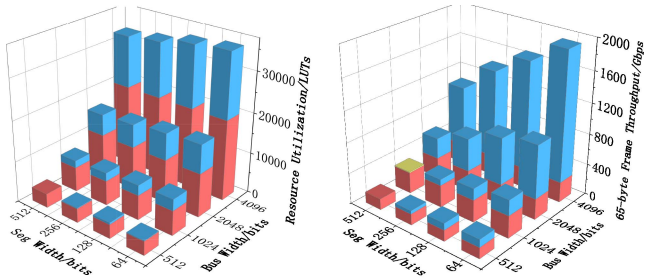


Fig. 3. Proposed segmented system architecture.



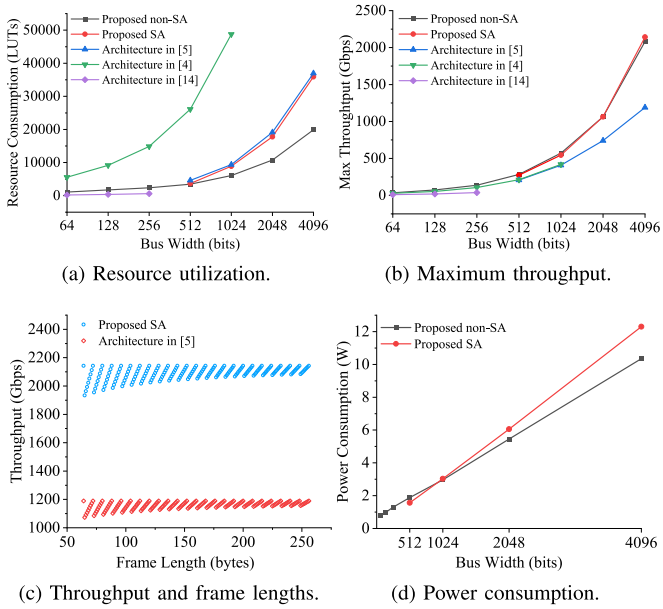
(a) Resource utilization. (b) 65-byte-frame throughput.

Fig. 4. Comparison between segmented system architecture and non-segmented system architecture.

system architecture and the proposed non-segmented system architecture. The yellow slice (bus width = 1024, segment width = 512) represents the decrement between the two architectures. Fig. 4(a) shows that the increment in resource utilization depends mainly on the bus width instead of the segment width. This is because the increment in resource utilization depends mainly on the number of duplicates of Regions 3 and 4, which depends only on the bus width. Fig. 4(b) shows that the increment in 65-byte-frame throughput is obvious for most cases. The only decrease in throughput is found when the bus width is 1024 bits and the segment width is 512 bits, where the two architectures have the same bus efficiency for 65-byte-frame throughput, and the non-segmented architecture has a slightly higher frequency. Therefore, 64 bits is chosen as the segment width in the rest of this brief. A detailed comparison between the segmented and non-segmented architectures can be found in the extended version of this brief [11].

IV. Experimental Results

There are three state-of-the-art studies [5], [4], [14]. The architecture in [4], [14] can be reprogrammed, whereas the architecture in [5] cannot be reprogrammed. The two proposed architectures are implemented with Virtex-7 XC7VX690T, and [5], [4], [14] use Virtex-7 XCVH870T, Virtex-6 XC6VLX550T and Stratix-V 5SGSED6N1F45I2, respectively. In this section, the two proposed architectures are compared with these works in terms of resource utilization and maximum throughput. The proposed segmented architecture is compared with [5] in terms of throughput on various frame lengths. The power consumption of two proposed architectures



(a) Resource utilization. (b) Maximum throughput.

(c) Throughput and frame lengths. (d) Power consumption.

Fig. 5. Synthesis result.

is reported also. In the following, we use *SA* to refer to the *segmented architecture*.

The synthesis result is illustrated in Fig. 5. Fig. 5(a) shows that the resource utilization of the proposed non-SA is lower than that of the architectures in [4] and [5] by 80.7%-87.5% and 25.1%-46.2%, respectively. The proposed SA has lower resource utilization by 81.7%-85.9% and 2.9%-20.8%. The lower resource utilization results from implementing the algorithms and method described in Section III, which can also guarantee high performance and programmability. The architecture in [14] has lower resource utilization than that of the non-SA by 74.4%-81.3%. The reasons for the lower resource utilization of [14] are as follows: 1) Reference [14] needs to process only half-filled and fully filled packets. In other words, the padding zeros problem is partly addressed. By contrast, the two proposed architectures and [5], [4] can fully address the padding zeros problem. 2) The cost of the Nios II IP core is not considered in [14]. By contrast, the two proposed architectures consider the cost of HWICAP. Moreover, it is difficult to scale the bus width of [14] up to 1024 bits.

Fig. 5(b) shows that the maximum throughput of the proposed non-SA is higher than that of the architecture in [4], [5], [14] by 24.2%-37.9%, 37.4%-75.0% and 259.4%-284.5%, respectively. The maximum throughput of the proposed SA is higher than that of the architecture in [4], [5] by 28.7%-30.2% and 32.2%-80.2%, respectively. The higher frequency leads to higher throughput, and the two proposed architectures can achieve higher frequency for the well-arranged pipelines in Regions 1, 2, and 4.

The throughput on frame lengths from 64 bytes to 256 bytes can be found in Fig. 5(c). Only [5] and the proposed SA are compared for their ability to process multiple frames in one word. The two architectures use a 4096-bit bus width and a 64-bit segment width; therefore, they have the same bus efficiency. The proposed SA has an 80.2% higher frequency and throughput than [5]. The lowest throughput of 1933.9 Gbps is achieved when the frame length is 65 bytes.

The power consumption of the two proposed architectures is illustrated in Fig. 5(d). They run at 500Mhz. The dataset comes from the post-implementation power reports generated by vivado. The power consumption is composed of static power consumption and dynamic power consumption. The static power consumption varies from 0.32 W to 0.48 W, and the dynamic power consumption increases linearly with increasing bus width. The power consumption of the proposed SA has a faster growth than that of the proposed non-SA. This is because the resource consumption of the proposed SA increases faster than that of the proposed non-SA.

The board-level implementation and the comparison with other works can be found in the extended version of this brief [11].

V. Conclusion and Future Work

Two algorithms and a method are proposed to realize low-cost, high-performance, and programmable CRC computation. These algorithms and the presented method can be used in segmented or non-segmented architectures. The synthesis results show that the proposed architectures can achieve lower resource utilization and higher throughput than two state-of-the-art architectures. The source code can be accessed in [1]. Our future work will focus on making the hardware reconfiguration method (HWICAP) technology independent.

References

[1] *S.C. of Low-Cost and Programmable CRC Implementation*. Accessed: Apr. 24, 2020. [Online]. Available: https://github.com/FPGA-Networking/Low-Cost-and-Programmable-CRC

[2] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc, "Design of ion-implanted MOSFET's with very small physical dimensions," *IEEE J. Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, Oct. 1974.

[3] 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.

[4] A. Akagic and H. Amano, "High-speed fully-adaptable CRC accelerators," *IEICE Trans. Inf. Syst.*, vol. 96, no. 6, pp. 1299–1308, 2013.

[5] L. Kekely, J. Cabal, and J. Kořenek, "Effective FPGA architecture for general CRC," in *Proc. Int. Conf. Archit. Comput. Syst.*, 2019, pp. 211–223.

[6] C. Toal, K. McLaughlin, S. Sezer, and X. Yang, "Design and implementation of a field programmable CRC circuit architecture," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 17, no. 8, pp. 1142–1147, Aug. 2009.

[7] *The P4 Language Specification, Version 1.0.5*, P4 Lang. Consortium, Stanford, CA, USA, Nov. 2018. [Online]. Available: https://p4.org/p4-spec/p4-14/v1.0.5/tex/p4.pdf

[8] M. Grymel and S. B. Furber, "A novel programmable parallel CRC circuit," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 19, no. 10, pp. 1898–1902, Oct. 2011.

[9] S. Gueron, "Speeding up CRC32C computations with Intel CRC32 instruction," *Inf. Process. Lett.*, vol. 112, no. 5, pp. 179–185, 2012.

[10] G. Campobello, G. Patane, and M. Russo, "Parallel CRC realization," *IEEE Trans. Comput.*, vol. 52, no. 10, pp. 1312–1319, Oct. 2003.

[11] H. Liu, Z. Qiu, W. Pan, J. Li, L. Zheng, and Y. Gao. (Apr. 2020). *Low-Cost and Programmable CRC Implementation Based on FPGA*. [Online]. Available: https://www.techrxiv.org/articles/Low-Cost_and_Programmable_CRC_Implementation_based_on_FPGA/12181494

[12] K. Vipin and S. A. Fahmy, "FPGA dynamic and partial reconfiguration: A survey of architectures methods and applications," *ACM Comput. Surveys*, vol. 51, no. 4, pp. 1–39, 2018.

[13] P. Orosz, T. Tóthfalusi, and P. Varga, "FPGA-assisted DPI systems: 100 Gbit/s and beyond," *IEEE Commun. Surveys Tuts.*, vol. 21, no. 2, pp. 2015–2040, 2nd Quart., 2019.

[14] M. Jubin and T. Nayak, "Reconfigurable very high throughput low latency VLSI (FPGA) design architecture of CRC 32," *Integration*, vol. 56, pp. 1–14, Jan. 2017.