High speed CRC with 64-bit generator polynomial on an FPGA

Article *in* ACM SIGARCH Computer Architecture News · December 2011

DOI: 10.1145/2082156.2082175 · Source: DBLP

2 authors, including:

Amila Akagic
Keio University
8 PUBLICATIONS **16** CITATIONS

High Speed CRC with 64-bit generator polynomial on an FPGA

Amila Akagić and Hideharu Amano
Department of Information and Computer Science, KEIO University
3-14-1 Hiyoshi, Yokohama, Japan 223-8522
{amila,hunga}@am.ics.keio.ac.jp

ABSTRACT

Deployment of jumbo frame sizes beyond 9000 bytes for storage systems is limited by 32-bit Cyclic Redundancy Checks used by a network protocol. In order to overcome this limitation we study possibility of using 64-bit polynomials in software and hardware, by using fastest multiple lookup tables algorithms for generating CRCs. CRC is a sequential process, thus the software based solutions are limited in throughput by speed and architectural improvements of a single CPU. We study tradeoff between using distributed LUTs and embedded BRAM in hardware implementations. Our results show that BRAM-based approach is the fastest hardware implementation, reaching maximum of 347.37 Gbps while processing 1024 bits at a time, which is 606x faster than the software implementation of the same algorithm running on Xeon 3.2 GHz with 2 MB of L2 cache. The proposed architectures have been implemented on Xilinx Virtex 6 LX550T prototyping device, requiring less than 1% of the device's resources. Our research show that throughput will continue to increase when we increase the number of processed bits at a time.

Keywords

Cyclic Redundancy Check, CRC, Reconfigurable architectures, FPGA, storage systems

1. INTRODUCTION

Recent advances in computer networks have resulted in a significant increase in the bandwidth of computer network links. One of the parameters that hasn't been changed since creation of Ethernet is maximum transmission unit (MTU), which is still 1500 bytes. There are studies that show that increasing MTU's size will decrease CPU utilization and provide higher end-to-end throughput [1], and some protocols are deploying jumbo frame sizes (eg. iSCSI) to reduce the effect of TCP frame overhead, with maximum frame size reaching 9000 bytes. The cause of this limit is 32-bit Cyclic Redundancy Check (CRC), which is effective only for Ethernet frame sizes under 12000 bytes. Processing longer frame sizes with 32-bit CRC will decrease effectiveness of CRC in error detecting during transmission.

One group of systems that could highly benefit from changing Ethernet frame size are storage-over-IP systems using iSCSI or FCIP, since they need to transfer large amount of data very fast, while maintaining data integrity. In order to consider using longer frame sizes, beyond 9000 bytes, higher degree CRC polynomial must be considered. The general rule of thumb is the higher the checksum length, the better the effectiveness of CRC in error detecting. Thus by doubling length of standard Ethernet CRC field, from 32 to 64 bits, effectiveness of CRC in error detecting will also be increased. Storage systems with this improvements can provide faster data transfers and better quality of service.

The process of calculating CRC values is known to be computationally intensive process, since it is highly sequential. Algorithms with 32-bit generator polynomial implemented in software are limited by speed and architectural improvements of today's processors. In the best case scenarios, fastest CRC algorithms can reach less than 10 Gbps throughput on the state-of-the-art Xeon 3 GHz processor with 4 MB of L2 cache [2], and these types of processors are still not affordable by most end users. Due to its sequentiality, it's not known how CRC can take advantage of multicore paradigm shift.

Deploying 64-bit generator polynomial is more complex and could pose significant burden on today's CPUs for high speed networks. We implemented five versions of fastest CRC algorithms with 64-bit polynomial in software and measured their performance and in this paper we present results. The maximum throughput was attained on Xeon 3 GHz processor with 4 MB of L2 cache and it was 2.6 Gbps while processing 64 bits of data at a time, with data ranged from 1 KB to 1 GB. Due to high requirements for cache space, other algorithms attained less throughput. Transmission speed of computer networks are developing much faster than general processors, so in order to support high speed links, we believe that computationally intensive tasks should be accelerated by deploying reconfigurable architectures.

We are proposing using reconfigurable hardware and we present our implementation of five algorithms on an FPGA. We are basing our solution on newly proposed CRC algorithms, Slicing-by-$\{8, 16, 32, 64, 128\}$, that are noted fastest ones implemented in software. The reason why we choose these algorithms is that they can process arbitrarily large amounts of data by deploying lookup tables. FPGAs have ability to access these tables in parallel, which can provide us with constant increase in throughput while doubling number of bits processed at a time. Our results show that maximal attainable throughput on Xilinx Virtex 6 LX550T while processing 1024 bits of data at a time is 347.4 Gbps. If we would to continue to increase number of bits processed at a time, throughput would continue to increase. We also study tradeoff between using LUT and BRAM components for implementations of the algorithms lookup tables.

Our contributions in this paper are: a) Software implementation of Slicing-by-$\{8, 16, 32, 64, 128\}$ algorithms with 64-bit generator polynomial, b) LUT- and BRAM-

based implementation of the same algorithms on an FPGA, c) detailed performance analysis of both implementations.

The objective for 40 and 100 GbE is to preserve minimum and maximum frame size of current IEEE 802.3 standard [3], but we believe that some special purpose systems, such as storage systems, backup systems, safety critical embedded system applications, fail safe systems etc. can benefit from exploiting larger frame sizes. New interface cards can be designed to support both - high and slow speed backward compatible messages. Also, there is a constant need for new standards and applications that require a high degree of data integrity and they can adopt our proposal.

The organization of this paper is as follows: we start out with a brief overview of CRC algorithms, highlighting Slicing-by-N_{64p} algorithms, after which we present software implementation and we analyze results in section 2 and 3, respectively. In section 4, we present our LUT- and BRAM-based implementation of Slicing-by-N_{64p} algorithms on the Xilinx Virtex 6 LX550T FPGA and analyze our results. We compare results from software and FPGA implementations in section 5. We present related work in section 6. and conclude the paper in section 7.

2. OVERVIEW OF CRC ALGORITHMS

In this section we provide a high level overview of CRC with emphasize on newly proposed CRC algorithms - Slicing-by-N [4], to provide sufficient context for readers while highlighting some major overheads of these algorithms.

CRC is an error detection scheme that detects corruption of digital content during data transmission, processing and storage. CRC is the most common error detection scheme used today, since it has low probability of undetected errors.

Traditionally, CRC of input data is calculated by performing long division operation between input data and fixed polynomial known as generator polynomial. An input data or a message M is treated as polynomial, where bit values are represented as coefficients of various powers of x, for example the message "01010100" is represented as $0 \times x^7 + 1 \times x^6 + 0 \times x^5 + 1 \times x^4 + 0 \times x^3 + 1 \times x^2 + 0 \times x^1 + 0 \times x^0$. Let the length of M be defined as l, then M can be represented as:

$$M(x) = m_{l-1}x^{l-1} + m_{l-2}x^{l-2} + ... + m_0 \qquad (1)$$

where m_{l-1} is most significant bit of the message M and and m_0 is least significant bit. Generator polynomial of length w is represented in the same manner:

$$G(x) = g_w x^w + g_{w-1}x^{w-1} + g_{w-2}x^{w-2} + ... + g_0 \quad (2)$$

Before performing long division process, message M is multiplied by x^w and then divided by $G(x)$. The remainder of this operation is a CRC value.

$$CRC(M) = M(x) \times x^w \mod G(x) \qquad (3)$$

The CRC is then affixed to the original message and transmitted to a receiver.

Due to interference on the data link, data might be corrupted during transport. The receiver will detect errors by performing similar process as the sender. First, received CRC will be removed, and the receiver will perform long division operation with the same generator polynomial, which is specified by the protocol used and it is known to both receiver and sender. Then it will compare received CRC value and its own. Any discrepancy between these two values indicates the presence of transmission errors in the received pair. In this case, the receiver will discard the message and request retransmission of the data.

Calculation of CRC can be implemented in hardware and software. Early hardware implementation is linear feedback shift register (LFSR), which calculates CRC value of N-bit message in N clock cycles. Input message is fed serially into a circuit, thus circuit's throughput is limited by its maximum operating frequency. Every bit in an input message requires one XOR and one shift operation.

Commonly used algorithm for software implementation is Sarwate algorithm [5]. It uses precomputed table filled with the remainders of long division operation between all possible 8-bit values, shifted to left by degree of a polynomial minus one bit, with the generator polynomial. In this case, eight XOR and eight shift operations (worst case scenario) are replaces with one lookup into a table. This algorithm process only one byte of data at a time.

In order to process a message M of length l, Sarwate algorithm requires a table of $2^l \times (w-1)$, where w is length of generator polynomial. Today's computers are capable of processing 32 or 64 bits of data, thus this algorithm would require $2^{32} \times (w-1)$ and $2^{64} \times (w-1)$, respectively. These tables cannot fit into today's cache units, so their contents have to be fetched from RAM constantly, causing significant performance drop.

To overcome these limitations, new algorithms have been proposed in [4] and they can read arbitrarily large amounts of data at the time, while requiring significantly less memory for its tables. Proposed Slicing-by-4 algorithm can read 32 bits of input data at a time and it doubles the performance of existing implementations of Sarware algorithm, while Slicing-by-8 triples the performance and reads 64 bits of input data at a time. Algorithms are based on two principles of modulo-2 arithmetic, bit slicing and bit replacement, which reduce memory usage. Table can fit into a moderate size cache units, which significantly reduces memory latency.

Basic iteration step is XORing input message with current CRC value, and then slicing intermediate value into smaller slices used to access smaller precomputed tables in parallel. The most commonly used length of a slice is 8 bits, since it requires a moderate size tables (1 table requires 1KB = 2^8 entries $\times$ 32 bit remainders) as well as a moderate number of instructions per message. Attempts to lower this length (eg. 4 bit slices) would decrease memory usage by half, but number of instructions would double, causing significant performance drop on a CPU. Slicing-by-4 algorithm requires four eight bit-slices - 4KB of memory, while Slicing-by-8 algorithm requires 8KB.

Contents of the tables depend on a position of the byte in data stream processed at a time. The remainders for the first table are calculated by performing long division operation between all possible 8 bit values, shifted by 32 to left, with generator polynomial. The second table hold all remainders from long division operation between all possible 8 bit values shifted by 48 with generator polynomial, and so forth. Detail analysis of the framework for generating these algorithms is presented in [4].

We call these algorithms Slicing-by-N_{64p} in short, where

$N \in \{8, 16, 32, 64, 128\}$, and they represent Slicing-by-8, Slicing-by-16, Slicing-by-32, Slicing-by-64 and Slicing-by-128, respectively. $64p$ marks 64-bit generator polynomial, in order to differentiate these algorithms with the ones deploying 32-bit generator polynomial.

3. SOFTWARE IMPLEMENTATION AND PERFORMANCE ANALYSIS

In this section we provide details of software implementation of CRC Slicing-by-N_{64p} algorithms, where $N \in \{8, 16, 32, 64, 128\}$. We implement these algorithms in software in order to understand their throughput limitations, and to compare results with hardware throughput. Until now, to the best of our knowledge, only algorithms with 32 bit generator polynomial are implemented and analyzed in software [4][2].

For 64-bit polynomial we chose the mostly used 64-bit generator polynomial CRC-64-ECMA-182, with hexadecimal representation - $0x42F0E1EBA9EA3693$. There are no studies about effectiveness of 64-bit polynomials in general, and since our primary concern is performance, we will not study effectiveness in this paper. It will be a subject of future work. Our implementation will work the same with every other CRC polynomial, regardless of its effectiveness.

We implemented the algorithms in C. In order to perform fair comparison we read from one memory location to avoid data cache misses. Input into programs ranged from 512 B to 1 GB of data. In this range throughput varied insignificantly, so we present only results for 1 GB of processed data. Programs are executed several hundred thousand times and average core program execution time is provided. By core program we mean only "for" loop that performs calculation of CRC.

The programs are executed on following Intel processors: (1) Xeon 3.2 GHz with 2 MB of L2 cache, (2) Xeon 3 GHz with 4 MB of L2 cache, (3) Core Quad 2.83 GHz with 6 MB of L2 cache and (4) Core2 2.4 GHz with 4 MB of L2 cache. All processors are multicore processors with shared cache, so each logical core has only half amount of L2 cache available. During execution of programs, CPU utilization for all the experiments was 100%.

In Figure 1. we present throughput of Slicing-by-$\{8, 16, 32, 64, 128\}_{64p}$ algorithms with 64 bit generator polynomial while processing 1GB of data on all four processors. Results show that throughput is decreasing each time we double the number of processed bits at a time. Algorithms that processes more data in parallel require more space in cache for lookup tables during execution time. One lookup table requires $2^8 \times 64$ bit $= 2KB$, thus Slicing-by-$\{8, 16, 32, 64, 128\}_{64p}$ require 16 KB, 32 KB, 64 KB, 128 KB and 256 KB space in cache, respectively. In general, cache memory is shared between all programs running on the machine, and some of data have to be prefetched prior than they are needed. This means that for some of our algorithms, tables could not be present during whole execution time because of their size, causing significant number of cache misses. Waiting for data to be transferred from RAM to cache causes significant latency problem. In the case of these algorithms, the more tables algorithm requires, the slower it gets. The same programs with 32-bit generator polynomial didn't show this characteristic until we tried processing 512 bits at time [2]. Prior to that number, throughput was increasing.

The other important observation is regarding processor (1) and (2). Processor (1) is the fastest processors in our pool, but has the least L2 cache memory (2 MB). The throughput on processors (1) is 35% less than on processor (2) for Slicing-by-8_{64p}, and from 57% - 62% less for other algorithms. This shows the importance of L2 cache size on performance of these algorithms. Even though it is shown that algorithms scale good with a speed of a processor, they are also dependent on the size of its cache unit.

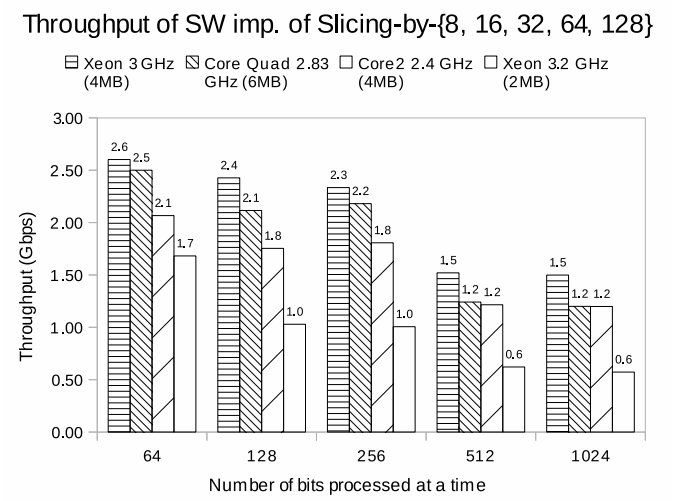

Figure 1: Throughput of Slicing-by-{8, 16, 32, 64, 128}$_{64p}$ software implementation with 64 bit generator polynomial while processing 1GB of data on four processors (1) - (4).

The results show that highest throughput was attained on processor (2) for Slicing-by-8_{64p} algorithm and it is 2.6 Gbps. Maximum throughput for other algorithms: Slicing-by-16_{64p} was 2.4 Gbps, Slicing-by-32_{64p} was 2.3 Gbps, Slicing-by-64_{64p} was 1.5 Gbps and Slicing-by-128_{64p} was 1.5 Gbps, attained on processor (2).

Deploying software implementation on link speeds such as 10, 40 or 100 Gbps would become significant bottleneck.

The major advantage of using software approach is its ability to adapt to different CRC standards. The implementation of other CRC 64-bit polynomial standards would require changing only contents of the tables.

4. HARDWARE IMPLEMENTATION AND PERFORMANCE ANALYSIS

In this section we describe how we implemented Slicing-by-N with 64 generator polynomial on an FPGA, and we present results.

The basic computational structure of Slicing-by-8_{64p} is shown in Figure 2. We read 64 bits every clock cycle from FIFO buffer and store it to a temporary register. This is the first stage in our pipeline. The initial step of the algorithm is to perform XOR operation between the first buffer value and a constant, defined by a CRC standard. In our case this value is 0xFFFFFFFFFFFFFFFF (crc_init_value in Figure 2). In every other iteration step, previously calculated CRC

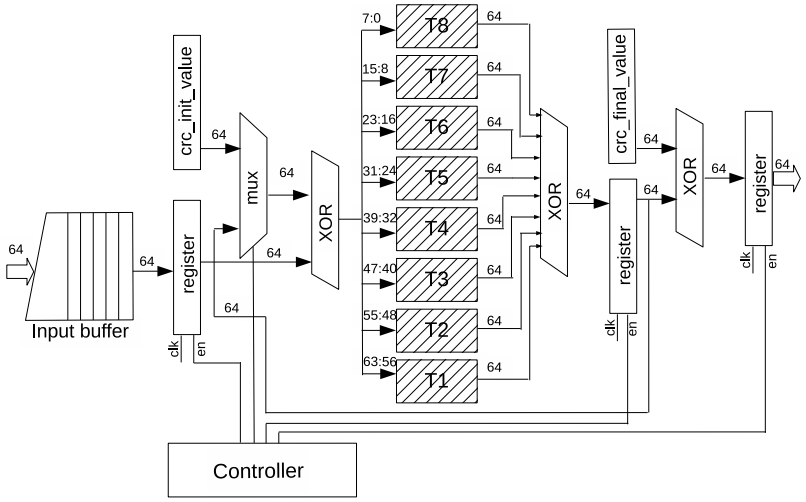

Figure 2: Slicing-by-8_{64p} computational structure.

Xilinx Virtex 6 LX550T	64		128		256		512		1024	
	LUT	BRAM	LUT	BRAM	LUT	BRAM	LUT	BRAM	LUT	BRAM
Slice LUTs (max 124800)	569	523	1095	688	2213	1172	4319	1504	8631	2230
Fully used LUT-FF pairs (max 969)	140	450	144	603	321	835	492	1208	759	1824
IOBs (max 1200)	133	133	197	197	325	325	581	581	1093	1093
BRAM (max 344)	-	8	-	16	-	32	-	64	-	128
Operating frequency (MHz)	391.67	468.02	365.49	430.91	349.42	332.36	322.62	332.36	297.1	347.37
Throughput (Gpbs)	24.48	29.25	45.69	53.86	87.36	83.09	161.31	166.18	297.1	**347.37**
TP/S	44.05	**85.99**	42.72	**80.17**	40.42	**72.6**	38.24	**113.14**	35.25	**159.51**

Table 1: Resource utilization of Slicing-by-{8, 16, 32, 64, 128}$_{64p}$ algorithms on the Xilinx Virtex 6 LX550T. 64, 128, 256, 512 and 1024 are numbers of bits processed at a time by the algorithms.

value is used as the second operand into the first XOR circuit. We describe this decision by using simple 2x1 multiplexer. The output from the first XOR circuit is the value that we have to slice into eight 8-bit slices. The slice is used as an index into $2^8 \times 64$ bits table. The CRC value is obtained by XORing these eight 64 bit remainders. The value is saved into a register, and with this the second stage of our pipeline is completed. In the third stage, a CRC value has to be XORed with a constant, also defined by a standard, and output from this operation is written into a register.

Implementation of other CRC algorithms have similar structure, except that each reads different number of bits from FIFO buffer and has different number of lookup tables, with connections between logical blocks being more complex.

The memory requirements for Slicing-by-N_{64p} algorithms grow proportionally with the number of bits processed at a time. In order to maintain increase in throughput, it is crucial to choose fast storage elements, and avoid any latency problems. The general rule is to use LUT-based method for smaller data sizes, since RAMs based on LUTs slow down if size grows larger. Modern FPGAs also feature embedded blocks to store relatively large amounts of data. We implemented algorithm's lookup tables by using two FPGA structures: FPGA LUTs (known as distributed memory) and Block RAM (known as embedded memory blocks). Since the algorithms don't require writing into the tables, we use BRAM to implement ROM modules, disabling its writing port. In order to implement tables in BRAM, they have to be synchronous.

Block RAM on Virtex 6 LX550T can be configured as general-purpose 36 Kb or 18 Kb RAM/ROM memory. Slicing-by-N_{64p} algorithms require only reading from the tables, so we chose single port BRAM that offers fast and flexible storage for large amounts of on-chip data. Our tables are organized as $2^8 \times 64$ bits, so we chose 32 Kb since it is the only option that supports read port width from 32 to 72 bits [6]. The maximum address width for this configuration is 9 bits, which can hold maximum of $2^9 \times 64 = 32$ Kb of data, exactly two of our tables. With additional logic, we could take advantage of this possibility, but it will limit performance of our circuit, because tables have to be read in parallel. Instead of utilizing one BRAM for two tables, we are using one 32 Kb BRAM block for each table.

We chose Xilinx Virtex 6 LX550T device (xc6vlx550t-2ff1760), primarily because it can support large number of IOBs (maximum of 1200). In Table 1. we present number of occupied resources, maximum operating frequency, attained throughput and number of throughput per slice (TP/S), obtained after synthesis and place and route processes are completed.

BRAM-based method overall achieves better throughput than LUT-based method (Figure 3). Until the Slicing-by-64_{64p} algorithm and 128 KB of data required, both methods were attaining similar throughput, but for the algorithm that process more data at a time, BRAM-based method is more efficient in the long run. The

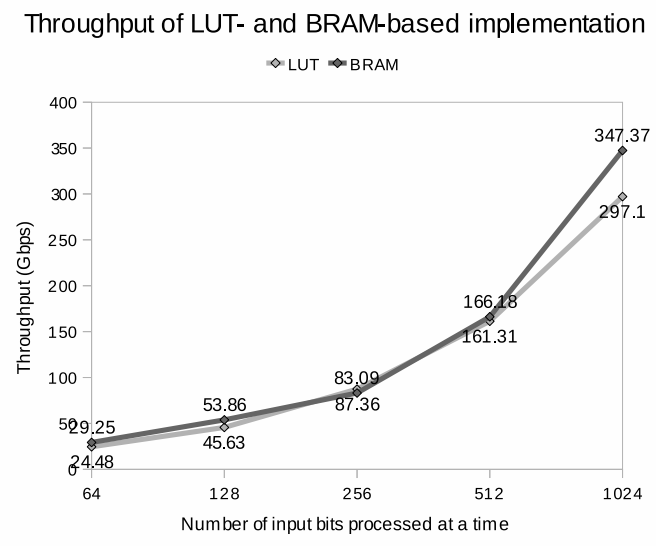

Figure 3: Throughput of LUT- and BRAM-based approaches for Slicing-by-{8, 16, 32, 64, 128}$_{64p}$ algorithms on Xilinx Virtex 6 LX550T.

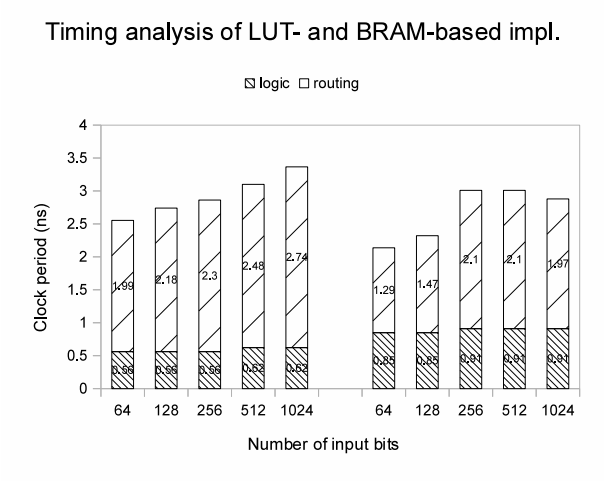

Figure 4: Minimum clock period for Slicing-by-{8, 16, 32, 64, 128}$_{64p}$ algorithms on Xilinx Virtex 6 LX550T obtained using static timing analysis.

number of resources in LUT-based approach is increased every time we process more data at a time, thus routing delay increases proportionly as seen in Figure 4. However, routing delay in LUT-based method increases more than BRAM-based method. Total delay that BRAM adds to our design is only 0.41 ns of logic delay for all five designs. There is additional routing delay on BRAM-based approach, but it is coming from other logic used in the design of particular algorithm, such as XOR gates with different number of inputs, wider registers and so forth. There is no additional routing delay coming from BRAM components, which makes BRAM-based approach more efficient for future expansions of the algorithms.

Throughput per slice (TP/S) shows efficiency of area utilization, and as expected, the results in Table 1. show that BRAM-based approach is more efficient, since it uses significantly less Slice LUTs than LUT-based approach.

There is no reason to stop increasing number of bits processed at a time, as long as FPGAs can capture that much data from a network interface. So, if we would to continue doubling data processed at a time, the BRAM-based method would provide better throughput at the cost of less resources used.

5. COMPARISON BETWEEN SOFTWARE AND FPGA IMPLEMENTATION

In this section we compare results from software and hardware implementation of Slicing-by-{8, 16, 32, 64, 128}$_{64p}$ algorithms.

We compare our results with software throughput while processing 1GB of input data, and present speedup in Figure 5. The maximum speedup is gained compared to throughput achieved on Intel Xeon 3.2 GHz processor with 2 MB of L2 cache and our implementation is 606x faster. The maximum throughput of software implementation, in idealized conditions, was attained by Slicing-by-8$_{64p}$ algorithm and it reached 2.6 Gbps running on Intel Xeon 3 GHz with 4 MB of L2 cache. Software implementation was not able to maintain increase in throughput by doubling number of bits processed at a time, because data from lookup tables could not fit into the cache, causing many cache misses. Fetching data from RAM caused significant latency problem, thus the total execution time increased, consequently decreasing throughput.

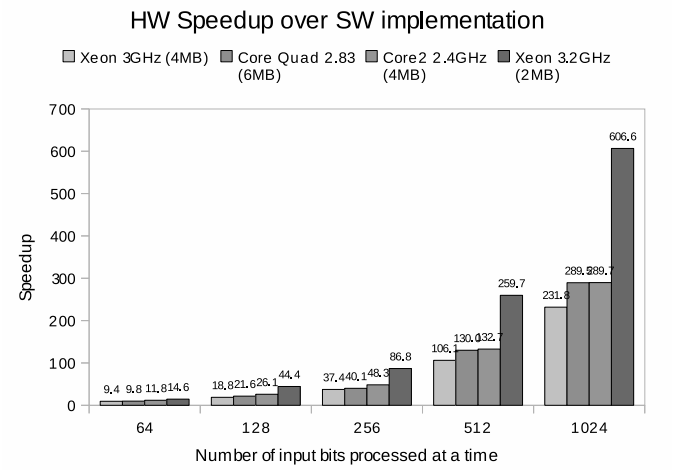

Figure 5: Speedup of FPGA implementation when compared with software implementation of Slicing-by-{8, 16, 32, 64, 128}$_{64p}$ algorithms on four state-of-the-art processors.

Throughput in hardware implementation is increasing with increase of number of bits processed at a time, because of FPGA's ability to access lookup tables in parallel. Increasing number of processed bits at a time only increased number of used resources on the FPGA, and the total execution time was affected only by routing delay of newly added resources. This means that it

would be possible to continue gaining higher throughput just by doubling the number of processed bits at a time.

FPGA can be programmed to directly communicate with network interface and generate CRCs as packets arrive, which will decrease interrupt processing cost of a CPU as well as protocol processing latency.

6. RELATED WORK

There is no related research on implementation of CRC algorithms deploying 64 bit generator polynomials. However there is substantial amount of research papers on algorithms deploying 32 bit generator polynomials [5] [7] [8] [4], as well as their hardware implementations.

Novel field programmable CRC computation circuit for VLSI implementation is presented in [9], with maximum throughput of 4.92 Gpbs while processing 32 bits of input data at a time, with possibility of scaling datapath to 64, 128 and 256-bits, with theoretical throughput of up to 40 Gbps. In [10] chips that can process 32 and 64-bits in parallel have been implemented in 0.35 micron technology. Chips operate on 180 MHz, with maximum throughput of 5.76 Gpbs for 32 bit and 64 bit processed every clock cycle. In [11] authors identified a recursive formula from which they derived parallel implementation, achieving maximum of about 4.3 Gbps while processing 32 bits at a time, occupying only 162 LUTs. [12] reports that by using 4 GFMAC (Galois Field multiplication and accumulation), 32 bit CRC can be generated in 2 to 3 cycles for 128 input bits, on a circuit with maximum clock speed of 200 MHz.

7. CONCLUSION

In this paper we present compact architecture of Slicing-by-N_{64p} algorithms with 64-bit generator polynomial on an FPGA, where $N \in \{8, 16, 32, 64, 128\}$. The algorithms are capable of reading and processing 64, 128, 256, 512 and 1024 bits at a time, respectively. Since many network protocols implement CRC in software, we also implemented Slicing-by-N_{64p} algorithms in software and ran programs on four state-of-the-art Intel processors, to measure its performance and understand its limitations. We also study tradeoffs between using distributed memory and embedded BRAMs during implementation on an FPGA.

The maximum throughput of software implementation, in the best case scenarios, was attained by Slicing-by-8_{64p} algorithm and it reached 2.6 Gbps running on Intel Xeon 3 GHz with 4 MB of L2 cache. Other algorithms that process more bits at a time were not able to maintain increase in throughput, because data from lookup tables could not fit into the cache, causing many cache misses. Time to fetch data from RAM added to a total execution time, consequently decreasing throughput.

Our results show that by increasing number of processed bits at a time, we can maintain continuous throughput growth on an FPGAs, reaching maximum of 347.37 Gbps for processing 1024 bits at a time with BRAM-based approach. FPGA's have ability to access lookup tables in parallel, thus increased number of lookup tables in algorithms didn't affect its performance. For every expansion in number of bits processed at a time, LUT based approach used more resources for algorithm's tables and other logic, which significantly affected routing delay. However, deployment of BRAMs doesn't add to routing delay, but adds only fixed amount of delay to logic. This property enables further expansion in number of bits processed at a time, enabling continuous growth in throughput for FPGAs. In terms of hardware cost, BRAM-based approach is most efficient solution. Throughput per slice (TP/S) in this setting is 19 - 78% higher than LUT-based approach, depending on an algorithm.

The maximum speedup of hardware implementation is gained compared to throughput achieved on Intel Xeon 3.2 GHz processor with 2MB of L2 cache and our implementation is 606x faster. This processor is fastest processor in our pool of available processors, but since it has least amount of cache, execution of programs took longer time to complete. This shows that algorithms are dependent not only on processor's speed, but also the size of its cache unit.

The software's flexibility to adapt to other CRC standards is also present in implementation on an FPGA, since only contents of the tables have to be changed. To support standards with different lengths of polynomial, our architecture needs minor changes.

Acknowledgments

This work is supported in part by a Grant-in-Aid for the Global Center of Excellence for High-Level Global Cooperation for Leading-Edge Platform on Access Spaces from the Ministry of Education, Culture, Sport, Science, and Technology in Japan.

8. REFERENCES

[1] Alteon Networks. Extended Frame Size for Next Generation Ethernets. staff.psc.edu/mathis/MTU/AlteonExtended Frames_W0601.pdf.

[2] Amila Akagic and Hideharu Amano. An FPGA Implementation of CRC Slicing-by-N algorithms. In *IEICE Tech. Rep.*, volume 110 of *RECONF2010-42*, pages 19–24, Fukuoka, Nov. 2010. Mon, Nov 29, 2010 - Wed, Dec 1 : Kyushu University (VLD, DC, IPSJ-SLDM, CPSY, RECONF, ICD, CPM).

[3] David Law John D'Amrosia and Mark Nowell. 40 Gigabit Ethernet and 100 Gigabit Ethernet Technology Overview. Ethernet Alliance.

[4] Frank L. Berry Michael E. Kounavis. A Systematic Approach to Building High Performance Software-Based CRC Generators. In *ISCC '05: Proceedings of the 10th IEEE Symposium on Computers and Communications*, pages 855–862, Washington, DC, USA, 2005. IEEE Computer Society.

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

[6] Xilinx. Virtex-6 Libraries Guide for HDL Designs, July 2010. http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/virtex6_hdl.pdf.

[7] D.C. Feldmeier. Fast software implementation of error detection codes. *Networking, IEEE/ACM Transactions on*, 3(6):640 –651, dec. 1995.

[8] S.M. Joshi, P.K. Dubey, and M.A. Kaplan. A new parallel algorithm for CRC generation. In *Communications, 2000. ICC 2000. 2000 IEEE International Conference on*, volume 3, pages 1764 –1768 vol.3, 2000.

[9] C. Toal, K. McLaughlin, S. Sezer, and Xin Yang. Design and Implementation of a Field Programmable CRC Circuit Architecture. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 17(8):1142 –1147, aug. 2009.

[10] T. Henriksson and Dake Liu. Implementation of fast CRC calculation. pages 563 – 564, jan. 2003.

[11] G. Campobello, G. Patane, and M. Russo. Parallel CRC realization. *Computers, IEEE Transactions on*, 52(10):1312 – 1319, oct. 2003.

[12] H.M. Ji and E. Killian. Fast parallel CRC algorithm and implementation on a configurable processor. volume 3, pages 1813 – 1817 vol.3, 2002.