An FPGA Implementation of a Modified Version of RED Algorithm

Fariborz Fereydouni_Forouzandeh and Otmane Ait Mohamed
Department of Electrical and Computer Engineering
Concordia University, Montreal, Canada
f_fereyd@ece.concordia.ca and ait@ece.concordia.ca

Abstract

Receiving large number of data packets at different baud rates and different sizes at gateways in very high-speed network routers may lead to a congestion problem and force them to drop some packets. Several algorithms have been developed to control this problem. Random Early Detection (RED) algorithm is well commonly used. In this paper, we present an FPGA implementation of a modified version of RED able to run as fast as 10 Gbps. Furthermore, we discuss three enhancements of the RED algorithm leading a better performance suitable for FPGA implementation.

1. Introduction

Bursty data traffic causing global synchronization leads to congestion problem, and this persuaded the researchers to innovate some mechanisms to avoid it. Among them the RED algorithm is well known mechanism to improve TCP's performance and has alleviated it efficiently. In this paper, we describe an FPGA implementation of RED, as well as, some improvements leading to more efficiency in average computation which will reduce packets loss. The rest of the paper is organized as follow: In Section 2, we briefly introduce the RED algorithm and discuss some of its features and advantages. Section 3 discusses our new FPGA implementation of RED. In section 4, we present our verification results. Finally, we conclude and discuss our future research plan.

2. RED (Random Early Detection)

In RED algorithm there are two threshold levels "Min_th" and "Max_th" as Minimum and Maximum threshold levels of Average queue length (Avg). RED algorithm calculates new Avg on every arriving packet and verifies if it is between the two threshold levels and then decides to drop based on this Avg instead of the queue length (q)[1]. As long as the Avg is below the Min_th, the newly arrived packet is queued. If the Avg is above the Max_th, the newly arrived packet is dropped. When the Avg is between the Min_th and Max_th the decision is based on three parameters: the probability factor Pb, a random number R (R in [0,1]) and a counter C which counts the number of queued packets since the last drop decision. Pb represents a uniform distribution over Min_th and Max_th. It is computed by the following formula:

$$Pb = Max\_P. (Avg - Min\_th)/(Max\_th - Min\_th)$$

where Max_p is a constant (Max_p in [1/30, 1/60]). Finally the drop decision is based upon the evaluation of the following conditions:

If C > 0 and C ≥ (Approximation of [R / Pb]) then
Drop the arrived packet

3. Implementation

In this paper, we propose an implementation of the RED algorithm that supports up to 10 Gbps targeting the XC2VP30 Virtex_II Pro from Xilinx FPGAs. First let discuss the timing requirements. At 10 Gbps, there is no enough time for arithmetic floating point multiplication, division or powers. We should complete all operations on the current packet before the next one come in. In addition, we should take into account the required time for external interface to scan or sample the result. In order, to estimate the maximum processing time of a given packet, we must consider several minimum size Ethernet packets, (72 – 8 = 64 bytes), arriving back to back. We should subtract 8 bytes from the 72 since the 8-bytes for Preamble and SFD fields are automatically generated by the recipient hardware [3]. This processing time should be equal to the necessary time to scan 64 bytes at 10 Gbit/s. If P_Time is the above time period, then;

$$P\_Time = 64 \text{ bytes} / (10 \text{ Gbit/s}) = 51.2 \text{ ns}$$

For more reliability in our design, we assume the P_Time as 48 ns. We decompose this time into eight 6 ns stages "S0 to S7". Thus the design requires a 166 MHz internal clock. S0 to S7 are used to register some results to be used in next stages but it does not mean that we have a pipelined design since packets arrive in different sizes and a random gap time

between them. This frequency is obtained by using the doubled output clock "CLK2X" provided by Xlinx FPGA as an internal clock. This allows us to provide an external clock of 83.33 MHz while clock specifications have been improved. Especially falling and rising skews. This is achieved by using the DLL (Delayed Locked Loop) feature of the Xlinx FPGA which provides a zero delay skew after it is locked in 3 or 4 clock cycles after reset.

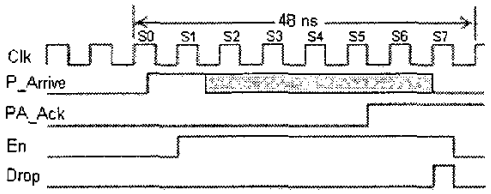

Figure-1: Timings of Drop signal

Duty-Cycle-Correction is another DLL's feature that allowed us to use both falling and rising edges of the clock. Figure-1 gives an example of a drop decision as it is taken within the 48 ns time frame. In this figure *P_arrive* indicates the arrival of a new packet, *En* signal enable the process and the *PA_Ack* signal acts as an Acknowledgement. In order to speed-up the process we have implemented the multiplication and division by add and shift operations. So, the results of multiplications and divisions are chosen very close to positive or negative powers of "2" because the RED algorithm is based on approximation [1]. In following subsection we present our implemented changes to the original RED algorithm in order to increase its efficiency.

1. Bing & all [4] proposed a way how to reduce the excessive packet dropping after a long-term congestion by halving the Avg on every arrival packet if the Avg is above Min_th. A closer look to the algorithm reveals that there is still some excessive packet dropping, because waiting for the queue to decrease as one half could still take much time while dropping all arrived packets before it halves. In this design we have broken down this long step to several small steps before it is halved, trying to get closer to queue length as long as Avg is bigger than q and Min_th. Step resolution of the Avg is a power of "2". As shown in Figure-2, effect of S_ODA (Stepped Over Drop Avoidance) is illustrated comparing with LPF (Low Pass Filter) (dashed) and ODA (Over Drop Avoidance) (thick gray). Step resolution in this design has been chosen as 2^4=16 and our algorithm for LPF/S_ODA becomes:

after every packet arrival and long-term congestion do
if (Avg>Min_th) and (q < (Avg – (Avg / Steps))) then
Avg <= Avg – (Avg / Steps)
else
Avg <= (1-w)Avg + wq
End if
End do

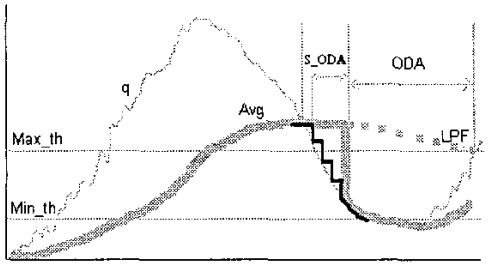

Figure-2: LPF/ODA compared with S-ODA

2- In the old RED algorithm, the new Avg is calculated only after every arriving packet. In our design we calculate the Avg after each packet is sent too. The advantage could be seen in the case where there is no packet arrival for a while such that queue is freeing up spaces due to sending packets and Avg is not affected, so it keeps invalid high value previously stored in it which may cause many excessive packet drops depending on Min_th and Max_th levels while queue has enough room to accommodate them.

3- In our implementation we have considered the actual packet length in bytes instead of the number of packets. This is in contrast to what it's done in [1] where it's assumed that the entire arrived packets are of the same length. So, they proposed two constants C1 (very small, say, almost equal to 2 ^ (-20)) and C2 to calculate Pb as follow:

C1 = Max_p / (Max_th – Min_th)
C2 = C1 . Min_th
Pb = C1.Avg – C2

In our case, since we are considering different arrival packet sizes, we have used a coefficient proportional to the Maximum packet size as a ratio of arrived packet size which must be interpolated to calculate the Pb as below:

Pb = (P_Size / Max_P_size).(C1.Avg – C2)

This implies that we are dealing with very small floating point numbers say, 2 to power of (-20) depending on Max_p, Max_th and Min_th. Although calculation is based on approximation, there is still no easy and fast way to find a good approximation for the result without using any multiplier or divider. In this paper a fast and easy heuristic method is exploited to do this job, of course with appropriate approximation. Note that the constant Max_P is already chosen:

Pb <= (P_size / Max_P_size) (Max_p) [(Avg – Min_th)/ (Max_th – Min_th)] E (1)
Packet Size ratio : (P_Size / Max_P_Size)
Already assumed : Max_p = 2 ^ (-5) E (2)
Avg distribution : (Avg – Min_th) / (Max_ th – Min_ th)
Packet Size Ratio: (64/1518) ≤ Packet Size Ratio ≤ 1
2 ^ (-4) ≤ Packet Size Ratio ≤ 2 ^ (0) E (3)
2 ^ (-5) ≤ Avg Ratio ≤ 2 ^ (0) E (4)

Assuming 8K (or 8192) and 24 K (or 24576) for nonflexible Max_th and Min_th in this design after a

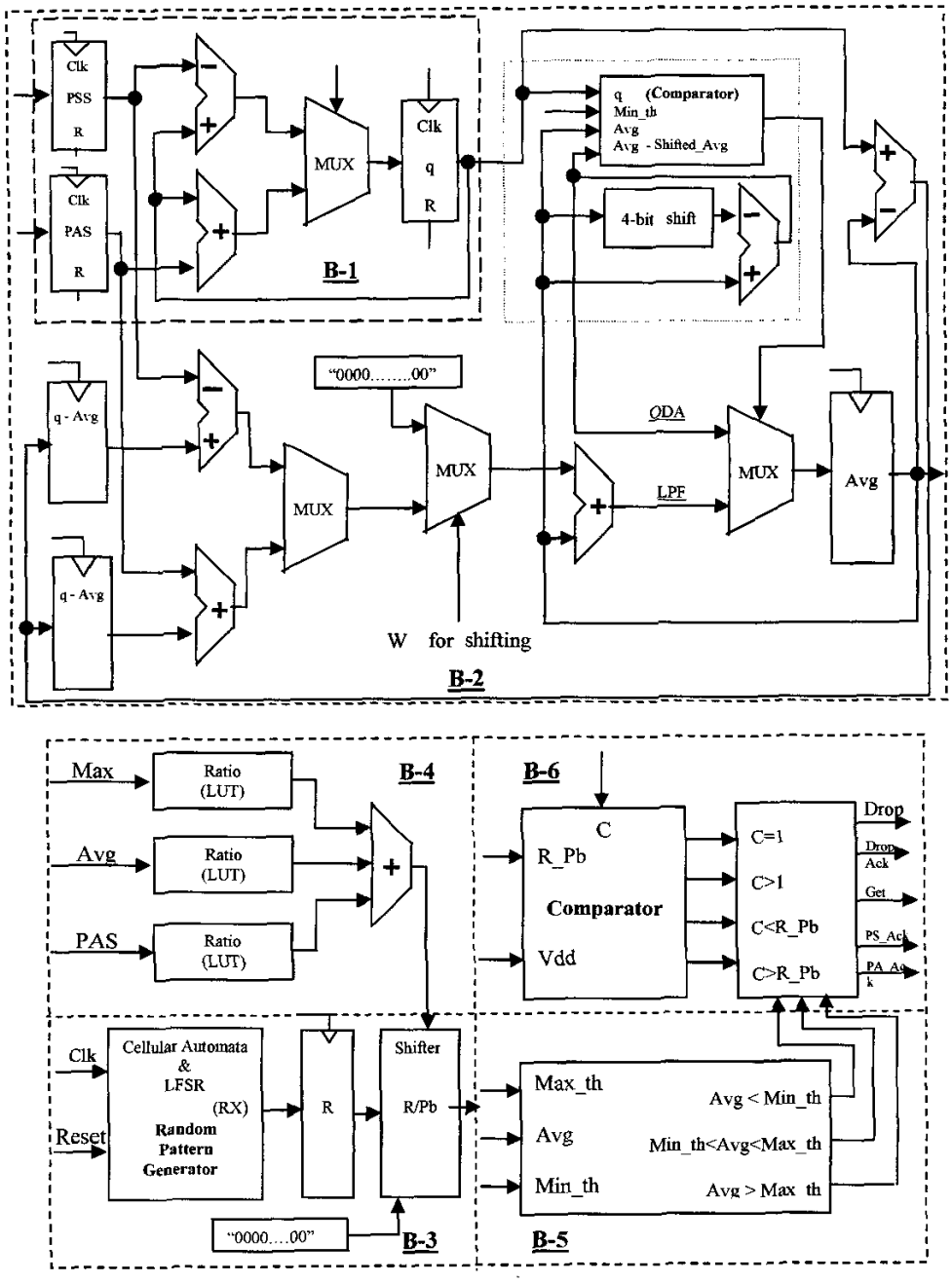

Figure-3: Block Diagram of the modified RED Algorithm

nonlinear approximation "six divide by two levels" between them we get the above Avg Ratio range E (4). In worst case we have to compromise with a maximum of 25% error for Avg ratio that is almost the same as RED approximation. Now by substituting the values of equations (2), (3) and (4) in (1) we get:

$$\begin{aligned} & Pb \rightarrow 2^{(-5)} * 2^{(0 \text{ to } -5)} * 2^{(0 \text{ to } -4)} \\ \text{or} \quad & Pb \rightarrow 2^{(-14)} \le Pb \le 2^{(-5)} \\ \text{then} \quad & R/Pb = R \,/\, (2^{(-5)} \text{ to } 2^{(-14)}) \\ & = R * (2^{(+5)} \text{ to } 2^{(+14)}) \end{aligned} \qquad E(5)$$

Since R is a random number in [0, 1] , then the result of equation (5) would be in the range [2^(5) , 2^(14)], therefore equation (5) implies to a 14-bit random number shifted between 0 to 9 bit to the right. Max_p is a constant, and Avg Ratio and

P_Size Ratio could be easily obtained from look up table. This implementation is illustrated in a few blocks in Figure-3 that describes the block diagram of our implementation where the input signals PA, PS, PAS(16-bit), PSS(16-bit), W(2-bit) represent Packet Arrival, Packet Sent, Arrived Packet Size, Sent packet size, and the weight of the queue, respectively. W is in the range of 2^(-6) and 2^(-9). In Block B-1 every arrived or sent packet size is increased or decreased by old queue length (q) even if both happen at the same time. Block B-2 calculates new Avg on every arriving or sent packet considering weight of queue "w", as it's done in LPF/S_ODA (see section 3. 1.). In case of LPF, Avg is approximated as [Avg + w (q – Avg)] in two clock cycles after receiving the packet size, otherwise as [Avg – (Avg/Steps)]. Block B-3 is a random Pattern generator. Although it's hard to generate real random numbers, a well efficient method is exploited in this design to generate the random number R necessary for the Drop decision. We have implemented it in our design to generate a 14-bit width random number. Our method consists of composing two standard pseudo random pattern generators; Linear Feedback Shift Register (LFSR) and Cellular Automata (CA) which is series of cells (Flip Flops). Our 14-bit random pattern generator generates up to 97% coverage out of 2^(14) =16384 before it repeats. On the other hand, sampling a random number (R) is done due to unknown events based on arrival packet time, and this makes a well likely real random number. Inputs to Block B-4 are Avg, Max_p, P_Size and R. As explained by the equations (2), (3) and (4) it provides easily R/pb. Block B-5 compares the Avg with Min_th and Max_th levels and issues the result signals to the final control unit. And Block B-6 receives all the provided signals and information and makes the appropriate decision to drop or get the arrived packet. Outputs of this block are Drop, Valid_Drop, Get, and PA and PS Acknowledgments.

4. Timing results and properties of design

The Design has been synthesized and implemented into an XC2VP30 Virtex_II Pro from Xilinx FPGAs. Timing results have met the specifications. According to the final Xilinx synthesis reports, every stage has responded in less than 6 ns and since the hardware is well parallelized, thus there is still room to place more optional features for future work. The RTL and behavioral versions have been simulated at the same start time supplying by a unique packet generator. Expected results have been achieved. Two simultaneous portions of the waveforms are shown in figure-4 as a sample of the results. The black filled portions show the burstiness of Drops "while: Avg is above Max_th", and each single bar illustrates a Packet Drop "while Avg is between Min_th and Max_th". As we can see, the number of Drop signals is almost the same for both cases. In this simulation input load is about 1.4 times the sending data load. There are obviously reasons for the differences. First, in RTL, calculations are based on approximation, but not in the behavioral model. Secondly, generating different random numbers in each abstraction level results in having drop decision in different times but their number is almost the same.

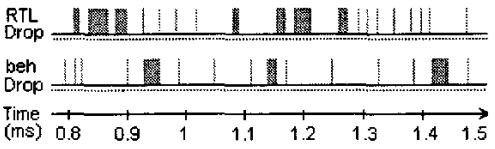

Figure-4: Drop events in behavioral and RTL.

5. Conclusion

In this paper, we have presented a new implementation of the RED algorithm incorporating several enhancements to avoid excessive packet drops and to achieve more improvements in average calculation. The method which is used for approximated calculations in order to speed up the process, calculating the average queue size on every packet sent, exploiting some features of the FPGA's DLL are major improvements. The design is scalable for higher frequency than 10 Gbps. As future work, we'll implement the pseudo packet generator inside the same FPGA (where the design is implemented) as a built in self test. Internal values to select nonflexible constants [1] Min_th and Max_th, and also dynamically choosing the appropriate constant value for weight of queue (W) which is very effective for example in low traffic situation after a long term congestion, would be well efficient.

6. References

[1] Sally Floyd, and Van Jacobson, "Random Early Detection Gateways for Congestion Avoidance", IEEE/ACM Transactions on Networking, Lawrence Berkeley Lab University of California, August, 1993,

[2] Sally Floyd and van Jacobson, "Random Early Detection Gateways for Congestion Avoidance", IEEE/ACM Trans. Networking, 1993 Vol.1, No. 4, (Aug 1993), pp.397-413.

[3] Scott Karlin and Larry Peterson, "Maximum Packet Rates for Full-Duplex Ethernet", Technical Report TR_645-02, Department of Computer Science, Princenton University, February 14,2002,

[4] Bing Zheng and Mohammed Atiquzzaman, "Low Pass Filter/Over Drop Avoidance (LPF/ODA): Int. J. Commun. Syst. 2002; 15:899-906 (DOI: 10.1002 / dac.571), School of Computer Science, Univ. of Oklahoma, Norman, OK 73019, USA. 18, Oct, 2002.