Worst Case Latency Minimization of HopliteRT FPGA-based NoC for Real-time Applications

Marat Bekmyrza, 20973628

Introduction

Common characteristic of many practical problems today is data intensive workloads. For example, in autonomous vehicles computer vision algorithms require processing of massive amounts of images in a timely manner. Distributed computation over multiple processing elements (PE) is trending among high performance state-of-the-art solutions to work with large data. However, their performance is contingent upon communication mechanism established between PEs. Therefore, low cost and high throughput are desirable parameters for the choice of communication interface. Such constraints are well suited for Networks-on-chip (NoC) designed to fit multiple PE systems. Hoplite [1] is an FPGA-based NoC that uses deflection routing to reduce the implementation cost of switches and remove the need for expensive buffers. The idea of deflection routing is to intentionally misroute one of the packets conflicting over an output port of the switch. However, in this design there is a possibility that an unlucky packet might always get misrouted and never reach its destination. Such behaviour is not acceptable for real-time applications that require guaranteed worst-case routing latency. Moreover, bufferless setup forces the switch to give the lowest priority to PE injection port so that valid packets from other ports are not lost. This might cause infinite source queueing delay for an unlucky PE which is always stalled due to continuous traffic by other peers. HopliteRT [2] aims to remove such livelocks and introduce deterministic upper bounds on the worst-case routing latency with the following modifications on the baseline Hoplite design: new routing strategy and two extra counters at the client side to deploy token bucket regulation policy at injection ports. The former helps to eliminate infinite travelling time of a misrouted packet; the latter enforces a limit on the waiting time to inject a packet by a source node which was previously stalled due to endless traffic by other nodes. Routing strategy of HopliteRT has been well optimized to increase the clock frequency and minimize the resource utilization (LUTs and FFs) of implementation on a Xilinx Virtex-7 FPGA. Thus, this project targets the PE side modification of HopliteRT: optimization of the token bucket regulator to minimize the total worst case source queuing delay. Token bucket regulators are realized using two cascaded counters that count up to certain controllable values denoted as regulator period $1/\rho$ and burstiness σ. Scope of this work is burstiness parameter optimization which was left as a future work by authors in [2].

Literature Review

A. Switch

HopliteRT topology is a unidirectional torus as shown in Fig. 1. Each node in Fig. 1 contains a switch and a PE wrapper with data transfer lines shown in Fig. 2. In torus topology, Hoplite uses Dimension Ordered Routing (DOR) policy [2]: packets traverse first in the X-ring (horizontal) towards desired X-coordinate, and then in the Y-ring (vertical) until the destination Y-coordinate is reached.

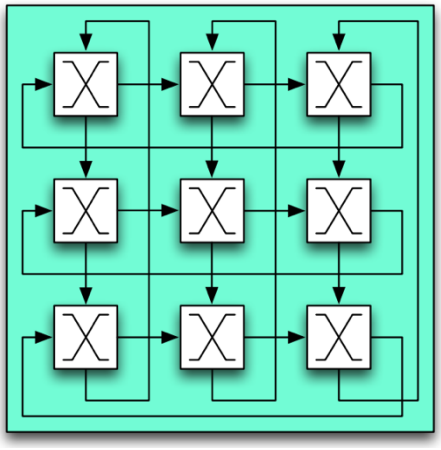

Figure 1. NoC topology: 2D Unidirectional Torus [1].

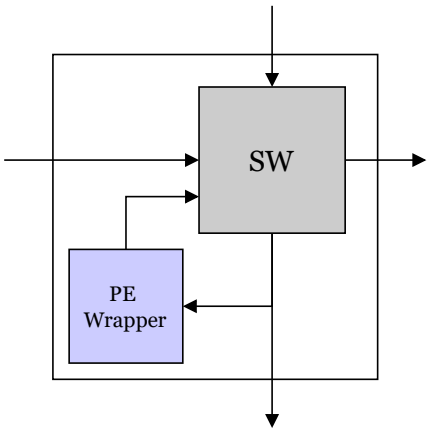

Figure 2. Data transfer lines of a NoC node: Switch and PE Wrapper.

In Fig. 2, each data transfer line includes data itself, valid signal and destination address of the packet. Switch module has three inputs for data transfer W (West), N (North), and PE and two outputs E (East) and S (South). S is also an input to PE wrapper. Switch contains a dual 3-to-1 mux, shown in Fig. 3, that routes inputs to corresponding output ports based on arbiter logic.

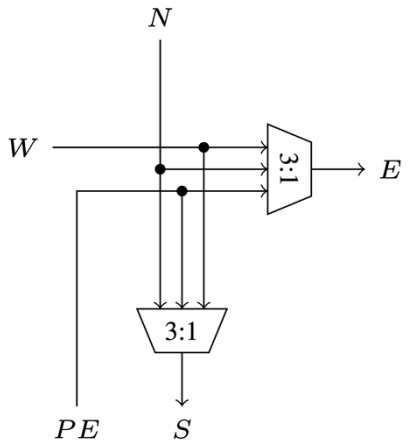

Figure 3. Microarchitecture of a switch router: dual 3-to-1 mux.

Switch router in Fig. 3 needs two 6-LUTs in Xilinx architecture due to 4 select bits + 3 input data bits. However, [2] suggests restricting some of the routing options to reduce select signal width down to 2 bits (reuse them for both muxes with different interpretations) and map the design to two 5-LUTs (which is one fracturing 6-LUT). This is the main optimization made on the switch side in [2].

Baseline Hoplite [1] had no path from N to E. Thus, any W packet was always mis-routed when there was a conflict with N over S. HopliteRT [2] adopts a new routing policy that allows N to E path as in Fig. 3. The key idea now is to always prioritize W traffic over N traffic. This allows X-ring packets to be conflict free, while Y-ring traffic can get deflected. However, deflected Y-ring packet will have higher priority over any other Y-ring traffic once it deflects onto the X-ring. In this way, packet deflects at most once on each row. While it could get deflected infinitely many times in the baseline design.

Table I and Table II show the arbitration logic of HopliteRT routing policy. This design has no PE→E + W→S route to accommodate switch mapping on two 5-LUTs by sharing common mux select signals. PE packets have the lowest priority and stall on conflicts.

W valid	N valid	PE valid	W wants S	PE wants S	Route
1	x	x	1	x	W→S + N→E
1	1	x	0	x	W→E + N→S
1	0	x	0	x	PE→S + W→E
0	1	x	x	x	PE→E + N→S
0	0	1	x	1	PE→S + W→E
0	0	1	x	0	PE→E + N→S
0	0	0	x	x	x

Table I. HopliteRT routing policy ('x' denotes 'don't care' values).

sel1	sel0	Route
0	0	W→E + N→S
0	1	W→S + N→E
1	0	PE→E + N→S
1	1	PE→S + W→E

Table II. Mux selects.

B. Processing Element

In Fig. 2, PE wrapper contains one PE and two counters. PE communicates with its associated switch using AXI interface protocol. In Fig. 4, two cascaded counters realize token bucket regulator at the PE injection port. Regulator parameters are ρ (rate of packet injection < 1 packet/cycle) and σ (maximum burst of consecutive packets ≥ 1). Free-running rate counter overflows after reaching $1/\rho$ and at each overflow one token is added to token counter. Token counter saturates at σ. PE is allowed to send a packet only when NoC is ready, and it has an available token. After packet transfer occurs, one token is drafted from the bucket.

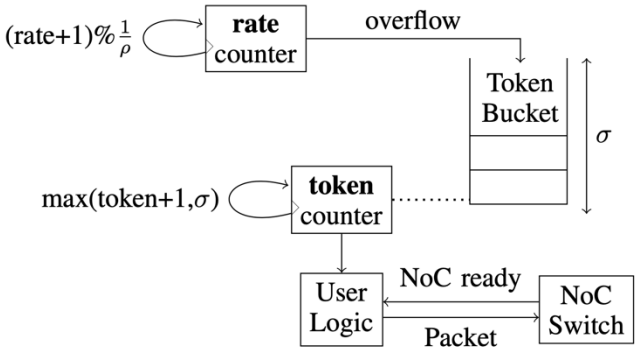

Figure 4. Cascaded counters that realize token bucket regulation policy [2].

C. Worst Case Latencies

There are two types of packet delays: in-flight and source queuing. The first one is total travelling time of an injected packet in the network. The second is the waiting time of the source node before it is allowed to inject a packet. In HopliteRT router strategy, Y-ring packet can be deflected at most once in each row. Therefore, worst-case in-flight latency T^i between nodes (X_1, Y_1) and (X_2, Y_2) on an $m \times m$ network [2]:

$$T^i = \Delta X + \Delta Y + (\Delta Y \times m) + 2$$

where, $\Delta X = (X_2 - X_1 + m)\%m$ and $\Delta Y = (Y_2 - Y_1 + m)\%m$ are the distances in the unidirectional torus topology shown in Fig. 1, and $\Delta Y \times m$ is the maximum distance due to deflections. Additional 2 is the latency due to source PE injection and destination PE exit cycles. In-flight delay only depends on the router policy, and it has already been fully optimized. Therefore, focus of this work is mainly on minimizing the source queuing delay.

Before analysing source queuing delay, concepts used in [2] should be introduced:

- flow f – transfer of sequence of k packets from source node to destination node with defined rate $f.\rho$ and burstiness $f.\sigma$ (in this work, network traffic has maximum of 1 flow/node).
- set of conflicting flows Γ_f^C of f – flows that block the injection of packets at the analyzed client (e.g., f_1 from $(0,0)$ to $(0,3)$ block injections of client at $(0,1)$ to S port).
- $\rho(\Gamma_f^C)$ and $\sigma(\Gamma_f^C)$ are cumulative sum of rates and burstiness, respectively, of all conflicting flows.

Then the delay T^s caused only by the burstiness of conflicting flows is as follows [2]:

$$T^s = \left\lceil \frac{\sigma(\Gamma_f^C)}{1-\rho(\Gamma_f^C)} \right\rceil$$

which is proportional to the cumulative burst length $\sigma(\Gamma_f^C)$, and inversely proportional to the available injection rate $1-\rho(\Gamma_f^C)$. Additionally, there is a possible delay due to empty token regulator.

Then, **Theorem 2** in [3] states that:

Assume $\rho(\Gamma_f^C) < 1$ and the client wishes to inject a sequence of $k \leq f.\sigma$ packets for flow f. Then the delay to inject all packets in the sequence is upper bounded by:

$$T^k = \left\lceil \frac{1}{f.\rho} \right\rceil - 1 + T^s + \left\lceil (k-1) \cdot \max\left(\frac{1}{f.\rho}, \frac{1}{1-\rho(\Gamma_f^C)} \right) \right\rceil$$

Note that condition $\rho(\Gamma_f^C) < 1$ is necessary for a flow to not suffer starvation [3]. The breakdown of the above delay is as follows [2]:

- the first packet in the sequence could suffer delay of $\left\lceil \frac{1}{f.\rho} \right\rceil - 1 + T^s$ which is the sum of delays due to token bucket regulation $\left\lceil \frac{1}{f.\rho} \right\rceil - 1$ and waiting time T^s for a free cycle due to conflicting flows.
- each successive packet experience delay of $\max\left(\frac{1}{f.\rho}, \frac{1}{1-\rho(\Gamma_f^C)} \right)$.

The above analysis holds for the sequence of packets at most equal to the burst length $f.\sigma$. The intuition suggested in [2] is that token bucket 'fills up' while the flow is being blocked by conflicting flows. In the extreme case when $f.\sigma = 1$, every packet in the sequence would experience the delay as if they were the first packet, i.e., $\left\lceil \frac{1}{f.\rho} \right\rceil - 1 + T^s$. Whereas, if $f.\sigma$ is large enough, token bucket could fill up while the successive packets are waiting for the conflicting flows to resolve, and they would not suffer both regulation and conflicting flow delays, only the one which is higher. However, large $f.\sigma$ increases T^s for other conflicting flows. When burst lengths $f.\sigma$ can be chosen freely, [2] suggests computing them by formulating an optimization problem.

Proposed Solution

The optimization problem is to minimize T^k over $f.\sigma$ values for each flow. Before formalizing the objective function, case when $k > f.\sigma$ should be considered. When sequence length is higher than the burst length, it can be concluded from the analysis in [2] that the total waiting time T_f of flow f would be:

$$T_f = \left\lceil \frac{k}{f.\sigma} \right\rceil T^{f.\sigma}$$

This equation agrees with the previous discussion that when $f.\sigma = 1$, every packet would experience a delay of the 'first' packet which is $T^1 = \left\lceil \frac{1}{f.\rho} \right\rceil - 1 + T^s$ and total $T_f = kT^1$. When $f.\sigma = k$, total waiting

time would be $T_f = T^k$ which is the same as if we used previous equation for the condition of $k \leq f.\sigma$. When $1 < f.\sigma < k$, then sequence of packets is divided into $\left\lceil \frac{k}{f.\sigma} \right\rceil$ batches of maximum length $f.\sigma$ and each batch could suffer maximum delay of $T^{f.\sigma}$.

Finally for a flow f with parameters $f.\sigma$ and $f.\rho$, total source queuing delay for a sequence of k packets:

$$T_f = \begin{cases} T^k, & k \leq f.\sigma \\ \left\lceil \frac{k}{f.\sigma} \right\rceil T^{f.\sigma}, & k > f.\sigma \end{cases}$$

where T^k is:

$$T^k = \left\lceil \frac{1}{f.\rho} \right\rceil - 1 + \left\lceil \frac{\sigma(\Gamma_f^C)}{1 - \rho(\Gamma_f^C)} \right\rceil + \left\lceil (k-1) \cdot \max\left(\frac{1}{f.\rho}, \frac{1}{1 - \rho(\Gamma_f^C)} \right) \right\rceil$$

Our **objective** is to minimize sum of all total delays for each flow f in the given network traffic:

$$\min_{f.\sigma} \sum_{f \in \text{traffic}} T_f$$

$$\text{s.t. } f.\sigma \geq 1, \qquad \forall f \in \text{traffic}$$

For a given network traffic, following values can be considered as constants for each flow f: rate $f.\rho$, set of conflicting flows Γ_f^C, consequently sum of rates of conflicting flows $\rho(\Gamma_f^C)$.

The first step in solving this problem was to test for the convexity by generating an example network traffic and checking the hessian matrix of the objective function. It turns out that this function is not convex even with the dropping of the ceiling operators. Therefore, we cannot rely on Gradient Descent methods. Another approach was to apply ADMM since our sum objective can be separated into individual functions and minimized using distributed optimization techniques. However, [4] suggests that ADMM does not guarantee convergence for nonconvex functions. Therefore, the choice has fallen to Simulated Annealing method that could probably provide a solution outside of local minima points. It was implemented using built-in `simulannealbnd` function provided in 'Global Optimization Toolbox' with default parameters.

Experimental Results

Network dimensions are (m, m). Optimization variable x has the same dimension of (m, m) since each node produces one flow. It has lower bound: $x(i, j) \geq 1$. Injection rate was set to $\rho = 1/m^2$ as in [2]. For each flow there is a length of packets sequence of $k \geq 1$. For the whole network it is represented as matrix K of dimension (m, m). Next, we need to represent set of conflicting flows. The choice was to use adjacency matrix representation A of size (m^2, m^2). For each m^2 flows (1 flow/node) in the network we have possible $m^2 - 1$ conflicting flows. Entry $A(i, j)$ is 1 when f_i has flow f_j in the set of conflicting flows, and 0 otherwise. Table III summarize variations in the network and traffic patterns used for different experiment setups. 10% sparsity of A means network is experiencing heavy traffic and there are many conflicting flows.

Parameter	Variations
m	value: [2, 4, 8, 16]
K	max value: [10, 100, 1000]
A	sparsity: [10%, 50%, 90%]

Table III. Variations of network and traffic parameters.

Table IV shows some the iteration results. A and K values are generated using uniform random function. Small $\boldsymbol{x0}$ means it is initialized as all 1's, whereas large means all 100's.

Sparsity A	Max Value K	Boundary $x = K$	Optimized x small $x0$	Optimized x large $x0$	Baseline $x = 1$
10%	10	8387	**7446**	8192	7529
10%	100	95929	**84756**	100434	87574
10%	1000	935551	**854208**	881858	918357
50%	100	34703	**33898**	36023	34058
50%	1000	200775	**187258**	196564	198371
90%	100	16545	**16118**	16496	16162
90%	1000	188541	**183394**	188545	184908

Table IV. Comparison of the optimized objective function value with the baseline, $m = 4$.

Discussion and Conclusions

The baseline model [2] has no optimization and sets $f.\sigma = 1$ for each flow in the traffic. However, in this work, it was shown that the optimal value for burstiness parameter can be found using Simulated Annealing method. Table IV shows sample results for network of size $m = 4$. For all the cases, optimized result with small $x0$ initialization shows the best performance. It is interesting to note that even with large $\boldsymbol{K}$ values, objective function value at the boundary $\boldsymbol{x} = \boldsymbol{K}$ is comparable to the baseline $\boldsymbol{x} = \mathbf{1}$. This shows the design trade-off: when burstiness is small, overall total delay increases since each packet experiences delay of the 'first' packet; whereas, for high burstiness, we decrease the delay for successive packets, however, suffer from increase in delays for conflicting flows. Thus, it makes sense that the optimal burstiness is found in between these boundaries, even though upper limit was set to $\mathbf{10}\boldsymbol{K}$.

Further research can be done to apply non-convex ADMM using distributed optimization. Injection rate (it was kept constant in this work) also can be considered as an optimization variable for the next work. In conclusion, project contributions are the following: it derives the objective function to calculate total source queuing delay; and it shows that although total delay for HopliteRT is a non-convex function, it can be minimized using Simulated Annealing by varying the burstiness parameter of the token bucket regulator introduced at the injection port of a client.

References

[1] N. Kapre and J. Gray, “Hoplite: Building austere overlay NoCs for FPGAs,” in International Conference on Field-Programmable Logic and Applications (FPL’15), 2015, FPL, 1–8.
[2] S. Wasly, R. Pellizzoni, and N. Kapre, “HopliteRT: An efficient FPGA NoC for real-time applications,” in International Conference on Field Programmable Technology (ICFPT’17), 2017, IEEE, 64–71.
[3] S. Wasly, et al. Analysis of Worst Case Latency for Hoplite FPGA-based NoC. Technical report, University of Waterloo, ECE, http://hdl.handle.net/10012/12600 [2017].
[4] S. Boyd, et al., “Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers,” Foundations and Trends in Machine Learning, vol. 3, no. 1, pp. 1-122, 2010.

Appendix

Matlab code of the objective function:

```
function y = objective(x, K, A)
% % %
% NoC has dimensions (m,m)
%
% inputs:
% x - matrix of sigma values, size (m,m) - optimization variable
% K - matrix of flow sequence lengths, size (m,m) - constant
% A - adjacency matrix of conflicting flows, size (m^2,m^2) - constant
%
% output:
% y - sum of ojective function values of each node
% % %
x = ceil(x);
m = size(x,1);
rho = 1/m^2;
rhos = rho*ones(m,m);

C_sigmas = zeros(m,m); % sum of conflicting sigmas
C_rhos = zeros(m,m);   % sum of conflicting rhos

for i=1:m
    for j=1:m
        C_sigmas(j,i) = A((i-1)*m+j,:) * reshape(x,m^2,1);
        C_rhos(j,i) = A((i-1)*m+j,:) * reshape(rhos,m^2,1);
    end
end

a = zeros(m,m);
Ts = ceil(C_sigmas./(1 - C_rhos));
a = a + 1/rho - 1 + Ts;
b = max(rhos.^(-1), (1 - C_rhos).^(-1));

y = 0;
for i=1:m
    for j=1:m
        if K(j,i) <= x(j,i)
            % when k <= f.sigma
            a(j,i) = a(j,i) + ceil((K(j,i) - 1) * b(j,i));
        else
            % when k > f.sigma
            a(j,i) = a(j,i) + ceil((x(j,i) - 1) * b(j,i));
            % multiply with number of batches
            batches = ceil(K(j,i)/x(j,i));
            a(j,i) = batches * a(j,i);
        end
        y = y + a(j,i);
    end
end
```

Function call:

```
m = 2; % 2, 4, 8, 16
x0 = ones(m); lb = ones(m);
K = randi(10,m,m); % 10, 100, 1000
A = double(rand(m^2,m^2)>0.1); % 0.1, 0.5, 0.9
A = A.*~eye(size(A)); % no conflict with itself
opts = optimoptions('simulannealbnd', 'MaxFunctionEvaluations', 1e6);
ObjectiveFunction = @(x) objective(x,K,A);
[x,fval,exitFlag,output] = simulannealbnd(ObjectiveFunction,x0,lb,10*K,opts);
```