Implementation of the Link Monitor for a minimal-state Max-Min-Fair Rate Regulation Mechanism

Dimitrios Giannopoulos

Thesis submitted in partial fulfillment of the requirements for the

Masters' of Science Degree in Computer Science and Engineering

University of Crete
School of Sciences and Engineering
Computer Science Department
Voutes University Campus, 700 13 Heraklion, Crete, Greece

Thesis Advisor: Prof. *Manolis G.H. Katevenis*

Thesis Co-Advisor: Dr. *Nikolaos Chrysos*

This work has been performed at and financially supported by the Computer Architecture and VLSI Systems (CARV) laboratory of the Institute of Computer Science of FORTH, through the European-Union-funded projects Euroserver (FP7, GA 610456) and ExaNeSt (H2020,GA 671553)

University of Crete
Computer Science Department

Implementation of the Link Monitor for a minimal-state Max-Min-Fair Rate Regulation Mechanism

Thesis submitted by
Dimitrios Giannopoulos
in partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science

THESIS APPROVAL

Author: ______________________
Dimitrios Giannopoulos

Committee approvals: ______________________
Manolis G.H. Katevenis
Professor, Thesis Supervisor

____
Angelos Bilas
Professor, Committee Member

____
Panagiota Fatourou
Associate Professor, Committee Member

Departmental approval: ______________________
Antonios Argyros
Professor, Director of Graduate Studies

Heraklion, October 2017

Implementation of the Link Monitor for a minimal-state Max-Min-Fair Rate Regulation Mechanism

Abstract

In cloud computing, a large number of servers within the same facility work collectively in order to complete challenging tasks. Current research examines ways to replace the expensive and high-energy-consumption processors of today, with lower-cost, energy-efficient ARM-based processor clusters. These environments will ultimately consist of many lower-capacity end-nodes, potentially increasing the intensity of inter-server communication. At the same time, VM migration, checkpointing and storage contribute to sudden traffic bursts, which are responsible for transient congestion phenomena.

In this thesis, we implement the link monitor logic of a novel congestion control scheme that can replace the TCP congestion control for RDMA transfers. This scheme can (i) throttle the offensive flows, (ii) keep the backlogs outside of the network, and (iii) allocate max-min-fair rates to flows.

The logic per output that we implemented does not require per-flow state inside the network, and relies on simple components that can readily be implemented on network interfaces and on switch outputs.

Furthermore this thesis covers the interpretation of the whole given scheme in abstract hardware architecture, as well as corner case resolution. The corner cases encountered are the division by zero and high division timing and area cost, that required modifications in the division component, abrupt traffic patterns that did not match our system's focus and were resolved with temporary static generic assignments, and communication protocol restrictions that could not be resolved but had to be worked around.

Most of the goals have been met and what is left is well defined for future work. It is the implementation of the per-end-node logic, the thorough practice in bigger scale systems, and the optimizations.

The simulation results demonstrate that the scheme reacts promptly to congestion, while the implementation results, prove that the scheme is scalable as the components only require about 231 LUTs and 171 Flip-Flops of our test environment FPGA.

Υλοποίηση του ελεγκτή συνδέσμων για έναν μηχανισμό ελάχιστης κατάστασης κατανομής Μέγιστου-Ελάχιστου Ρυθμού Ροών

Περίληψη

Στο cloud computing, ένας μεγάλος αριθμός διακομιστών μέσα στις ίδιες εγκαταστάσεις λειτουργούν συλλογικά για να ολοκληρώσουν απαιτητικές εργασίες. Η τρέχουσα έρευνα εξετάζει τους τρόπους αντικατάστασης των δαπανηρών και υψηλής ενέργειας επεξεργαστών του σήμερα, με φθηνότερα, οικονομικά αποδοτικότερα, συγκροτήματα επεξεργαστών αρχιτεκτονικής ARM. Αυτά τα περιβάλλοντα τελικά θα αποτελούνται από πολλούς τελικούς κόμβους χαμηλότερης χωρητικότητας, ενδεχομένως αυξάνοντας την ένταση της επικοινωνίας μεταξύ των διακομιστών. Η μετανάστευση εικονικών μηχανών, η ροή ελέγχου και η αποθήκευση συμβάλλουν σε ξαφνικές ριπές κυκλοφορίας, οι οποίες ευθύνονται για φαινόμενα παροδικής συμφόρησης.

Σε αυτή την εργασία, υλοποιούμε την παρακολούθηση συνδέσμων ενός νέου συστήματος ελέγχου συμφόρησης που μπορεί να αντικαταστήσει τον έλεγχο συμφόρησης του TCP για μεταφορές RDMA. Το σχέδιό μας μπορεί να: (ι) να επιταχύνει τις επιθετικές ροές, (ιι) να κρατήσει τις συσσωρεύσεις εκτός δικτύου, και (ιιι) να κατανείμει μέγιστα-ελάχιστα-δίκαιο ρυθμό στις ροές. Το πρωτόκολλο δεν απαιτεί διατήρηση κατάστασης ροής στο εσωτερικό του δικτύου και αξιοποιεί απλά στοιχεία που μπορούν εύκολα να υλοποιηθούν σε διεπαφές δικτύου και σε εξόδους μεταγωγέων.

Επιπλέον, αυτή η διπλωματική εργασία καλύπτει την ερμηνεία ολόκληρου του δοθέντος μηχανισμού σε αφηρημένη αρχιτεκτονική υλικού, καθώς και την επίλυση γωνιακών περιπτώσεων. Οι γωνιακές περιπτώσεις που αντιμετωπίσαμε είναι η διαίρεση με μηδεν και υψηλό κόστος χρόνου και επιφάνειας διαίρεσης, που απαιτούσαν τροποποιήσεις στο στοιχείο διαίρεσης, απότομου είδους κυκλοφορία που δεν ταιριάζουν με την εστίαση του συστήματος μας και επιλύθηκαν με προσωρινές γενικές στατικές εκχωρήσεις, και περιορισμούς πρωτοκόλλου επικοινωνίας που δεν μπορούσαν να επιλυθούν αλλά έπρεπε το σύστημά μας να προσαρμοστεί πάνω σε αυτούς.

Τούτου λεχθέντος, οι περισσότεροι από τους στόχους έχουν επιτευχθεί και ό,τι έχει απομείνει είναι καλά καθορισμένο για μελλοντική εργασία. Πρόκειται για την υλοποίηση της λογικής ανά τελικό κόμβο, της εμπεριστατωμένης πρακτικής σε συστήματα μεγαλύτερης κλίμακας, και τις βελτιστοποιήσεις.

Τα αποτελέσματα προσομοίωσης αποδεικνύουν ότι το σύστημα μπορεί να αντιδράσει άμεσα στην συμφόρηση, ενώ στην εφαρμογή, αποδεικνύεται ότι το σχέδιο είναι κλιμακώσιμο καθώς η εφαρμογή απαιτεί περίπου 231 LUTs και 171 Flip-Flops της FPGA του δοκιμαστικού μας περιβάλλοντος..

Acknowledgements

This thesis was financially supported by the Institute of Computer Science (ICS) and the Foundation for Research and Technology - Hellas (FORTH), and more specifically the financial support that was provided through the European-Union-funded projects Euroserver (FP7, GA 610456) and ExaNeSt (H2020,GA 671553).

To me, this thesis is important, but not by itself. What is more important to me is the road to here, from the beginning of my studies. For this, I would like to thank my advisor, professor Manolis G.H. Katevenis, for accepting me as volunteer in the laboratory of Computer Architecture and VLSI systems (CARV) of the Foundation for Research and Technology - Hellas (FORTH) in 2013, giving room for many opportunities. I would like to thank George Kalokerinos for his guidance during my first months of practice, and everyone in the CARV laboratory for their great support during all this time.

Furthermore I would like to thank George Constantinidis, Vangelis Vassilakis and Stelios Mavridis for their extensive assistance during my studies, and Apostolis Glenis and Vassilis Papaefstathiou for assisting my survival during my Erasmus studies in Sweden that amounted greatly in my personal development.

More specifically on my master's studies, I would like to thank again professor Manolis G.H. Katevenis for advising me through my studies, for the great opportunity on the thesis subject and for his input on the work. I would like to thank Nikolaos Chrysos for supervising my studies and my thesis, and Apostolis Glenis for his advice and overall support during the two years. I would like to thank Antonis Psistakis, for his extensive feedback and proofreading of the thesis as well as Giannis Vardas and Nikolaos Chrysos for the simulation inputs and the feedback on this thesis.

I would like to thank the student secretariat, and especially Mrs Stella Syntichaki who acted like a second mother to me and helped me through everything despite her constant heavy workload.

Last but not least, I would like to thank my family and friends for their love and support, for being by my side in good times and not losing faith in me during misfortune.

You are always a student, never a master.

Conrad Hall

We can only see a short distance ahead, but we can see plenty there that needs to be done.

Alan Turing

A pessimist says the glass is half empty, an optimist says the glass is half full, and an engineer says the glass is too big.

Scott Edward Shjefte

The way to succeed is to double your failure rate.

Thomas J. Watson

Contents

Table of Contents i

List of Tables iii

List of Figures v

1 Introduction 1

1.1 RDMA transfers to replace TCP stack . . . 2
1.2 Prototyping platform . . . 2
1.3 Contributions . . . 3
1.4 Outline . . . 4

2 Max-min-fair Rate Regulation Without keeping Per-flow State (MaRRWiPS) 5

2.1 Terms and definitions . . . 5
2.2 The Congestion Control issue . . . 6
2.3 MaRRWiPS high level description . . . 7
2.4 Max-min Fairness . . . 7
2.5 Fair Share Rate (FSR) Calculation . . . 8
2.6 Realistic Fair Share Rate Allocation . . . 9
2.6.1 Descriptive examples . . . 10
2.6.2 Desired Rate . . . 11
2.7 Optimizations . . . 12
2.7.1 Short-circuit notification . . . 12
2.7.2 Flow Initialization messages . . . 12
2.7.3 Flow Termination messages . . . 13
2.8 Simulation . . . 13
2.8.1 RRP setting . . . 13
2.8.2 Description of experiments . . . 14
2.8.3 Convergence to max-min-fair rate allocation . . . 15
2.8.4 Flow completion times of latency-sensitive flows. . . . 16
2.8.5 Fairness in scenarios with local and remote flows . . . 17

3 Tools, Technologies, and Methodology **19**
3.1 Hardware Resources 19
3.2 Software Resources 19
3.3 Simulation tools 20
3.4 Implementation tools 20

4 Implementation **21**
4.1 High Level Design 21
4.1.1 Current Rate and Desired Rate Representation 25
4.2 Contention Point, the link monitor implementation 26
4.3 Handling corner cases 27
4.4 Implementation on FPGAs 32
4.4.1 AXI Implementation Attempt 33
4.4.2 EXAnet Implementation Attempt 33
4.5 Reaction point at DMA engines 34
4.6 Broadcasting a common RRP clock event 35

5 Conclusion and Future work **37**

Bibliography **39**

A Verilog interpretation examples **i**

List of Tables

2.1 CR/DR disambiguation 12

4.1 FRP layout 24
4.2 FPGA Area utilization for two different clock frequency inputs . . 34

List of Figures

1.1 An everyday-life example of Congestion 2
1.2 QFDB 3

2.1 A switch that directs too many inputs to a single output could potentially suffer from Congestion 6
2.2 Division of a link's capacity among 8 flows. 9
2.3 Round trip of a Flow Rate Packet that passes through two contention points (CPs). The current rate field inside the FRP may change on every switch from source to destination. 10
2.4 Example timeline of a single Contention Point (CP). In every RRP, the CP receives one FRP message from each active transfer, indicating the sending rate of each flow. 11
2.5 Network topology and flows used in Experiment 1. 14
2.6 Experiment 1: Convergence to max-min-fair allocation. 14
2.7 Network topology and flows used in Experiments 2 and 3. 16
2.8 Experiment 2: Flow completion times of latency-sensitive (small) flow 2. 17
2.9 Experiment 2: Flow completion times of all flows. 17
2.10 Simulation results for Experiment 3, with local and remote flows competing for a link. 18

4.1 An abstract idea of the congestion control's effect on the network . 21
4.2 An abstract idea of the mechanism's components 22
4.3 An example setup of the Congestion Control Scheme 23
4.4 Abstract Functional overview of the Contention Point 26
4.5 Detailed Functional overview of the Contention Point 27
4.6 Outline of the register transfer level (RTL) implementation of the CP. 29
4.7 Full register transfer level (RTL) implementation of the CP. 31

Chapter 1

Introduction

The industry of computing has converged towards building large Data-Centres (DCs) to achieve High Performance Computing (HPC) facilities. High Performance Computing generally refers to the practice of aggregating computing power in a way that delivers much higher performance than one could get out of a typical desktop computer or workstation in order to solve large problems in science, engineering, or business. Warehouse-scale data-centres and large HPC systems rely on efficient networks to interconnect a constantly increasing number of memories and end-nodes, which hold powerful and costly processors. Ongoing research proposes to replace the expensive, high-end server processors of today with simpler microservers in order to reduce the energy consumption and the cost of the system as a whole. In such microserver farms, the network resource is even more critical, because a larger number of distributed node peers need to communicate in order to achieve a given task [1]. But such network demand can easily cause a phenomenon known as congestion.

Network Congestion and Congestion Control

Network Congestion occurs when demand exceeds capacity [2], and in HPC systems that rely on critical timing it can prove devastating. To very briefly demonstrate it with a picture of modern life see Figure 1.1, and this phenomenon is usually avoided by *overprovisioning*, or to put it bluntly, 'throwing' money at the problem, in this example building wider roads. This solution compromises efficiency and therefore leads to another research topic that we are going to elaborate on next. Congestion control is about using the network as efficiently as possible. In order to minimize network cost and space, a mechanism is utilized to enforce a policy that would allow traffic to flow uninterrupted without creating congestion. For maximum efficiency, a system implementing this mechanism should be developed in a bottom-up fashion starting from the hardware that contains it, but also making the higher levels aware of the mechanism. At the same time, this new system gives us the opportunity to redesign from scratch the network I/O stack and to optimize it for low-overhead, high-speed communication.

Figure 1.1: An everyday-life example of Congestion

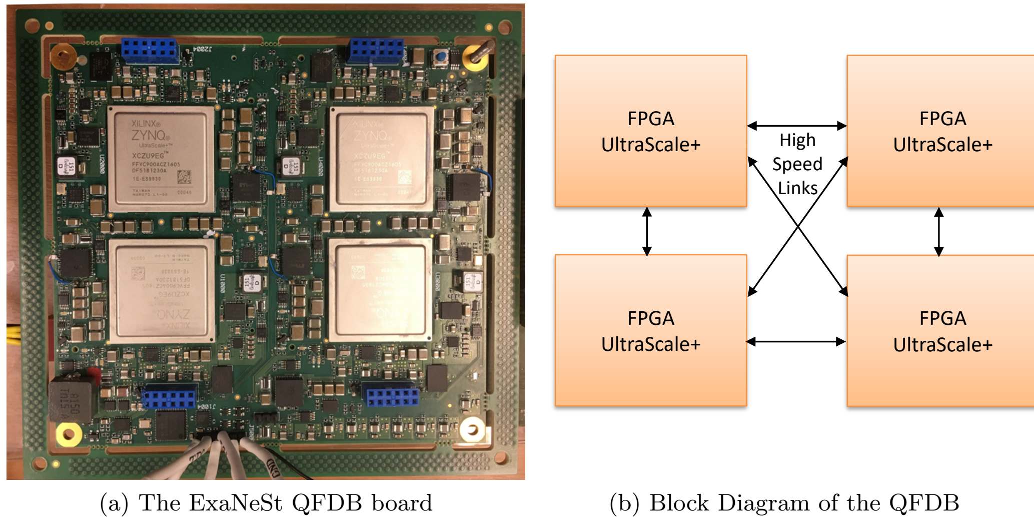
1.1 RDMA transfers to replace TCP stack

In systems relying on shared interconnects, it is crucial that congested flows from one workload do not badly interfere with flows from other workloads. To mitigate the detrimental effects of congestion today, most data-centres rely on TCP congestion control. However, the TCP/IP protocol perceives congestion from packet drops that cause sources to reduce their transmission rate and packet dropping is costly [3]. When congestion has progressed as far as causing packets to be dropped, it is already too late to act, which would hint that this method is sluggish and suboptimal, and so the industry is looking for alternatives [4]. In this thesis, in order to properly attack congestion, we avoid the TCP/IP stack altogether and instead use RDMA transfers reducing latency at the same time. We develop a new congestion control mechanism to fairly allocate rates to multiple flows. Our mechanism:

- throttles the offensive flows
- keeps the backlogs outside of the network
- allocates max-min-fair rates to flows

1.2 Prototyping platform

We are currently working for the ExaNeSt project [5] which develops solutions for Exascale HPC, on a system that consists of multiple Ultrascale+ FPGAs interconnected with high speed serial links with FPGA implemented switching (See



(a) The ExaNeSt QFDB board (b) Block Diagram of the QFDB

Figure 1.2: QFDB

Figure 1.2). Each FPGA has 4 ARM®Cortex-A53 cores and 2 Cortex-R5 real-time processors. So far the prototyping platform is a miniature version of the big ware-house scale system suggested before. The final system will consist of millions of computing cores that communicate together in clusters to achieve high performance. The task of this thesis is to implement and study the congestion control mechanism as given by the patented scheme [6], that will be described in Chapter 2. The mechanism should ensure that all congestion is avoided in the most efficient way, including the size of itself, as in such a big system, multiple instances of hardware will need to be placed, and therefore it should be of minimal area, timing and power consumption.

1.3 Contributions

This thesis revolves around the context of Max-Min-Fairness (MMF), and particularly the congestion control scheme given by Manolis Katevenis [6]. This thesis covers the implementation of one primary and one secondary hardware modules for the aforementioned MMF scheme. Implementation means:

- Schematic drawings
- Synthesizable Verilog code description of the drawings
- Testbench code that tested the Verilog code
- Simulation results and waveforms of the above testbench code
- FPGA configuration binaries generated by the Xilinx Vivado tools with the Verilog code input

- Run logs and output result files of the Vivado tools that show an area utilisation as low as 231 LUTs and 171 Flip-Flops, and a maximum achievable clock frequency of 300MHz in the UltraScale+ FPGA that we used

Specifically the contributions of this author in this thesis are:

1. Understanding [6] and translating it into an abstract hardware architecture including the rest of the parts of the scheme that were not implemented in this thesis

2. Implementation of per-outgoing-link monitor logic which must exist in all the network nodes (See Section 2.5 for a high level description and Section 4.2 for the implementation)

3. Implementation of the synchronisation period module (See Sections 2.6 and 4.6)

4. Corner case resolution and inclusion in the implemented logic (See Sections 4.3, 4.4, and 4.6); The corner cases and a brief explanation of their resolution are:

    - Division by zero, was resolved by incrementing the 0 to 1, while modifying other counters to compensate
    - Extreme division timing and area cost, was resolved by using a ready, optimized division engine
    - Abrupt traffic patterns that the scheme is not focusing on, were resolved by temporarily applying some backup static rules
    - Communication protocol restrictions, were not resolved but had to be worked around

1.4 Outline

The outline of this thesis is the following: After Chapter 1 which contains the Introduction, we move to Chapter 2, that contains the theoretical aspect of the work, explains terms and definitions, introduces and describes the new congestion control method and further justifies the necessity of the work, as well as its solutions. Chapter 3 explains the tools that were used, the methodology and all the technology that this work involved. Moving on to Chapter 4, we explain the simulations that were carried out; The simulations were an important input for this work's process. The main part of the work is described in Chapter 5, which includes the implementation of the design in hardware, and the rationale behind specific choices. The work concludes in Chapter 6 with the general findings of the work as well as the direction for future work.

Chapter 2

Max-min-fair Rate Regulation Without keeping Per-flow State (MaRRWiPS)

In this chapter, we will explain the congestion control scheme that is under development, that includes the link monitor that we implement in this thesis and we get into all the theoretical details that are involved. Before getting to the details of the policy we should first define all terms involved, some of which might seem trivial but we should make sure that the reader is fully aware of what they are reading, and any ambiguity is avoided. After clearing the terms and a brief clarification of the issue we are working on, we will describe the scheme in a top-down fashion, starting from the general features and later explaining each feature on its own. Lastly, this chapter gives descriptive examples for better understanding of the scheme and examines optimization potential.

2.1 Terms and definitions

Interconnection networks have converged to using *Flow-oriented* network traffic for communication. A *Flow* is a sequence of packets sent from a same source to a same destination that concern the same application.

Rate of transmission is how many bytes of a packet a user, or a flow, can pass through a link or port per second. The rate of a fully utilized link or port is referred to as its *Capacity*.

With the term *Contention* we refer to the phenomenon of a single resource e.g. a network link or a switch port, that is requested by two or more users of the resource. For example, two or more flows incoming from different sources in a switch going to the same destination would create contention on that port and link, or would content for it. The switch is designed exactly for resolving such contentions, so it can resolve it by storing one or more packets that would not be sent immediately in a buffer, or queue, sending one, and with a deterministic

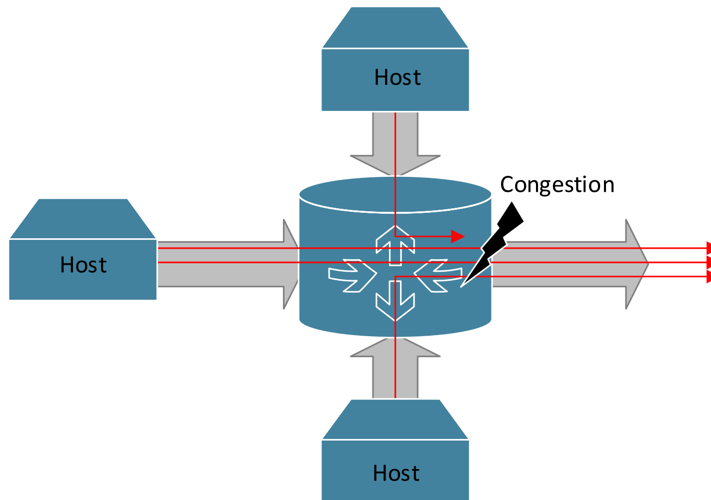


Figure 2.1: A switch that directs too many inputs to a single output could potentially suffer from Congestion

behaviour it would continue to send all packets of contenting flows one after the other until all are sent. If contention is not handled correctly and the resources are not abundant, contention can lead to Congestion. Figure 2.1 depicts the idea that was mentioned.

Congestion has been briefly defined in Chapter 1, but it is worth to differentiate it from *Contention*. The main difference is that *Congestion* is the phenomenon of output contention for periods of time long enough that the buffers of the switch would fill up, usually having to drop incoming packets and therefore harming the performance of the network. The main cause of this is that sources inject traffic in the network unaware, or ignoring other sources, that in turn act similarly. For that reason, if many flows from different sources have to pass through one link or port, and these sources inject traffic recklessly, it is easy to surpass the capacity of the resource, and if this persists through time, eventually the buffers will fill up. Switches are built to solve contention and despite the overprovisioning, still they cannot avoid congestion altogether. To properly mitigate congestion the network needs to utilize a *Congestion Control Mechanism*.

2.2 The Congestion Control issue

Many Congestion Control Schemes have been proposed in the past [7, 8, 9, 10, 11, 12, 13, 14] but the research has not stopped. This is caused by the network policies that apply and also to some requirements that congestion control schemes must satisfy [15]. Such requirements are:

Low overhead, if the overhead is high then the scheme would most likely create congestion rather than nullifying it.

Fairness, the fundamental of contest resolving is for all parties to receive 'fair' service. Fairness is a very ambiguous subject in resource sharing, some researchers claim that if everyone gets a non-zero share of the resource the scheme is fair. Others insist that everyone must have equal portion of the resource to achieve fairness, while others would suggest that different background should affect the equality (e.g. packets that have travelled the longest, or smaller packets should have priority). Nonetheless, the matter remains unresolved.

Responsiveness, the scheme must apply fairness quickly, otherwise congestion will build up and become visible by end users. Furthermore, if it is too slow then by the time it applies fairness, congestion might have disappeared, or relocated elsewhere, rendering the whole scheme pointless.

2.3 MaRRWiPS high level description

This scheme resolves congestion in a very responsive way, experimentally within 20 - 40 μs, avoids starvation by allocating Fair Share Rate (FSR) amounts to all flows, and utilizes links to their capacity by applying Max-Min Fairness.

Before getting into the details, we will briefly overview the features of the scheme. To sum everything up, all sources send state messages, referred to as Flow Rate Packets (FRPs), to their destinations periodically. These messages carry a Current Rate (CR) and a Desired Rate (DR) field which is initially set as the rate of injection. All link monitors that these pass through may or may not decrease the CR and DR fields according to the FSR of the link that is calculated periodically and reflects the traffic going through it. Upon arriving to the destination, the packet is sent back to the source containing the bottleneck information and the source changes its injection rate to that specified, sets the CR to that amount and sets the DR to the rate it desires to achieve.

We will now clarify all the features mentioned in detail before giving examples.

2.4 Max-min Fairness

If the aggregate rate of all flows on a link is less or equal to its capacity, the traffic is said to be *feasible.*

By definition [16, 17, 18] Max-min fairness (MMF) is a feasible allocation of rates that satisfy the following condition: An increase of any rate of the flows involved comes at the cost of a decrease of some already smaller rate. To put it more simply, MMF equally distributes the available capacity of a link to all flows that pass through it, but if a flow cannot use all of the capacity allocated to it then the remainder is distributed equally to all able flows.

That being said, the essence of MMF is that it can potentially utilize the capacity of a link, if for example a flow is bottlenecked on a specific link, then on all the other links that it passes through, there is no reason for it to receive any higher rate, as it will be unable to utilize it, therefore all those other links

can reduce the rate given to it to the amount it can receive at its bottleneck, and utilize the remainder capacity with other flows.

To ensure MMF is applied correctly, the granularity of network multi-path routing must be adjusted from per-packet to per-flow. All packets from the same flow must absolutely pass from exactly the same route, otherwise applying MMF will be pointless. Multi-path can still be used, but not for packets of the same flow.

To apply MMF, we need information about each link and each flow, but our purpose in the scheme is to not store any flow oriented information. How this is done is explained further below. First we must explain how to determine a flow's rate judging by the route it follows. This is exactly the work of the link monitor that we are implementing. In that route, we determine which link allocates the least capacity on this flow, by receiving feedback from the link monitors, and establish this as its bottlenecked rate. From then on, all other link monitors the flow passes through will change the rate allocated to it to the bottlenecked rate. Then those monitors can redistribute the remainder of the capacity to all other flows.

Determining the bottleneck rate is simple, inject the flow in the network with a field containing the rate at which it was injected, let us call this Current Rate (CR), and each node the flow passes through will check if the outgoing link it will pass through can sustain CR amount of rate for this flow, if it can, it will pass the packet onwards unmodified, while if the rate that the flow can sustain through this node is less than CR, the CR field will be replaced by an amount CR', which equals the rate this node can sustain for this flow, by the link monitor logic, and passed onwards. In the end, the CR field will contain the minimum of the rates sustainable in the path.

This way of finding the bottleneck also resolves the question on how to have information about a flow without storing it- the flow will carry it. More accurately, after getting to the destination node, the flow will report the CR field to the source and the source will inject traffic of this flow at this rate from then on. We assume there are no malicious users in the network, and we will not address fault tolerance in this work.

2.5 Fair Share Rate (FSR) Calculation

We have not yet elaborated on how each node will distribute the capacity of each link fairly. For simplicity let's initially assume there is n constant flows passing through one link. Assuming all links want to use the link to its capacity C, it would be considered fair to distribute a rate of $\frac{C}{n}$ to each one, so let us go ahead and call this the Fair Share Rate (FSR) of the link. In fact, this would also be the case if all flows did not want to use the full capability of the link, but still needed a rate greater than FSR.

Let us now explore the case where one or more flows need less rate than FSR. In

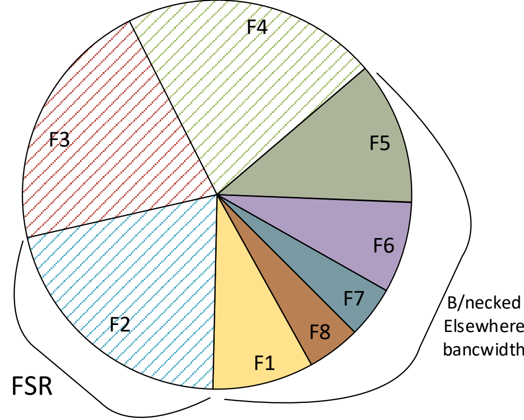


Figure 2.2: Division of a link's capacity among 8 flows.

that case, as we have mentioned the remainder of the capacity must be distributed in an even manner to the rest of the flows. Such flows, either truly do not need more rate or are *bottlenecked elsewhere*, to us they are not different so we refer to them as *bottlenecked elsewhere* flows, while we refer to flows that need more or equal rate to FSR as *bottlenecked here*. We aggregate the rate of the *bottlenecked elsewhere* flows (B) and we subtract this amount from the capacity. The remainder is the capacity that *bottlenecked here* flows have to share, so each one would receive $\frac{C-B}{M}$ units, where M is the number of *bottlenecked here* flows. Figure 2.2 displays a pie chart of a link's capacity distributed among 8 flows, we can see that small flows F1, F5, F6, F7, F8 are considered *bottlenecked elsewhere* and receive all the rate they request, while bigger flows F2, F3, F4 fairly share the rest of the capacity even though they probably could use more.

In our algorithm, a portion of the link capacity, $C \cdot \alpha$, is reserved and not allocated to flows, creating some headroom that can accommodate for control overhead. Increasing parameter α will on one hand allow the network queues to drain faster the transient backlogs that can be formed during congestive episodes. On the other hand, an extremely high value for α will result in flow backlogs staying at their sources, outside the network, while network links are idling. The FSR is finally calculated by the following formula:

$$FSR = \frac{C \cdot (1 - \alpha) - B}{M}$$

2.6 Realistic Fair Share Rate Allocation

Previously we discussed scenarios of static flows, and such scenarios are not realistic. Flows have no deterministic beginning or end. They can start without any warning and end at any point in time. Therefore keeping track of the flows that are bottlenecked on each node, or on other nodes is difficult.

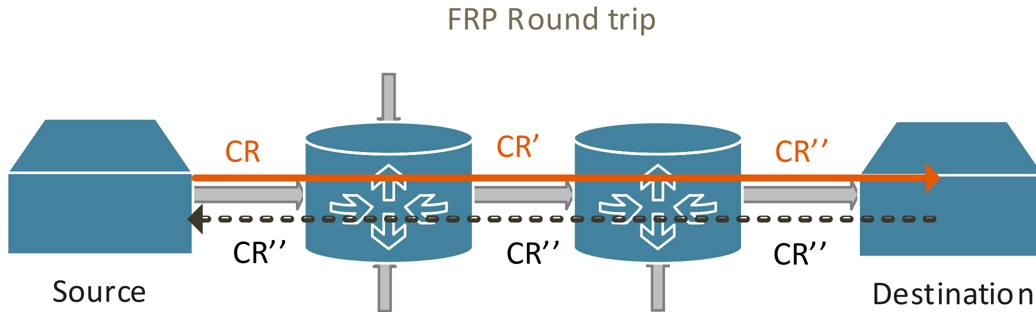


Figure 2.3: Round trip of a Flow Rate Packet that passes through two contention points (CPs). The current rate field inside the FRP may change on every switch from source to destination.

In order to resolve this we use a periodic sampling technique to determine, in every period, how many flows are currently active and update the FSR for this period. All flows must be aware of this, as they need to announce activity in every such period. But because we are not keeping flow information, each flow must only announce itself once per period, with an announcement that differs from normal packets, so that flows can send more than one packet per period. We refer to packets carrying announcement information as Flow Rate Packets (FRPs). We refer to those periods as Rate Re-evaluation Periods (RRPs), and are usually in the magnitude of 20 μs.

At the end of every RRP, in every outgoing link, information on how many flows received, how many of them were bottlenecked on that link, and how many were bottlenecked elsewhere is gathered and a new FSR is calculated, then all other counters (such as M and B) are reset. On the next period, incoming flows have to compare themselves with this new calculated FSR to determine if they are *bottlenecked here* or elsewhere, and this will increment the respective counter.

In the end of this RRP, the same procedure will repeat, using the information of this period to recalculate the FSR. Note that if no flows stopped or started during this time, the FSR should remain the same, as all flows must have sent FRPs.

Intuitively, the algorithm computes the bandwidth that the M locally bottlenecked flows would receive by a fair scheduler, considering also that the present link cannot (and should not) modify the rates of flows *bottlenecked elsewhere*.

2.6.1 Descriptive examples

Figure 2.3 displays a round trip of an FRP in an example topology. The source sets the CR field of the FRP message equal to the current maximum sending rate

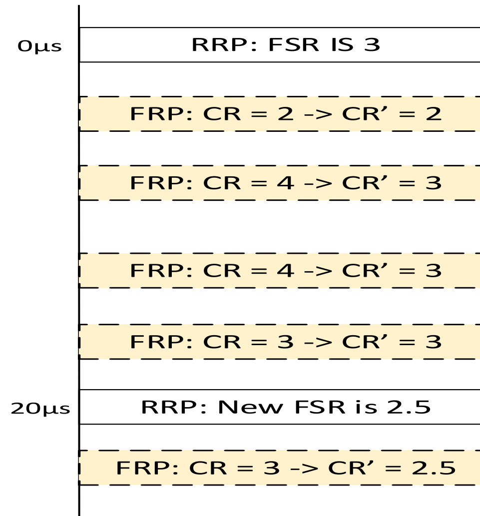


Figure 2.4: Example timeline of a single Contention Point (CP). In every RRP, the CP receives one FRP message from each active transfer, indicating the sending rate of each flow.

of the flow. After passing through the first Contention Point (CP), in the first n on the way, the CR field might be adjusted to the link's needs and so the FRP now carries CR'. Similarly the next CP in the next router adjusts the CR' to CR" and passes it to the destination. On the destination, the FRP is injected back in the network, in the opposite direction, but this time with a flag set so that routers do not modify the CR field. Effectively, when it reaches the source, the FRP carries the maximum rate that the network can sustain from this flow.

Figure 2.4 presents the timeline of a CP with a RRP of $20\mu s$ and capacity 10 units, it shows how the RRP triggers the FSR calculation, which in this example is calculated to be 3 units according to previous events. After the end of the RRP, the CP receives several FRPs and makes sure all of them get their new CR (CR') modified accordingly to not surpass the FSR. After the end of the RRP, the CP recalculates the FSR according to the formula, having 2 flows bottlenecked on it and 5 throughput units *bottlenecked elsewhere*. The FSR is (10 - 5)/2 = 2.5, ignoring parameter α for the sake of simplicity.

2.6.2 Desired Rate

We explained how we gather information about a bottlenecked flow using the Current Rate (CR) field. What we have not mentioned yet is that after the destination replies with the bottleneck rate and the source changes the rate that it injects, it also changes the CR that will be included in the FRP to the new rate. This creates a problem, since the rate can never increase- once congestion has been resolved the rate of the flow can not increase.

CR	DR
·Unmodified at source before retransmit ·At source, reduces rate only	·Reset to original before retransmit ·At source, potentially restores rate
·Changes value at CP, if value is bigger than FSR	

Table 2.1: CR/DR disambiguation

To correct that, the FRPs also include a field called *Desired Rate* (*DR*). DR works exactly the same way as CR, while the FRP is in the network, but the difference is that the source will always set the DR field to the rate it desires, irrelevant of the bottleneck rate it received. CPs will first look at the DR to see if they can satisfy it, and then look at the CR, they will modify both CR and DR, if needed. CR and DR differences are very confusing, so Table 2.1 points out the main differences.

2.7 Optimizations

2.7.1 Short-circuit notification

Assuming a flow starts abruptly with a very large injection rate, it will take a full Round Trip Time (RTT) until its rate gets regulated. To mitigate this, an optimization technique can be applied: An instant response is generated from the link monitor that observed excessive rate request and forwarded directly to the source. With this, the instability will last much less than waiting a full RRP. This however comes at a cost, as common occurrence will flood the network with responses. To keep the situation under control, the packet will only be generated and passed back in extreme circumstances, when the flow requests a much bigger rate than the link can allocate.

2.7.2 Flow Initialization messages

When a new flow commences, the CPs inside the network are still unaware of its presence. What will happen in this case is that within the next RRP period the flow will issue an FRP. Subsequently, the affected CPs will update their FSR at the end of this FRP period, and will start throttling flows taking into account the presence of the new flow in the next RRP. This latency can induce large backlogs inside the network.

To accelerate this procedure, flows can inject a *flow-init* FRP together or in advance of the first network packet. Flow-init FRPs trigger a prompt update of the link's fair share, and may also cause short-circuit notifications.

2.7.3 Flow Termination messages

If a flow wants to stop transmitting, it should send a *flow-stop* FRP message. With this message, it lets the CPs know that its bandwidth is not being used any more. This allows CPs to distribute their rates right on the next RRP, so that flow rates can reach an optimal state, and utilization can reach its maximum.

Note that this message is the only exception to the *only one FRP per RRP* rule, and can only be sent after the normal FRP was sent. Unlike other FRP messages, this will be handled in a very different manner, as its purpose is to reduce counters- if the counters did not contain the values to be subtracted, the result could be disastrous.

2.8 Simulation

In this section we present some computer simulations [19] that can assist the understanding of the proposed scheme. In these experiments, the scheme is referred to as Distributed Max-Min Fairness (DMMF). To evaluate DMMF, we use event-driven simulations based on Omnet++ [20] in 10 Gb/s networks.

We use the implementation of TCP that comes together with the INET library in Omnet. The TCP is configured with maximal receiver buffer capacity of 21KB, no delayed ACKs, maximum segment size of 1500Bytes and the Congestion Control algorithm is TCPReno. We test lossless TCP, which is essentially TCP in a network with flow control that prevents packet drops, and two types of lossy TCP, one performing drop-tail at switch inputs and one at switch output. The network zero-load round-trip time in our simulations is equal to two microseconds.

With DMMF, the transfers are initiated by user applications which fire up the DMA engine. The DMA engine generates network packets with maximum size of 256B. Each flow communicates messages from its source to the destination host using RDMA transfer. DMMF is configured with flow-initialization FRP messages. In DMMF, we set the parameter α to be equal to 0.05, so the available useful bandwidth is 9.5 Gb/s.

2.8.1 RRP setting

The RRP must not be too long in order for the system to react promptly to traffic changes. On the other hand, if we set it too small, then the number of FRPs will flood the network. FRP messages may need to travel with high priority in order to not get stuck behind other traffic. In our experiments, the length of standalone FPRs is 20 bytes, and the RRP is set at 20 microseconds. Thus, the overhead of a single active flow is 20 bytes every 20 microseconds, i.e., 8 Mb/s. This overhead is minor when a few flows congest a single link of 10 Gb/s, as is expected at the periphery of the network inside the rack. The capacity of core (inter-rack) links, which may carry many more flows, will be considerably higher: one thousand flows will only waste 8 Gb/s, i.e., less than 10% of the aggregate capacity of 100 Gb/s

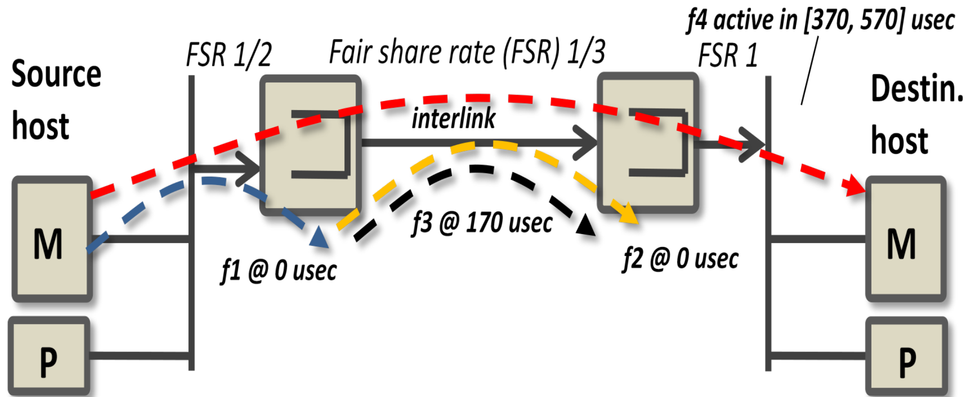
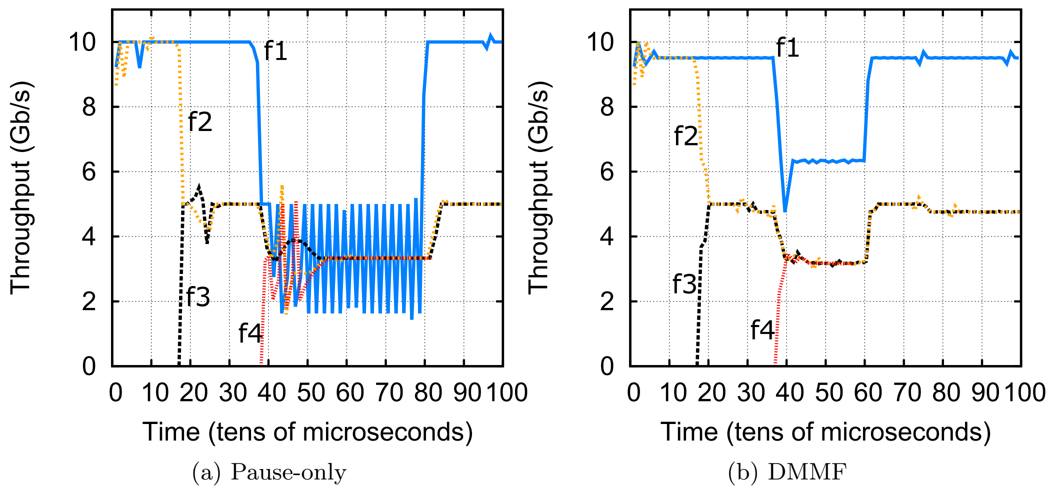


Figure 2.5: Network topology and flows used in Experiment 1.



(a) Pause-only (b) DMMF

Figure 2.6: Experiment 1: Convergence to max-min-fair allocation.

link. To further reduce the overhead, one may try to piggyback FPRs over payload packets or to increase the RRP.

2.8.2 Description of experiments

We report the results from three experiments, which stress the areas that we want to focus on.

- Experiment 1: *Convergence to max-min-fair rate allocation*
- Experiment 2: *Flow completion times of latency-sensitive flows*

- Experiment 3: *Fairness in scenarios with local and remote flows*

2.8.3 Convergence to max-min-fair rate allocation

In our first experiment, we configure four flows in the custom network topology depicted in Figure 2.5. All links run at 10 Gb/s. Each flow communicates messages from its source host to its destination host, using RDMA transfers. Only one source host and one destination host are shown. The other nodes shown in the figure are switches; one input and one output port is shown at each switch. In reality, the switches have additional input and output ports, which we omit for clarity.

Flows f1 and f4 are sourced at the left host and diverge inside the network. Flows f2, f3 and f4 pass through a common link, which will become their bottleneck during the experiment. The flows start sending at different times, as described below.

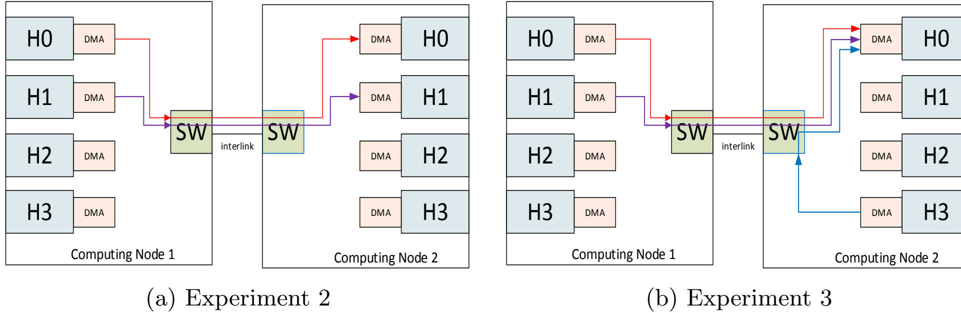
- Flows f1 and f2 last throughout the simulation experiment
- Flow f3 is activated at time 170 μs, and stays on till the end of the experiment
- Flow f4 is triggered at time 370 μs and goes down at time 570 μs

Before 170 μs, only flows f1 and f2 are active; therefore, no link is congested, and the flows can reach full link rate. At the same time, no backlog is formed inside the network. Problems emerge together with flow f3, which together with flow f2 congest one network link (the one having the label "interlink" in the figure).

In our tests, we examine the DMMF algorithm as well as Pause-only (link-level) flow control with no congestion management. As shown in Figure 2.6(a), with Pause-only flow control, once flow 3 becomes active, f2 and f3 get 5 Gb/s because they share a link. Subsequently, a backlog is formed in front of this link (interlink backlog curve). Because the network is flow controlled, backlogs are also formed upstream, inside compute nodes 1 and 2, which source flows f2 and f3 respectively. However, the backlog does not reach compute node 0, which sources flow f1, because the path of f1 is disjoint from that of both f2 and f3. Effectively, the rate of f1 is unaffected.

The interplay among flows becomes more complicated once flow f4 is activated at time 370 μs. Since f2, f3 and f4 share a link, they all get a rate of 3.33 Gb/s. However, due to Pause-only flow control, the backlog of f4 reaches compute node 0, wherein it strikes flow f1. Flow f1 shares a link with flow f4, therefore, the local fair share of each is 5 Gb/s. Since f4 is constrained at 3.33 Gb/s, ideally f1 should get 10-3.33 = 6.66 Gb/s.

Nevertheless, as seen in Figure 2.6(a), with Pause-only flow control, f1 gets as little as 3.33 Gb/s. This happens due to head-of-line blocking inside the queue that f1 shares with f4: because of downstream flow control, this queue drains f4's packets at a rate of 3.33 Gb/s, effectively limiting the departure rate of f1 packets as well.



(a) Experiment 2 (b) Experiment 3

Figure 2.7: Network topology and flows used in Experiments 2 and 3.

Next, we examine how the proposed congestion control scheme can correct this behaviour. The results are shown in Figure 2.6(b). When the congestion begins, flow f1 is first throttled to rate 0.5, because it shares a link with f4. However, at the next RRP, f4 informs the corresponding CP that its CR is 0.33. Effectively, f1 can use the excess bandwidth and get the ideal, max-min-fair rate of 6.66 Gb/s. In this example, the protocol converges to the correct max-min-fair rates in two RRP periods, i.e. 40 usec.

2.8.4 Flow completion times of latency-sensitive flows.

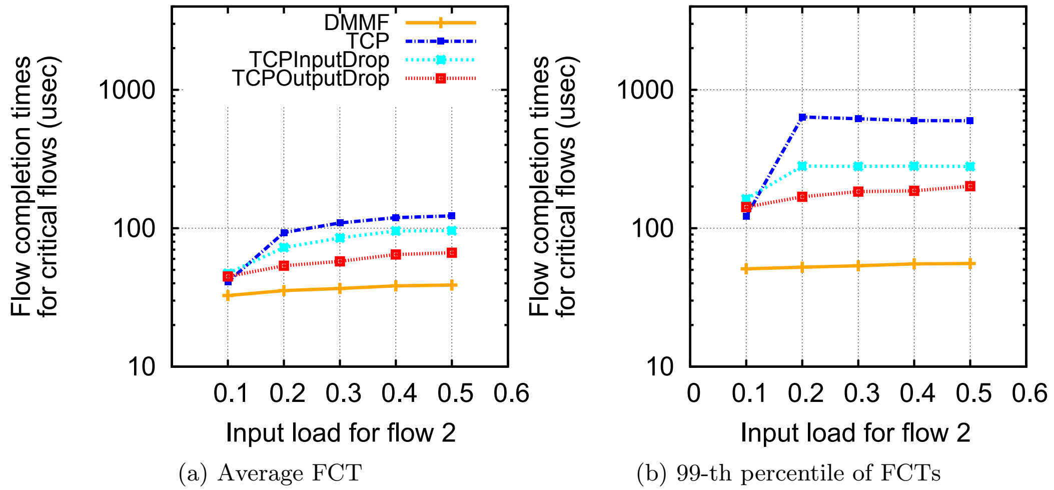
Figure 2.7(a) depicts the network topology and the traffic flows that we simulated in the second experiment. The network consists of two Computing Nodes. Each computing node has four hosts and one computing node switch. The two computing node switches are connected with a single link, labelled interlink in the figure. The interlink is the link to be congested by many flows which send packets form computing node 1 to computing node 2.

For this experiment, flow 1 sends large messages of 1 MBytes at 5 Gbits/s, whereas Flow 2 sends smaller (3 and 30 KBytes) messages, at a varying rate.

Figure 2.8 and Figure 2.9 depict the Flow Completion Times (FCTs) as a function of the sending rate of flow 2, which is displayed on the x-axis. Note that given that flow 1 sends at 5 Gb/s, the network saturates when the input load of flow 2 is 0.5.

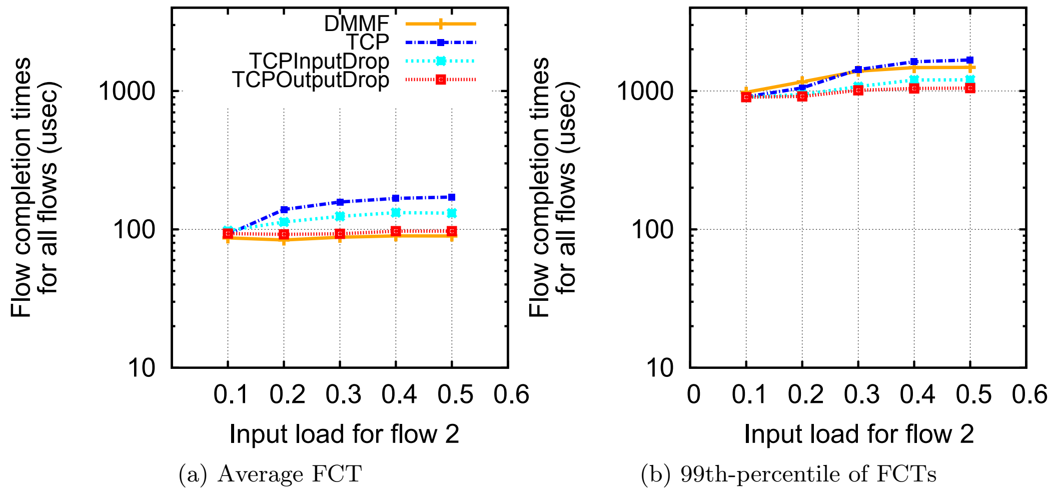
In Figure 2.8, the y axis shows the flow completion time (FCT) of Flow 2, whereas in Figure 2.9, it shows the FCT for all flows. From Figure 2.8, comparing the FCT of flow 2, we observe that DMMF outperforms TCP. Especially for the critical (small) flows, the proposed scheme achieves almost 2x smaller latency than TCP.

As the input load increases, the FCT of DMMF increases at a significantly lower rate than with TCP implementations. This proves that DMMF is more stable and reliable than TCP in case where the input load exceeds a certain limit.



(a) Average FCT (b) 99-th percentile of FCTs

Figure 2.8: Experiment 2: Flow completion times of latency-sensitive (small) flow 2.



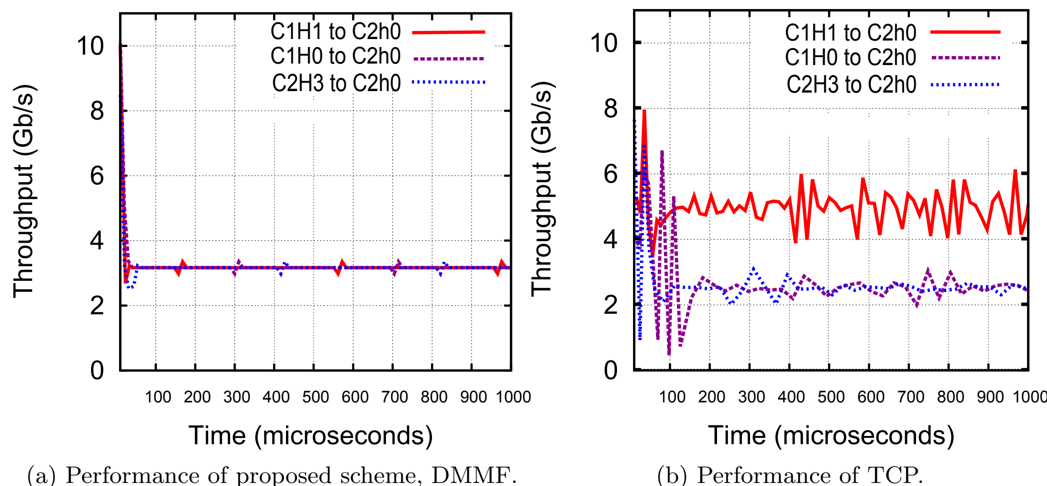
(a) Average FCT (b) 99th-percentile of FCTs

Figure 2.9: Experiment 2: Flow completion times of all flows.

In Figure 2.9, comparing the FCT of all flows, we observe that DMMF, albeit decreasing the latency of small flows, does not degrade significantly the performance of small flows. Overall, these results show that with the proposed scheme, the FCT of small (latency-sensitive) flows is significantly better than that of the TCP.

2.8.5 Fairness in scenarios with local and remote flows

In the third experiment (see Figure 2.7(b)), two hosts in Computing Node 1 (H0 and H1) send packets to a host in Computing Node 2 (H0) that is also receiving



(a) Performance of proposed scheme, DMMF.

(b) Performance of TCP.

Figure 2.10: Simulation results for Experiment 3, with local and remote flows competing for a link.

packets from another host in Computing Node 2 (H3). All flows send packets at full link speed.

For DMMF congestion control, the CP in switch 1 in front of the interlink tries to allocate a fair share of the rate on the interlink to all flows; thus, every flow receives control messages at its source in order to adjust its sending rate. The CP in switch of Computing Node 2 acts similarly for the flows towards H0.

The results are presented in Figure 2.10. The max throughput for DMMF is 9.5 Gb/s since 0.5 Gb/s is reserved for control messages. What is obvious from these results is that DMMF allocates a fair share of the link's rate to all flows without distinguishing between local and remote ones. In contrast, lossless TCP allocates two times more bandwidth to the local flows than to remote ones.

Chapter 3

Tools, Technologies, and Methodology

In this chapter we will go over the tools as well as the technology and methodology used in the development of this thesis. We mention most of these resources in their respective section but e decided that we should summarize everything used in one section. We cover all hardware, software resources, as well as all tools involved in simulation and implementation.

3.1 Hardware Resources

We will begin by describing what physical machine resources we used. Of course we used workbench computers for the whole process. Other hardware includes ZedBoard [21] and Zynq UltraScale+ [22] FPGA boards for implementation testing. ZedBoards we chosen as the initial platform due to their abundance in the laboratory, and UltrScale+ were used for final testing due to them being the main components of the prototyping platform mentioned in Section1.2.

3.2 Software Resources

Software used includes the Xilinx Vivado design suite [23] for developing the HDL modules, the same suite was used for synthesizing the designs for FPGA testing. Implementation, placement and simulation tools of the same suit were used. The reasons for using the Xilinx tools is mainly the expertise that we have with them, and the laboratory already owns licensing for use.

Microsoft tools used are the Microsoft Visio [24] software, which was used for creating the figures and Microsoft Word [25] that was used for intermediate text and note management,

Notepad++ [26] was used for writing the final text, and MikTex engine [27] was used for LaTeXcompilation for the production of the final text.

3.3 Simulation tools

The simulations that were given used an OMNeT++ [20] environment. This environment was used because it is very modular, as you can freely create and manipulate any topology needed and due to previous expertise of the operators. The simulation scenarios and topology were written in C++. Figures that will be displayedin the respective section were drawn using the gnuplot [28] tool as it is the most common practice in academic works and can be handled with great precision.

3.4 Implementation tools

As already mentioned Xilinx Vivado suite was mostly used for the implementation. The HDL language used is Verilog, due to expertise of the author as well as it being the language of choice in current proects of the laboratory.

The design patterns and techniques follow the Xilinx Synthesis and Simulation Design Guide [29] in order to ensure that translation from code to hardware is precise.

The module creation process followed the Vivado Design Suite User Guide [30].

Chapter 4

Implementation

In this Chapter we will cover the implementation process from the very beginning until the end of this thesis. We will cover all details including design choices and why they were made, as well as information that could help future readers of this thesis understand or even implement the scheme on their own, given the code material. The chapter is written in a top-down fashion starting with a very abstract level and reaching low levels with specific details.

4.1 High Level Design

Before implementing anything we make designs. We need to plot out what we need, following levels of abstraction. In the most abstract level, we are looking into a black box network that achieves maximal utilization.

Our aim is that assuming uniform traffic, at the beginning the aggregate rate of the output of the network will be either equal to the aggregate capacity of the output links, or equal to the aggregate input rates. This means that either the incoming rate is feasible for the output and can be passed over, or that the outputs are used maximally. This has to change quickly though as the network traffic should converge to all sources injecting feasible traffic and soon the aggregate rate of the inputs should equal the capacity and the aggregate rate of the outputs. Since in practice, the traffic will not be uniform, our goal is tend to what is said above as optimally as possible.

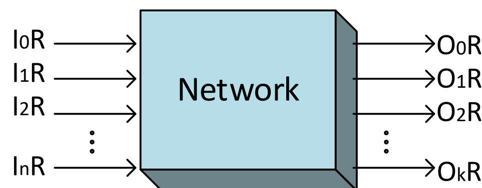


Figure 4.1: An abstract idea of the congestion control's effect on the network

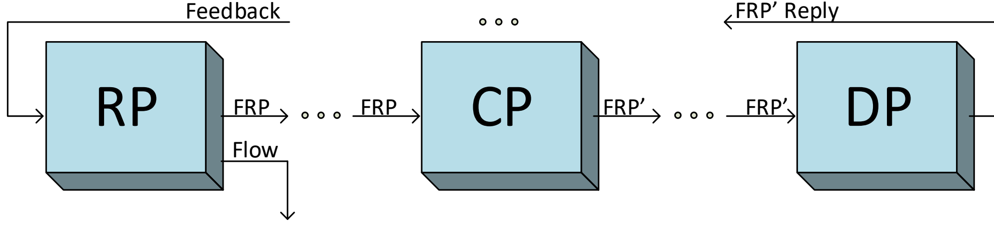


Figure 4.2: An abstract idea of the mechanism's components

In Figure 4.1, I_0R represents Rate of Input 0, O_0R represents Rate of Output 0 and so on. Assuming that the Aggregate rate of outgoing links is $A_o = \sum_{i=0}^{k} O_iR$ and C is the aggregate capacity of the links, our aim is during instability to be able to achieve at all times:

$$A_o = min(\sum_{i=0}^{n} I_iR \ , \ C)$$

And after convergence we want to optimally achieve:

$$A_o = \sum_{i=0}^{n} I_iR = C$$

To start implementing we move to abstraction level 2, which contains the network nodes, for simplicity we ignore most of the components involved and will only focus on the components that implement our congestion controller. In this abstract level we demonstrate 3 modules as shown in Figure 4.2.

- The *Reaction Point (RP)*, that is the module responsible for applying the rate changes in the source depending on feedback. It is positioned at the ingoing part of the switch, meaning that all flows pass from there and its logic determines if the incoming data is normal traffic to be passed down to the switch fabric. If it is an FRP or an FRP reply that has not arrived back to the source, it is also passed to the switch fabric. Only FRPs that were sent by this node and returned back are treated differently. In that case, they forward the new rate restriction to the DMA engine to be applied directly.

- The *Contention point (CP)*, is the link monitor and is present in all outgoing links in all switches in the network, and is responsible for observing the rate (Current Rate (CR)) of Flow Rate Packets (FRPs) that pass through it and modifying some fields of it if necessary, according to the algorithm we explained in Chapter 2. Any packet that is passing through and is not an FRP is forwarded as is. An FRP that reached its destination in the past and is now heading back to the source, befalls in the same category, and is also forwarded as is.

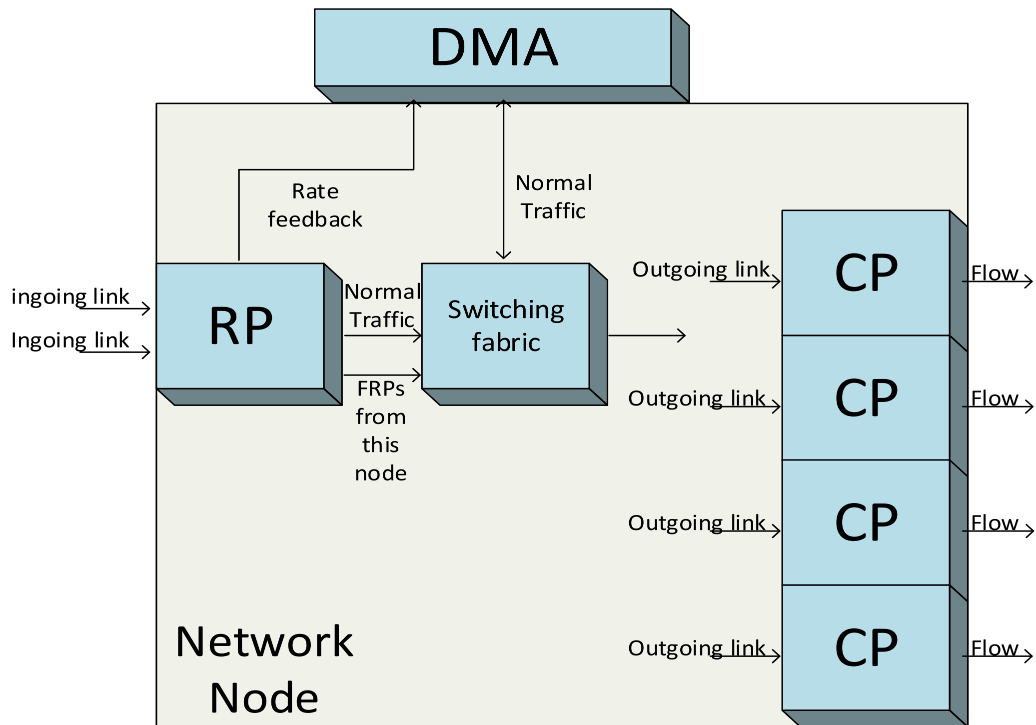


Figure 4.3: An example setup of the Congestion Control Scheme

- The *Destination Point (DP)*, which can exist either in the input or in the output of all links, is responsible for determining whether the destination has been reached, in which case it will swap the source and destination fields, set the forward flag to 0, notifying the network that this FRP is going back to the source and should not be touched, and inject the packet back to the network.

In our implementation we design the CP in a way that it fulfils the duties of the DP as well- we only mention it separately for clarity.

Figure 4.3 displays an example setup of a node in the congestion control scheme. It is a rather small example as we are not aiming for nodes with 2 inputs and 4 outputs, but it only serves the purpose of explanation. In the example, we can see that all inputs are connected directly to the Reaction Point. The reaction point determines if the input is an FRP reply for one of its flows in which case it has to notify the DMA engine of the acquired feedback, or for any other packet it will pass it to the switching fabric. At the same time the Reaction Point must generate and inject an FRP for every active flow whenever an RRP ticks, and does so by handing the FRP packet to the switching fabric. The switching fabric is a conventional switching module, it will determine whether the packet was aiming for this node, in which case it will pass it over to the DMA, or if it was moving elsewhere, in which case it will direct it to the correct output. Contention Points are present

field	#bits	notes
stop	1	FRP stop message
init	1	FRP init message
FRP	1	FRP message
FW	1	Packet has not reached destination yet
CR	8-24	Current Rate field
DR	8-24	Desired Rate field

Table 4.1: FRP layout

in all outputs, and they determine whether the packet must be processed (see Section 4.2) or just passed onwards. Assuming all nodes are built in a similar manner, they can properly communicate to resolve congestion.

At this point, it is useful to define our packet layout for future reference. Normally the layout is defined by the protocol of communication, but in some fields of the packet, either in the header, footer or even the payload, if needed, we can add our own necessary fields. Table 4.1 displays these fields.

- *stop*: The stop flow bit determines whether the packet is an FRP stop packet. In that case, the init and FW flags are ignored (and the FRP flag is assumed to be 1 by default in many implementations that have another type field indicating FRP). When this packet arrives to the destination, it is not injected to return, but is instead dropped.

- *init*: The init flow bit determines whether the packet is an initialization message, in which case it will be handled as explained in Section 2.7.2. Similarly, it is assumed that the FRP flag is 1 in implementations that allow it.

- *FRP*: This is the most important flag, and the least important at the same time. It declares that these fields represent an FRP message. The reason why it might not be important is that in most protocols there will be another 'type' field outside the FRP fields, so that it is known to other mechanisms as well. But because the scheme is designed to be universal we include this information as well. For optimization purposes it can be omitted during implementation.

- *FW*: This flag requires that the FRP flag is set, and determines that the FRP is moving forward, towards the destination. When arriving to the destination this flag will be set to 0 to notify the nodes it passes through not to process it.

- *CR*: The Current Rate field consists of 8 to 24 bits (can also be more or fewer, but we use this range) and displays the rate in which the flow is transmitting or is forced to transmit if there is a bottleneck. The rate is set to what the

process desired at the beginning but is modified according to feedback. We will talk about the representation below.

- *DR*: The Desired Rate field consists of similar bits to the CR, but it displays the rate at which the flow desires to transmit. Unlike the CR, this field is refreshed to the desire of the process always at the source, but it is also reduced when encountering a bottleneck.

4.1.1 Current Rate and Desired Rate Representation

The initial plan was for the fields to be 8 bit long floating point encoded. We understand that normally floating point encoding would require 32 bits, but we took the concept and applied it to fewer bits, as we wanted the FRP fields to be as small as possible, so as to reduce overhead. We can use 4 bits for the exponent and 4 bits for the mantissa. There will be no sign bits as there are only positive numbers and negative exponents as all rates are represented as a fraction of the maximum bandwidth (100Gbit/s or 1Tbit/s). In this way you can represent numbers in a range of 2^{15} ($2^{1111_{(2)}}$) and can have an error of around 3% (The minimum distance between numbers is $1 - \frac{1}{2} - \frac{1}{4} - \frac{1}{8} - \frac{1}{16}$, which is around 6%, but if you also round the number it is as if you use 5 bits for precision therefore the error is around 3%).

That concept can also be applied to more bits, increasing the mantissa will yield better precision, while increasing the exponent will result in an increase of range.

All that being said, in our implementation we use integer numbers of 8 to 10 bits.

There are two reasons for not using floating point encoding. The first reason is because unless the mantissa is significantly big there will always be some error. This might not seem so big a problem but it might lead to underutilization or worse overutilization of the links, which is exactly what we are aiming to avoid.

The second reason is because floating calculations are costly and difficult, and on top of that having some floating point numbers and some integers (e.g. number of flows) enhances the burden. Either the integers have to be converted to floats, which is extra overhead because you need to calculate the exponent and estimate the mantissa, or even worse, you have to implement modules that can perform calculations between integers and floating point numbers.

The reason we insist on the numbers being small is because one of the operations needed is division, which can very easily compromise timing by requiring a lot of processing. The division is a very complicated operation to perform in hardware and in our implementation we have not implemented this operation. We use a divider generated by Xilinx tools as we will mention later, and with bigger sizes it would require several processor cycles for a division to complete.

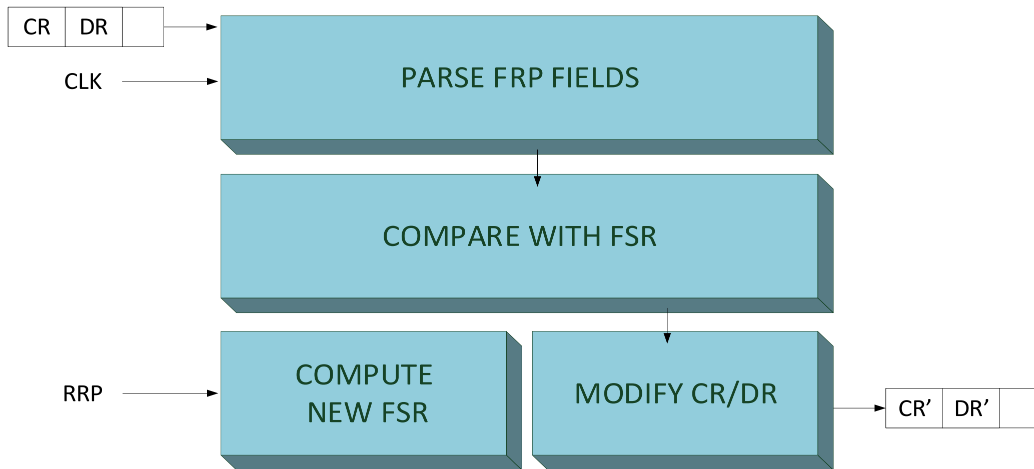


Figure 4.4: Abstract Functional overview of the Contention Point

4.2 Contention Point, the link monitor implementation

We will implement the contention point (CP) logic, which actively implements the functions of the link monitor and is needed in all switch outputs in hardware, but first we have to make a functional description for it. The purpose of this block is to throttle the congestive flows. Messages that pass through a CP are divided into 4 groups:

- Normal payload traffic that passes through the switch is not processed
- FRPs going forward are processed as explained in Chapter 2 and further detailed in this section
- FRP responses that reached their destination and are now travelling back to their source are not processed.
- Flow-init and flow-stop messages inform the network about a new or a finished transmission and need special processing.

We will move to abstraction level 3 and show a functional description of the CP. Figure 4.4 displays the overview of the sub-processes of the module.

We assume a clock input, an RRP input and the FRP fields in the format we presented before. Our goal is to output an FRP message upon receiving an FRP message with or without editing the CR and DR fields according to the scheme description.

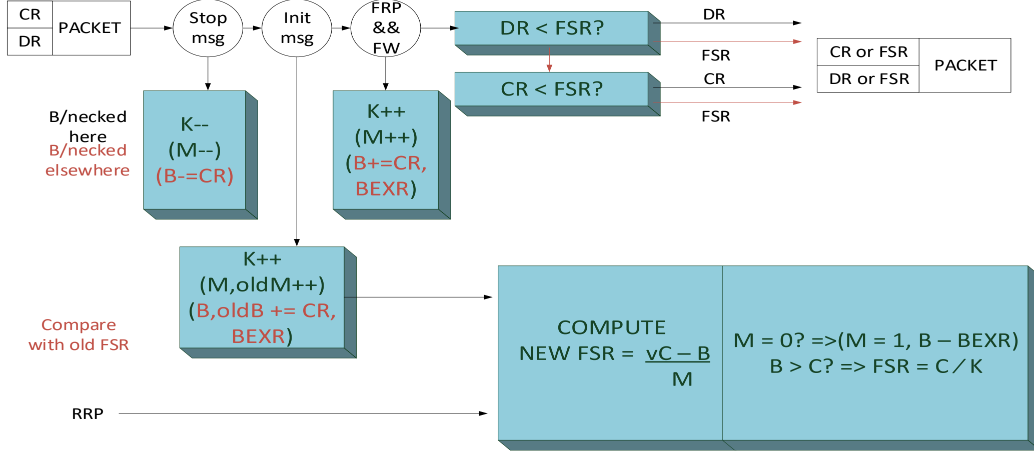


Figure 4.5: Detailed Functional overview of the Contention Point

4.3 Handling corner cases

At this point, we are deep enough to start explaining corner-case problems that have occurred and how we solve them.

- The aggregate capacity (B) of flows *bottlenecked elsewhere* exceeds the total capacity (C). This may happen when many flows start transmitting simultaneously, thus not allowing the system to realize where the real bottlenecks lie. Then, the FSR equation yields a negative FSR. To handle this case, we temporarily set the FSR as:

$$FSR = \frac{C}{K}$$

  where K is the total number of flows passing through the link.

- The number of flows bottlenecked on this link (M) is zero. In this case, the FSR equation will attempt division by 0. We handle this as follows: If B is also 0, then we do not have any flow active, so we do not care about the FSR. If on the other hand $B \neq 0$, then we use the maximum of the bottlenecked elsewhere rates value, b_{max}, that we have kept in a register, set $M = 1$, and decrement B by b_{max}.

- New flows sending init messages must not wait for an RRP clock event in order to get a rate assigned to them. Instead, the CPs may give them a feasible rate immediately. A problem is that in every RRP clock event, the M and B variables of the CP are reset to zero (0) and therefore the CP cannot assign a legitimate rate to these new flow without waiting for all flows to send their FRPs, which will only happen at the end of the RRP.

  To address this issue, the CP keeps the *old* (previous RRP) FSR, M and B values in registers, and uses these old values to find a sending rate for

the new flow, considering that the new flow was present in the previous RRP. In particular, the CP compares the CR field with the old FSR, and correspondingly updates the old M and B values. It then calculates a new FSR using these updated values.

Getting into more detail on level 4 displayed in Figure 4.5 we can now see what operations are triggered from what inputs. Note here that K represents the total number of flows (bottlenecked or not).

- Upon reception of a stop message, the module should reduce the total amount of flows, and if the received CR corresponds to a flow that is *bottlenecked here*, it should decrement M by 1. Otherwise it should decrement B by the CR amount. It should then forward the message to the next node in line, without any other processing

- Upon reception of an init message, the module will increment the total amount of flows by one, and then depending on the bottleneck status, it should increment either M by 1, or B by CR. It will do the same for the old counters mentioned in the corner cases. The init message will trigger an FSR recalculation, and use the newly calculated FSR for the rest of the procedure. The rest of the procedure is identical to the procedure that occurs upon reception of a normal FRP message, detailed bellow

- Upon reception of a 'normal' FRP message- a message that is not a stop message or an init message- and the fields FRP and FW are set to 1, the module will compare the CR and DR fields to the FSR, and again depending on the bottleneck status and similarly to the init message handling, it will increment either M or B, but not the old values. Depending on the comparison it might also change the CR, the DR, or both to the value of FSR. More specifically:

  - If the DR is bigger than the FSR, and the CR is bigger then the FSR, they both change their value to the FSR and the packet is forwarded
  - If the DR is bigger than the FSR but the CR is smaller, the DR changes its value to FSR and the CR remains as is, and the packet is forwarded.
  - If the DR is smaller than the FSR and the CR is also smaller than the FSR, both remain intact and the packet is forwarded

- Upon reception of a packet, that is none of the above types according to its flags, it will be forwarded to the output without any processing

- Upon the event of RRP, or any FSR calculation such as init message triggered recalculation, the new FSR is calculated as described in Section 2.5, and the corner cases are resolved as described in this section

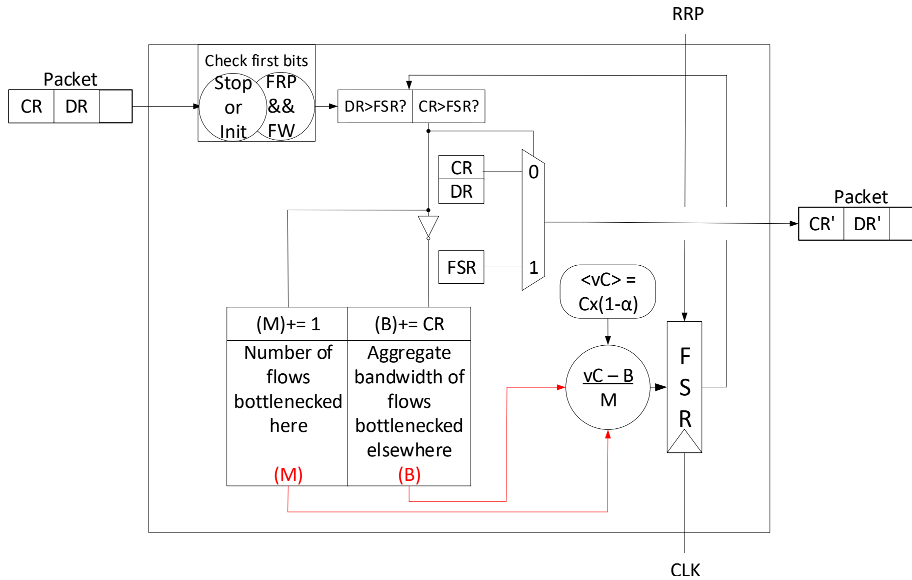


Figure 4.6: Outline of the register transfer level (RTL) implementation of the CP.

Our circuit that implements the CP is fairly simple, a coarse version is shown in Figure 4.6, but we will elaborate on the full design. Our implementation is generic, and does not assume any specific network packet format, such as Ethernet. We assume an FRP data packet input, an RRP clock input, and a data output. We deliberately ignore needed input/output signals like ready, valid, reset as well as many inner structures as they do not contribute significantly in the understanding of the concept.

The datapath is oriented around the incoming packet and the incoming RRP signal. Upon receiving an FRP packet, a CP first determines if this is a flow-init or a flow-stop message. The CP also maintains variables M, B and FSR, as explained in Section 2.5.

If the message is a forward FRP, the CP compares the FSR with the CR. If FSR $\leq$ CR, it increases M and updates the CR field of the packet with FSR. Otherwise, it increases B by the CR value of the packet.

At the end of the RRP, i.e. at every RRP clock event, the CP calculates the new FSR value and keeps the old values M, B and FSR variables in its "old" registers; these are not displayed in this figure for clarity, but will be shown in the next figure for deeper elaboration.

The full circuit is displayed in Figure 4.7, and it includes all sub-components and corner case solutions. Note that it diverges slightly from common RTL design conventions and that is to mitigate excess complexity. However, great effort has been made to ensure there is no ambiguity in the demonstration. For this same purpose, we use several colours to mitigate confusion. A rough labelling of the colours is:

- red - represents some actions related to stop messages
- blue - represents some actions related to init messages
- gold - represents some actions related to the B counters, mostly to avoid confusion with the Mcounters that have wiring in the same area
- green - represents the path of the RRP signal
- turquoise - represents the corner case solutions

We will now explain all the functionality in detail, starting from packet reception. It is not displayed in the schematic, but checking for a stop message or init message also includes checking for the packet to be an FRP.

As already mentioned before, the first check is for a stop message. If the packet has the stop-message flag and the FRP flag, if needed, set. The red arrows and lines show the circuit responsible for the stop message process. Upon identification of such packet, it will reduce the total amount of flows (K) by 1, and in order to decrement the correct corresponding counters, both current and old values, it will determine whether the flow was *bottlenecked here* or *elsewhere*, by following the forward FRP path, as the red line, extending to the right of the comparison indicates. At the end of the red line the path does not stop, but rather the default circuit, that handles normal FRP packets, takes over and completes the task, propagating the stop packet to the output. As implied before the flag bits are propagated by default.

If the packet is not a stop message FRP, the circuit will continue to check whether it is a flow init message. If so, it will broadcast the blue line across the circuit which causes the following things:

- The *bottlenecked here/elsewhere* counters will increment together with their old values. If the flow is *bottlenecked elsewhere* and its rate is the highest of the other *bottlenecked elsewhere* flows, its rate will be stored in the b_{max} register
- The FSR value will be recalculated- Note the turquoise segments that cover the corner cases
- The CR will be compared with the old FSR, but the CR will be replaced by the newly calculated FSR, if needed, when the FRP is propagated to the next node. Note that DR does not matter for init messages- it will always equal to the CR, since it is by default the first message sent by a flow

Upon finishing the tasks it will propagate the init-message FRP, changing the CR field if needed, but leaving the flags intact.

If the packet is neither a stop message nor an init message, it can be a normal FRP, in which case it will be checked for the forward flag to determine if it is going forward or backwards. If it is going backward, or of it is not an FRP at all,

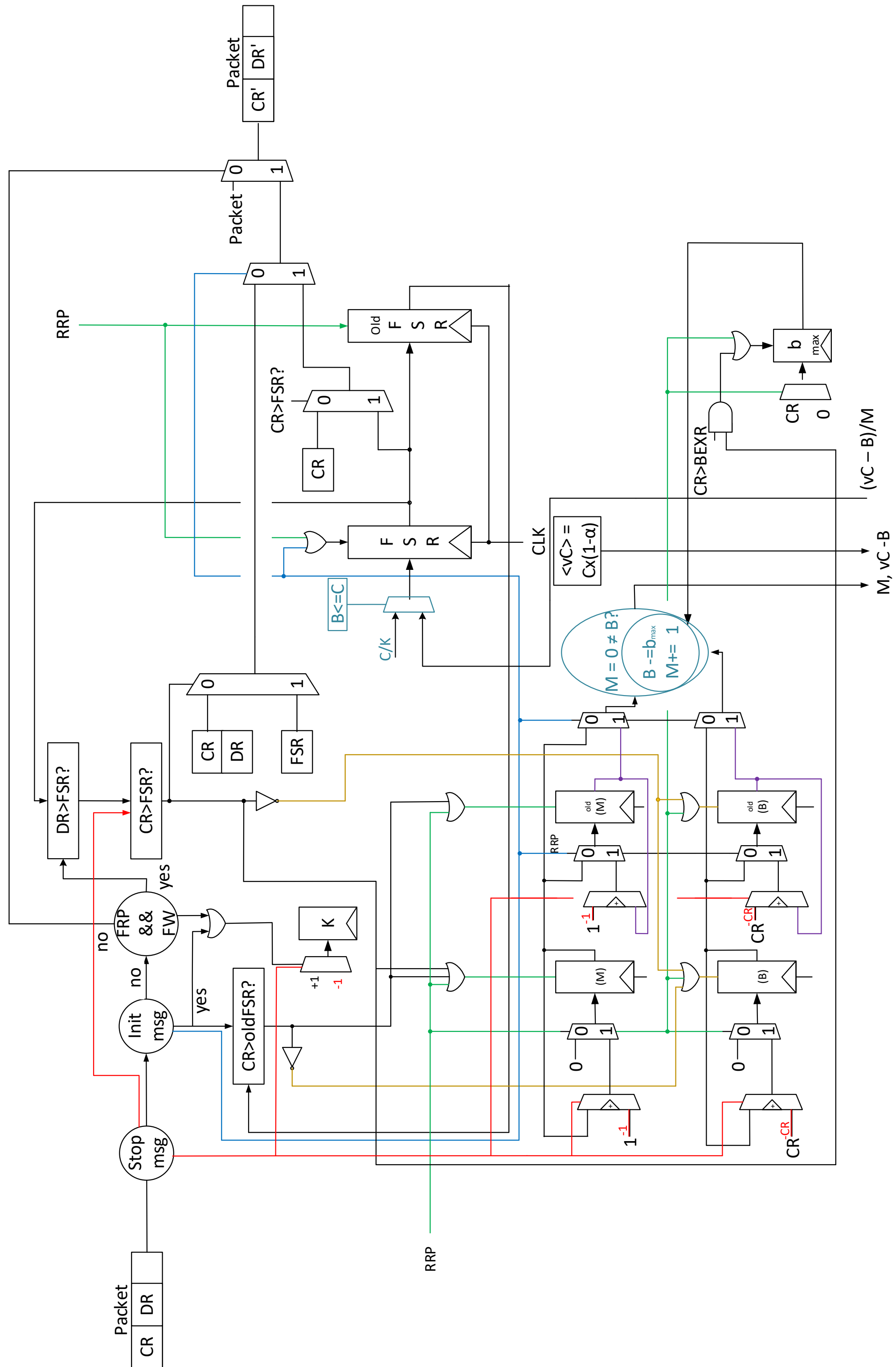


Figure 4.7: Full register transfer level (RTL) implementation of the CP.

then the packet will be forwarded to the output as is, as the rightmost multiplexor in Figure4.7 displays. If the FRP is going forward, it will compare the CR and DR with the FSR and forward the lower one each time to the output. In the figure there should be two multiplexors, one for each field, but because they are completely identical we display a merged version for simplicity. The FRP will be forwarded to the output with the flags remaining intact and the CR and DR, that were chosen in comparison.

At the same time, the circuit will increment the corresponding counter depending on the result of the comparison between the FSR and the CR. Again, if the flow is *bottlenecked elsewhere* and its rate is the highest of the other *bottlenecked elsewhere* flows, its rate will be stored in the b_{max} register.

When the RRP ends, the RRP signal is triggered. The RRP signal path is outlined in green. During the RRP, the 'old', previous values of the registers are updated, and the current values, as well as b_{max}, are reset to 0. The current FSR register is not reset to 0, but rather a new FSR is recalculated.

Here we would like to note a big difference between Figure 4.6 and Figure 4.7. In the former figure, the division that calculates the FSR, takes place inside the module, while in the later, the division does not appear, and in its place there are outputs of the divider and the divisor, and an input of the result. This is because the division is very complex and time consuming, so many designs will prefer to avoid creating their own, and use an optimized module, as we did by using the Xilinx Divider Generator [31]. We will talk about our implementations and module layout later.

It is also notable that during the FSR calculation, the first two corner cases might appear, if M is 0 and B is not, then we will attempt to divide by 0 so we act as the solution mentioned in the corner cases section. The turquoise pieces in the middle lower part of the schematic display that solution. M is assumed to be 1 and B is assumed to be b_{max} units less.

If the aggregate rate of flows *bottlenecked elsewhere* (B) is bigger than the capacity, which can only happen when a sudden burst of many small new flows, then for the short period that this will last, we divide the capacity to number of flows, seen again in turquoise in the middle of the schematic.

4.4 Implementation on FPGAs

As mentioned in Chapter 3, we developed the hardware using the Verilog hardware description language on the Xilinx Vivado tools. We will briefly elaborate on the process from the beginning until the point we have reached now.

After completing all the abstraction level schematics, demonstrated earlier in this chapter we were confident that we knew what we are creating. We used Figure 4.7 as a point of reference and coded in Verilog language from scratch, starting from the registers, multiplexors, comparators, adders and subtracters in that order, and then we added the wiring to connect them all together. We then

added the Xilinx Divider Generator tool to generate a divider for our purpose, and we wired it with the rest of the circuit accordingly. See the Appendix for a demonstration of what each of those units is represented as in Verilog code, as well as how the divider module is instantiated.

Next thing on the procedure of module development is debugging. We ran Verilog tests on the design, with different parameter values (like CR size and Capacity) and clock and RRP frequencies, using Verilog testbench modules on the Xilinx Vivado Simulator. Some of the tests were parallel test-verifications, using functions to predict the state of the system and if the predicted state was not the same as the actual state, this would mean the discovery of a bug. See the Appendix for a small example test, as well as part of the state prediction function.

The tests that we have mentioned so far are only behavioural simulation tests. That means that the simulator makes a functional plan of the code and simulates how the circuit logic reacts depending on the test inputs. This however does not take into account physical constraints such as timing and area complexity, and so the next step was to synthesize and implement the design for a specific FPGA. For the first tests a ZedBoard [21] FPGA was used. After running the synthesis and implementation tools of Vivado, we performed the same tests using a clock frequency of 150.25MHz and a RRP of 20μs. The tests in the end were successful in timing restrictions and the results showed low device utilization as expected.

The next step was to implement the module as a component in an existing design and test it in hardware. At that point, we were using the AXI protocol for communication between devices, so the module needed an AXI interface to be able to receive outgoing packets, process them, and pass them to the next node.

4.4.1 AXI Implementation Attempt

In order for the scheme to communicate using the AXI protocol, we needed our design to follow the interface conventions. We implemented a wrapper in Verilog to connect our implementation to the AXI Interface. Simulations were ran but ultimately it was never implemented on an actual hardware FPGA as it was discovered that Xilinx implementation of AXI communication components did not apply Quality of Service as the Protocol dictated and treated all traffic equally, thus not allowing the FRP packets to have high priority. With no guarantee of immediate transmission, those packets could be stuck in queues, and on arrival, their information would already be invalid. Similar problem was affecting the RRP transmission as we will explain later in Section4.6.

4.4.2 EXAnet Implementation Attempt

Ultimately the AXI Interface caused a lot of other difficulties for the project and it was replaced for node to node communication by a custom network communication protocol, which we will refer to as EXAnet. The protocol is pretty simple, each data transmission consists of 3 transfers, the header, the payload, and the footer.

Module	LUTs	Flip-Flops
Divider	108	83
CP	117	86
Total	225	171
Total %	0.08%	0.03%

156.250 MHz clock frequency

Module	LUTs	Flip-Flops
Divider	105	83
CP	125	86
Total	231	171
Total %	0.08%	0.03%

300.030 MHz clock frequency

Table 4.2: FPGA Area utilization for two different clock frequency inputs

The channel width is 128 bits but we do not need to explicitly explain each of the fields, only the bits that affect us, which are the type field in the header, and the user bits field in the footer. The type field in the header is important as a specific 5-bit combination will represent an FRP message. The actual data for the FRP will be contained in the footer user bits and the payload can freely be used to transmit data.

What the above means is that the FRPs can exist in the network with potentially zero overhead, as active flows will just need to contain an FRP on the first transmission of every RRP. At the same time, there is no need for the module to check for FRP flag internally as the type field covers this comparison. The module container or wrapper will determine whether the packet should be processed by this module or not.

The module with and without the EXAnet wrapper has been placed, synthesized and implemented on an UltraScale+ design and Table 4.2 displays the timing and area results that Vivado outputs. We use a clock frequency of 156.250 MHz as it is the Frequency of the system we are working on, but the module can achieve a Frequency of up to 300.030MHz on the UltraScale+.

Ultimately, it has not been downloaded on the FPGA to be tested yet, due to limited time and lack of understanding that could not catch up with the rapid ongoing development of the design, and is left for future work. That being said we are confident that upon verification, only a small number of errors could arise and the probable cause is imprecise timing calculation by the Vivado tool, which can be compensated by adding one cycle delay, using pipeline registers, before the data output, or by adding 1 to 4 cycles delay after the FSR calculation. This last solution will be of help, if timing is compromised during a burst of init-message FRPs, arriving back-to-back.

4.5 Reaction point at DMA engines

A Reaction Point is implemented at each DMA engine, which sources rate controlled DMA channels. The reaction point receives feedback for the flows it is responsible for, and adjusts the rate accordingly. This rate limiting function can be implemented using a leaky-bucket or a priority heap, similar to QCN's [12]

reaction points. For instance, the DMA engine inside the processing system of UltraScale+ [32] already implements rate control on the eight (8) virtualized DMA channels that it provides.

Some systems provide user control over the rate the DMA is sending. On these systems, the Reaction Point only needs to parse the returning FRPs, and pass the information on to the DMA. Other systems do not provide such feature, and in those cases, it is the Reaction Point that should enforce it. To do so, it must be placed on the output of DMA channels, and have storage capacity to hold packets as it injects them to the network in the appropriate rate.

So far, we have not made any implementation attempts for the second scenario, but we have attempted to experiment with the first scenario in our project environment. This remains to be completed as future work.

4.6 Broadcasting a common RRP clock event

Our protocol assumes that a central clock generates synchronous events to all modules with a frequency of 50kHz for an RRP of 20 microseconds. For rack-scale interconnects, a custom protocol can be used to achieve this distribution of a common clock. For instance, in general LAN networks, the Precision Time Protocol (PTP) is already approaching these accuracy levels [33].

It should be noted that many communication protocols will not allow a single bit/line of communication for RRP transmission. Therefore RRP has to be transmitted as a normal data packet, but we are hopeful that such protocols will allow high priority enforcement and the RRP can be transmitted without delay. For this work, the RRP event had to be created as a different module. After research we have concluded that there are two optimal choices, each on its own field.

- Ad-hoc chain transmission
  The period is generated by all nodes - every node transmits the RRP to all its connections, and ignores all RRP notifications within a small time after receiving one notification. This allows networks to have dynamic topology, but it is quite traffic expensive

- Specific node transmission
  The period is generated by a central node that passes it to all its connections. Only nodes that are out of reach from the central node need to receive the RRP from a chain. This results in minimum traffic but the topology can not change dynamically.

The previous statements implied the general concept of flow and rate counting as to (a) define a global constant RRP, and (b) arrange so that all flows periodically transmit FRPs equally spaced in time by RRP among themselves. In practice, this has the problems of (i) arbitrary phases of when each FRP is transmitted, even if all of them are transmitted with the same period RRP, and (ii) variability of the

delays in-between, when each FRP is generated and when it reaches downstream switches.

One method to resolve these problems is for all flow sources to transmit their FRPs not only within the same period, RRP, but also with the same phase i.e. synchronized in time. To achieve this, all source nodes and all switches have a timer, and these timers are kept synchronized among themselves, for example using periodic control messages. These synchronized timers define the boundaries between the successive rate re-evaluation periods, so that these boundaries are synchronous across the entire network (within a tolerance, that is a small percentage of the RRP). Then, all flow sources may transmit one FRP during the first half of each RRP.

In general, the RRP transmission might create timing implications, as the time for the signal to be transmitted might be big enough to bring the devices out-of-sync, each being in a different period than the other. By that, numbers such as the aggregate throughput will be unreliable, but this problem is commonly faced in distributed-system algorithms and has yet to be solved completely and therefore all we can do is just be aware of it.

The other problem is that if you use certain implementations of protocols such as the Xilinx AXI implementation for communication among devices, you can not have a single bit that is atomic and direct. You have to use the implemented features, potentially sending big packets of data, without any priority enforcement and risk delaying in queues along the way.

Chapter 5

Conclusion and Future work

Modern high-speed networks shift away from traditional TCP-based communication, adopting RDMA transfers in order to achieve lower latency. These networks will need hardware-based congestion management in order to deal with queue and buffer overload. In this work, we described an efficient, accurate and fair congestion control scheme that can be readily implemented inside the hardware of RDMA engines and switches. Our simulation results have demonstrated that our solution reacts promptly to congestion, throttles the offensive flows by allocating bandwidth in a max-min-fair manner, and that it reduces the flow completion time. We have implemented the contention point logic in FPGA hardware. In our implementation for a Xilinx Ultrascale+ FPGA, the CPs needed at the outputs of a 16-port switch occupy around 1% of the FPGA resources and easily achieve and surpass the targeted frequency of 156.25 MHz.

The outcomes of this work are not limited to the implementation of the contention point module. An outcome far more important is that the whole procedure of this work greatly developed our understanding on the topic of congestion, congestion control, the networking needs of High Performance Computing facilities, and how to effectively manage large bandwidth capacities.

At the beginning, this thesis appeared to be engineering work, taking a patented scheme and implementing it. On the contrary, this work had a significant academic aspect, since the failures in design, the lacking feasibility, the dead ends, corner cases and all other forms of adversity encountered, incited research activity, which is now interwoven in this thesis. We might not have completely reached the goal of implementing congestion control in an environment of thousands of cores, but the field is set for future work to do so.

In future work, we will complete what is left of the Contention Point, loading the EXAnet design on the UltraScale+ FPGA and making verification tests and measurements. We will also add the short circuit notifications optimization that is not featured yet. The next step is to actively claim control over the sending rate of DMAs with our Reaction Point logic, perhaps by implementing another module that could be placed at the source of any node, using any protocol.

The last step of the work is to make sure that the scheme lives up to its name; it must be able to perform well in exascale systems, but this test will require further progress of the ExaNeSt project as a whole.

Bibliography

[1] Y. Durand, P. M. Carpenter, S. Adami, A. Bilas, D. Dutoit, A. Farcy, G. Gaydadjiev, J. Goodacre, M. Katevenis, M. Marazakis, E. Matus, I. Mavroidis, and J. Thomson, "Euroserver: Energy efficient node for european micro-servers," in *2014 17th Euromicro Conference on Digital System Design*, Aug 2014, pp. 206–213.

[2] M. Welzl, *Network congestion control: managing Internet traffic*. John Wiley & Sons, 2005.

[3] D. Crisan, A. S. Anghel, R. Birke, C. Minkenberg, and M. Gusat, "Short and Fat: TCP Performance in CEE Datacenter Networks," in *Proc. IEEE Hot Interconnects*, 2011.

[4] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center tcp (dctcp)," in *Proceedings of the ACM SIGCOMM 2010 Conference*, ser. SIGCOMM '10. New York, NY, USA: ACM, 2010, pp. 63–74. [Online]. Available: http://doi.acm.org/10.1145/1851182.1851192

[5] Exanest project. [Online]. Available: http://www.exanest.eu/

[6] Manolis Katevenis , FORTH, "Dynamic max-min fair rate regulation apparatuses, methods, and systems," in *US Patent Application Number 14/864,355 filed 24 Sep 2015, and Int. (EPO) Application Number PCT/EP2015/072048*, see also USPTO Publication Number US 2016/0087899 A1, published on 24 Mar 2016.

[7] J. Duato, I. Johnson, J. Flich, F. Naven, P. Garcia, and T. Nachiondo, "A new scalable and cost-effective congestion management strategy for lossless multistage interconnection networks," in *11th International Symposium on High-Performance Computer Architecture*, Feb 2005, pp. 108–119.

[8] Q. Liu, R. D. Russell, and E. G. Gran, "Improvements to the infiniband congestion control mechanism," in *2016 IEEE 24th Annual Symposium on High-Performance Interconnects (HOTI)*, Aug 2016, pp. 27–36.

[9] P. Faizian, M. S. Rahman, M. A. Mollah, X. Yuan, S. Pakin, and M. Lang, "Traffic pattern-based adaptive routing for intra-group communication in dragonfly networks," in *2016 IEEE 24th Annual Symposium on High-Performance Interconnects (HOTI)*, Aug 2016, pp. 19–26.

[10] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson, "Bbr: Congestion-based congestion control," *Queue*, vol. 14, no. 5, pp. 50:20–50:53, Oct. 2016. [Online]. Available: http://doi.acm.org/10.1145/3012426.3022184

[11] B. Cronkite-Ratcliff, A. Bergman, S. Vargaftik, M. Ravi, N. McKeown, I. Abraham, and I. Keslassy, "Virtualized congestion control," in *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM '16. New York, NY, USA: ACM, 2016, pp. 230–243. [Online]. Available: http://doi.acm.org/10.1145/2934872.2934889

[12] M. Alizadeh, B. Atikoglu, A. Kabbani, A. Lakshmikantha, R. Pan, B. Prabhakar, and M. Seaman, "Data center transport mechanisms: Congestion control theory and ieee standardization," in *2008 46th Annual Allerton Conference on Communication, Control, and Computing*, Sept 2008, pp. 1270–1277.

[13] Y. Zhang and N. Ansari, "Fair quantized congestion notification in data center networks," *IEEE Transactions on Communications*, vol. 61, no. 11, pp. 4690–4699, November 2013.

[14] N. Chrysos, F. Neeser, R. Clauberg, D. Crisan, K. M. Valk, C. Basso, C. Minkenberg, and M. Gusat, "Unbiased quantized congestion notification for scalable server fabrics," *IEEE Micro*, vol. 36, no. 6, pp. 50–58, Nov 2016.

[15] R. Jain, "Congestion control in computer networks: issues and trends," *IEEE NETWORK MAGAZINE*, vol. 4, pp. 24–30, 1990.

[16] J.-Y. Le Boudec, "Rate adaptation, congestion control and fairness: A tutorial," *Web page, November*, 2005.

[17] D. P. Bertsekas, R. G. Gallager, and P. Humblet, *Data networks*. Prentice-hall Englewood Cliffs, NJ, 1987, vol. 2.

[18] N. Dukkipati, M. Kobayashi, R. Zhang-Shen, and N. McKeown, "Processor sharing flows in the internet," in *Proceedings of the 13th International Conference on Quality of Service*, ser. IWQoS'05, 2005, pp. 271–285.

[19] Ioannis Vardas , Nikolaos Chrysos, "Simulation runs in omnet++," *Internal work*, CARV ICS, October 2016.

[20] A. Varga and R. Hornig, "An overview of the omnet++ simulation environment," in *Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2008, p. 60.

[21] "Zedboard development kit." [Online]. Available: http://zedboard.org/

[22] "Zynq ultrascale+ mpsoc." [Online]. Available: https://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html

[23] "Vivado design suite." [Online]. Available: https://www.xilinx.com/products/design-tools/vivado.html

[24] "Microsoft visio flowchart software." [Online]. Available: https://products.office.com/en-us/visio

[25] "Microsoft word, word processor." [Online]. Available: https://products.office.com/en-US/word

[26] "Notepad++ source code editor." [Online]. Available: https://notepad-plus-plus.org

[27] "Miktex tex compiler." [Online]. Available: https://miktex.org

[28] "gnuplot graphing utility." [Online]. Available: http://www.gnuplot.info/

[29] "Xilinx synthesis and simulation design guide," Xilinx Inc, Tech. Rep., December 2009. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx11/sim.pdf

[30] "Vivado design suite user guide," Xilinx Inc, Tech. Rep., April 2014. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_1/ug910-vivado-getting-started.pdf

[31] "Xilinx divider generator," October 2016. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/div_gen/v5_1/pg151-div-gen.pdf

[32] V. Boppana, S. Ahmad, I. Ganusov, V. Kathail, V. Rajagopalan, and R. Wittig, "Ultrascale+ mpsoc and fpga families," in *Hot Chips 27 Symposium (HCS), 2015 IEEE*. IEEE, 2015, pp. 1–37.

[33] A. Mahmood, G. Gaderer, H. Trsek, S. Schwalowsky, and N. Kerö, "Towards high accuracy in ieee 802.11 based clock synchronization using ptp," in *Precision Clock Synchronization for Measurement Control and Communication (ISPCS), 2011 International IEEE Symposium on*. IEEE, 2011, pp. 13–18.

Appendix A

Verilog interpretation examples

```
always @(posedge i_clk) //on positive edge of clock
begin
        if (reset)            //if the reset signal is on
                Q <= 1'b0;  // Q = 1bit value 0
        else
                if(EN)        // otherwise check Enable
                        Q <= D;       // then Q = input
end
```

Listing A.1: Example of a Flip-Flop with synchronous reset and enable that triggers on the positive edge of the clock

```
always @(in0 or in1 or in2 or in3 or sel)
 begin
 //for every different value of select (sel)
 //output a different input
        if (sel == 2'b00) D = in0;
        else if (sel == 2'b01) D = in1;
        else if (sel == 2'b10) D = in2;
        else D = in3;
 end
```

Listing A.2: Example of a 4to1 signle bit multiplexor

```
always @(posedge i_clk)
begin
        if (reset)
                Q <= 1'b0;
        else
                if(EN)
                        if (sel == 2'b00) Q <= in0;
                        else if (sel == 2'b01) Q <= in1;
                        else if (sel == 2'b10) Q <= in2;
                        else Q <= in3;
end
```

Listing A.3: An unorthodox Example of a 4to1 single bit multiplexor with a flip flop

Note the technique of Listing A.3 is not advised by the guide, but as it saves time and space we use it commonly and the Xilinx compiler translates it correctly.

```
Divider my_divider(     // Module name and instance name
//.wire_name_of_module(wire_to_connect_from_top_module)
        .aclk(clk),
        .dividentData(CminusB),
        .dividentValid(divInValid),
        .divisorData(M),
        .divisorValid(divInValid),
        .divOutData(FSRcalc),
        .divOutValid(FSRcalc_valid)
        );
```

Listing A.4: Divider Instantiation Example

```
task RRP;
        thisFSR = (thisCap - thisB)/ thisM;
endtask
task MorB;
        begin
        if( i_data[11:4] > thisFSR )
                thisM = thisM + 1;
                $display("M is : %d", thisM);
        else
                thisB = thisB + i_data[11:4];
                $display("B is : %d", thisB);
        end
endtask
```

Listing A.5: Concept of State Prediction

```
task GEN;
        begin
        if(random == 1'b1)
                i_data =  $urandom;
        else
        begin
                //DR_CR
                i_data[19:4] =  16'b10101010_10101010;
                i_data[3:0]  =  4'b1100; //flags
        end
        end
endtask

initial
begin
        i_RRP =     1'b0;
        aclk    =  1'b0;
        reset   =  1'b1;
        i_data  = 20'b0;
        i_valid =  1'b0;

        #100;
        reset   =  1'b0;
        #100;

        for (i=0;i < 16;i=i+1)begin
                @(posedge aclk);#6;
                GEN();
                i_valid = 1'b1;
                #3.4 i_valid = 0;
        end
        ...
```

Listing A.6: Example test