Leveraging Domino to Implement RCP in a Stateful Programmable Pipeline

MD Iftakharul Islam, Javed I Khan

Department of Computer Science
Kent State University
Kent, OH, USA.

Outline

Rate Control Protocol (RCP)

- RCP is a router-assisted congestion control mechanism where routers allocate a feedback rate to each flow.

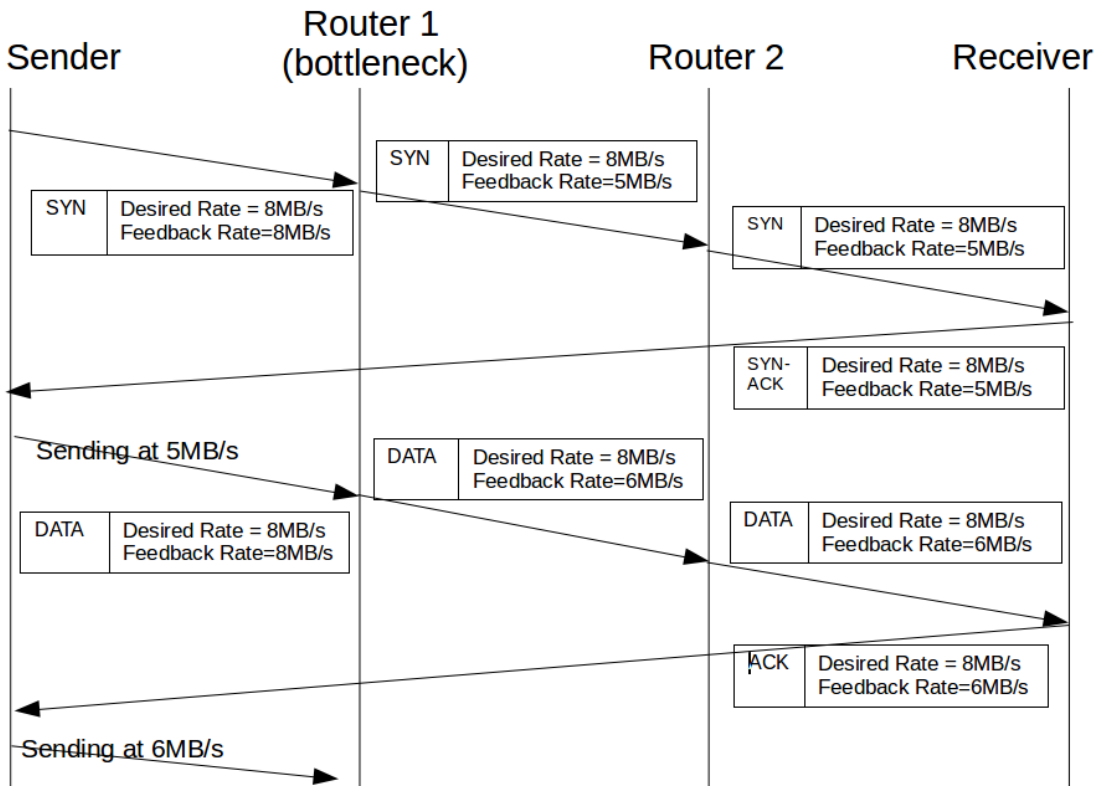


Figure: Workflow of RCP

RCP Implementation

- Router assisted congestion control algorithms such as RCP *[IWQoS 2005]*, XCP *[SIGCOMM 2002]* and NC-TCP *[ICNP 2017]* exhibit very promising results compared to their host centric counterparts.
  - Reduces flow completion time
  - Reduces queuing delay
  - Increases throughput and fairness
- They however are very hard to implement in the router dataplane.
- Currently there is no real deployment of RCP (to the best of our knowledge).
  - There are several implementations of RCP based on RMT switch, FPGA, NPU and so on.
  - None of them can achieve the line rate needed for a high-speed router (around 1 billion packets per second)
- This paper presents an implementation of RCP that can achieve line rate.

Challenges of line-rate RCP implementation

- RCP calculates throughput based on queue occupancy, spare capacity, RTT and so on.
- This is why, RCP needs to maintain states in the router data plane.
- Recently programmable pipeline based RMT switches have emerged as an alternative to hardware switches.
  - Example: Barefoot Tofino, Cavium Xplaint, Brodcom Jericho, etc.
- RMT switches maintain stateful memories such as counters, registers and meters in a centralized manner.
- Accessing those centralized memories from multiple pipelines requires explicit synchronization which is not scalable.
- This is why, RMT switches are not suitable for implementing RCP.
- Recently RCP has been developed in a Cavium Xplaint switch *[NSDI, 2017]*, but it's not clear from the paper if it can achieve line-rate.

Stateful Programmable Pipeline

- Recently Domino *[SIGCOMM, 2016]* presents stateful programmable pipeline where each pipeline stages keeps a local state.
- This is why, this approach does not need to access centralized states. So it does not need any synchronization.
- This paper presents an implementation of RCP for stateful programmable pipeline.
- We use Domino programming language to program the stateful programmable pipeline.
  - It is noteworthy that Domino paper also stated that they have implemented RCP in router data plane, but their version of RCP lacked many essential elements of RCP.

Stateful Programmable Pipeline

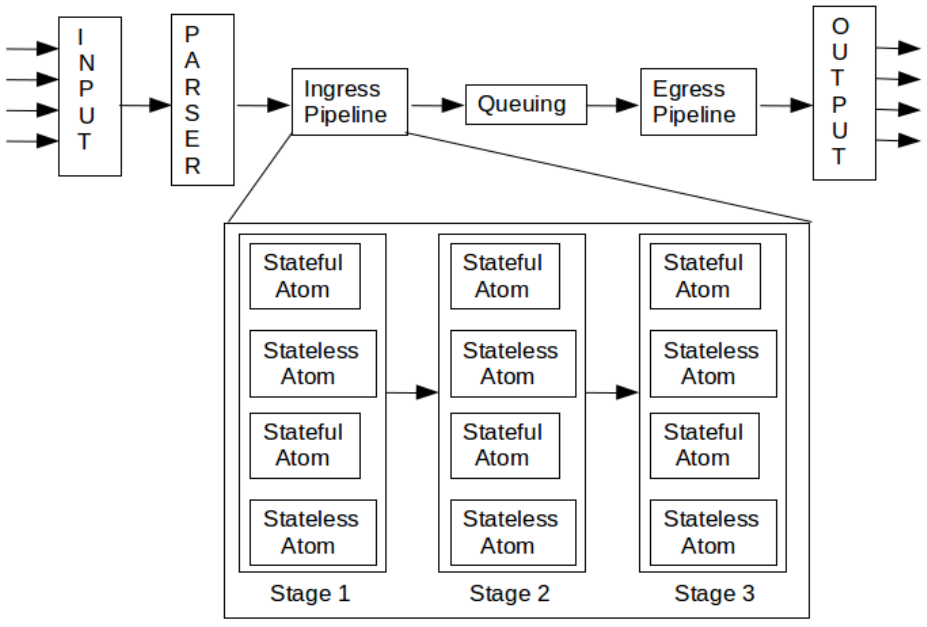


Figure: Abstract router model based on stateful pipeline

- Atoms are circuitry developed based on ALU, MUX and so on.
- Atoms in a pipeline can operate on a packet in parallel.
- Each atom can process one packet in each clock cycle.
- This is why, the pipeline can output 1 packet in every clock cycle.
- If the atoms run at 1 GHz, then the pipeline can process 1 billion packets per second.

Stateful Programmable Pipeline

- Currently actual hardware does not exist.
- But such stateful programmable pipeline can be realized using a VLIW processor.
- Domino has provided SystemVerilog implementation of the hardware pipeline showing that it can run at 1 GHz with a 32-nm standard-cell library.
- Domino programming language has been developed to develop program for stateful pipeline.
- Domino-compiler enables us to evaluate a Domino program without needing the actual hardware.
- Domino-compiler follows *all-or-nothing* approach.
  - **A program compiled by domino-compiler is guarented to run on the hardware at line-rate. If the program cannot run on the hardware at line-rate, the compilation fails**.

A look inside actual router hardware

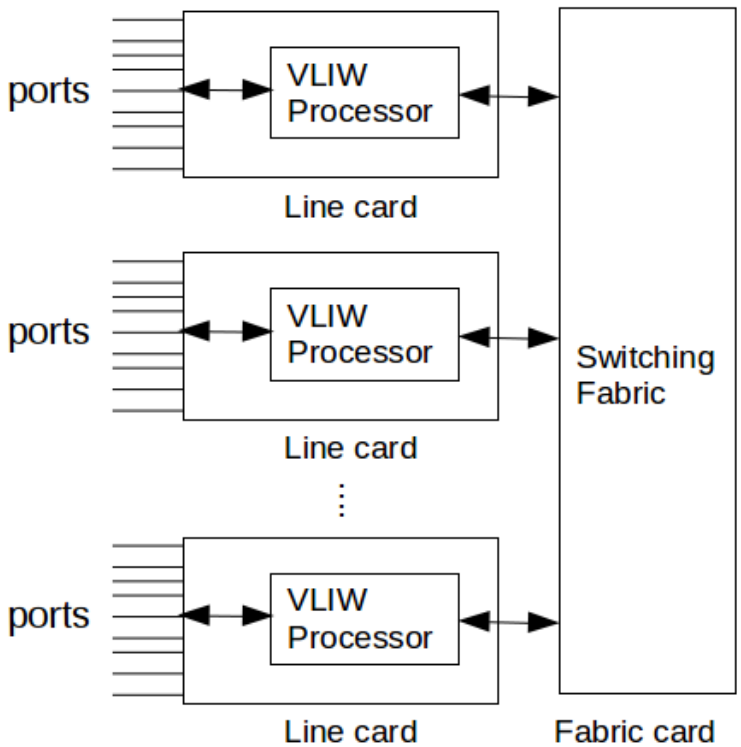


Figure: VLIW processor implements the parser and the ingress and the egress pipeline. Switching fabric works as the queue between the ingress and the egress pipeline.

Congestion in a router data-plane

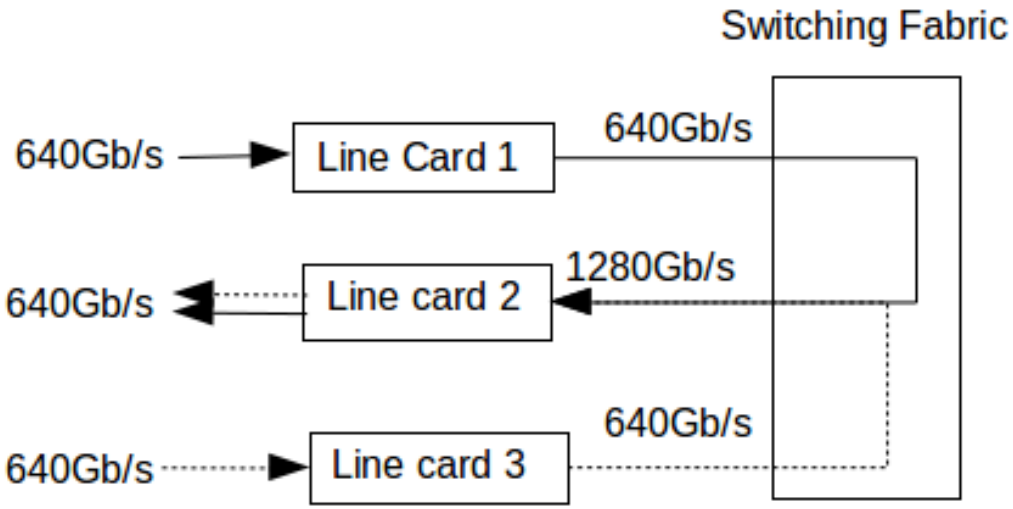


Figure: Congestion occurs due to the inability of the egress pipeline to forward packets at the incoming rate. This is why, RCP program would be executed in the egress pipeline.

Rate calculation in RCP

- RCP protocol calculates the feedback rate as:

$$R(t) = R(t - RTT_a) + \frac{\alpha.S - \beta.\frac{Q(t)}{RTT_a}}{\hat{N(t)}} \quad (1)$$

where
$R(t)$ = Feedback rate
$Q(t)$ = Queue size (for the line card)
C = Link capacity
RTT_a = Running average of RTT
S = Spare capacity
$\hat{N(t)}$ = Estimated number of flows
α and β are stability constants.

- If there a spare capacity ($S > 0$), RCP increases the feedback rate
- If there a persistent queue ($Q(t) > 0$), RCP decreases the feedback rate.

Rate calculation in RCP

- Calculating $\hat{N}(t)$ however is very expensive.
- RCP approximates $\hat{N}(t)$ as $\hat{N}(t) = \frac{C}{R(t-RTT_a)}$
- This is why, the RCP control rate becomes

$$\begin{aligned} R(t) &= R(t - RTT_a) + \frac{\alpha.S - \beta.\frac{Q(t)}{RTT_a}}{\frac{C}{R(t-RTT_a)}} \\ &= R(t - RTT_a)[1 + \frac{\alpha.S - \beta.\frac{Q(t)}{RTT_a}}{C}] \end{aligned} \tag{2}$$

- Here RTT_a is used as the control interval.
- However we want the control interval to be smaller than the RTT_a so that the router can react to the spare capacity and queuing delay sooner.
- RCP scales the aggregate change by $\frac{T}{RTT_a}$ where T is the desired control interval.
- All these equations are presented in original RCP paper.

Rate calculation in RCP

- RCP control equation becomes:

$$R(t) = R(t - T)[1 + \frac{\frac{T}{RTT_a}.(\alpha.S - \beta.\frac{Q(t)}{RTT_a})}{C}] \quad (3)$$

- We set T = 50 ms
- We set $\alpha = 1.0$ and $\beta = 0.5$ (chosen based on RCP's stability analysis).
- RTT_a is calculated as $RTT_a = .98 * RTT_a + .02 * RTT_{packet}$ for each incoming packet
- S is calculated as $S = C - \frac{B}{1000T}$ MB where B is the number of bytes received during last T milliseconds.

Implementation

- We have implemented RCP using Domino (*details are in the paper*).
- Domino has notion called *packet transaction* where we implement the packet processing logic as if the program will be applied on the packet at once.
- Domino-compiler generates a dependency graph from the packet transaction and maps it to stateful pipeline stages (*the dependency graph would be found in paper*)
- It is noteworthy that packet size, RTT, queue length, time, etc are made available to packet transaction as packet-header and meta-data as following:

```
/****Packet headers and meta-data*******/
struct Packet {
  int size_bytes;
  int rtt; //Parsed from RCP header
  int queue;
  int Rp; //RCP feedback rate
  int tick;
};
```

Implementation

- Packet parser is responsible for parsing the packet header and producing RTT, feedback rate and so on.
  - RCP sender and receiver put those information in RCP header.
  - Domino does not parse the packet. It simply assumes that packet is already parsed.
  - Packet parsing is rather trivial (can be defined by a P4 program)
- Time, queue length, packet size, etc are made available as a meta-data.
  - In-band Network Telemetry (INT) standard adopted by several switch manufacturers
- Domino has 7 predefined atoms. We need to provide the atom type in order to compile the Domino program. Here we use SUB atom to compile our program.

SUB atom

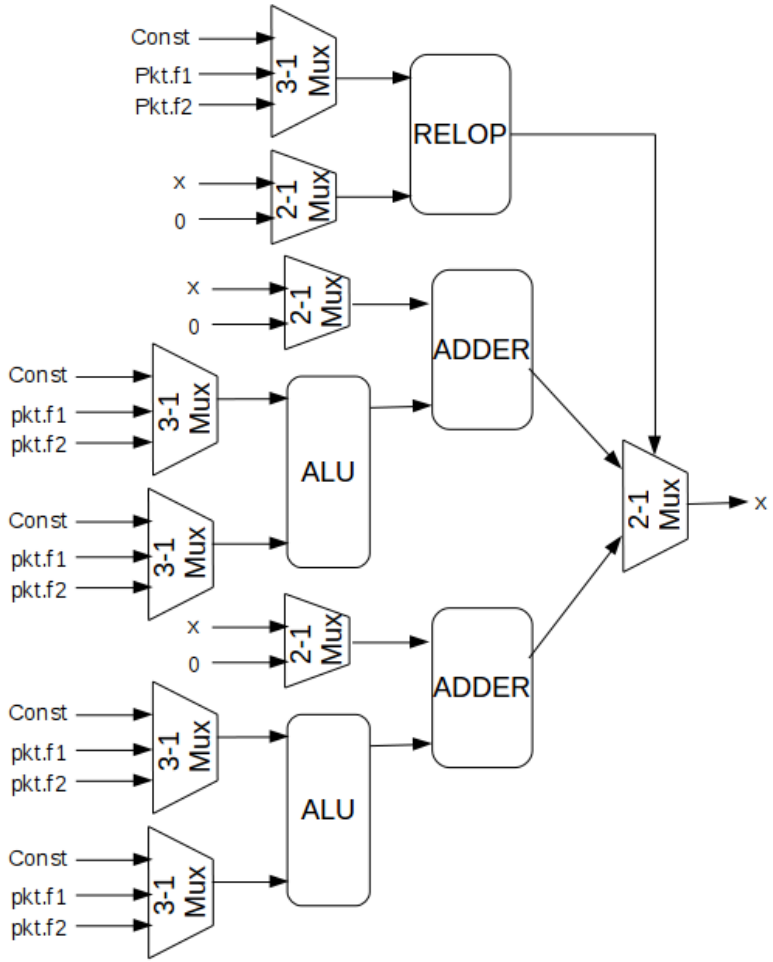
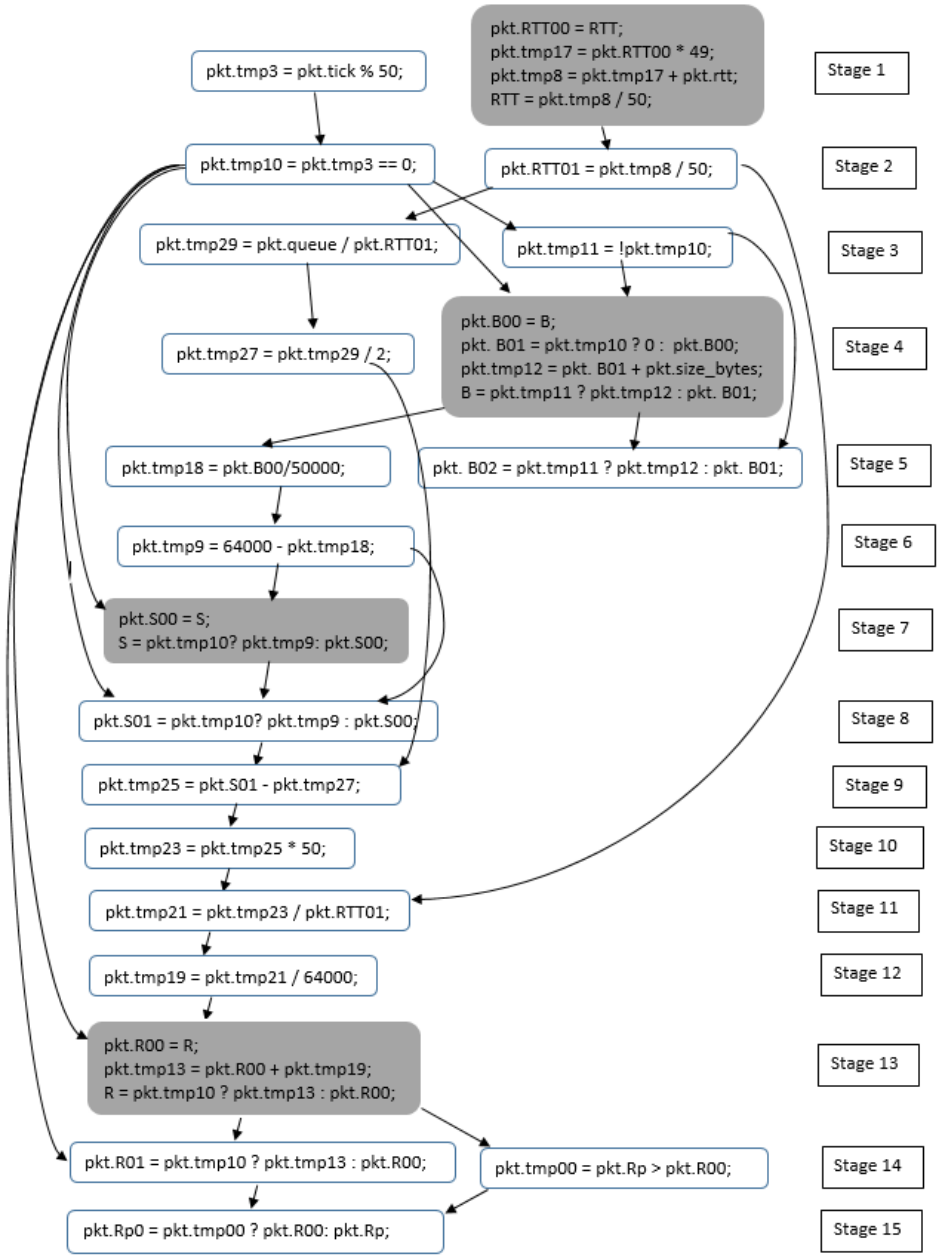


Figure: Atom used for compiling RCP. Here MUX is a multiplexer. RELOP is a relational operator ($>, <, ==, !=$). x is a state variable. ALU is the arithmetic logic unit. $pkt.f1$ and $pkt.f2$ are packet fields. *Const* is a constant operand.

Implementation

- It has shown that SUB atom can run at 1 GHz.
- Domino-compiler internally uses SKETCH tool to map between a domino program and the underlying atom
- Domino compiler can compile our program with a 15 stage pipeline where each pipeline stage contains 2 SUB atoms.
- This shows that RCP can be implemented on a VLIW processor at line-rate.

Execution of the RCP program in a 15-stage pipeline



Execution of the RCP program in a 15-stage pipeline

- Here grey boxes represents a stateful atom (SUB atom).
- White boxes are stateless atom which can be realized by a simple ALU.
- It is noteworthy that a Domino program is mapped to the pipeline during compilation. This is why it results a very deterministic results (unlike super scaler or general purpose processors)

Thank You