Requirements Discussion of Link Level-Flow Control for Next Generation Ethernet

M. Gusat, C. Minkenberg, R. Luijten, T. Engbersen and J. Navaridas

IBM Research GmbH, Zurich

Monterey Jan. 23rd, 2007

Purpose

1. Ethernet's link-level flow control (LL-FC) requirement selection
   => Separate LL-FC objectives
   1. must have (correctness)
   2. could have (performance)
   3. should not have...

2. Agree on a trade-off hierarchy: ordered list

3. Vote and record a requirements document before Florida Plenary

4. Observations
   i. No solutions are proposed. However, selected examples are shown to illustrate the case.
   ii. To avoid confusing terminology we propose the use of:
      a) LL-FC instead of PAUSE: abstract protocol mechanics
      b) "flow control domain" (FCD) instead of Prio, VL, VC: whenever possible...

Outline

- LL-FC possible requirements
  1. Loss
  2. Blocking
  3. Deadlocks
  4. Scalability

- A canonical LL-FC

- Conclusions

- Explanatory segments
  - Definitions, examples, exemplary solutions

LL-FC Requirements – 1: Loss

1. Compatible: Legacy Ethernet
   a) Defaults to **lossy** operation (virtually all legacy installations)
   b) Obeys legacy *LL-FC* semantics, when PAUSE is needed

2. **Lossless**: IPC & storage apps + SAN & StAN emulation

   LL Options
   a) In-order delivery (IOD , see (5.2.b) in Deadlocks)
   b) Reliable delivery (RD) => LL retransmission

3. (1) + (2) => **Dual** mode: simultaneous lossy & lossless
   - e.g. VL[0] defaults to lossy (no LL-FC), and,
   - VL[1] uses LL-FC for losslessness (VL = Virtual Lane)

LL-FC Requirements – 2: Blocking

4. Free of first-order blocking
   a) Priority blocking, aka priority inversion requires
      1) Multiple prios (e.g. 2-8, 802.1p) or VLs distinctly LL-FCed, and,
      2) De-blocking mechanism

   b) HOL_1-blocking, aka hogging requires Virtual Output Queueing (VOQ) demultiplexing
      1) Full: VOQ-arity = switch port count (24-256 !), or,
      2) Partial: a smaller VOQ subset (4-16) + reuse mechanism

- Observations
  - ✓ (a) and (b) are orthogonal; e.g., a 32-port switch with 8 priorities elicits up to 256 "flow control domains" (1 FCD = queue + channel)
  - ✓ Higher order blocking is NOT a LL-FC objective.

LL-FC Requirements – 3: Deadlocks

5. Free of deadlocks: Three types

1) Circular dependence deadlocks

   Why? Un-ordered access to mutually blocking resources.

   ✓ a) Memory-to-memory (inter-switch deadlock in bidirectional networks)

      » Solution requires partitioning

   ✓ b) Load/Store, Request/Reply (transaction-induced deadlock)

      » requires: 2 FCDs + strict ordering rules

2) Routing deadlocks

   Why? Cycles in the routing graph (multipath)

      » e.g., LL-FC solutions typically employ 2-4 virtual channels (VC)

LL-FC Requirements – 4: Scalability

6. Maintains losslessness and performance with increase of
   a) Signaling speed (1 – 100 Gbps)
   b) Link length (0.1 – 1000s m)

- Must reduce, ideally eliminate, the performance sensitivity to RTT (normalized delay*Bw)

Meeting All Requirements: The Canonical LL-FC Solution

Combining the LL-FC requirements for a discussion base :

1. Dual mode (lossy & lossless): 2 FCDs
2. No priority blocking: 8 FCDs
3. No HOL_1-blocking: 64 FCDs
4. No RQ/RP deadlocks: 2 FCDs
5. No routing deadlocks: 4 FCDs
6. No FCD is "overloaded" with multiple functions...

⇒ Canonical LL-FC = 2x8x64x2x4 = 8192 FCDs (aka VLs, VCs)...
... per switch! ☺

Conclusions

I. Requirements above are demanded in datacenter applications.
   - Performance, efficiency and power increase in importance
   - Correctness of operation in not optional (no “disable”)

II. The brute force (canonical) approach is not feasible for modern datacenter switches
   - Switch memory, if $M=O(\#ports^2, \#prios, RTT)$
   - LL-FC overhead (1000s of FCD IDs)
   - Logic and scheduling complexity.

III. Requirements must be prioritized
   - Sensible compromises
   - Enable the LL-FC of next-generation Ethernet.

Explanatory Segments: Definitions and examples

- First-order blocking
- Priority inversion and a 'Bulldozer' example
- HOL_1-blocking, aka hogging, in a VL-rich architecture
- Deadlocks:
  - 1. Circular dependence deadlocks
    - a) Memory-to-memory (inter-switch, 1st order deadlock)
    - b) Load/Store, Request/Reply (transaction-induced deadlock)
  - 2. Routing loops
- Scalability:
- losslessness *and* performance sensitivity to RTT

Blocking Phenomena in Packet Switching

- HOL- vs. Priority- blocking: 2 distinct blocking classes

- The difference?
  - Priority-blocking acts one-way-only, according to the prio ordering rules
    - easy on TX: any form of strict/fixed prio preemptive scheduling
    - hard on RX: full dedication per prio req'd => static partitioning (no sharing)
  - HOL-blocking acts multiway (any-to-any blocking)
    - is hard on the traffic source: needs full demux solutions, e.g., VOQ
    - is easy on the RX buffer: memory-sharing is possible

Priority Blocking
and
Priority Elevation Solution

Outline

- Datacenter switches & the NASA Mars Pathfinder
- Strict priority QoS scheduling
- Priority blocking
- Blocking phenomena in packet switching
- Solutions against priority-blocking
  - Bulldozer architecture
  - Selected simulation results
    - Implementation optimisations
- Conclusions on priority inversions in datacenter ICTNs

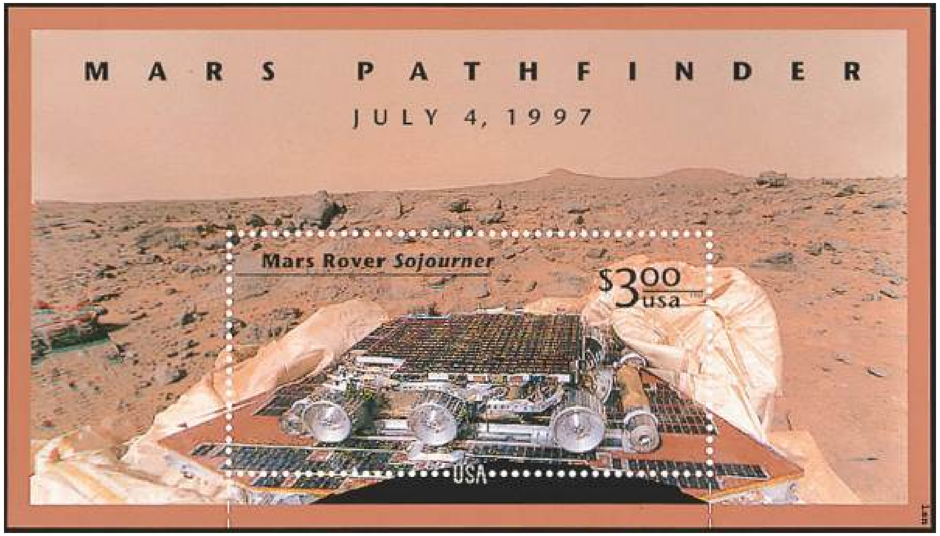
What's Common Between Switches and the Mars Pathfinder?

Interconnect switches

- Symptoms:
  - loss of T_{put}
    - no work-conservation
  - increased delays
    - no QoS compliance

affects the high-priority traffic, i.e., the premium customers...

  - if OS mutex / semaphores are based on correct QoS implementation => [deadlock]



NASA Mars Pathfinder [mission]

- Symptoms:
  - system [resets]
  - data loss
  - mission endagered...

Why?

Preemptive Scheduling

- Preemptive = exhaustive = fixed = strict prio: Basic QoS scheduling discipline
- Rigorously defines total/partial ordering rules based on a finite set of priorities
- Universally agreed (!), well-understood principles, simple implementations, yet...

- Two issues of preemptive scheduling in lossless networks
  1. [indefinite] *starvation* potential for low-prios
  2. *priority blocking*, i.e. when stale pkts of low prio flows block the forward progress of higher prio flows

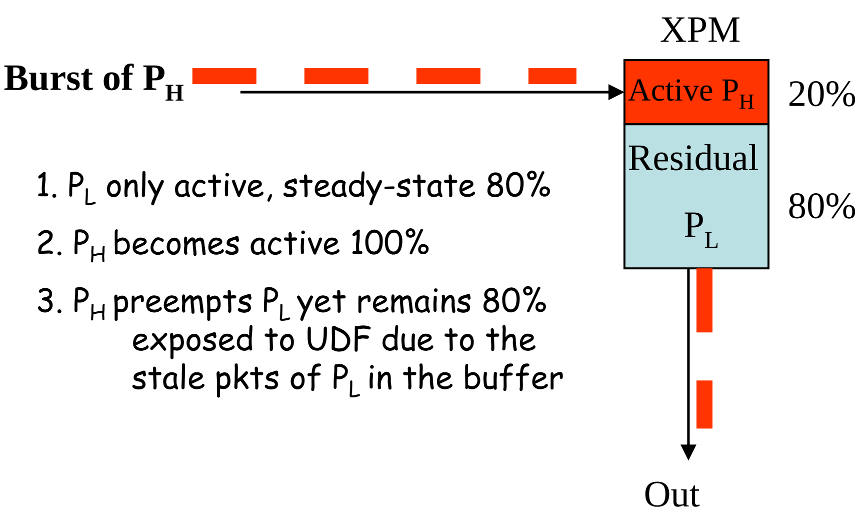
- Prio-blocking in packet switching
  - may occur whenever flows of different prios share a buffer (see next foil)
  - is [a spatial] cousin of the priority inversion problem in real-time/OS scheduling
  - aggravates proportionally with
    - no. of prios
    - link delays and buffer size
    - traffic burstiness

Priority-Blocking: When and How Can Happen?

- 4 typical requirements for datacenter interconnect switches:
  - 1) 3-8 **strict** priorities, compliant to IEEE 802.1p/q
  - 2) **Work-conservation** of any mixture of priorities
  - 3) **Hosed flows**: (∀) single {I, O}-tuple is 100% WC (*prio-indep.*)
  - 4) **Losslessness**

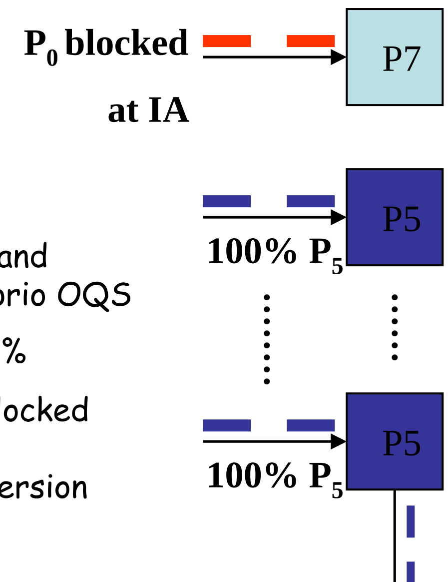
In typical implementations the buffers are shared between priorities

- **Self-induced Underflow**



1. P_L only active, steady-state 80%
2. P_H becomes active 100%
3. P_H preempts P_L yet remains 80% exposed to UDF due to the stale pkts of P_L in the buffer

Push-thru Blocking

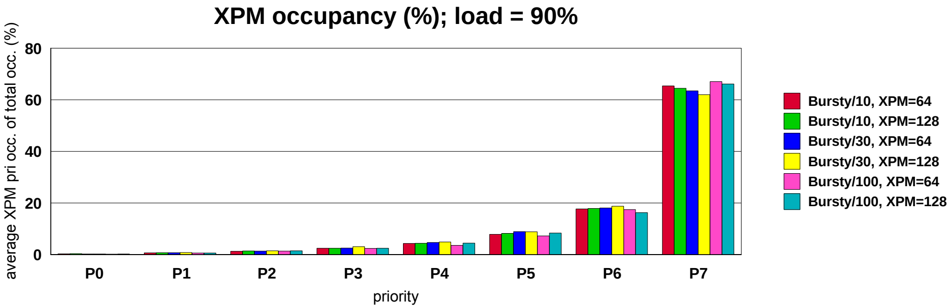


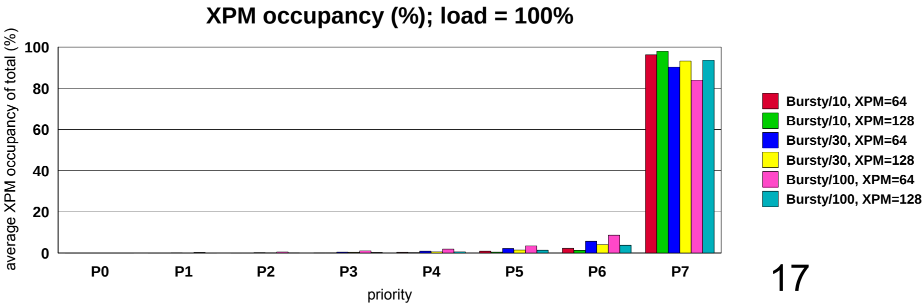
1. P_5 active at $I_{1..63}$, 100%
2. P_7 becomes active at I_0 and blocks under strict prio OQS
3. P_0 activates at I_0 at 100%
4. P_0 remains indefinetly blocked behind P_7 due to an uncontrolled prio inversion

How often does prio-blocking occur?

- Occurence/frequency of priority blocking
- Preliminary simulations
  - 64x buffered Xbar switch,1M-packet cycles
  - Bursty traffic, uniformly distributed over priorities and outputs
- Configurations
  - Load = 100%, 90%
  - Memory size (credits) = 64, 128 (192, 256) pkts
  - Avg. burst size = 10, 30, 100
- Buffer occupancy per crosspoint memory (XPM)
  - average occupancy of the entire column

Results - XPM occupancy data shows:
90% load: prio 7 > 60% memory occ.
100% load: prio 7 > 90% memory occ.
Motivation: A solution to prio-blocking is *really needed* to clear the stale p7 traffic out of the XPM to make way for higher priorities





Two Exemplary Solutions Against Priority-blocking

- S1. Full Demultiplexing (aka brute force)
  - A distinct RTT of queuing capacity is dedicated per priority
    - wasteful by a factor of P and theoretical (can't build it)
      On average only 1-out-of P prios is scheduled at any given instance. The observed memory utilization per prio with non-pathological traffic is extremely low => more wastefulness

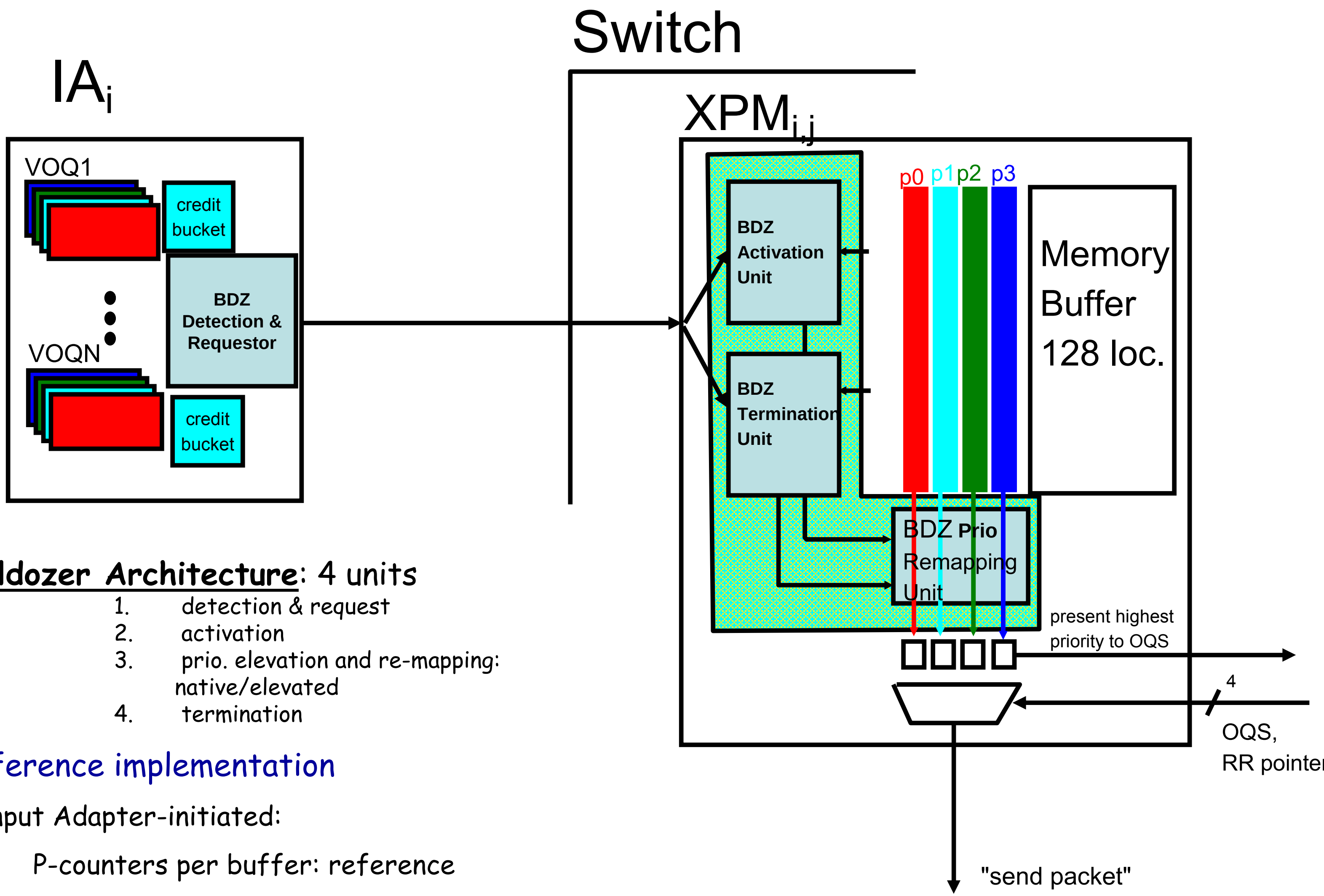
  E.g., a 64-port @ 40Gbps: $M = N^2 \times P \times RTT = 4K \times 8 \times 64 =$ 167MB!

  => Flows must share the limited fast memory capacity.

- Priority inheritance: P_H act as
  - S2. "Bulldozer": push forward the P_L towards the output
    Flush the stale buffer in "priority-elevation" mode
    - higher latency (deterministic) for P_H => justified for longer bursts
    - priority disturbance/unfairness (upper bounded)
    + lossless

Next: Sketch "Bulldozer" (BDZ) Architecture and Simulation Results

Bulldozer Architecture: The Basic Elements



Bulldozer Architecture: 4 units

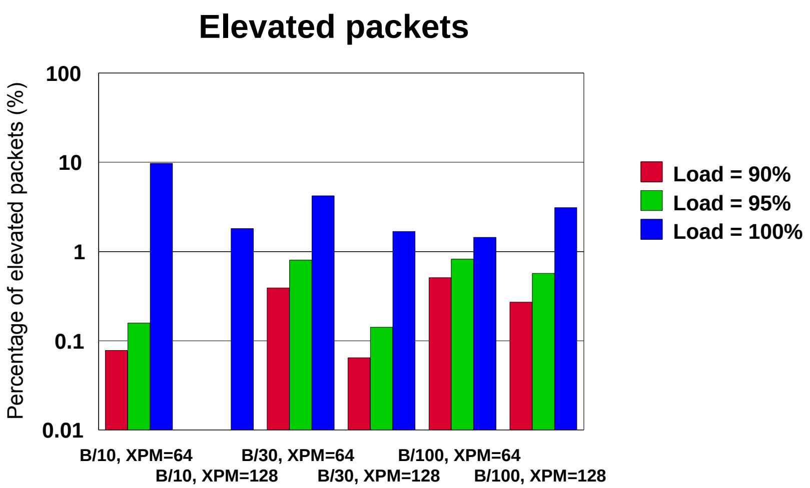
1. detection & request
2. activation
3. prio. elevation and re-mapping: native/elevated
4. termination

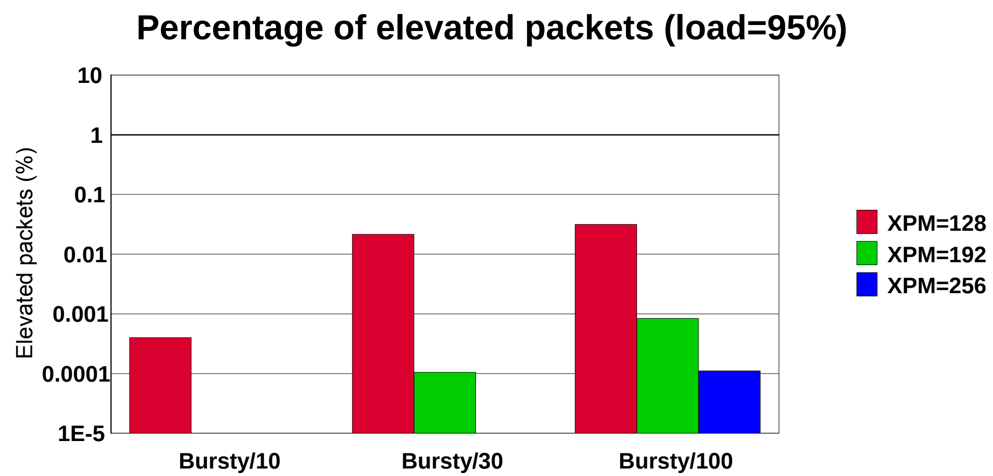
Reference implementation

* Input Adapter-initiated:

P-counters per buffer: reference

Simulation Results: Unfairness of Bulldozer





- Unfairness: any elevation is a statistical perturbation of the ordering rule imposed by the original priority set

- Baseline unfairness: ~ 1%
  - detrimental to P_H
  - neutral to P_{Med}
  - beneficial to P_L

- Optimisations
  - increase the XPM size 2x, 3x, 4x
  - increase the threshold spacing in IA

- Result
  - 2x buffer yields 10x improvement
  - Better delay/jitter performance
  - Significant savings in silicon area

Conclusions

- As a basic QoS scheduling discipline, strict priority must be supported
- Starvation still remains its best-known issue
- Priority-blocking needs attention because it impacts:
    - work-conservation
    - delay, jitter, and QoS in general

- Priority elevation is an appealing solution for lossless switches with:
    - limited buffering capacity
    - non-negligible RTTs
    - support for more than 2-3 prios
    - expected high burstiness
    - Its cons --latency & priority perturbation-- are strictly upper-bounded and practically reducible < ε
- Memory reduction @ ε < 0.01%
    - from $M = N^2 \times P \times RTT = 4K \times 8 \times 64 =$ 167 MB
    - to $M = N^2 \times 2 \times RTT = 4K \times 2 \times 64 =$ 42 MB

- *Acks: All the Prizma group in IBM ZRL has contributed to this work.*

For details see the Globecom 2003 paper .

Priority Inversion Reference

Memory Sharing Mechanism for Ordered Priority Set with Preemptive Service

- Our problem: How to share a limited resource, eg, memory M or reception buffer (RXB), among an ordered set of P priority classes served by a preemptive scheduler? The resource M is strictly sufficient for any single priority class. The sufficiency rule could be based on work-conservation, or any other constraint imposed by a specific application. Eg, in CIOQs with internal support for large RTTs, the highest prio currently available must acquire [within a bounded time interval - potential conflict with PIP] the full M=RTT of RXB. In most general case, the highest prio class must acquire the full resource ASAP.
  Obstacles to overcome - two classes of spatial *priority blocking*: (a) self-induced starvation; (b) push-thru, or chain, blocking (aka priority inversion). Both blocking classes are unbounded. Classical solution against spatial prio blocking is to dedicate an M-resource to each prio class.
  Here we propose a solution to spatial prio blocking by the means of prio elevation. Please note the subtle distinction from the temporal prio inversion and its PIP solution described below.
- ***Background:*** The related problem of temporal priority blocking (more specifically, inversion) is over two decades old in the field of real-time OS. Its classic solution in the temporal domain is known since '86 [1,2]; it became widely spread after the '97 Martian module failure. We acknowledge it as predecessor and cousin of our research on Bulldozer mechanisms.
- How does BDZ work? It needs an:
- Detection & Request unit, either in the upstream comm. device, or local (in switch-initiated BDZ);
- Activation unit, that grants or rejects the RQ from above;
- elevation mechanism
  - locally it must distinguish between elevated and native units
  - globally it is a strict / preemptive scheduler
- termination unit.
- * Priority inversion problem: http://www.netrino.com/Publications/Glossary/PriorityInversion.html
  * Martian Bug and its solution: http://www.kohala.com/start/papers.others/pathfinder.html
  http://catless.ncl.ac.uk/Risks/19.49.html
- Reference (aka, related "prior" art)
  The first mention of Priority Inheritance:
  1. L.Sha, R.Rajkumar and J.P.Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization", CMU-CS-87-181, ComputerScience Department, Carnegie-Mellon University, December 1987.
  2. (IEEE version) L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. In IEEE Transactions on Computers, vol. 39, pp. 1175-1185, Sep. 1990.

HOL_1-Blocking
Example in IBA

IBA has 16 VLs: Is this Sufficient?

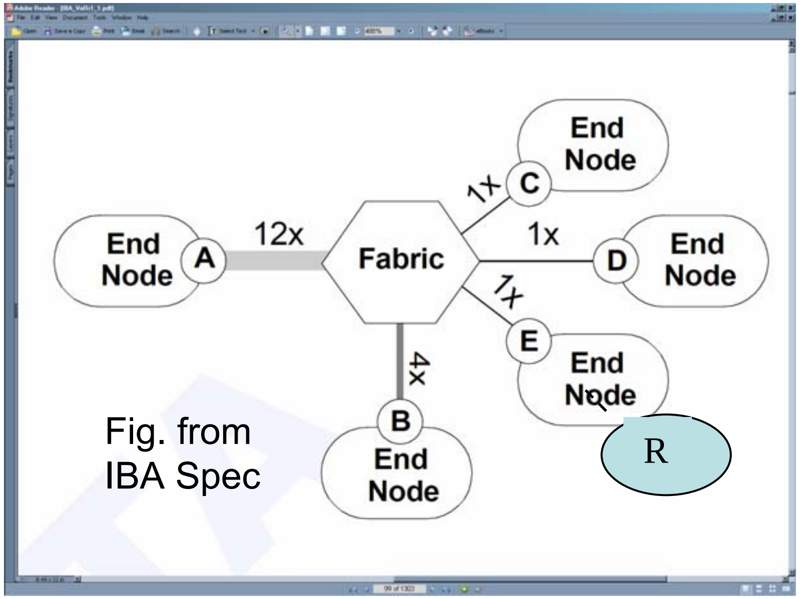
- IBA has 15 FC-ed VLs for QoS
  - SL-to-VL mapping is performed per hop, according to capabilities
- However, IBA doesn't have VOQ-selective LL-FC
  - "selective" = per switch (virtual) output port
- So what?
  - Hogging - aka buffer monopolization, HOL_1-blocking, output queue lockup, single-stage congestion, saturation $tree_{(k=0)}$
- How can we prove that *hogging really occurs* in IBA?
  - A. Back-of-the-envelope reasoning
  - B. Analytical modeling of stability and work-conservation (papers available)
  - C. Comparative simulations: IBA, PCI-AS etc. (next slides)

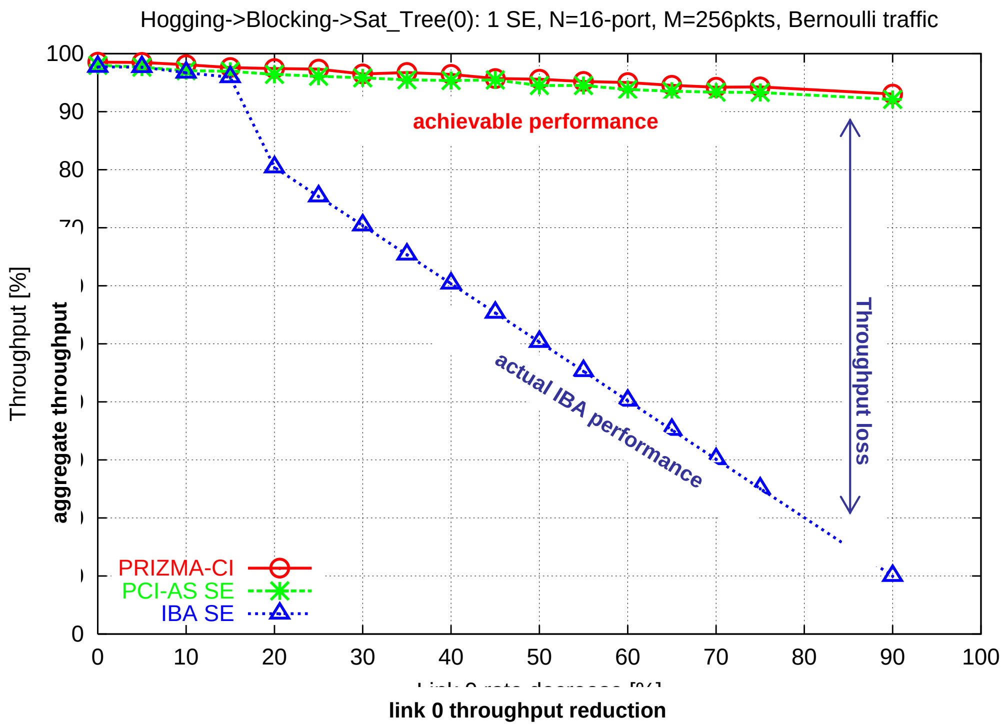
IBA SE Hogging Scenario

- Simulation: parallel backup to a RAID across an IBA switch
  - TX / SRC
    - 16 independent IBA sources, e.g. 16 "producer" CPU/threads
    - SRC behavior: greedy, using any communication model (UD)
    - SL: BE service discipline on a single VL
      - (the other VLs suffer of their own ☺)
  - Fabrics (single stage)
    - 16x16 IBA generic SE
    - 16x16 PCI-AS switch
    - 16x16 Prizma CI switch
  - RX / DST
    - 16 HDD "consumers"
    - t_0 : initially each HDD sinks data at full 1x (100%)
    - t_{sim} : during simulation HDD[0] enters thermal recalibration or sector remapping; consequently
      - » HDD[0] progressively slows down its incoming link throughput: 90, 80,..., 10%

First: Friendly Bernoulli Traffic

- 2 Sources (A, B) sending @ (12x + 4x) to 16*1x End Nodes (C..R)





Myths and Fallacies about HOL_1-blocking

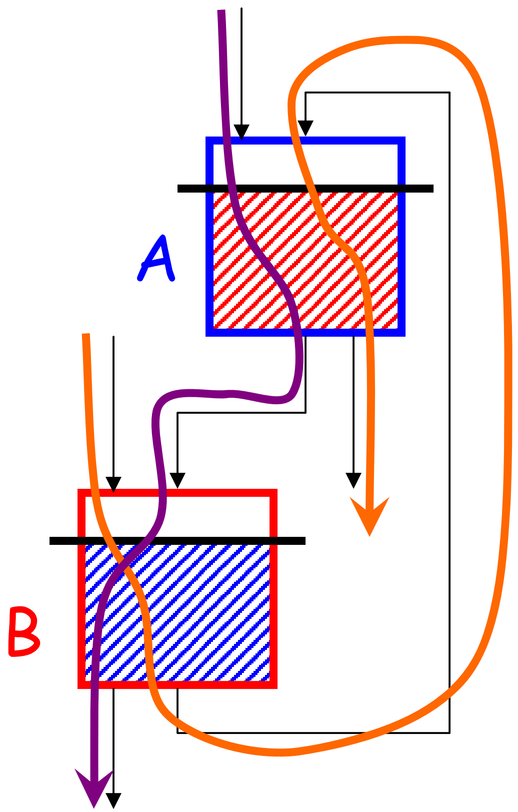
- Isn't IBA's static rate control sufficient?
- No, because it is STATIC

- IBA's VLs are sufficient...?!
- No.
  - VLs and ports are orthogonal dimensions of LL-FC
    - 1. VLs are for SL and QoS => VLs are assigned to prios, not ports!
    - 2. Max. no. of VLs = 15 << max (SE_degree x SL) = 4K

- Can the SE buffer partitioning solve hogging in 1-hop systems?
- No.
  - 1. Partitioning makes sense <u>only</u> w/ Status-based FC (per bridge output port - see PCIe/AS SBFC);
    - IBA doesn't have a native Status-based FC
  - 2. Sizing becomes the *issue* => we need dedication per I and O ports
    - $M = O(SL * \max\{RTT, MTU\} * N^2)$ very large number!
    - Papers (available) and theoretical disertations prove stability and work-conservation, but the amounts of required M are large

M2M Circular Dependency Deadlocks

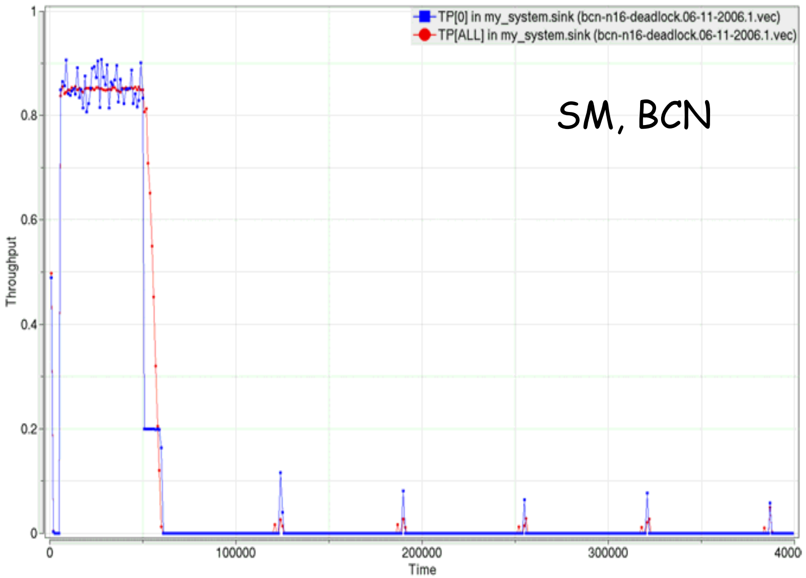
The Mechanism of LL-FC-induced Deadlocks

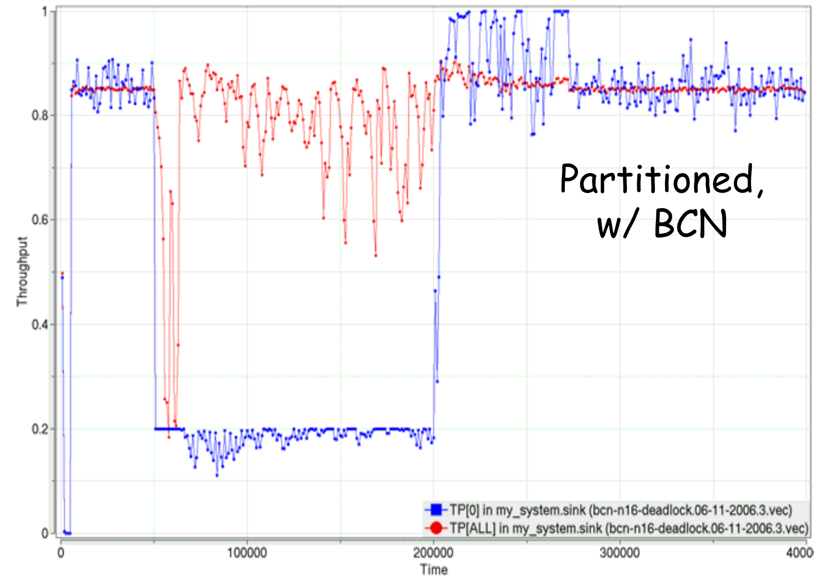


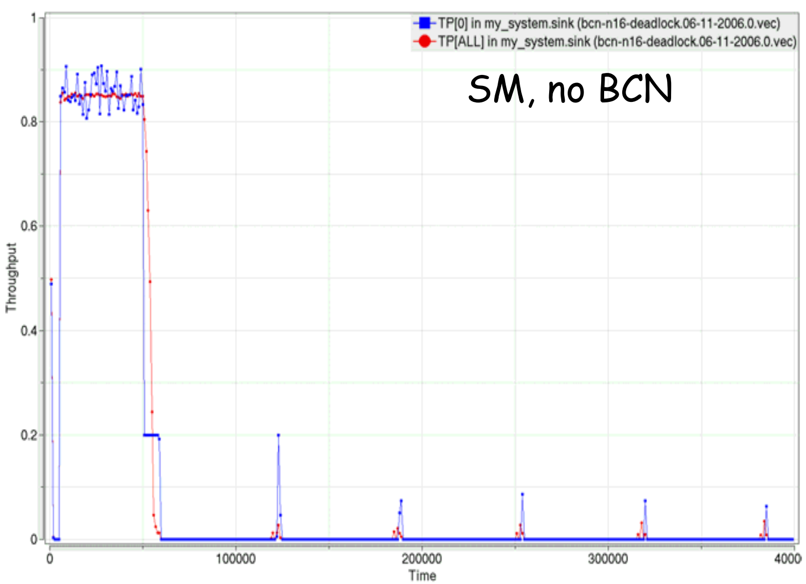
- When incorrectly implemented, LL-FC-based flow control can cause hogging and deadlocks

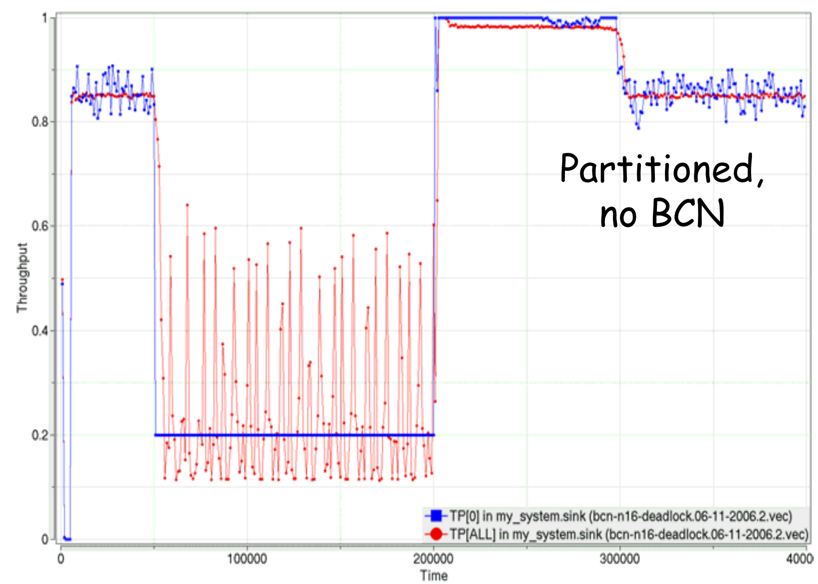
- LL-FC-deadlocking in shared-memory switches:
  - Switches A and B are both full (within the granularity of an MTU or Jumbo) => LL-FC thresholds exceeded
    - All traffic from A is destined to B and viceversa
  - Neither can send, waiting on each other indefinitely: Deadlock.
  - Note: Traffic from A never takes the path from B back to A and vice versa
    - Due to shortest-path routing

LL-FC-caused Deadlocks in BCN Simulations 16-node 5-stage fabric Bernoulli traffic

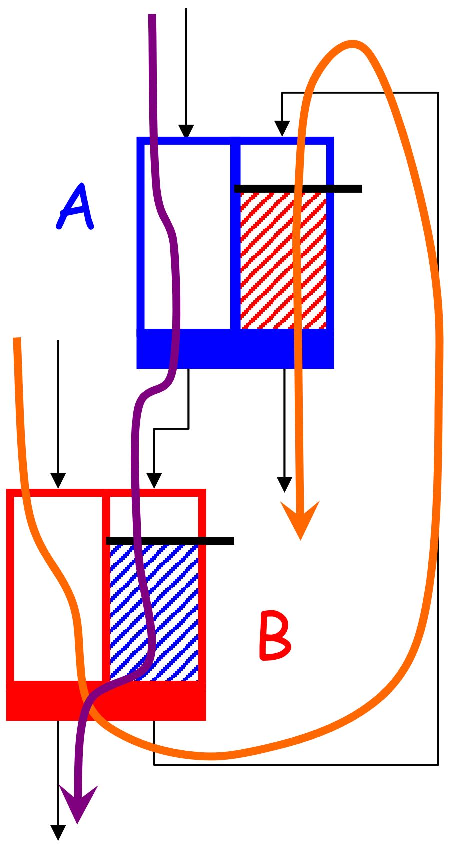








Typical Solution to Defeat this Deadlock: Partitioning

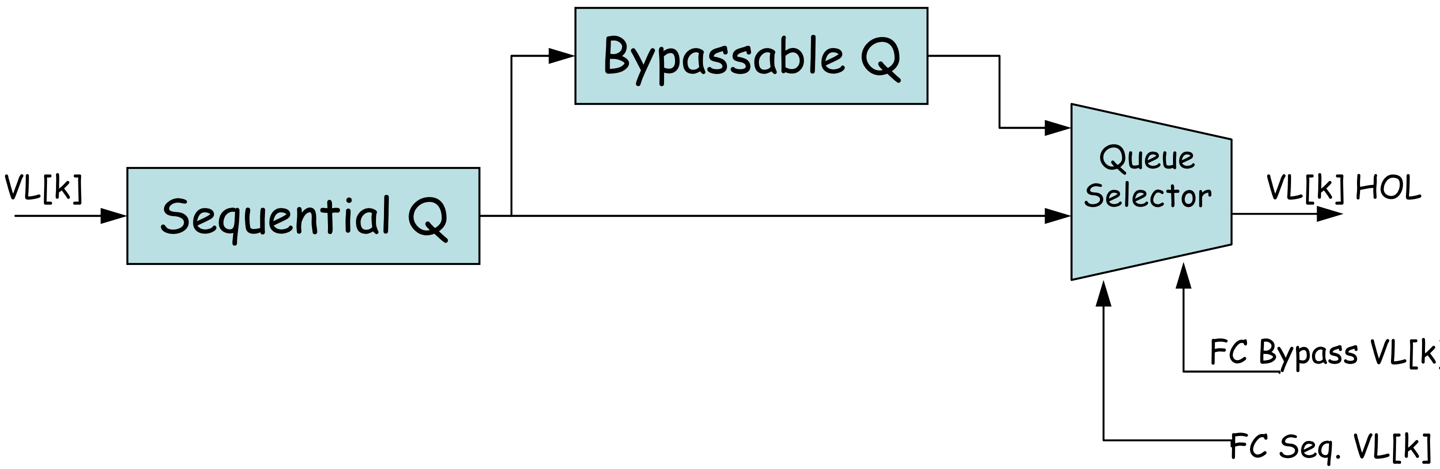


- Architectural: Assert LL-FC on a per-input basis
  - No input is allowed to consume more than 1/N-th of the shared memory
  - All traffic in B's input buffer for A is guaranteed to be destined to a different port than the one leading back to A (and vice versa)
  - Hence, the circular dependence has been broken!

- Confirmed by simulations
  - Assert LL-FC on input i:
    - $occ_{mem} >= T_h$ *or* $occ[i] >= T_h/N$
  - Deassert LL-FC on input i:
    - $occ_{mem} < T_h$ *and* $occ[i] < T_l/N$
  - $Q_{eq} = M / (2N)$

... this deadlock is solved!

Breaking the Rq/Reply Deadlock: Bypass Queue



Ordering rules (PCIe-compatible)

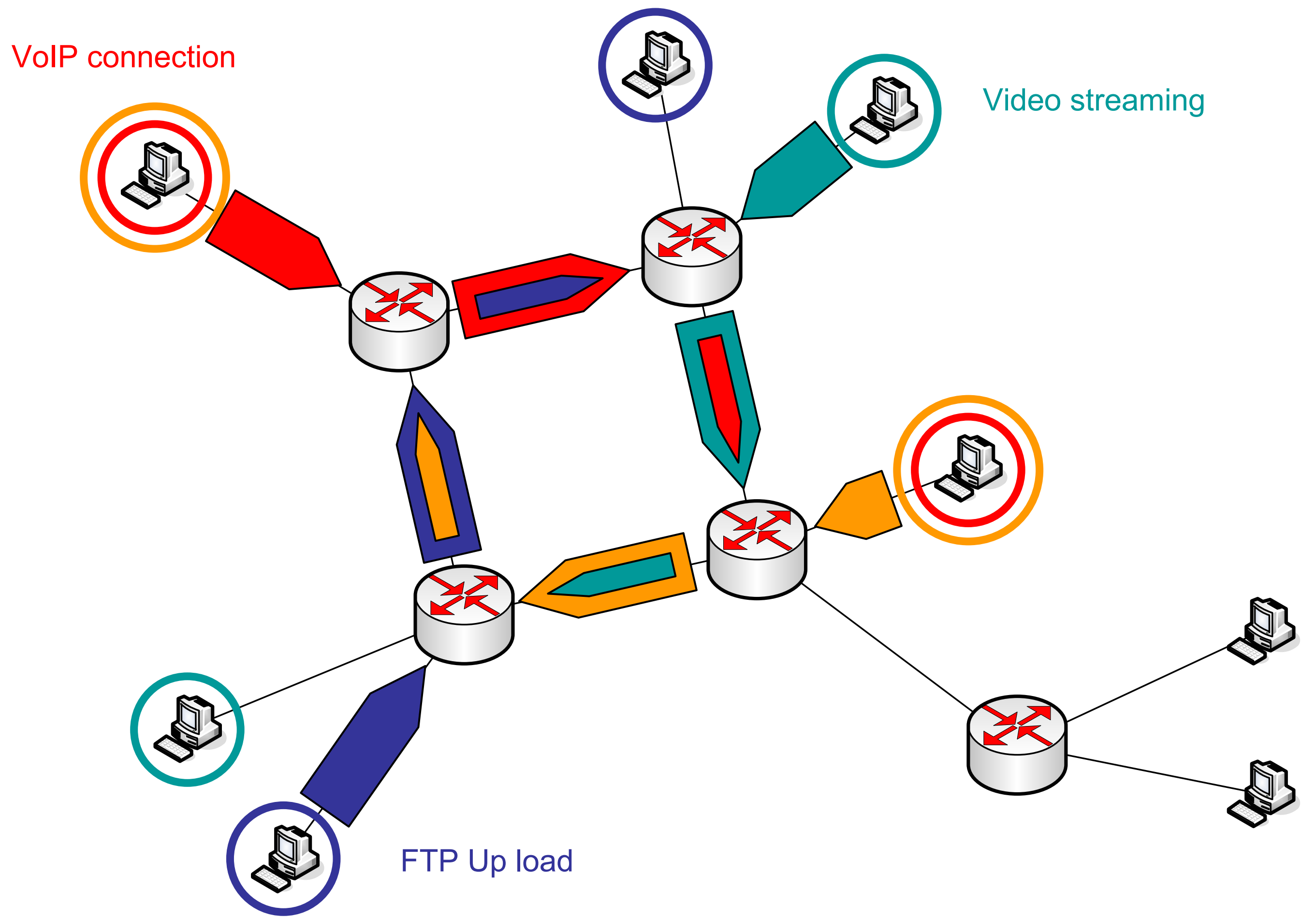
1. Seq. Q is FIFO => maintains VL[k] default ordering.
2. Bypass Q is FIFO => local ordering.
3. HOL of Seq. Q is served ahead of the Bypass Q, if the latter is LL-FC blocked => deadlock avoidance.
4. HOL of Bypass Q can not be served as long as an older pkt. exists in the Seq. Q => inter-queue ordering.

... this is the kernel of ordering rules.

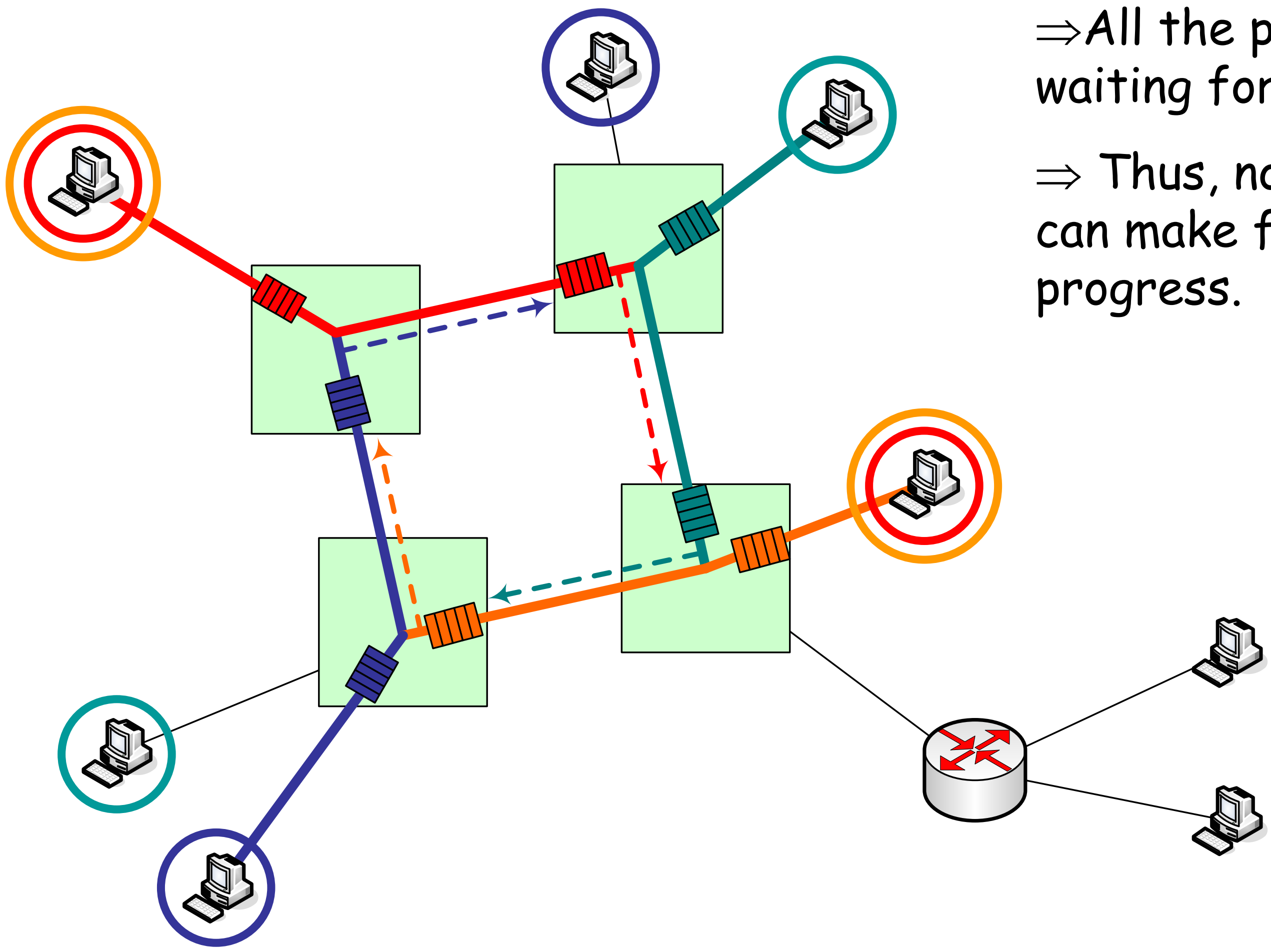
Routing Deadlocks

Routing Deadlock Scenario

Def.: Cyclic dependency relationship between two or more resources that are waiting on each other to free resources, but without freeing their own. Resources: physical (hardware) or logical (software)



Deadlocked Buffers: Dependency Loop in the Routing Graph

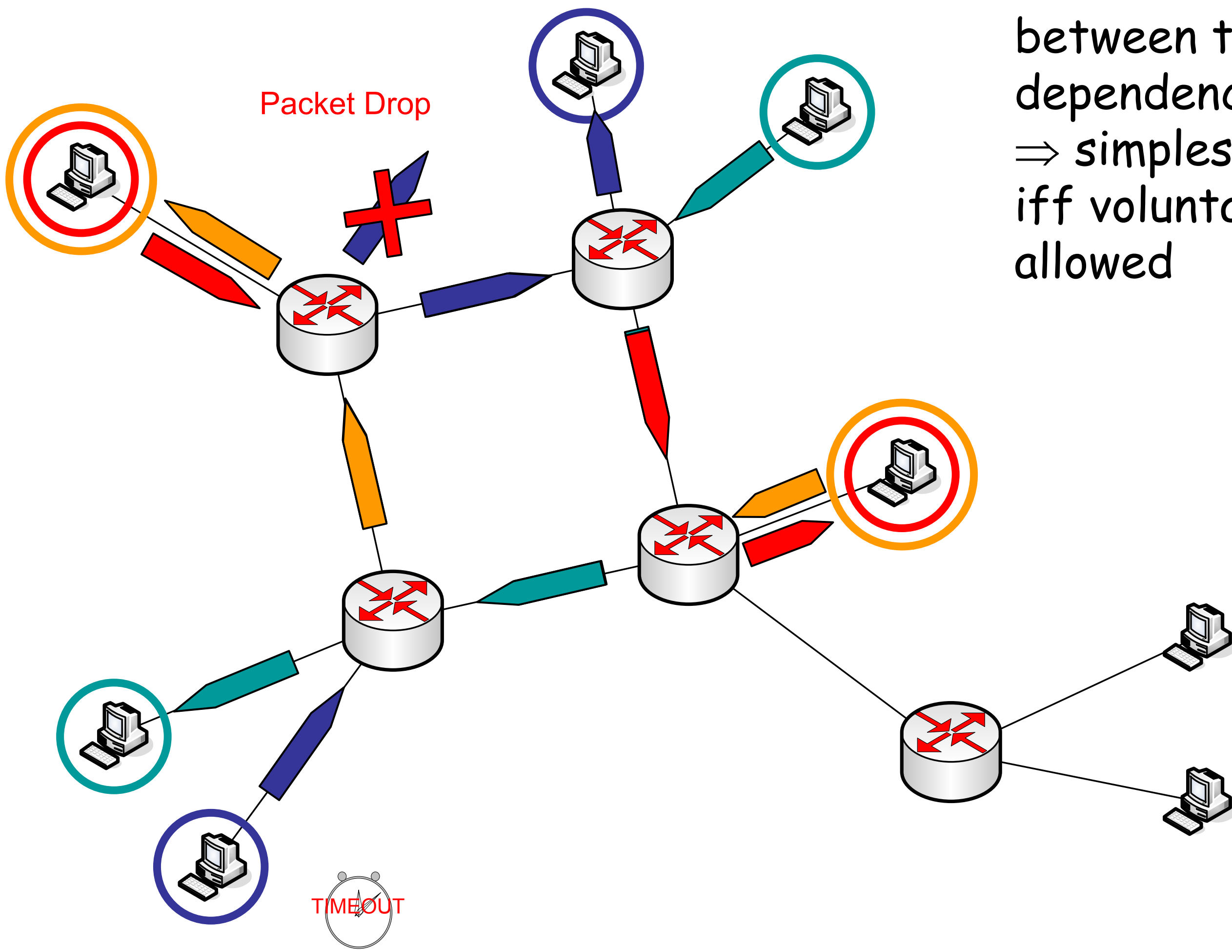


All buffers in this network cycle are full

⇒All the packets are waiting for each other

⇒ Thus, no message can make forward progress.

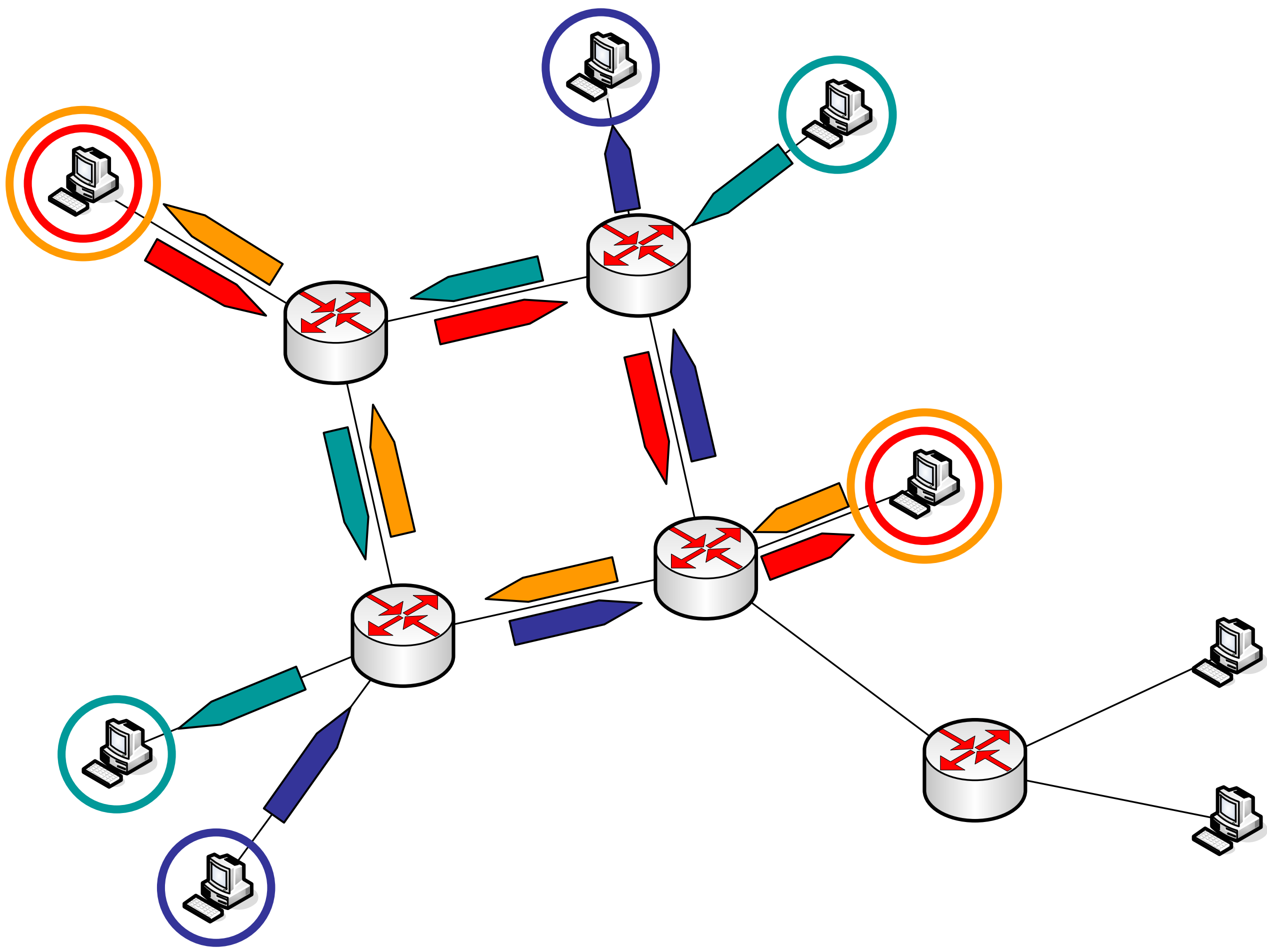
Deadlock Recovery in Lossy Networks



Packet **drop**
=> frees deadlocked resources
⇒ eliminates cycles between their inter-dependencies.
⇒ simplest solution, iff voluntary loss is allowed

Deadlock Avoidance by Ordering: Deadlock-free Routing

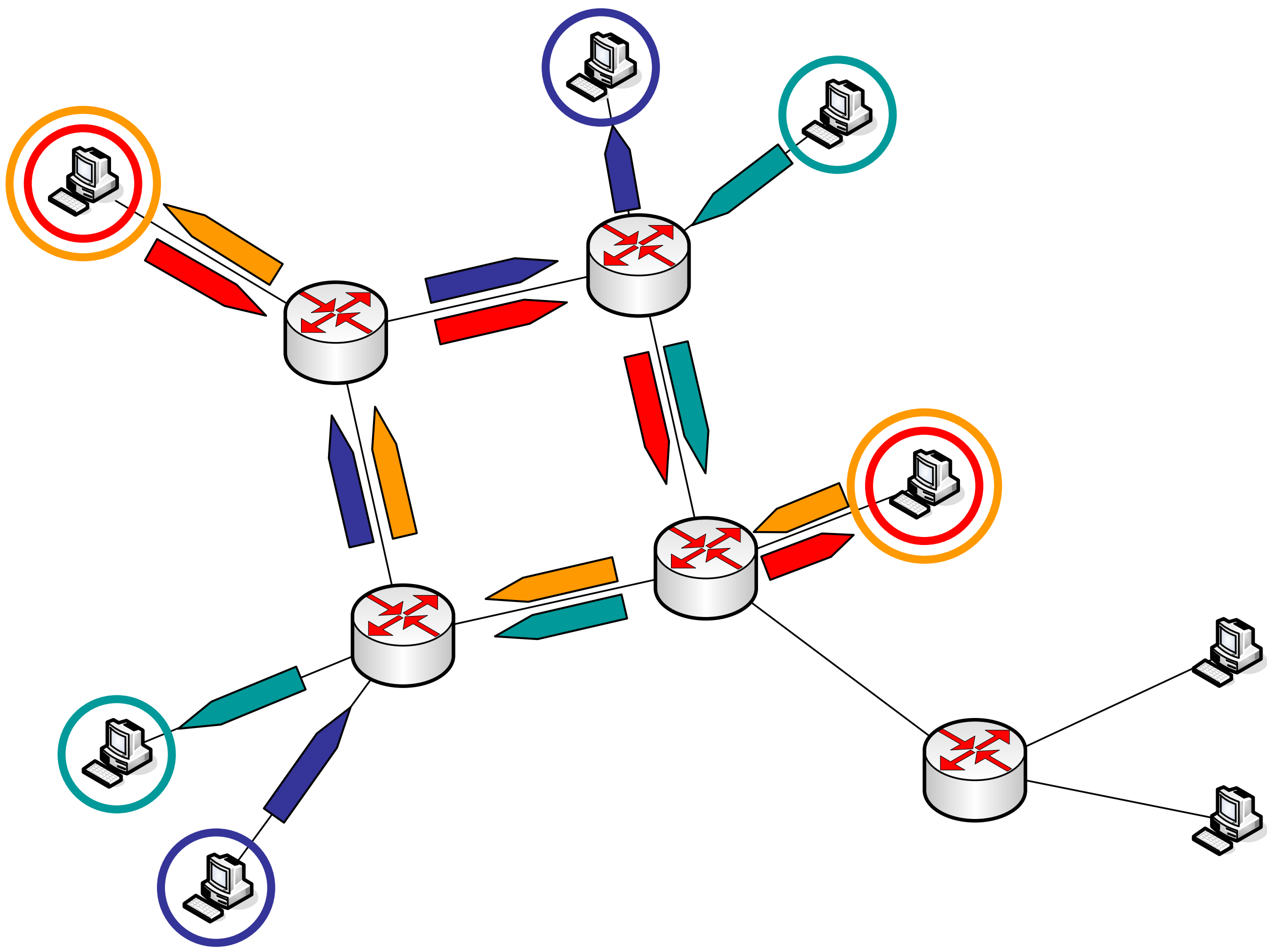
Deadlock-free algorithm => Certain turns will be forbidden in order to eliminate cycles. In figure below left-up and right-down turns are prohibited.



Deadlock Avoidance or Recovery: Virtual Channels

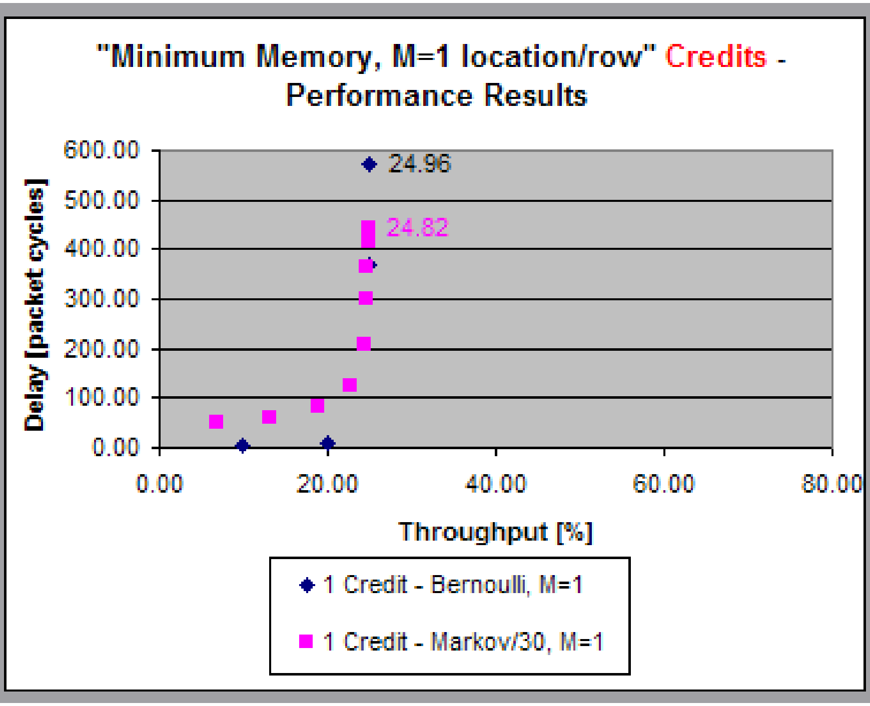
1. Split physical links into several VCs
2. Define the restrictions / ordering rules in the use of VCs to avoid / recover from deadlocks.

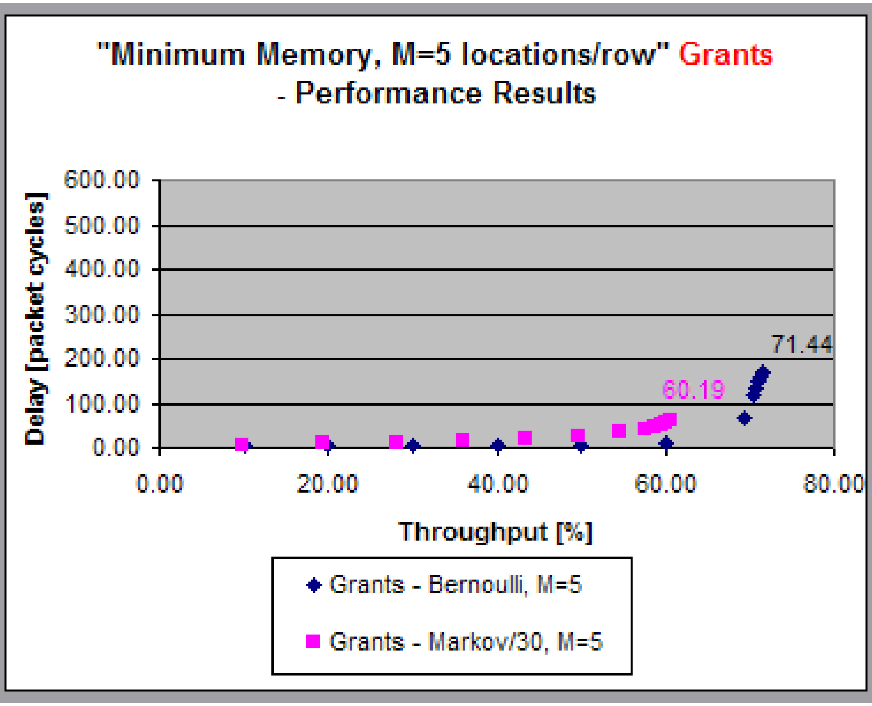
=> Enables fully or partially adaptive routing.



RTT-Sensitivity

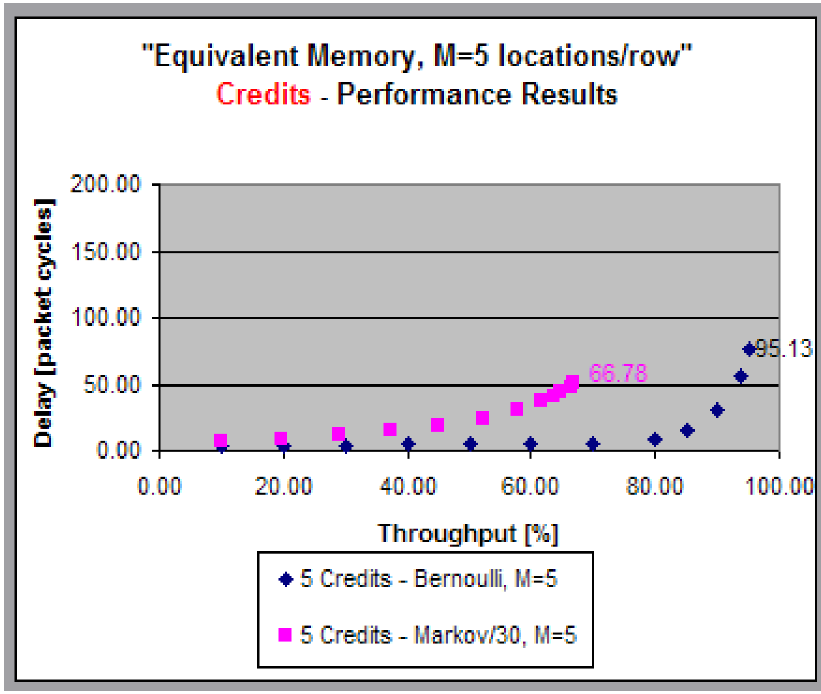
Correctness: Min. Memory for "No Loss"

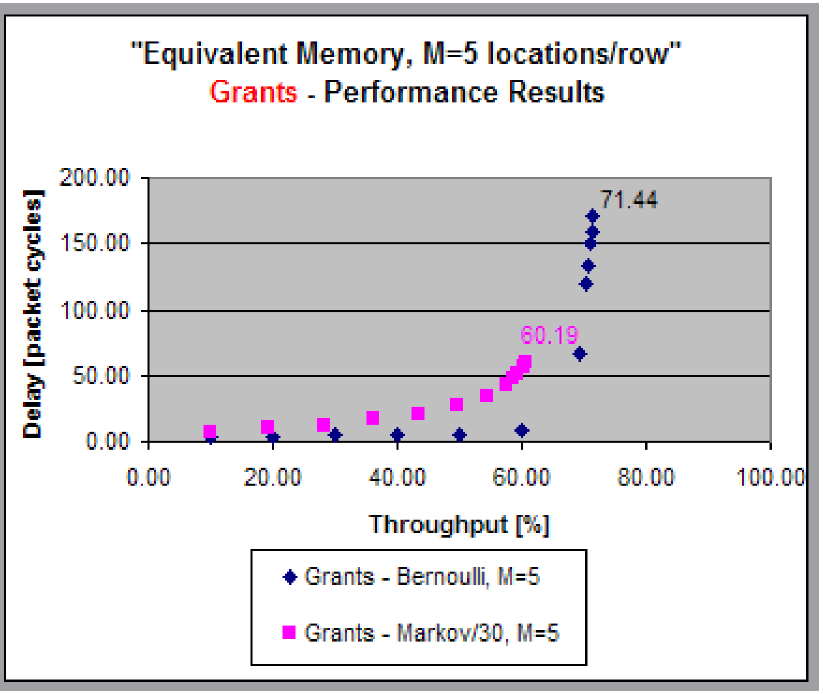




- "Minimum": to operate lossless => $O(RTT_{link})$
  - Credit : 1 credit = 1 memory location
  - Grant : 5 (=RTT+1) memory locations
- Credits
  - Under full load the single credit is constantly looping between RX and TX RTT=4 => max. performance = f(up-link utilisation) = 25%
- Grants
  - Determined by slow restart: if last packet has left the RX queue, it takes an RTT until the next packet arrives

Performance of Credit vs. Grant @ M = RTT+1





- "Equivalent" = 'fair' comparison
  1. Credit scheme: 5 credit = 5 memory locations
  2. Grant scheme: 5 (=RTT+1) memory locations

Performance loss for LL-FC/Grants is due to lack of underflow protection, because if M < 2*RTT the link is not work-conserving (pipeline bubbles on restart)

For equivalent (to credit) performance, M=9 is required for LL-FC...
...however, this is not an endorsement of any specific scheme!