

18-742 Fall 2012

# Parallel Computer Architecture

## Lecture 20: Speculation+Interconnects III

Prof. Onur Mutlu

Carnegie Mellon University

10/24/2012

# New Review Assignments

---

- **Due: Sunday, October 28, 11:59pm.**
- Das et al., “Aergia: Exploiting Packet Latency Slack in On-Chip Networks,” ISCA 2010.
- Dennis and Misunas, “A Preliminary Architecture for a Basic Data Flow Processor,” ISCA 1974.
  
- **Due: Tuesday, October 30, 11:59pm.**
- Arvind and Nikhil, “Executing a Program on the MIT Tagged-Token Dataflow Architecture,” IEEE TC 1990.

# Due in the Future

---

## ■ Dataflow

- Gurd et al., “The Manchester prototype dataflow computer,” CACM 1985.
- Lee and Hurson, “Dataflow Architectures and Multithreading,” IEEE Computer 1994.

## ■ Restricted Dataflow

- Patt et al., “HPS, a new microarchitecture: rationale and introduction,” MICRO 1985.
- Patt et al., “Critical issues regarding HPS, a high performance microarchitecture,” MICRO 1985.

# Project Milestone I Presentations (I)

---

- When: October 26, in class
- Format: 9-min presentation per group, 2-min Q&A, 1-min grace period
- What to present:
  - The problem you are solving + your goal
  - Your solution ideas + strengths and weaknesses
  - **Your methodology to test your ideas**
  - **Concrete mechanisms you have implemented so far**
  - **Concrete results you have so far**
  - **What will you do next?**
  - What hypotheses you have for future?
  - How close were you to your target?

# Project Milestone I Presentations (I)

---

- You can update your slides
- Send them to me and Han by 2:30pm on Oct 26, Friday
  
- Make a lot of progress and find breakthroughs
  
- Example milestone presentations:
  - <http://www.ece.cmu.edu/~ece742/2011spring/doku.php?id=project>
  - [http://www.ece.cmu.edu/~ece742/2011spring/lib/exe/fetch.php?media=milestone1\\_ausavarungnirun\\_meza\\_yoon.pptx](http://www.ece.cmu.edu/~ece742/2011spring/lib/exe/fetch.php?media=milestone1_ausavarungnirun_meza_yoon.pptx)
  - [http://www.ece.cmu.edu/~ece742/2011spring/lib/exe/fetch.php?media=milestone1\\_tumanov\\_lin.pdf](http://www.ece.cmu.edu/~ece742/2011spring/lib/exe/fetch.php?media=milestone1_tumanov_lin.pdf)

# Last Few Lectures

---

- Speculation in Parallel Machines
- Interconnection Networks
- Guest Lecture: Adam From, ARM

# Today

---

- Transactional Memory (brief)
- Interconnect wrap-up

# Review: Speculation to Improve Parallel Programs

---

- Goal: reduce the impact of serializing bottlenecks
  - Improve performance
  - Improve programming ease
- Examples
  - Herlihy and Moss, "Transactional Memory: Architectural Support for Lock-Free Data Structures," ISCA 1993.
  - Rajwar and Goodman, "Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution," MICRO 2001.
  - Martinez and Torrellas, "Speculative Synchronization: Applying Thread-Level Speculation to Explicitly Parallel Applications," ASPLOS 2002.
  - Rajwar and Goodman, "Transactional lock-free execution of lock-based programs," ASPLOS 2002.



# Review: Speculative Lock Elision

---

- Many programs use locks for synchronization
- Many locks are not necessary
  - Stores occur infrequently during execution
  - Updates can occur to disjoint parts of the data structure
- Idea:
  - Speculatively assume lock is not necessary and execute critical section without acquiring the lock
  - Check for conflicts within the critical section
  - Roll back if assumption is incorrect
- Rajwar and Goodman, “[Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution](#),” MICRO 2001.

# Review: Dynamically Unnecessary Synchronization

---

```
a)      1. LOCK(locks->error_lock)
          2. if (local_error > multi->err_multi)
          3.     multi->err_multi = local_err;
          4. UNLOCK(locks->error_lock)

b)      Thread 1                Thread 2

LOCK(hash_tbl.lock)
var = hash_tbl.lookup(X)
if (!var)
    hash_tbl.add(X);
UNLOCK(hash_tbl.lock)

                                LOCK(hash_tbl.lock)
                                var = hash_tbl.lookup(Y)
                                if (!var)
                                    hash_tbl->add(Y);
                                UNLOCK(hash_tbl.lock)
```

**Figure 1.** *Two examples of potential parallelism masked by dynamically unnecessary synchronization.*

# Transactional Memory

# Transactional Memory

---

- Idea: Programmer specifies code to be executed atomically as transactions. Hardware/software guarantees atomicity for transactions.
- Motivated by difficulty of lock-based programming
- Motivated by lack of concurrency (performance issues) in blocking synchronization (or “pessimistic concurrency”)

# Locking Issues

---

- Locks: objects only one thread can hold at a time
  - Organization: lock for each shared structure
  - Usage: (block) → acquire → access → release
  
- Correctness issues
  - Under-locking → data races
  - Acquires in different orders → deadlock
  
- Performance issues
  - Conservative serialization
  - Overhead of acquiring
  - Difficult to find right granularity
  - Blocking

# Locks vs. Transactions

---

## Lock issues:

- Under-locking → data races
- Deadlock due to lock ordering
- Blocking synchronization
- Conservative serialization

## How transactions help:

- + Simpler interface/reasoning
- + No ordering
- + Nonblocking (Abort on conflict)
- + Serialization only on conflicts

- Locks → pessimistic concurrency
- Transactions → optimistic concurrency

# Transactional Memory

---

- Transactional Memory (TM) allows arbitrary multiple memory locations to be updated atomically (all or none)
- Basic Mechanisms:
  - **Isolation and conflict management**: Track read/writes per transaction, detect when a conflict occurs between transactions
  - **Version management**: Record new/old values (where?)
  - **Atomicity**: Commit new values or abort back to old values → all or none semantics of a transaction
- Issues the same as other speculative parallelization schemes
  - Logging/buffering
  - Conflict detection
  - Abort/rollback
  - Commit

# Four Issues in Transactional Memory

---

- How to deal with unavailable values: predict vs. wait
- How to deal with speculative updates: logging/buffering
- How to detect conflicts: lazy vs. eager
- How and when to abort/rollback or commit



# Many Variations of TM

---

- Software
  - High performance overhead, but no virtualization issues
- Hardware
  - What if buffering is not enough?
  - Context switches, I/O within transactions?
  - Need support for virtualization
- Hybrid HW/SW
  - Switch to SW to handle large transactions and buffer overflows

# Initial TM Ideas

---

- **Load Linked Store Conditional Operations**
  - Lock-free atomic update of a single cache line
  - Used to implement non-blocking synchronization
    - Alpha, MIPS, ARM, PowerPC
  - Load-linked returns current value of a location
  - A subsequent store-conditional to the same memory location will store a new value only if no updates have occurred to the location
  
- **Herlihy and Moss, ISCA 1993**
  - Instructions explicitly identify transactional loads and stores
  - Used dedicated transaction cache
  - Size of transactions limited to transaction cache

# Herlihy and Moss, ISCA 1993

---

Our transactions are intended to replace short critical sections. For example, a lock-free data structure would typically be implemented in the following stylized way (see Section 5 for specific examples). Instead of acquiring a lock, executing the critical section, and releasing the lock, a process would:

1. use `LT` or `LTX` to read from a set of locations,
2. use `VALIDATE` to check that the values read are consistent,
3. use `ST` to modify a set of locations, and
4. use `COMMIT` to make the changes permanent. If either the `VALIDATE` or the `COMMIT` fails, the process returns to Step (1).

# Current Implementations of TM/SLE

---

- Sun ROCK
  - Dice et al., “Early Experience with a Commercial Hardware Transactional Memory Implementation,” ASPLOS 2009.
- IBM Blue Gene
  - Wang et al., “Evaluation of Blue Gene/Q Hardware Support for Transactional Memories,” PACT 2012.
- IBM System z: Two types of transactions
  - Best effort transactions: Programmer responsible for aborts
  - Guaranteed transactions are subject to many limitations
- Intel Haswell

# Some TM Research Issues

---

- How to virtualize transactions (without much complexity)
  - Ensure long transactions execute correctly
  - In the presence of context switches, paging
- Handling I/O within transactions
  - No problem with locks
- Semantics of nested transactions (more of a language/programming research topic)
- Does TM increase programmer productivity?
  - Does the programmer need to optimize transactions?

# Interconnects III: Review and Wrap-Up

# Last Lectures

---

- Interconnection Networks
  - Introduction & Terminology
  - Topology
  - Buffering and Flow control
  - Routing
  - Router design
  - Network performance metrics
  - On-chip vs. off-chip differences

# Some Questions

---

- What are the possible ways of handling contention in a router?
- What is head-of-line blocking?
- What is a non-minimal routing algorithm?
- What is the difference between deterministic, oblivious, and adaptive routing algorithms?
- What routing algorithms need to worry about deadlock?
- What routing algorithms need to worry about livelock?
- How to handle deadlock?
- How to handle livelock?
- What is zero-load latency?
- What is saturation throughput?
- What is an application-aware packet scheduling algorithm?



# Routing Mechanism

---

## ■ Arithmetic

- Simple arithmetic to determine route in regular topologies
- Dimension order routing in meshes/tori

## ■ Source Based

- Source specifies output port for each switch in route
- + Simple switches
  - no control state: strip output port off header
- Large header

## ■ Table Lookup Based

- Index into table for output port
- + Small header
- More complex switches

# Routing Algorithm

---

- Types
  - **Deterministic:** always choose the same path
  - **Oblivious:** do not consider network state (e.g., random)
  - **Adaptive:** adapt to state of the network
  
- How to adapt
  - Local/global feedback
  - Minimal or non-minimal paths

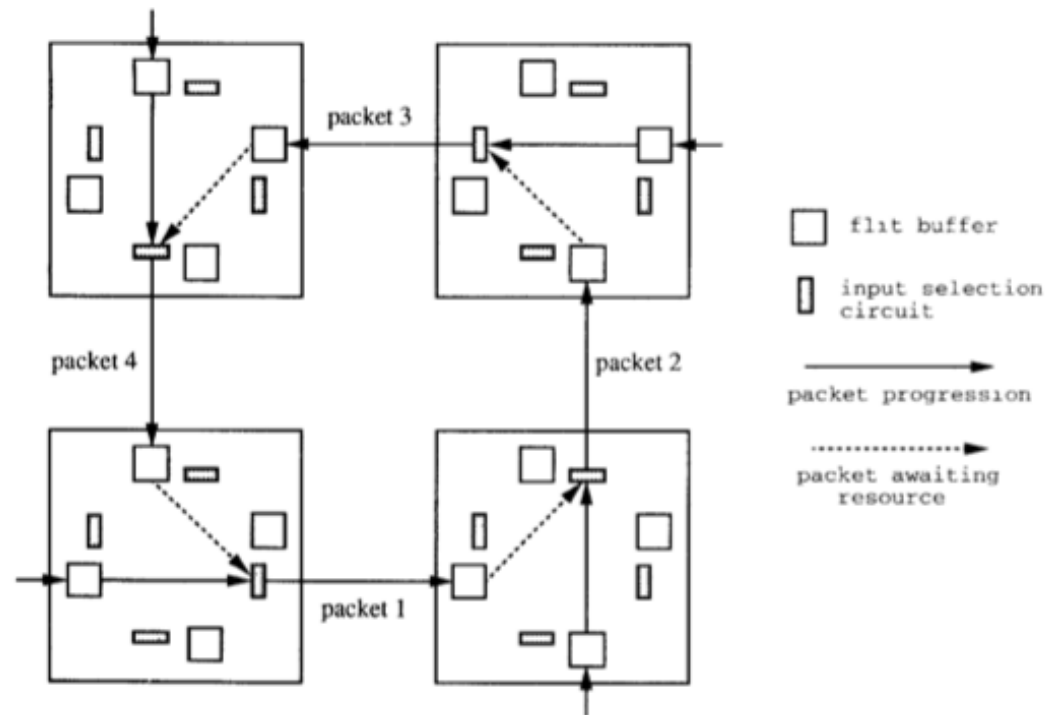
# Deterministic Routing

---

- All packets between the same (source, dest) pair take the same path
  - Dimension-order routing
    - E.g., XY routing (used in Cray T3D, and many on-chip networks)
    - First traverse dimension X, then traverse dimension Y
- + Simple
- + Deadlock freedom (no cycles in resource allocation)
- Could lead to high contention
- Does not exploit path diversity

# Deadlock

- No forward progress
- Caused by circular dependencies on resources
- Each packet waits for a buffer occupied by another packet downstream



# Handling Deadlock

---

- Avoid cycles in routing
  - Dimension order routing
    - Cannot build a circular dependency
  - Restrict the “turns” each packet can take
  
- Avoid deadlock by adding virtual channels
  
  
  
  
  
  
  
  
  
  
- Detect and break deadlock
  - Preemption of buffers

# Turn Model to Avoid Deadlock

## ■ Idea

- Analyze directions in which packets can turn in the network
- Determine the cycles that such turns can form
- Prohibit just enough turns to break possible cycles

- Glass and Ni, “[The Turn Model for Adaptive Routing](#),” ISCA 1992.

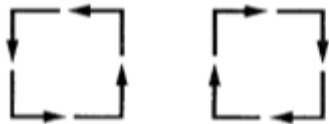


FIG. 2. The possible turns and simple cycles in a two-dimensional mesh.



FIG. 3. The four turns allowed by the  $xy$  routing algorithm.

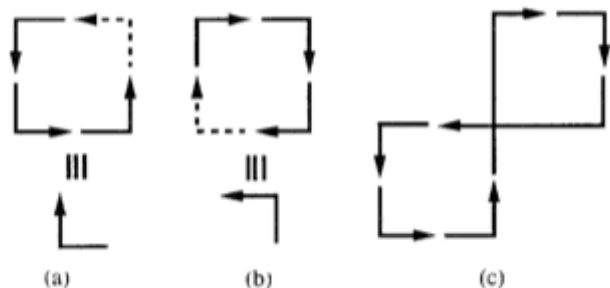


FIG. 4. Six turns that complete the cycles and allow deadlock.

# Valiant's Algorithm

---

- An example of oblivious algorithm
  - Goal: Balance network load
  - Idea: Randomly choose an intermediate destination, route to it first, then route from there to destination
    - Between source-intermediate and intermediate-dest, can use dimension order routing
- + Randomizes/balances network load
- Non minimal (packet latency can increase)
- Optimizations:
    - Do this on high load
    - Restrict the intermediate node to be close (in the same quadrant)

# Adaptive Routing

---

## ■ Minimal adaptive

- Router uses network state (e.g., downstream buffer occupancy) to pick which “productive” output port to send a packet to
  - Productive output port: port that gets the packet closer to its destination
- + Aware of local congestion
- Minimality restricts achievable link utilization (load balance)

## ■ Non-minimal (fully) adaptive

- “Misroute” packets to non-productive output ports based on network state
- + Can achieve better network utilization and load balance
- Need to guarantee livelock freedom



# More on Adaptive Routing

---

- Can avoid faulty links/routers
  - Idea: **Route around faults**
- + Deterministic routing cannot handle faulty components
- Need to change the routing table to disable faulty routes
    - Assuming the faulty link/router is detected

# Real On-Chip Network Designs

---

- Tileria Tile64 and Tile100
- Larrabee
- Cell

# On-Chip vs. Off-Chip Differences

---

## Advantages of on-chip

- **Wires are “free”**
  - Can build highly connected networks with wide buses
- **Low latency**
  - Can cross entire network in few clock cycles
- **High Reliability**
  - Packets are not dropped and links rarely fail

## Disadvantages of on-chip

- **Sharing resources with rest of components on chip**
  - Area
  - Power
- **Limited buffering available**
- **Not all topologies map well to 2D plane**

# Tilera Networks

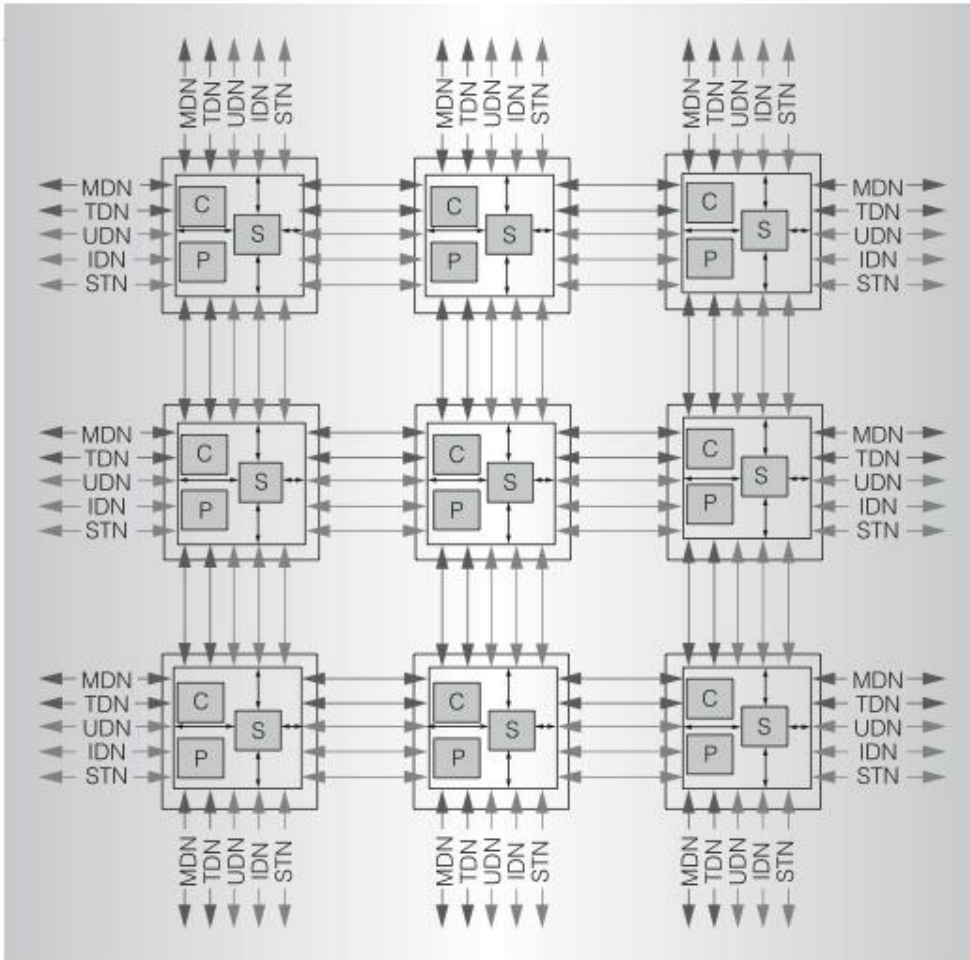


Figure 3. A 3 × 3 array of tiles connected by networks. (MDN: memory dynamic network; TDN: tile dynamic network; UDN: user dynamic network; IDN: I/O dynamic network; STN: static network.)

- 2D Mesh
- Five networks
- Four packet switched
  - Dimension order routing, wormhole flow control
  - TDN: Cache request packets
  - MDN: Response packets
  - IDN: I/O packets
  - UDN: Core to core messaging
- One circuit switched
  - STN: Low-latency, high-bandwidth static network
  - Streaming data

We did not cover the following slides in lecture.  
These are for your preparation for the next lecture.

# Research Topics in Interconnects

---

- Plenty of topics in on-chip networks. Examples:
- **Energy/power** efficient/proportional design
- **Reducing Complexity**: Simplified router and protocol designs
- **Adaptivity**: Ability to adapt to different access patterns
- **QoS and performance isolation**
  - Reducing and controlling interference, admission control
- **Co-design of NoCs with other shared resources**
  - End-to-end performance, QoS, power/energy optimization
- **Scalable topologies** to many cores
- Fault tolerance
- Request prioritization, priority inversion, coherence, ...
- New technologies (optical, 3D)

# Packet Scheduling

---

- Which packet to choose for a given output port?
  - ❑ Router needs to prioritize between competing flits
  - ❑ Which input port?
  - ❑ Which virtual channel?
  - ❑ Which application's packet?
- Common strategies
  - ❑ Round robin across virtual channels
  - ❑ Oldest packet first (or an approximation)
  - ❑ Prioritize some virtual channels over others
- Better policies in a multi-core environment
  - ❑ Use application characteristics

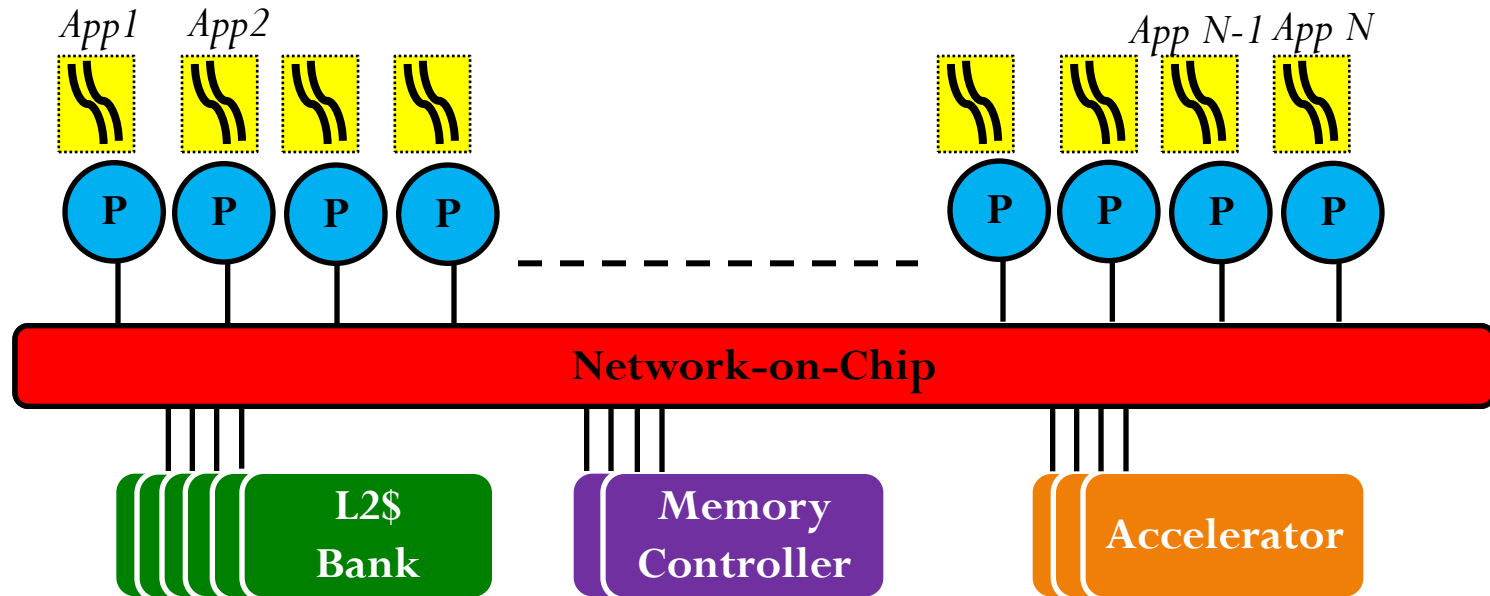
# Application-Aware Packet Scheduling

Das et al., “[Application-Aware Prioritization Mechanisms for On-Chip Networks](#),” MICRO 2009.



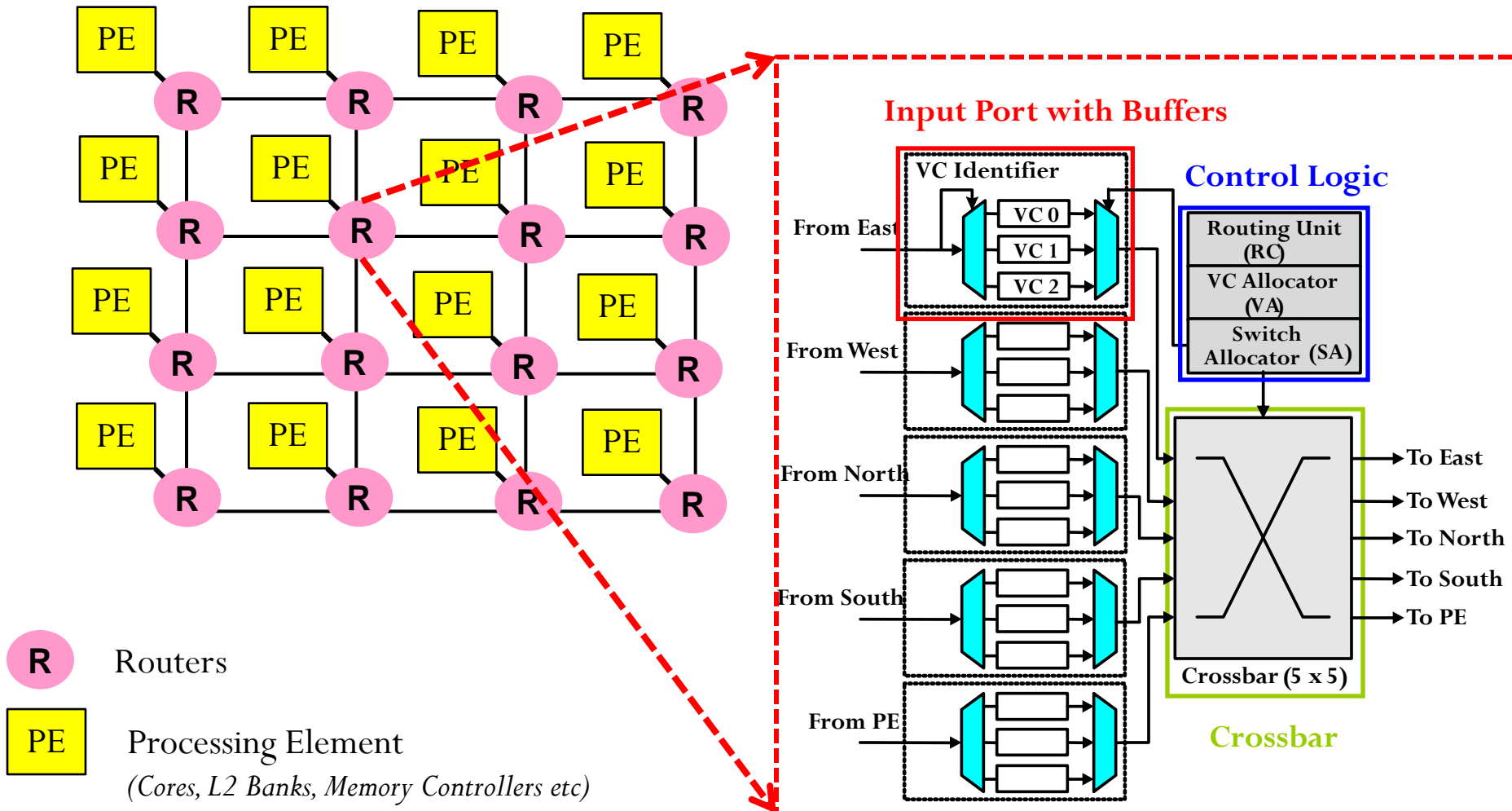
# The Problem: Packet Scheduling

---

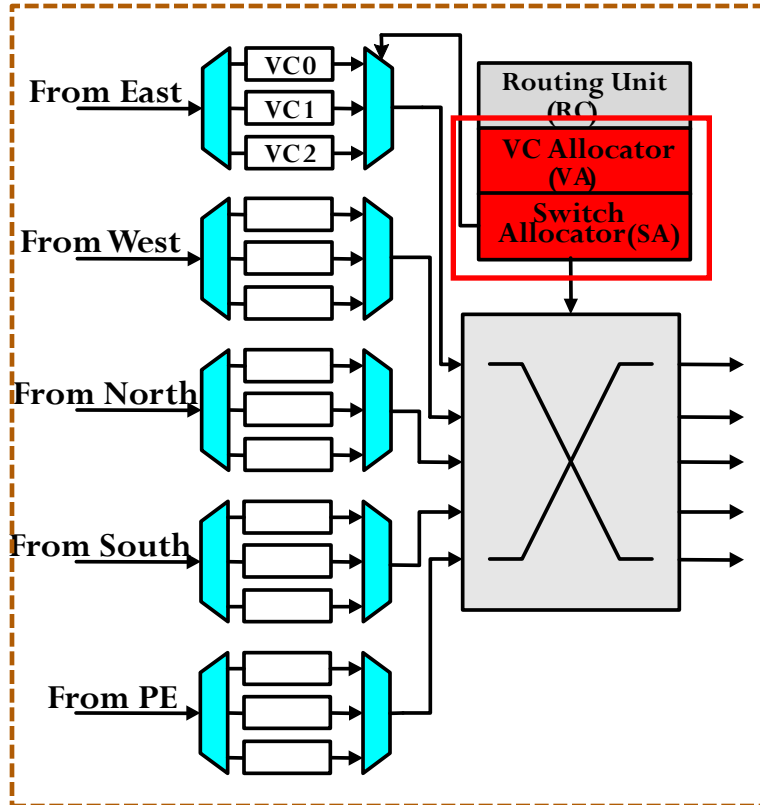


Network-on-Chip is a **critical** resource  
**shared** by **multiple applications**

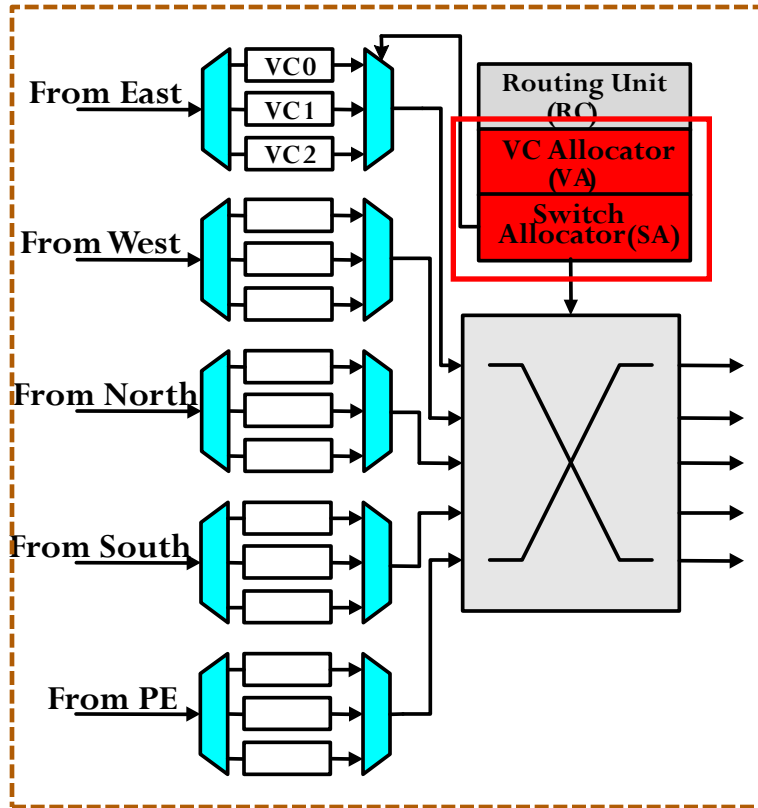
# The Problem: Packet Scheduling



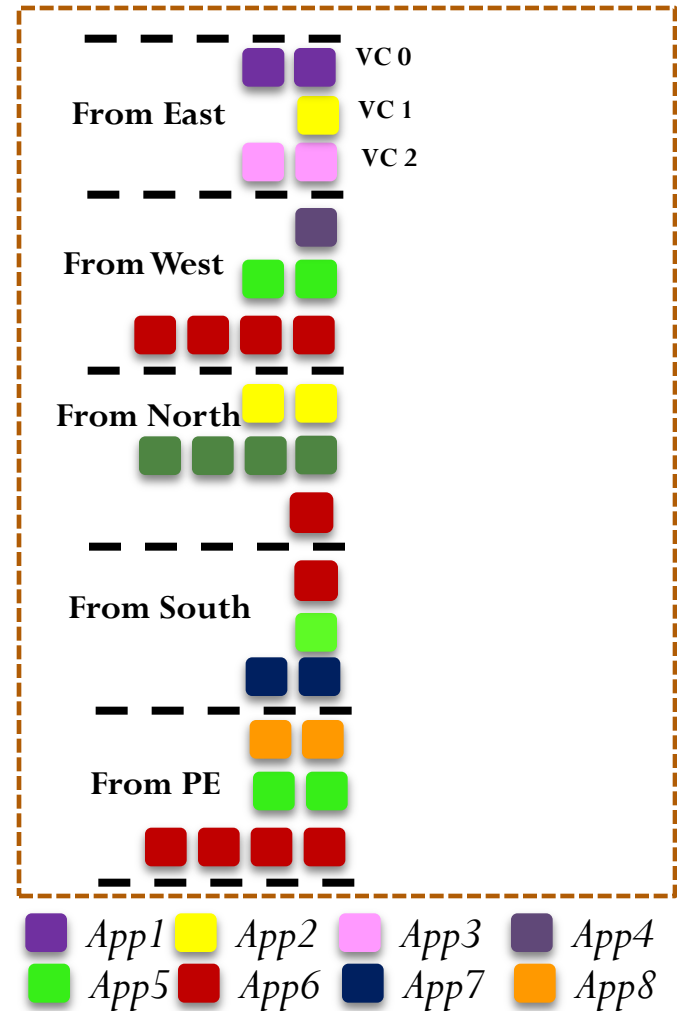
# The Problem: Packet Scheduling



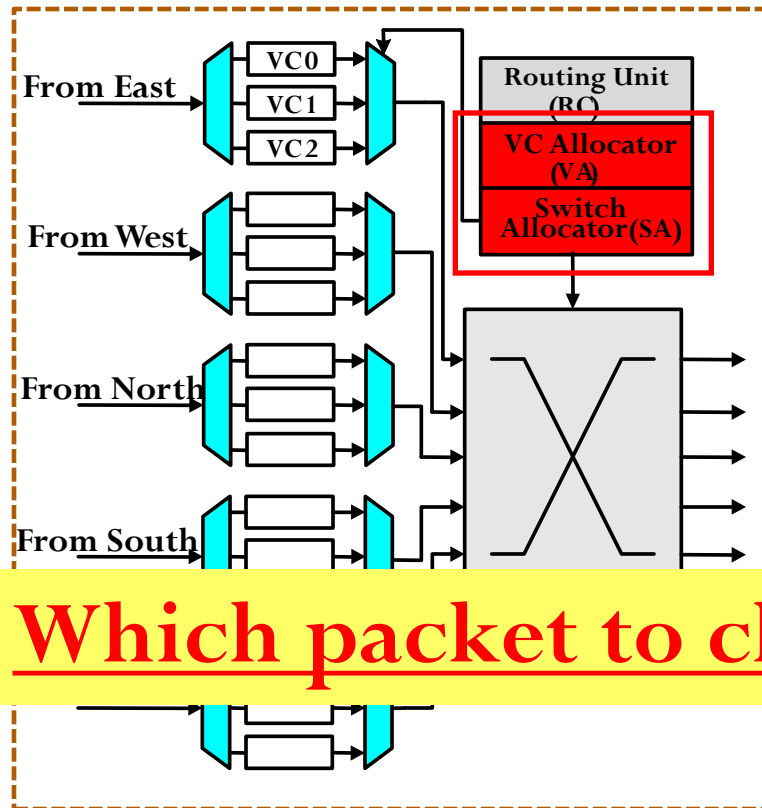
# The Problem: Packet Scheduling



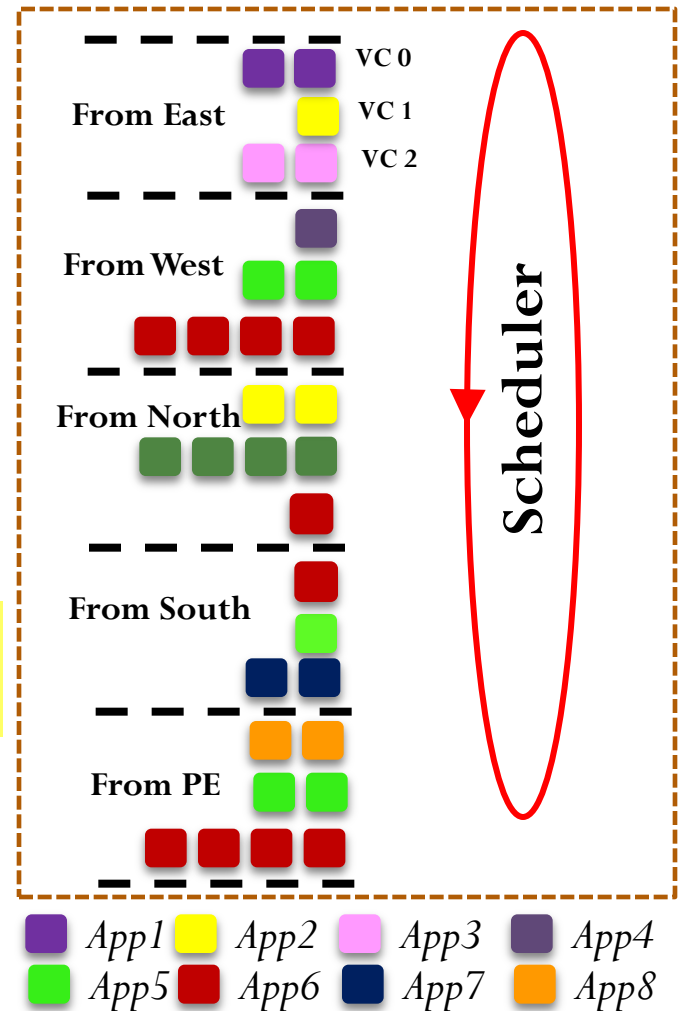
Conceptual  
View



# The Problem: Packet Scheduling



Conceptual  
View



# The Problem: Packet Scheduling

---

- Existing scheduling policies
  - Round Robin
  - Age
- Problem 1: **Local** to a router
  - Lead to contradictory decision making between routers: packets from one application may be prioritized at one router, to be delayed at next.
- Problem 2: **Application oblivious**
  - Treat all applications **packets equally**
  - But applications are heterogeneous
- **Solution** : Application-aware global scheduling policies.

# Motivation: Stall Time Criticality

---

- Applications are **not homogenous**
- Applications have different **criticality** with respect to the **network**
  - Some applications are network latency sensitive
  - Some applications are network latency tolerant
- Application's **Stall Time Criticality (STC)** can be measured by its average network stall time per packet (**i.e. NST/packet**)
  - **Network Stall Time (NST)** is number of cycles the processor stalls waiting for network transactions to complete

# Motivation: Stall Time Criticality

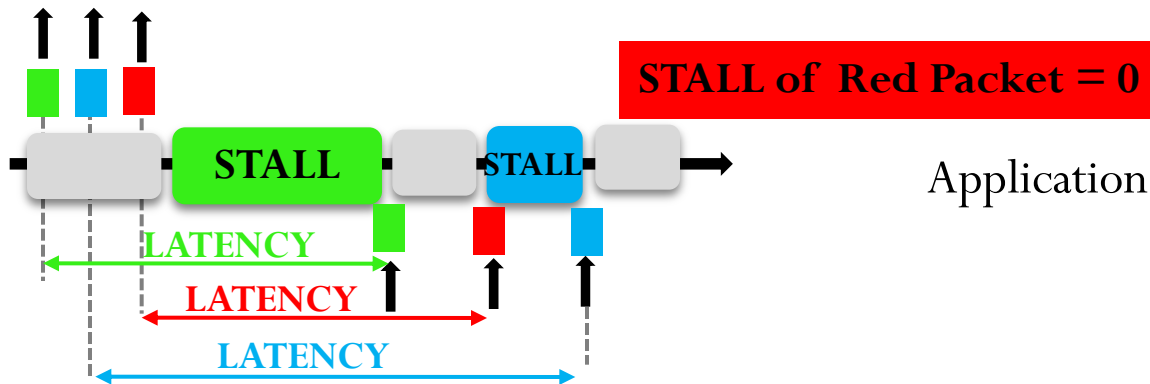
---

- Why applications have different network stall time criticality (STC)?
  - **Memory Level Parallelism (MLP)**
    - **Lower MLP leads to higher STC**
  - **Shortest Job First Principle (SJF)**
    - **Lower network load leads to higher STC**
  - **Average Memory Access Time**
    - **Higher memory access time leads to higher STC**



# STC Principle 1 {MLP}

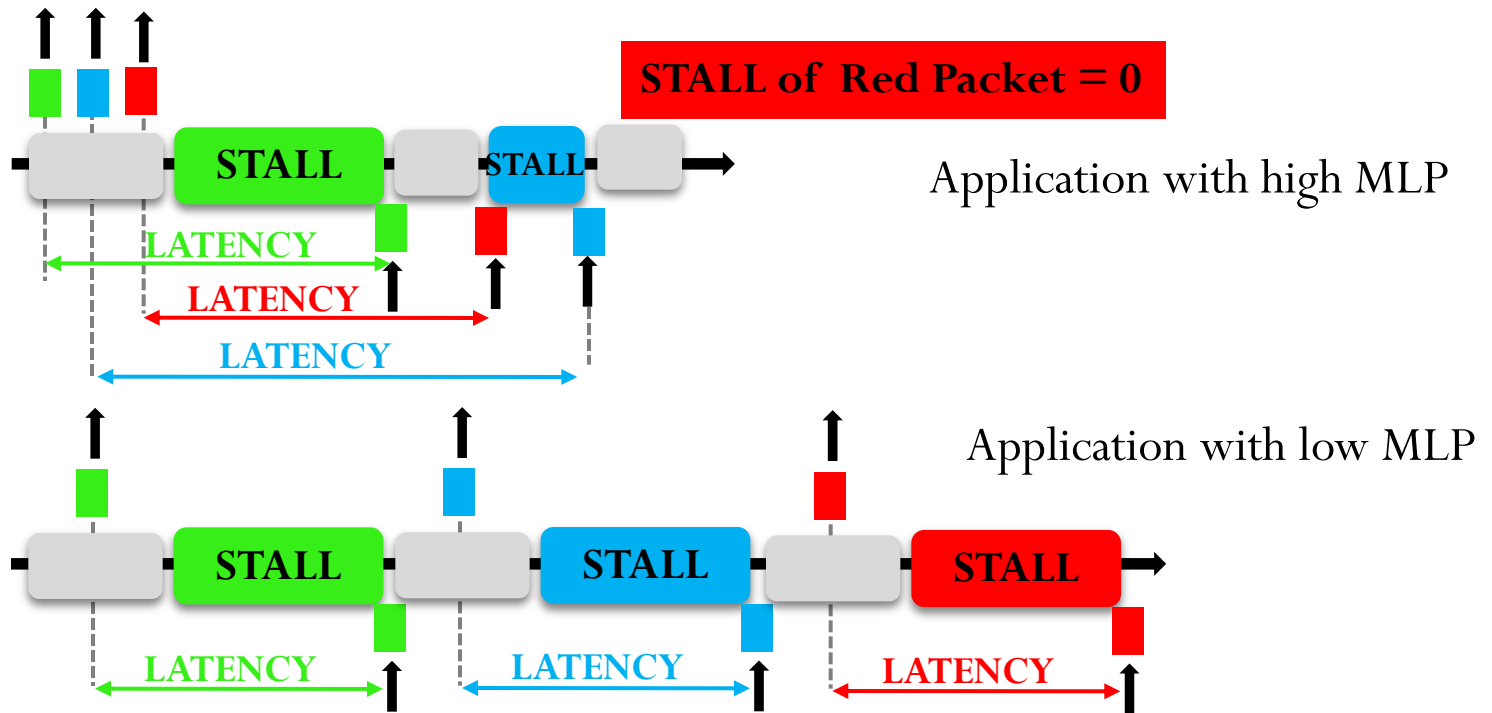
■ Compute



Application with high MLP

- Observation 1: **Packet Latency  $\neq$  Network Stall Time**

# STC Principle 1 {MLP}



- Observation 1: **Packet Latency  $\neq$  Network Stall Time**
- Observation 2: A low MLP application's packets have higher criticality than a high MLP application's

# STC Principle 2 {Shortest-Job-First}

Light Application

Heavy Application

Running ALONE

■ Compute



Baseline (RR) Scheduling



4X network slow down



1.3X network slow down

SJF Scheduling



1.2X network slow down



1.6X network slow down

Overall system throughput {weighted speedup} increases by 34%

# Solution: Application-Aware Policies

---

- Idea
  - Identify stall time critical applications (i.e. network sensitive applications) and prioritize their packets in each router.
- Key components of scheduling policy:
  - Application Ranking
  - Packet Batching
- Propose low-hardware complexity solution

# Component 1 : Ranking

---

- Ranking distinguishes applications based on Stall Time Criticality (STC)
- Periodically **rank** applications based on Stall Time Criticality (STC).
- Explored many **heuristics** for quantifying STC (Details & analysis in paper)
  - Heuristic based on **outermost private cache Misses Per Instruction (L1-MPI)** is the most effective
  - **Low L1-MPI => high STC => higher rank**
- Why Misses Per Instruction (L1-MPI)?
  - Easy to Compute (low complexity)
  - Stable Metric (unaffected by interference in network)

# Component 1 : How to Rank?

---

- Execution time is divided into fixed “ranking intervals”
  - Ranking interval is 350,000 cycles
- At the end of an interval, each core calculates their L1-MPI and sends it to the Central Decision Logic (CDL)
  - CDL is located in the central node of mesh
- CDL forms a ranking order and sends back its rank to each core
  - Two control packets per core every ranking interval
- Ranking order is a “partial order”
  
- Rank formation is **not** on the **critical path**
  - Ranking interval is significantly longer than rank computation time
  - Cores use older rank values until new ranking is available

# Component 2: Batching

---

- Problem: **Starvation**
  - Prioritizing a higher ranked application can lead to starvation of lower ranked application
- Solution: **Packet Batching**
  - Network packets are grouped into finite sized batches
  - **Packets of older batches are prioritized over younger batches**
- Alternative batching policies explored in paper
- **Time-Based Batching**
  - New batches are formed in a periodic, synchronous manner across all nodes in the network, every  $T$  cycles

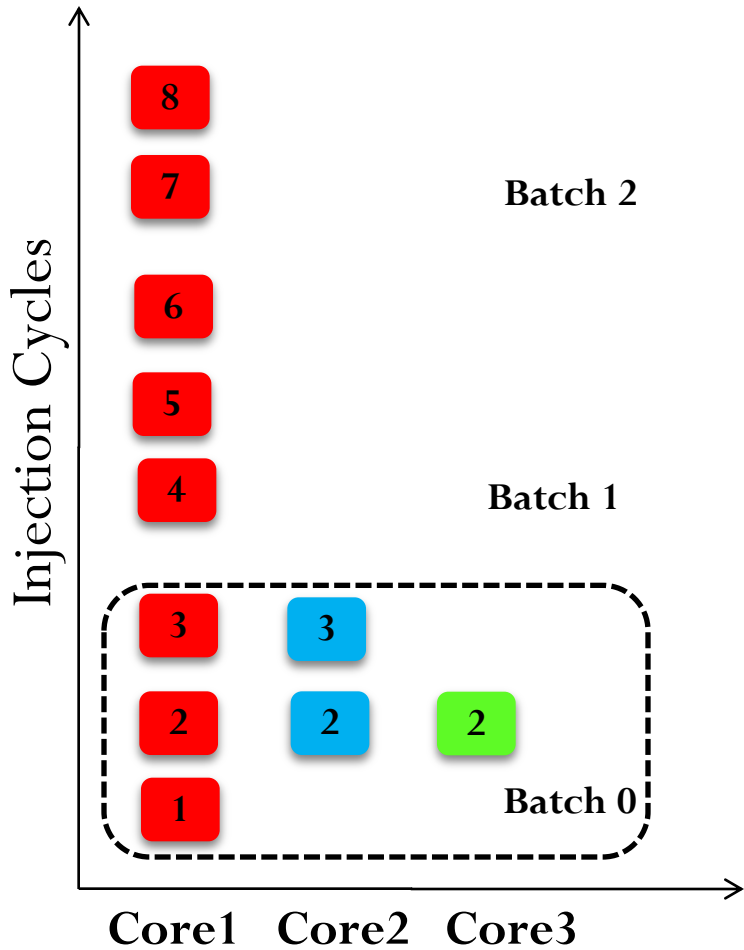
# Putting it all together

---




- Before injecting a packet into the network, it is tagged by
  - Batch ID (*3 bits*)
  - Rank ID (*3 bits*)
- Three tier priority structure at routers
  - **Oldest batch first** (*prevent starvation*)
  - **Highest rank first** (*maximize performance*)
  - **Local Round-Robin** (*final tie breaker*)
- Simple hardware support: priority arbiters
- **Global coordinated scheduling**
  - Ranking order and batching order are same across all routers



# STC Scheduling Example

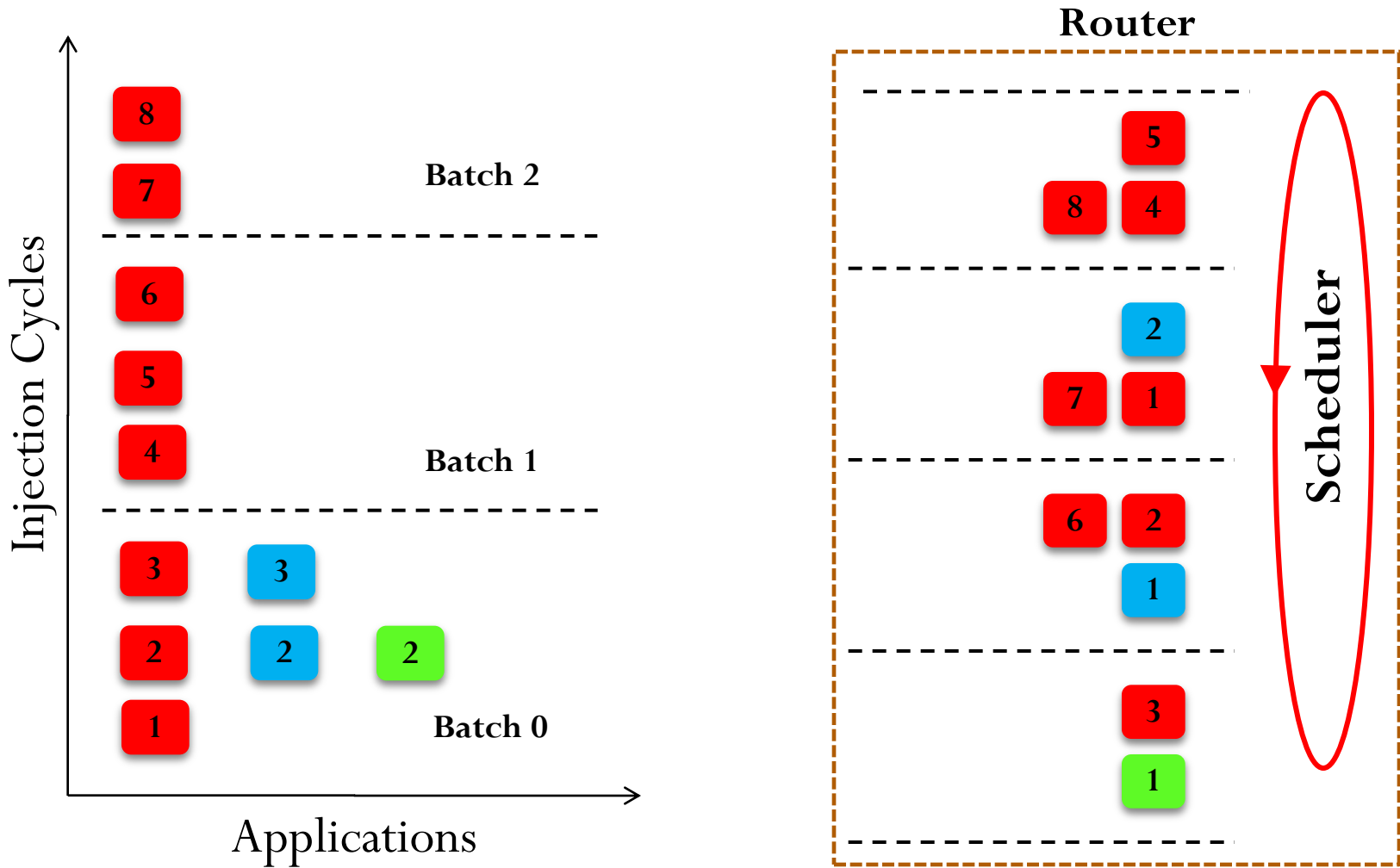


Batching interval length = 3 cycles

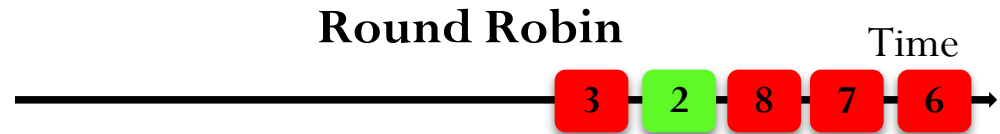
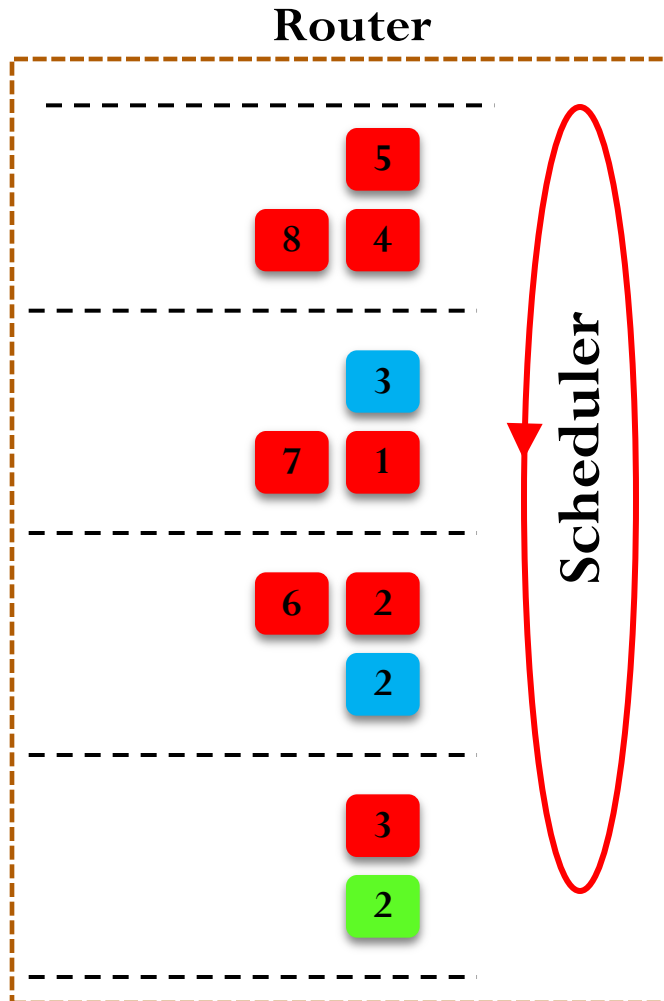
Ranking order =   

**Packet Injection Order at Processor**

# STC Scheduling Example

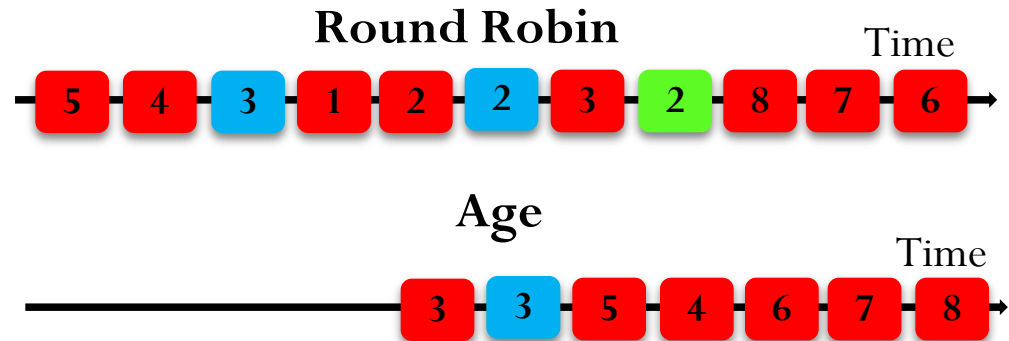
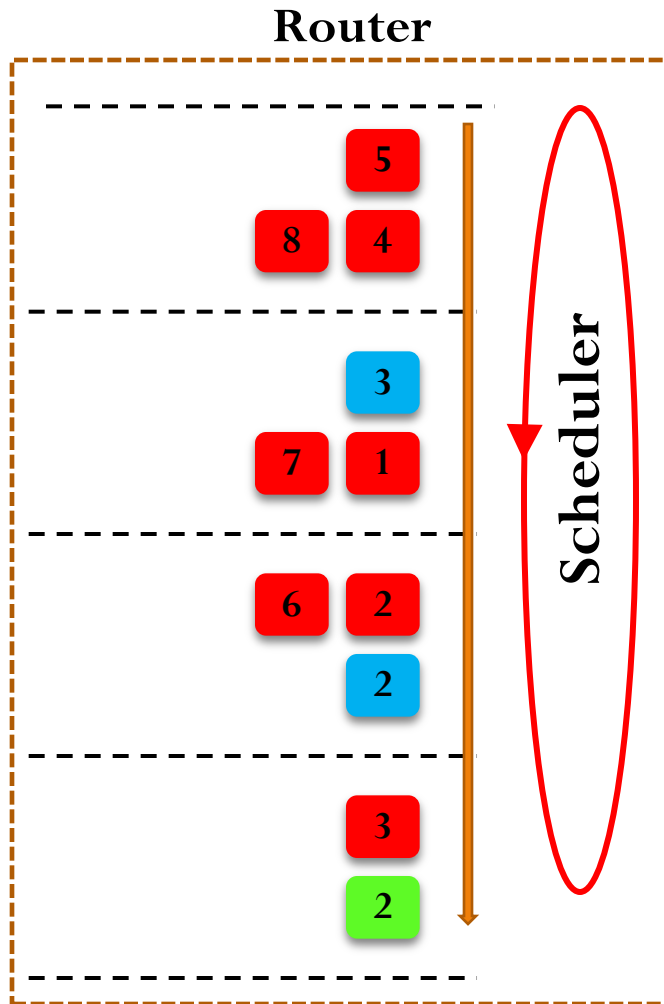


# STC Scheduling Example



	STALL CYCLES			Avg
RR	8	6	11	8.3
Age				
STC				

# STC Scheduling Example

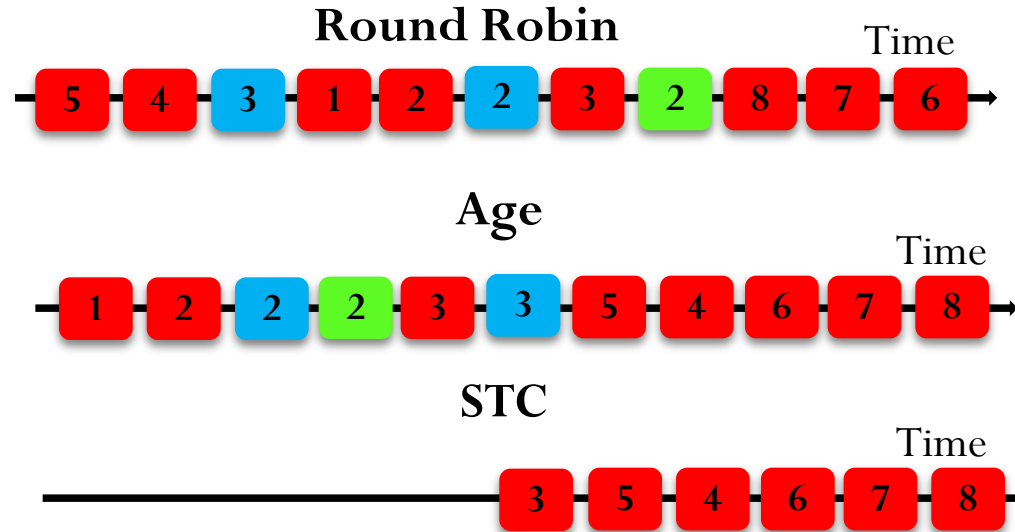
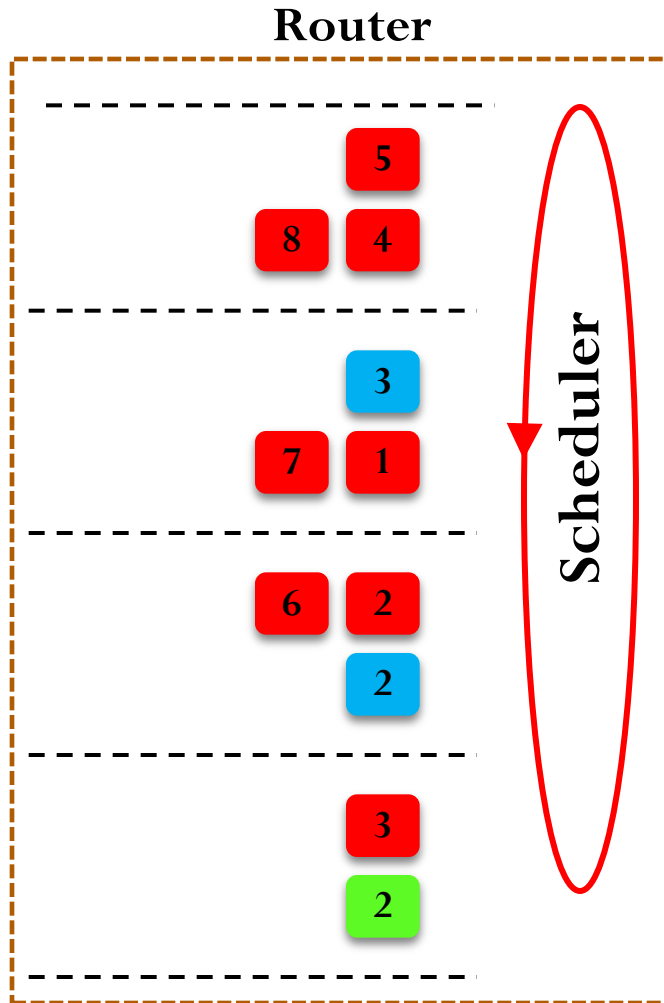


	STALL CYCLES			Avg
RR	8	6	11	8.3
Age	4	6	11	7.0
STC				

Ranking order



# STC Scheduling Example



STALL CYCLES				Avg
RR	8	6	11	8.3
Age	4	6	11	7.0
STC	1	3	11	5.0

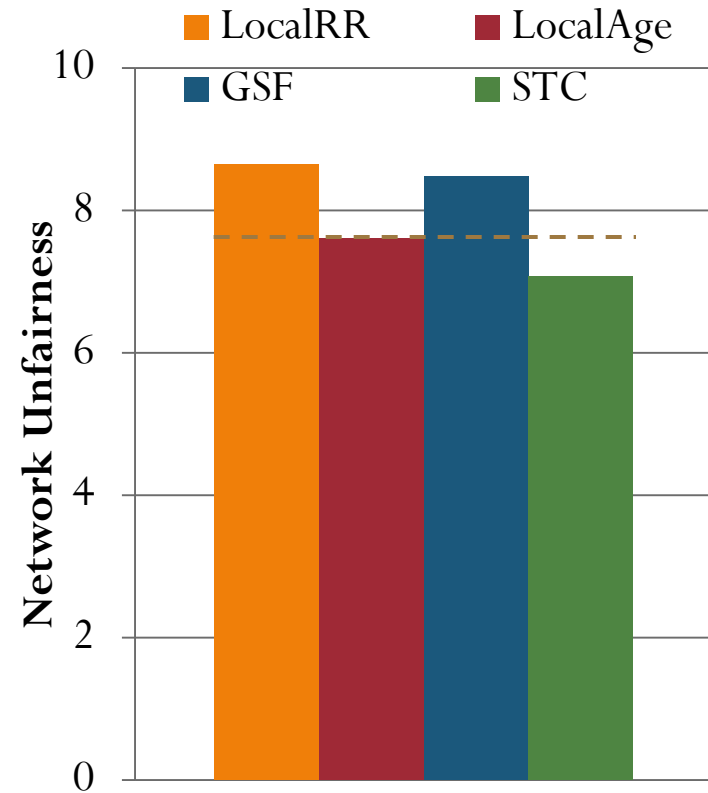
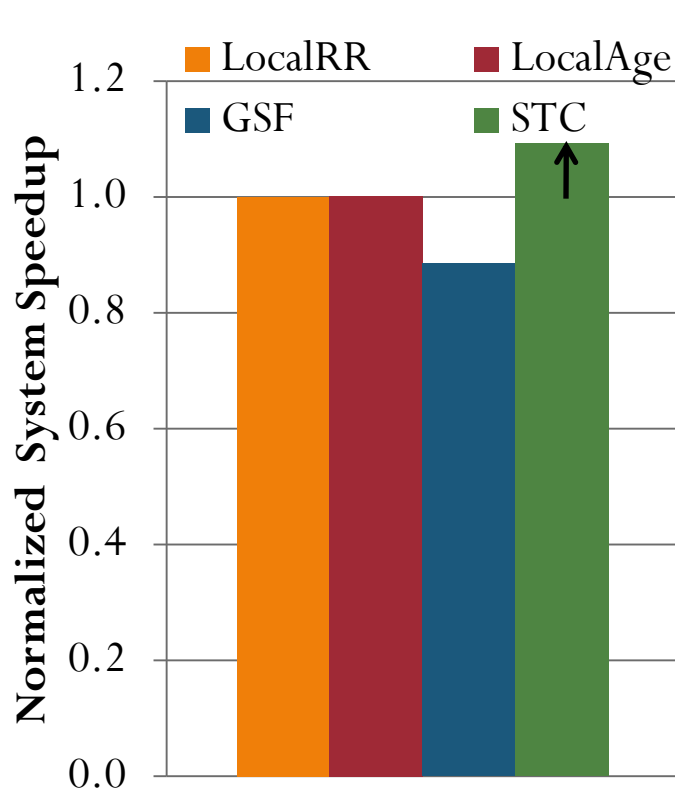
# Qualitative Comparison

---

- **Round Robin & Age**
  - Local and application oblivious
  - Age is biased towards heavy applications
    - heavy applications flood the network
    - higher likelihood of an older packet being from heavy application
- **Globally Synchronized Frames (GSF)** [Lee et al., ISCA 2008]
  - Provides **bandwidth fairness** at the expense of **system performance**
  - Penalizes heavy and bursty applications
    - Each application gets equal and fixed quota of flits (credits) in each batch.
    - Heavy application quickly run out of credits after injecting into all active batches & stall till oldest batch completes and frees up fresh credits.
    - Underutilization of network resources

# System Performance

- STC provides 9.1% improvement in weighted speedup over the best existing policy {averaged across 96 workloads}
- Detailed case studies in the paper



# Slack-Driven Packet Scheduling

Das et al., "Aergia: Exploiting Packet Latency Slack in On-Chip Networks,"  
ISCA 2010.



# Packet Scheduling in NoC

---

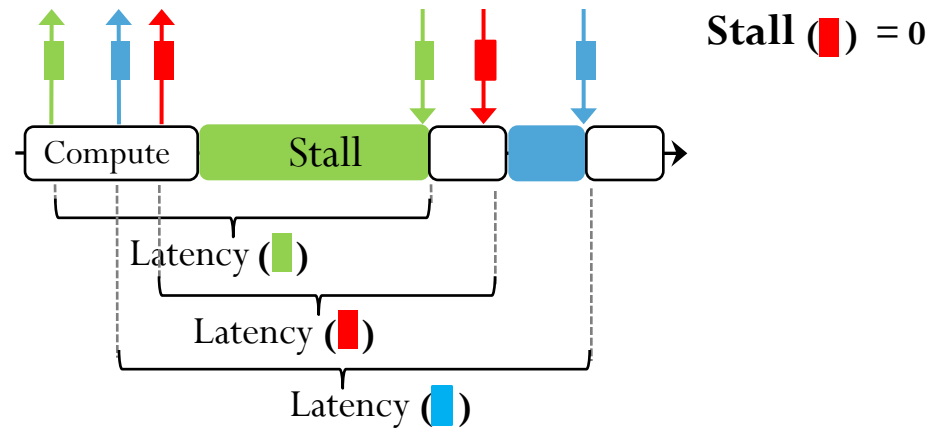
- Existing scheduling policies
  - Round robin
  - Age
- Problem
  - Treat all packets equally
  - Application-oblivious
- Packets have **different criticality**
  - Packet is critical if latency of a packet affects application's performance
  - Different criticality due to memory level parallelism (MLP)

All packets are not the same...!!!



# MLP Principle

---



Packet Latency  $\neq$  Network Stall Time

Different Packets have different criticality due to MLP

Criticality(■) > Criticality(■) > Criticality(■)

# Outline

---

- Introduction
  - Packet Scheduling
  - Memory Level Parallelism
- Aergia
  - Concept of Slack
  - Estimating Slack
- Evaluation
- Conclusion

# What is Aérgia?

---



- Aérgia is the spirit of laziness in Greek mythology
- Some packets can afford to **slack!**

# Outline

---

- Introduction
  - Packet Scheduling
  - Memory Level Parallelism
- Aergia
  - Concept of Slack
  - Estimating Slack
- Evaluation
- Conclusion

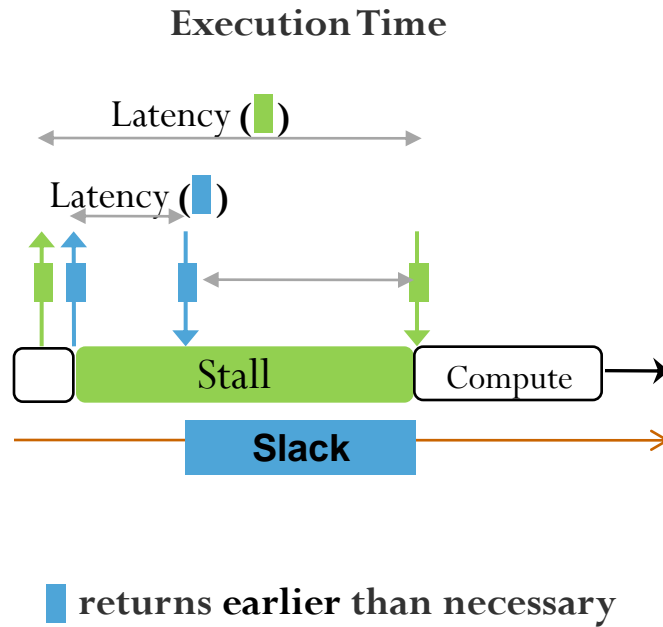
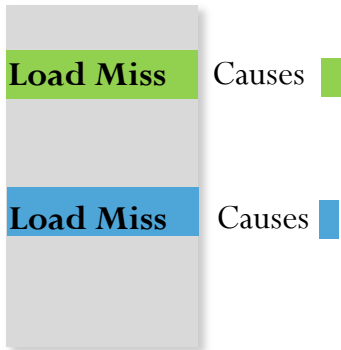
# Slack of Packets

---

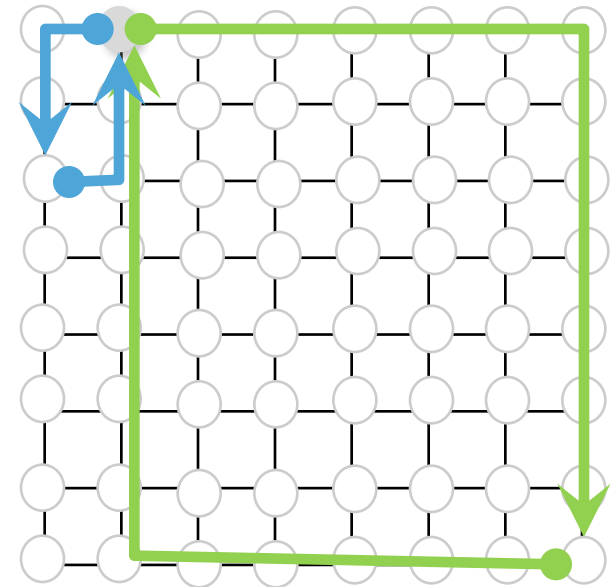
- What is slack of a packet?
  - Slack of a packet is number of cycles it can be delayed in a router without reducing application's performance
  - **Local network slack**
- Source of slack: Memory-Level Parallelism (MLP)
  - Latency of an application's packet hidden from application due to **overlap** with latency of pending cache miss requests
- Prioritize packets with **lower slack**

# Concept of Slack

Instruction Window



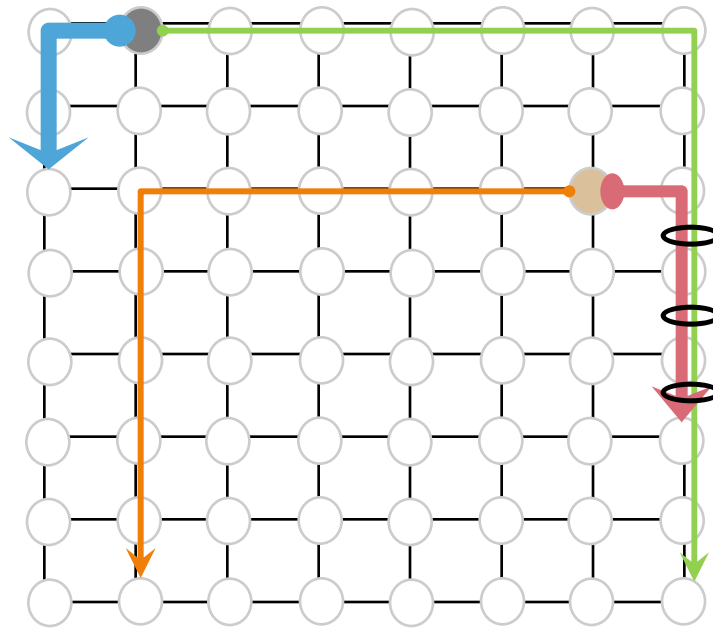
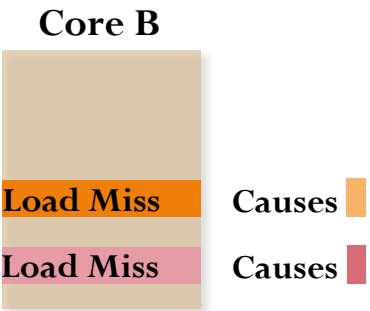
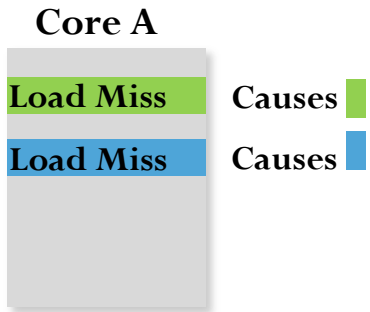
Network-on-Chip





$$\text{Slack (■)} = \text{Latency (■)} - \text{Latency (■)} = 26 - 6 = 20 \text{ hops}$$



Packet(■) can be delayed for available slack cycles without reducing performance!

# Prioritizing using Slack



Packet	Latency	Slack
	13 hops	0 hops
	3 hops	10 hops

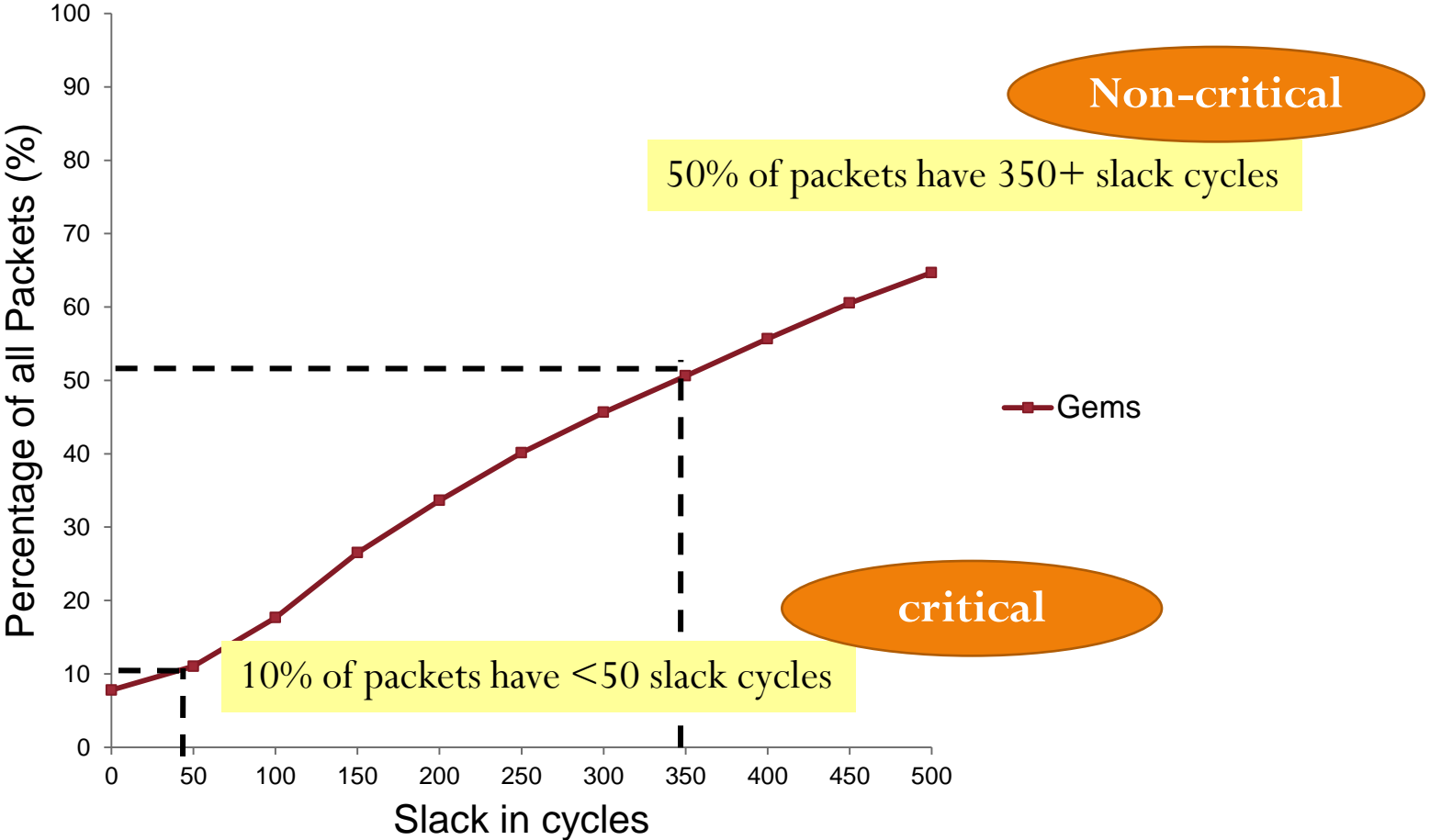
 Interference at 3 hops

Slack() > Slack ()

Prioritize 

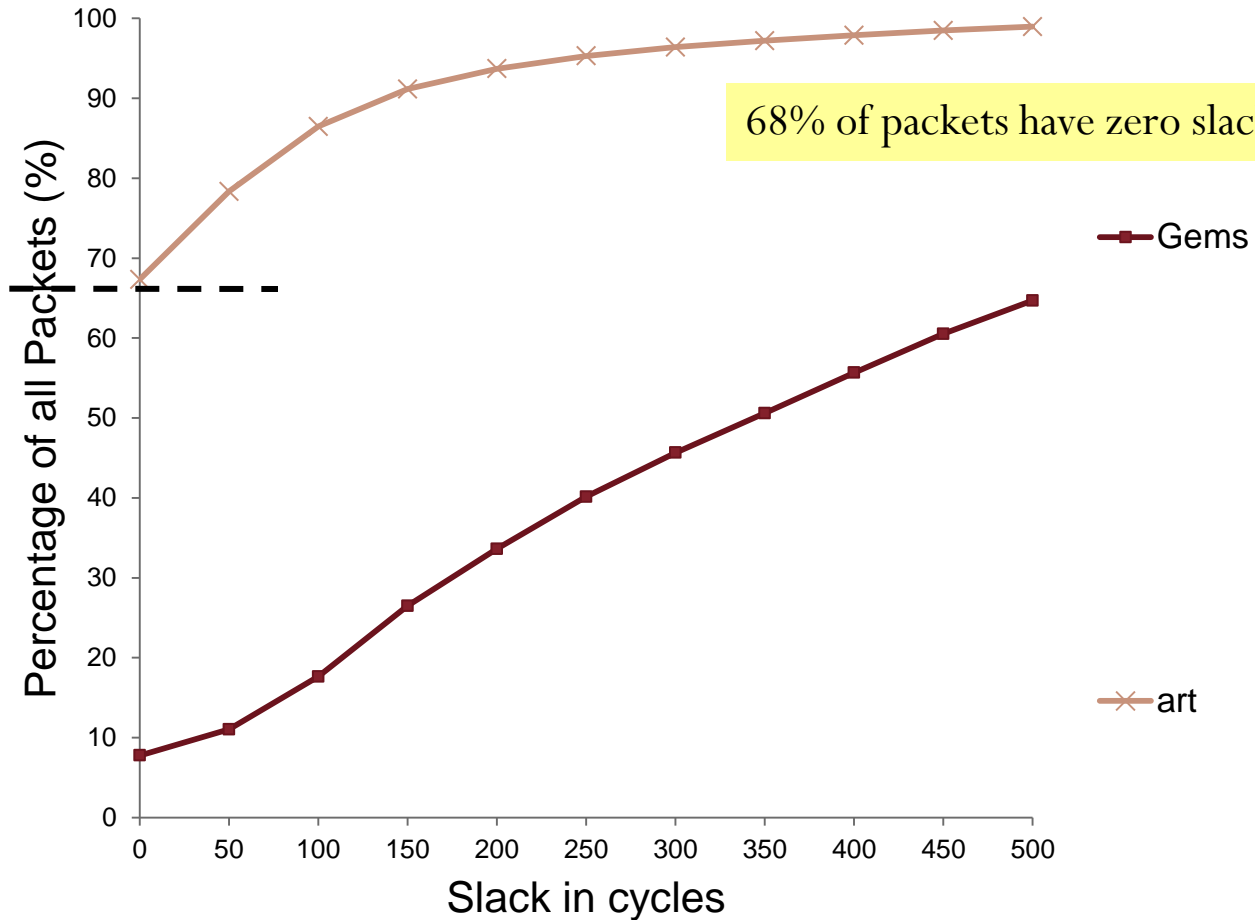


# Slack in Applications

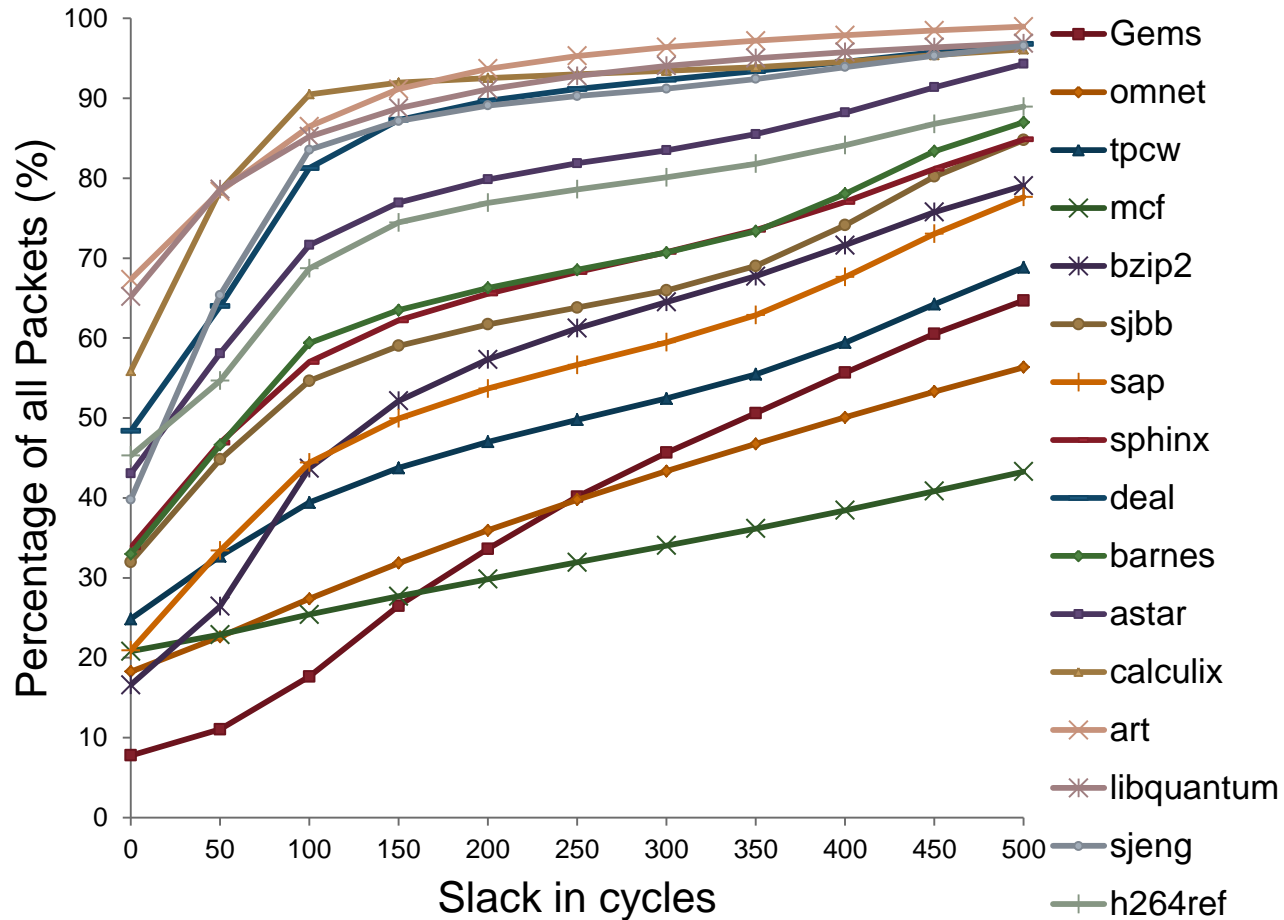


# Slack in Applications

---



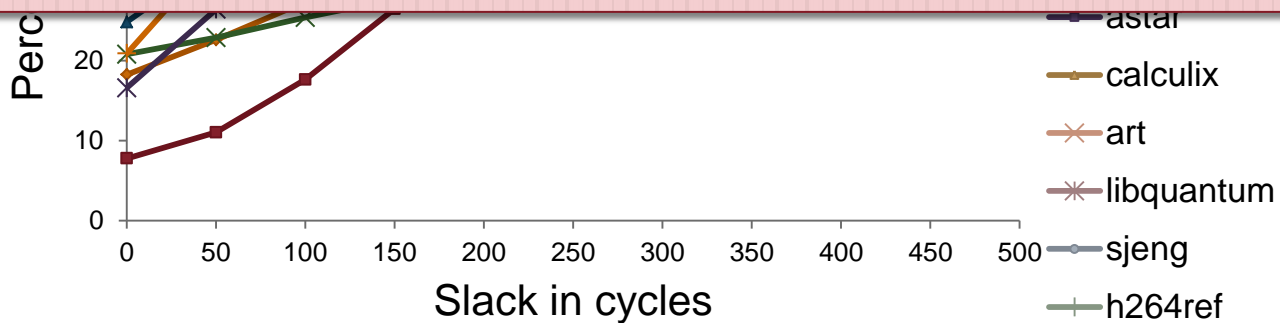
# Diversity in Slack



# Diversity in Slack



Slack varies **between** packets of **different** applications



Slack varies **between** packets of a **single** application

# Outline

---

- Introduction
  - Packet Scheduling
  - Memory Level Parallelism
- Aergia
  - Concept of Slack
  - Estimating Slack
- Evaluation
- Conclusion

# Estimating Slack Priority

---

**Slack (P)** = Max (Latencies of P's Predecessors) – Latency of P

**Predecessors(P)** are the packets of outstanding cache miss requests when P is issued

- Packet latencies not known when issued
- Predicting latency of any packet Q
  - Higher latency if Q corresponds to an L2 miss
  - Higher latency if Q has to travel farther number of hops

# Estimating Slack Priority

---

- Slack of P = Maximum Predecessor Latency – Latency of P

- Slack(P) = 

PredL2 (2 bits)	MyL2 (1 bit)	HopEstimate (2 bits)
--------------------	-----------------	-------------------------

**PredL2:** Set if any predecessor packet is servicing L2 miss

**MyL2:** Set if P is NOT servicing an L2 miss

**HopEstimate:** Max (# of hops of Predecessors) – hops of P

# Estimating Slack Priority

---

- How to predict L2 hit or miss at core?
  - *Global Branch Predictor* based L2 Miss Predictor
    - Use Pattern History Table and 2-bit saturating counters
  - *Threshold* based L2 Miss Predictor
    - If #L2 misses in “M” misses  $\geq$  “T” threshold then next load is a L2 miss.
- Number of miss predecessors?
  - List of outstanding L2 Misses
- Hops estimate?
  - Hops  $\Rightarrow \Delta X + \Delta Y$  distance
  - Use predecessor list to calculate slack hop estimate



# Starvation Avoidance

---

- Problem: **Starvation**
  - Prioritizing packets can lead to starvation of lower priority packets
- Solution: **Time-Based Packet Batching**
  - New batches are formed at every  $T$  cycles
  - Packets of older batches are prioritized over younger batches

# Putting it all together

---

- Tag header of the packet with priority bits before injection

**Priority (P) =**

Batch (3 bits)	PredL2 (2 bits)	MyL2 (1 bit)	HopEstimate (2 bits)
-------------------	--------------------	-----------------	-------------------------

- Priority(P)?
  - P's batch *(highest priority)*
  - P's Slack
  - Local Round-Robin *(final tie breaker)*

# Outline

---

- Introduction
  - Packet Scheduling
  - Memory Level Parallelism
- Aergia
  - Concept of Slack
  - Estimating Slack
- Evaluation
- Conclusion

# Evaluation Methodology

---

- 64-core system
  - x86 processor model based on Intel Pentium M
  - 2 GHz processor, 128-entry instruction window
  - 32KB private L1 and 1MB per core shared L2 caches, 32 miss buffers
  - 4GB DRAM, 320 cycle access latency, 4 on-chip DRAM controllers
- Detailed Network-on-Chip model
  - 2-stage routers (with speculation and look ahead routing)
  - Wormhole switching (8 flit data packets)
  - Virtual channel flow control (6 VCs, 5 flit buffer depth)
  - 8x8 Mesh (128 bit bi-directional channels)
- Benchmarks
  - Multiprogrammed scientific, server, desktop workloads (35 applications)
  - 96 workload combinations

# Qualitative Comparison

---

- **Round Robin & Age**

- Local and application oblivious
- Age is biased towards heavy applications

- **Globally Synchronized Frames (GSF)**

[Lee et al., ISCA 2008]

- Provides **bandwidth fairness** at the expense of **system performance**
- Penalizes heavy and bursty applications

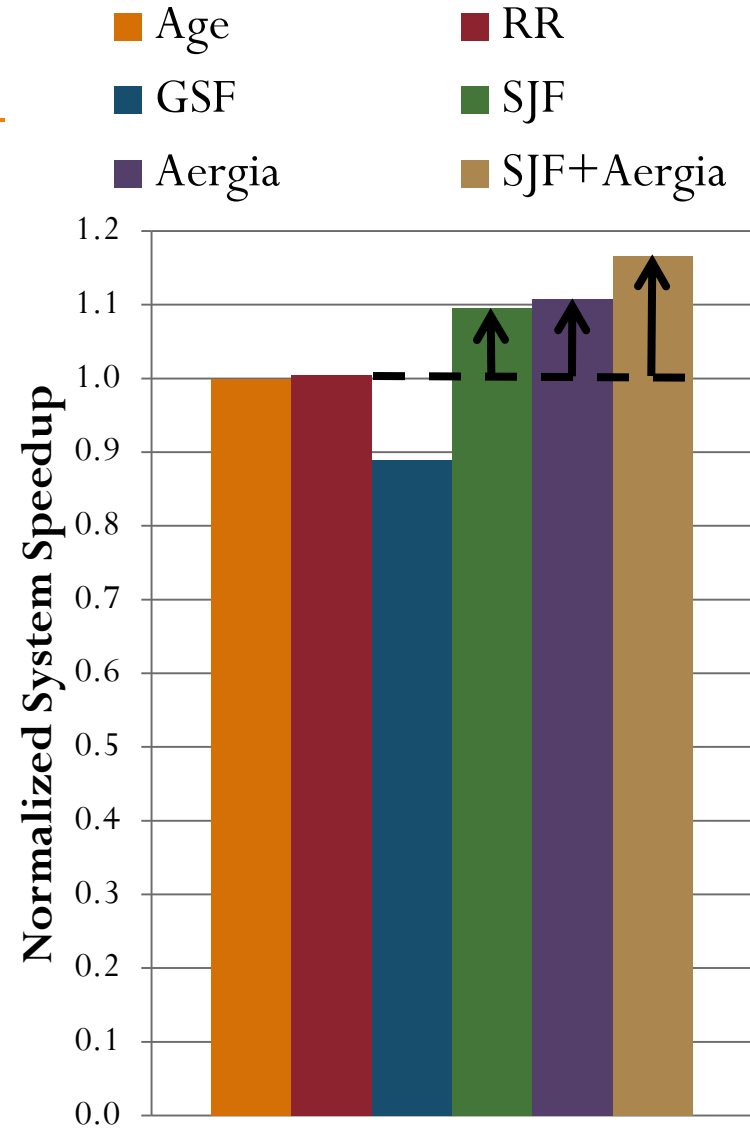
- **Application-Aware Prioritization Policies (SJF)**

[Das et al., MICRO 2009]

- **Shortest-Job-First Principle**
- Packet scheduling policies which prioritize network sensitive applications which inject lower load

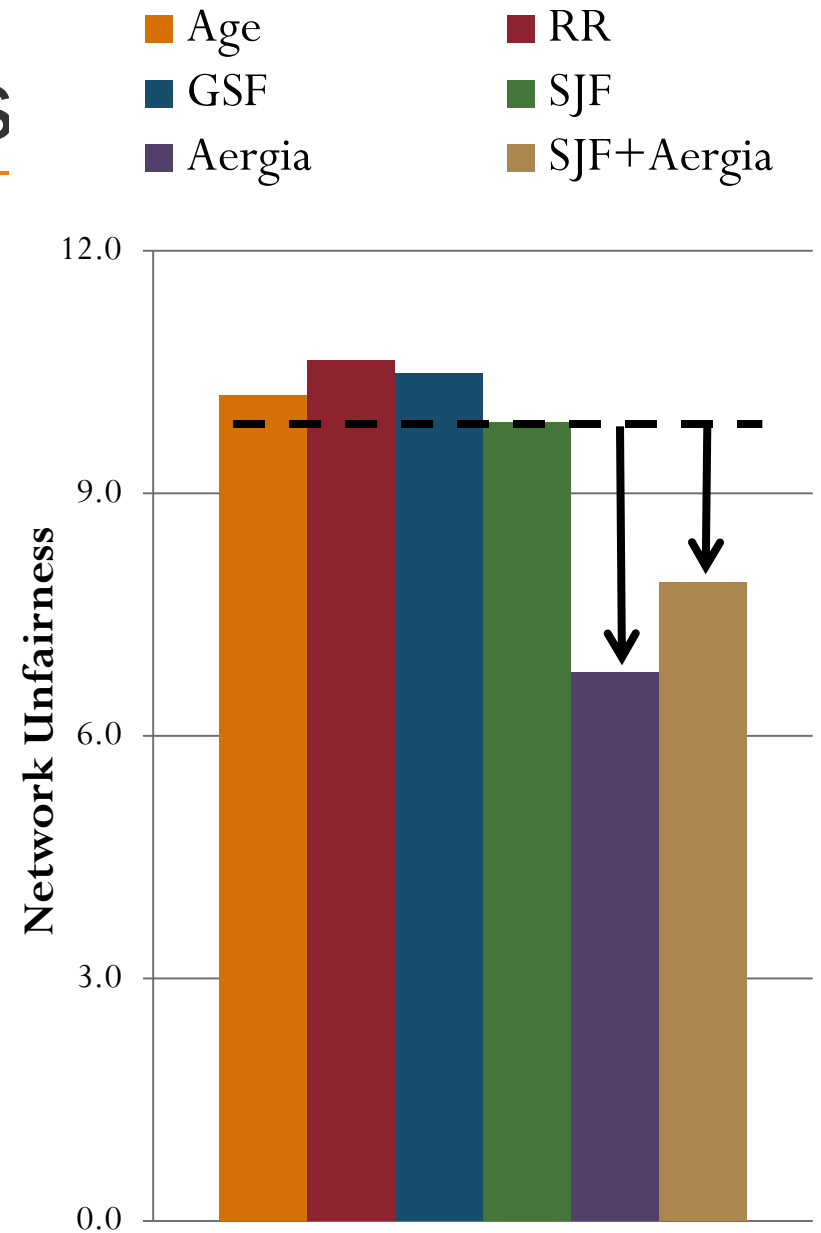
# System Performance

- SJF provides 8.9% improvement in weighted speedup
- Aergia improves system throughput by 10.3%
- Aergia+SJF improves system throughput by 16.1%



# Network Unfairness

- SJF does not imbalance network fairness
- Aergia improves network unfairness by 1.5X
- SJF+Aergia improves network unfairness by 1.3X



# Conclusions & Future Directions

---

- Packets have different criticality, yet existing packet scheduling policies **treat all packets equally**
- We propose a new approach to packet scheduling in NoCs
  - We define **Slack** as a key measure that characterizes the relative importance of a packet.
  - We propose **Aérgia** a novel architecture to accelerate low slack critical packets
- Result
  - Improves system performance: 16.1%
  - Improves network fairness: 30.8%