Operating Systems
(Honor Track)

Scheduling 4:
Scheduling in Modern Computer Systems

Xin Jin

Spring 2026

Acknowledgments: Ion Stoica, Berkeley CS 162

Scheduling in Modern Computer Systems

- FCFS
  - SOSP’17 ZygOS
- RR
  - NSDI’19 Shinjuku
- SJF, SRTF, MLFQ
  - NSDI’19 Tiresias
- Fairness
  - NSDI’11 DRF
  - NSDI’16 FairRide

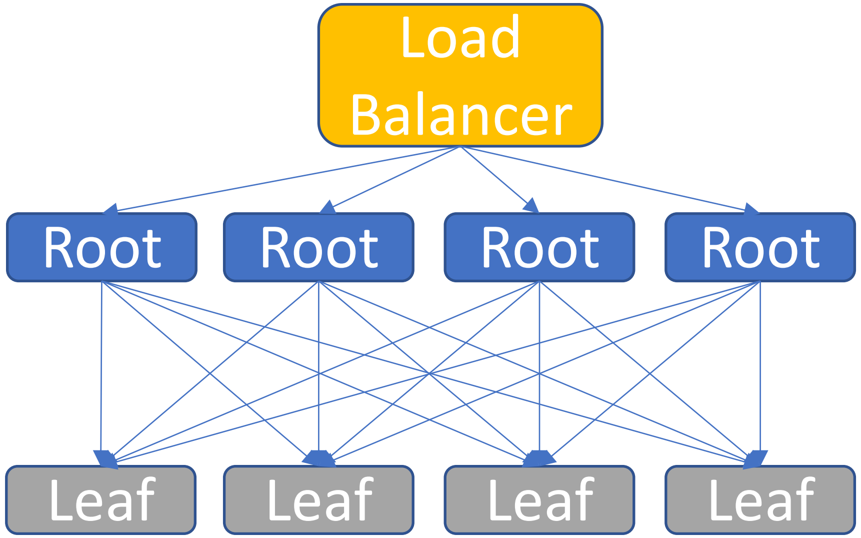
ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks

George Prekas, **Marios Kogias**, Edouard Bugnion

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

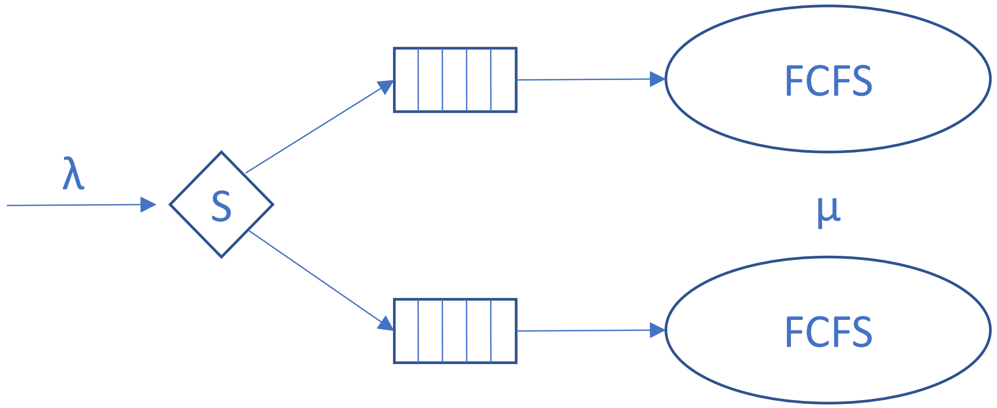
Problem: Serve μs-scale RPCs

- Applications: KV-stores, In-memory DB
- Datacenter environment:
  - Complex fan-out – fan-in patterns
- Tail-at-scale problem
- Tail Latency Service-Level Objectives
- Goal: Improve throughput at an aggressive tail latency SLO
- How? Focus within the leaf nodes
  - Reduce system overheads
  - Achieve better scheduling



Elementary Queuing Theory

- Processor
  - FCFS
  - Processor Sharing
- Multi/Single Queue
- Inter-arrival Distribution (λ)
  - Poisson
- Service Time Distribution (μ)
  - Fixed
  - Exponential
  - Bimodal



- No OS overheads
- Independent of service time
- Upper performance bound

Baseline

System	Linux		Dataplanes
Networking	Kernel (epoll)	Kernel (epoll)	Userspace
Connection Delegation	Partitioned	Floating	Partitioned
Complexity	Medium	High	Low
Work Conservation	✗	✓	✗
Queuing	Multi-Queue	Single Queue	Multi-Queue

Can we build a system with low overheads that achieves work conservation?

Upcoming

- Key Observations:
  - Single queue systems perform **theoretically** better
  - Dataplanes, despite being multi-queue systems, perform **practically** better

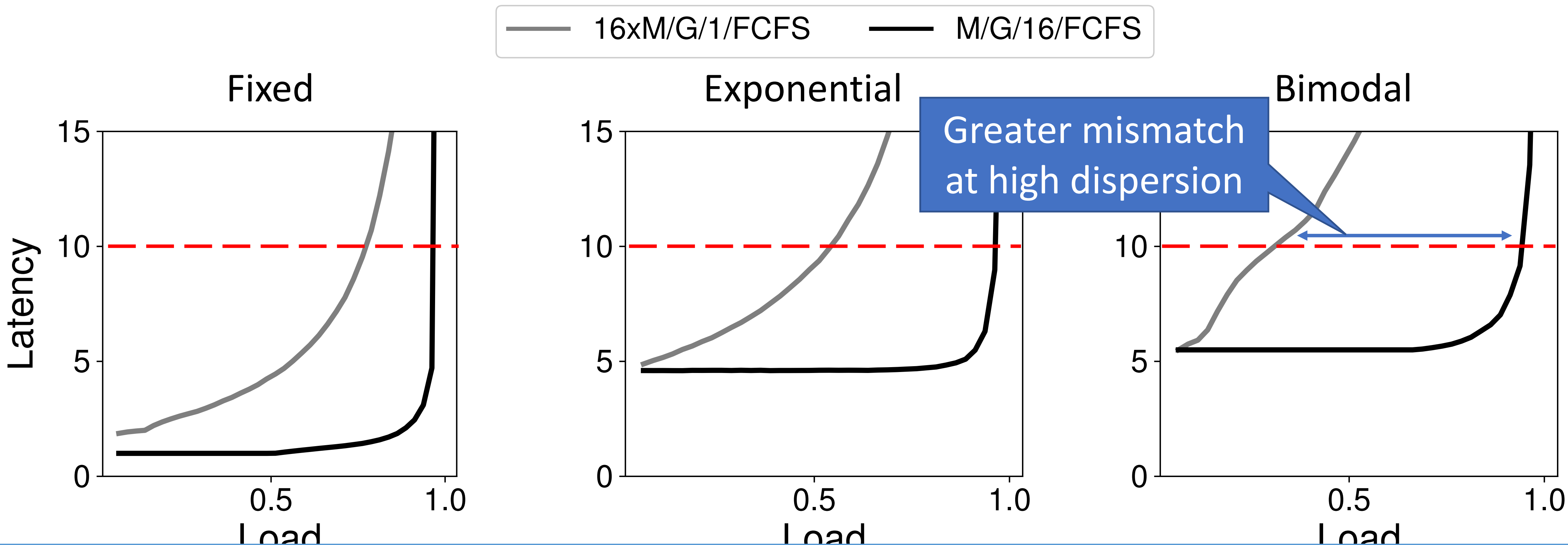
- Key Contributions
  - ZygOS combines the best of the two worlds:
    - Reduced system overheads similar to dataplanes
    - Convergence to a single-queue model

Analysis

- Metric to optimize: Load @ Tail-Latency SLO
- Run timescale-independent simulations
- Run synthetic benchmarks on real system

- Questions:
  - Which model achieves better throughput?
  - Which system converges to its model at low service times?

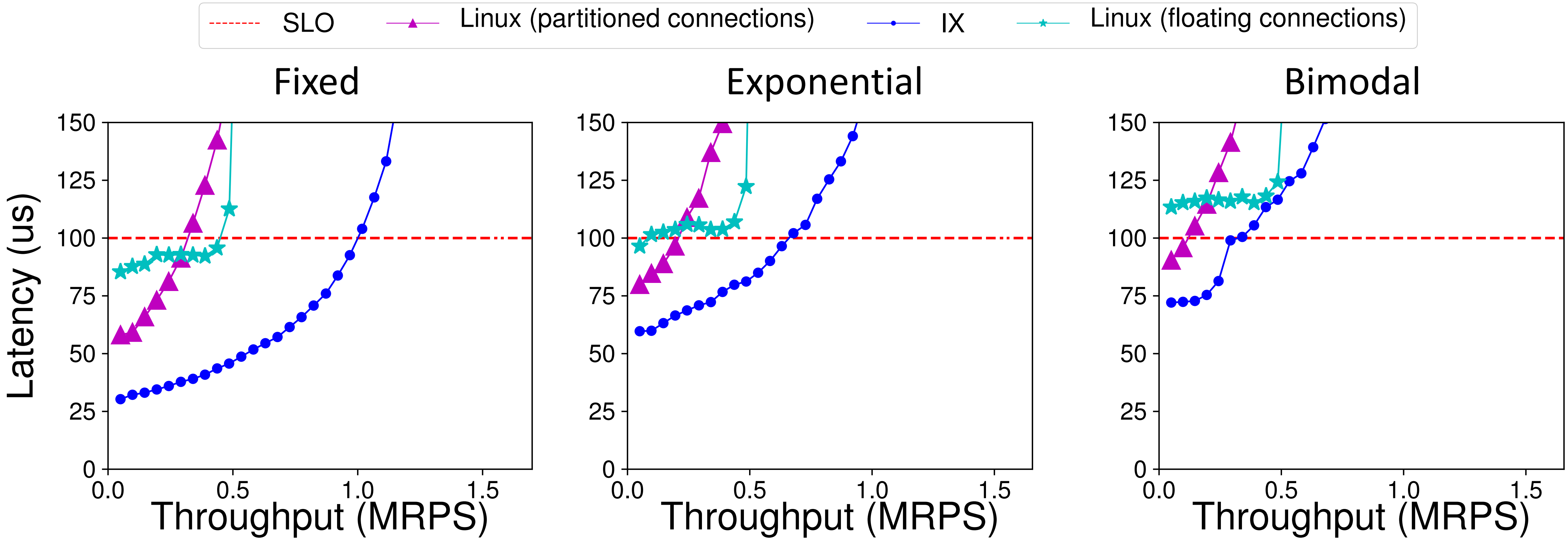
Latency vs Load – Queuing model



Single queue models provide better throughput at SLO because of **transient load imbalance**

SLO: 10 x AVG[service_time]

Latency vs Load – Service Time 10μs

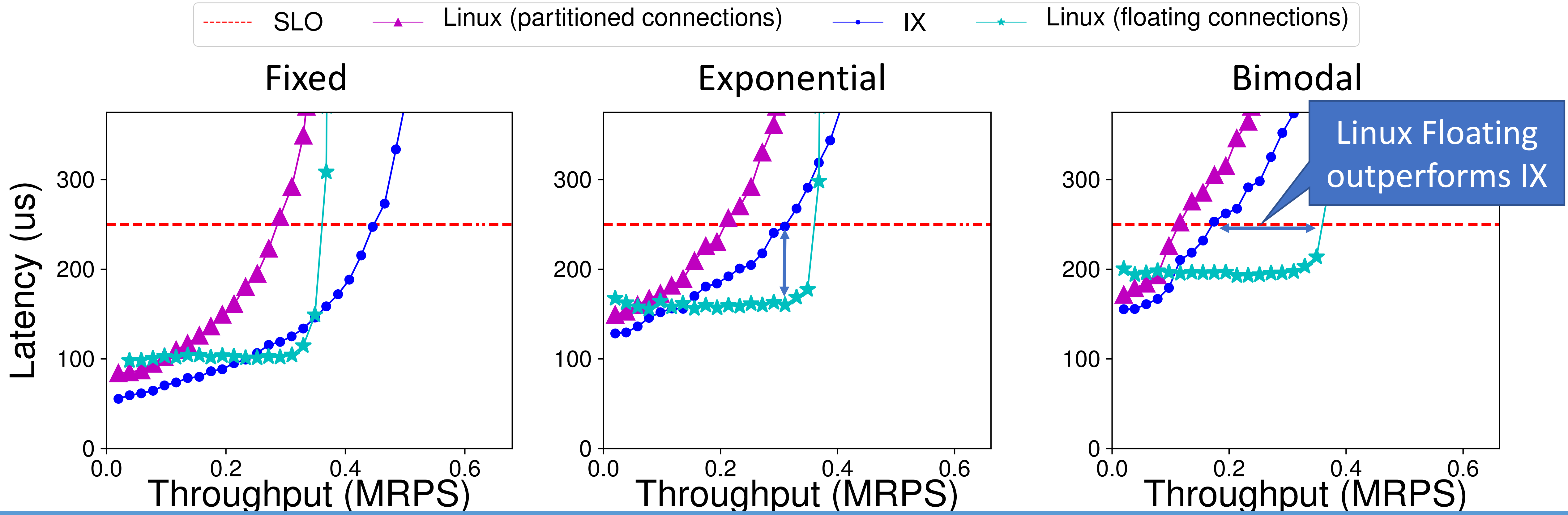


99^{th} percentile latency

SLO: 10 x AVG[service_time]

IX, Belay et al. OSDI 2014

Latency vs Load – Service Time 25µs



Dataplanes perform better **only** in very low service times with low dispersion

SLO: 10 x AVG[service_time]

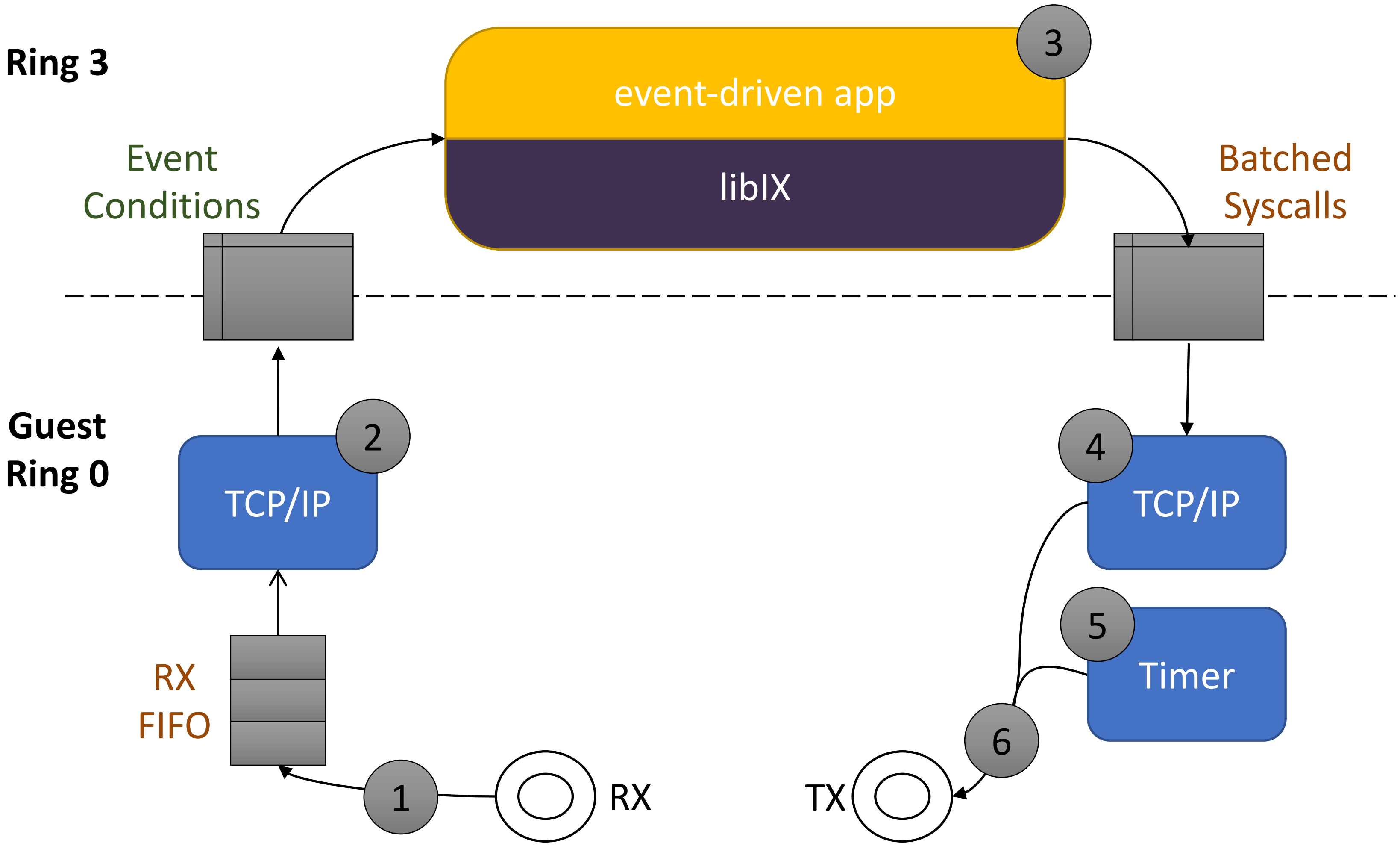
IX, Belay et al. OSDI 2014

ZygOS Approach

- Dataplane aspect:
  - Reduced system overheads
  - Share nothing network processing
- Single Queue system
  - Work conservation
  - Reduction of head of line blocking

Implement **work-stealing** to achieve work-conservation in a dataplane

Background on IX



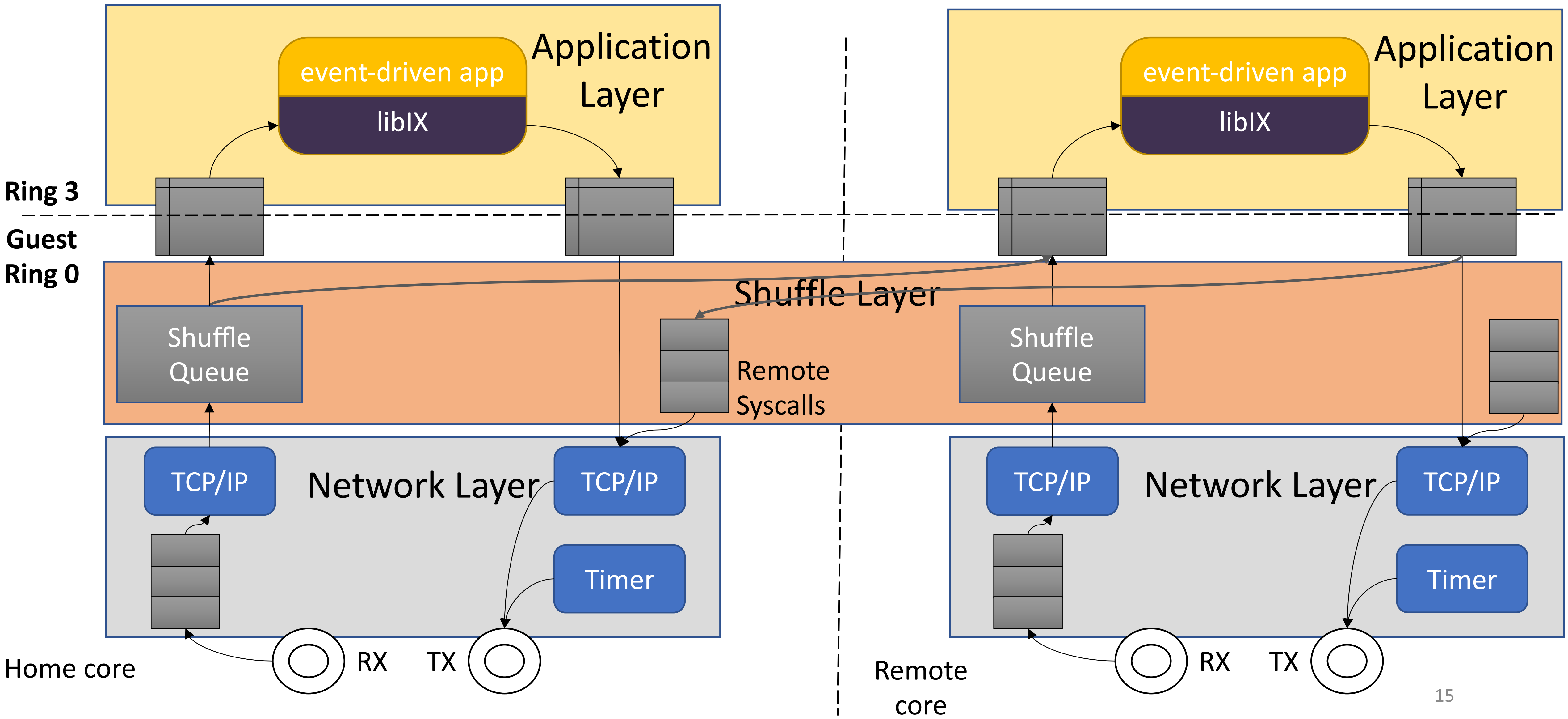
IX Design
ZygOS Design

1. Application layer
   Event based application
   that is agnostic to work-stealing

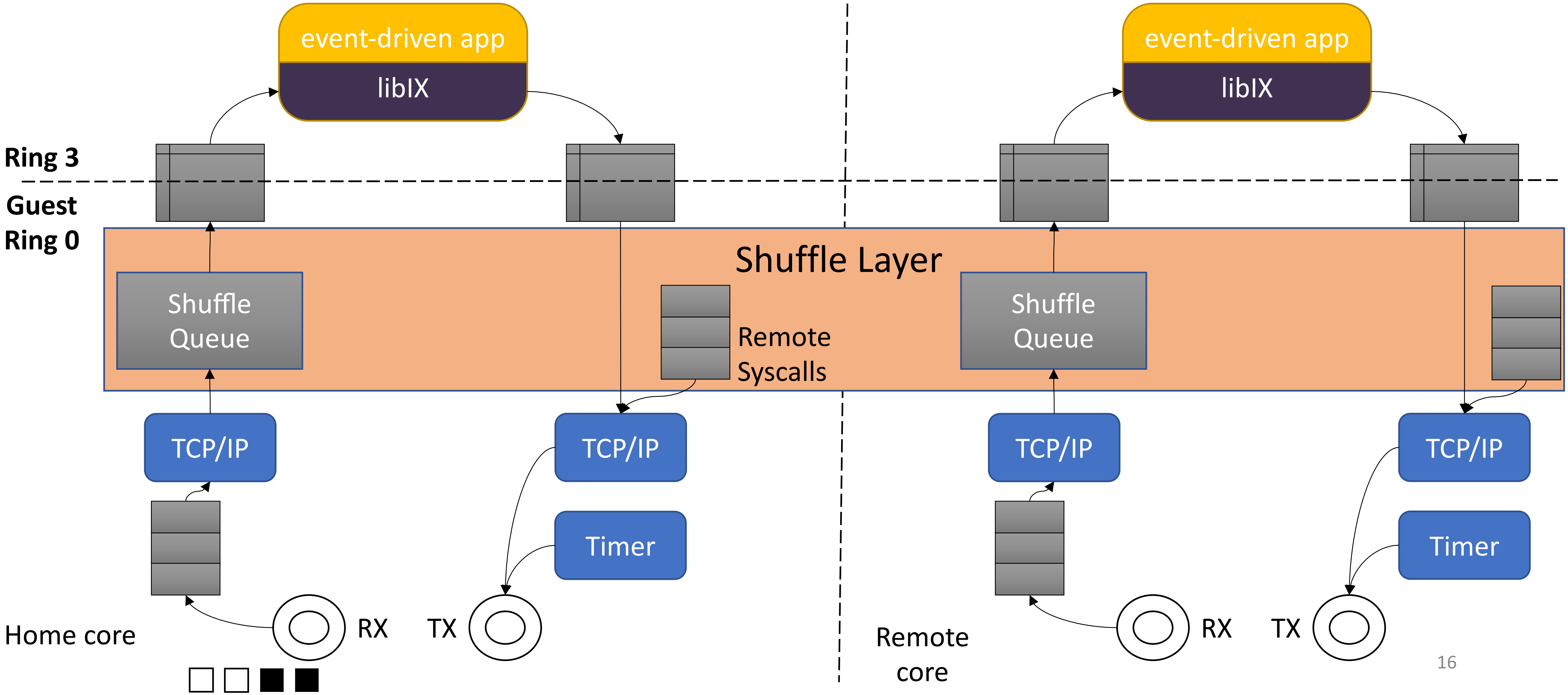
2. Shuffle layer
   Includes a per core list of ready connections that allows stealing

3. Network layer
   Coherence- and sync-free network processing

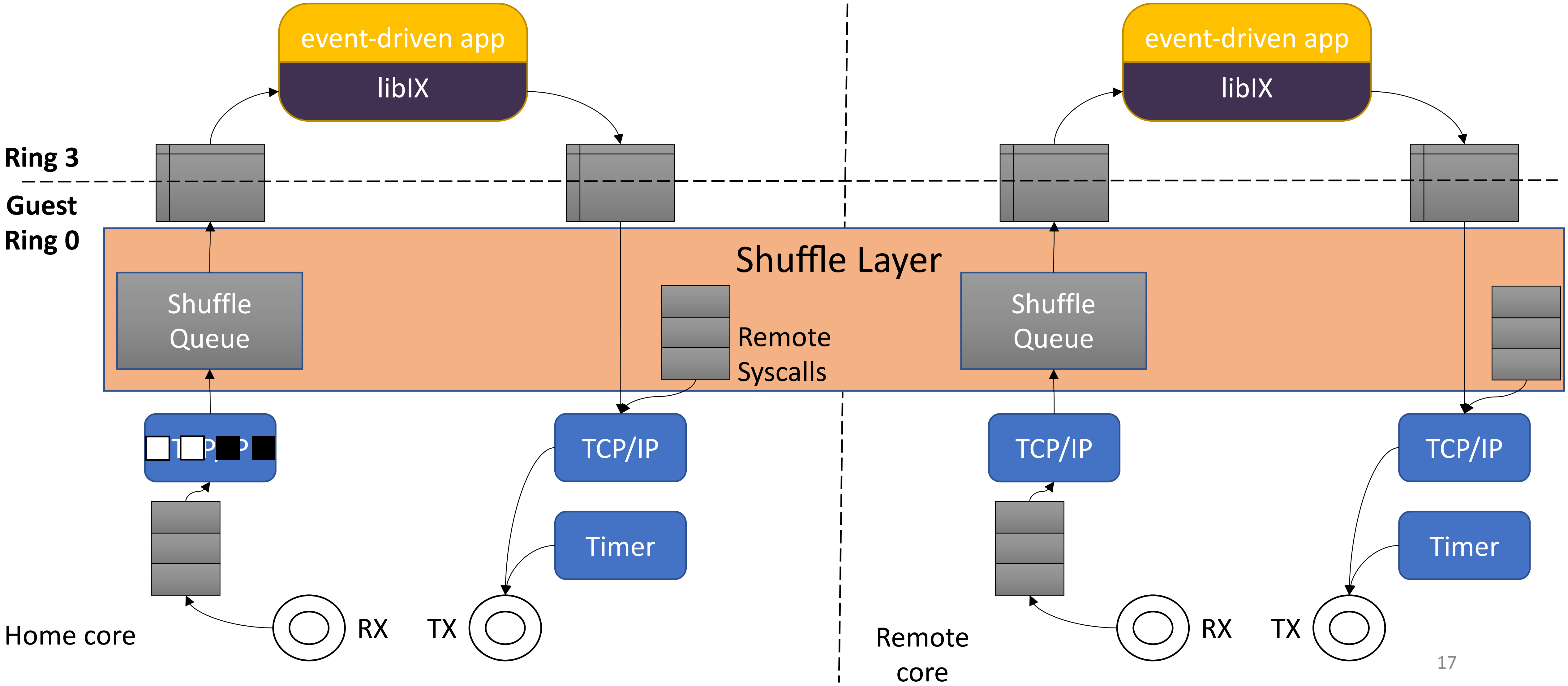
ZygOS Architecture



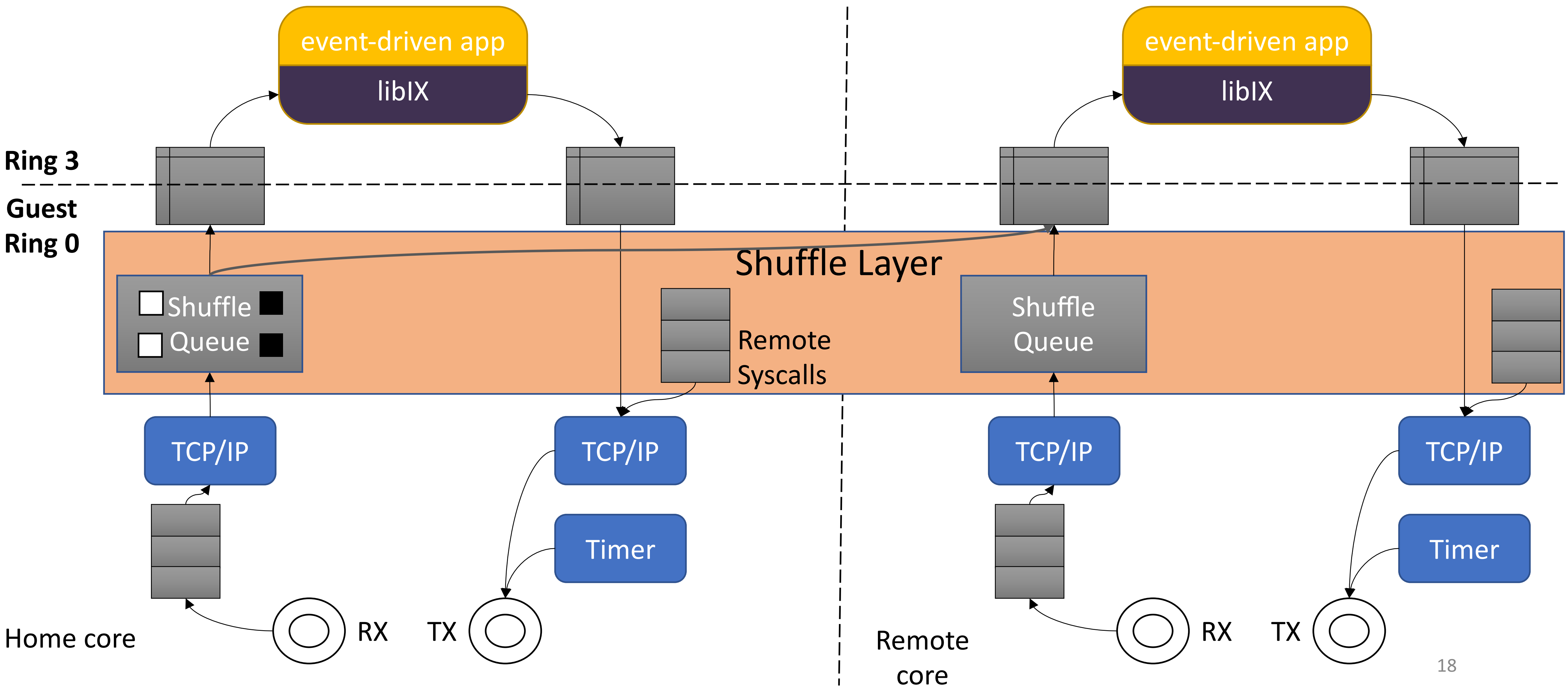
Execution Model



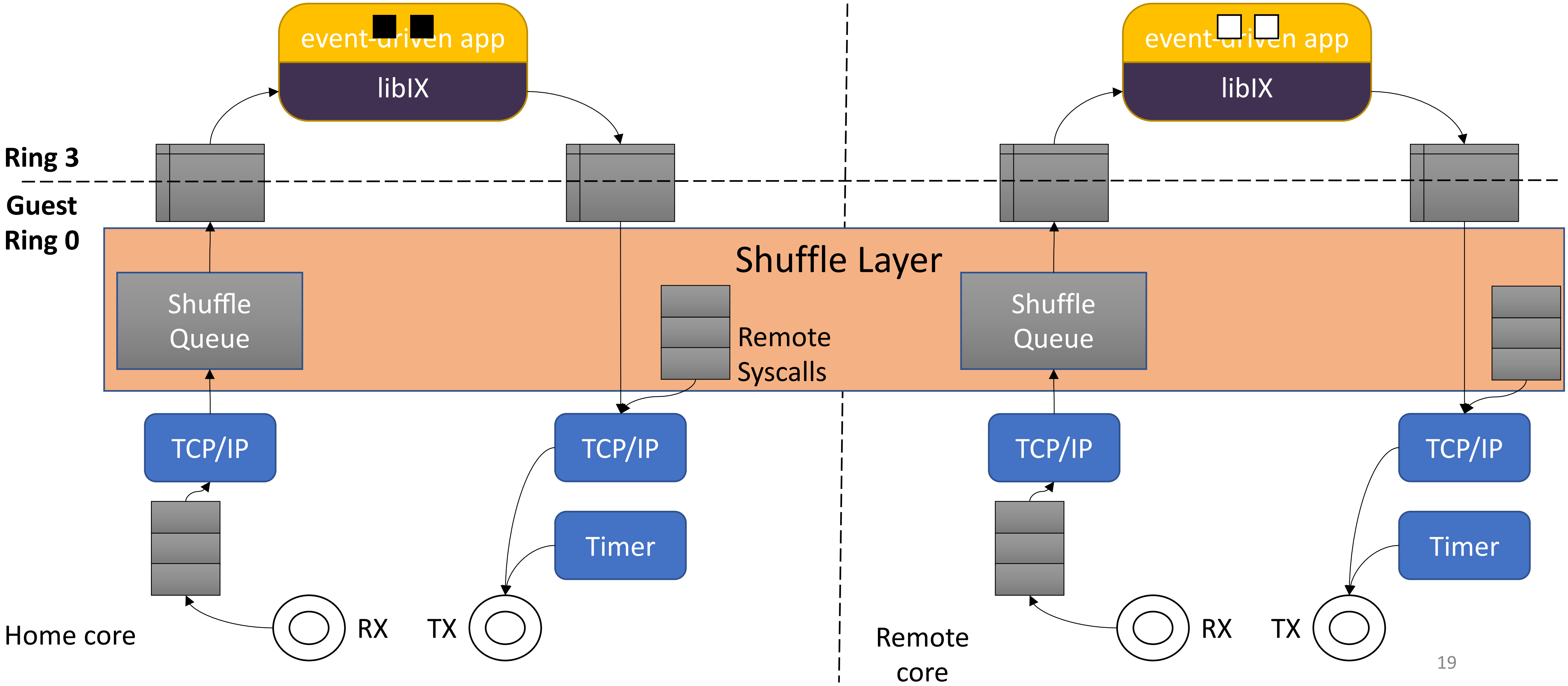
Execution Model



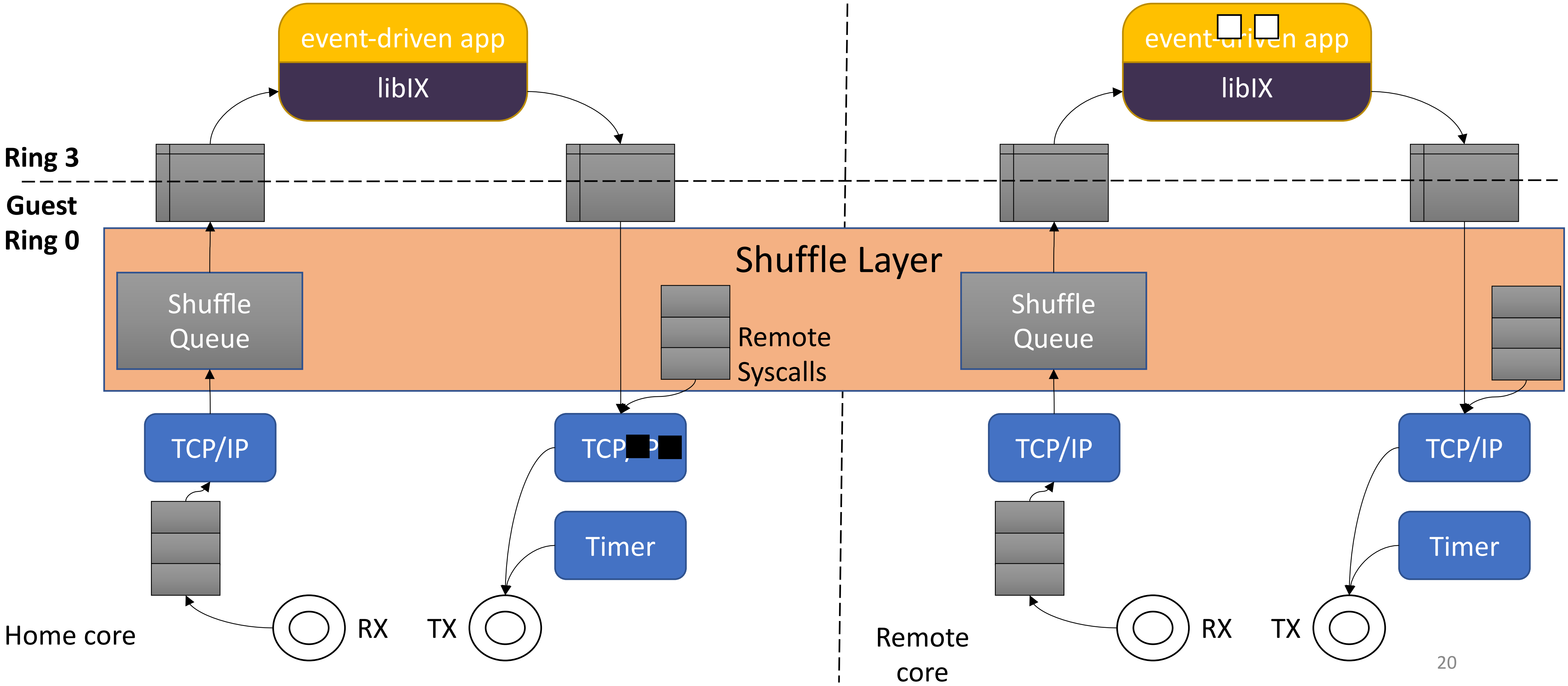
Execution Model



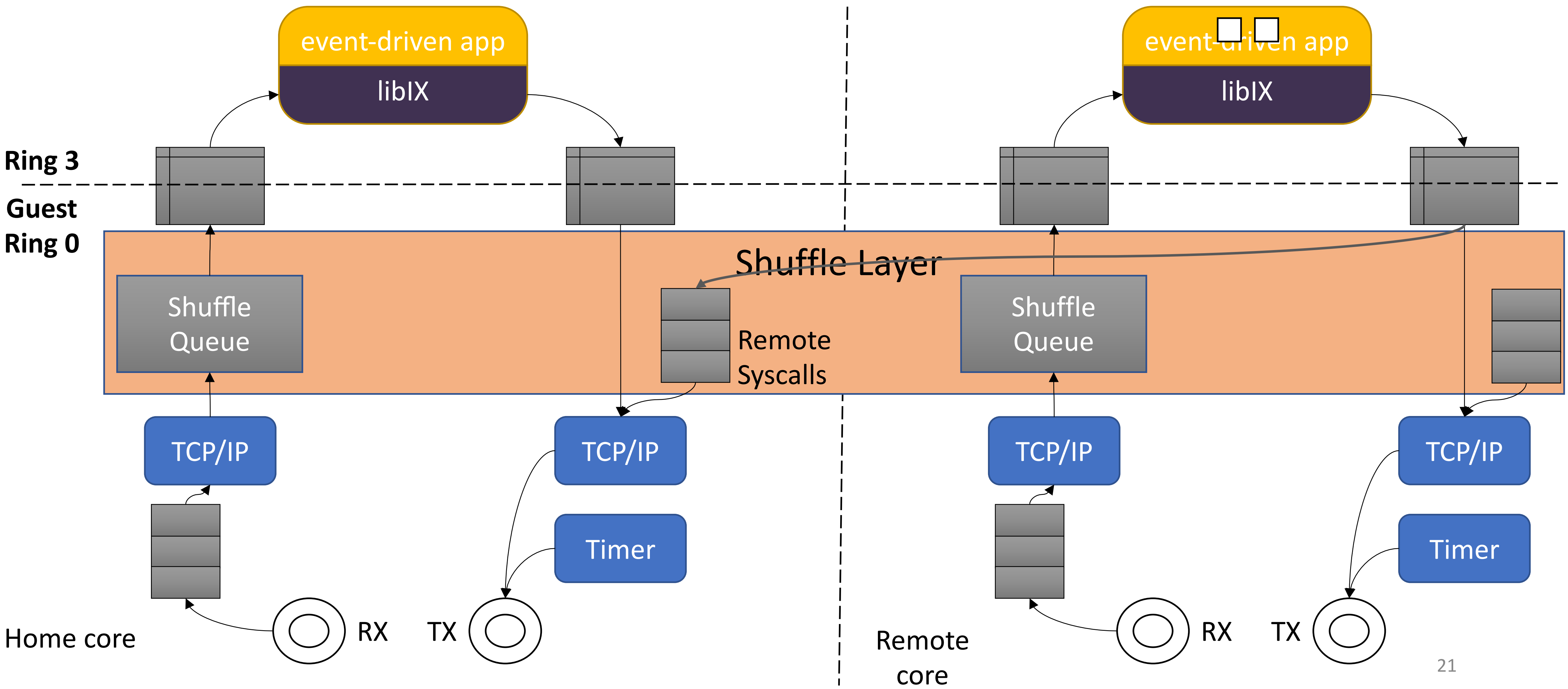
Execution Model



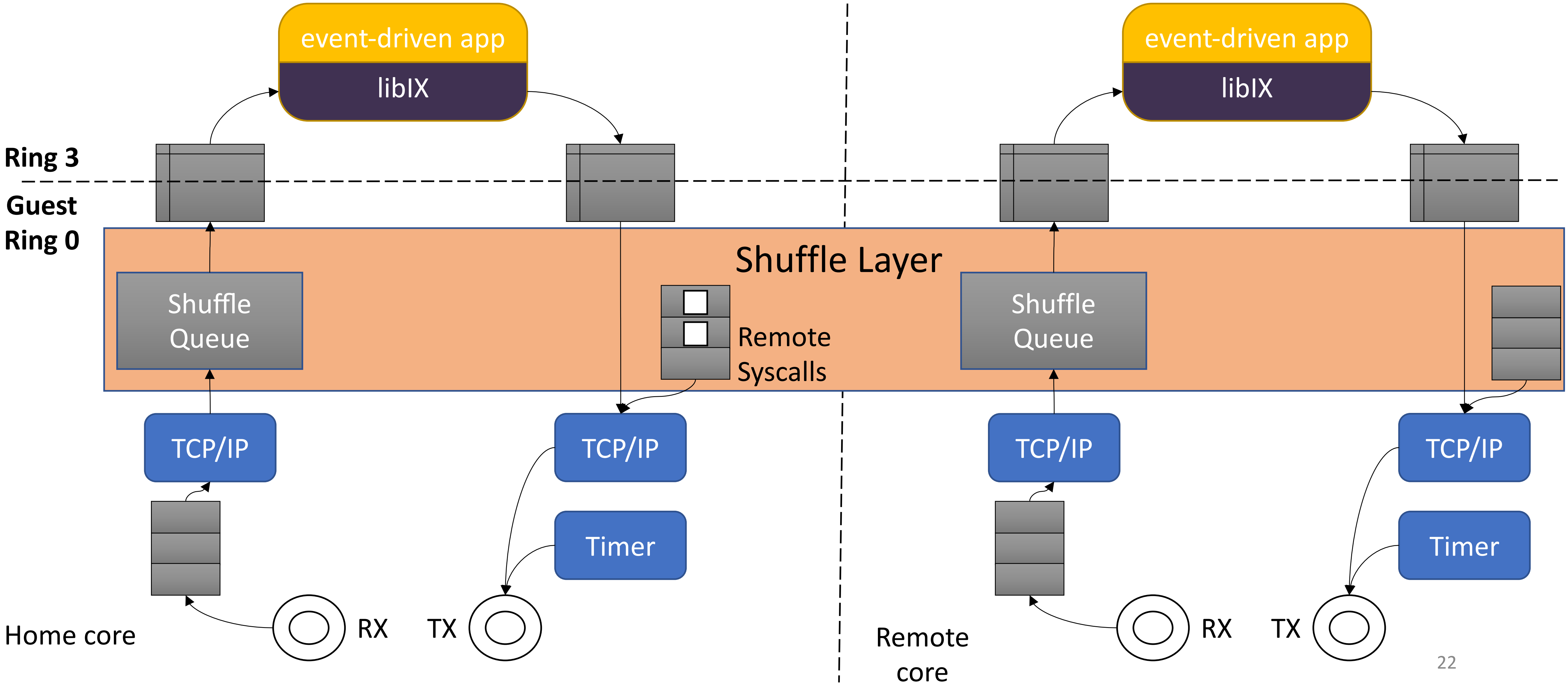
Execution Model



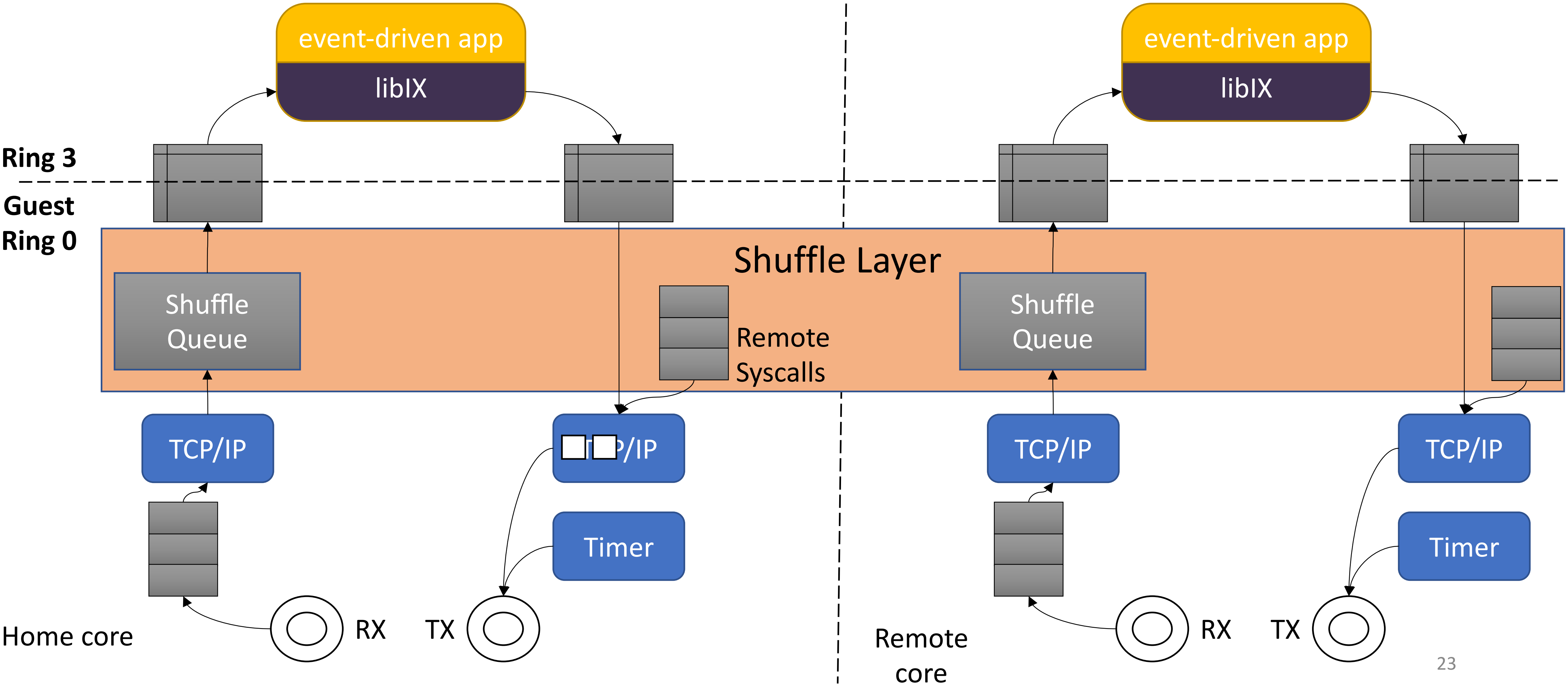
Execution Model



Execution Model



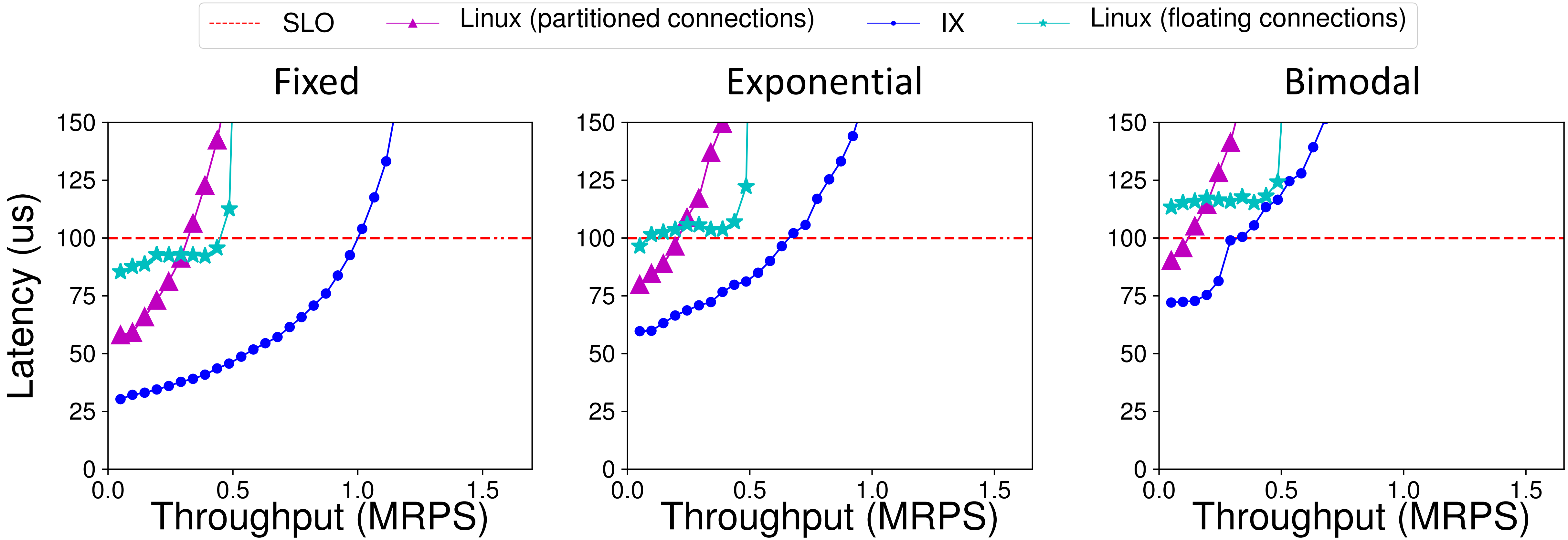
Execution Model



Evaluation Setup

- Environment:
  - 10+1 Xeon Servers
  - 16-hyperthread server machine
  - Quanta/Cumulus 48x10GbE switch
- Experiments:
  - Synthetic micro-benchmarks
  - Silo [SOSP 2013]
  - Memcached
- Baselines:
  - IX
  - Linux (partitioned and floating connections)

Latency vs Load – Service Time 10µs

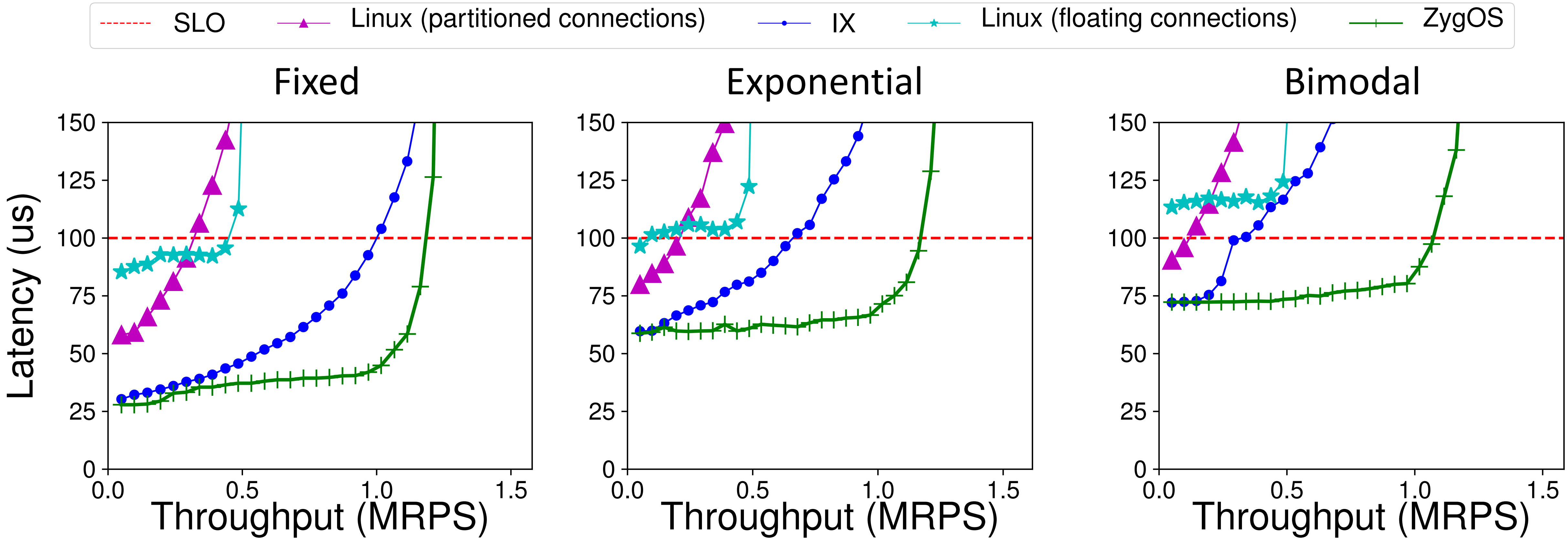


99th percentile latency

SLO: 10 x AVG[service_time]

IX, Belay et al. OSDI 2014

Latency vs Load – Service Time 10μs

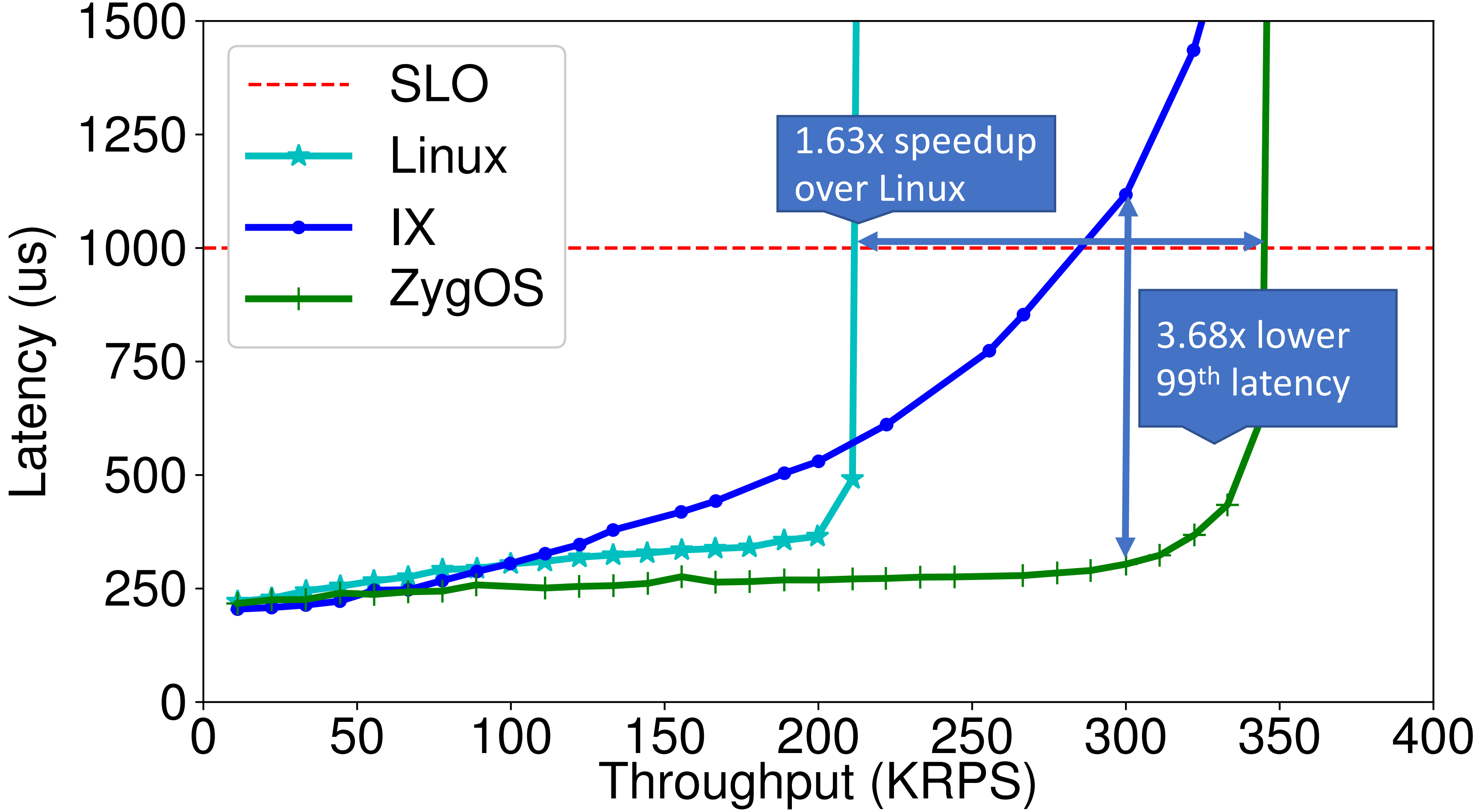


99th percentile latency

SLO: 10 x AVG[service_time]

IX, Belay et al. OSDI 2014

Silo with TPC-C workload



Conclusion

Fork me on GitHub

ZygOS: A datacenter operating system for low-latency

- Reduced System overheads
- Converges to a single queue model
- Work conservation through work stealing
- Reduce HOL through light-weight IPIs

We ♥ opensource

https://github.com/ix-project/zygos

Scheduling in Modern Computer Systems

- FCFS
  - SOSP’17 ZygOS
- RR
  - NSDI’19 Shinjuku
- SJF, SRTF, MLFQ
  - NSDI’19 Tiresias
- Fairness
  - NSDI’11 DRF
  - NSDI’16 FairRide

Tiresias

A GPU Cluster Manager for Distributed Deep Learning

Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin,

Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang (Harry) Liu, Chuanxiong Guo

UNIVERSITY OF MICHIGAN

Microsoft ByteDance UNIST ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY Alibaba Group

GPU Cluster for Deep Learning Training

- Deep learning (DL) is popular
  - *10.5*× increase of DL training jobs in Microsoft
  - DL training jobs require GPU
    - Distributed deep learning (DDL) training with multiple GPUs
- GPU cluster for DL training
  - *5*× increase of GPU cluster scale in Microsoft [1]

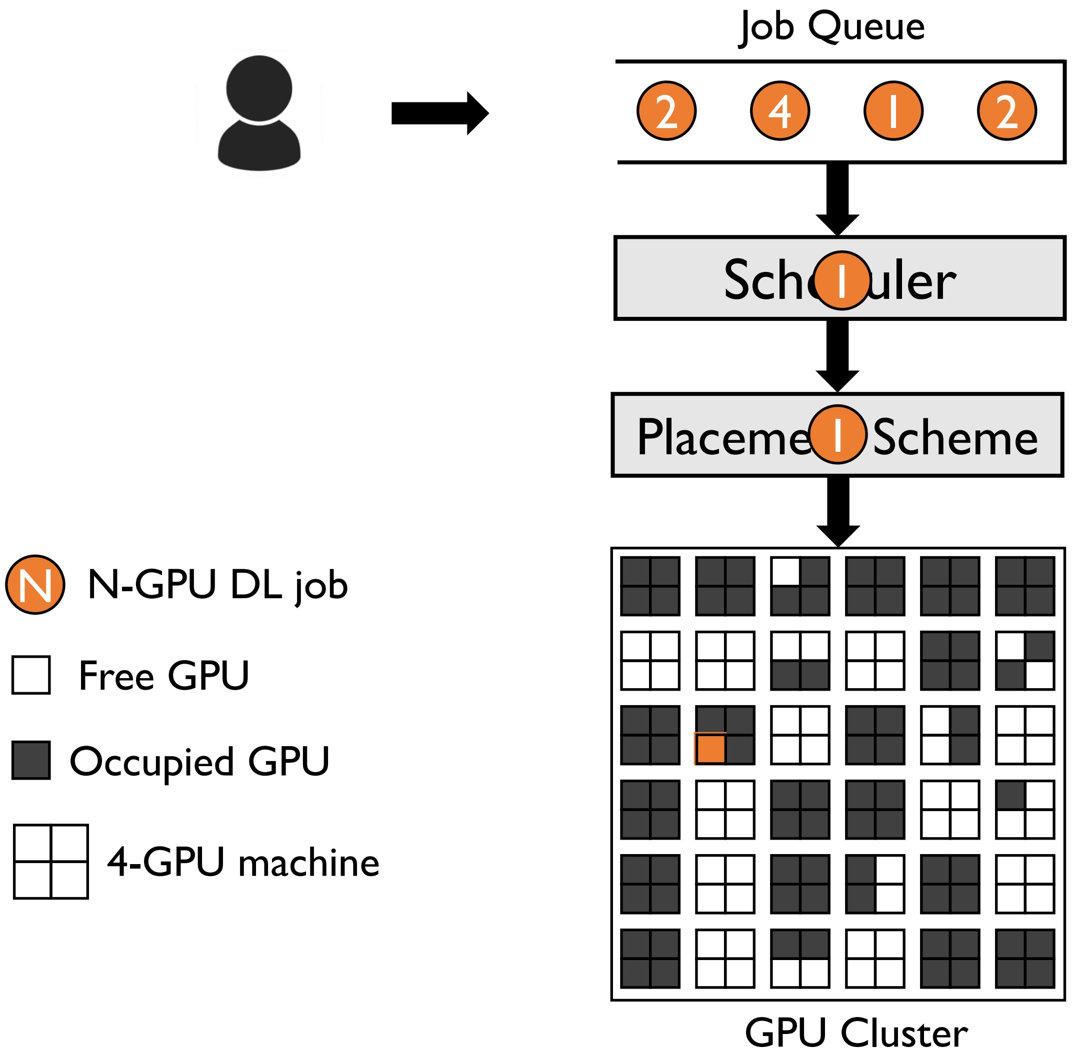
Google Lens

Siri

How to efficiently manage a GPU cluster for DL training jobs?

[1]. Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads. https://arxiv.org/abs/1901.05758

GPU Cluster Manager



Design Objectives

Minimize

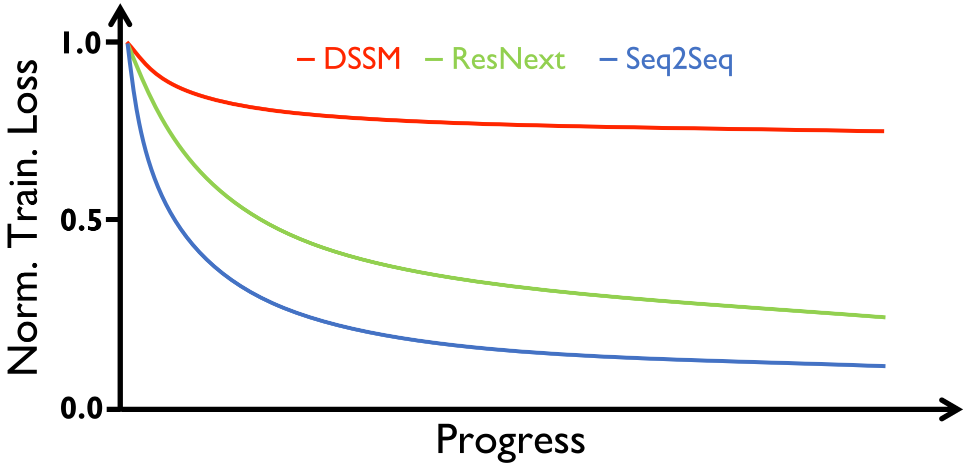
Cluster-Wide Average Job Completion Time (JCT)

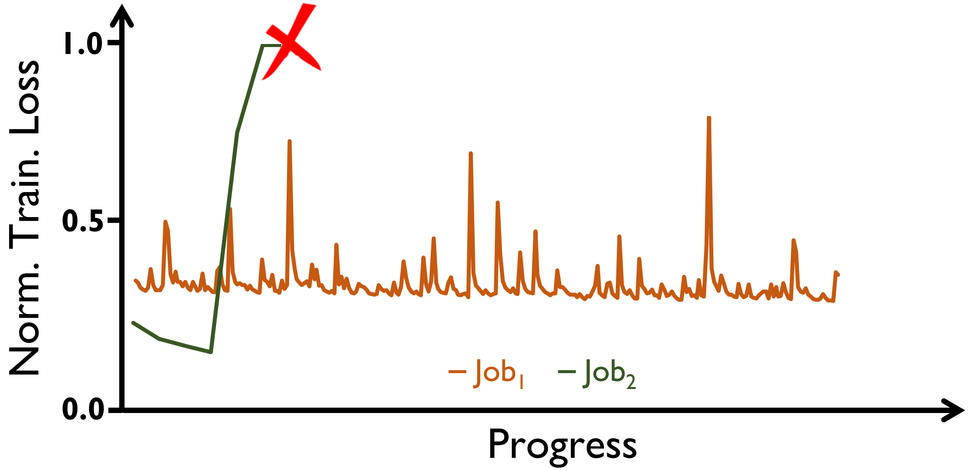
Achieve

High Resource (GPU) Utilization

Challenge 1: Unpredictable Training Time

- Unknown execution time of DL training jobs
  - Job execution time is useful when minimizing JCT
- Predict job execution time
  - Use the smooth loss curve of DL training jobs (*Optimus* [1])



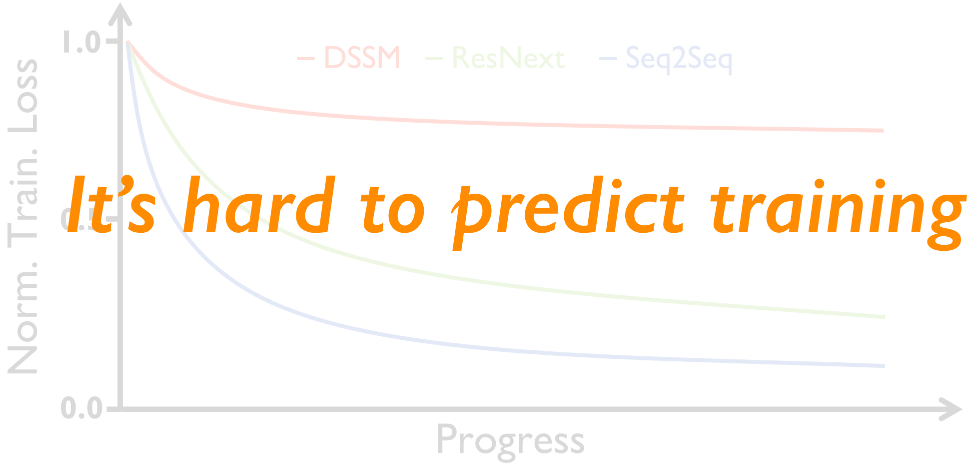


[1]. Optimus: An Efficient Dynamic Resource Scheduler for Deep Learning Clusters, EuroSys'18

Challenge 1: Unpredictable Training Time

- Unknown execution time of DL training jobs
  - Job execution time is useful when minimizing JCT

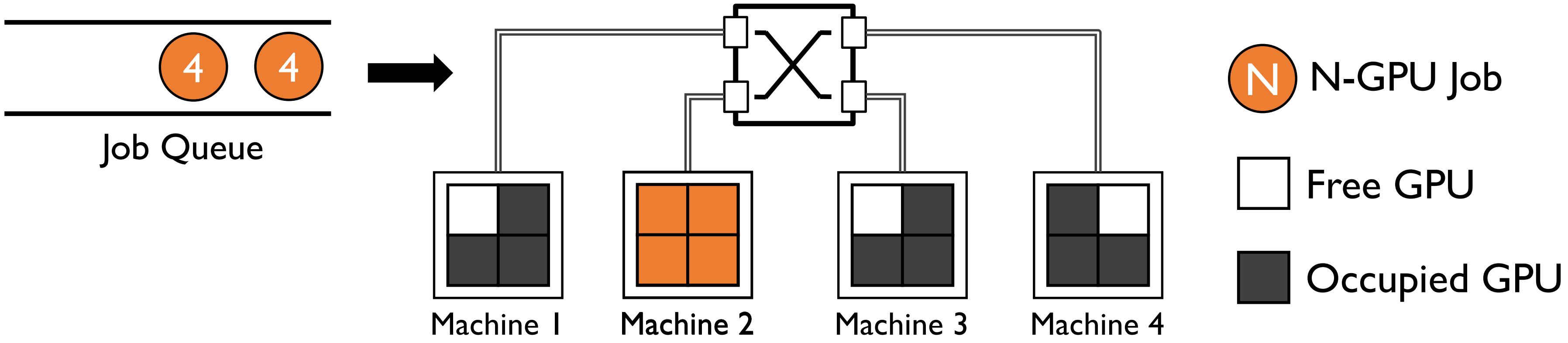
- Predict job execution time
  - Use the smooth loss curve of DL training jobs (*Optimus* [1])



It's hard to predict training time of DL jobs in many cases

Challenge II: Over-Aggressive Job Consolidation

- Network overhead in DDL training
- *Consolidated placement* for good training performance
  - *Fragmented free GPUs in the cluster*
  - *Longer queuing delay*



Prior Solutions

	I. *Unpredictable Training Time* (*Scheduling*)	II. *Over-Aggressive Job Consolidation* (*Job Placement*)
Optimus[1]	None	None
YARN-CS	*FIFO*	None
Gandiva[2]	*Time-sharing*	*Trial-and-error*

[1]. Optimus: An Efficient Dynamic Resource Scheduler for Deep Learning Clusters, EuroSys'18
[2]. Gandiva: Introspective Cluster Scheduling for Deep Learning, OSDI'18

Tiresias

A GPU cluster manager for Distributed Deep Learning Without Complete Knowledge

1. Age-Based Scheduler

   Minimize JCT without complete knowledge of jobs

2. Model Profile-Based Placement

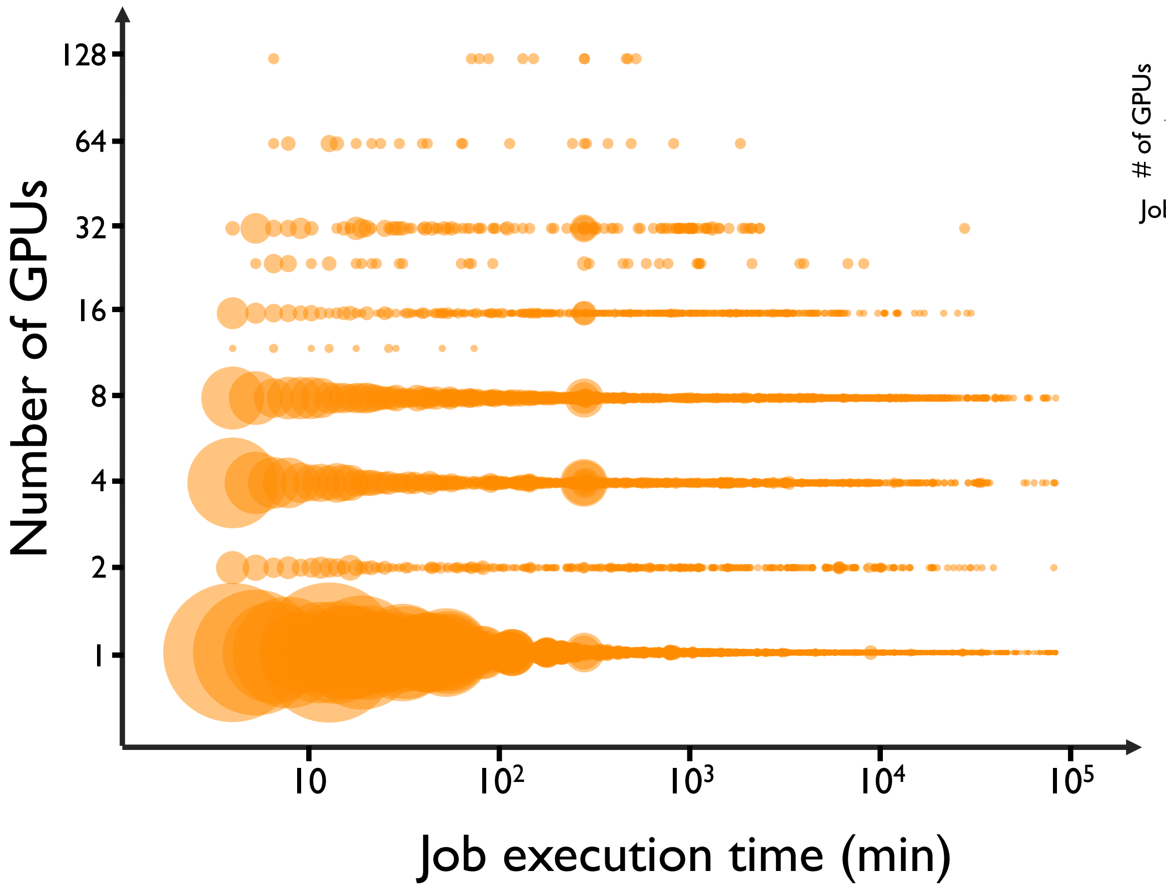
   Place jobs without additional information from users

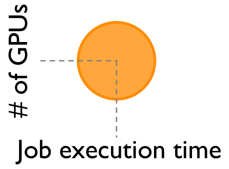
Challenge 1

How To Schedule DL Training Jobs Without Complete Job Information?

Characteristics of DL Training Jobs

- Variations in both temporal and spatial aspects

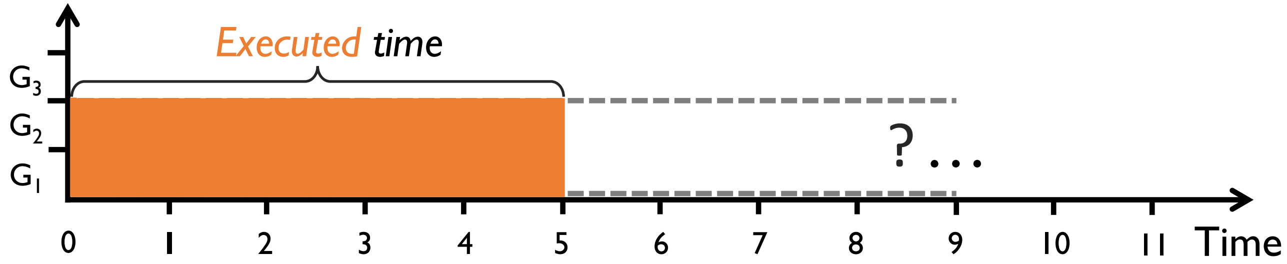




Scheduler should consider both ***temporal and spatial*** *aspects of DL training jobs*

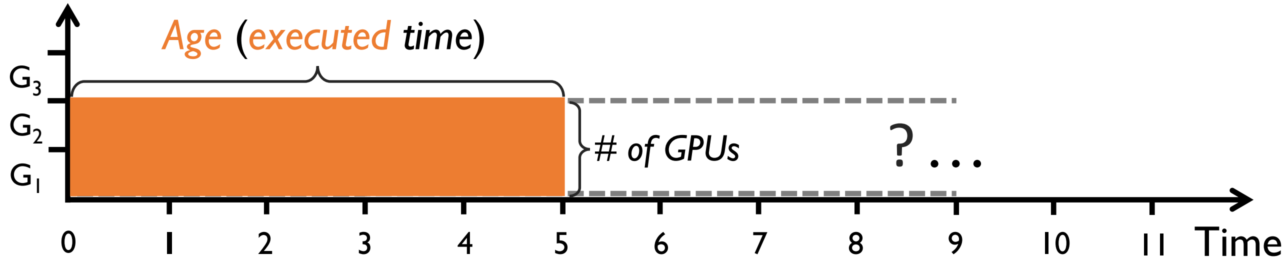
Available Job Information

1. Spatial: number of GPUs
2. Temporal: *executed* time



Age-Based Schedulers

- ***Least-Attained Service*** [1] (LAS)
  - Prioritize job that has the shortest executed time



[1]. Feedback queueing models for time-shared systems. JACM, 1968
[2]. Multi-armed bandit allocation indices. Wiley, Chichester, 1989

Two-Dimensional Age-Based Scheduler (2DAS)

- Age calculated by two-dimensional attained service
  - i.e., a job's ***total executed GPU time*** (# of GPUs × executed time)

- No prior information
  - *2D-LAS*

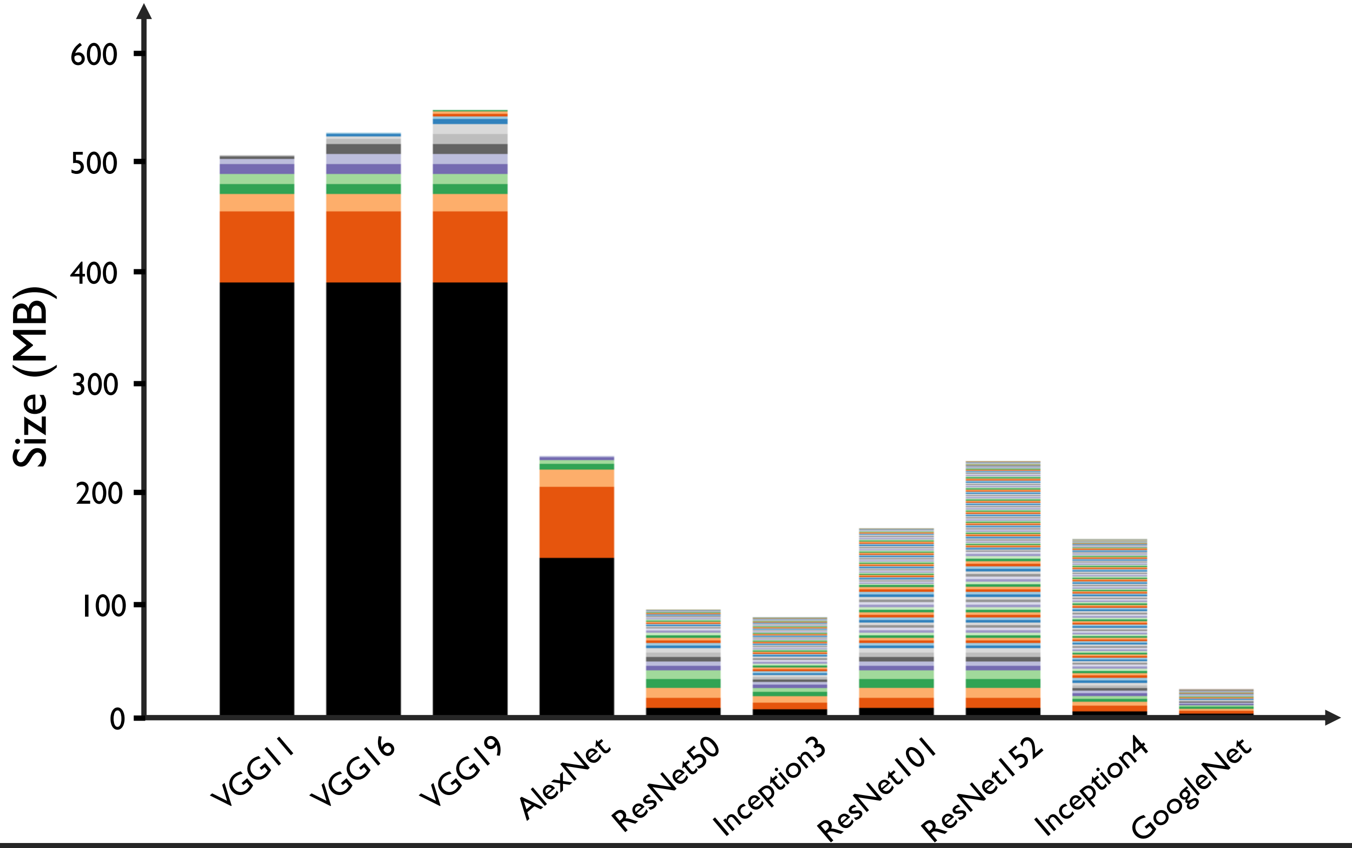
Fewer Job Switches: Discretized 2D-LAS (MLFQ)

Challenge II

How to Place DL Jobs Without Hurting Training Performance?

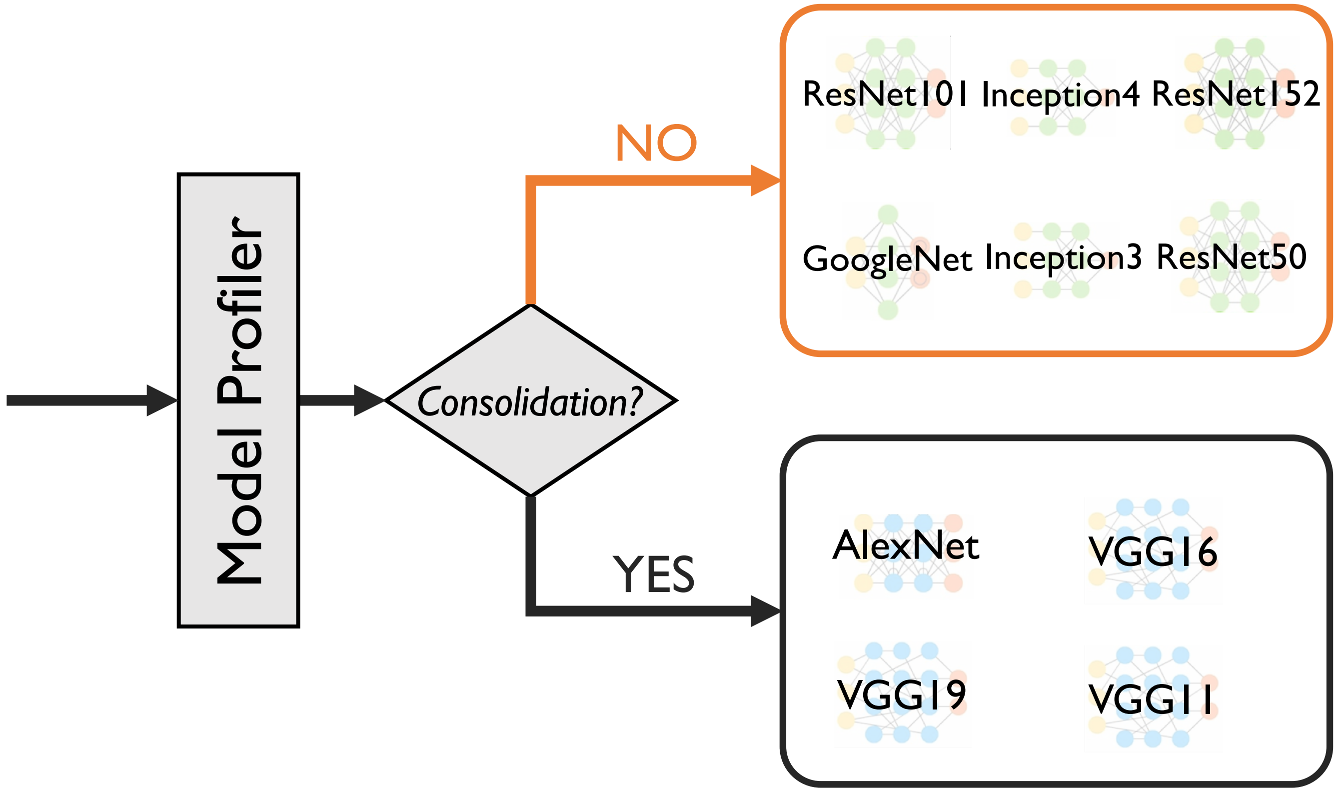
Characteristics of DL Models

- Tensor size in DL models
  - *Large tensors* cause network imbalance and contention



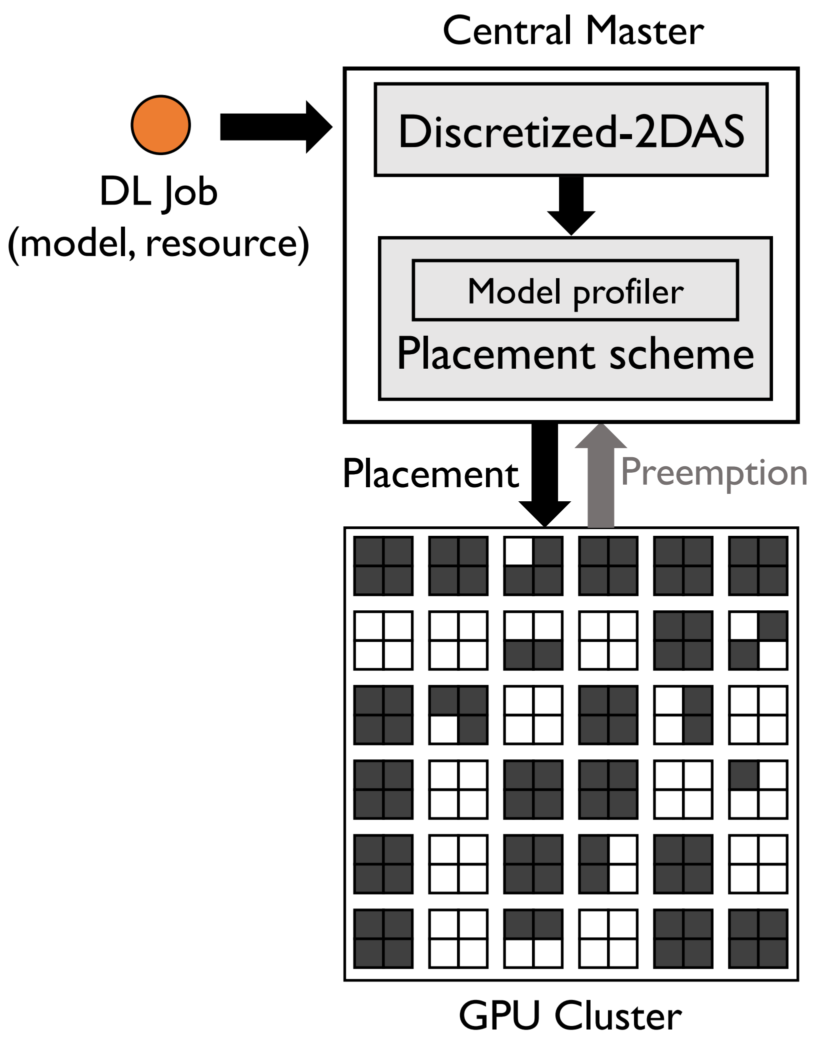
Consolidated placement *is needed when the model is* ***highly skewed*** *in its tensor size*

Model Profile-Based Placement



Tiresias

Central Master
Network-Level Model Profiler



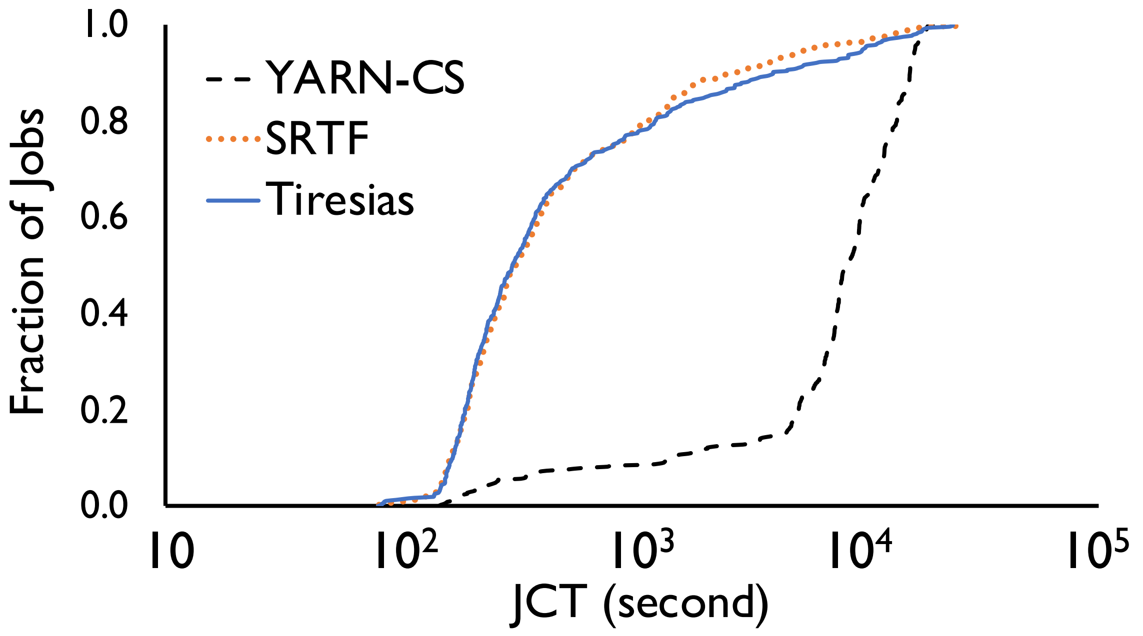
Evaluation

60-GPU
Testbed Experiment

Large-scale &
Trace-driven Simulation

JCT Improvements in Testbed Experiment

- Testbed – Michigan ConFlux cluster
  - 15 machines (4 GPUs each)
  - 100 Gbps RDMA network

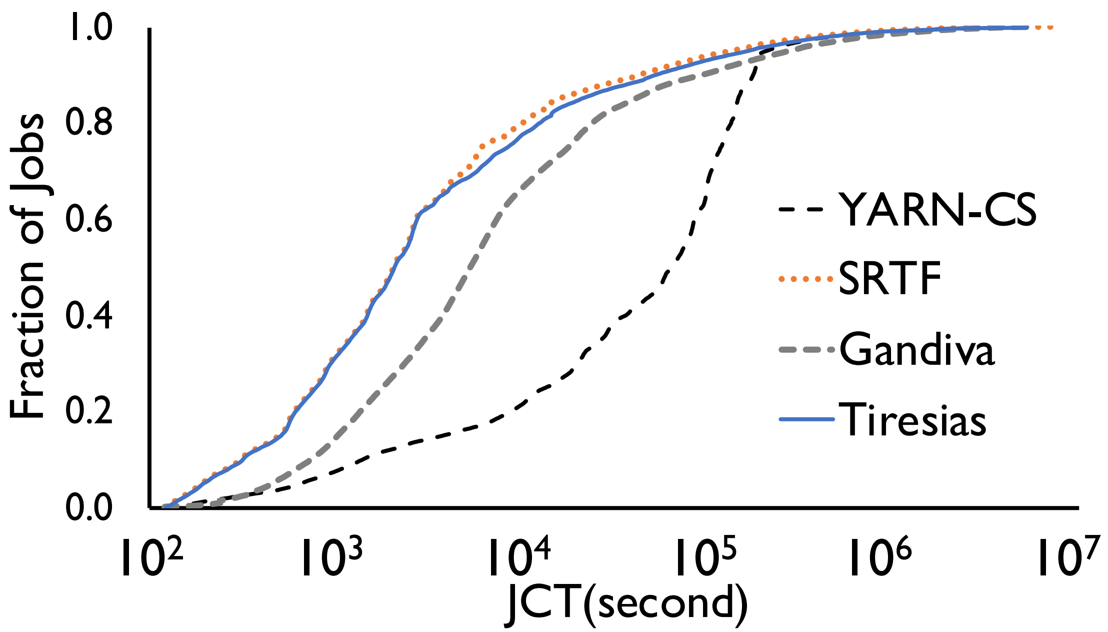


Avg. JCT improvement (w.r.t. YARN-CS): 5.5×

Comparable performance to SRTF

JCT Improvements in Trace-Driven Simulation

- Discrete-time simulator
  - 10-week job trace from Microsoft
  - 2,000-GPU cluster



Avg. JCT improvement (w.r.t. Gandiva): 2×

Tiresias

A GPU cluster manager for Distributed Deep Learning Without Complete Knowledge

- Optimize JCT with no or partial job information
- Relax placement constraint without hurting training performance
- Simple, practical, and with significant performance improvements

UNIVERSITY OF MICHIGAN

https://github.com/SymbioticLab/Tiresias

Scheduling in Modern Computer Systems

- FCFS
  - SOSP’17 ZygOS
- RR
  - NSDI’19 Shinjuku
- SJF, SRTF, MLFQ
  - NSDI’19 Tiresias
- Fairness
  - NSDI’11 DRF
  - NSDI’16 FairRide

Dominant Resource Fairness (DRF)

Fair Allocation of Multiple Resource Types

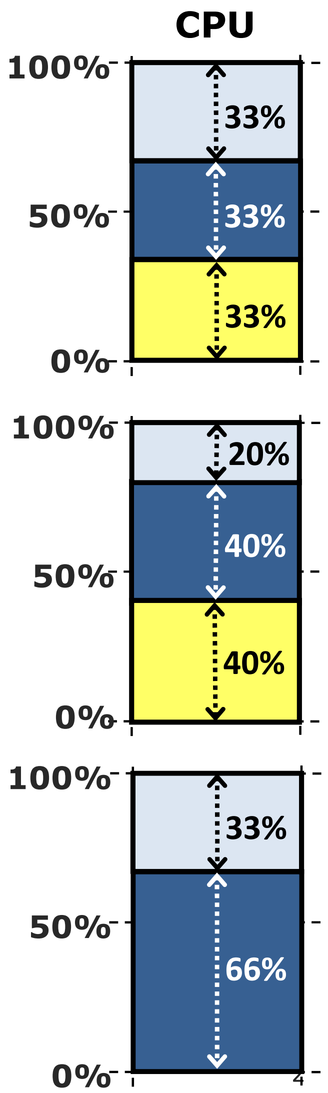
Ali Ghodsi, Matei Zaharia
Benjamin Hindman, Andy Konwinski,
Scott Shenker, Ion Stoica

University of California, Berkeley

alig@cs.berkeley.edu

What is fair sharing?

- n users want to share a resource (e.g. CPU)
  - Solution:
    Allocate each 1/n of the shared resource
- *Generalized by max-min fairness*
  - Handles if a user wants less than its fair share
  - E.g. user 1 wants no more than 20%
- Generalized by *weighted max-min fairness*
  - Give weights to users according to importance
  - User 1 gets weight 1, user 2 weight 2



How to define fairness?

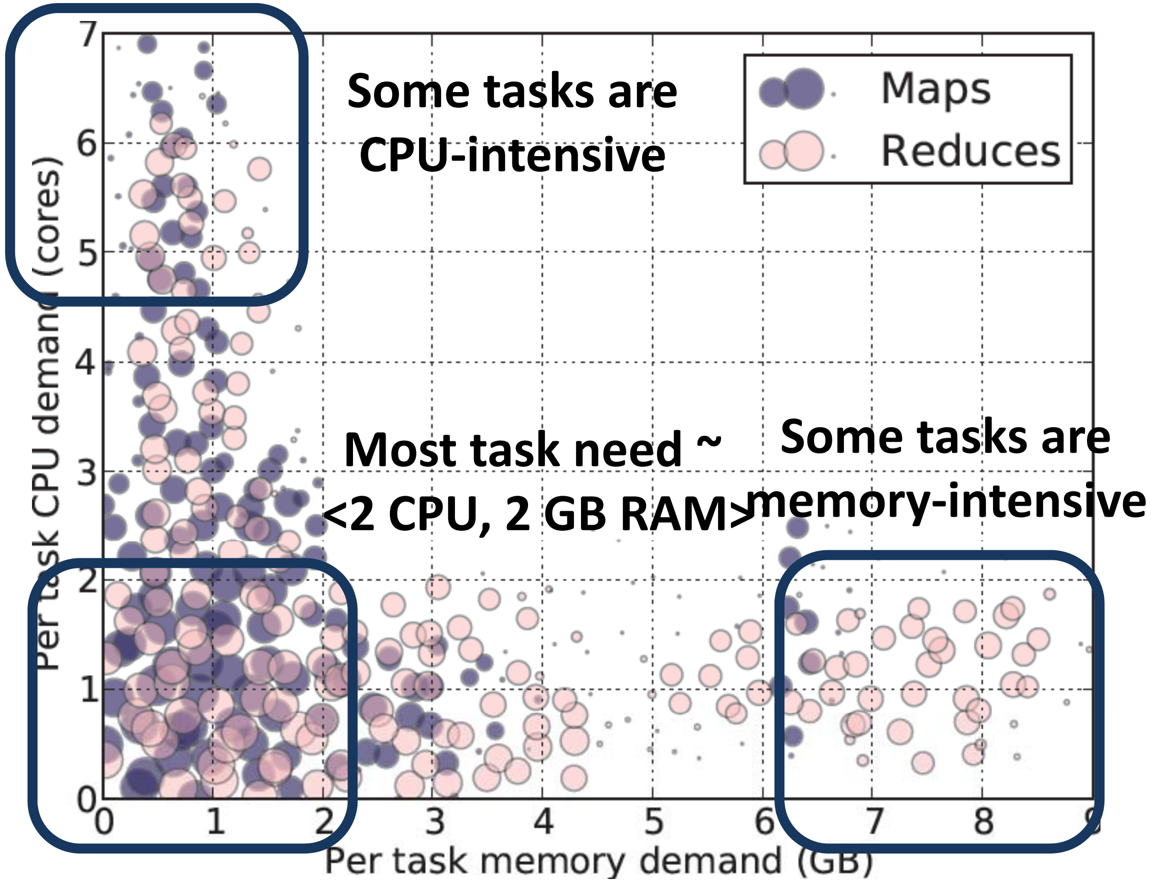
- Share guarantee
  - Each user can get at least 1/n of the resource
  - But will get less if her demand is less
- Stragegy-proof
  - Users are not better off by asking for more than they need
  - Users have no reason to lie
- Pareto efficiency
  - It is not possible to increase the utility of a user without decreasing the utility of at least another user
  - It leads to maximizing system utilizaiton subject to satisfying other constraints

Why is max-min fairness not enough?

- Job scheduling in datacenters is not only about CPUs
  - Jobs consume CPU, memory, disk, and I/O

- Does this pose any challenge?

Heterogeneous Resource Demands

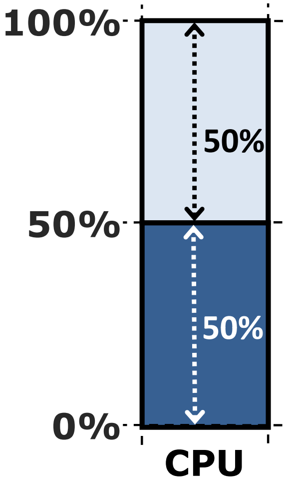


2000-node Hadoop Cluster at Facebook (Oct 2010)

Problem

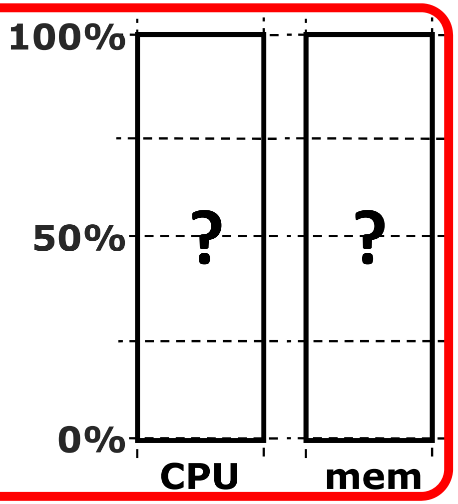
Single resource example

- 1 resource: CPU
- User 1 wants **<1 CPU>** per task
- User 2 wants **<3 CPU>** per task



Multi-resource example

- 2 resources: CPUs & mem
- User 1 wants **<1 CPU, 4 GB>** per task
- User 2 wants **<3 CPU, 1 GB>** per task
- ***What's a fair allocation?***



Problem definition

How to fairly share multiple resources when users have heterogenous demands on them?

Model

- Users have *tasks* according to a *demand vector*
  - e.g. **<2, 3, 1>** user's tasks need 2 R_1, 3 R_2, 1 R_3
  - Not needed in practice, measure actual consumption
- Resources given in multiples of demand vectors
- Assume divisible resources

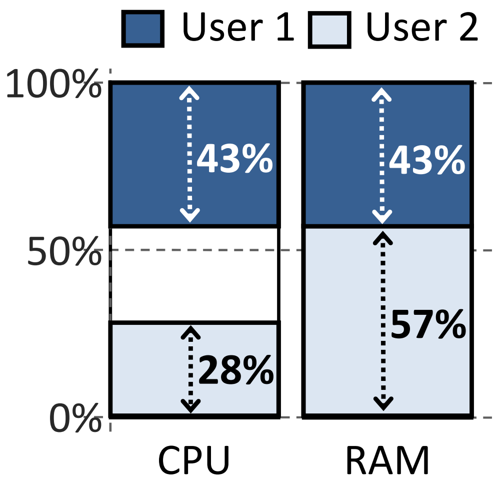
A Natural Policy

- *Asset Fairness*
  - Equalize each user's *sum of resource shares*

- Cluster with 70 CPUs, 70 GB RAM
  - U_1 needs <2 CPU, 2 GB RAM> per task
  - U_2 needs <1 CPU, 2 GB RAM> per task

A Natural Policy

- *Asset Fairness*
  - Equalize each user's *sum of resource shares*



Problem

User 1 has < 50% of both CPUs and RAM

Better off in a separate cluster with 50% of the resources

- Asset fairness yields
  - U_1: 15 tasks: 30 CPUs, 30 GB ($\sum$=60)
  - U_2: 20 tasks: 20 CPUs, 40 GB ($\sum$=60)

Dominant Resource Fairness

- A user's *dominant resource* is the resource she has the biggest share of
  - Example:

    Total resources: **<10 CPU, 4 GB>**

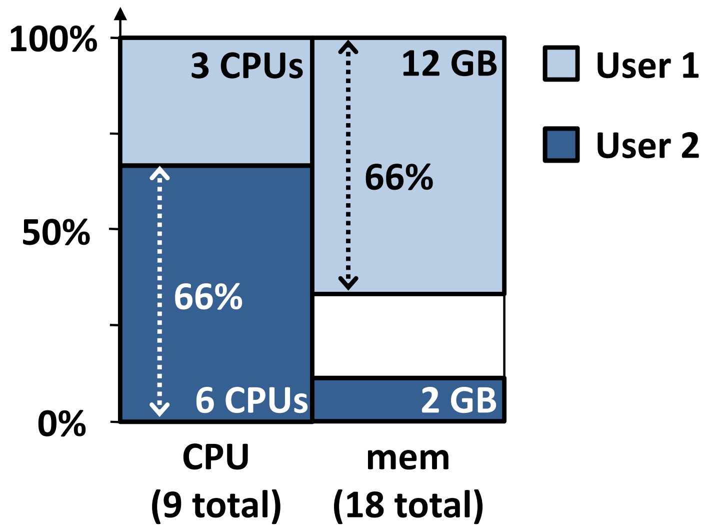
    User 1's allocation: **<2 CPU, 1 GB>**

    Dominant resource is memory as 1/4 > 2/10 (1/5)

- A user's *dominant share* is the fraction of the dominant resource she is allocated
  - User 1's dominant share is **25%** (1/4)

Dominant Resource Fairness (2)

- ***Apply max-min fairness to dominant shares***
- Equalize the dominant share of the users
  - Example:
    Total resources: **<9 CPU, 18 GB>**
    User 1 demand: **<1 CPU, 4 GB>** dom res: **mem**
    User 2 demand: **<3 CPU, 1 GB>** dom res: **CPU**



How would an economist solve it?

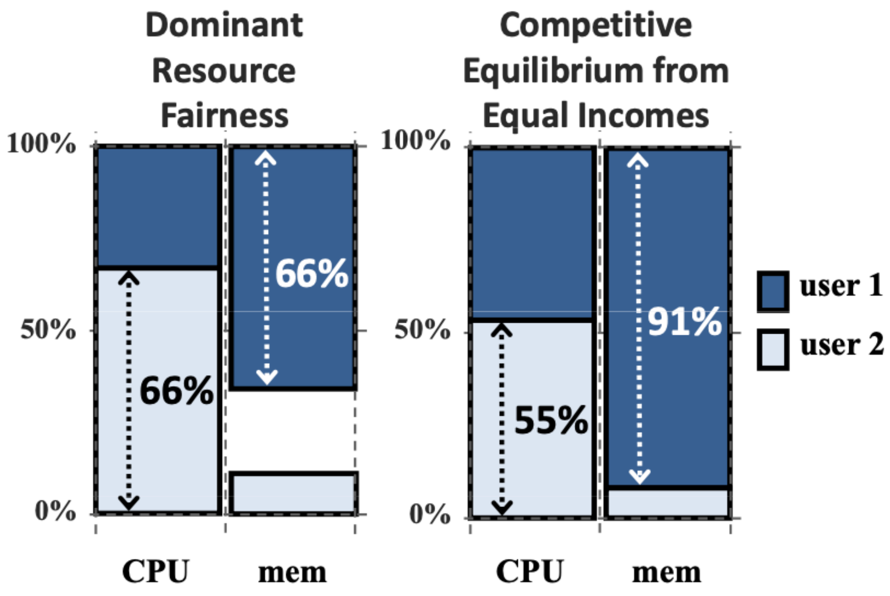
- Let the market determine the prices

- *Competitive Equilibrium from Equal Incomes (CEEI)*
  - Give each user 1/n of every resource
  - Let users trade in a perfectly competitive market

- **Not strategy-proof!**

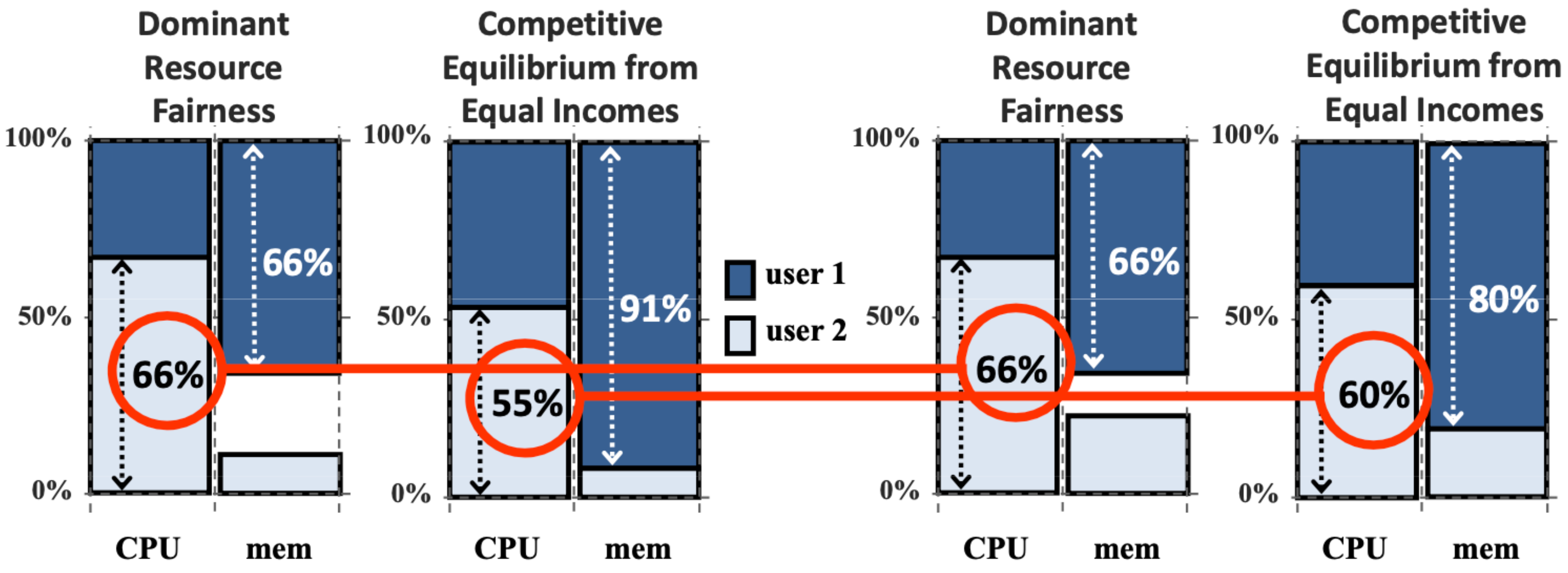
DRF vs CEEI

- User 1: <1 CPU, 4 GB> User 2: <3 CPU, 1 GB>
  - DRF more fair, CEEI better utilization



DRF vs CEEI

- User 1: <1 CPU, 4 GB> User 2: <3 CPU, 1 GB>
  – DRF more fair, CEEI better utilization



- User 1: <1 CPU, 4 GB> User 2: <3 CPU, 2 GB>
  – User 2 increased her share of both CPU and memory

Properties of Policies

Property	Asset	CEEI	DRF
Share guarantee		✔	✔
Strategy-proofness	✔		✔
Pareto efficiency	✔	✔	✔
Envy-freeness	✔	✔	✔
Single resource fairness	✔	✔	✔
Bottleneck res. fairness		✔	✔
Population monotonicity	✔		✔
Resource monotonicity			

Scheduling in Modern Computer Systems

- FCFS
  - SOSP’17 ZygOS
- RR
  - NSDI’19 Shinjuku
- SJF, SRTF, MLFQ
  - NSDI’19 Tiresias
- Fairness
  - NSDI’11 DRF
  - NSDI’16 FairRide

amplab UC BERKELEY

FairRide: Near-Optimal Fair Cache Sharing

Qifan Pu,
Haoyuan Li,
Matei Zaharia,
Ali Ghodsi,
Ion Stoica

Caches are crucial

ALLUXIO

Spark

memCached

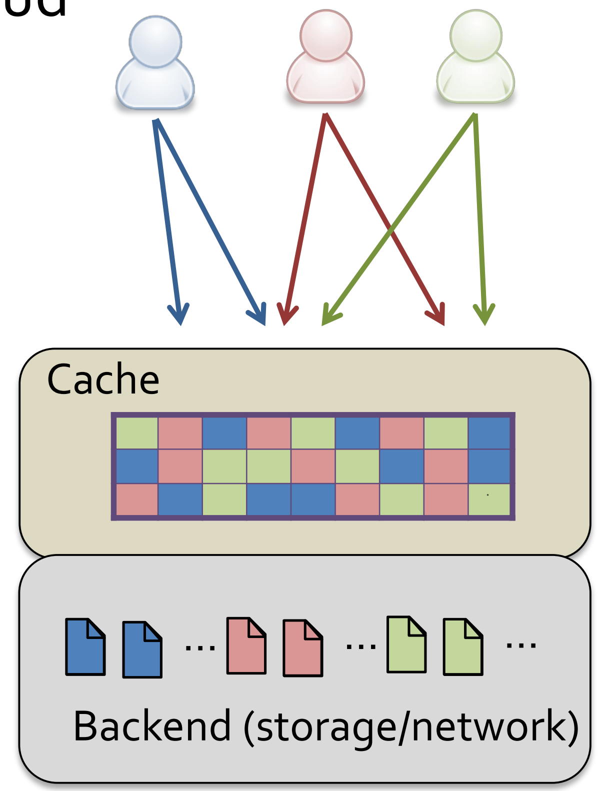
Cache sharing

- Increasingly, caches are shared among multiple users
  - Especially with the advent of cloud

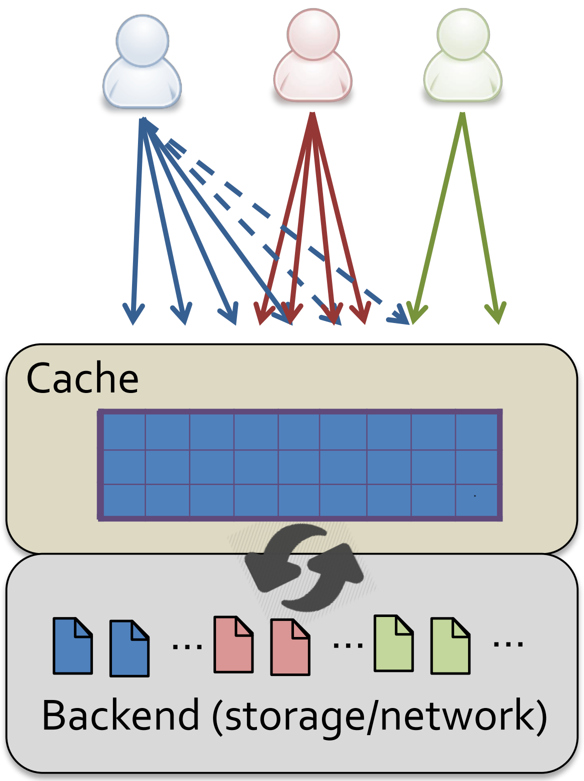
Benefits:

– Provide low latency

– Reduce backend load



Problems with cache algorithms



- LRU, LFU, LRU-K…
  - Cache data likely to be accessed in the future
- Optimize global efficiency
- Single user gets arbitrarily small cache
- Prone to strategic behavior

A simple model

- Users access equal-sized files at constant rates
  - r_{ij} the rate user *i* accesses file *j*

- A allocation **policy** decides which files to cache
  - p_j the % of file *j* put in cache

- Users care their hit ratio $HR_i = \dfrac{total\_hits}{total\_accesses} = \dfrac{\sum_j p_j r_{ij}}{\sum_j r_{ij}}$
  - user *i*'s hit ratio:

◆ Results hold with varied file sizes, access partial files, p_j is binary, etc.

Properties

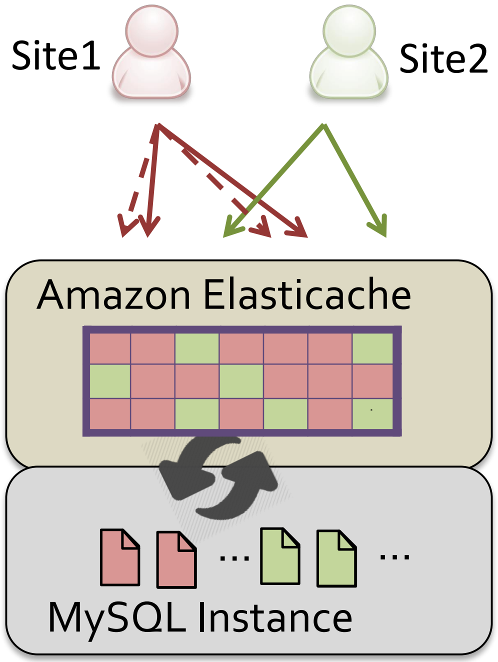
- Isolation Guarantee **(Share Guarantee)**
  – No user should be worse off than static allocation

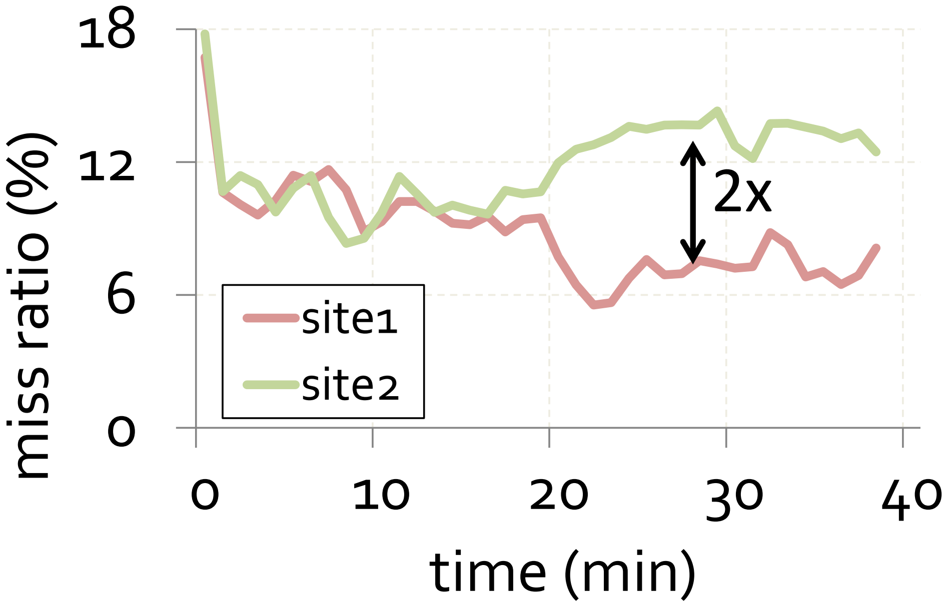
- Strategy-Proofness
  – No user can improve by cheating

- Pareto Efficiency
  – Can't improve a user without hurting others

Strategy proofness

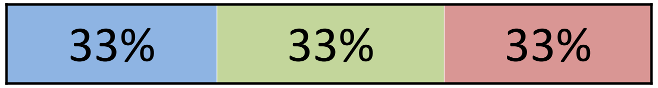
- Very easy to cheat, hard to detect
  – e.g., by making spurious accesses
- Can happen in practice



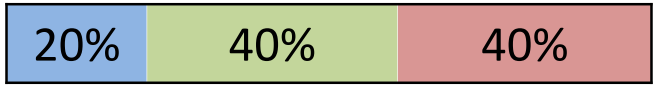


What is *max-min fairness*?

- *Maximize* the the user with *minimum* allocation
  - Solution: allocate each **1/n** (fair share)

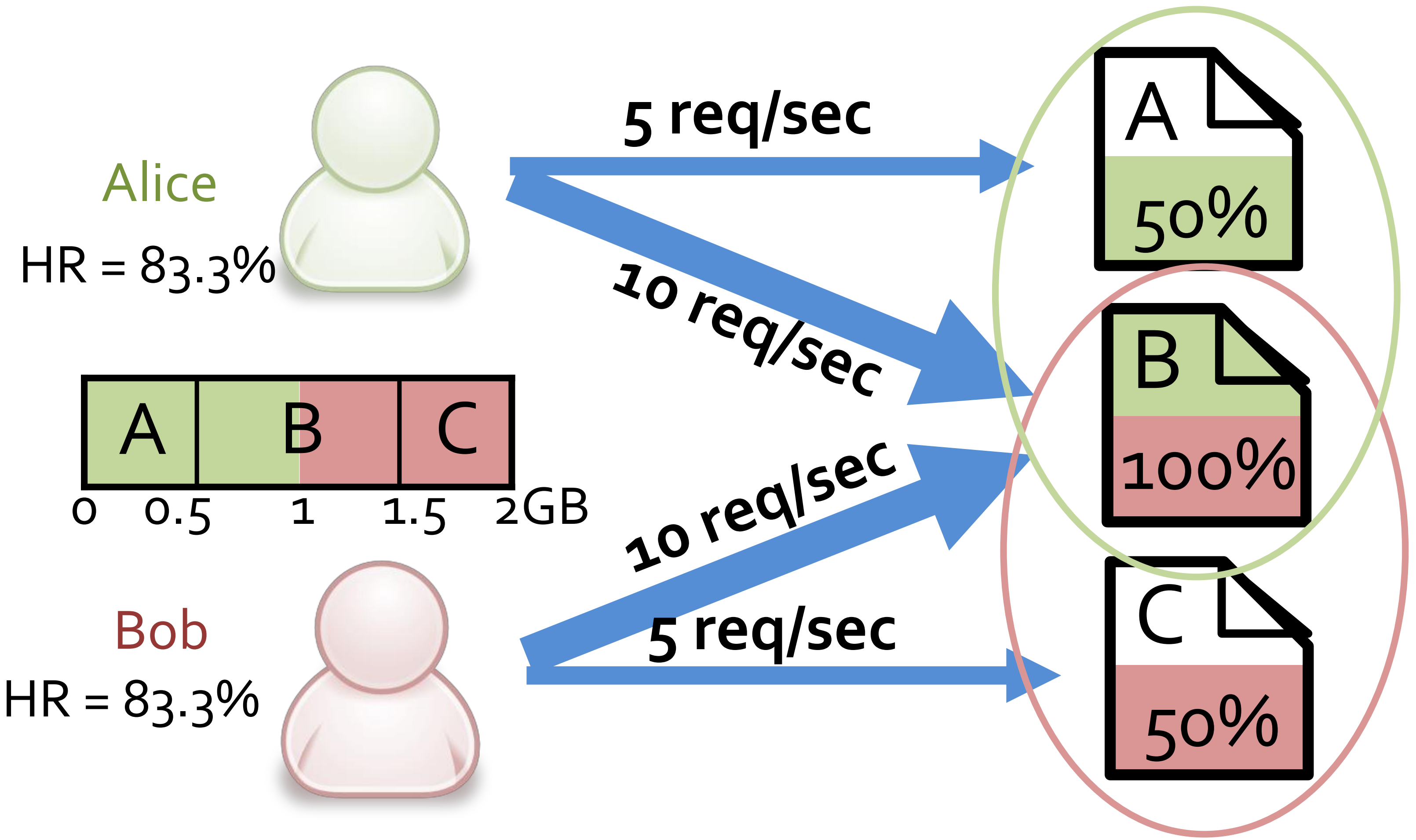


  - Handles if some users want less than fair share



- Widely successful to other resources:
  - OS: round robin, prop sharing, lottery sched…
  - Networking: fair queueing, wfq, wf2q, csfq, drr…
  - Datacenter: DRF, Hadoop fair sched, Quincy…

An example

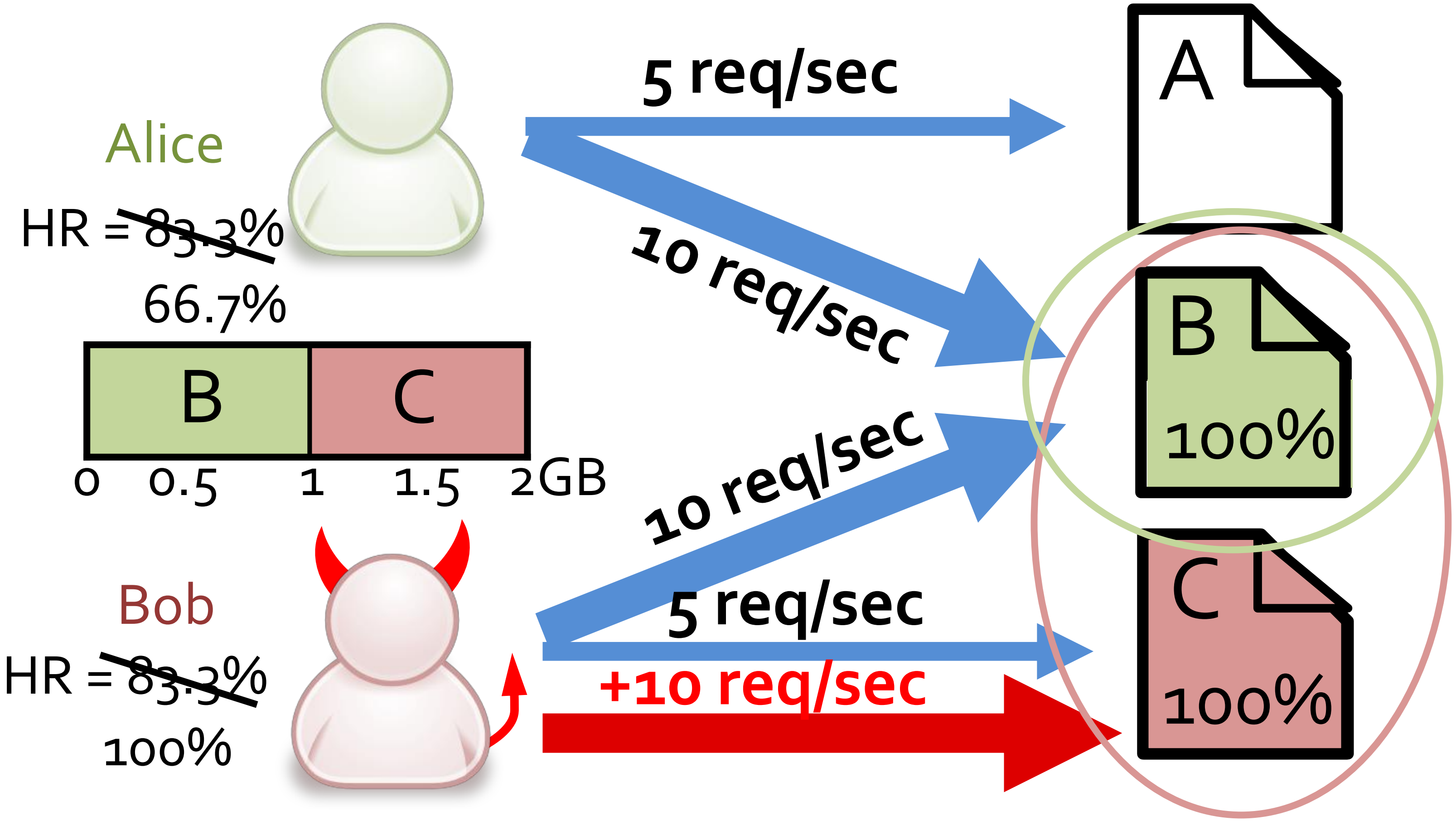


file sizes = 1GB, total cache = 2GB

Properties

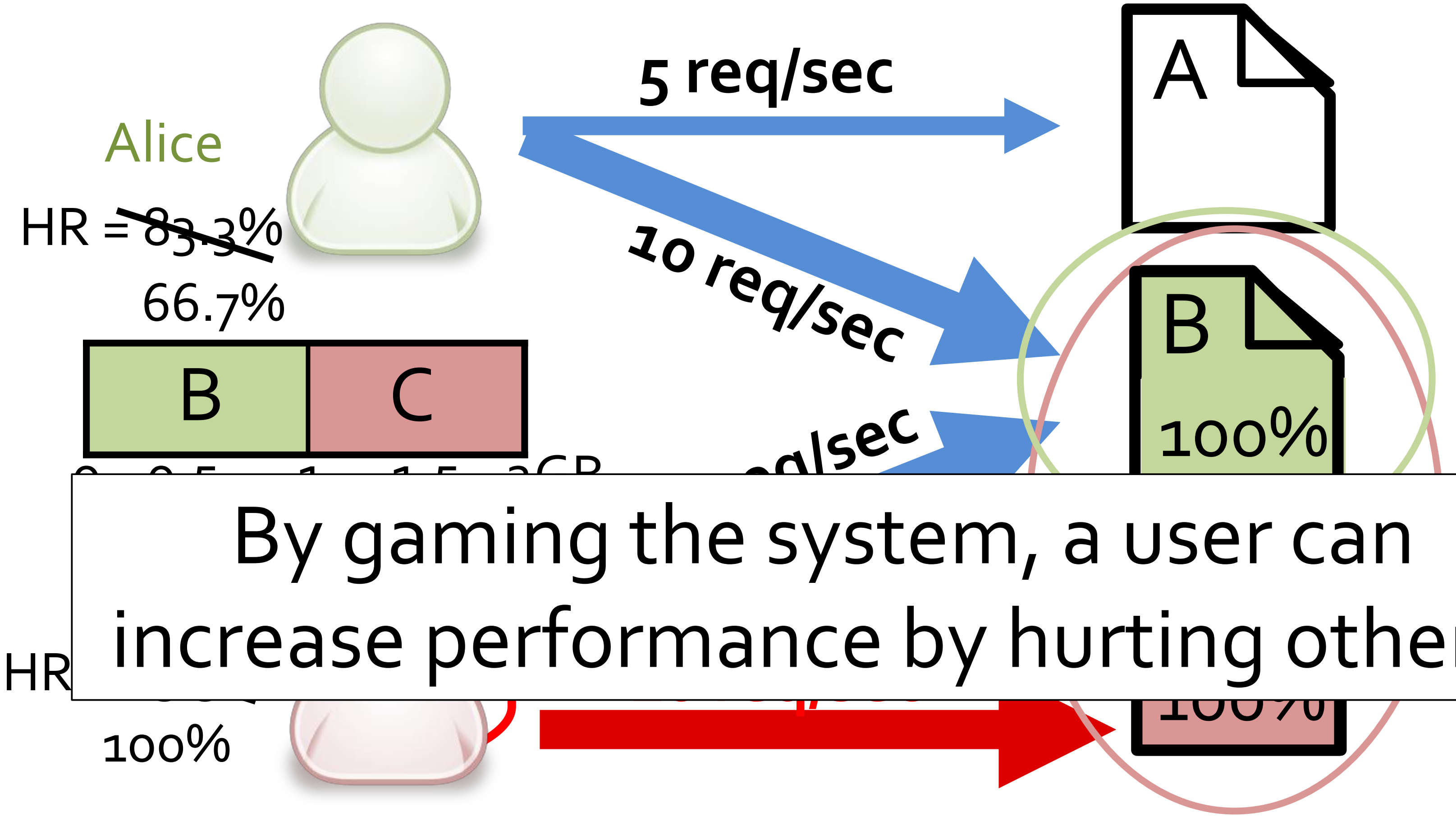
	Isolation Guarantee	Strategy Proofness	Pareto Efficiency
max-min fairness	✓	?	✓

An example



file sizes = 1GB, total cache = 2GB

An example



By gaming the system, a user can increase performance by hurting others!

file sizes = 1GB, total cache = 2GB

Properties

	Isolation Guarantee	Strategy Proofness	Pareto Efficiency
max-min fairness	✓	✗	✓
static allocation	✓	✓	✗
priority allocation	✗	✓	✓
max-min rate	✗	✓	✗
…	…	…	…

Theorem

No allocation policy can satisfy **all three** properties!

- Best we can do: two of three.

FairRide

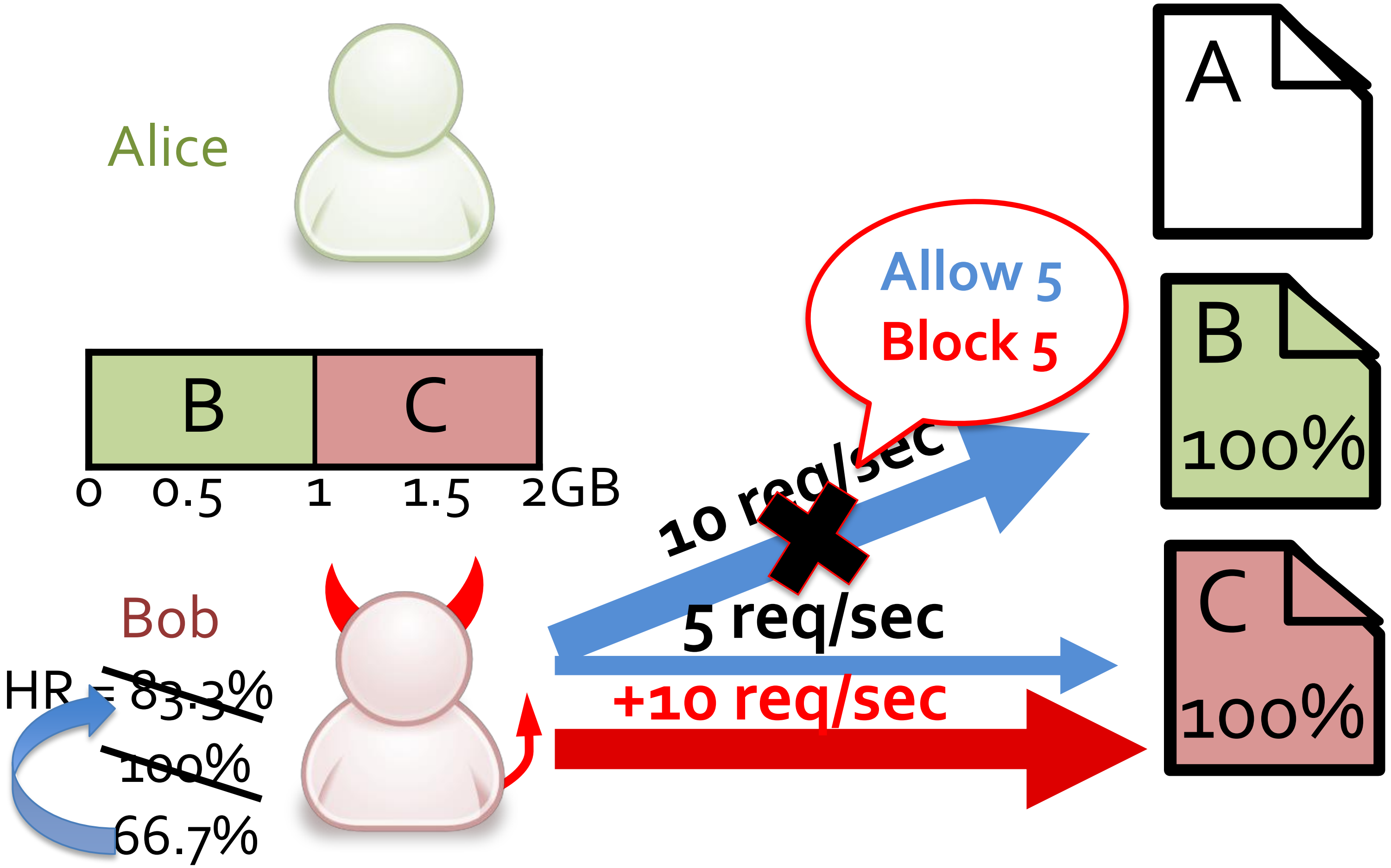
- Starts with max-min fairness
  - Allocate 1/n to each user
  - Split "cost" of shared files equally among shared users

- Only difference:

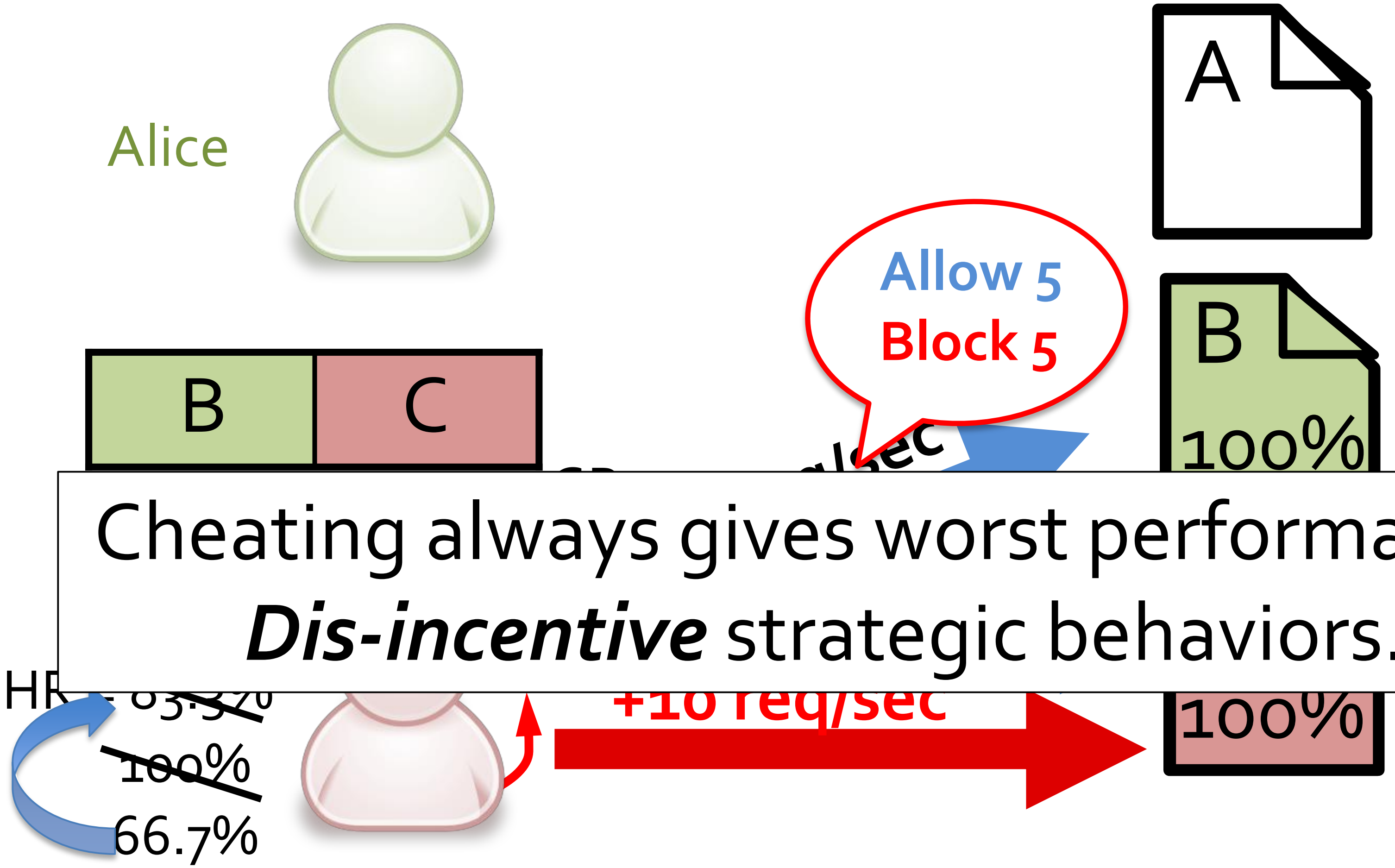
  blocking users who don't "pay" from accessing

- Probabilistic blocking: with some probability
  - Implemented with delaying

FairRide: Blocking



FairRide: Blocking



Cheating always gives worst performance.

Dis-incentive strategic behaviors.

Probabilistic blocking

- FairRide blocks a user with ***p(nj) = 1/(nj+1)*** probability
  - *nj* is number of other users caching file *j*
  - e.g., p(1)=50%, p(4)=20%

- The best you can do in a general case
  - **Less blocking does not prevent cheating**

Properties

	Isolation Guarantee	Strategy Proofness	Pareto Efficiency
max-min fairness	✓	✗	✓
static allocation	✓	✓	✗
priority allocation	✗	✓	✓
max-min rate	✗	✓	✗
FairRide	✓	✓	Near-optimal

Discussion

- What have you learned?
- Which paper(s) do you like? Why?
- Which paper(s) do you dislike? Why?
- Can you compare them to the classic scheduling policies?
- Can you come up with new ideas?