

# Operating Systems (Honor Track)

## Memory 5: Memory Management in Modern Computer Systems

Xin Jin

Spring 2023

# Memory Management in Modern Computer Systems

- Memory Abstraction
  - NSDI'14 FaRM
- Demand paging: remote memory over RDMA
  - NSDI'17 InfiniSwap
  - OSDI'20 AIFM
- Demand paging: memory swapping between GPU memory and host memory
  - OSDI'20 PipeSwitch

# FaRM: Fast Remote Memory

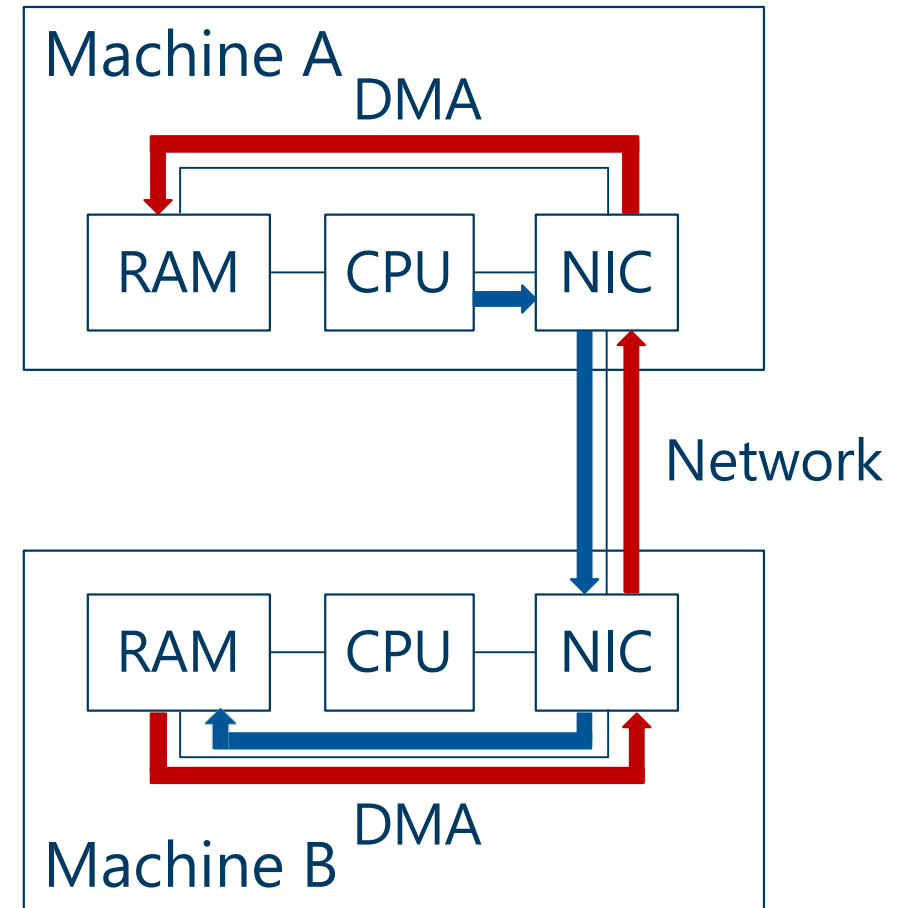
Aleksandar Dragojević, Dushyanth Narayanan,  
Orion Hodson, Miguel Castro

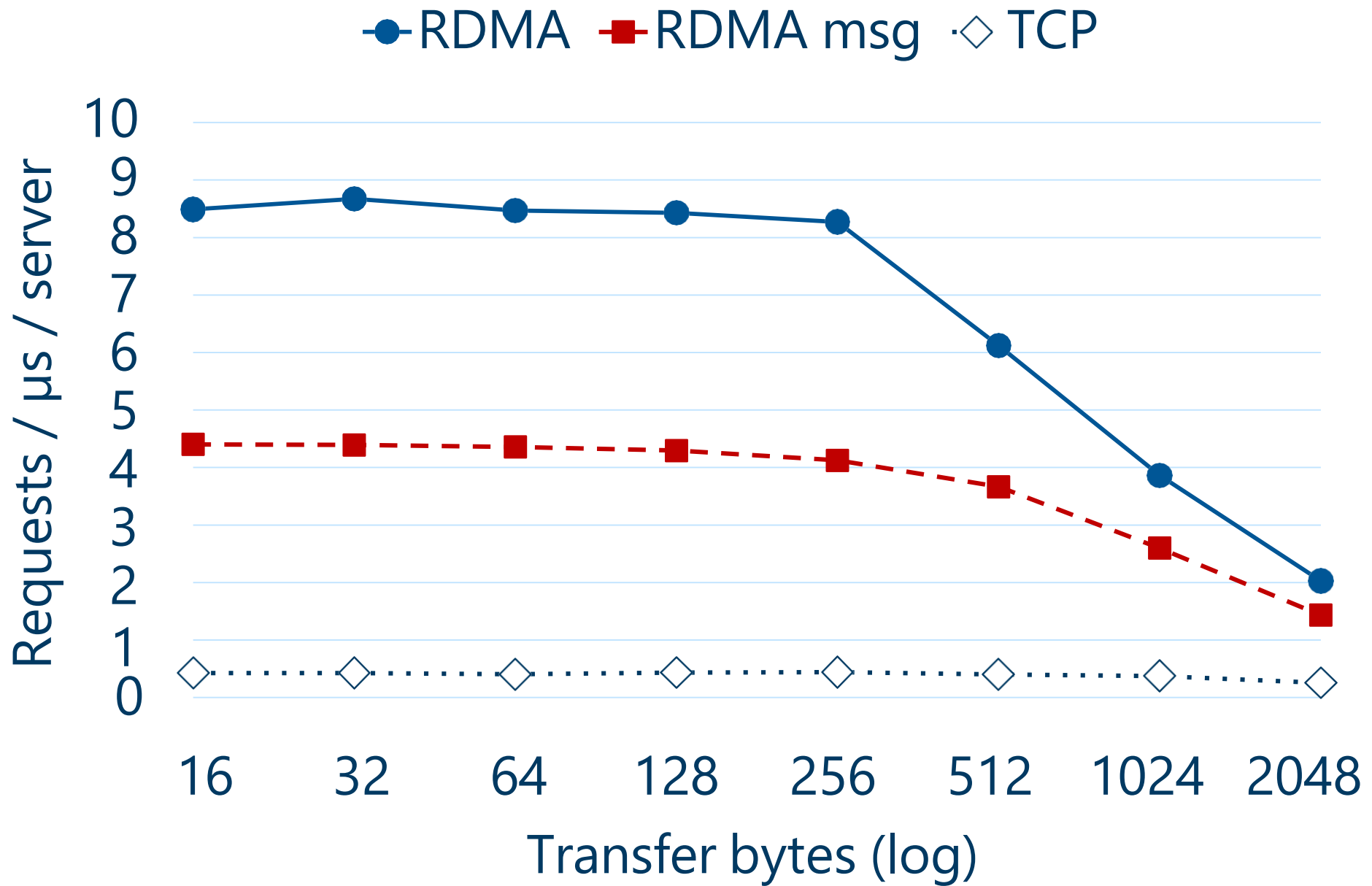
# Hardware trends

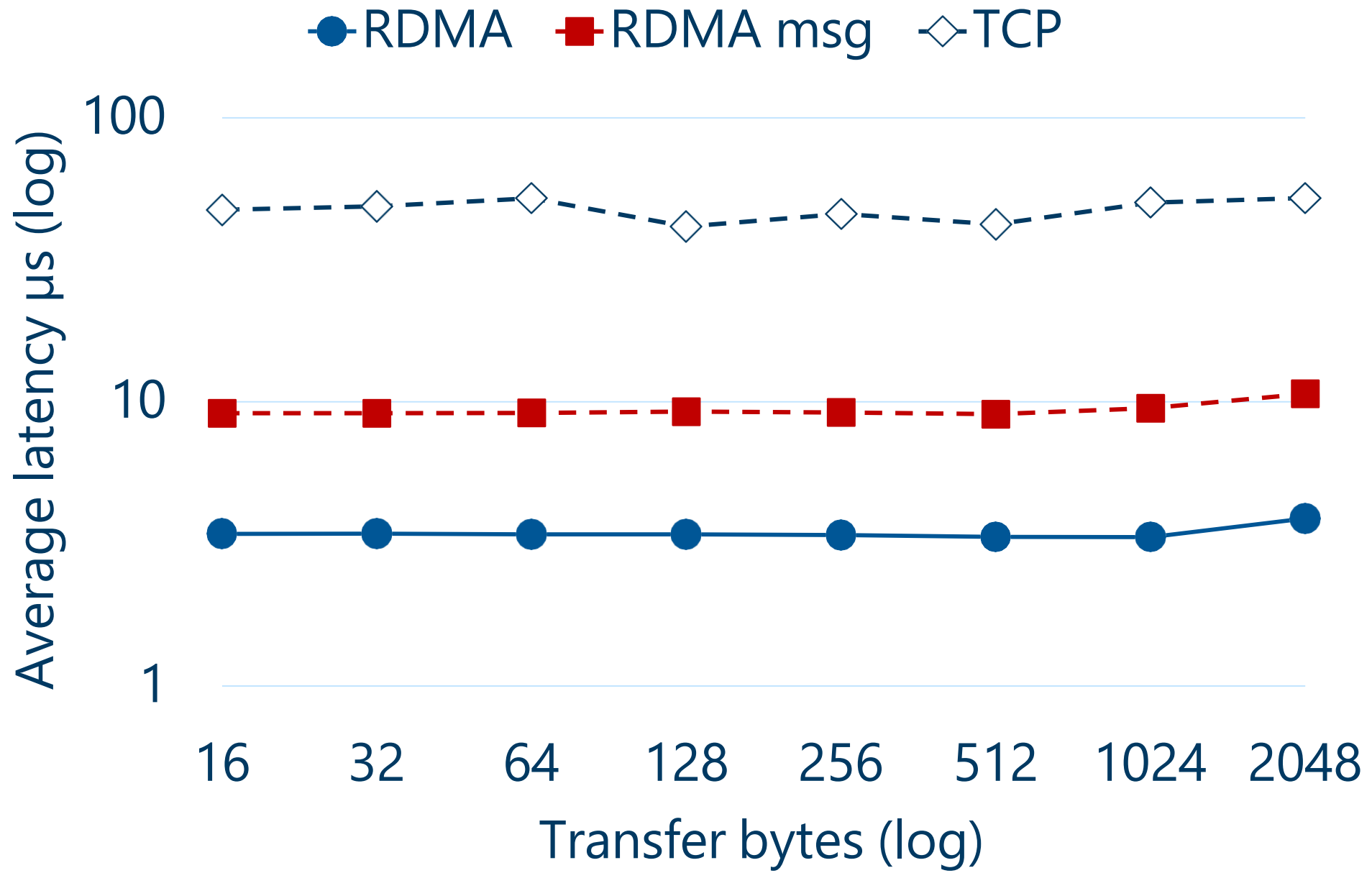
- Main memory is cheap
  - 100 GB – 1 TB per server
  - 10 – 100 TBs in a small cluster
- New data centre networks
  - 40 Gbps throughput (100 this year)
  - 1-3  $\mu$ s latency
  - RDMA primitives

# Remote direct memory access

- Read / write remote memory
  - NIC performs DMA requests
- FaRM uses RDMA extensively
  - Reads to directly read data
  - Writes into remote buffers for messaging
- Great performance
  - Bypasses the kernel
  - Bypasses the remote CPU







# Applications

- Data centre applications
  - Irregular access patterns
  - Latency sensitive
- Data serving
  - Key-value store
  - Graph store
- Enabling new applications



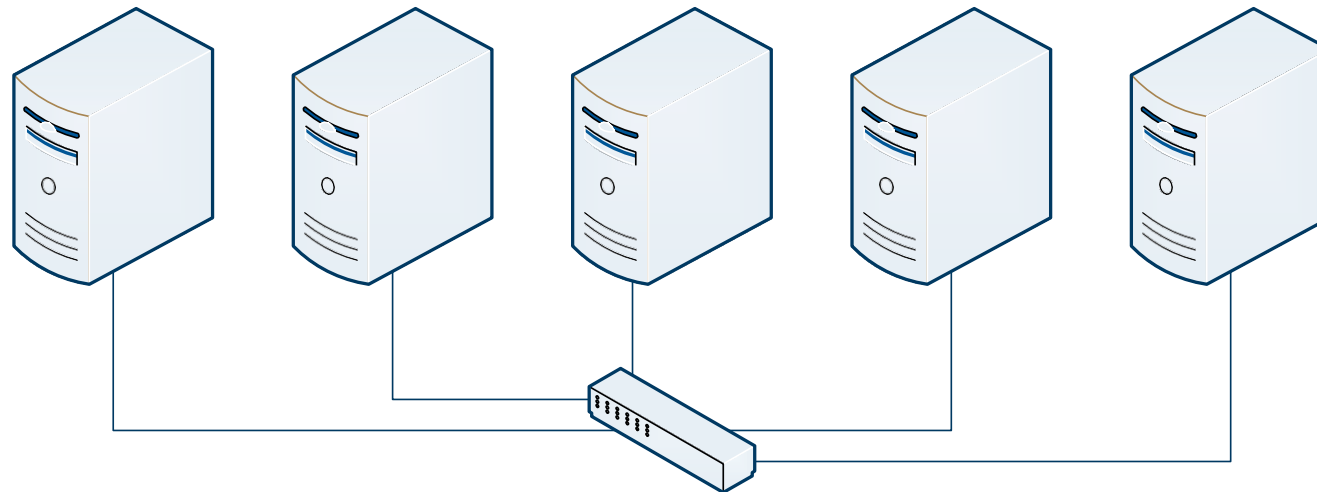
# How to program a modern cluster?

We have:

- TBs of DRAM
- 100s of CPU cores
- RDMA network

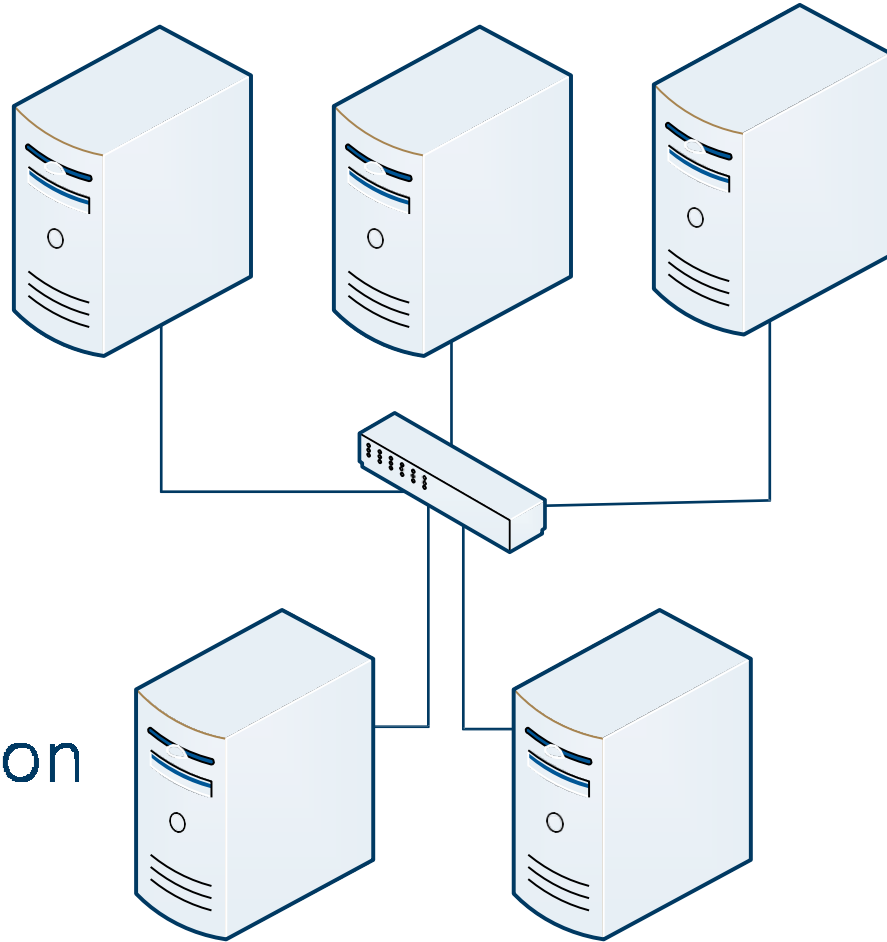
Desirable:

- Keep data in memory
- Access data using RDMA
- Collocate data and computation



# Traditional model

Servers: store data

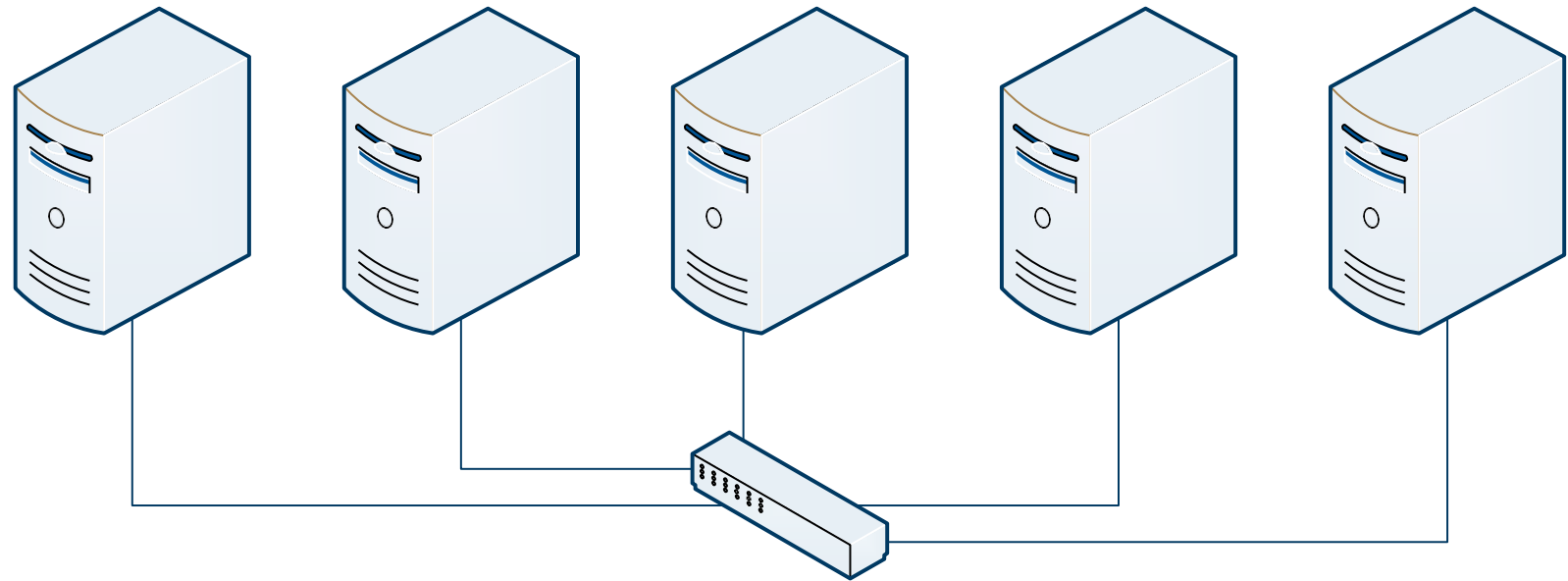


Clients: execute application

# Symmetric model

Access to local memory is much faster

Server CPUs are mostly idle with RDMA

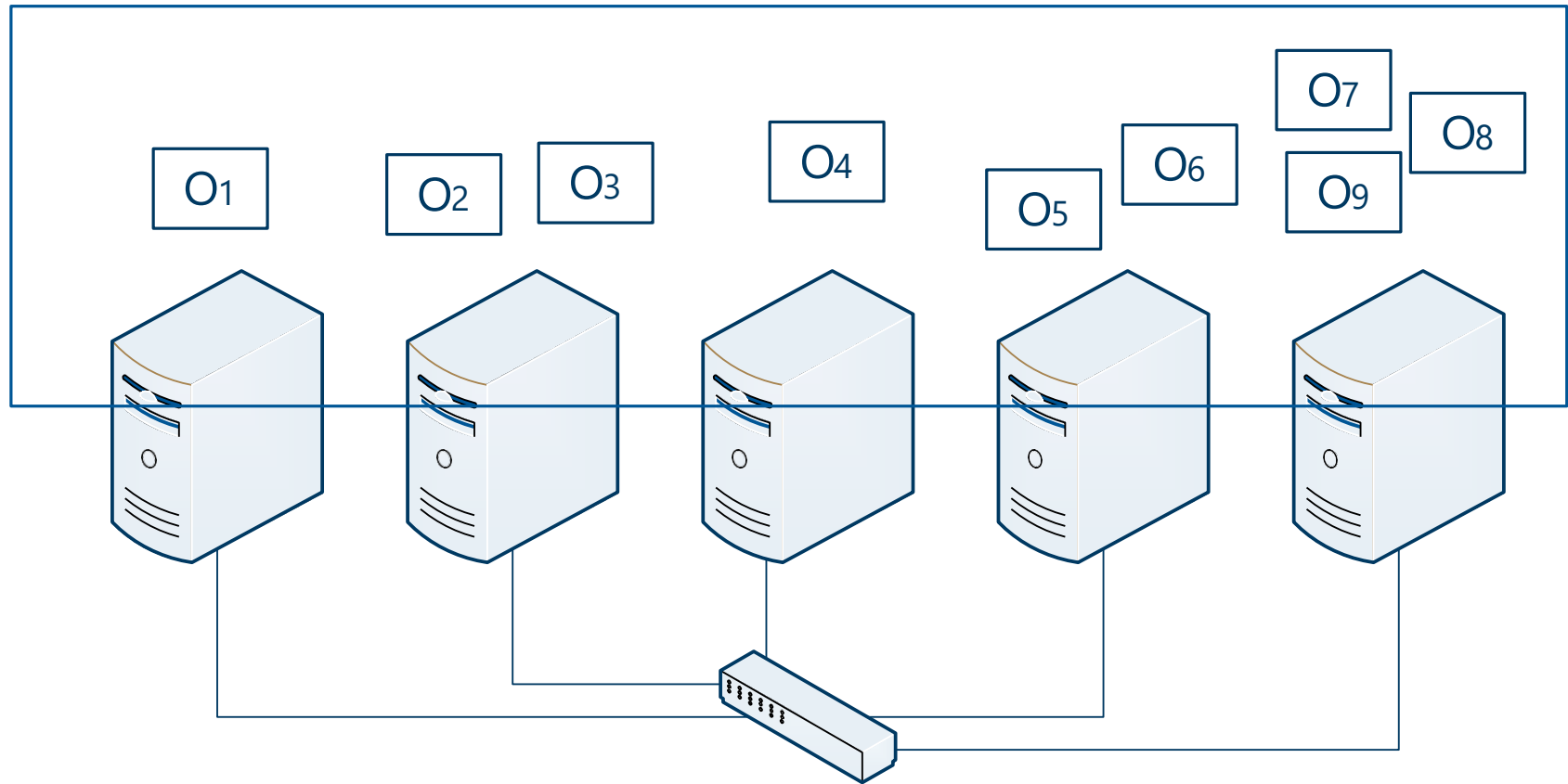


Machines store data and execute application

# Shared address space

Supports direct RDMA of objects

Programmability a welcome bonus



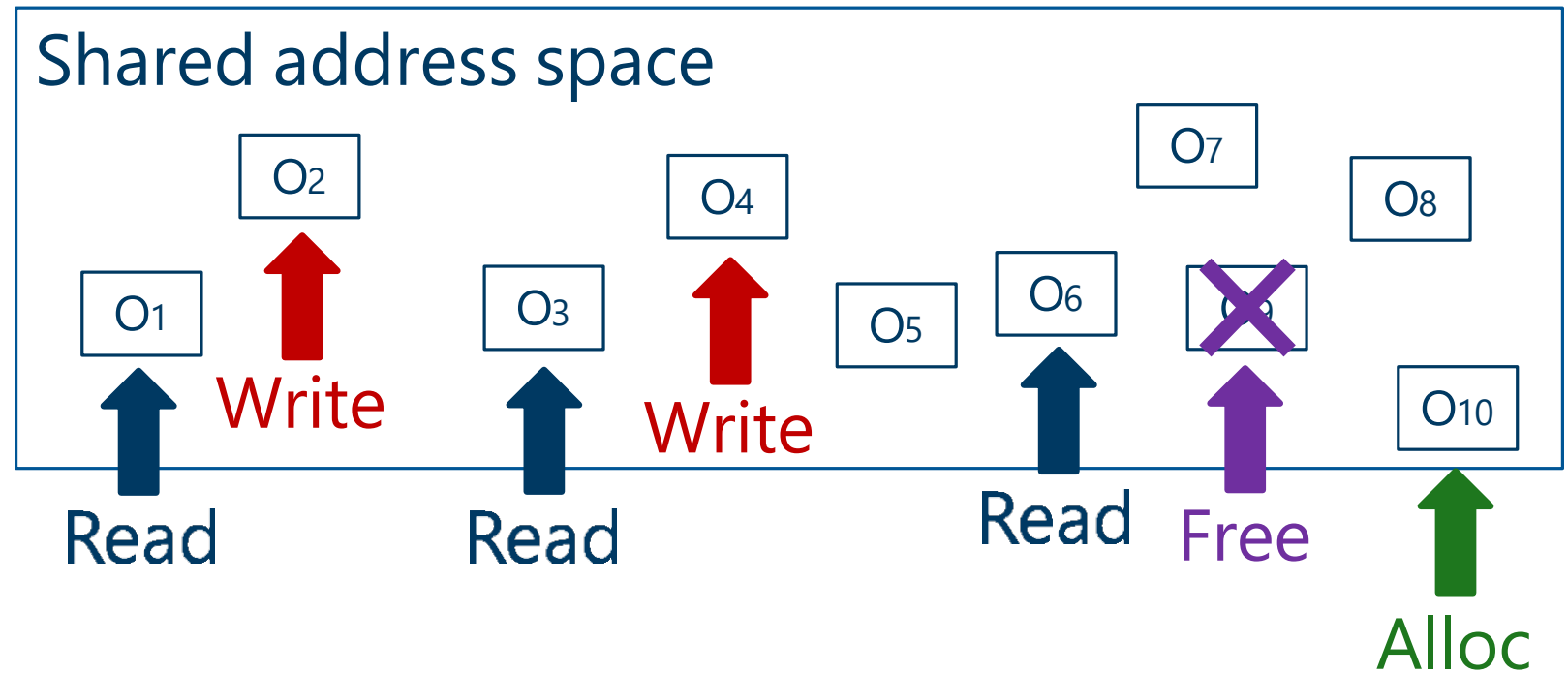
# Shared address space

General primitive

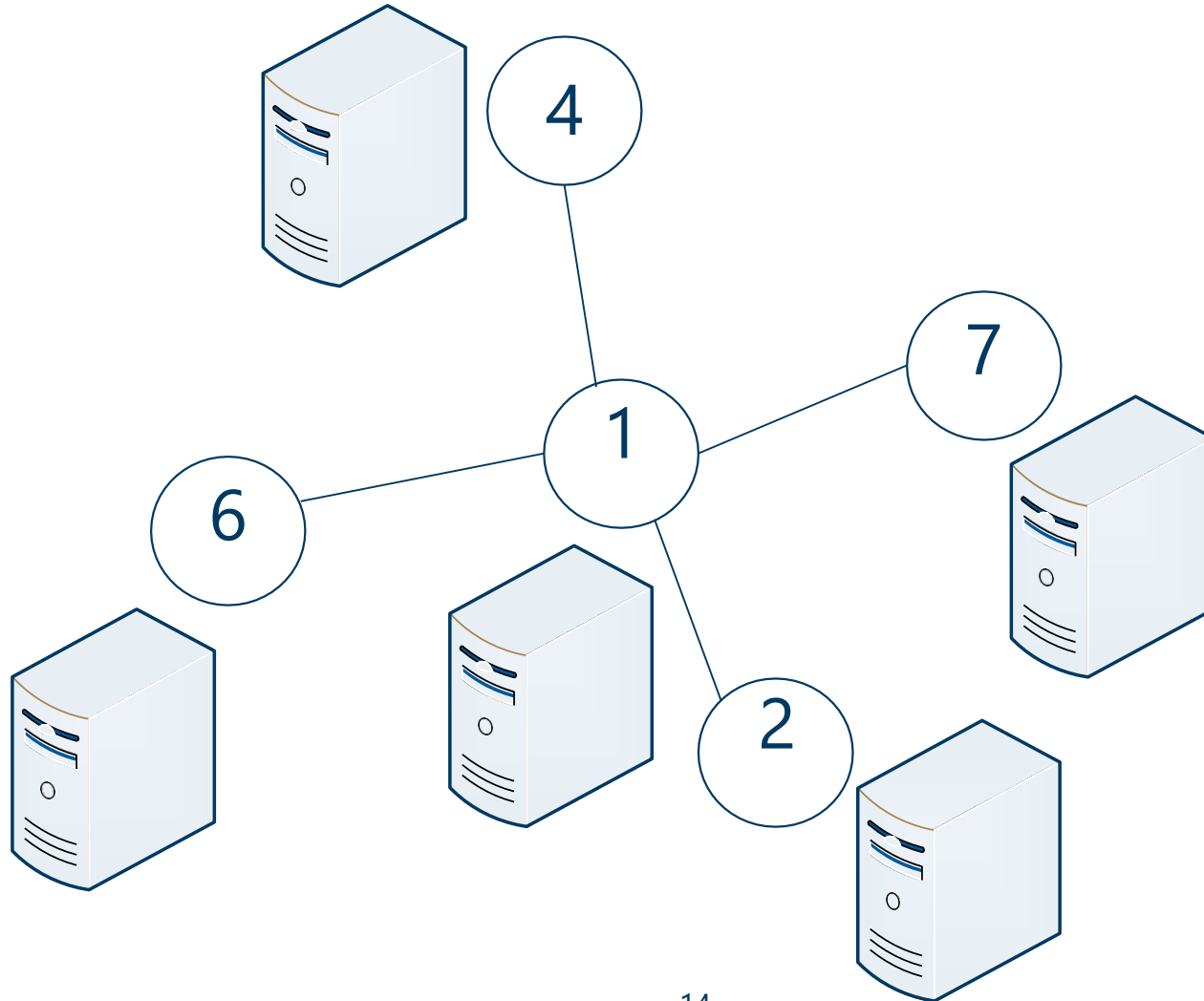
Strong consistency:  
serializability

Transparent:

- location
- concurrency
- failures



# Optimizations: locality awareness

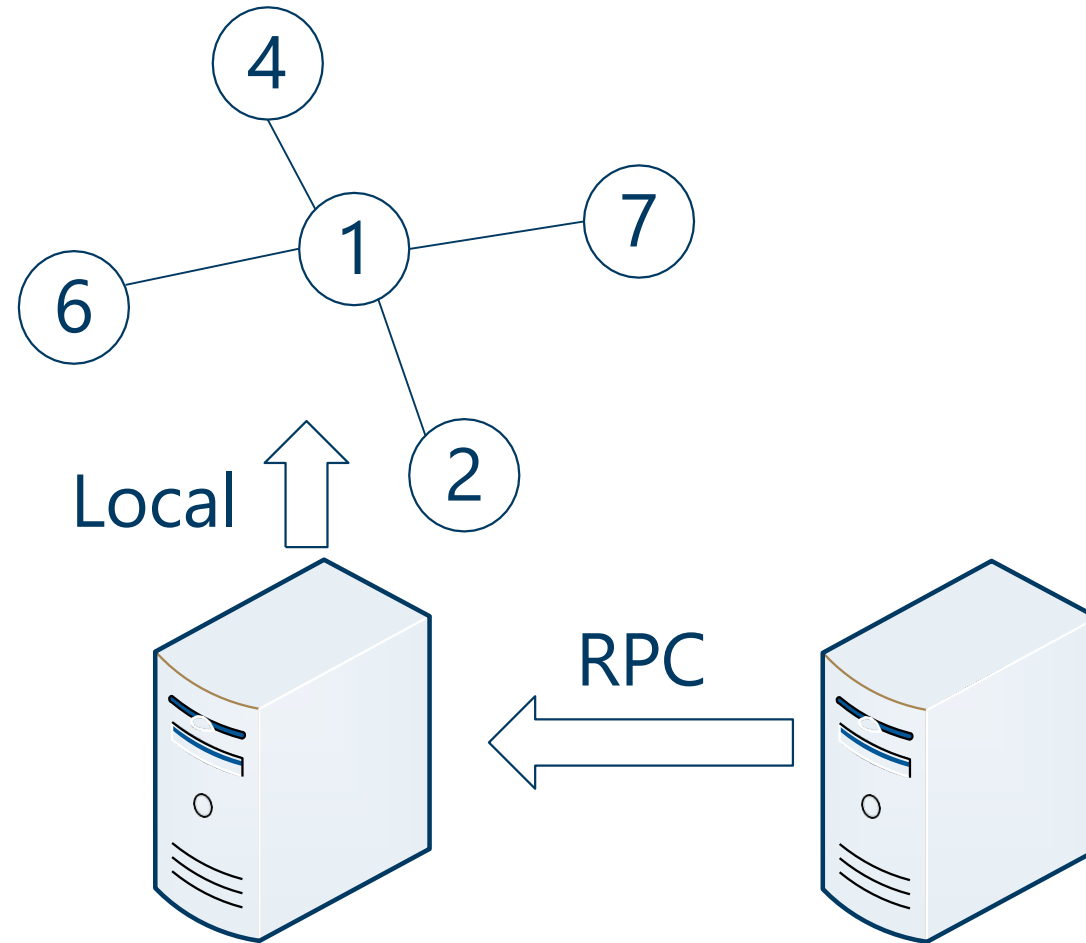


# Optimizations: locality awareness

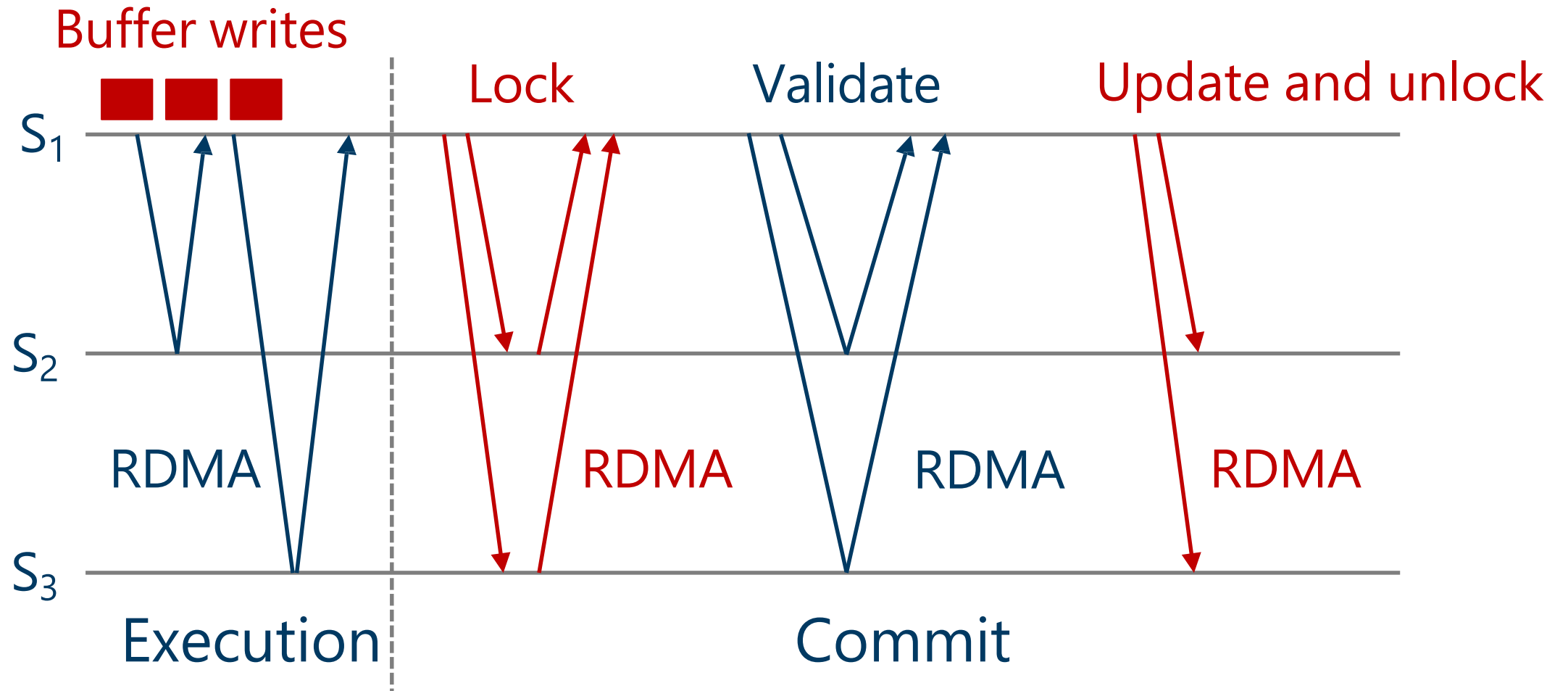
Collocate data  
accessed together

Ship computation  
to target data

Optimized  
single server  
transactions



# Transactions





# TAO [Bronson '13, Armstrong '13]

- Facebook's in-memory graph store
  - Workload
    - Read-dominated (99.8%)
    - 10 operation types
  - FaRM implementation
    - Nodes and edges are FaRM objects
    - Lock-free reads for lookups
    - Transactions for updates
- 6 Mops/s/srv  
(10x improvement)
- 42  $\mu$ s average latency  
(40 – 50x improvement)

# FaRM

- Platform for distributed computing
  - Data is in memory
  - RDMA
- Shared memory abstraction
  - Transactions
  - Lock-free reads
- Order-of-magnitude performance improvements
  - Enables new applications

# Memory Management in Modern Computer Systems

- Memory Abstraction
  - NSDI'14 FaRM
- Demand paging: remote memory over RDMA
  - NSDI'17 InfiniSwap
  - OSDI'20 AIFM
- Demand paging: memory swapping between GPU memory and host memory
  - OSDI'20 PipeSwitch

# Efficient Memory Disaggregation with Infiniswap

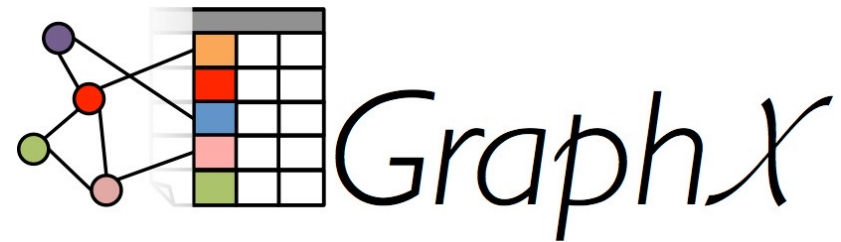
Juncheng Gu, Youngmoon Lee, Yiwen Zhang,  
Mosharaf Chowdhury, Kang G. Shin



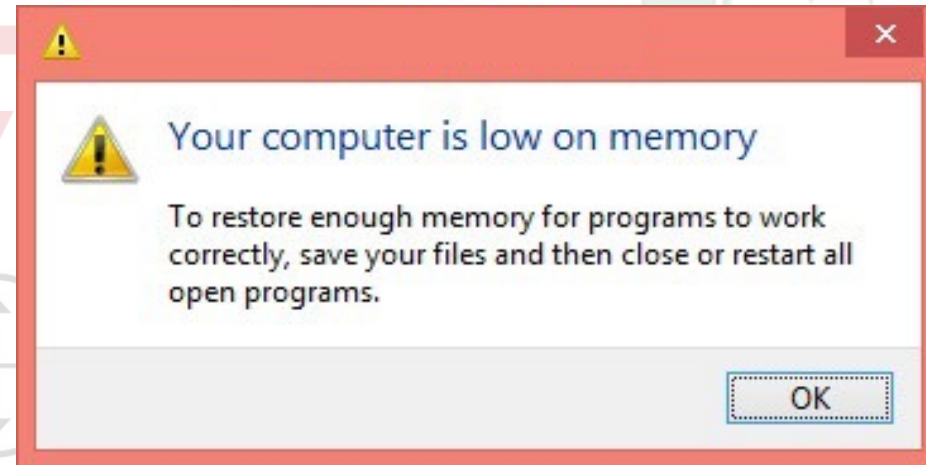
# Agenda

- **Motivation and related work**
- Design and system overview
- Implementation and evaluation
- Future work and conclusion

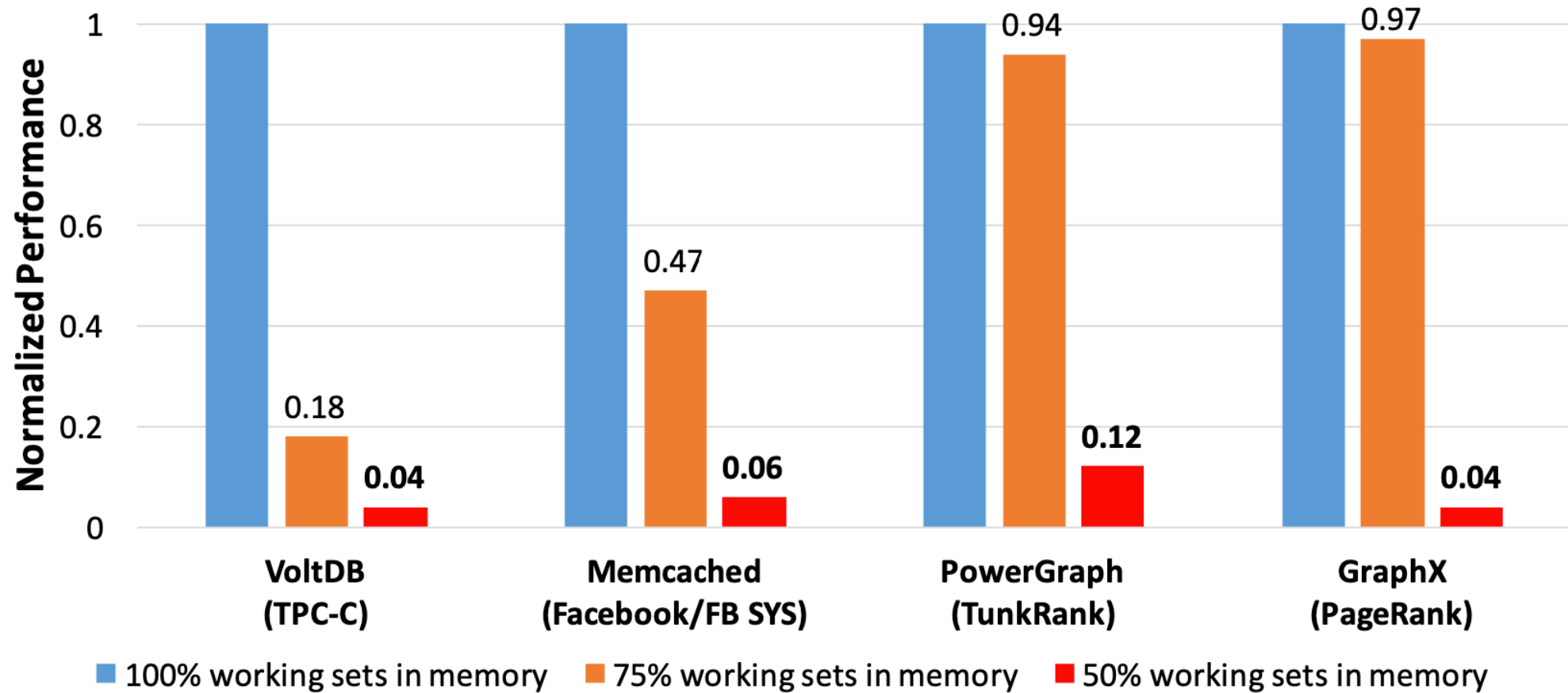
# Memory-intensive applications



# Memory-intensive applications



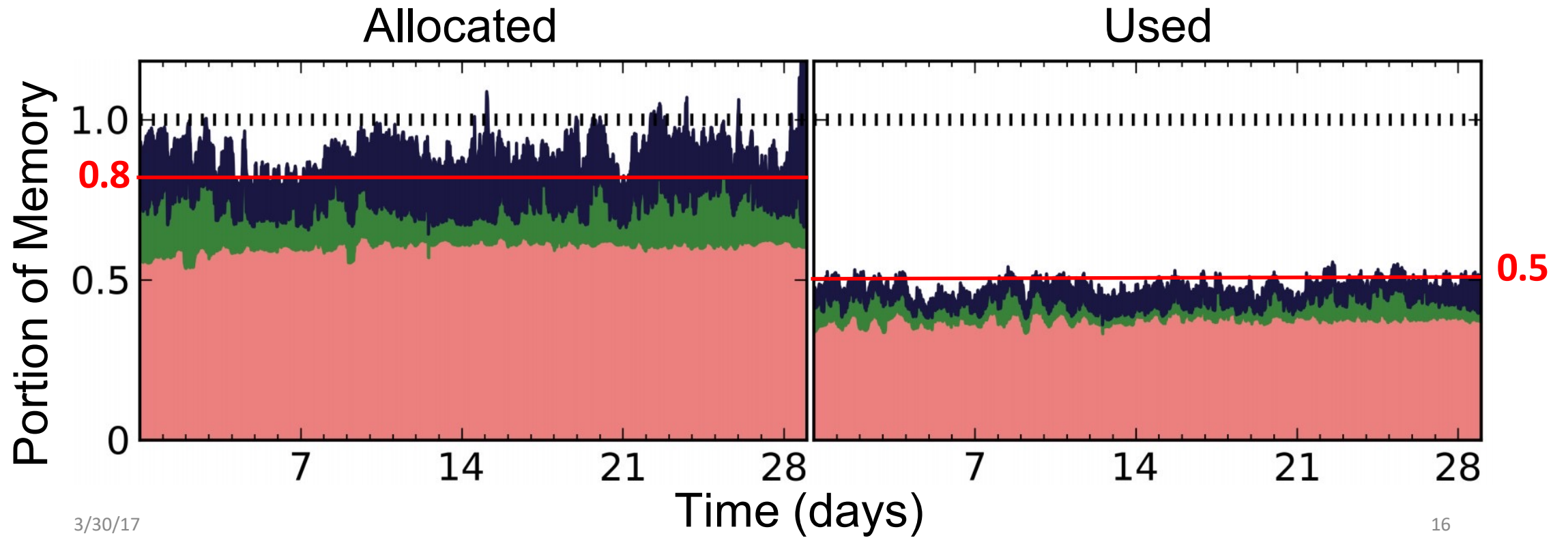
# Performance degradation





# Memory underutilization

- Google Cluster Analysis<sup>[1]</sup>



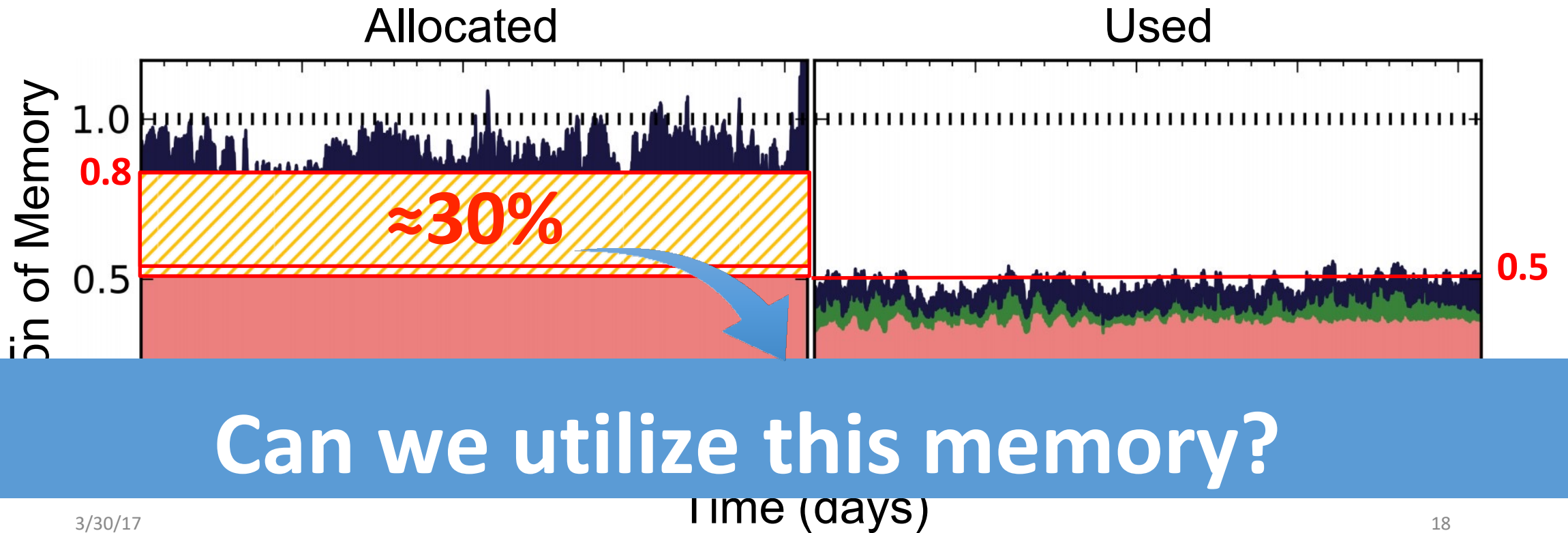
3/30/17

16

[1] Reiss, Charles, et al. "Heterogeneity and dynamicity of clouds at scale: Google trace analysis." *SoCC'12*.

# Memory underutilization

- Google Cluster Analysis<sup>[1]</sup>

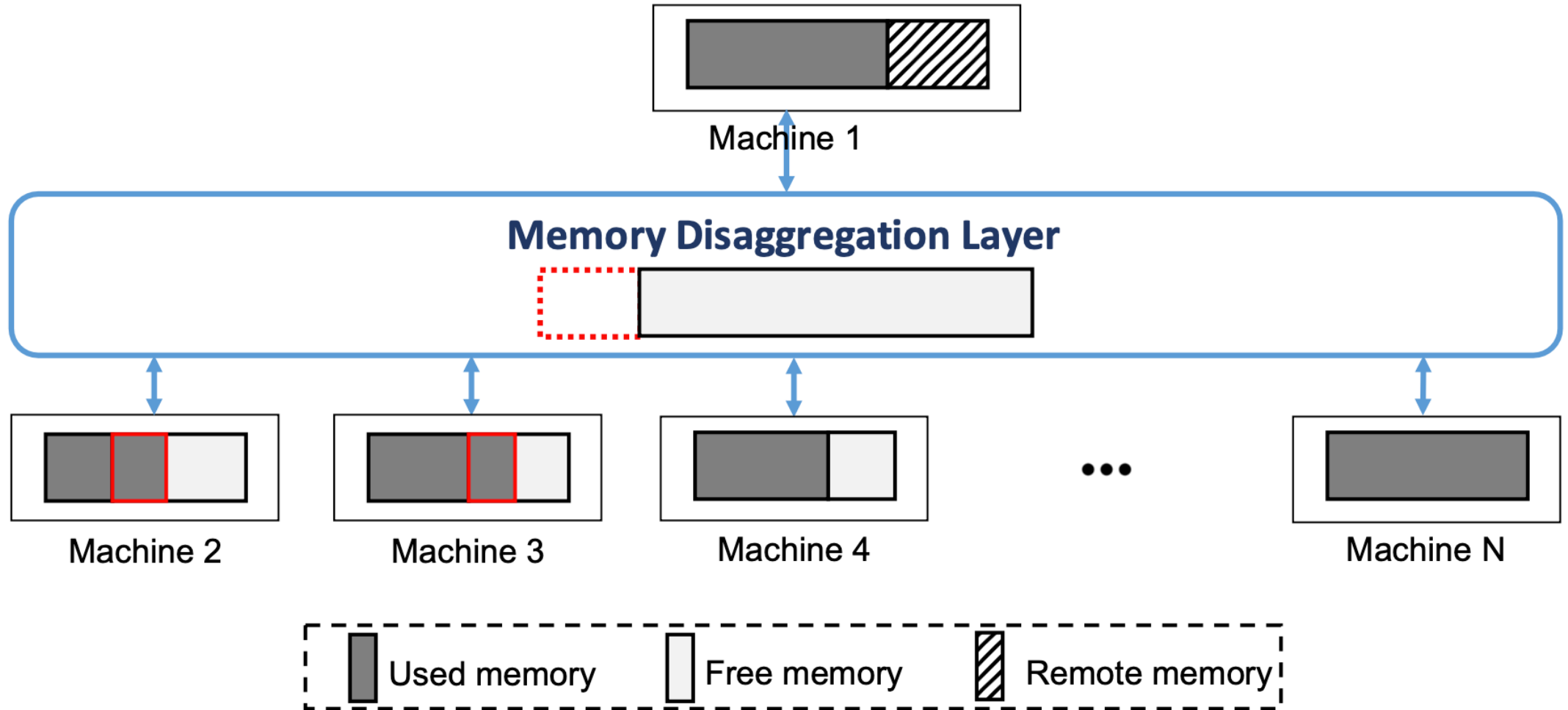


3/30/17

18

[1] Reiss, Charles, et al. "Heterogeneity and dynamicity of clouds at scale: Google trace analysis." *SoCC'12*.

# Disaggregate free memory



# What are the challenges?

- **Minimize deployment overhead**
  - **No hardware design**
  - **No application modification**
- **Tolerate failures**
  - e.g. network disconnection, machine crash
- **Manage remote memory at scale**

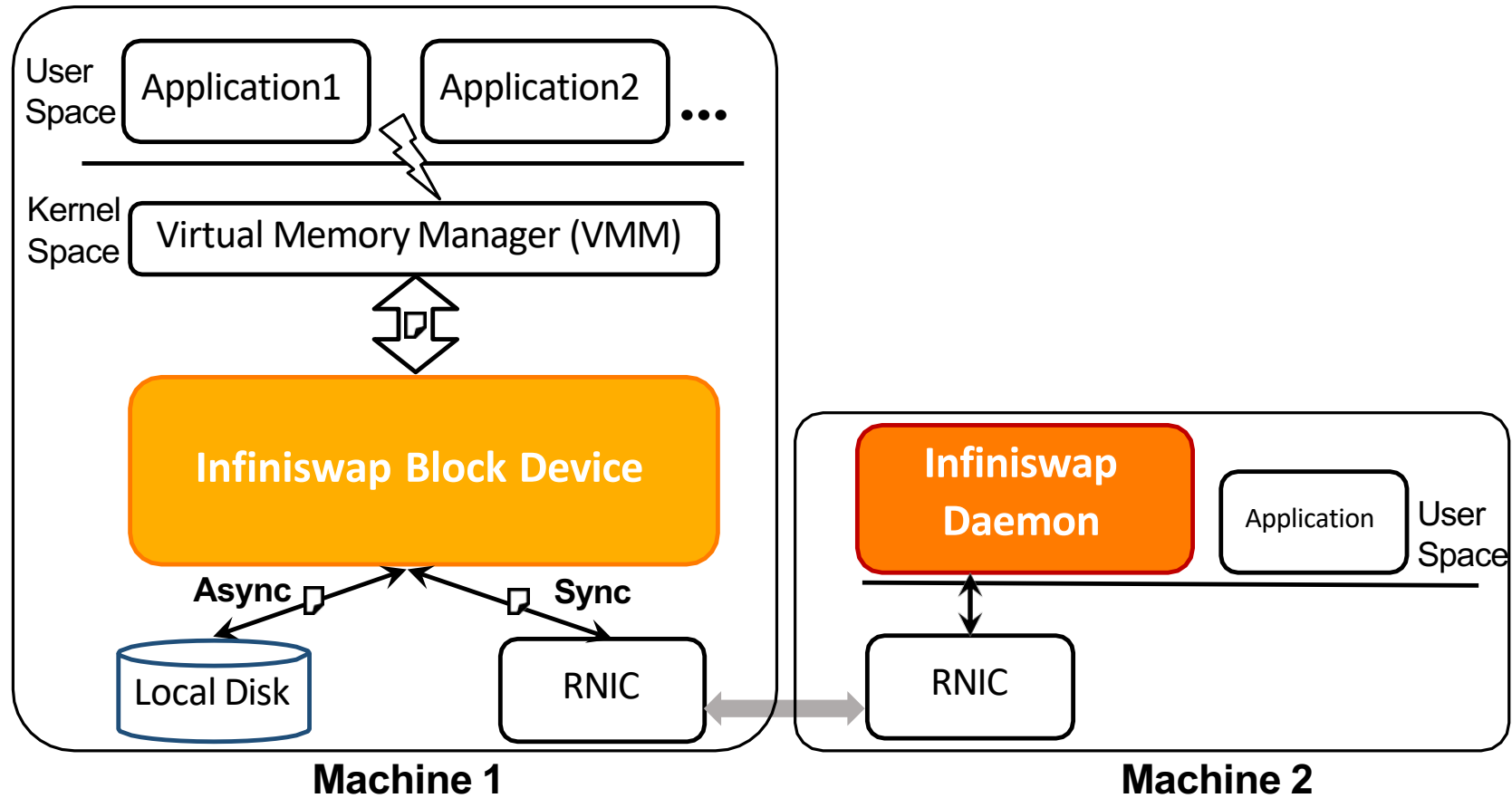
# Recent work on memory disaggregation

	No HW design	No app modification	Fault-tolerance	Scalability
<b>Memory Blade</b> [ISCA'09]	✗	✓	✓	✓
<b>HPBD</b> [CLUSTER'05] / <b>NBDX</b> <sub>[1]</sub>	✓	✓	✗	✗
<b>RDMA key-value service</b> (e.g. HERD[SIGCOMM'14], FaRM[NSDI'14])	✓	✗	✓	✓
<b>Intel Rack Scale Architecture (RSA)</b> <sub>[2]</sub>	✗	✓	✓	✓
<b>Infiniswap</b>	✓	✓	✓	✓

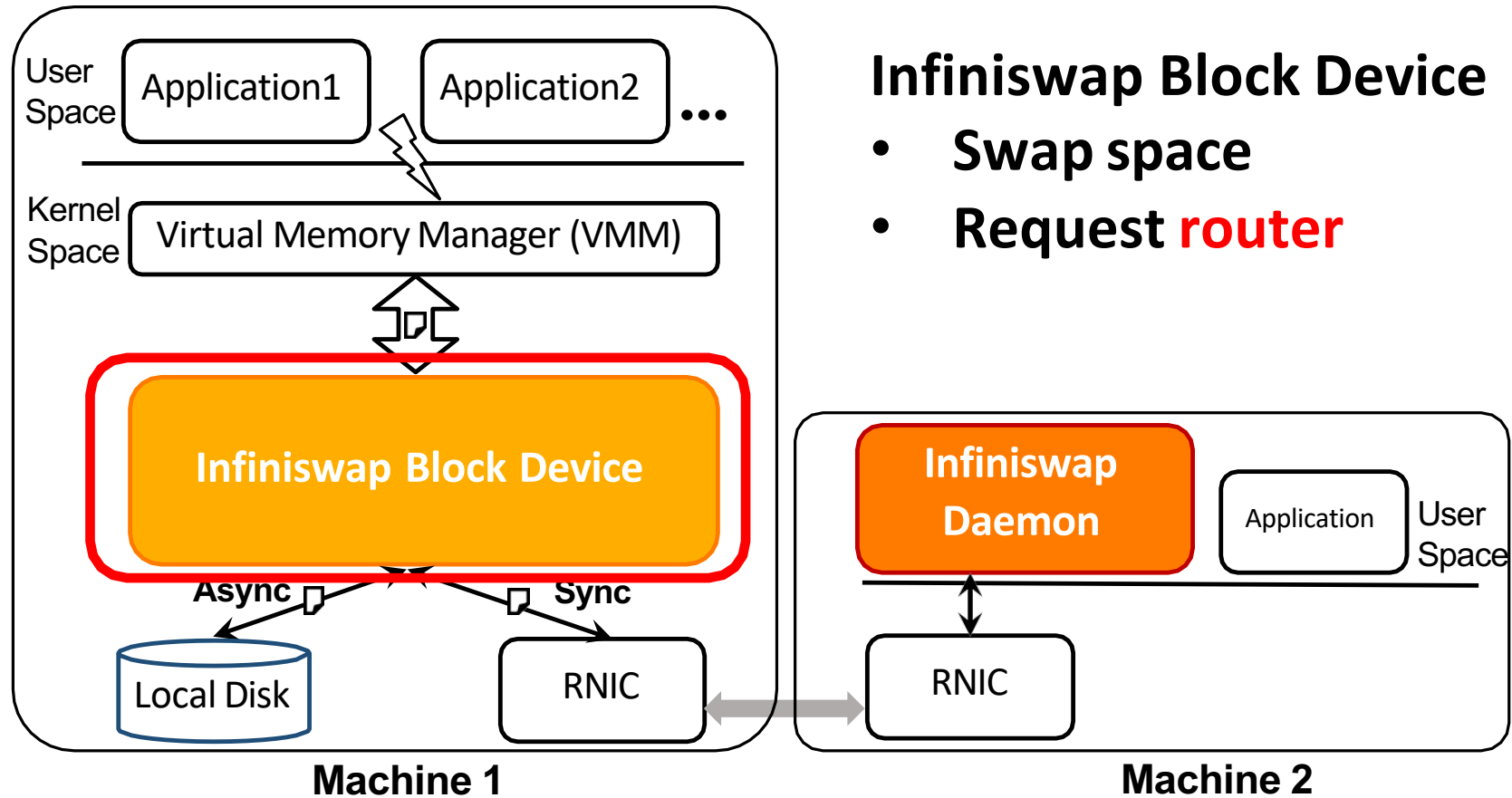
# Agenda

- Motivation and related work
- **Design and system overview**
- Implementation and evaluation
- Future work and conclusion

# System Overview



# System Overview

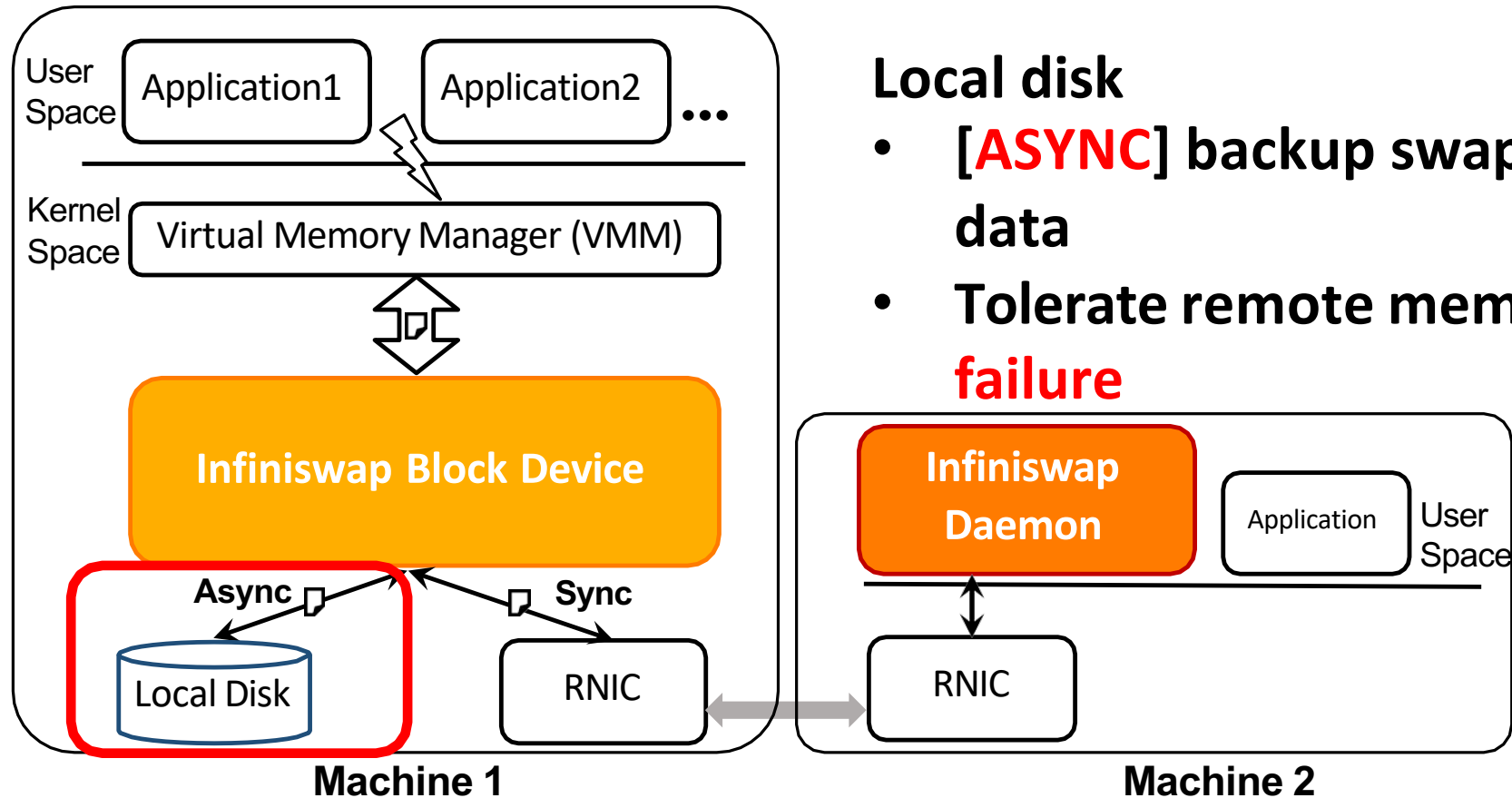


## Infiniswap Block Device

- Swap space
- Request **router**



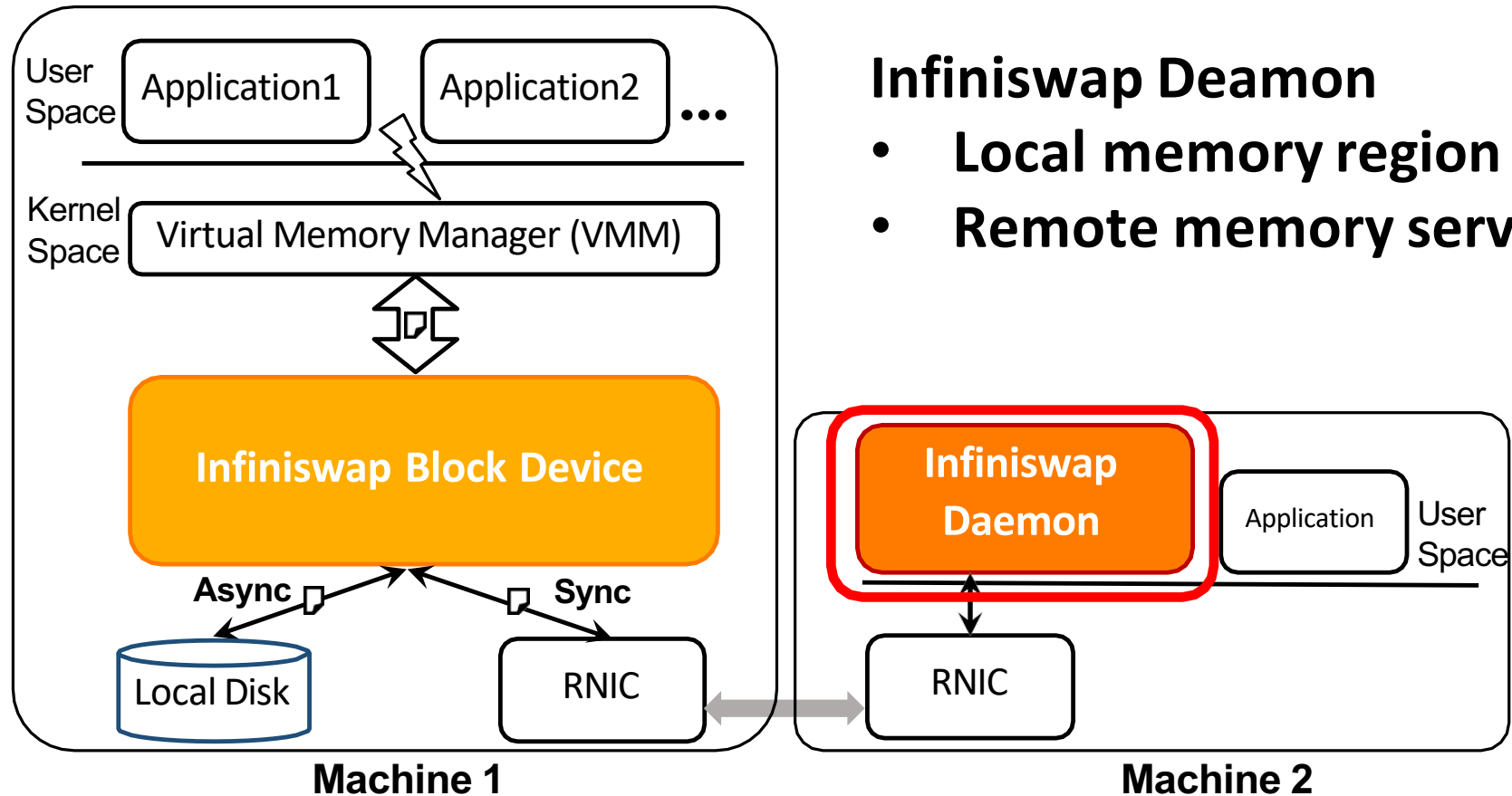
# System Overview



## Local disk

- **[ASYNC]** backup swapped-out data
- Tolerate remote memory **failure**

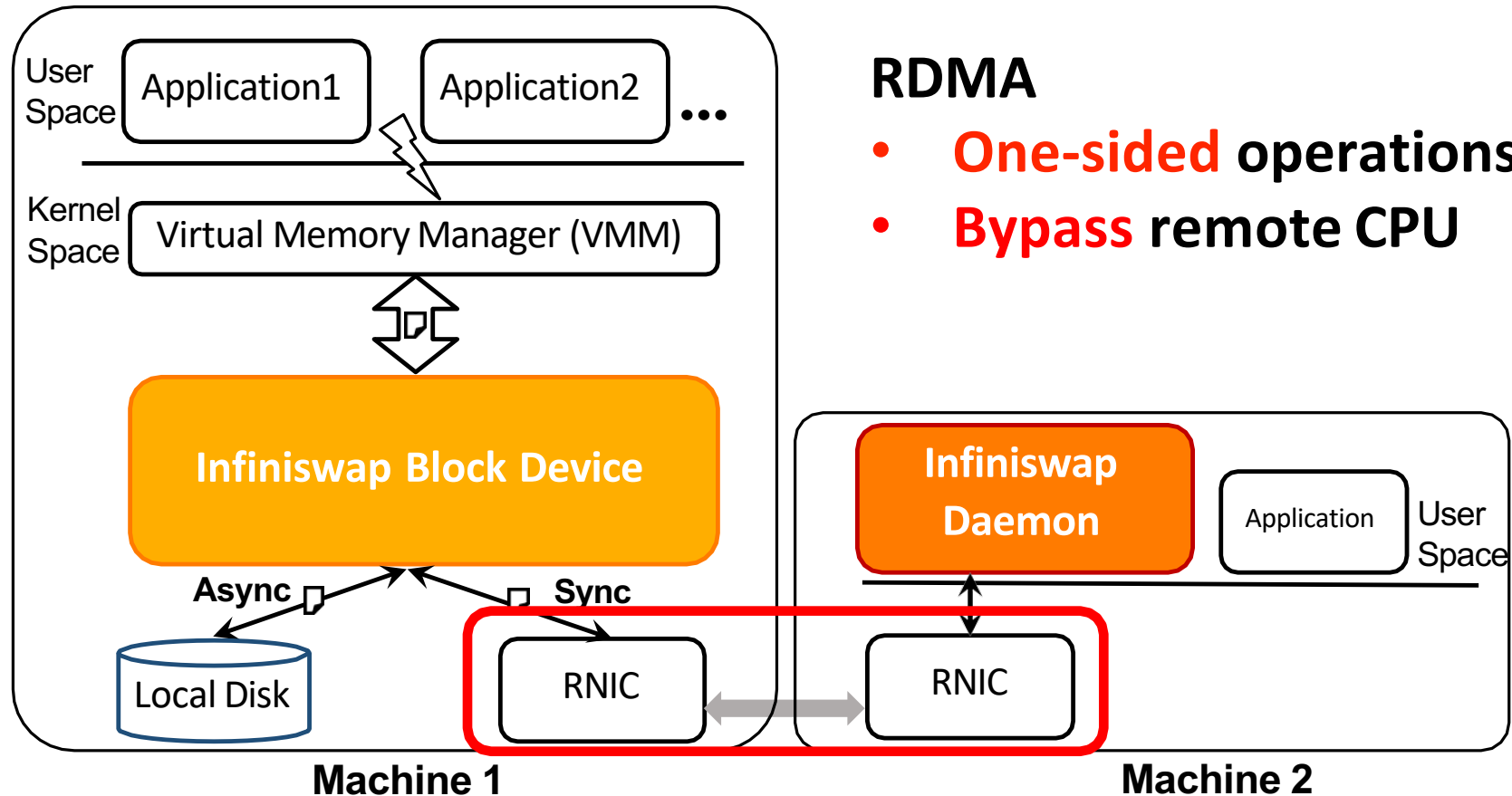
# System Overview



## Infiniswap Daemon

- **Local memory region**
- **Remote memory service**

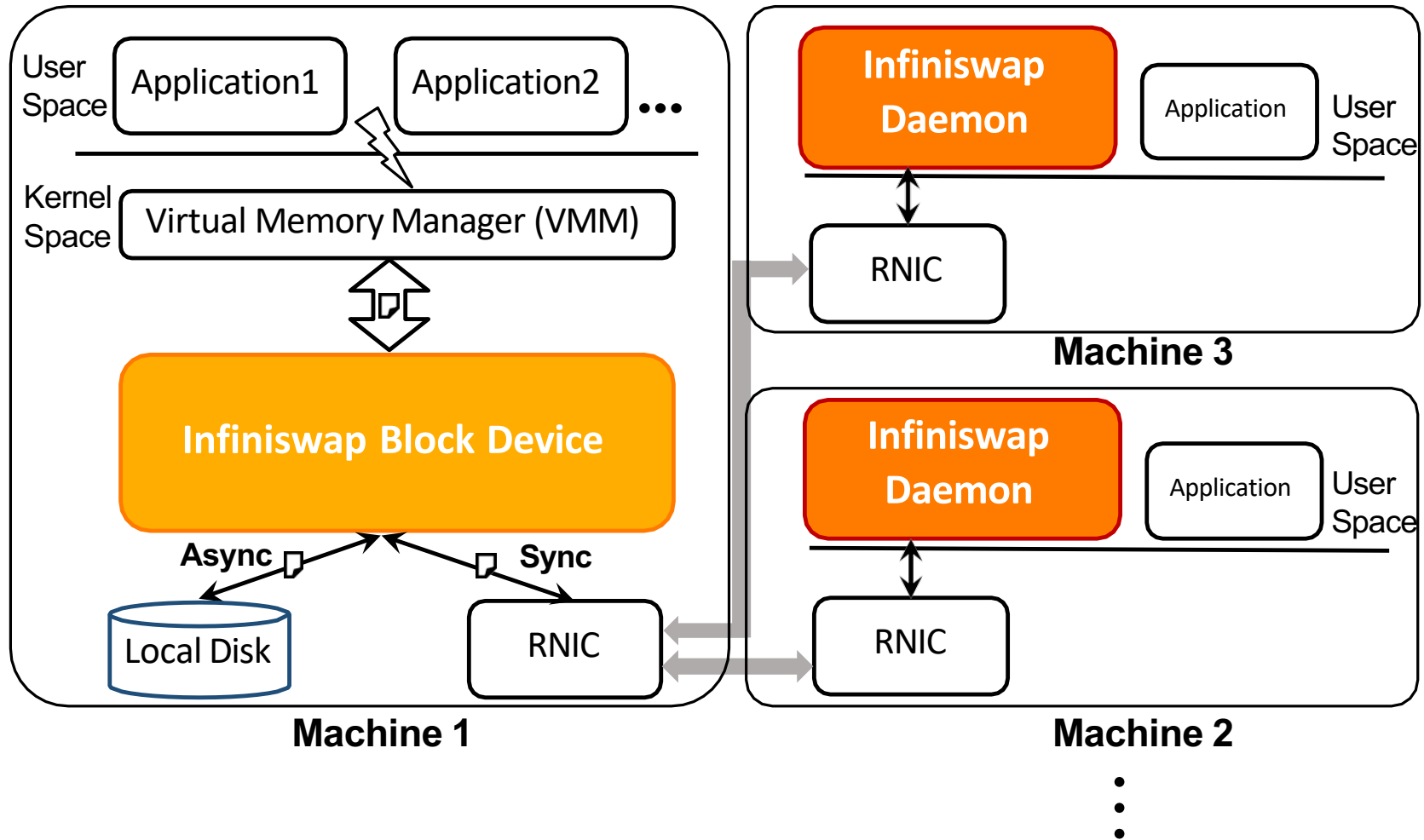
# System Overview



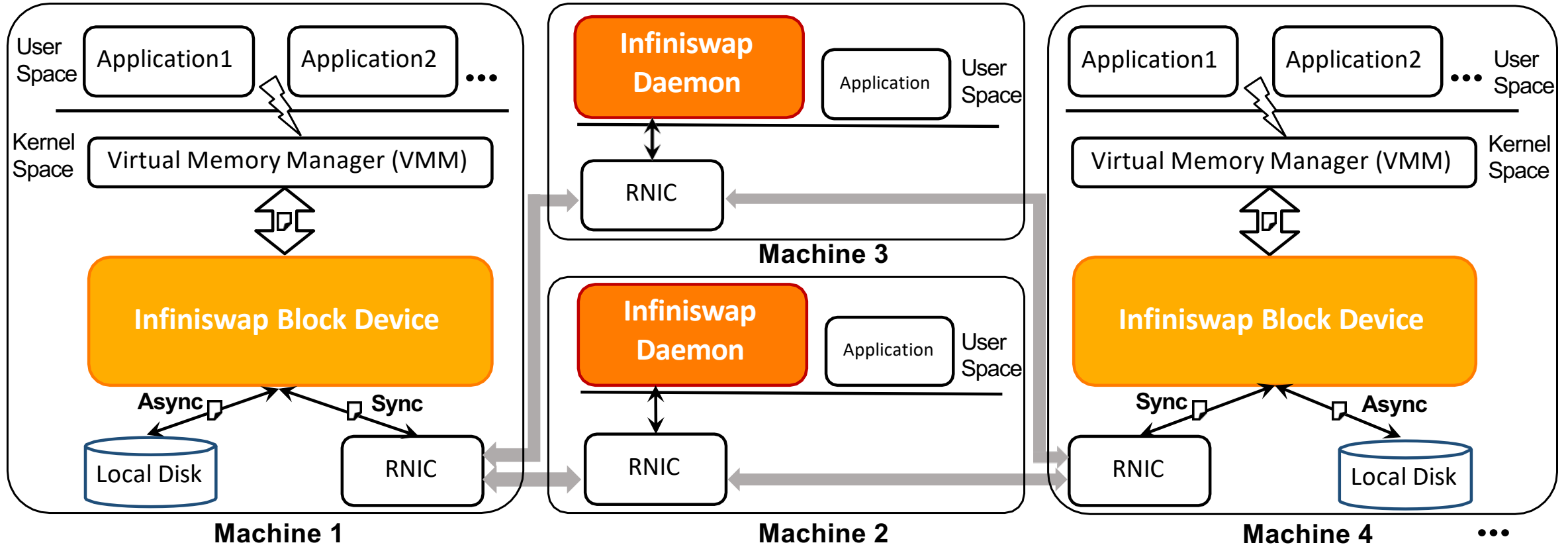
# How to meet the design objectives?

Objectives	Ideas
No hardware design	Remote paging
No application modification	
Fault-tolerance	Local backup disk

# One-to-many



# Many-to-many



# Many-to-many

## How to scale remote memory?

- How to **find** remote memory in the cluster?
- Which remote mapping should be **evicted**?

# How to meet the design objectives?

**Objectives**

No hardware design

No application modification

Fault-tolerance

**Scalability**

**Ideas**

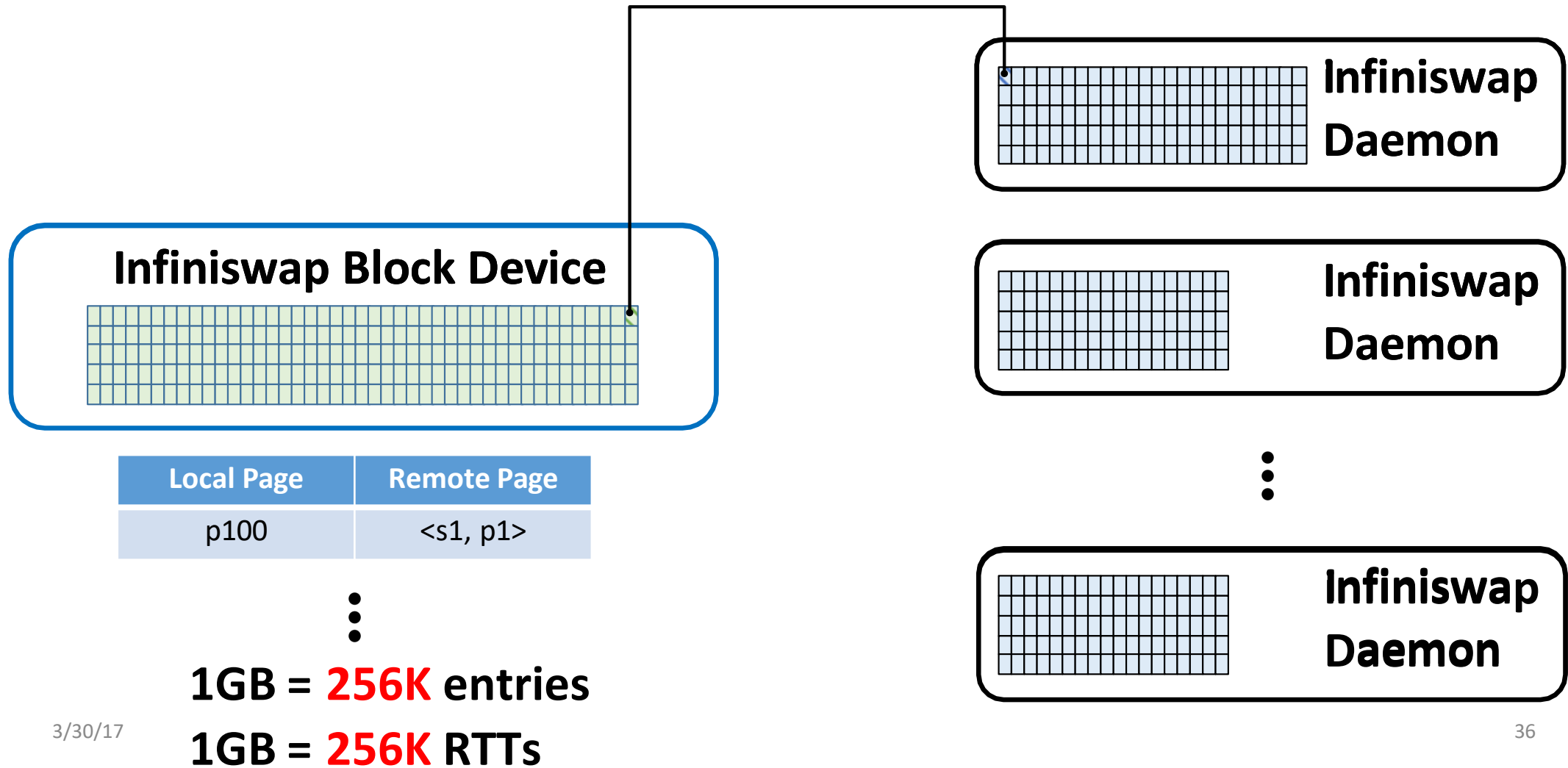
Remote paging

Local backup disk

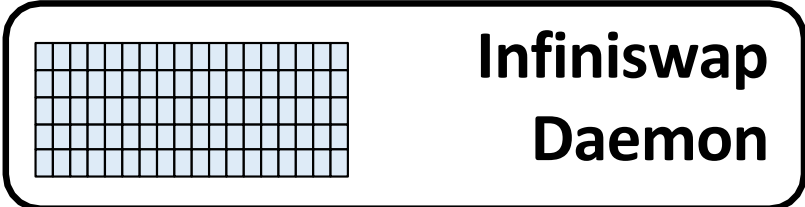
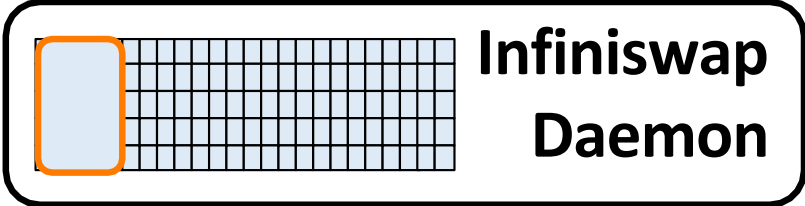
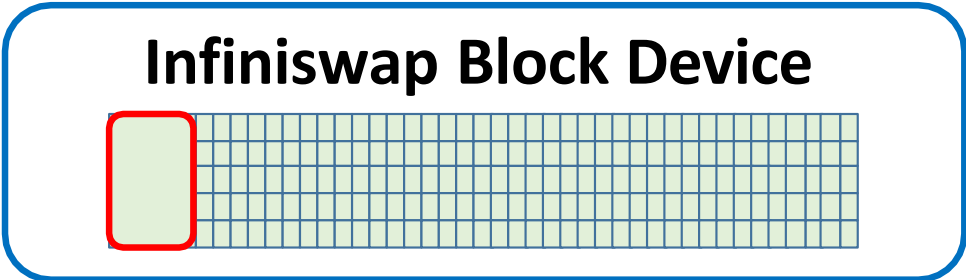
**Decentralized remote memory  
management**



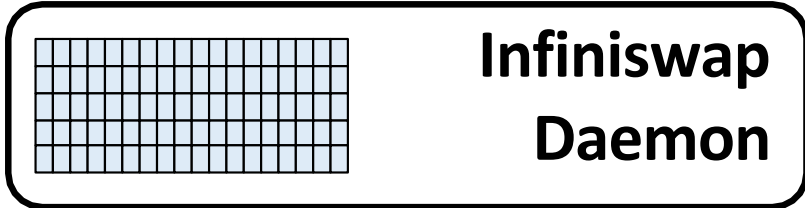
# Management unit: memory page?



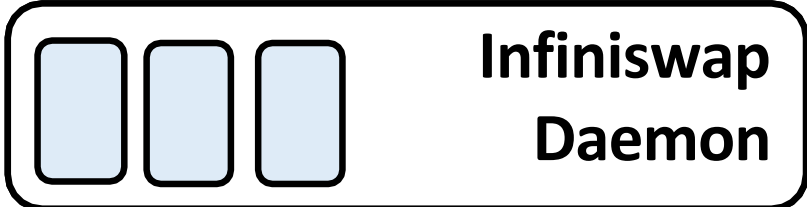
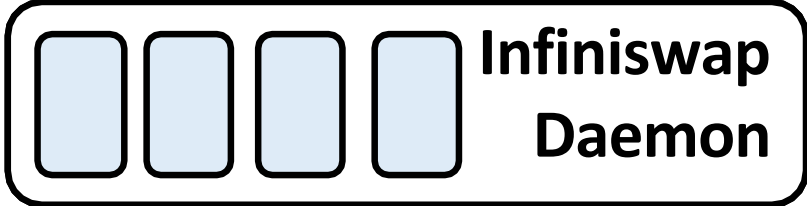
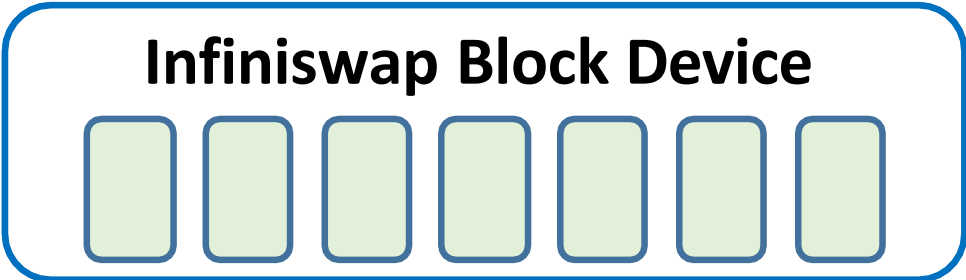
# Management unit: memory slab!



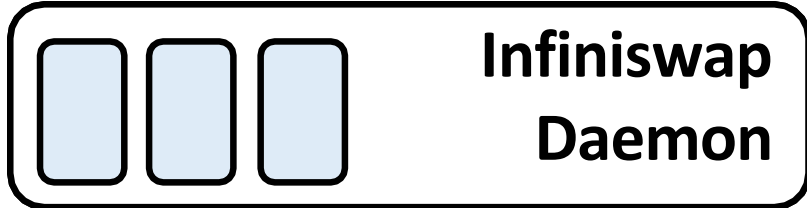
⋮



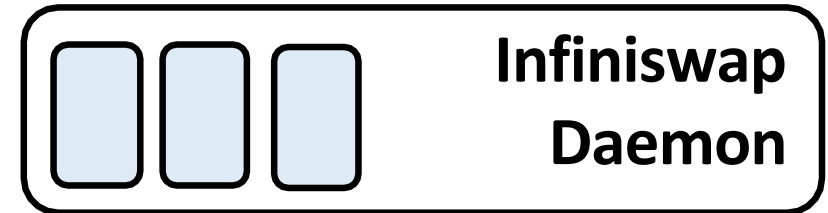
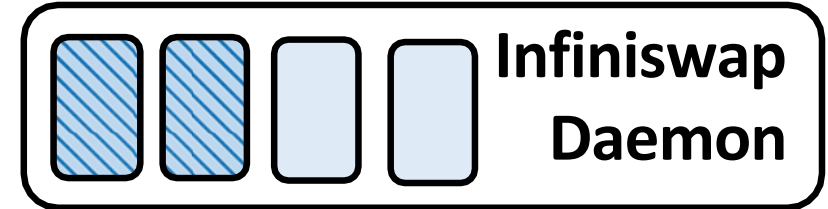
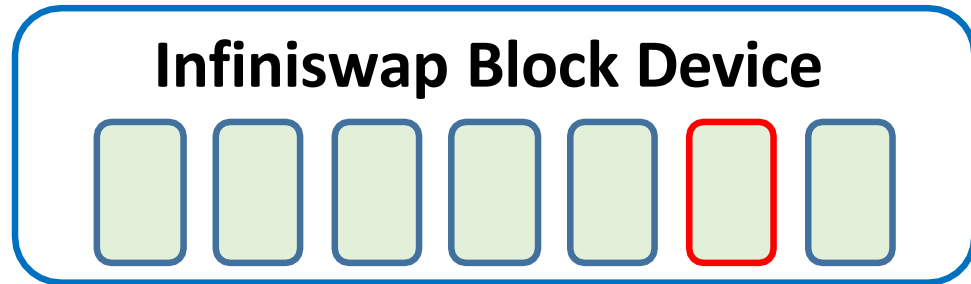
# Management unit: memory slab!



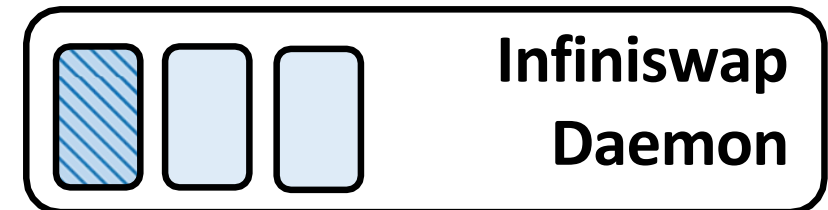
⋮



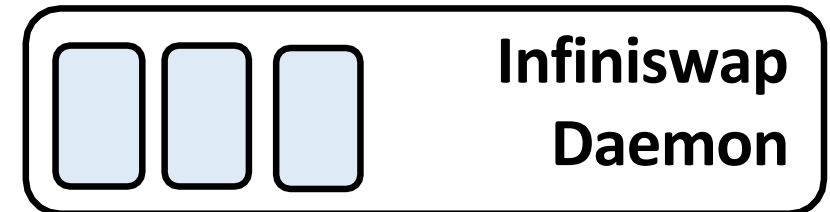
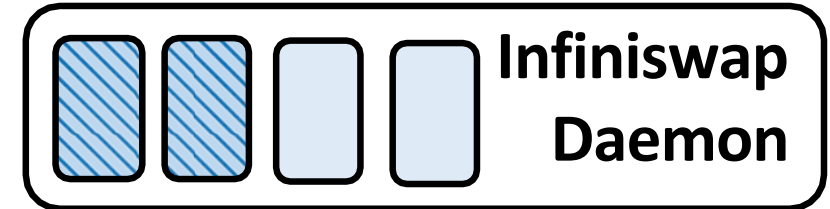
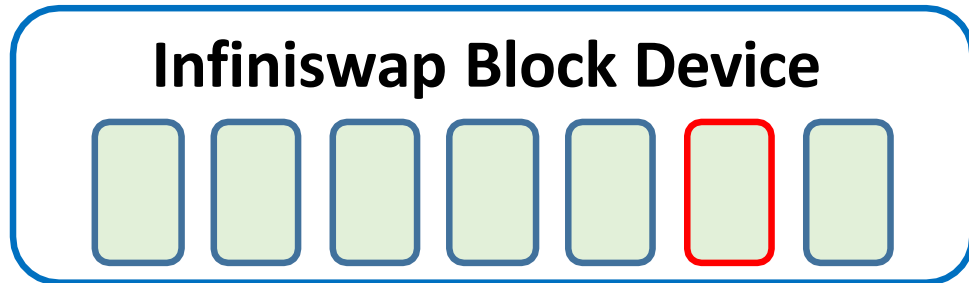
# Which remote machine should be selected?



⋮



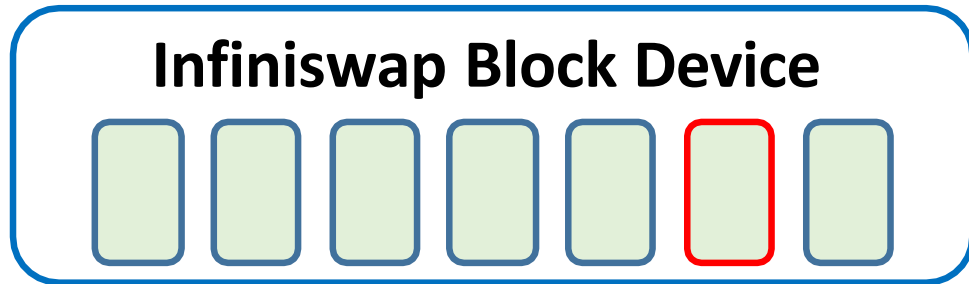
# Which remote machine should be selected?



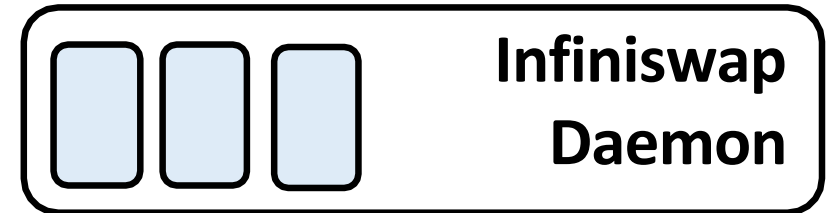
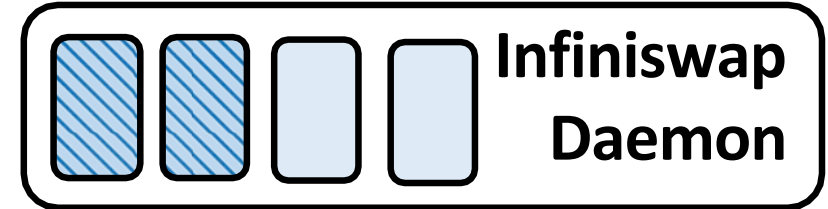
⋮

Goal: **balance** memory utilization

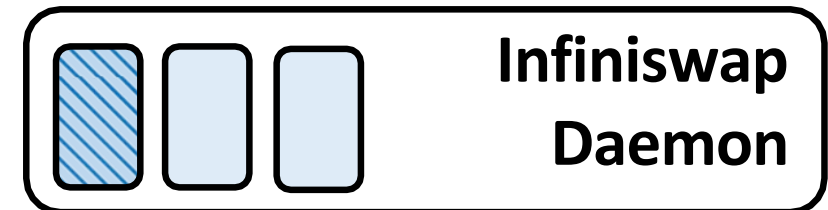
# Which remote machine should be selected?



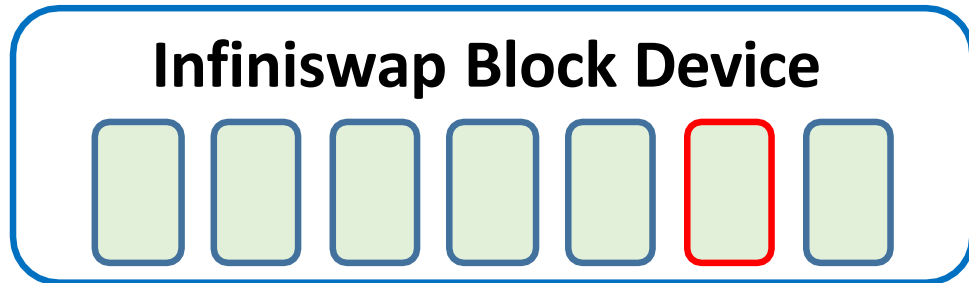
► **Central controller**



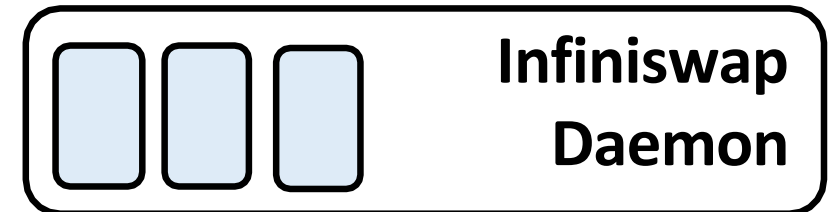
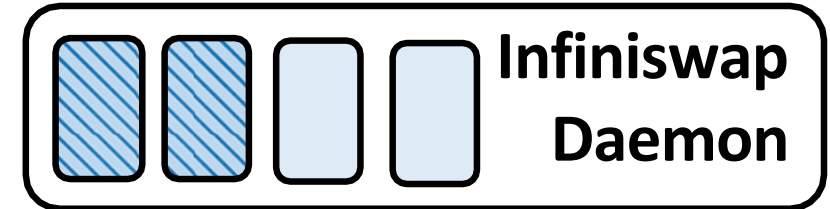
⋮



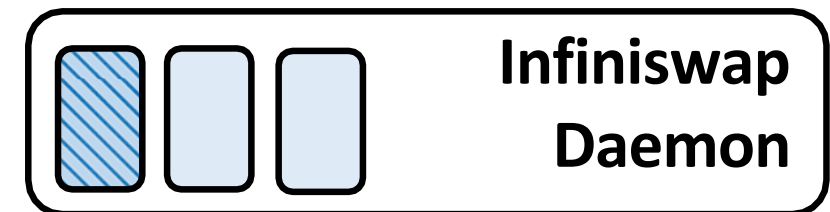
# Which remote machine should be selected?



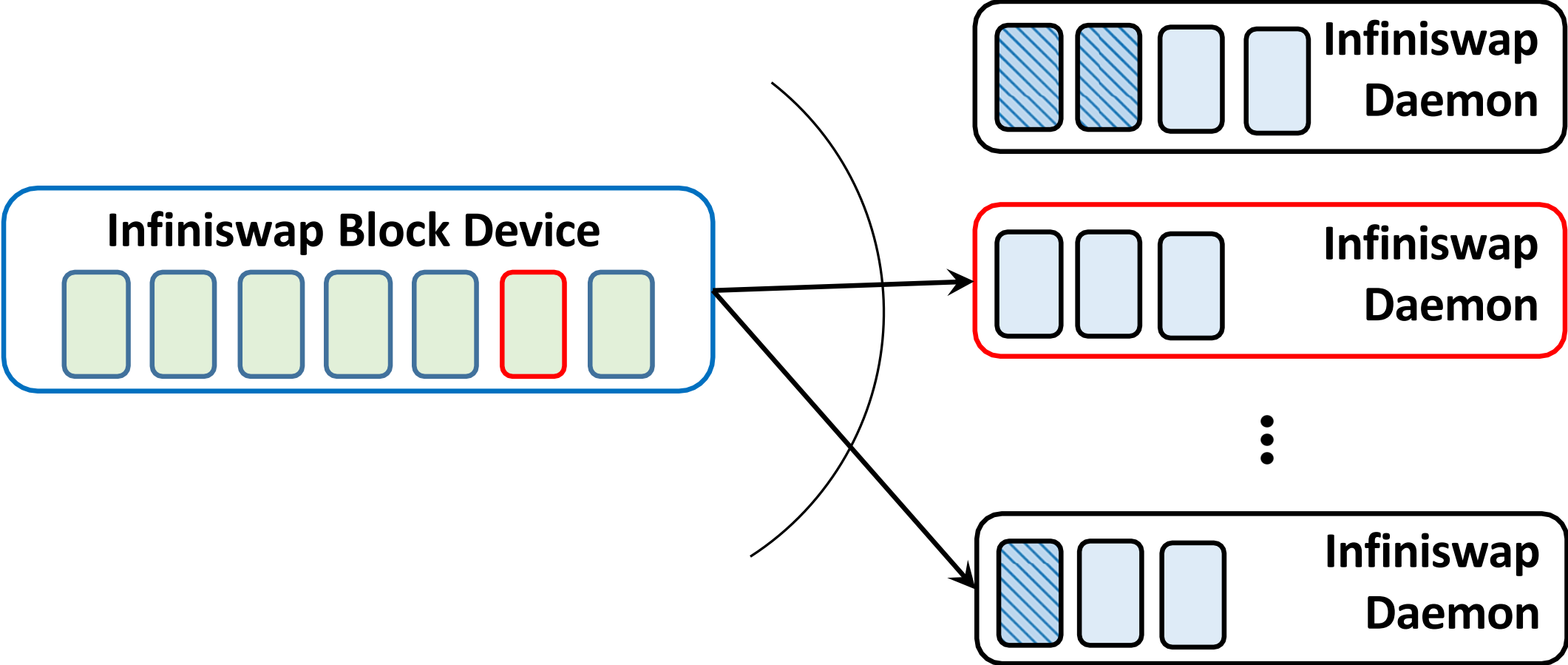
- ~~▶ Central controller~~
- ▶ Decentralized approach



⋮



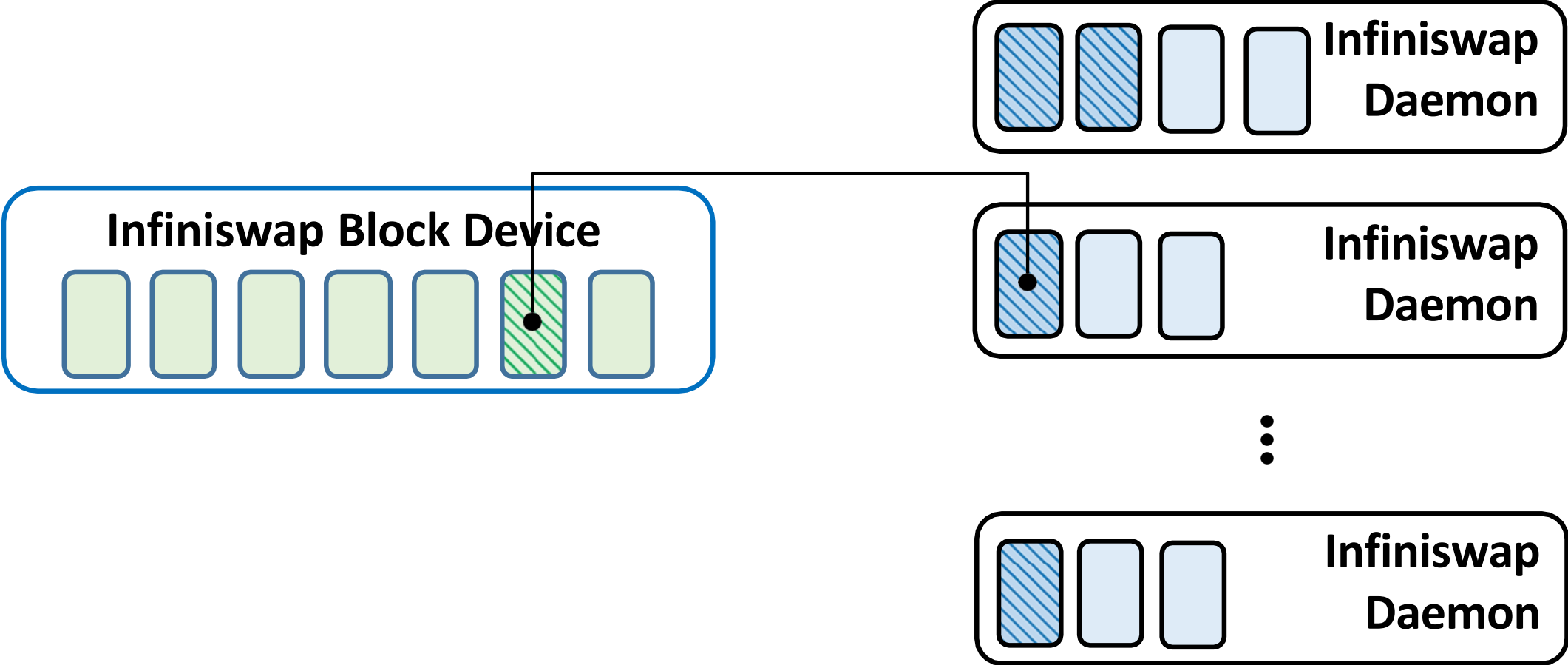
# Power of two choices<sup>[1]</sup>



[1] Mitzenmacher, Michael. "The power of two choices in randomized load balancing.", Ph.D. thesis, U.C. Berkeley, 1996



# Power of two choices<sup>[1]</sup>



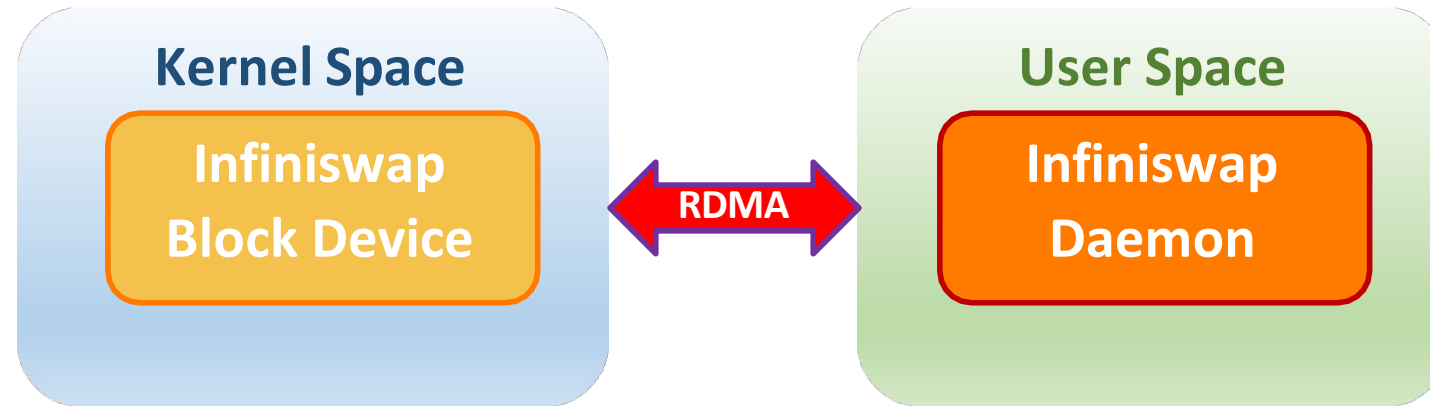
[1] Mitzenmacher, Michael. "The power of two choices in randomized load balancing.", Ph.D. thesis, U.C. Berkeley, 1996

# Agenda

- Motivation and related work
- Design and system overview
- **Implementation and evaluation**
- Future work and conclusion

3/30/17

# Implementation



- **Connection Management**

- **One** RDMA connection per active block device - daemon pair

- **Control Plane**

- **SEND, RECV**

- **Data Plane**

- **One-sided** RDMA READ, WRITE

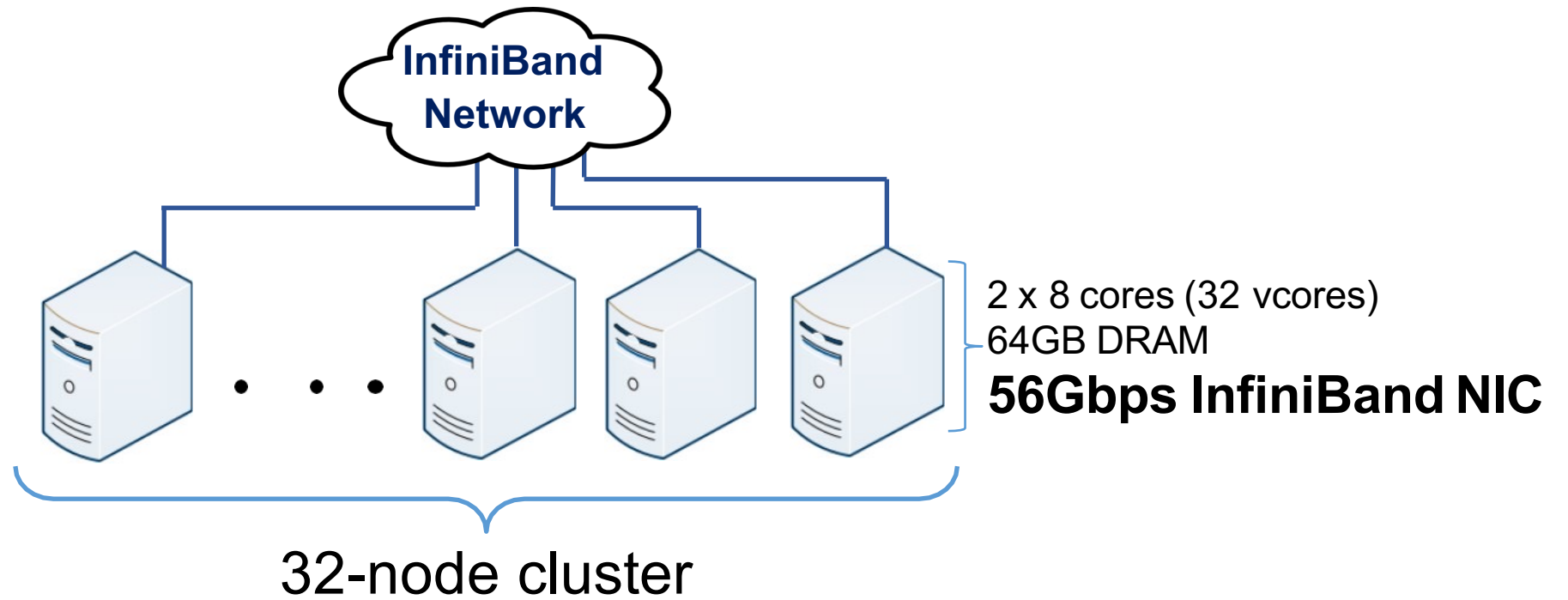
3/30/17

# What are we expecting from Infiniswap?

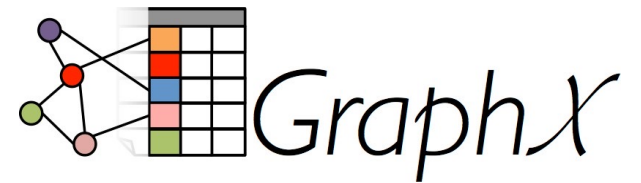
- **Application performance**
- **Cluster memory utilization**
- Network usage
- Eviction overhead
- Fault-tolerance overhead
- Performance as a block device

⋮

# Evaluation

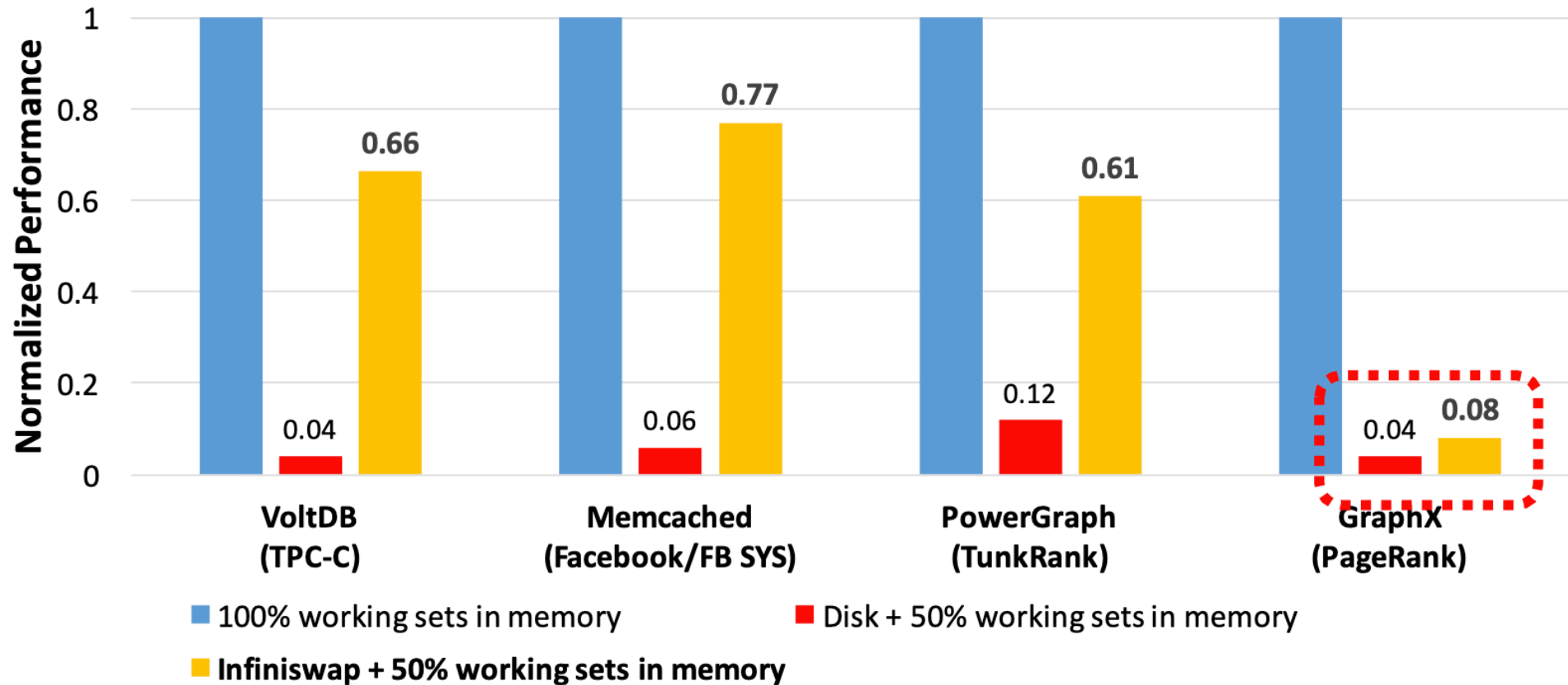


3/30/17



# Application performance

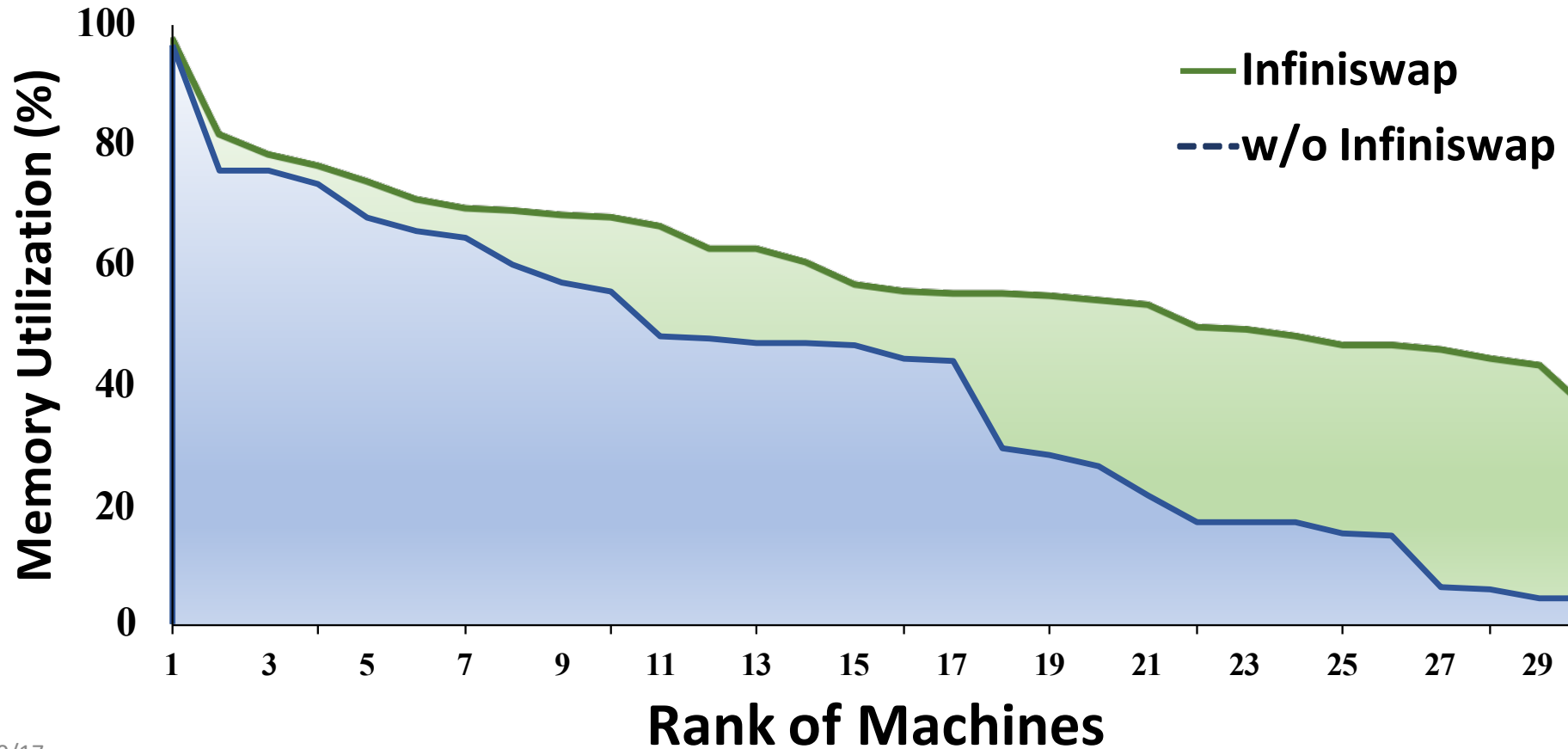
- 50% working sets in memory



- Application performance is improved by 2-16x

# Cluster memory utilization

- 90 containers (applications), mixing all applications and memory constraints.



3/30/17

60

- Cluster memory utilization is improved from **40.8%** to **60%** (1.47x)

# Agenda

- Motivation and related work
- Design and system overview
- Implementation and evaluation
- **Future work and conclusion**



# Limitations and future work

- **Trade-off in fault-tolerance**
  - Local disk is the bottleneck
  - Multiple remote replicas
    - Fault-tolerance vs. space-efficiency
- **Performance isolation among applications**

# Conclusion

- **Infiniswap: remote paging over RDMA**
  - Application performance
  - Cluster memory utilization
- **Efficient, practical memory disaggregation**
  - No hardware design
  - No application modification
  - **Fault-tolerance**
  - **Scalability**

<https://github.com/Infiniswap/infiniswap.git>

# Memory Management in Modern Computer Systems

- Memory Abstraction
  - NSDI'14 FaRM
- Demand paging: remote memory over RDMA
  - NSDI'17 InfiniSwap
  - OSDI'20 AIFM
- Demand paging: memory swapping between GPU memory and host memory
  - OSDI'20 PipeSwitch

# AIFM: High-Performance, Application-Integrated Far Memory

Zain (Zhenyuan) Ruan\*   Malte Schwarzkopf†   Marcos K. Aguilera‡   Adam Belay\*

\*MIT CSAIL

†Brown University

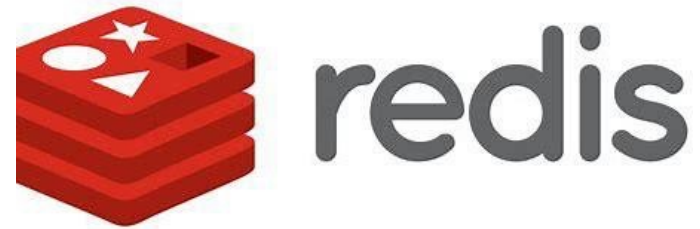
‡VMware Research



# In-Memory Applications



Data Analytics



Web Caching



Database



Graph Processing

# Memory Is Inelastic

- Limited by the server physical boundary.
- Applications cannot overcommit memory.

## Opening a 20GB file for analysis with pandas

Asked 2 years, 8 months ago   Active 1 year, 4 months ago   Viewed 81k times



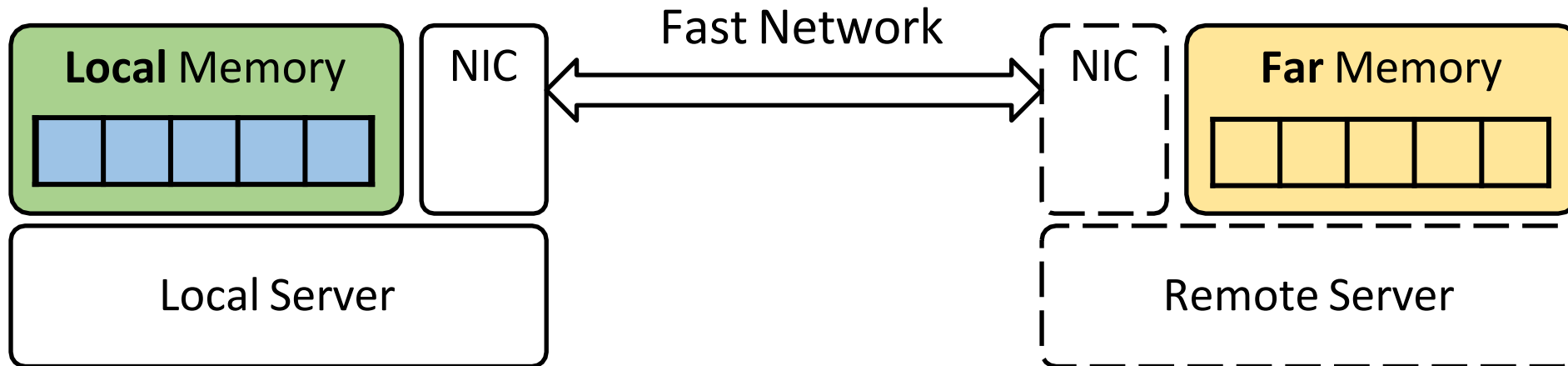
20

I am currently trying to open a file with pandas and python for machine learning purposes it would be ideal for me to have them all in a DataFrame. My RAM is 32 GB. I keep getting memory errors.

- Expensive solution: overprovision memory for peak usage.

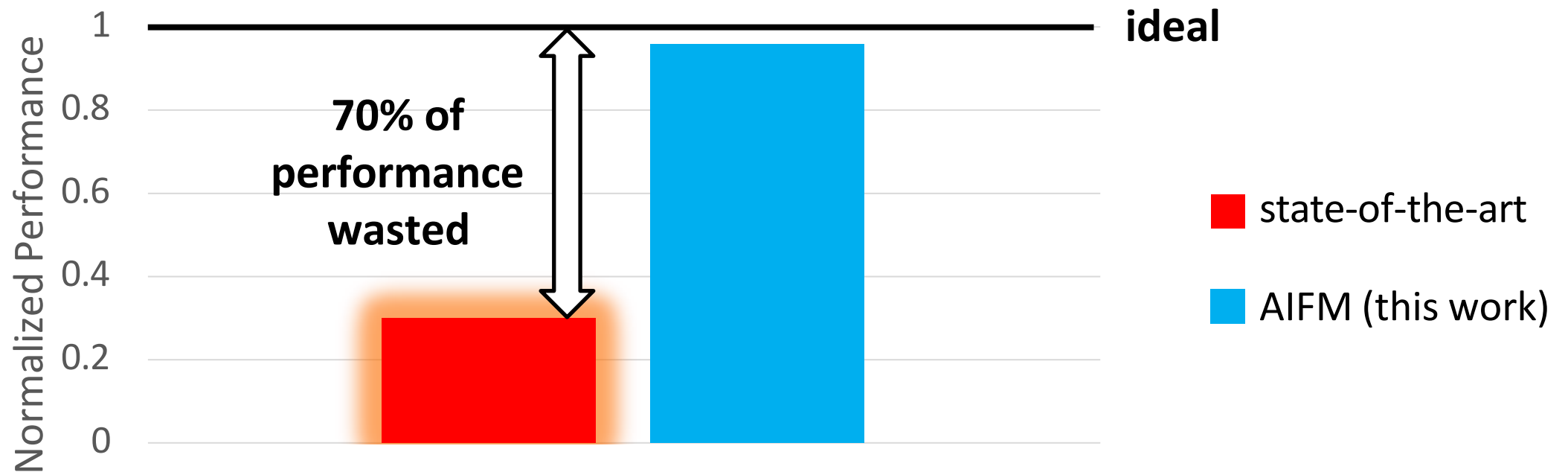
# Trending Solution: Far Memory

- Leverage the idle memory of remote servers (with fast network).



# Existing Far-Memory Systems Perform Poorly

- Real-world Data Analytics from Kaggle.
  - Provision **25%** of working set in local mem.
- Goal: reclaim the wasted performance.



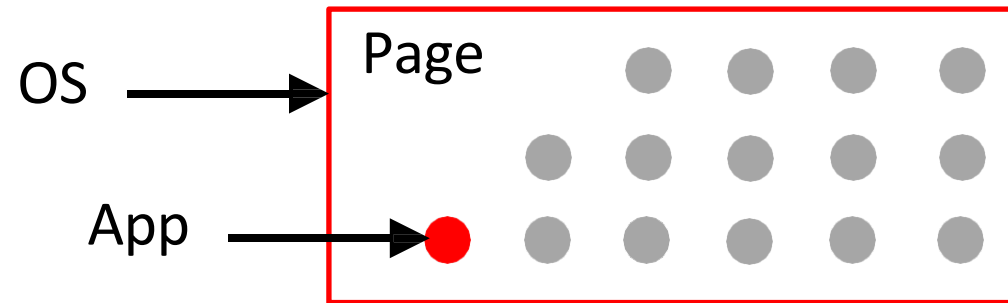


# Why Do Existing Systems Waste Performance?

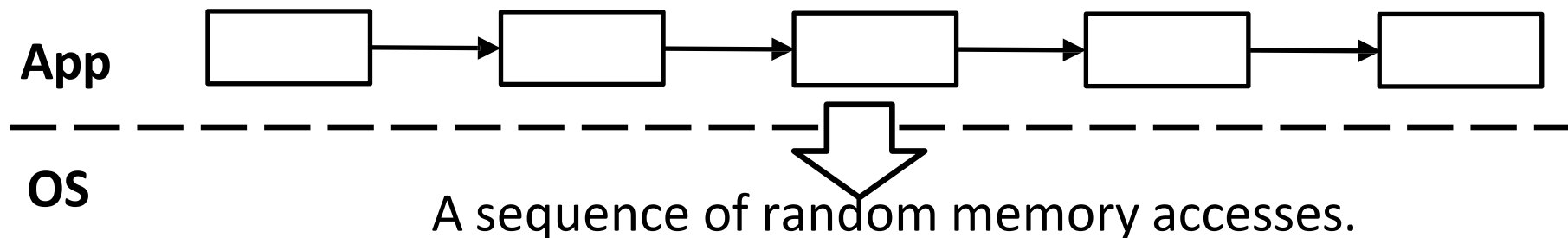
- Problem: based on **OS paging**.
  - Semantic gap.
  - High kernel overheads.

# Challenge 1: Semantic Gap

- Page granularity → **R/W amplification.**

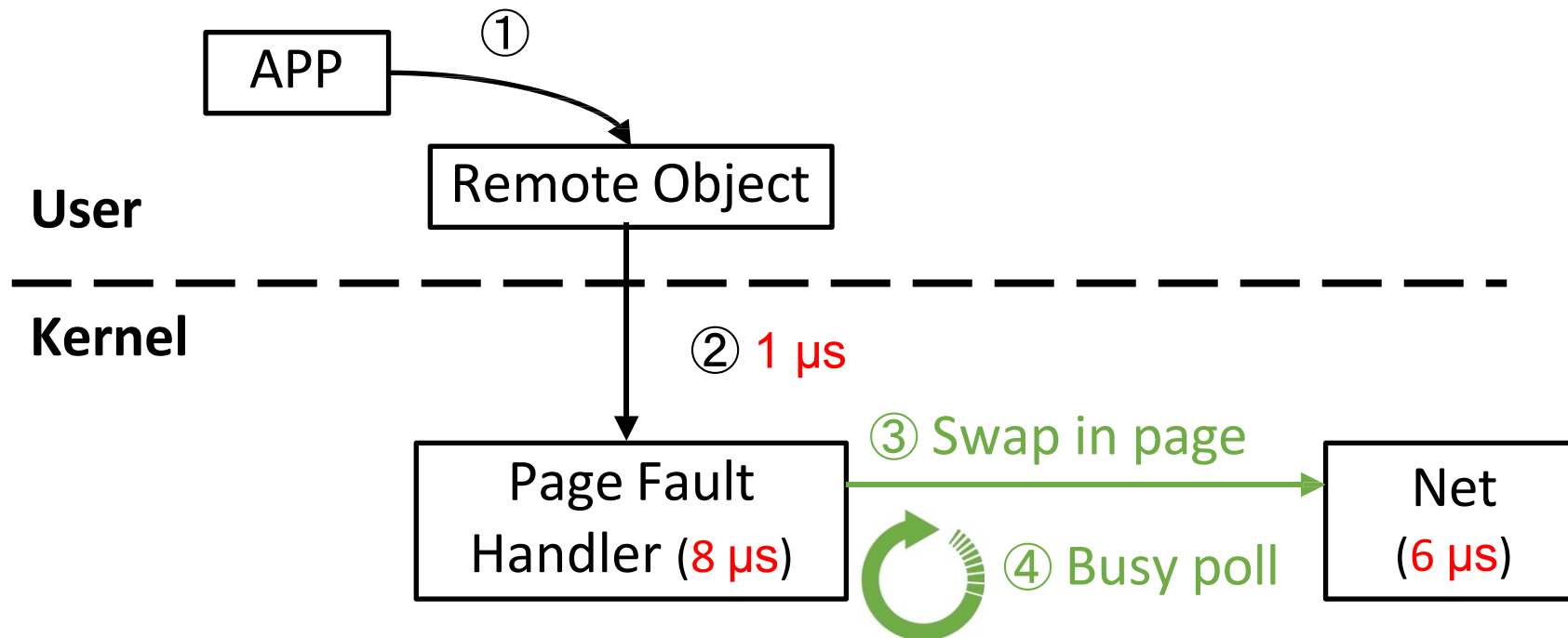


- OS lacks app knowledge → **hard to prefetch, etc.**

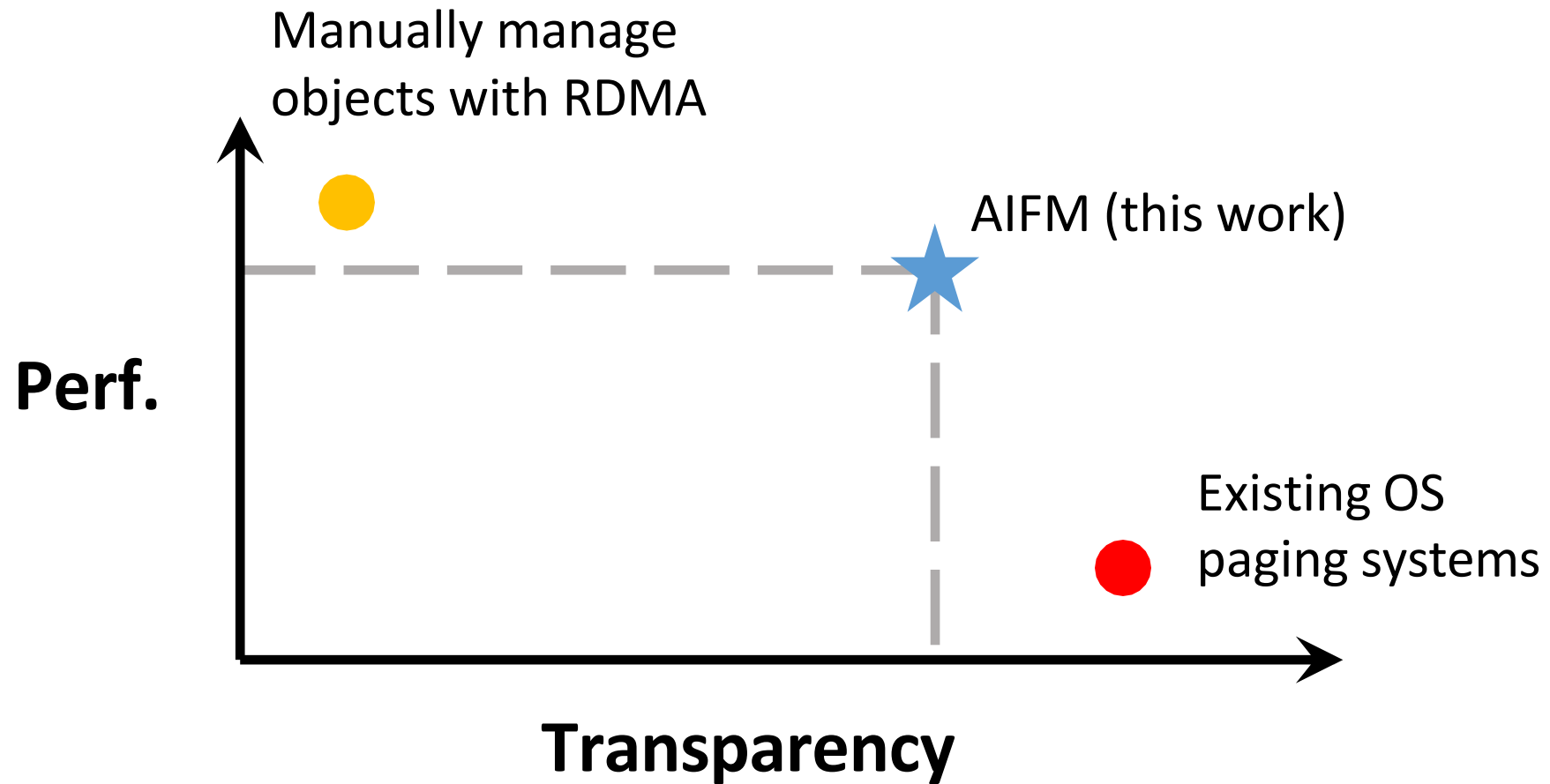


# Challenge 2: High Kernel Overheads

- **Expensive page faults.**
- Busy Polling for in-kernel net I/O → **burn CPU cycles.**



# Design Space



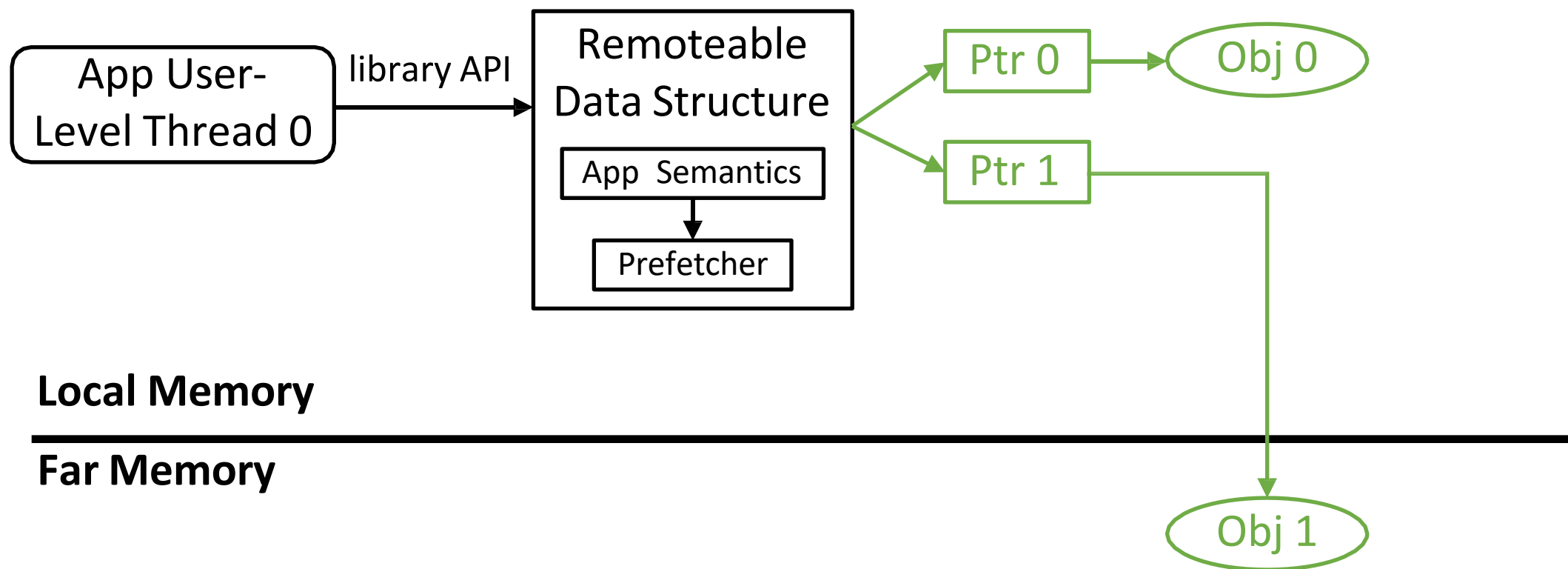
# AIFM's Design Overview

➤ Key idea: swap memory using a userspace runtime.

Challenge	Solution
<b>1. Semantic gap</b> (Amplification, Hard to prefetch)	Remoteable Data structure library
<b>2. Kernel overheads</b> (page faults, busy poll for net I/O)	Userspace runtime
<b>3. Impact of Memory Reclamation</b> (pause app threads)	Pauseless evacuator
<b>4. network BW &lt; DRAM BW</b>	Remote Agent

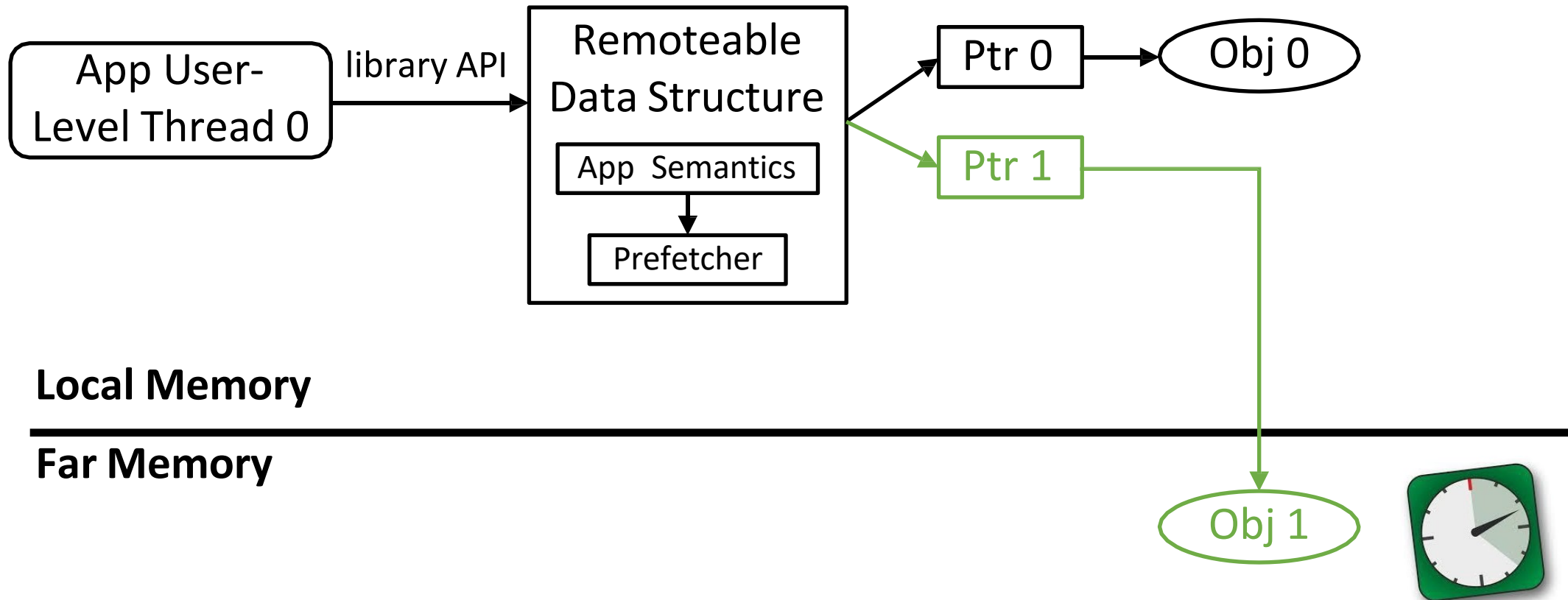
# 1. Remoteable Data Structure Library

➤ Solved challenge: semantic gap.



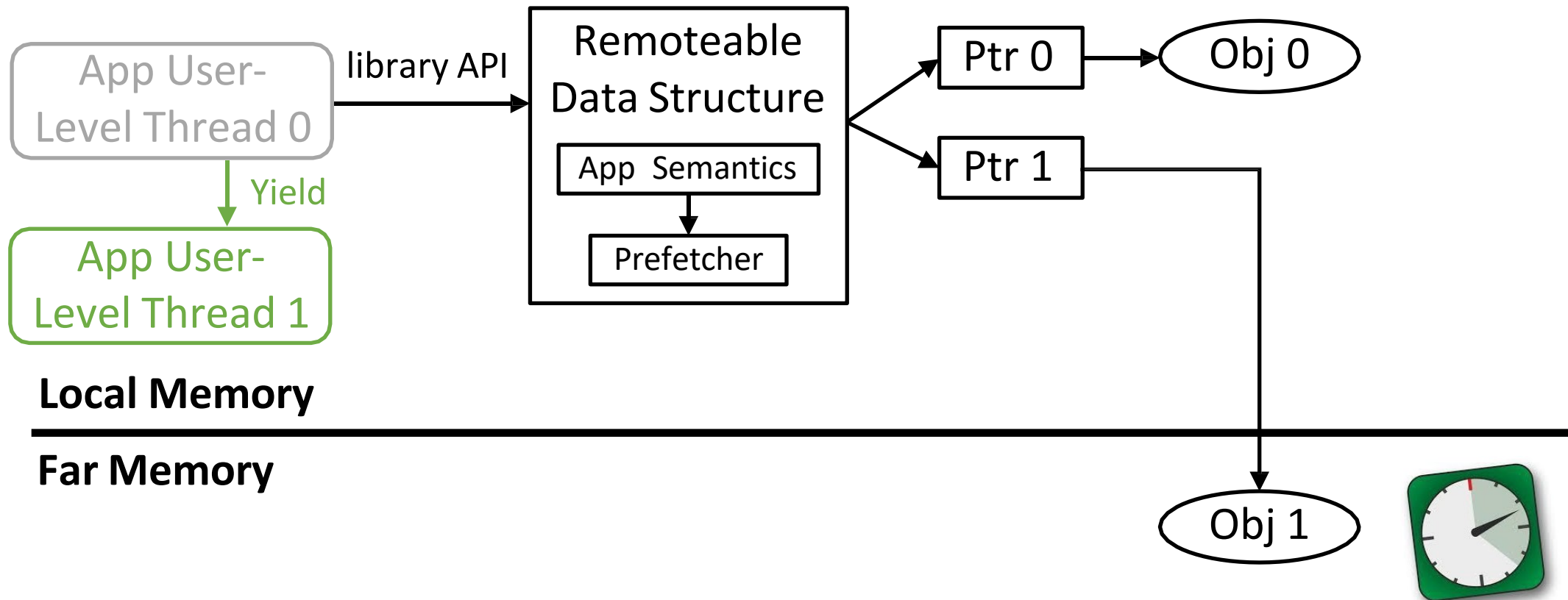
## 2. Userspace Runtime

➤ Solved challenge: kernel overheads.



## 2. Userspace Runtime

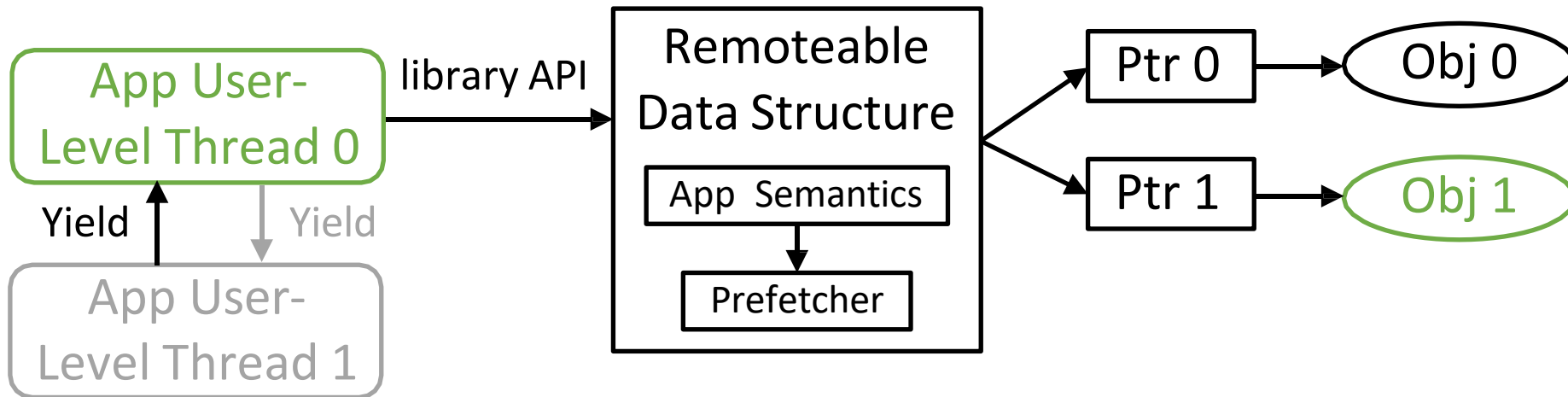
➤ Solved challenge: kernel overheads.





## 2. Userspace Runtime

➤ Solved challenge: kernel overheads.



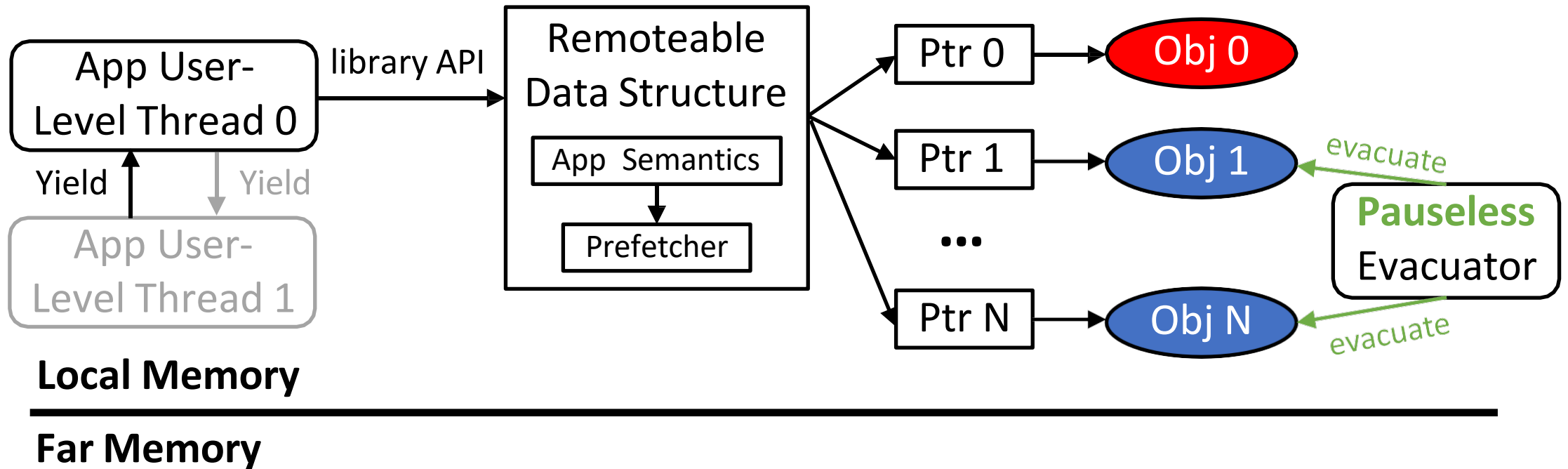
**Local Memory**

---

**Far Memory**

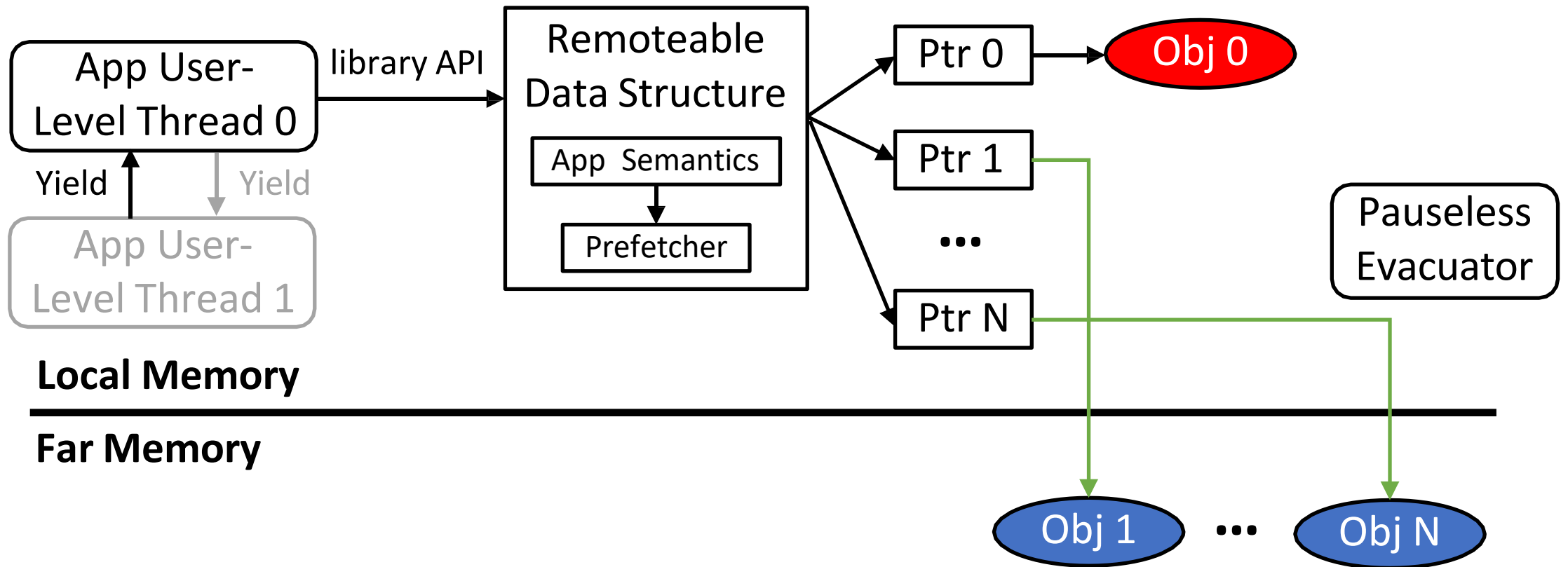
# 3. Pauseless Evacuator

➤ Solved challenge: impact of memory reclamation.



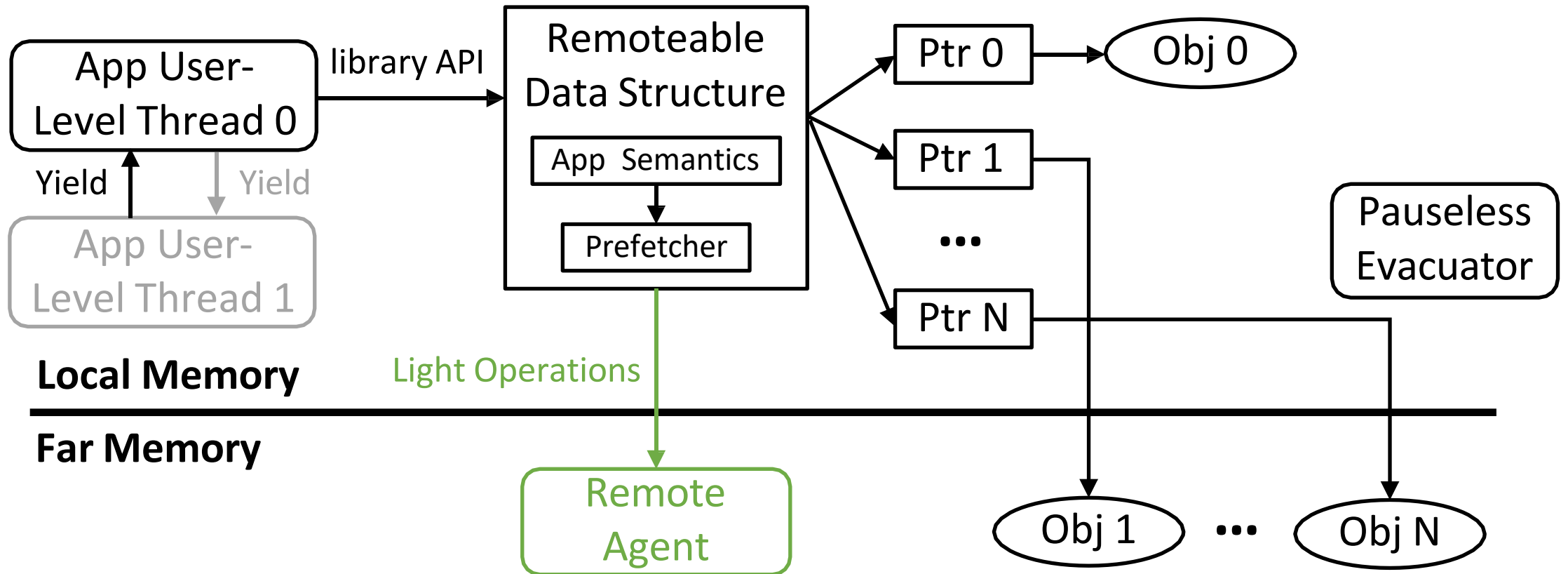
# 3. Pauseless Evacuator

➤ Solved challenge: impact of memory reclamation.



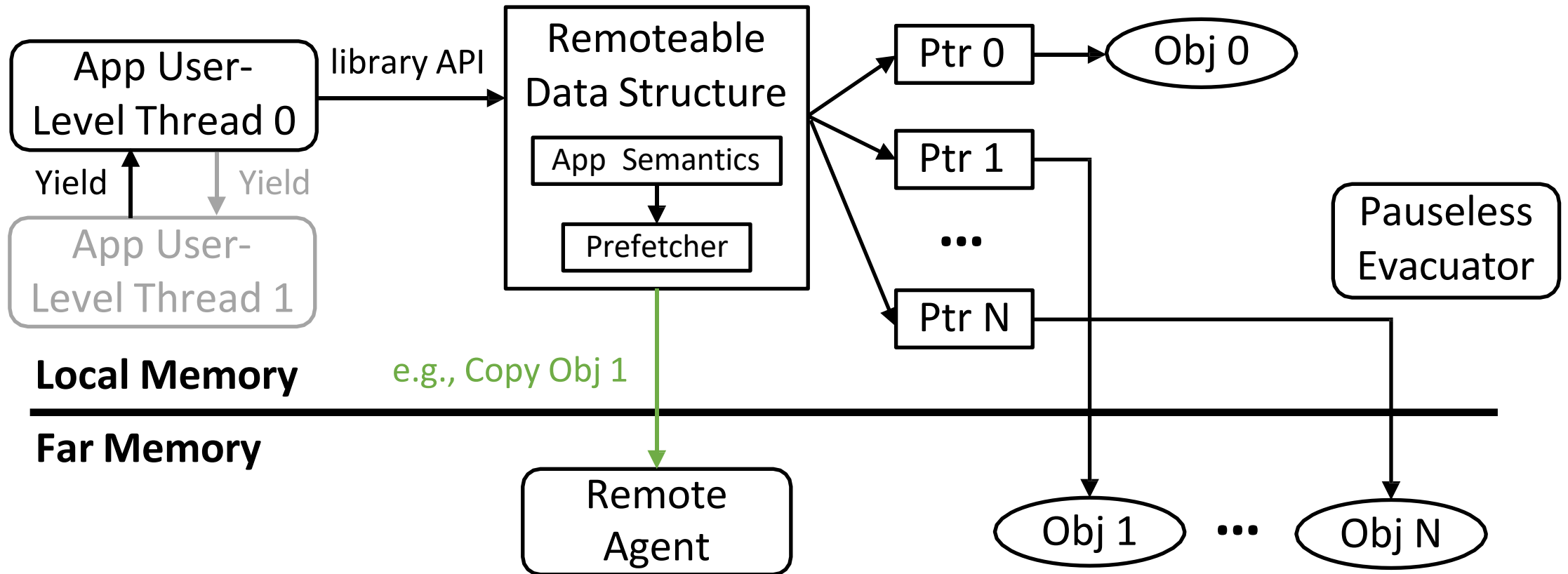
# 4. Remote Agent

➤ Solved challenge: network BW < DRAM BW.



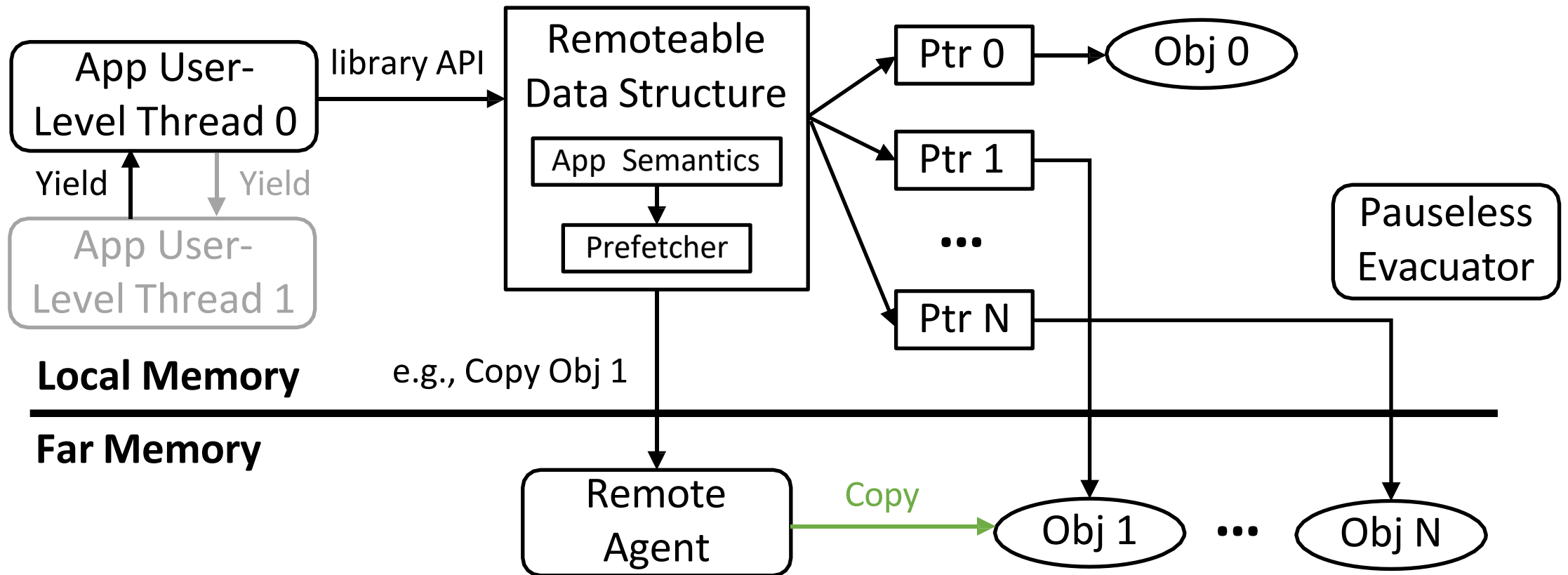
# 4. Remote Agent

➤ Solved challenge: network BW < DRAM BW.



# 4. Remote Agent

➤ Solved challenge: network BW < DRAM BW.



# Sample Code

```
std::unordered_map<key_t, int> hashtable;  
std::array<LargeData> arr;
```

```
LargeData foo(std::list<key_t> &keys_list) {  
    int sum = 0;  
    for (auto key : keys_list) {  
  
        sum += hashtable.at(key);  
    }  
  
    LargeData ret = arr.at(sum);  
    return ret;  
}
```

# Sample Code

```
RemHashTable<key_t, int> hashtable;  
RemArray<LargeData> arr;
```

```
LargeData foo(RemList<key_t> &keys_list) {
```

```
    int sum = 0;
```

```
    for (auto key : keys_list) {
```

*Prefetch list data.*

```
        DerefScope scope;
```

```
        sum += hashtable.at(key, scope);
```

*Cache hot objects.*

```
    }
```

```
    DerefScope scope;
```

```
    LargeData ret = arr.at</*don't cache*/ true>(sum, scope);
```

*Avoid polluting local mem.*

```
    return ret;
```

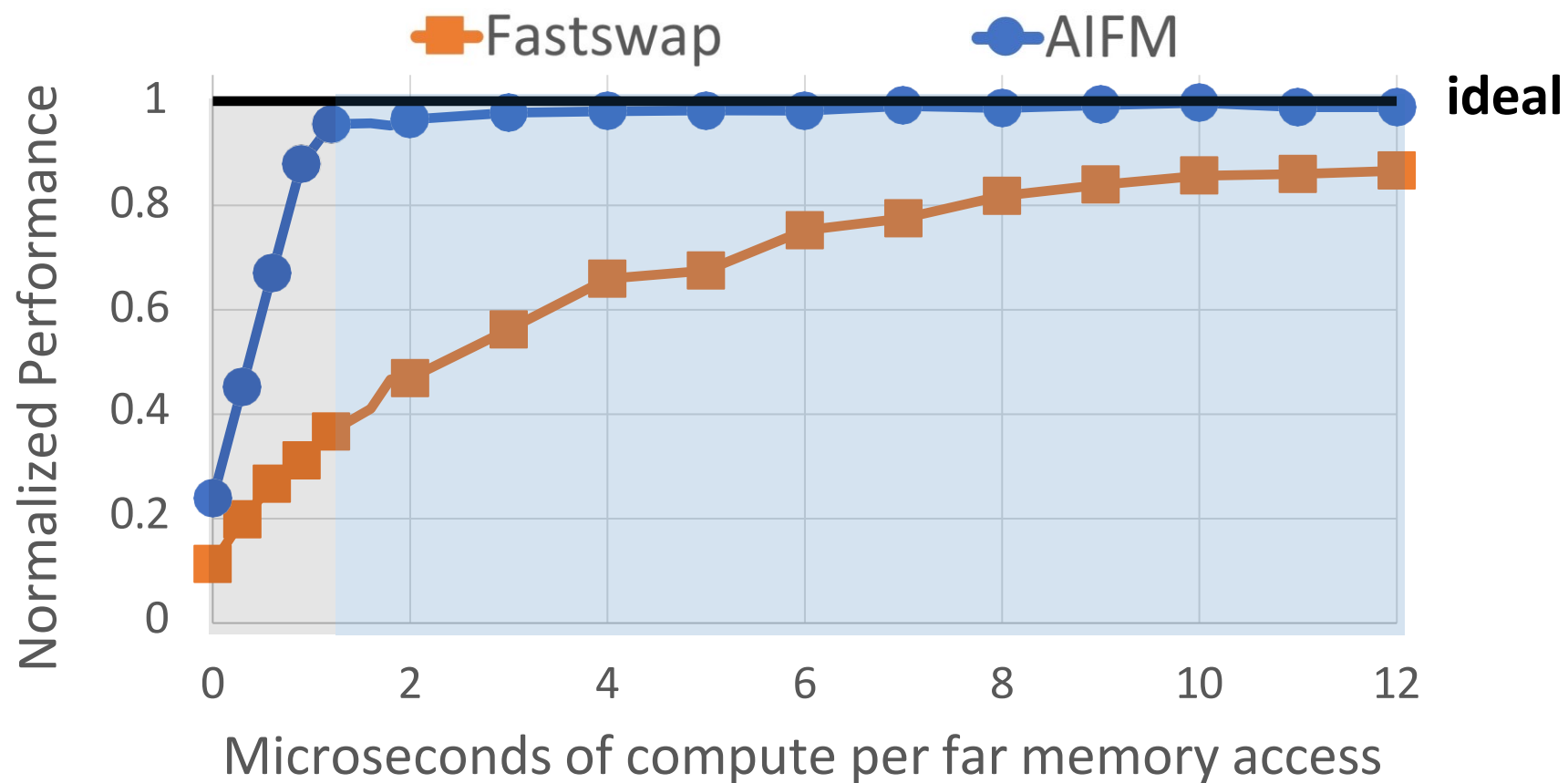
```
}
```



# Implementation

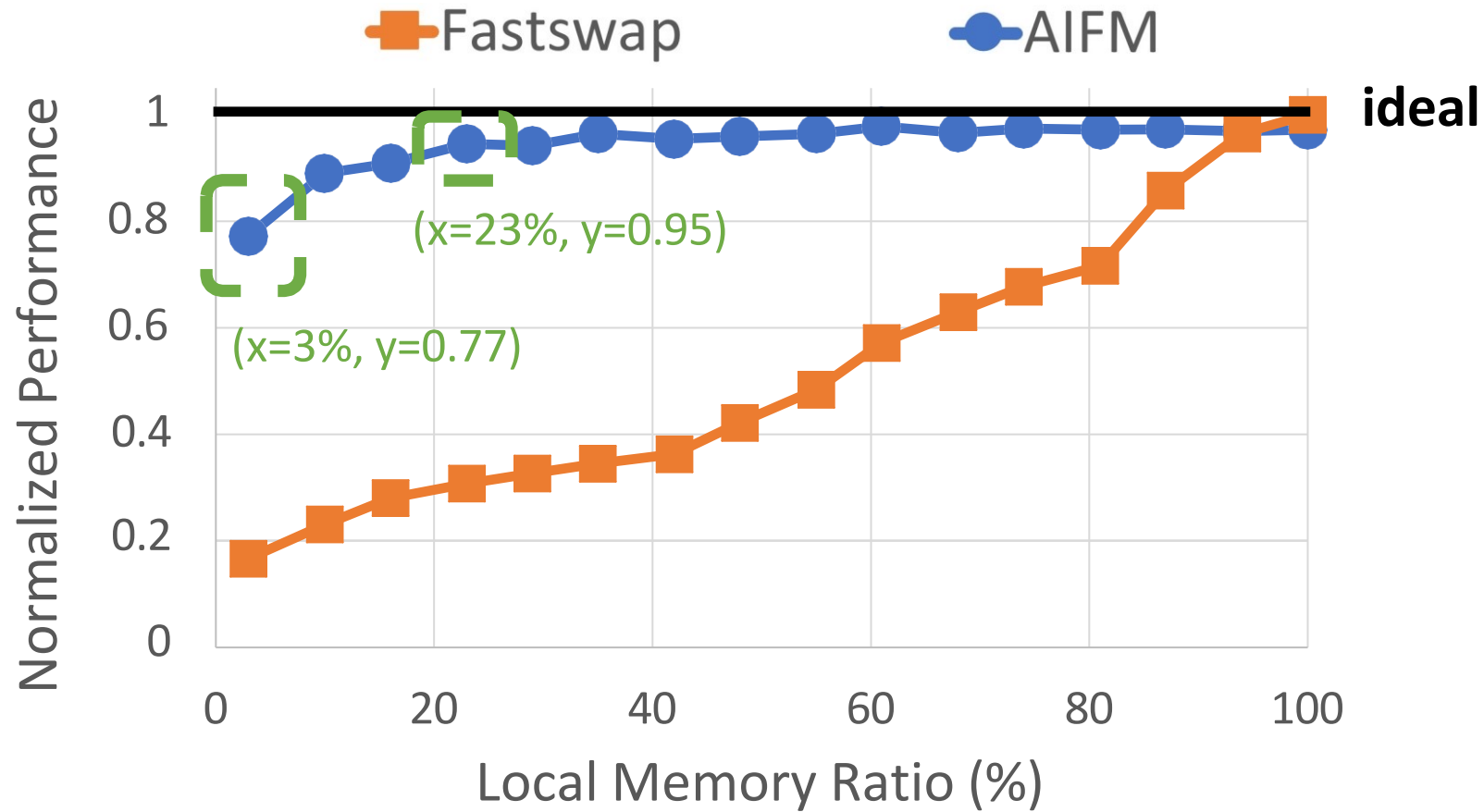
- Implemented 6 data structures.
  - Array, List, Hashtable, Vector, Stack, and Queue.
- Runtime is built on top of Shenango [NSDI' 19].
- TCP far-memory backend.
- LoC: 6.5K (runtime) + 5.5K (data structures) + 0.8K (Shenango)

# Performance on Different Compute Intensities



**AIFM hides far memory latency with moderate compute.**

# NYC Taxi Analysis (C++ DataFrame)



**AIFM achieves near-ideal performance with small local memory.**

# Other Experiments

- Synthetic web frontend: up to **13X end-to-end** speedup.
- Data structures microbenchmarks: up to **61X** speedup.
- Design Drill-Down.

Read our paper for details.

# Related Work

- OS-paging systems.
  - Fastswap [EuroSys' 20], Leap [ATC' 20]
- Distributed shared memory.
  - Treadmarks [IEEE Computer' 96]
- Garbage collection (GC).

# Conclusion

- AIFM: Application-Integrated Far Memory.
- Key idea: swap memory using a userspace runtime.
  - Data Structure Library: captures application semantics.
  - Userspace Runtime: efficiently manages objects and memory.
- Achieves 13X end-to-end speedup over Fastswap.
- Code released at <https://github.com/AIFM-sys/AIFM>

Please send your questions to us

[zainruan@csail.mit.edu](mailto:zainruan@csail.mit.edu)

# Memory Management in Modern Computer Systems

- Memory Abstraction
  - NSDI'14 FaRM
- Demand paging: remote memory over RDMA
  - NSDI'17 InfiniSwap
  - OSDI'20 AIFM
- Demand paging: memory swapping between GPU memory and host memory
  - OSDI'20 PipeSwitch

# *PipeSwitch*: Fast Pipelined Context Switching for Deep Learning Applications

Zhihao Bai, Zhen Zhang, Yibo Zhu, Xin Jin





Deep learning powers intelligent applications in many domains



# Training and inference

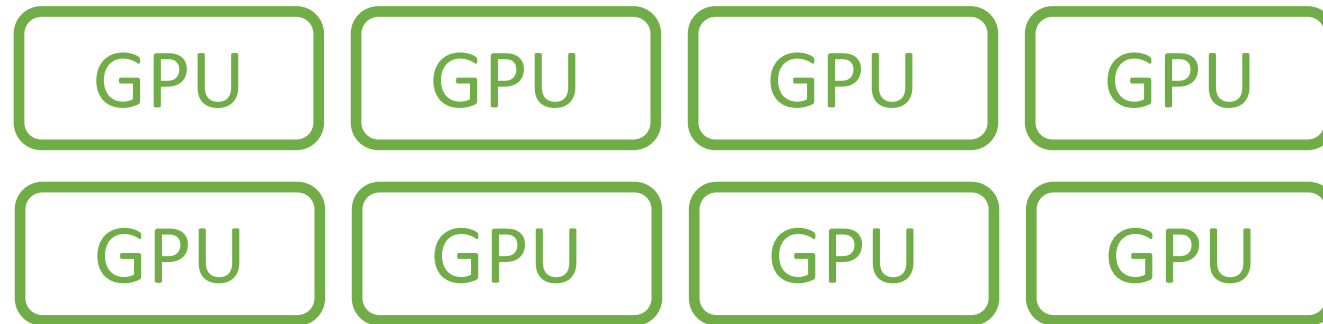


High throughput

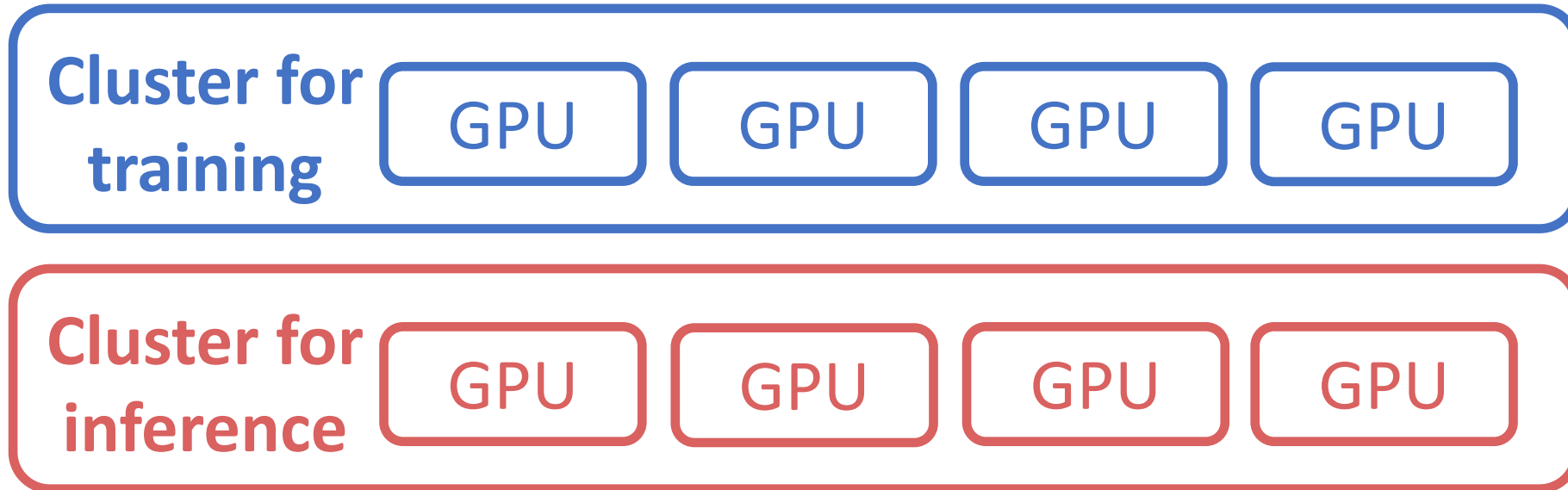


Low latency

# GPUs clusters for DL workloads

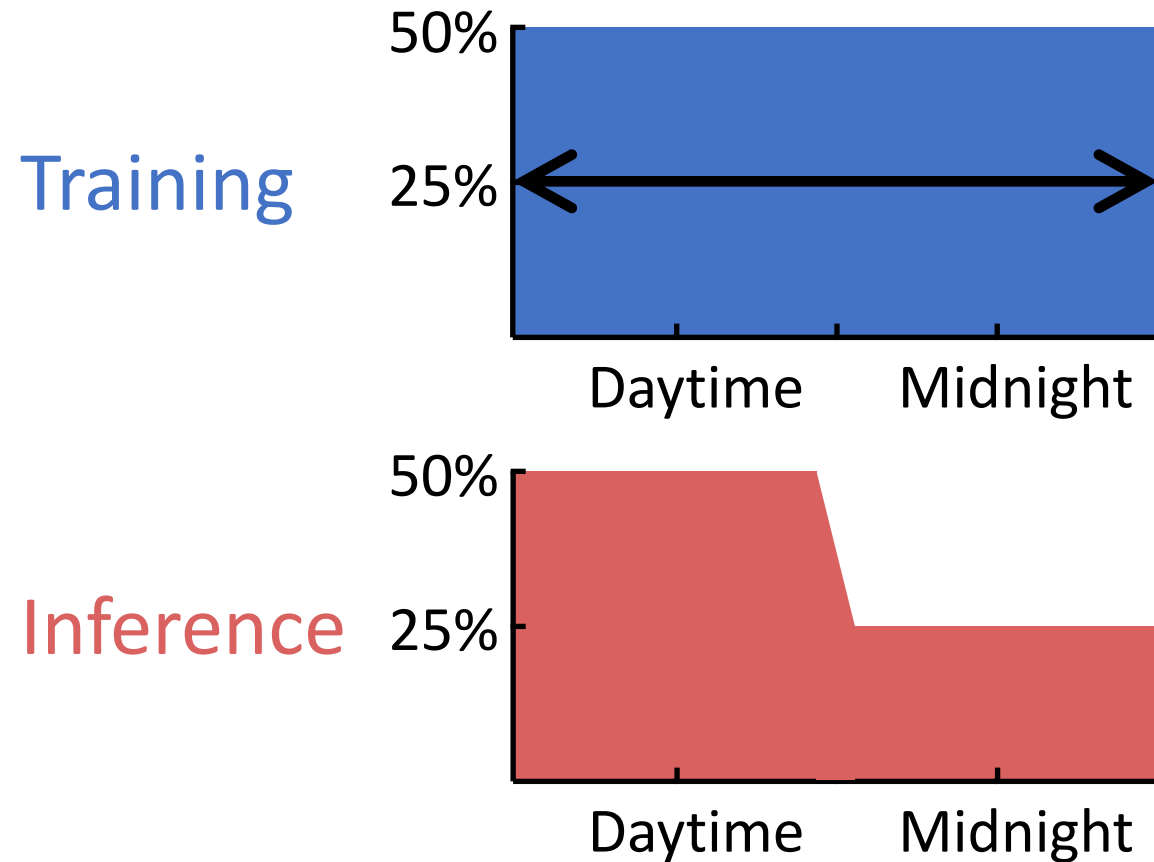


# Separate clusters for training and inference

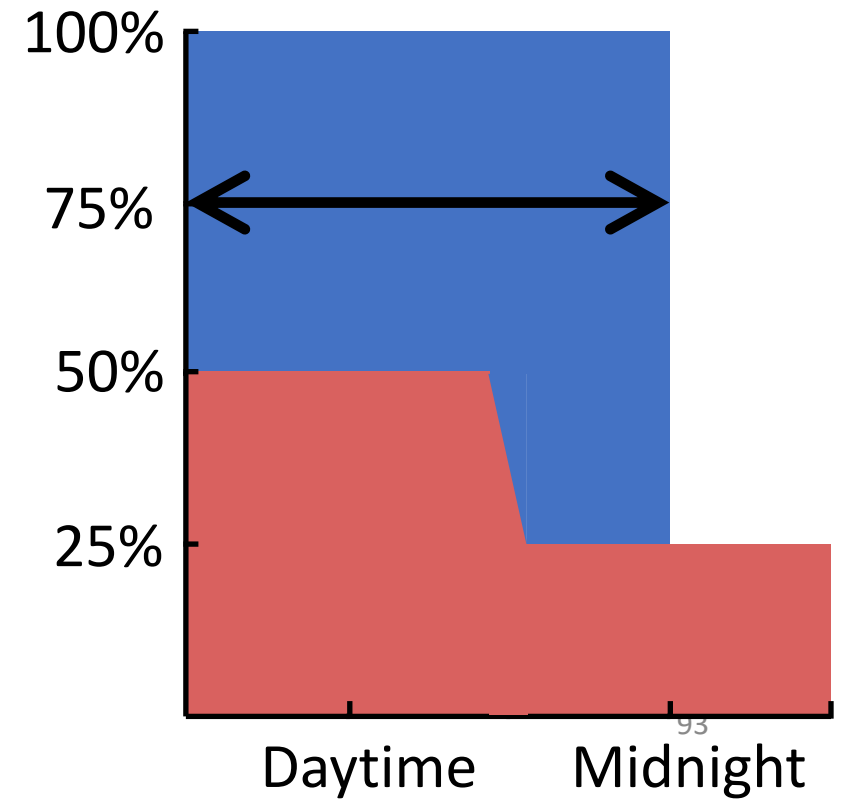


# Utilization of GPU clusters is low

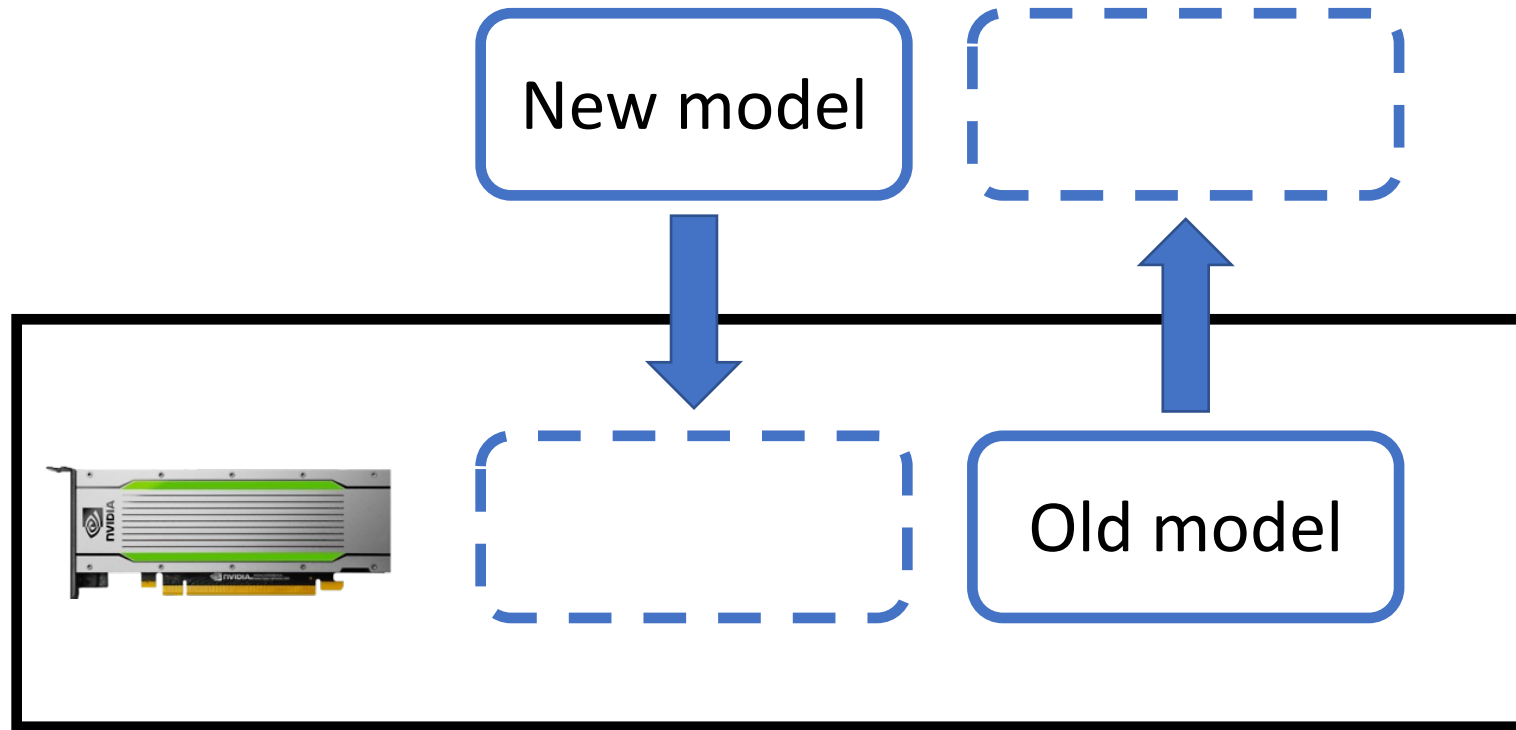
Today: separate clusters



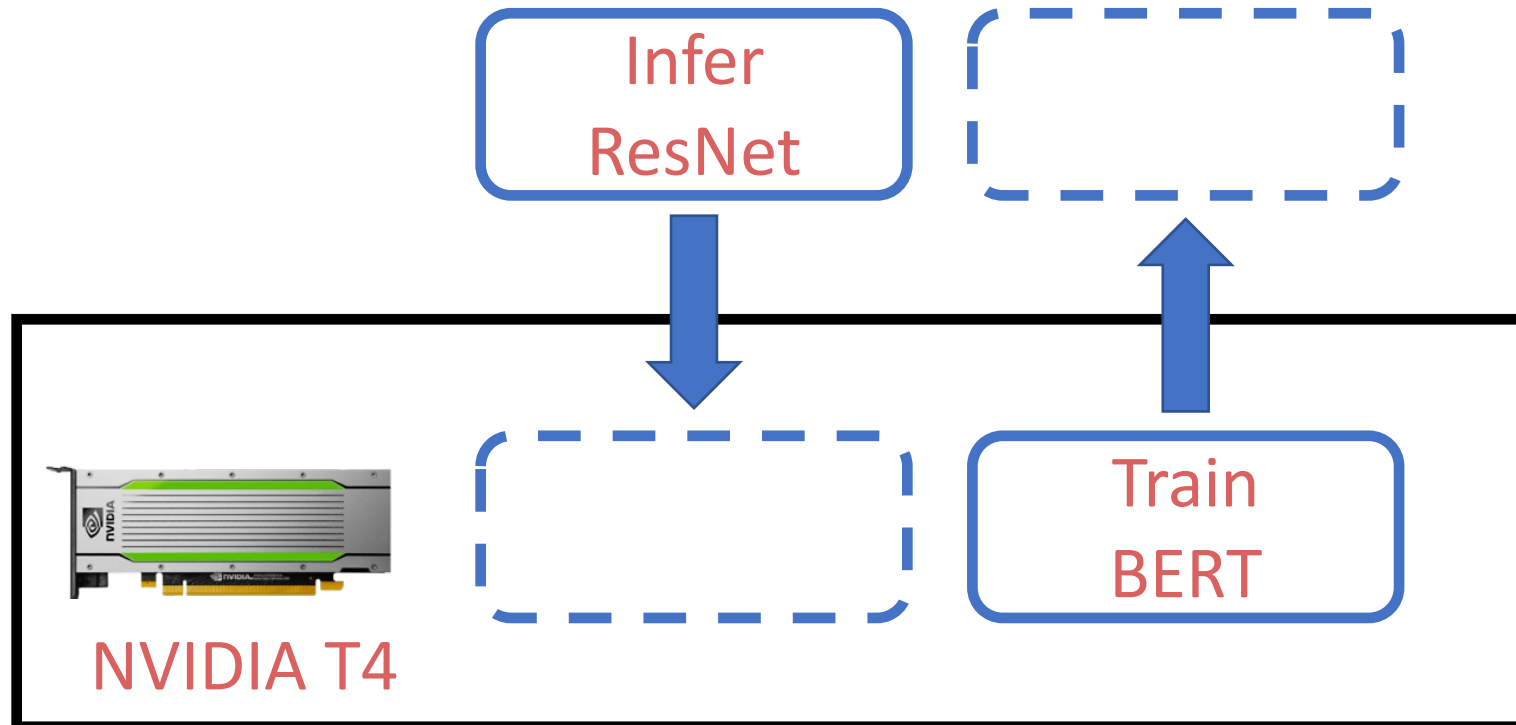
Ideal: shared clusters



# Context switching overhead is high




# Context switching overhead is high



**Latency: 6s**

# Drawbacks of existing solutions

- 
- NVIDIA MPS
    - High overhead due to contention
  - Salus[MLSys'20]
    - Requires all the models to be preloaded into the GPU memory

**Latency: 6s**



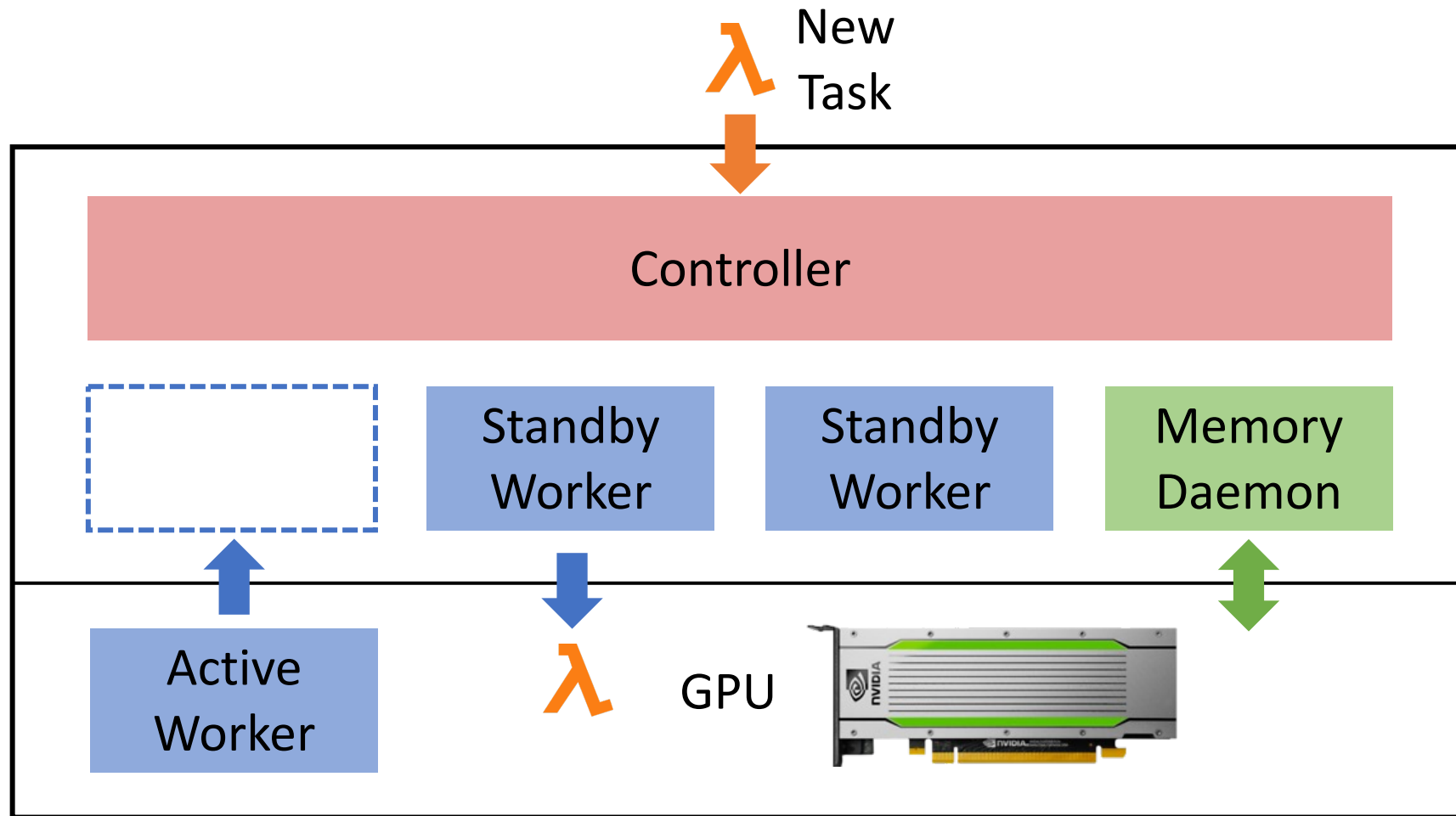
# Goal: fast context switching



- Enable GPU-efficient **multiplexing** of multiple DL apps with **fine-grained time-sharing**
- Achieve **millisecond-scale** context switching latencies and high throughput

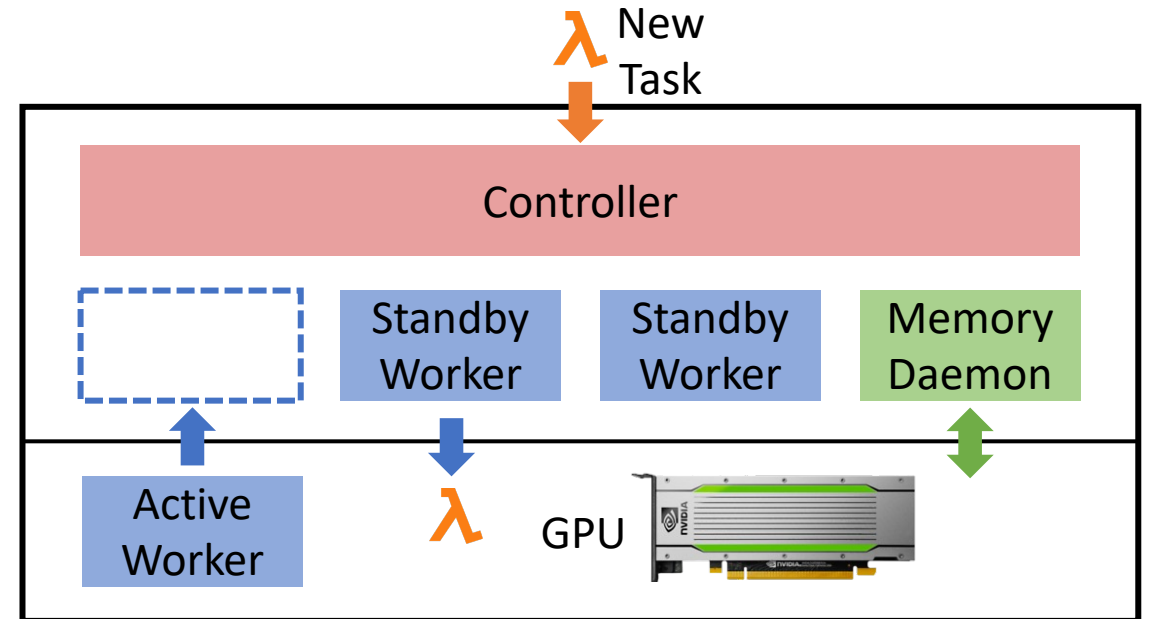
**Latency: 6s**

# PipeSwitch overview: architecture



# PipeSwitch overview: execution

- Stop the current task and prepare for the next task.
- Execute the task with pipelined model transmission.
- Clean the environment for the previous task.



# Sources of context switching overhead

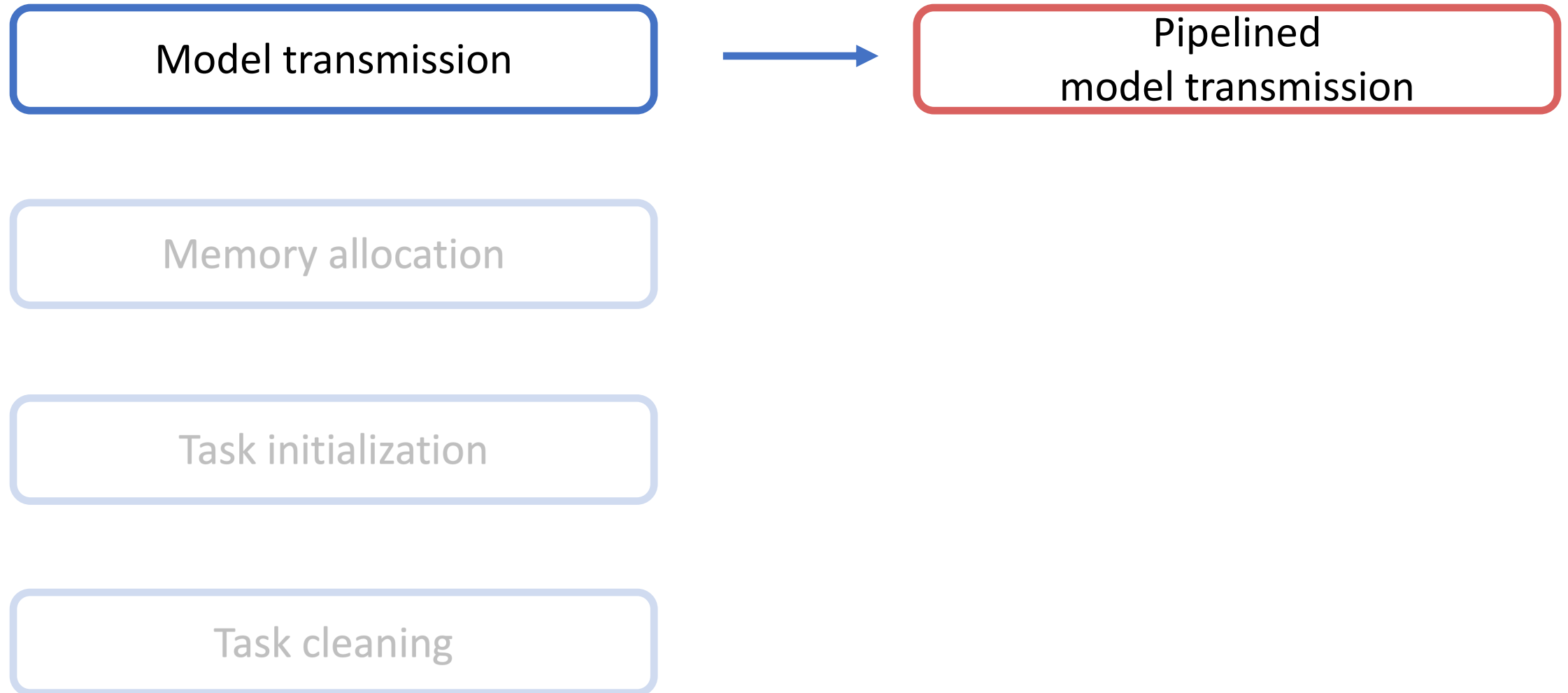
Model transmission

Memory allocation

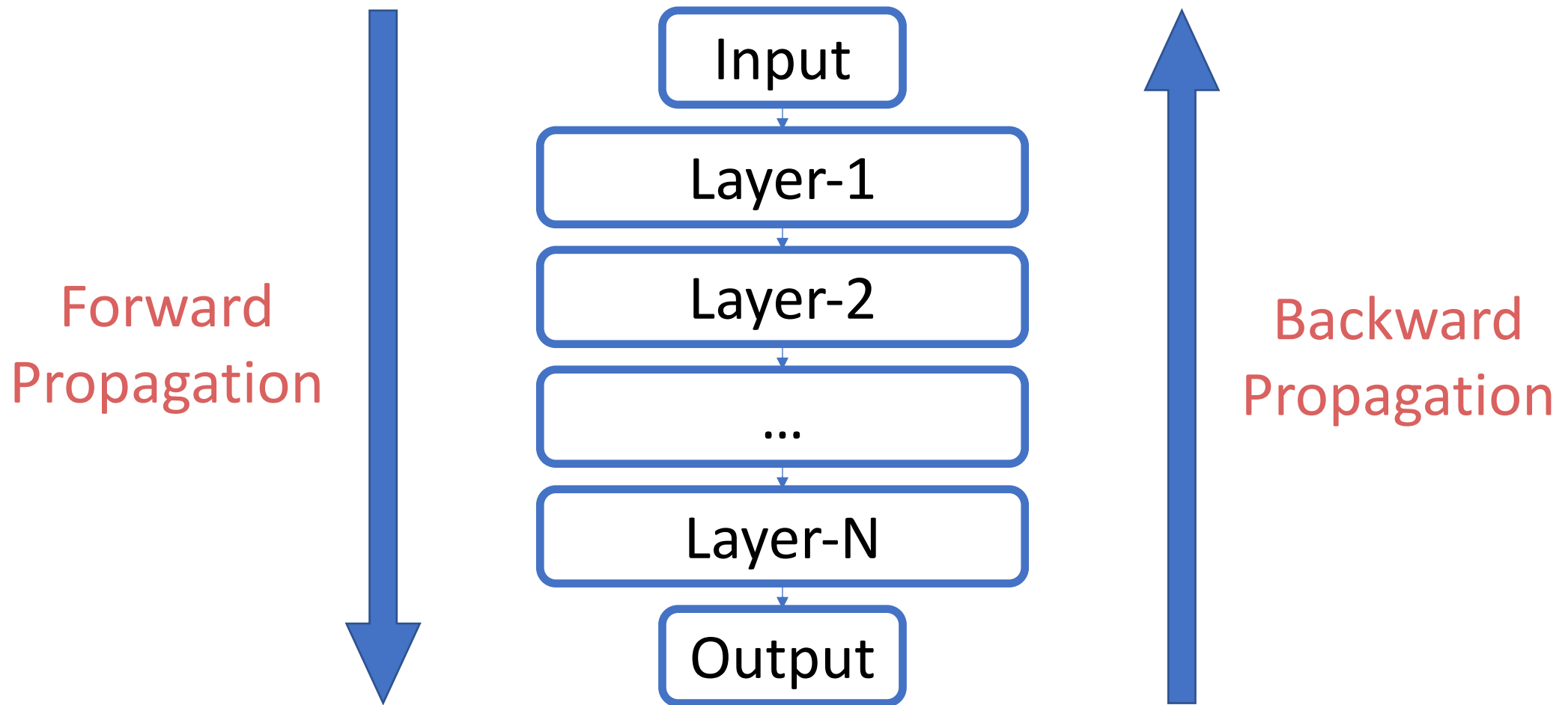
Task initialization

Task cleaning

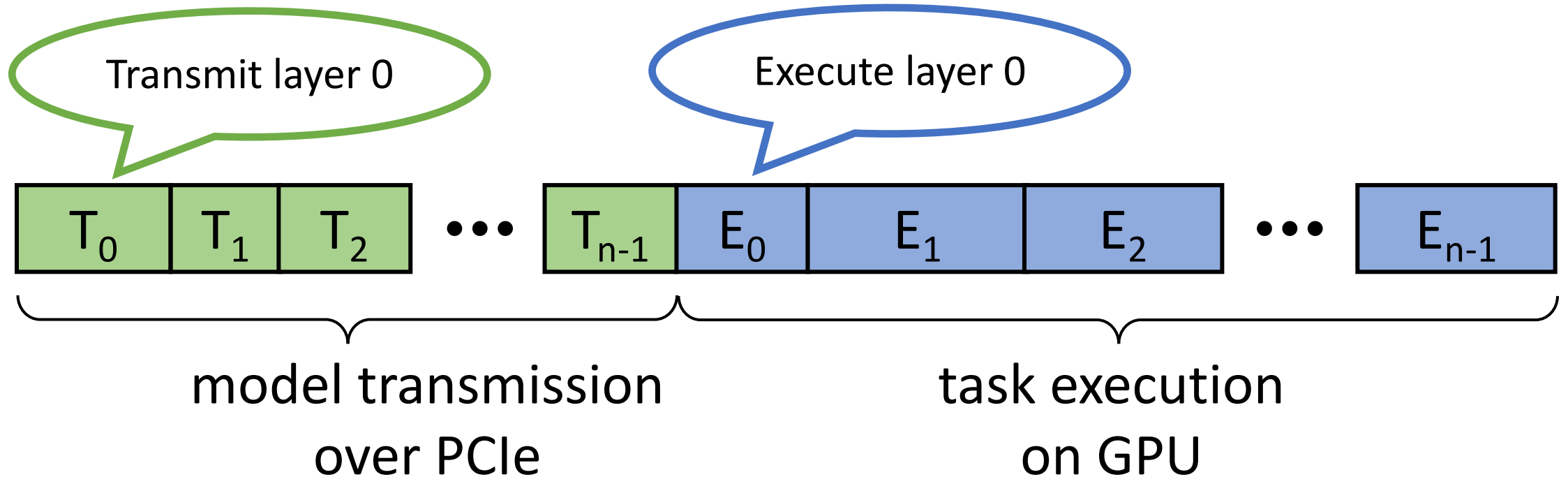
# How to reduce the overhead?



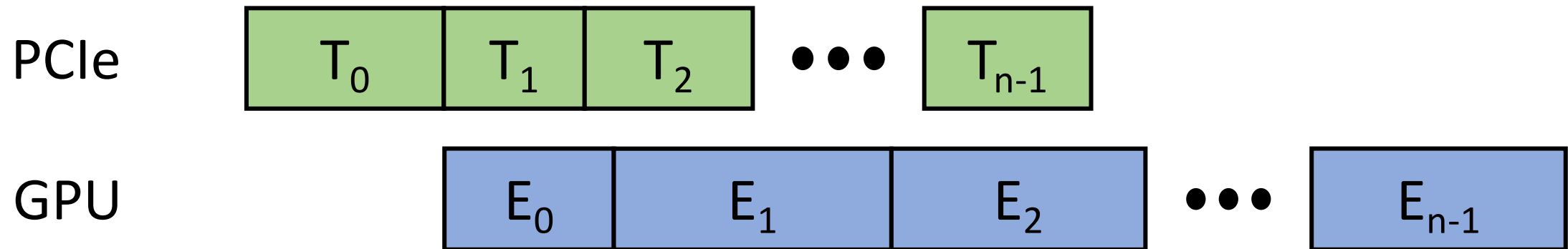
# DL models have layered structures



# Sequential model transmission and execution

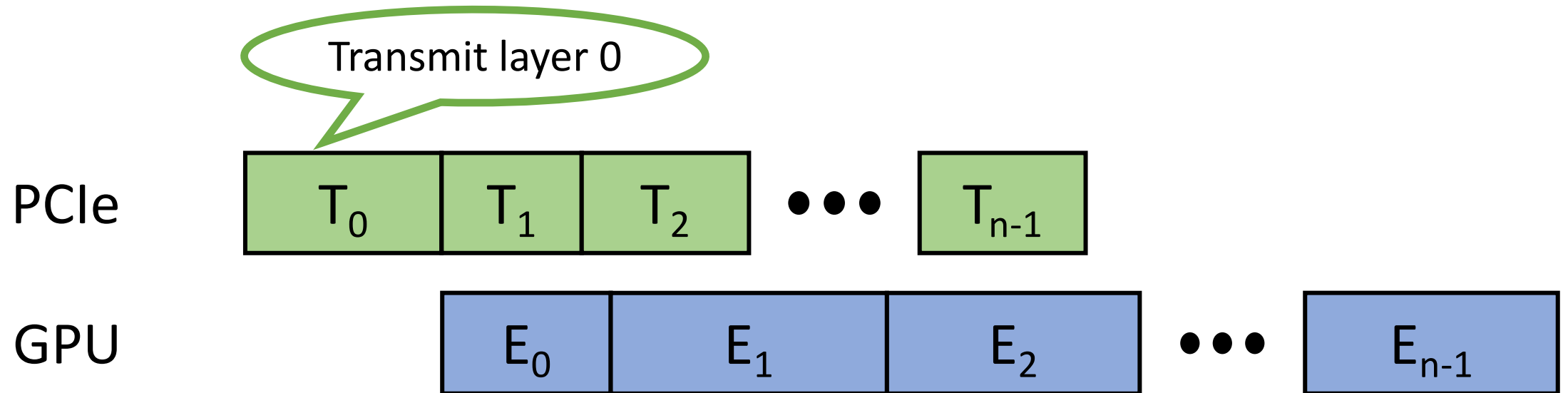


# Pipelined model transmission and execution

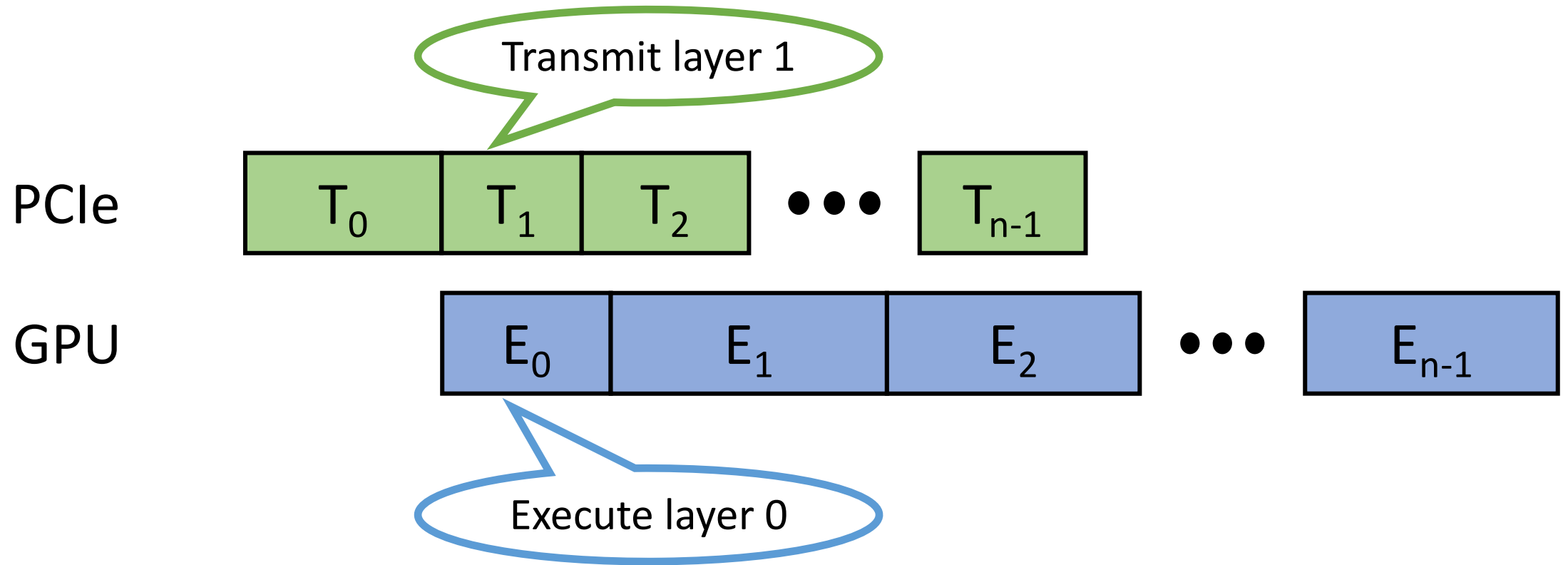




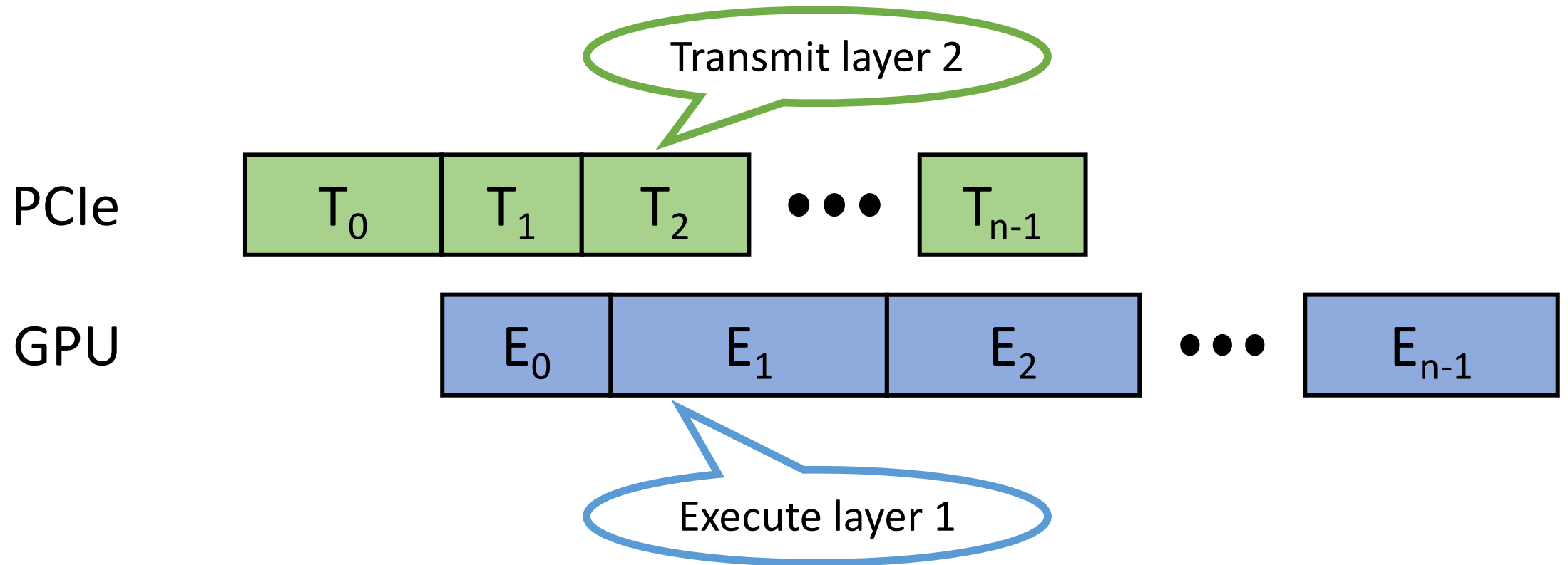
# Pipelined model transmission and execution



# Pipelined model transmission and execution



# Pipelined model transmission and execution



# Pipelined model transmission and execution

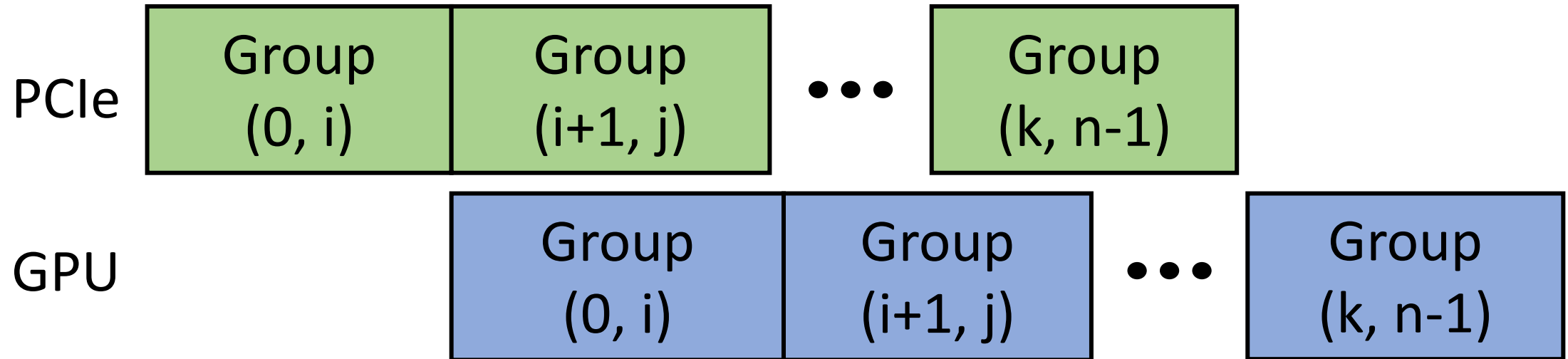
PCIe

GPU

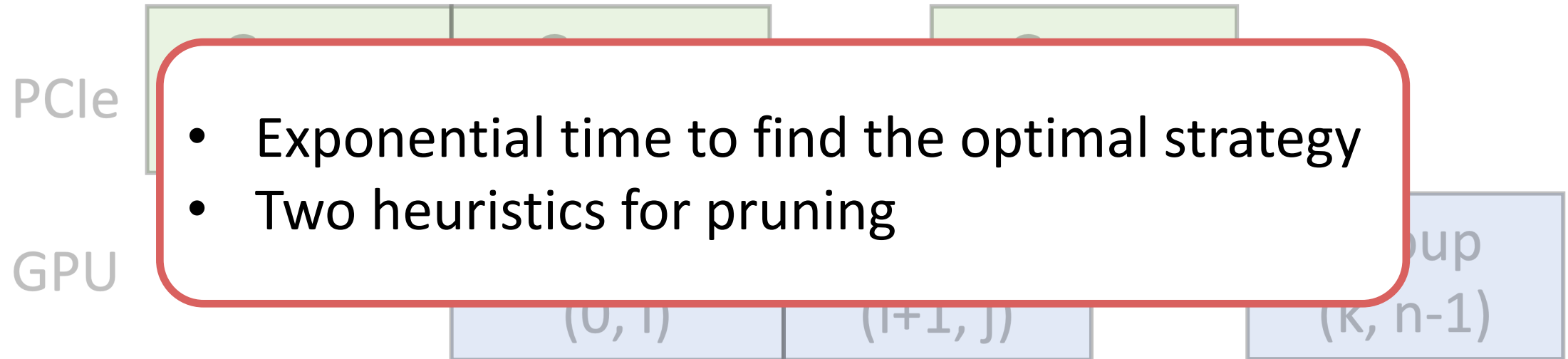
1. Multiple calls to PCIe;
2. Synchronize transmission and execution.

$E_{n-1}$

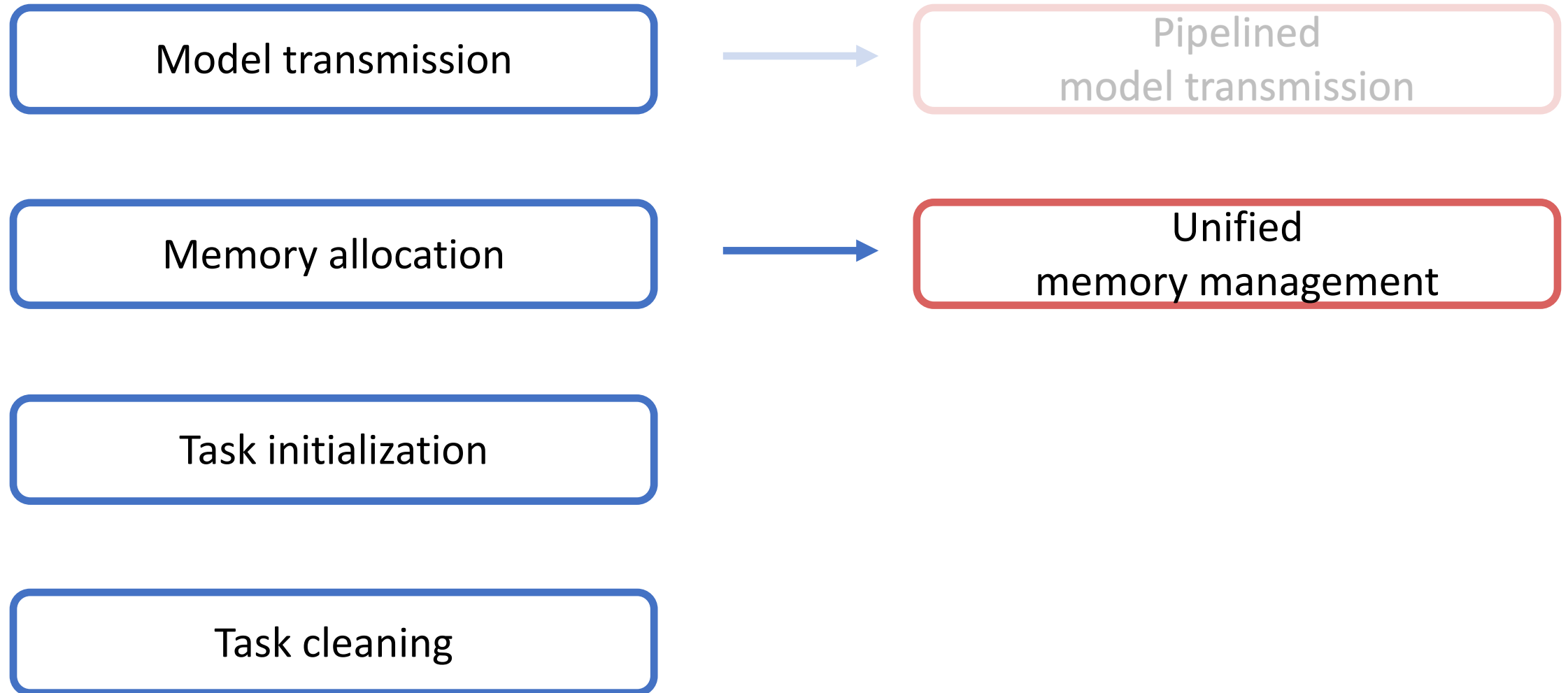
# Pipelined model transmission and execution



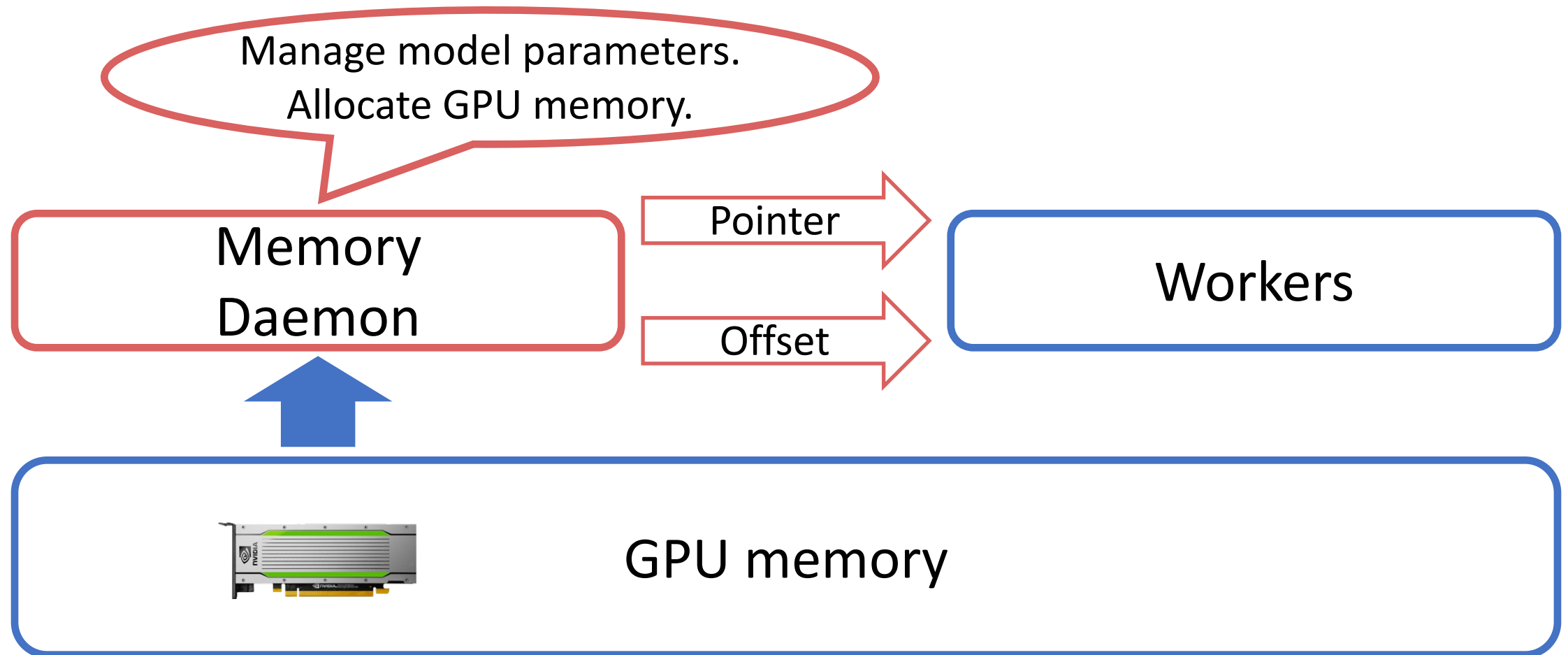
# Pipelined model transmission and execution



# How to reduce the overhead?

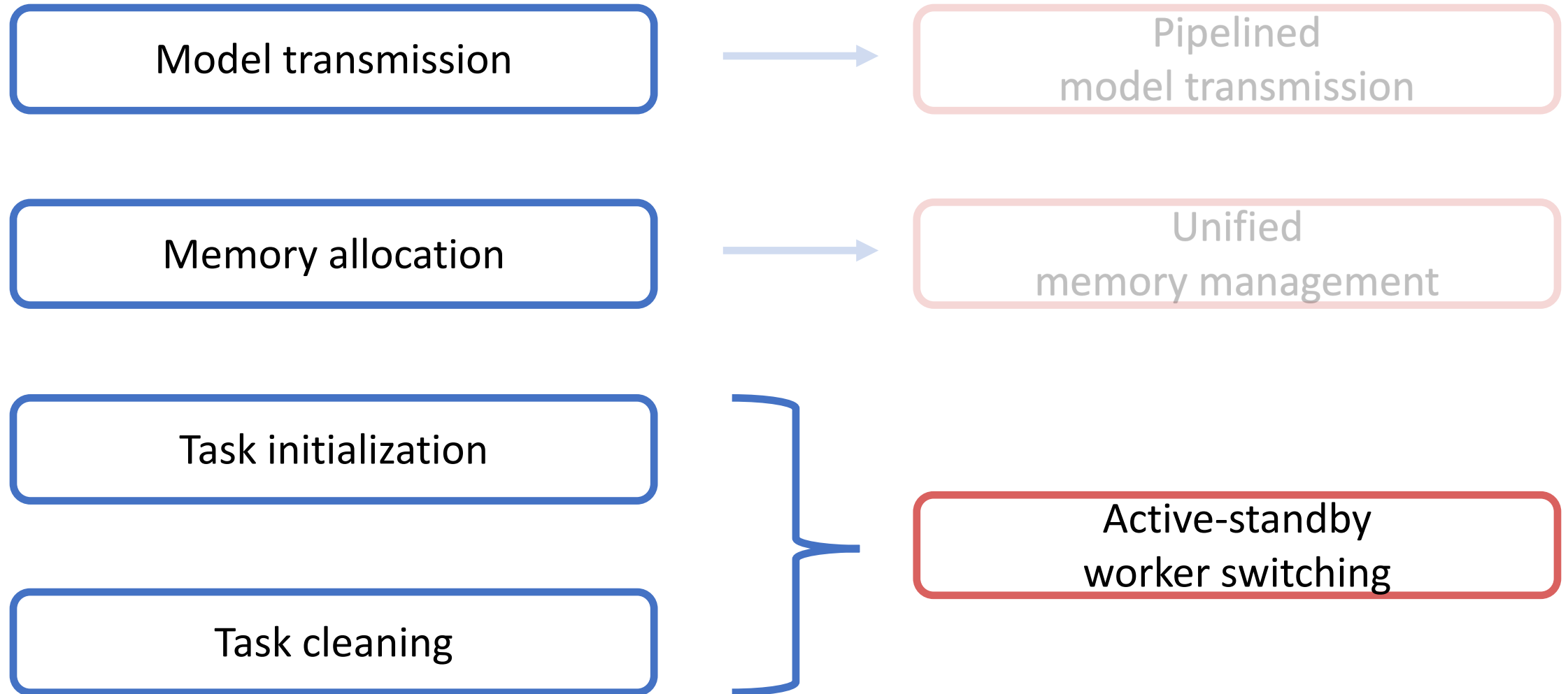


# Unified memory management

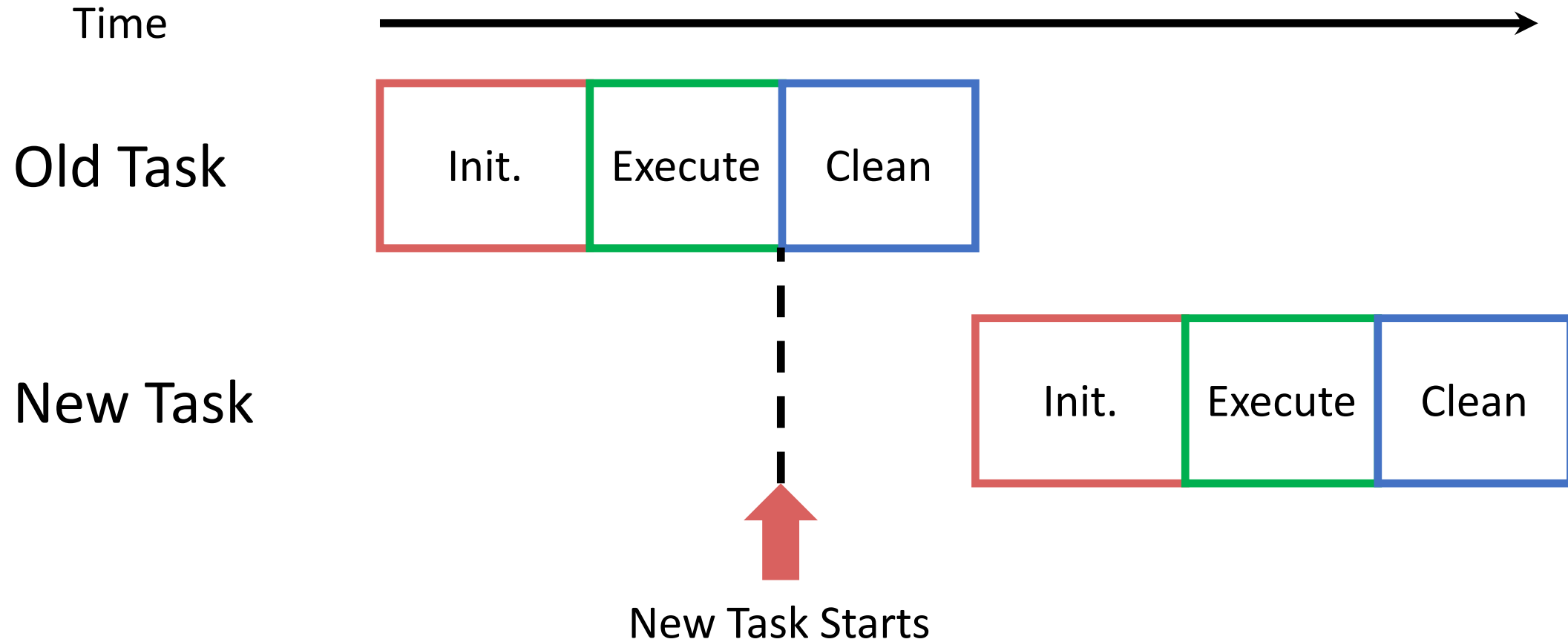




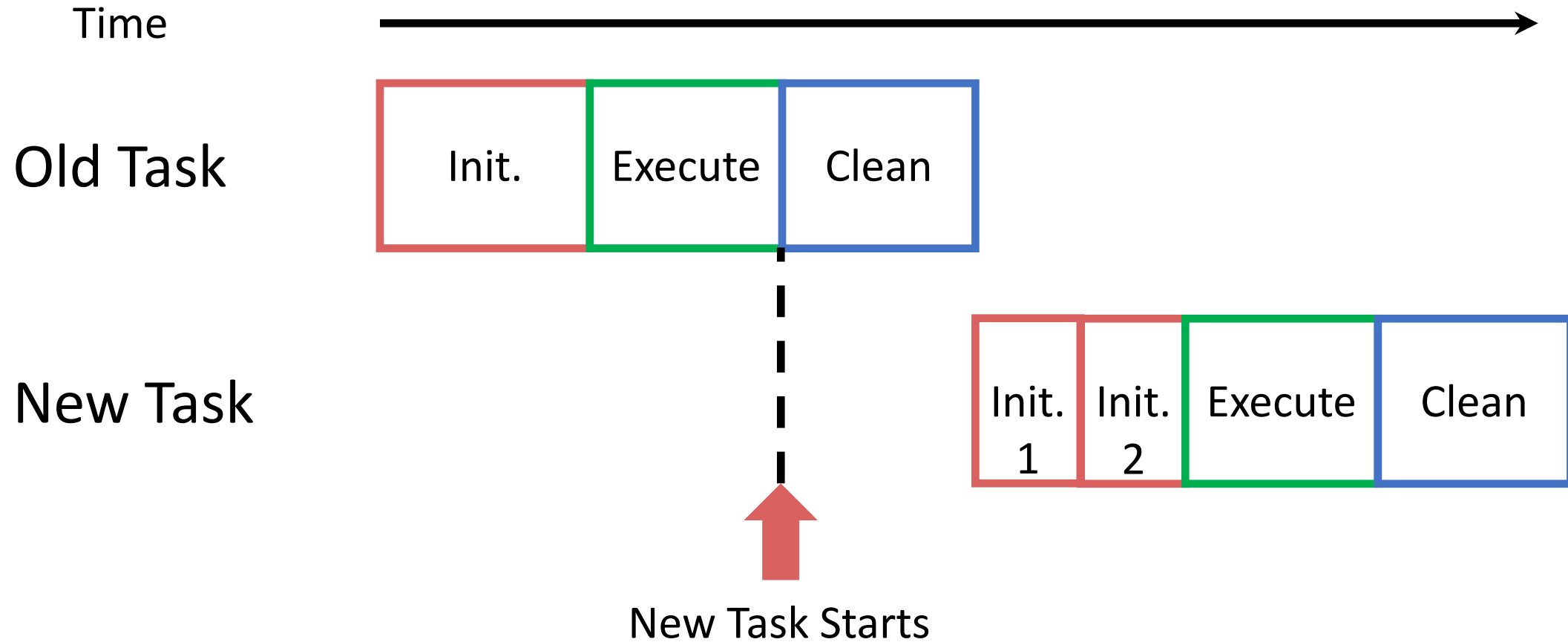
# How to reduce the overhead?



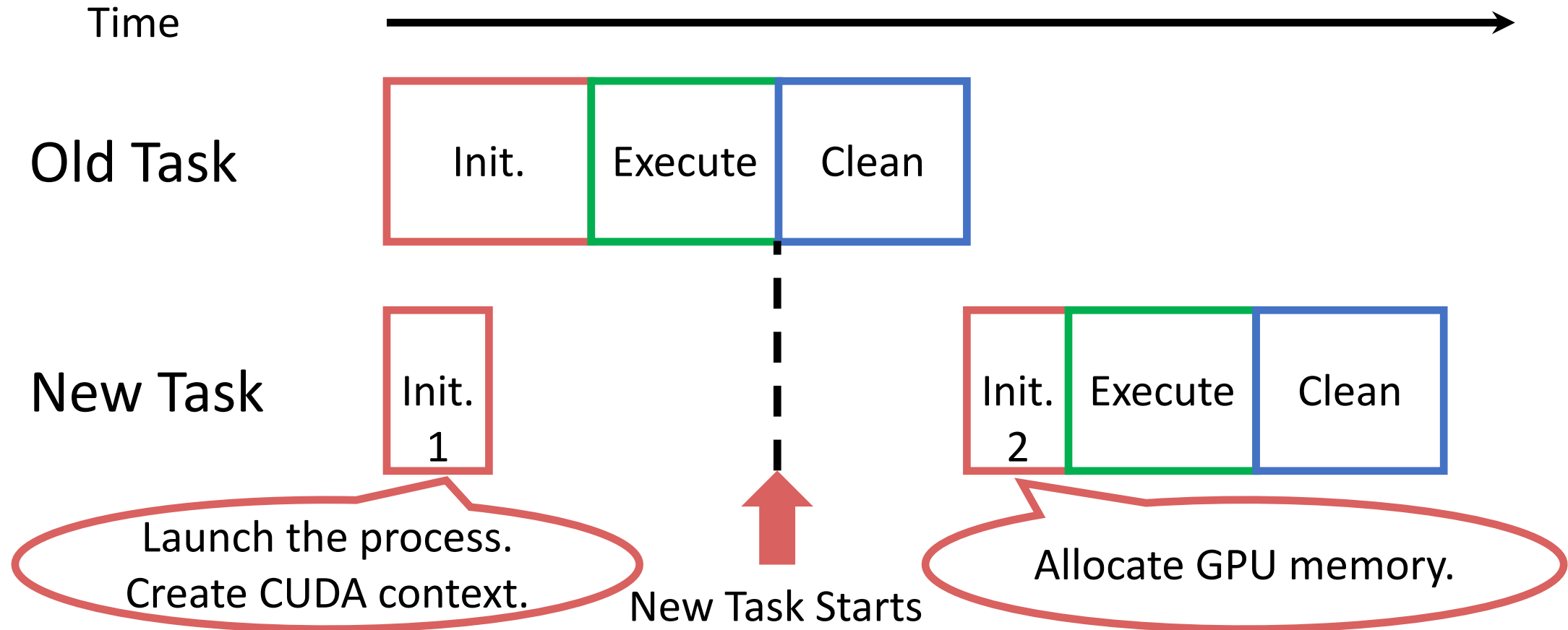
# Active-standby worker switching



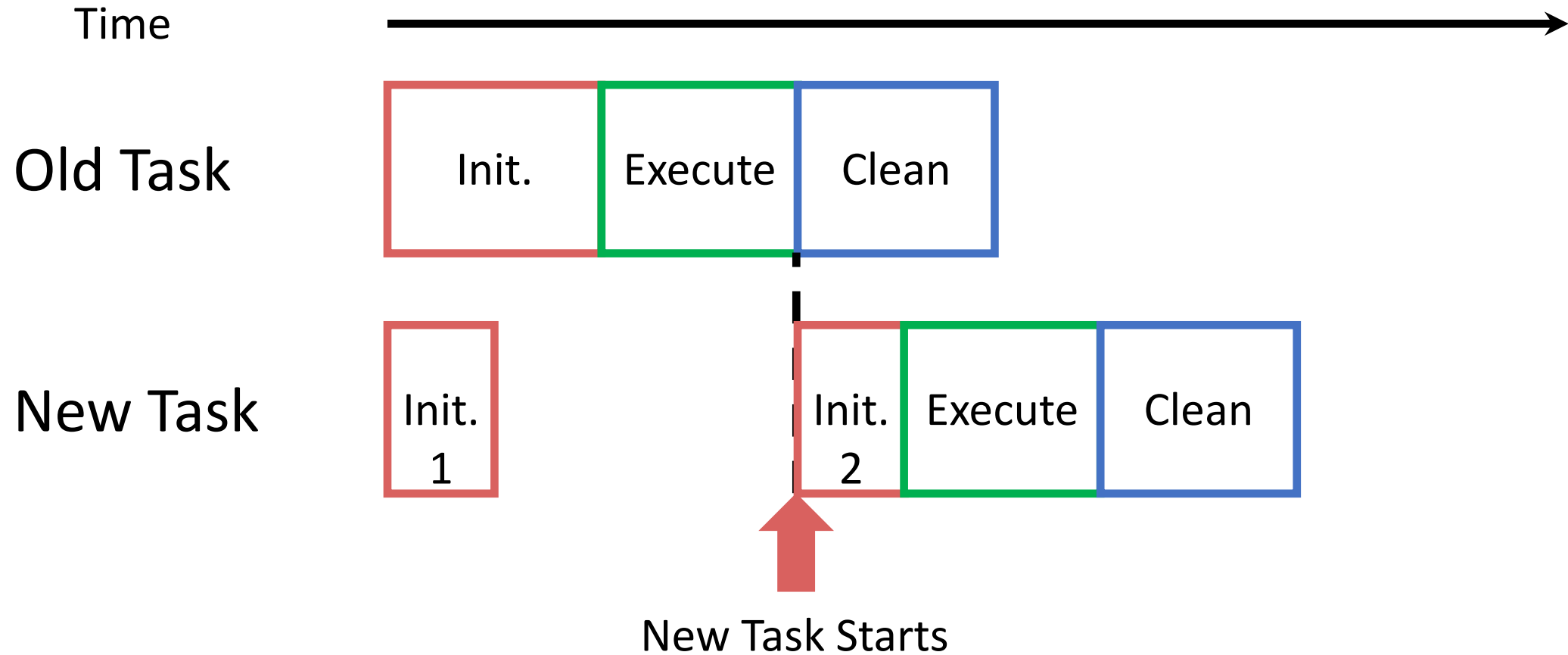
# Active-standby worker switching



# Active-standby worker switching



# Active-standby worker switching



# Implementation

- Testbed: AWS EC2
  - p3.2xlarge: **PCIe 3.0x16**, NVIDIA Tesla **V100** GPU
  - g4dn.2xlarge: **PCIe 3.0x8**, NVIDIA Tesla **T4** GPU
- Software
  - CUDA 10.1
  - PyTorch 1.3.0
- Models
  - ResNet-152
  - Inception-v3
  - BERT-base

# Evaluation

- Can PipeSwitch satisfy SLOs?
- Can PipeSwitch provide high utilization?
- How well do the design choices of PipeSwitch work?

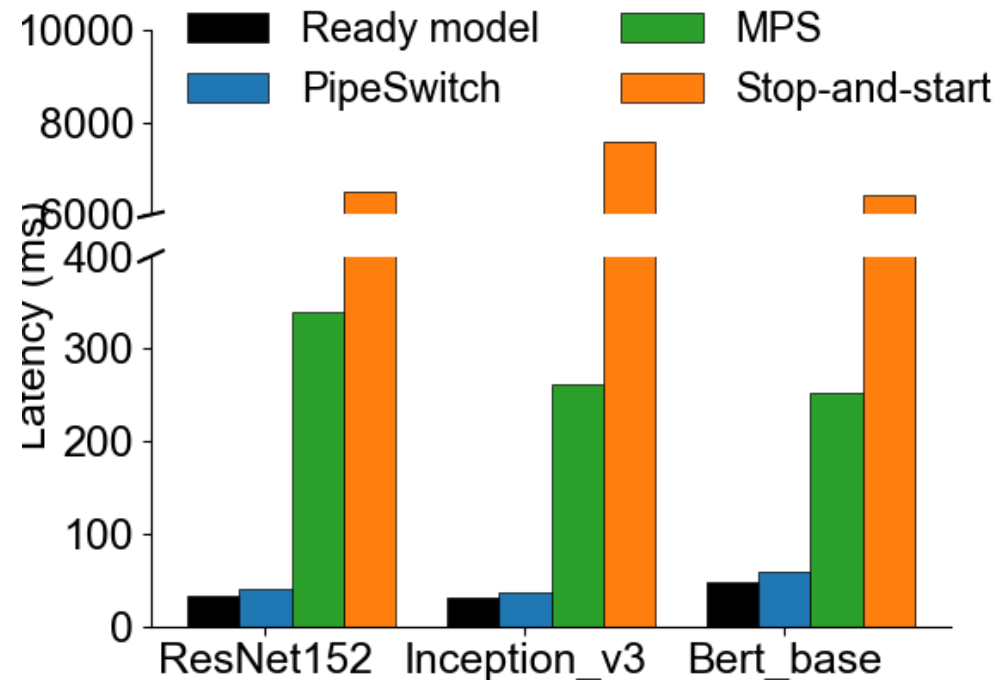
# Evaluation

- Can PipeSwitch satisfy SLOs?
- Can PipeSwitch provide high utilization?
- How well do the design choices of PipeSwitch work?

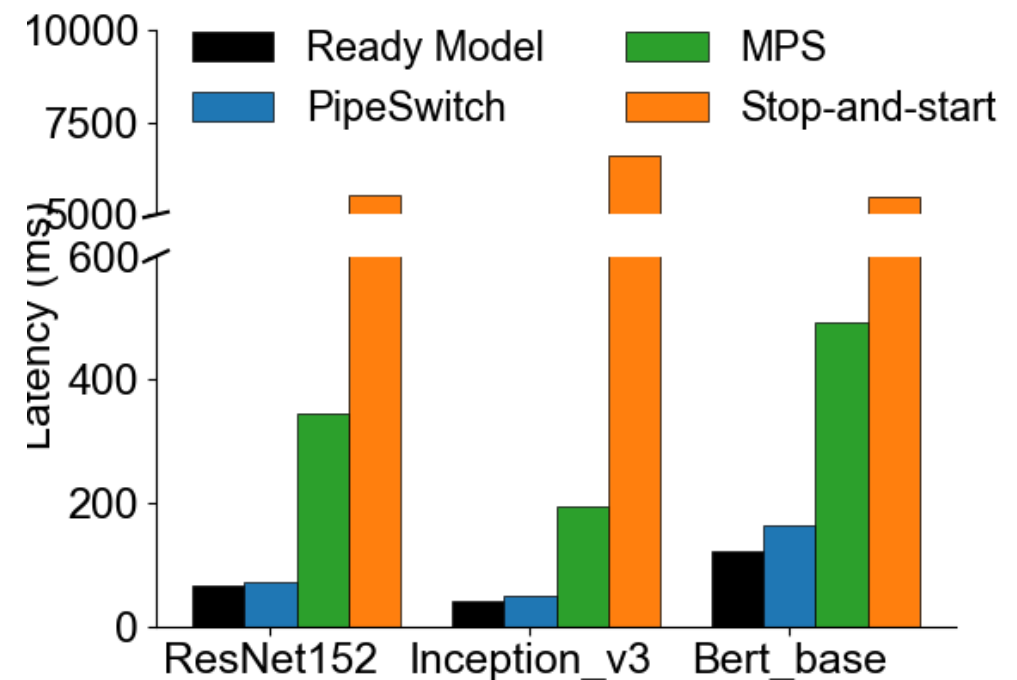


# PipeSwitch satisfies SLOs

## NVIDIA Tesla V100

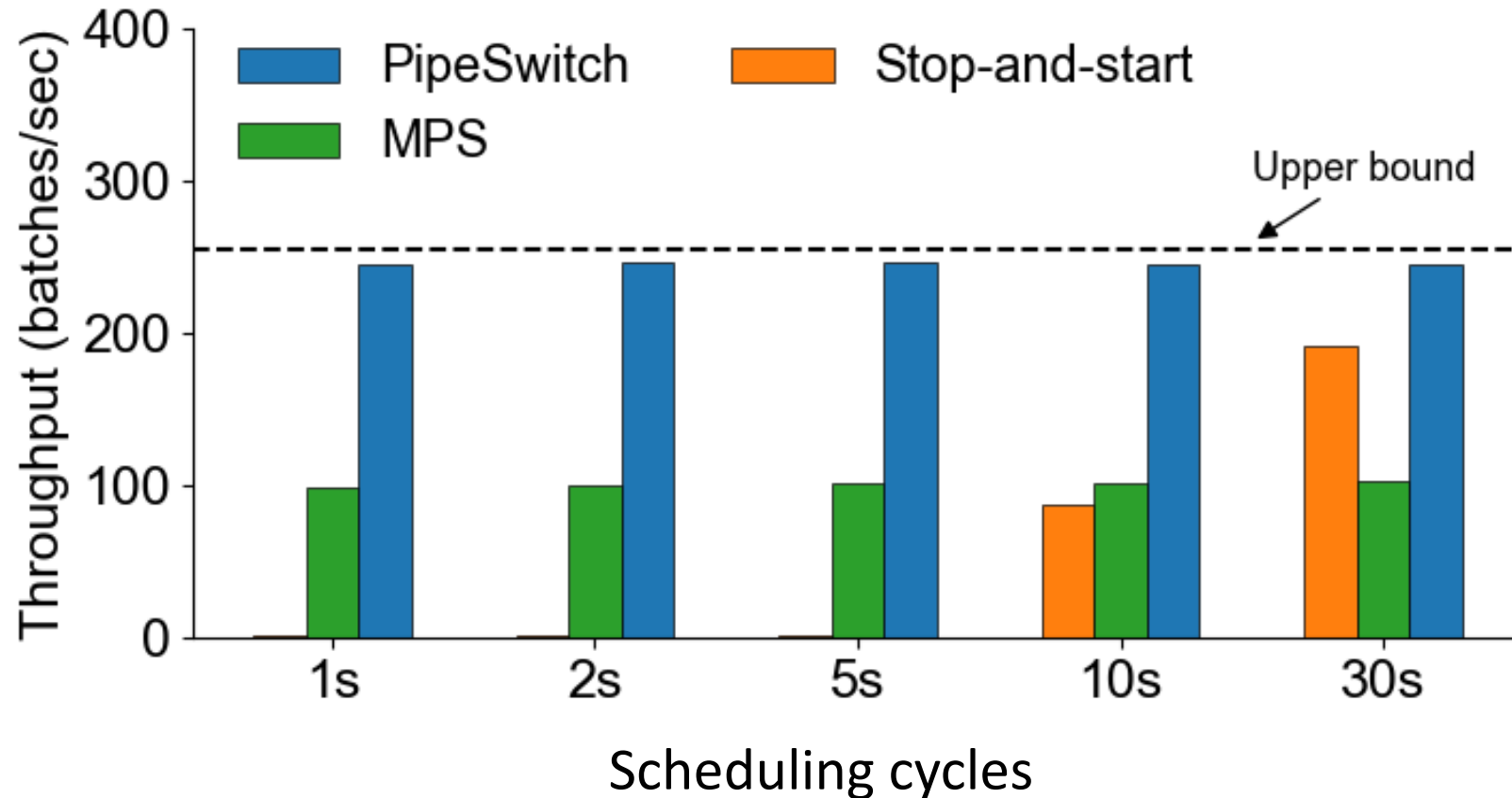


## NVIDIA Tesla T4



**PipeSwitch achieves low context switching latency.**

# PipeSwitch provide high utilization



**PipeSwitch achieves near 100% utilization.**

# Summary

- GPU clusters for DL applications suffer from low utilization
  - Limited share between training and inference workloads
- PipeSwitch introduces pipelined context switching
  - Enable GPU-efficient multiplexing of DL apps with fine-grained time-sharing
  - Achieve millisecond-scale context switching latencies and high throughput