

Operating Systems (Honor Track)

Memory 5: Memory Management in Modern Computer Systems

Xin Jin

Spring 2026

Memory Management in Modern Computer Systems

- Memory Abstraction
 - NSDI'14 FaRM
 - SOSp'23 vLLM
- Demand paging: remote memory over network
 - NSDI'17 InfiniSwap
 - OSDI'20 AIFM
- Demand paging: memory swapping between GPU memory and host memory
 - OSDI'20 PipeSwitch
 - NSDI'23 TGS

FaRM: Fast Remote Memory

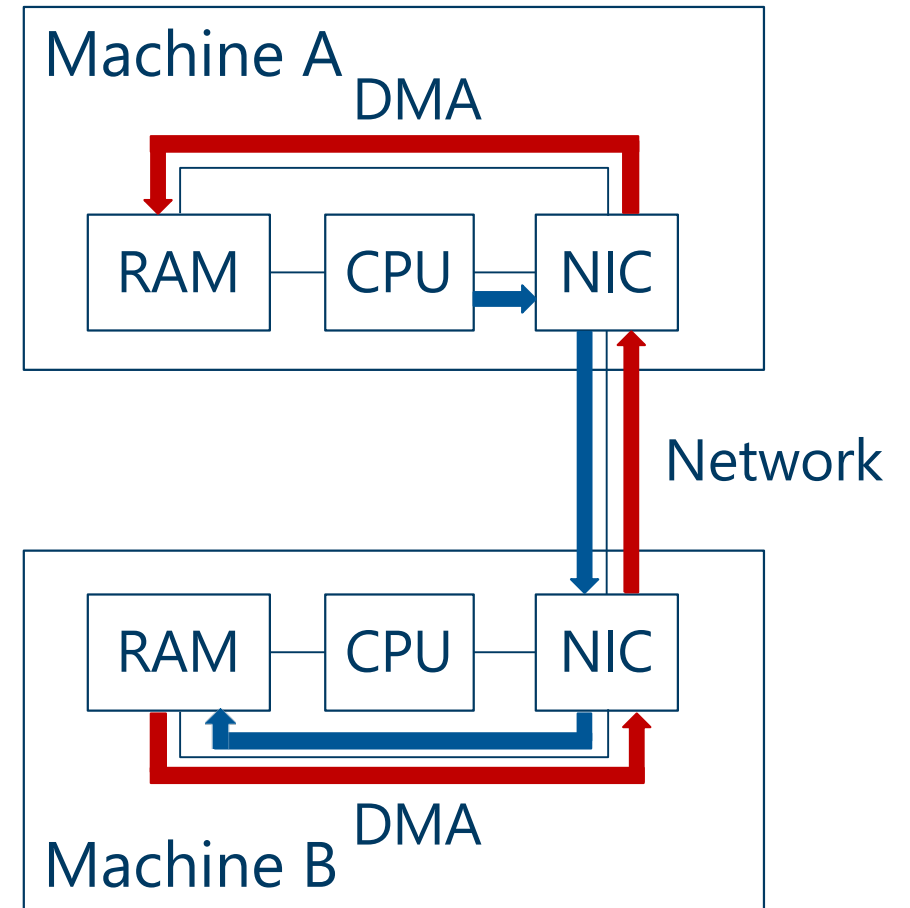
Aleksandar Dragojević, Dushyanth Narayanan,
Orion Hodson, Miguel Castro

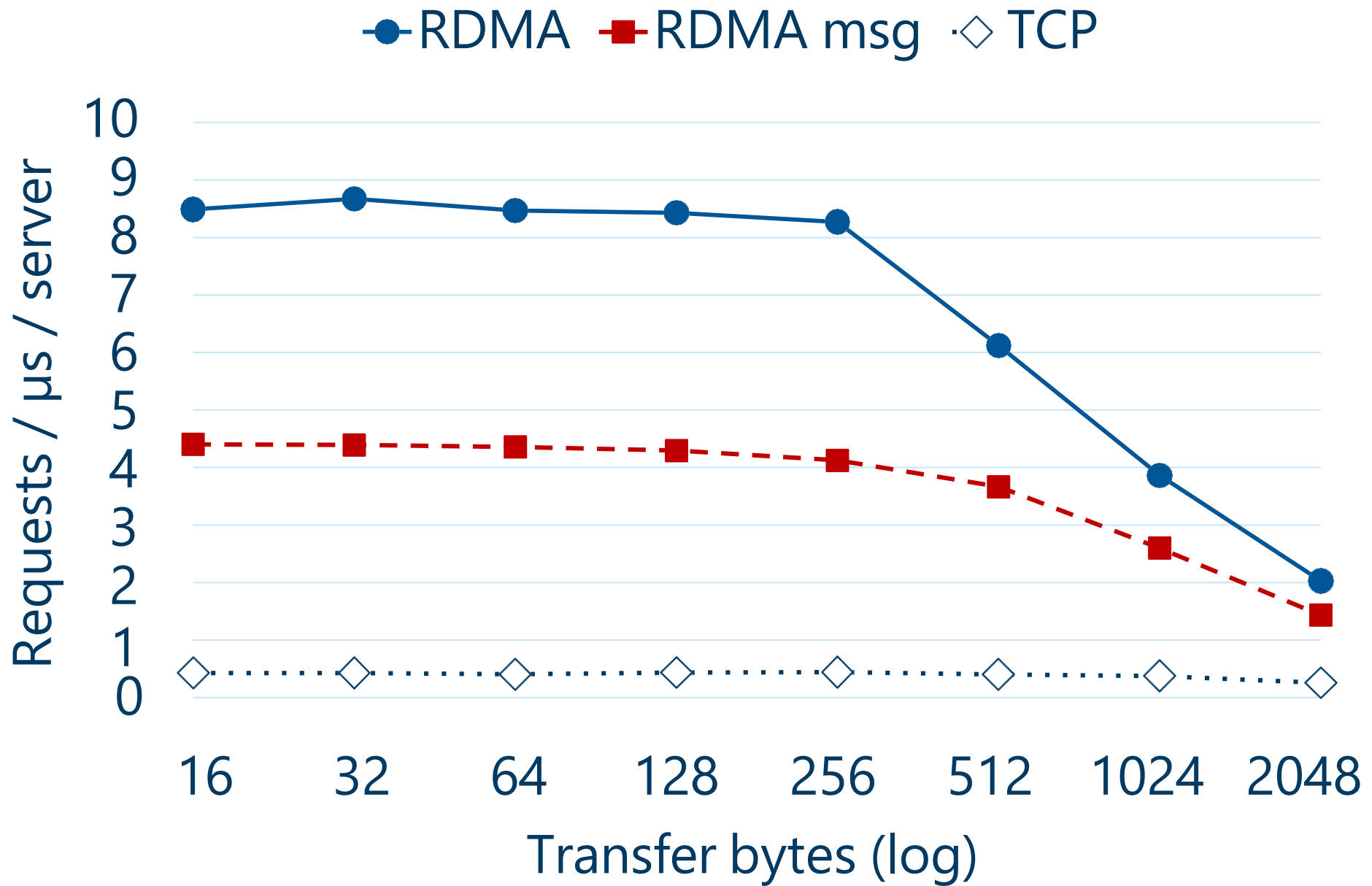
Hardware trends

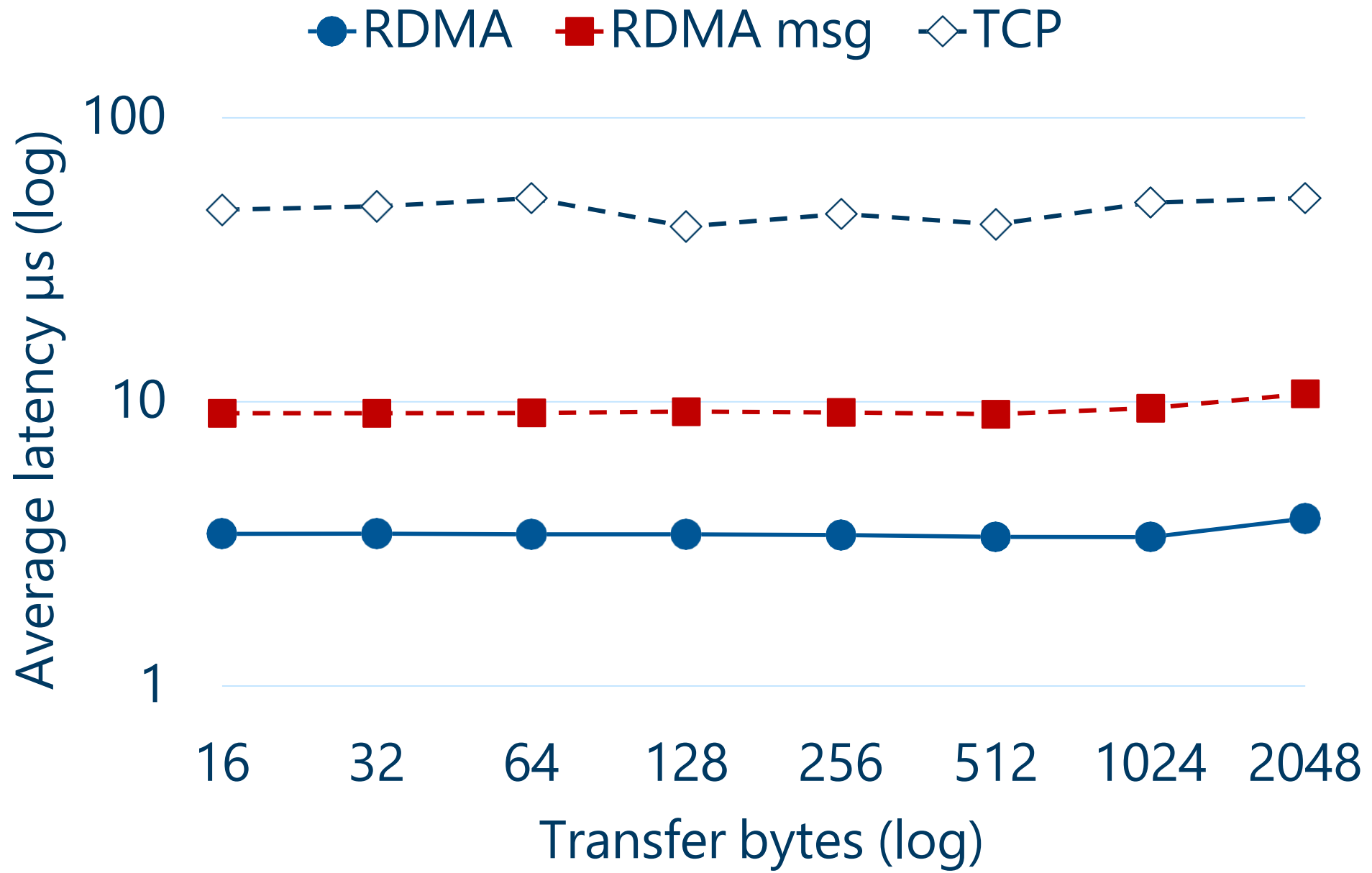
- Main memory is cheap
 - 100 GB – 1 TB per server
 - 10 – 100 TBs in a small cluster
- New data centre networks
 - 40 Gbps throughput (100 this year)
 - 1-3 μ s latency
 - RDMA primitives

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

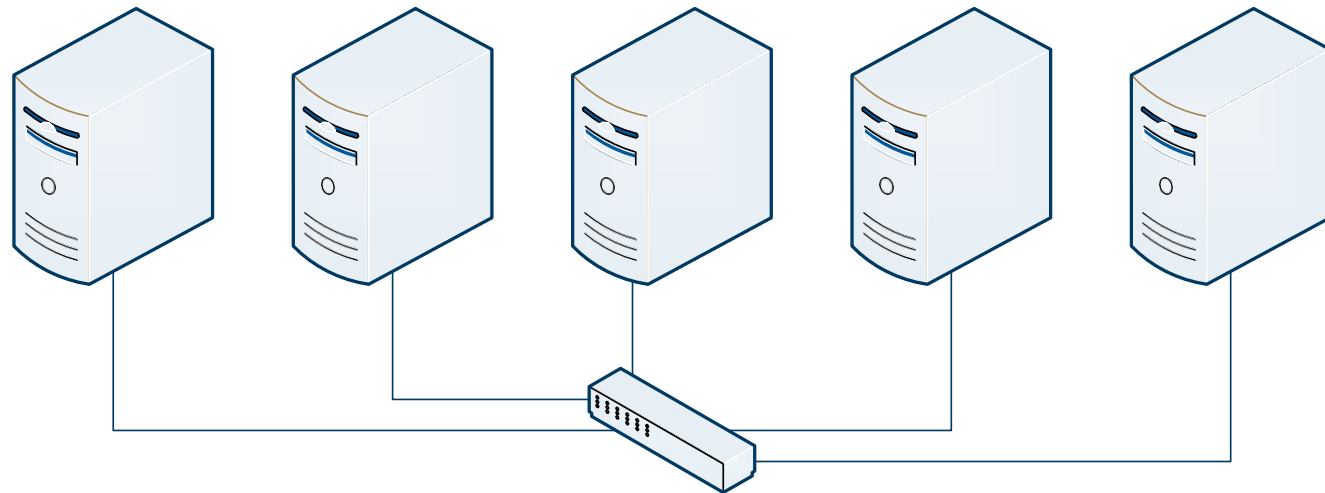
How to program a modern cluster?

We have:

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

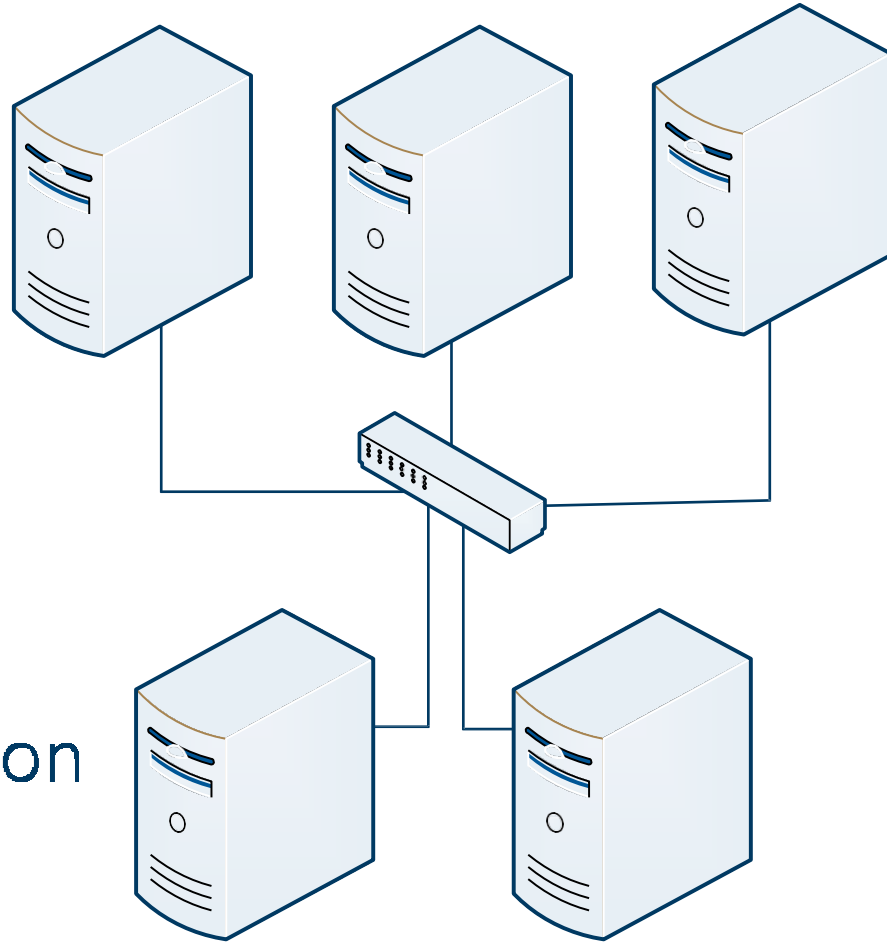
Desirable:

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



Traditional model

Servers: store data

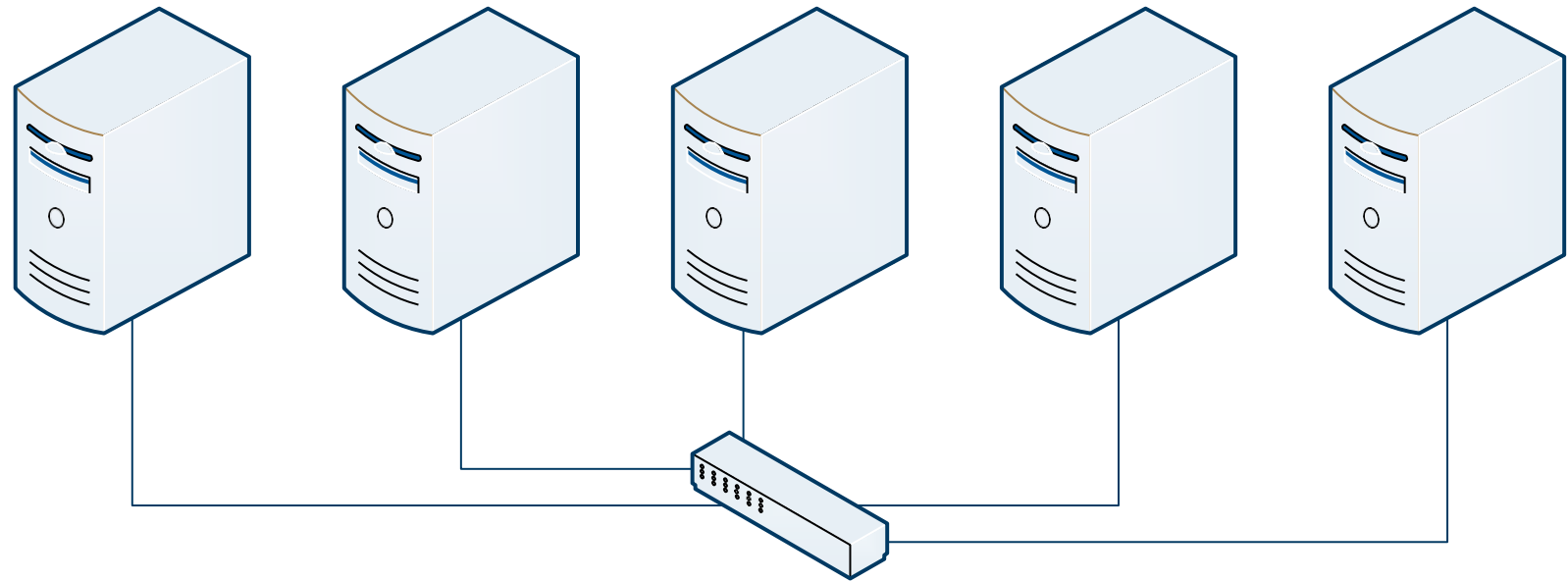


Clients: execute application

Symmetric model

Access to local memory is much faster

Server CPUs are mostly idle with RDMA

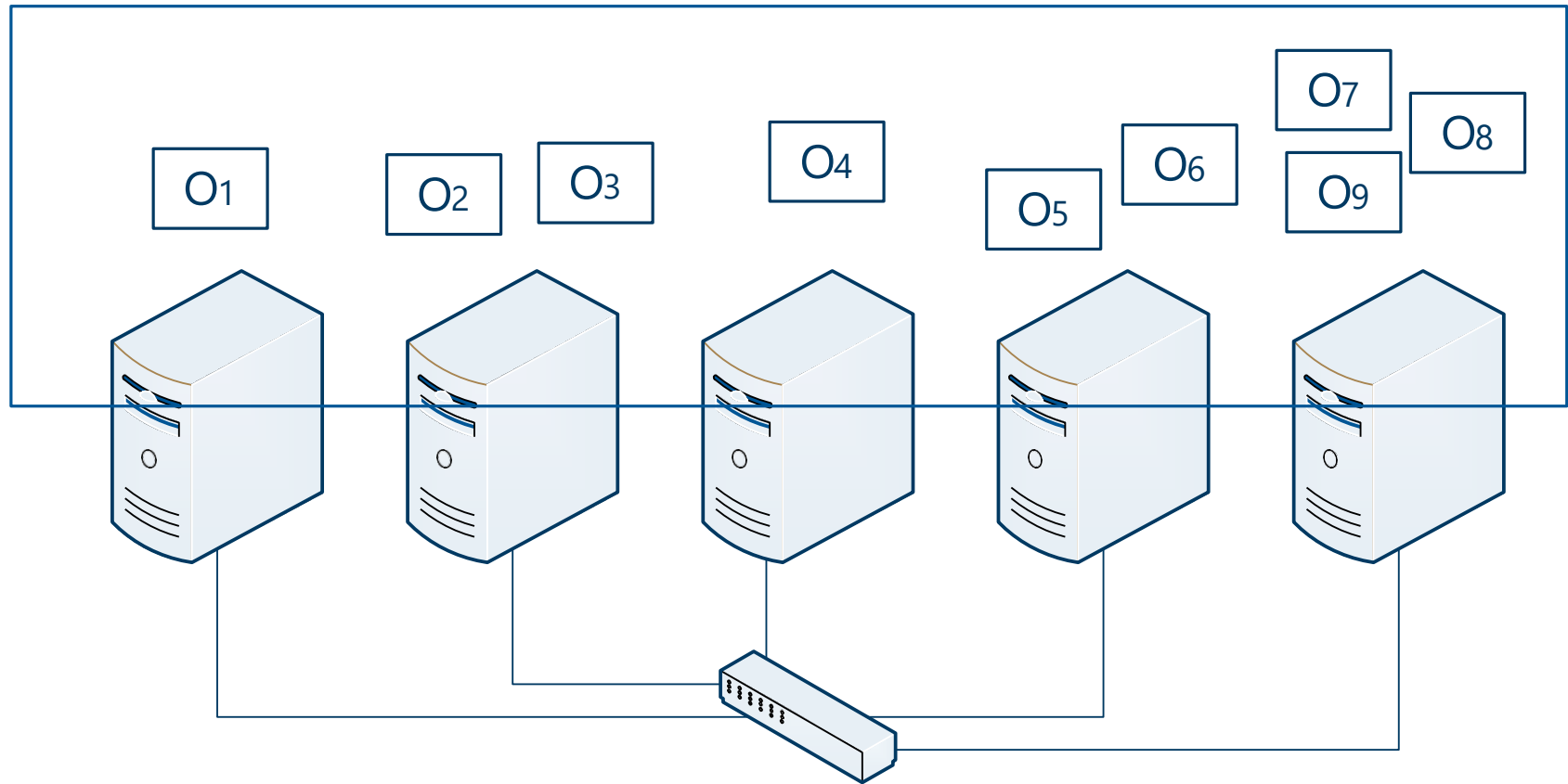


Machines store data and execute application

Shared address space

Supports direct RDMA of objects

Programmability a welcome bonus



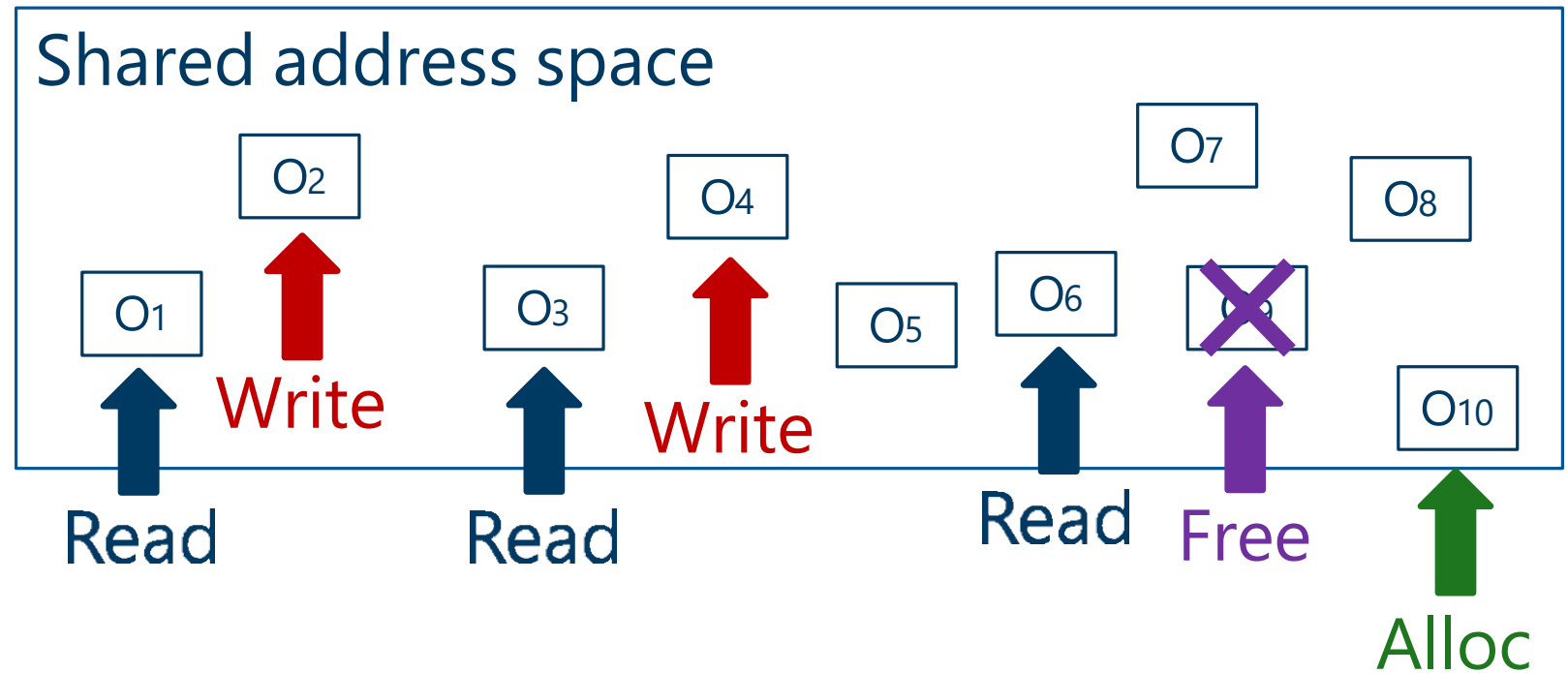
Shared address space

General primitive

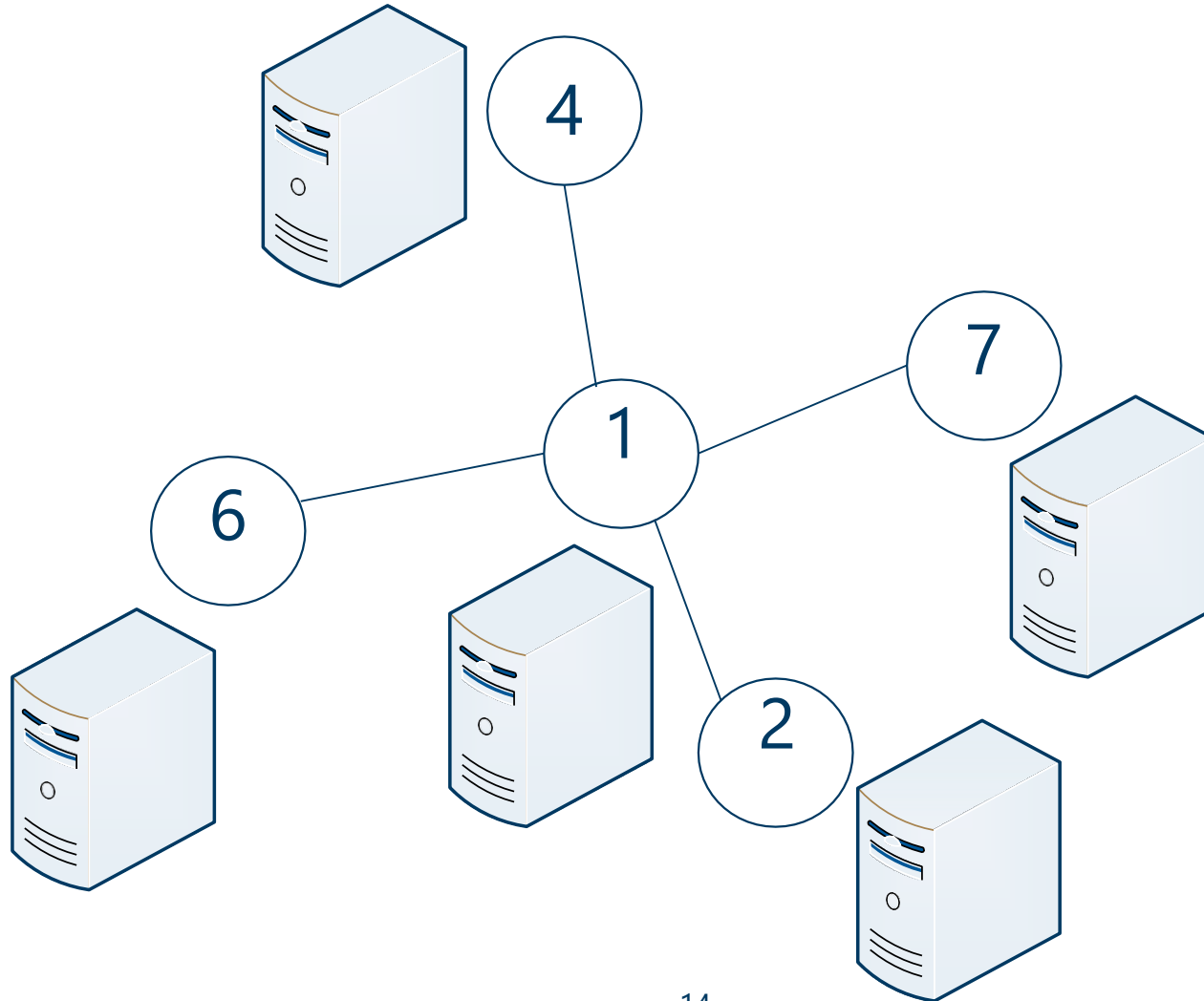
Strong consistency:
serializability

Transparent:

- location
- concurrency
- failures



Optimizations: locality awareness

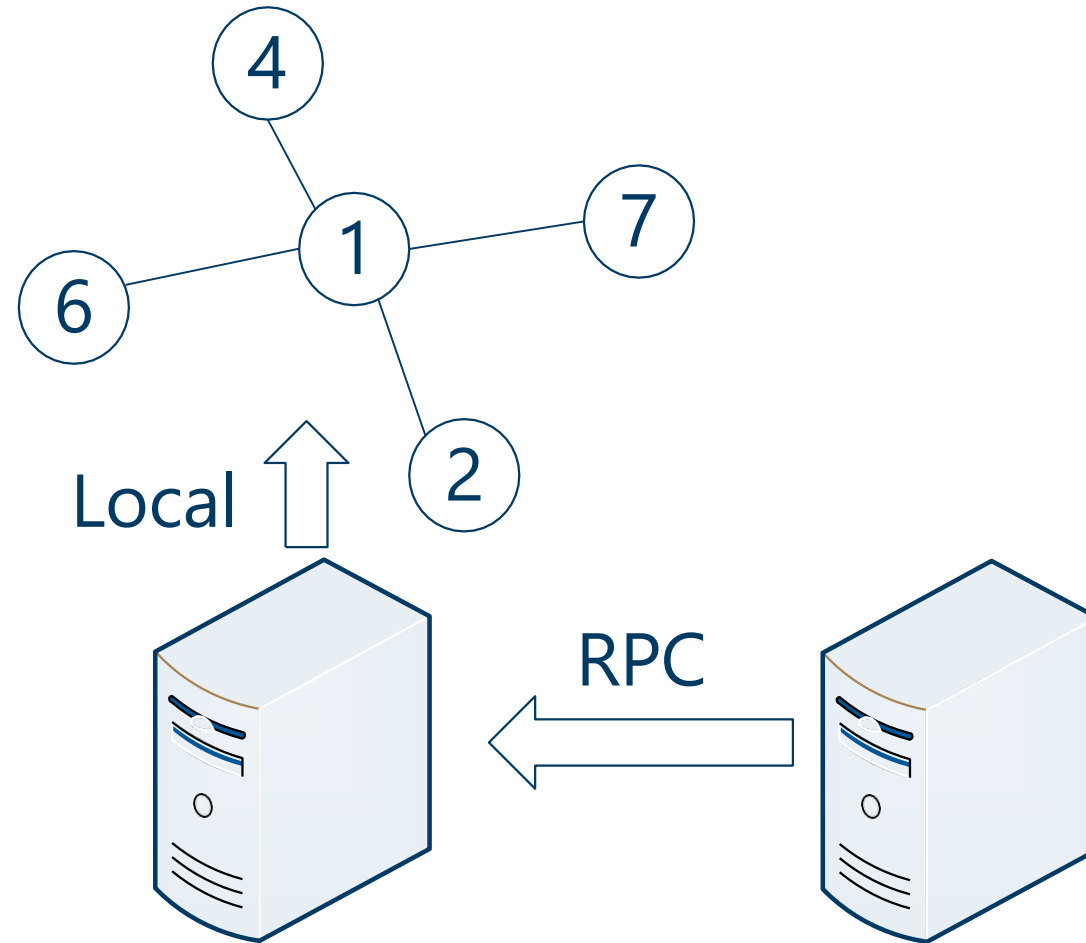


Optimizations: locality awareness

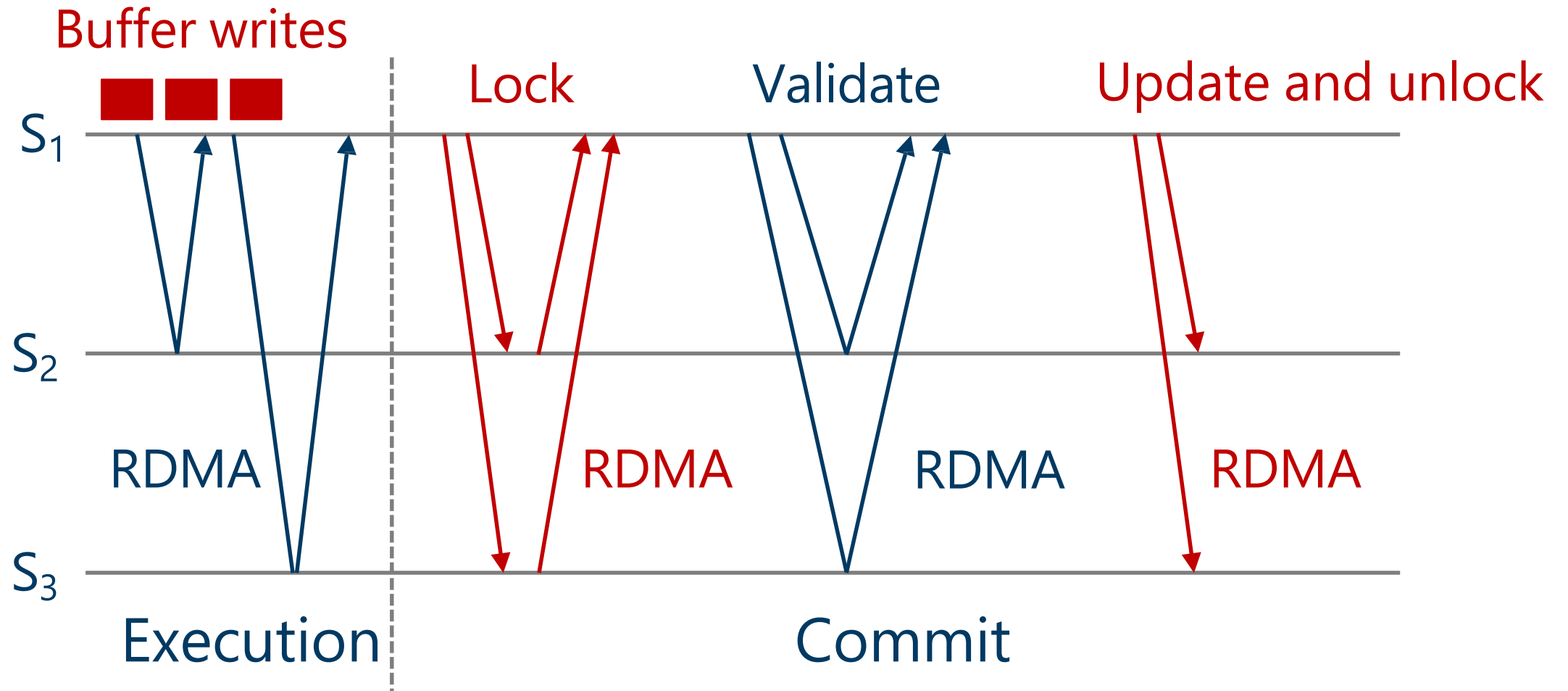
Collocate data
accessed together

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 μ 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
 - SOSPP'23 vLLM
- Demand paging: remote memory over network
 - NSDI'17 InfiniSwap
 - OSDI'20 AIFM
- Demand paging: memory swapping between GPU memory and host memory
 - OSDI'20 PipeSwitch
 - NSDI'23 TGS



Efficient Memory Management for Large Language Model Serving with PagedAttention

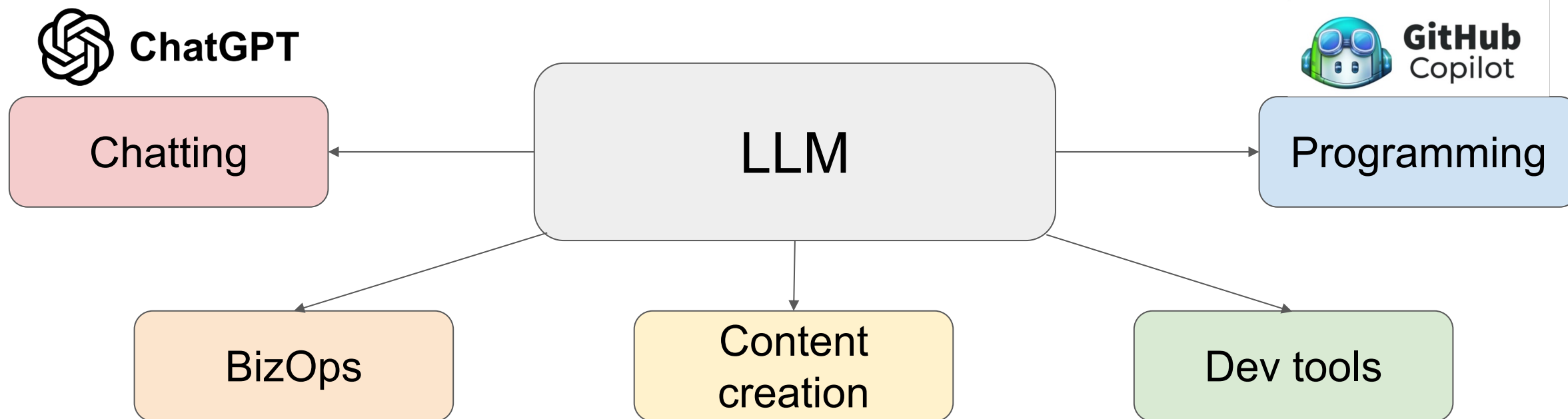
Woosuk Kwon^{1,*} **Zhuohan Li**^{1,*} Siyuan Zhuang¹ Ying Sheng^{1,2} Lianmin Zheng¹
Cody Hao Yu³ Joseph E. Gonzalez¹ Hao Zhang⁴ Ion Stoica¹

¹UC Berkeley ²Stanford University ³Independent Researcher ⁴UC San Diego

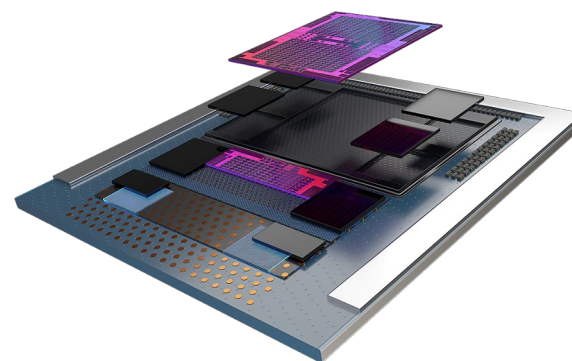
**Equal contribution*

SOSP 2023

The era of Large Language Models (LLMs)



Served by GPUs



Serving LLMs is (surprisingly) slow and expensive

- A single A100 GPU can only serve **< 1** request per second
 - Moderate size of model (13B parameters) and input
- A ton of GPUs are required for production-scale LLM services

 r/LocalLLaMA · 8 days ago
by Financial_Stranger52

Join

...

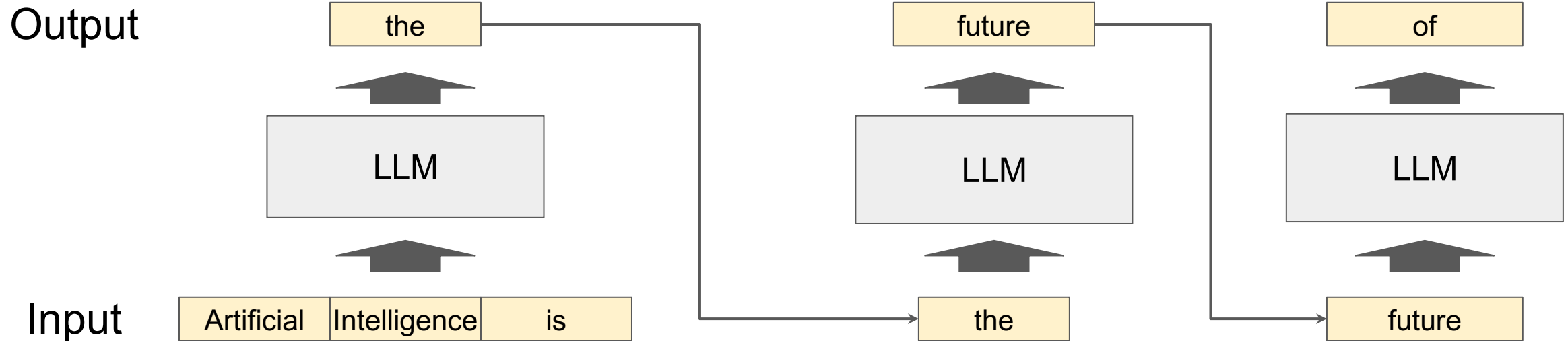
Is local LLM cheaper than ChatGPT API?

ChatGPT api only costs 0.002 dollar for 1k token. I found that LLMs like llama output only 10-20 tokens per second, which is very slow. And such machines costs over 1 dollar per hour. It seems that using api is much cheaper. Based on these observations, it seems that utilizing the ChatGPT API might be a more affordable option.



Microsoft warns of service disruptions if it can't get enough A.I. chips for its data centers

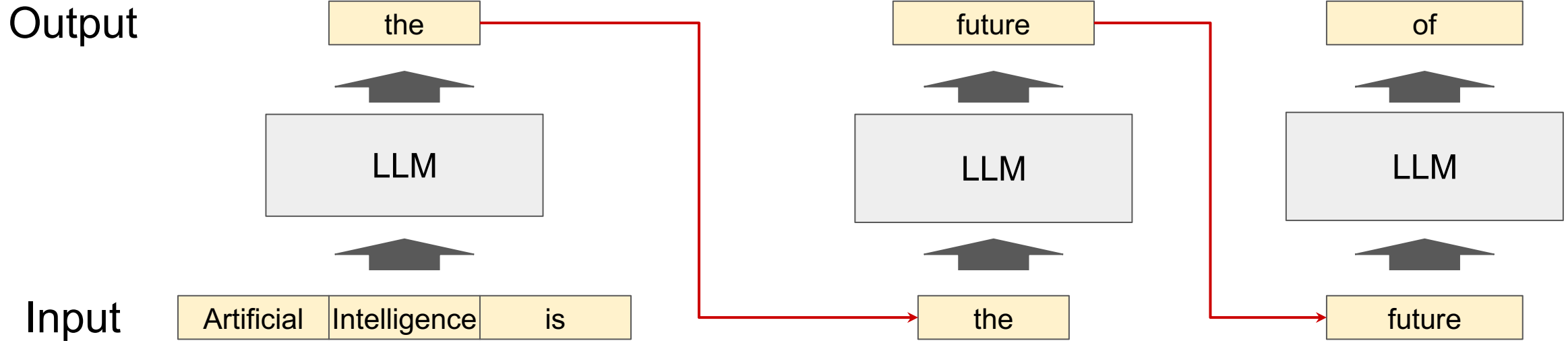
Inference process of LLMs



Repeat until the sequence

- Reaches per-request maximum length (e.g., 2048 tokens)
- Generates certain tokens (e.g., “<|end of sequence|>”)

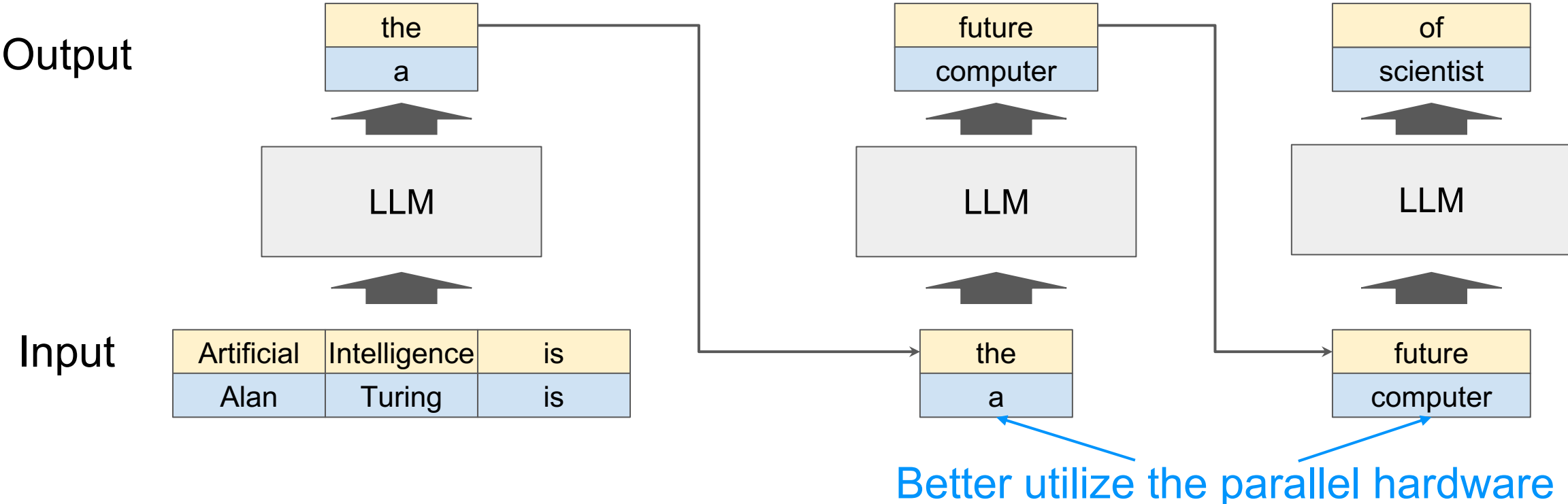
Inference process of LLMs



Why is it **slow** and **inefficient**?

- The sequential dependency between output tokens makes it difficult to fully utilize the parallelism in GPUs

Solution: Batching multiple requests together



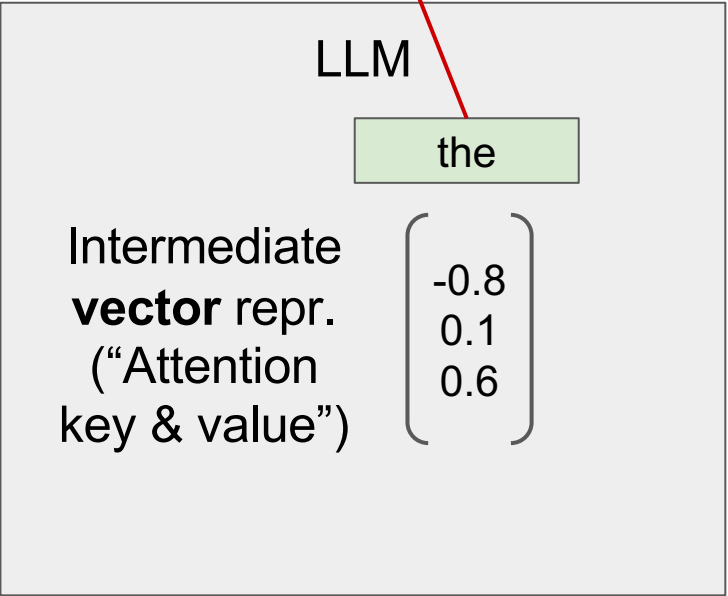
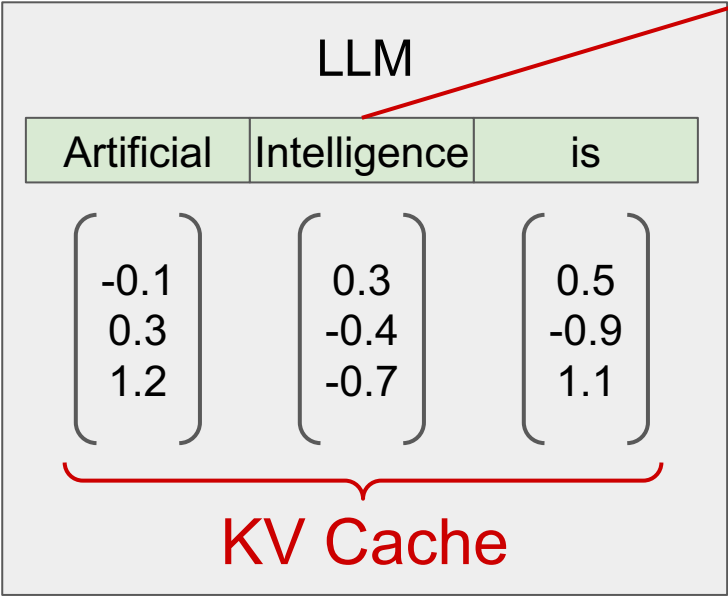
However, we observed that the batch size is significantly limited by the inefficient memory management for “***KV Cache.***”

(Attention) KV Cache

Output

the

future



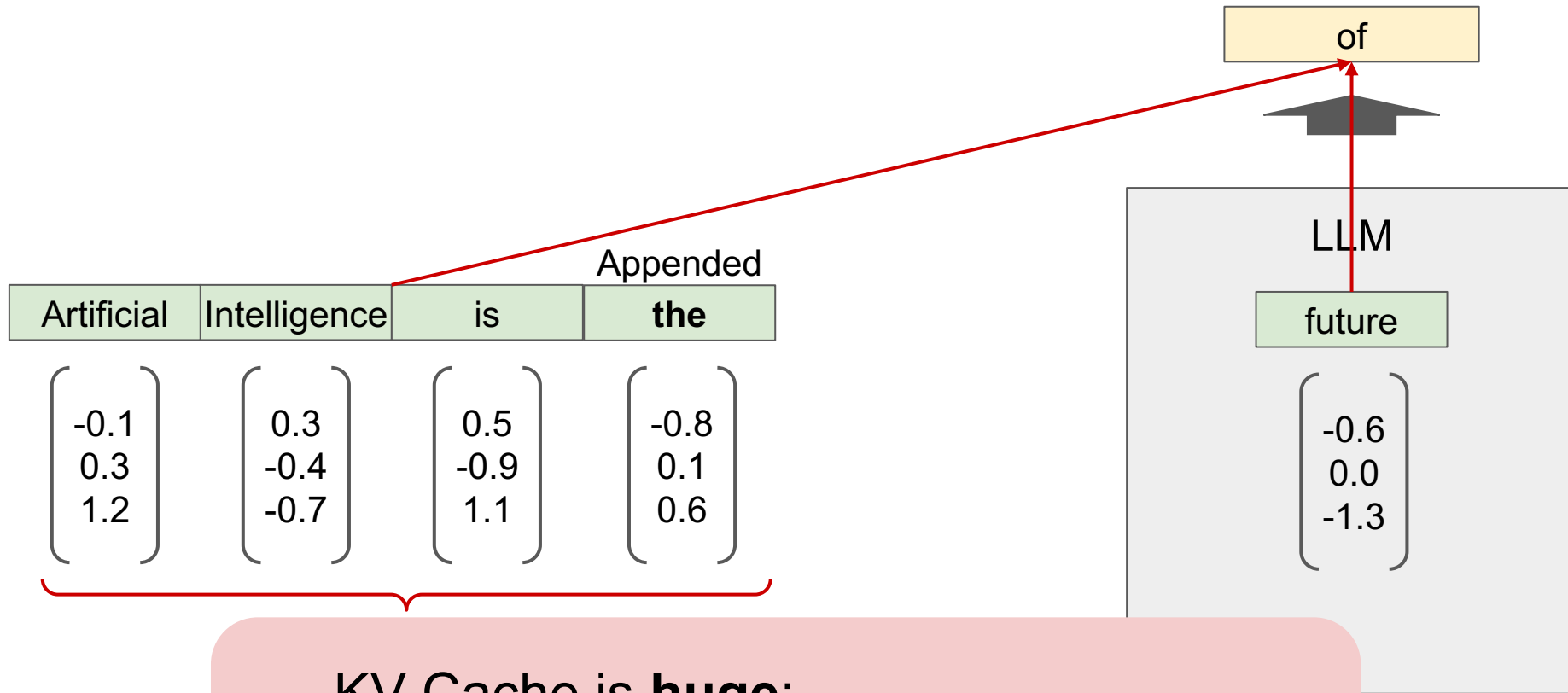
Input

Artificial Intelligence is

the

KV Cache dynamically grows and shrinks

Output



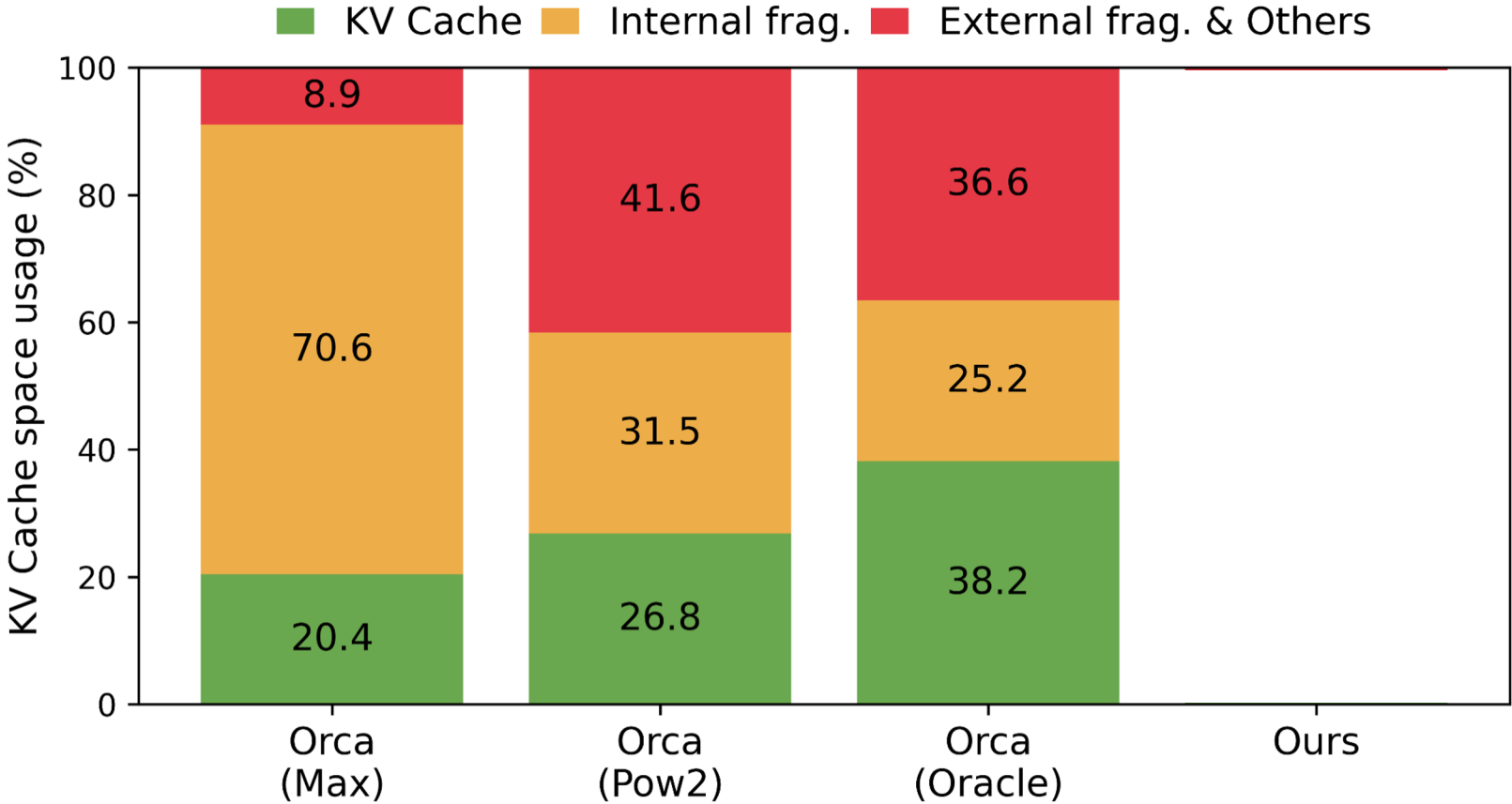
KV Cache is **huge**:

- Each token: ~1 MB.
- One full request: ~several GBs.

Input

Significant memory waste in KV Cache space

- Only **20–40%** of KV Cache space is utilized to store actual token states

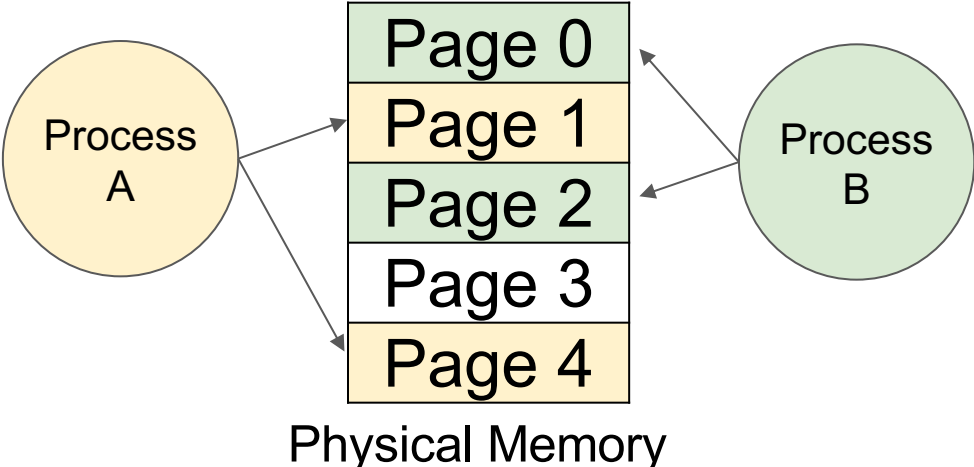


* Yu et al. “Orca: A Distributed Serving System for Transformer-Based Generative Models” (OSDI 22).

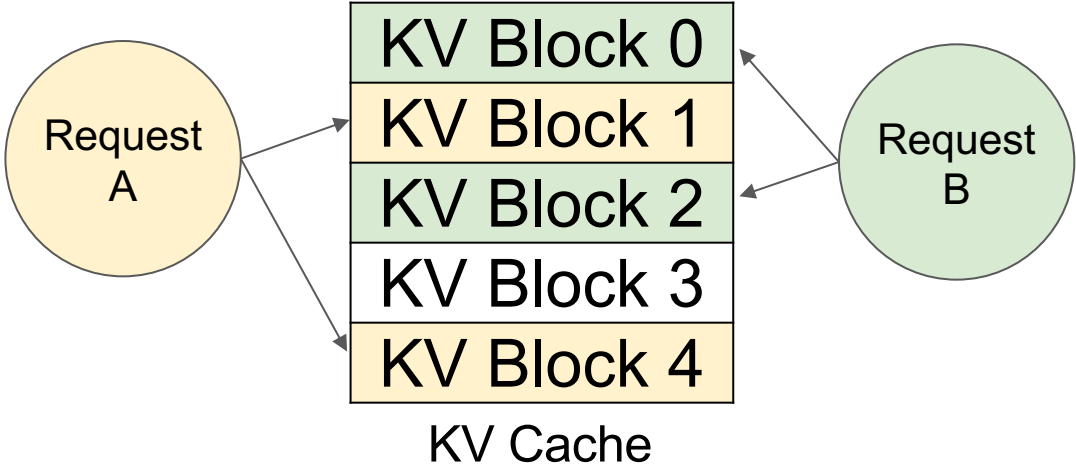
PagedAttention

- **Application-level** memory **paging** and **virtualization** for attention KV Cache

Memory management in OS

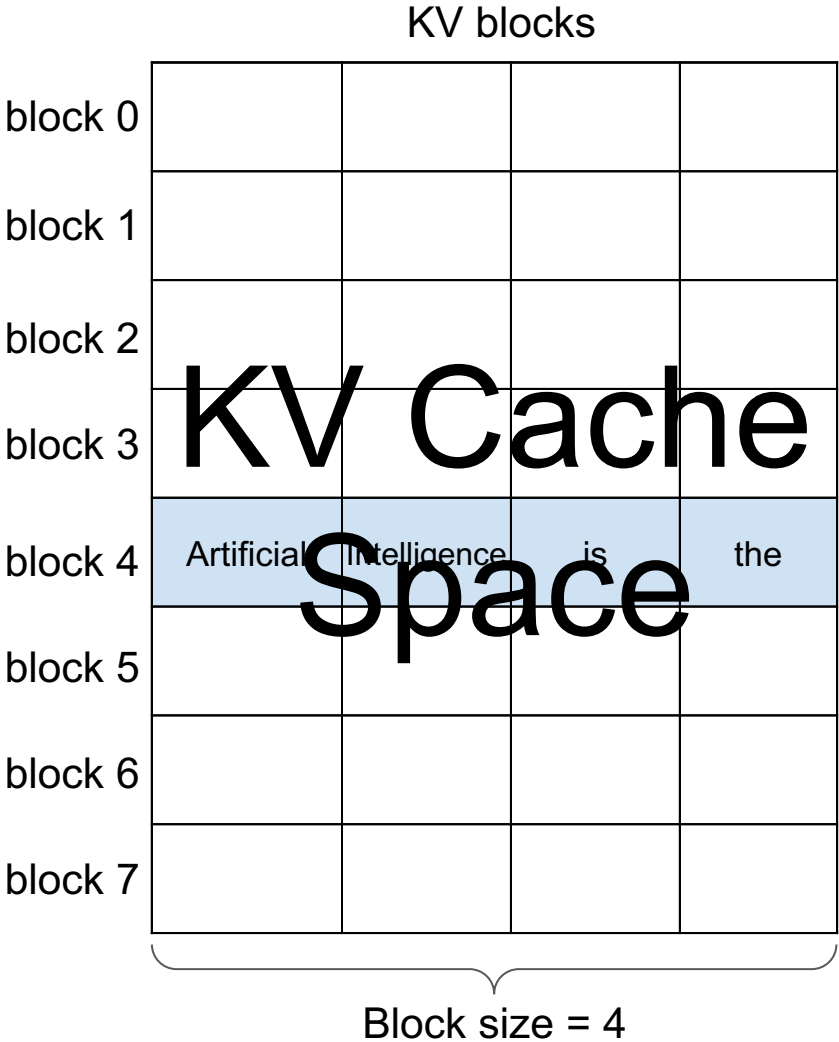


PagedAttention

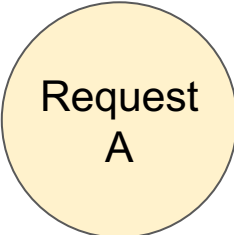


Paging KV Cache space into KV blocks

- KV block is a **fixed-size** contiguous chunk of memory that can store KV token states **from left to right**



Virtualizing KV Cache



Prompt: "Alan Turing is a computer scientist"

Logical KV blocks

block 0	Alan	Turing	is	a
block 1	computer	scientist		
block 2				
block 3				

Block table

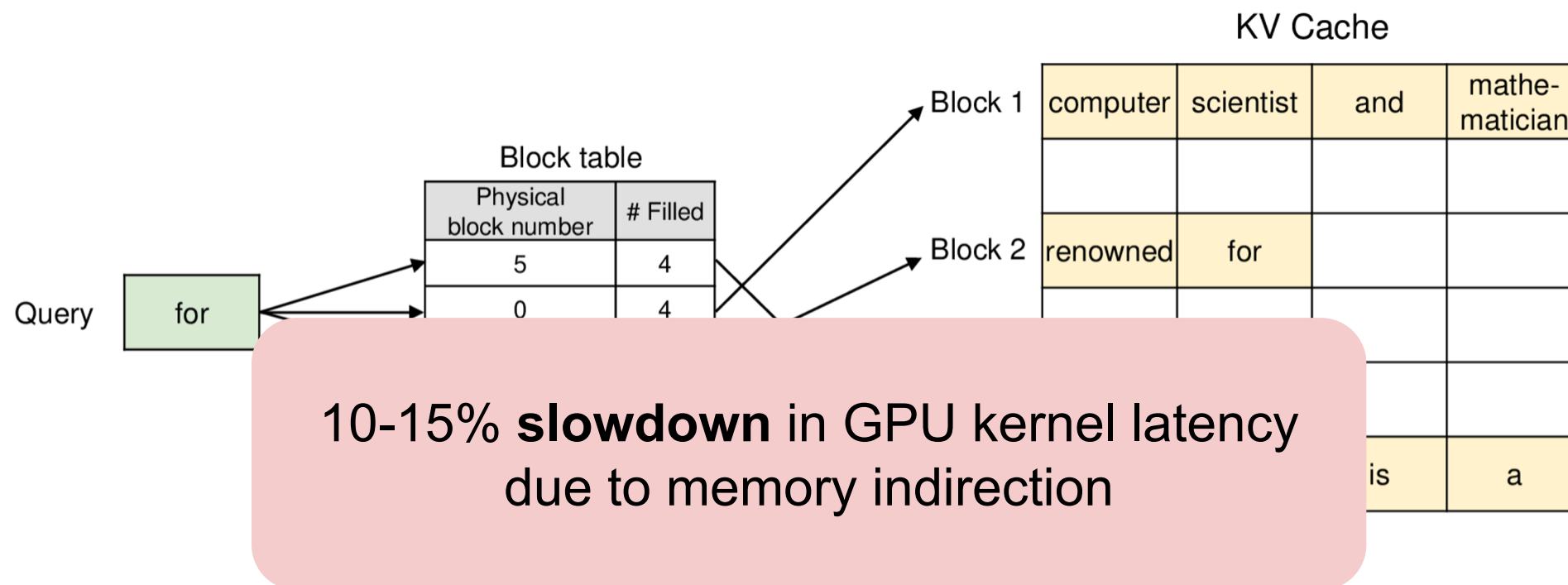
Physical block number	# Filled
7	4
1	2
-	-
-	-

Physical KV blocks

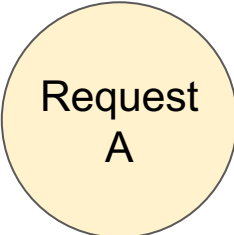
block 0				
block 1	computer	scientist		
block 2				
block 3				
block 4				
block 5				
block 6				
block 7	Alan	Turing	is	a

Attention mechanism with virtualized KV Cache

1. Fetch non-contiguous KV blocks using the block table
2. Apply attention operation on the fly



Memory management with PagedAttention



Prompt: "Alan Turing is a computer scientist"
Completion: "and"

Logical KV blocks

block 0	Alan	Turing	is	a
block 1	computer	scientist		
block 2				
block 3				

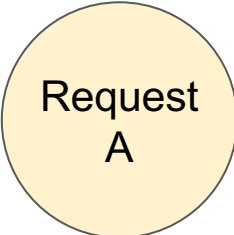
Block table

Physical block number	# Filled
7	4
1	2
-	-
-	-

Physical KV blocks

block 0				
block 1	computer	scientist		
block 2				
block 3				
block 4				
block 5				
block 6				
block 7	Alan	Turing	is	a

Memory management with PagedAttention



Prompt: "Alan Turing is a computer scientist"
Completion: "and"

Logical KV blocks

block 0	Alan	Turing	is	a
block 1	computer	scientist	and	
block 2				
block 3				

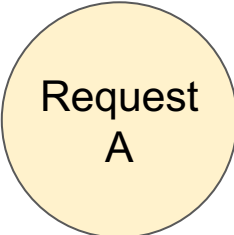
Block table

Physical block number	# Filled
7	4
1	2
-	-
-	-

Physical KV blocks

block 0				
block 1	computer	scientist		
block 2				
block 3				
block 4				
block 5				
block 6				
block 7	Alan	Turing	is	a

Memory management with PagedAttention



Prompt: "Alan Turing is a computer scientist"
Completion: "and"

Logical KV blocks

block 0	Alan	Turing	is	a
block 1	computer	scientist	and	
block 2				
block 3				

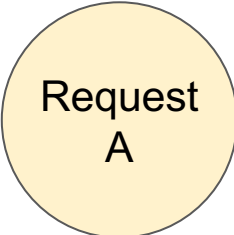
Block table

Physical block number	# Filled
7	4
1	3
-	-
-	-

Physical KV blocks

block 0				
block 1	computer	scientist	and	
block 2				
block 3				
block 4				
block 5				
block 6				
block 7	Alan	Turing	is	a

Memory management with PagedAttention



Prompt: "Alan Turing is a computer scientist"
 Completion: "and mathematician"

Logical KV blocks

block 0	Alan	Turing	is	a
block 1	computer	scientist	and	mathematician
block 2				
block 3				

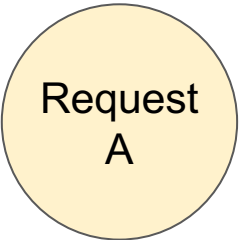
Block table

Physical block number	# Filled
7	4
1	4
-	-
-	-

Physical KV blocks

block 0				
block 1	computer	scientist	and	mathematician
block 2				
block 3				
block 4				
block 5				
block 6				
block 7	Alan	Turing	is	a

Memory management with PagedAttention



Prompt: "Alan Turing is a computer scientist"
Completion: "and mathematician renowned"

Logical KV blocks

block 0	Alan	Turing	is	a
block 1	computer	scientist	and	mathem atician
block 2	renowned			
block 3				

Block table

Physical block number	# Filled
7	4
1	4
5	1
-	-

Physical KV blocks

block 0				
block 1	computer	scientist	and	mathem atician
block 2				
block 3				
block 4				
block 5	renowned			
block 6				
block 7	Alan	Turing	is	a

Allocated on demand

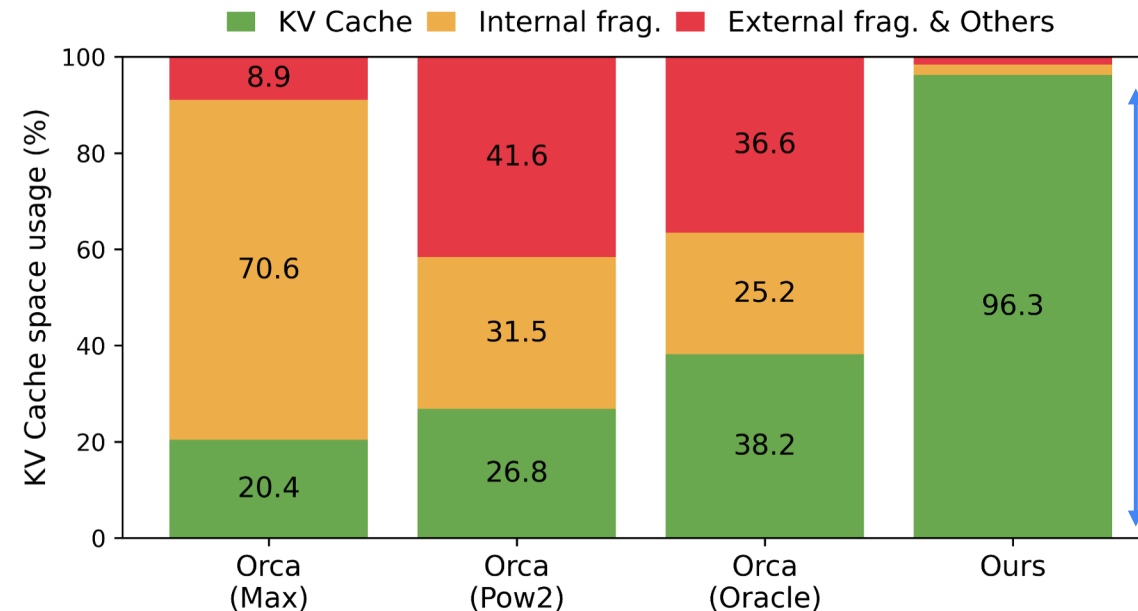


Memory efficiency of PagedAttention

- Minimal internal fragmentation
 - Only happens at the last block of a sequence
 - **# wasted tokens / seq < block size**
 - Sequence: $O(100) - O(1000)$ tokens
 - Block size: $O(10)$ tokens
- No external fragmentation

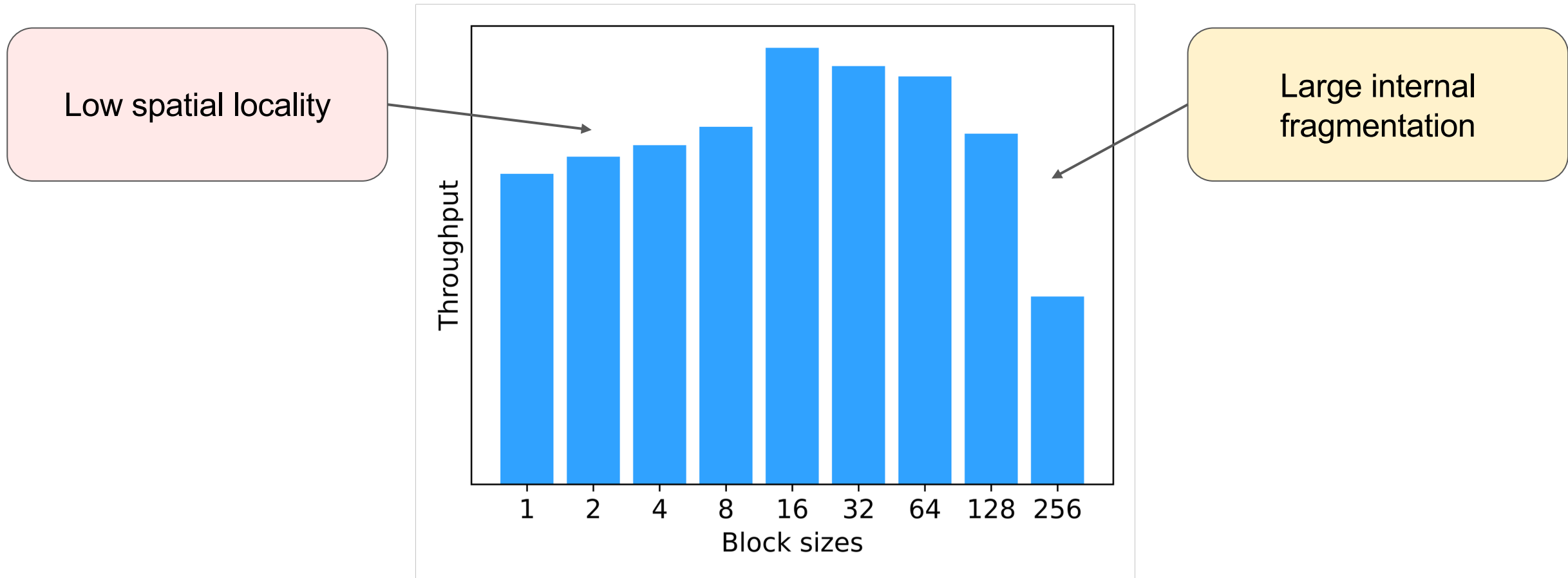
Alan	Turing	is	a
computer	scientist	and	mathematician
renowned			

Internal fragmentation



2.5-5x improvement

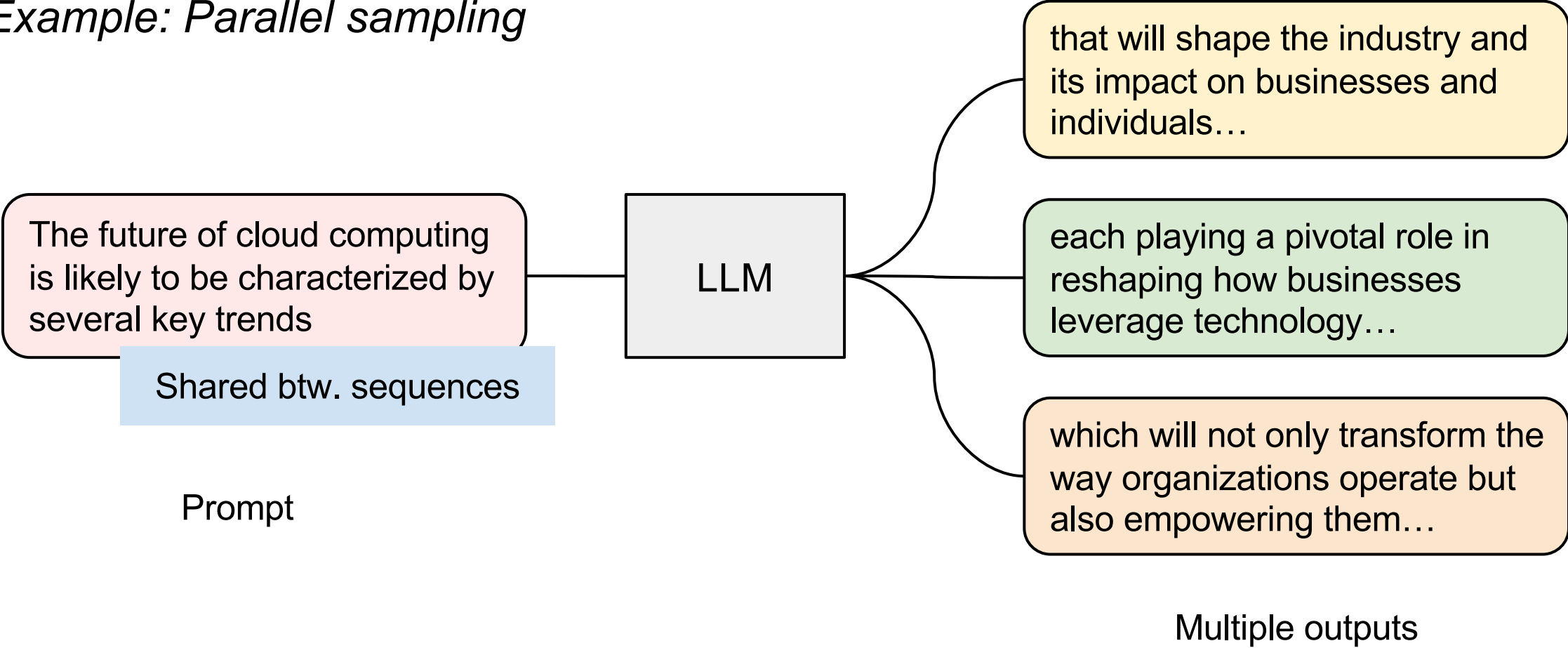
Configuring the block size



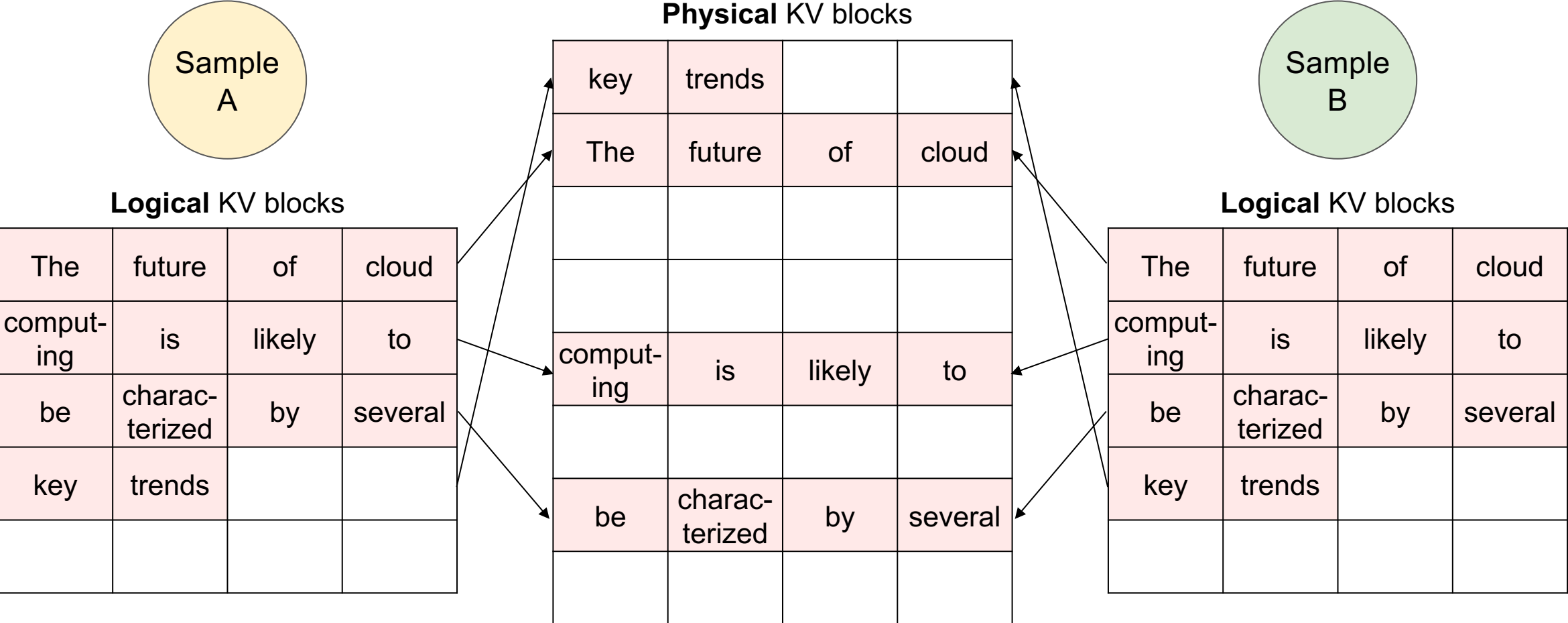
- **Block size 16** works generally well in practice

Paging enables sharing

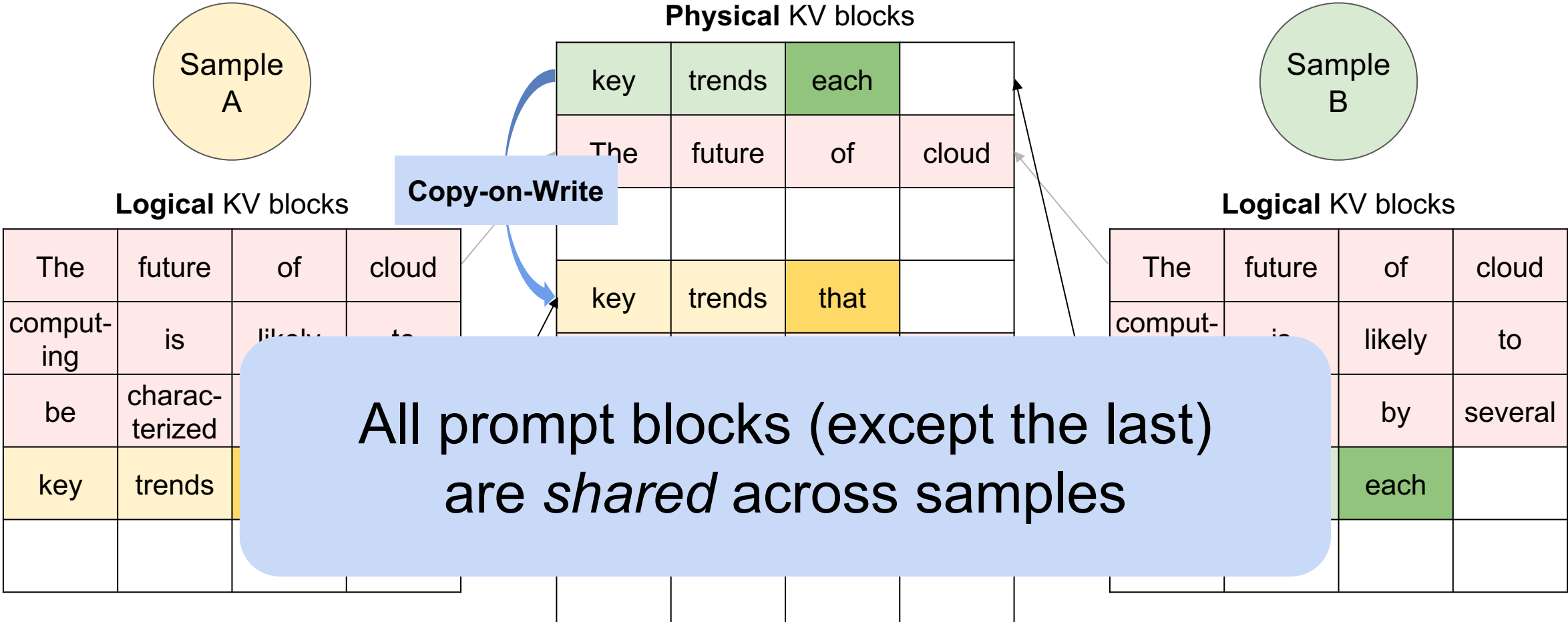
Example: Parallel sampling



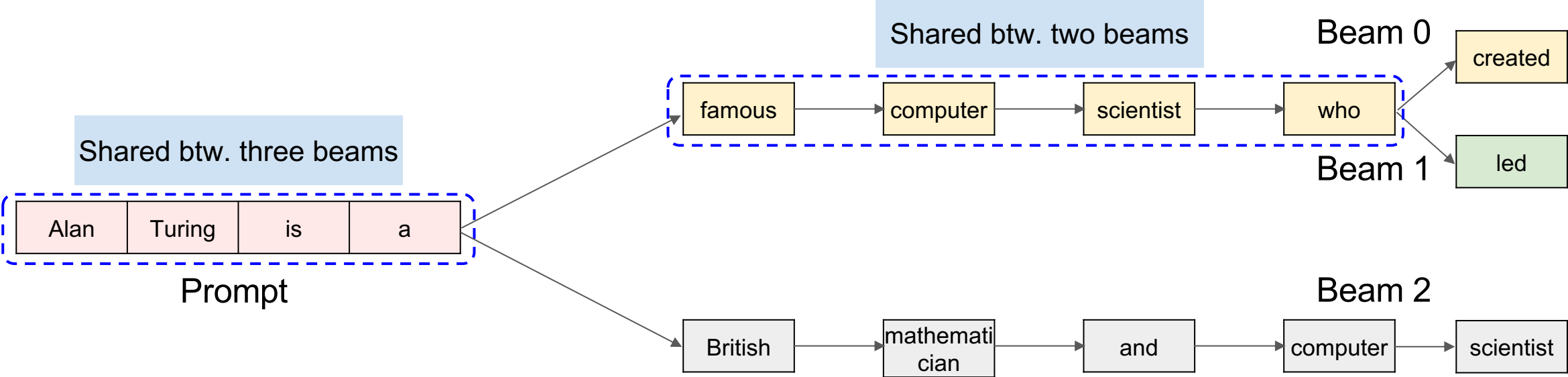
Sharing KV blocks



Sharing KV blocks

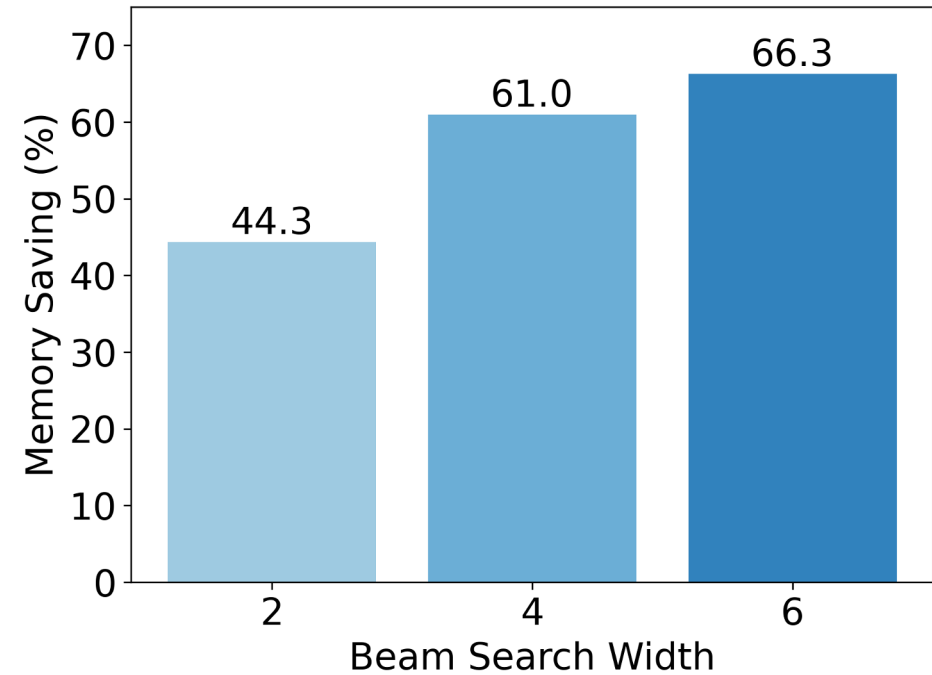
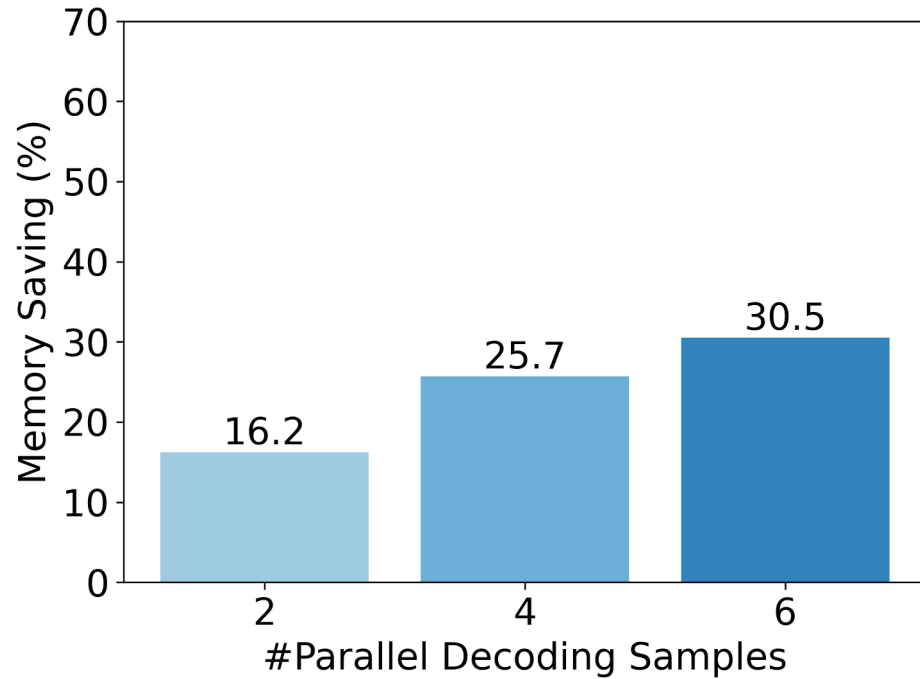


More complex sharing: beam search



- Similar to process tree (fork & kill)
- Efficiently supported by paged attention and copy-on-write mechanism

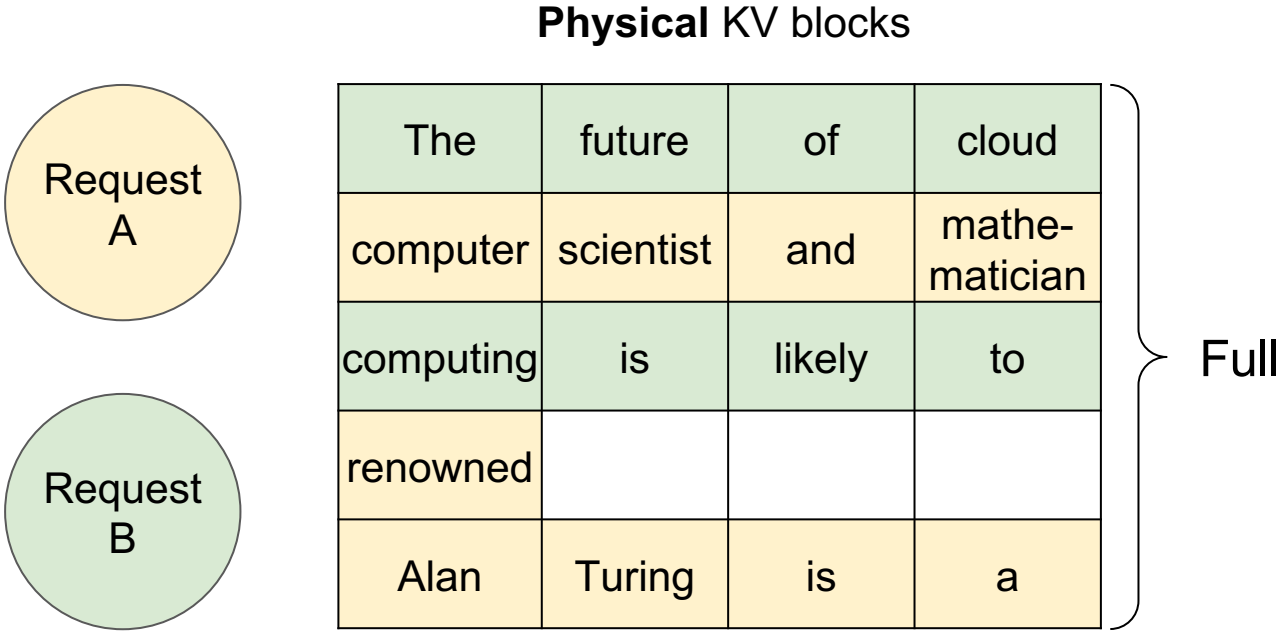
Memory saving via sharing



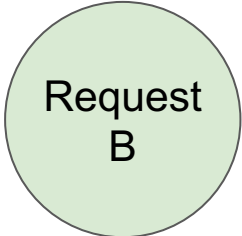
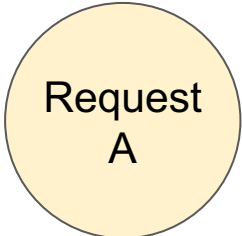
Percentage = (#blocks saved by sharing) / (#total blocks without sharing)

OPT-13B on 1x A100-40G with ShareGPT dataset

Out of KV Block Memory

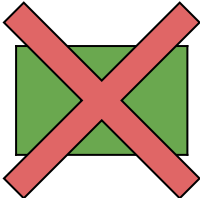


Out of KV Block Memory

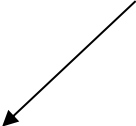


Physical KV blocks

The	future	of	cloud
computer	scientist	and	mathe- matician
computing	is	likely	to
renowned	for		
Alan	Turing	is	a



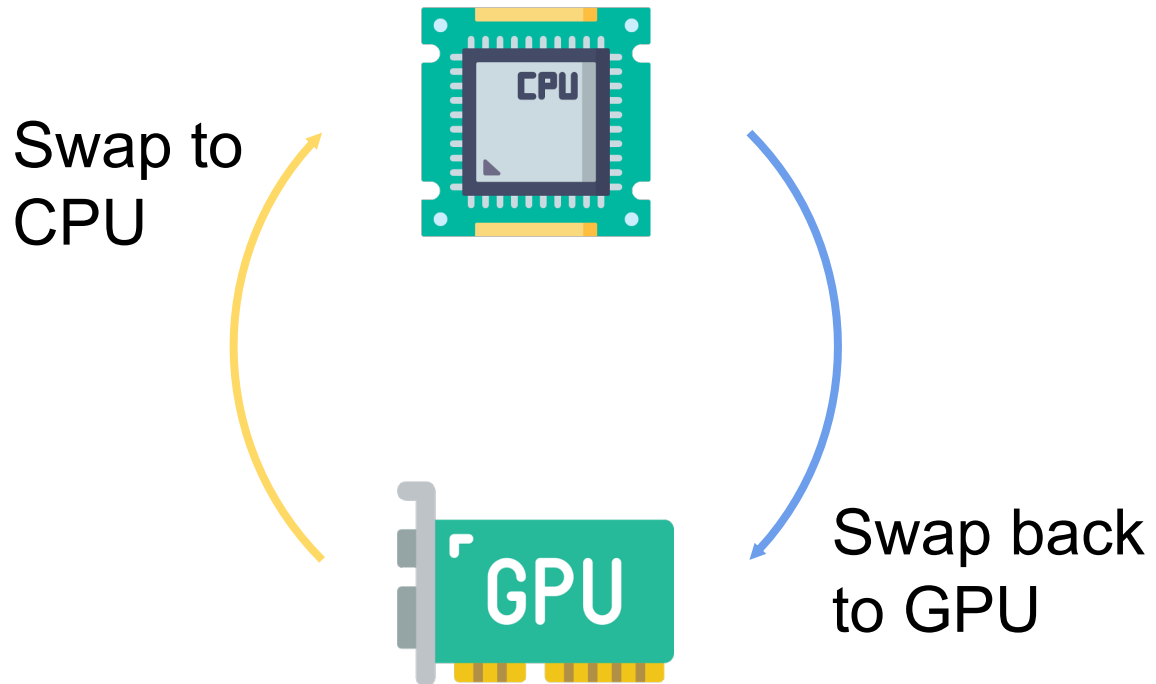
Cannot allocate a new physical block for Request B



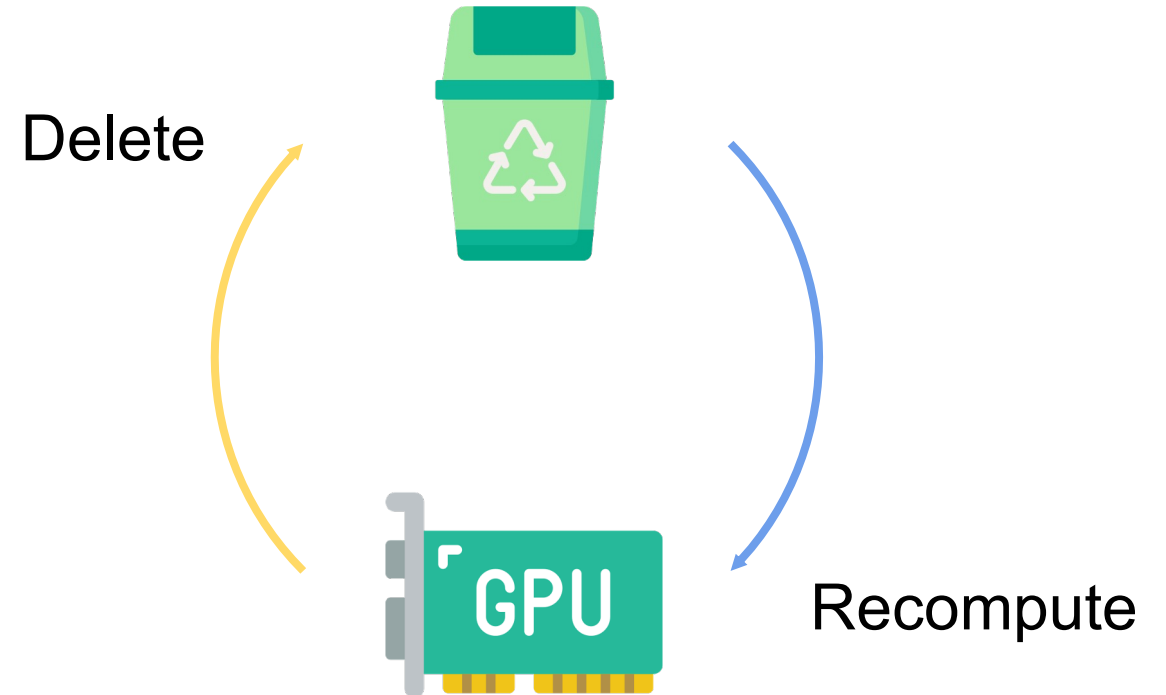
Request Preemption & Recovery

Goal: Free some requests' KV cache to let others run first.

Option 1: Swapping



Option 2: Recomputation



Notes on Preemption & Recovery

Swap/recompute **the whole request**, since all previous tokens are required every step.

Swapping: smaller block sizes → higher overhead due to small data transfers.

Recomputation: surprisingly fast since all token's KV cache can be computed in parallel.

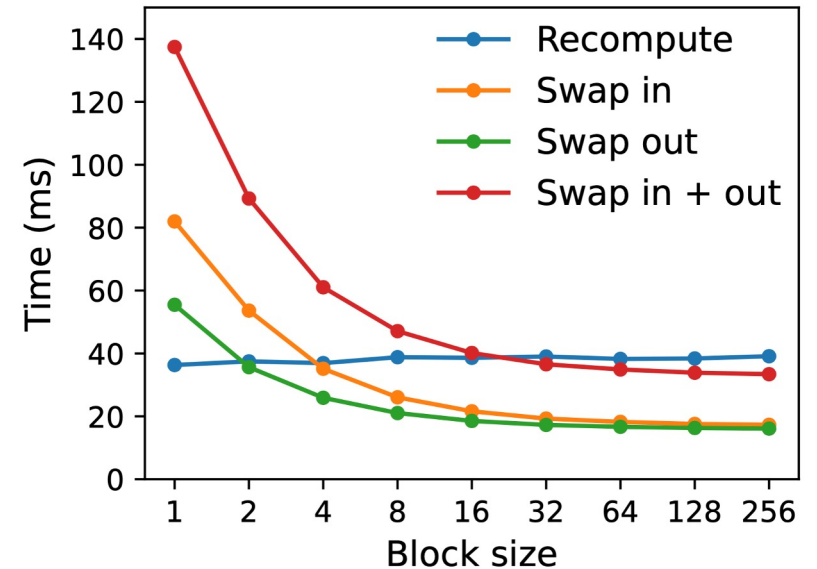


Figure: Swap/Recomputation latency of 256 tokens.

Our strategy: Use recomputation when possible with FCFS policy

Comparison with Operating System Virtual Memory

Analogies

OS pages ↔ KV blocks

- Reduce memory fragmentation

Shared pages across processes

↔ Shared KV blocks across samples

- Reduce memory waste via sharing

Differences

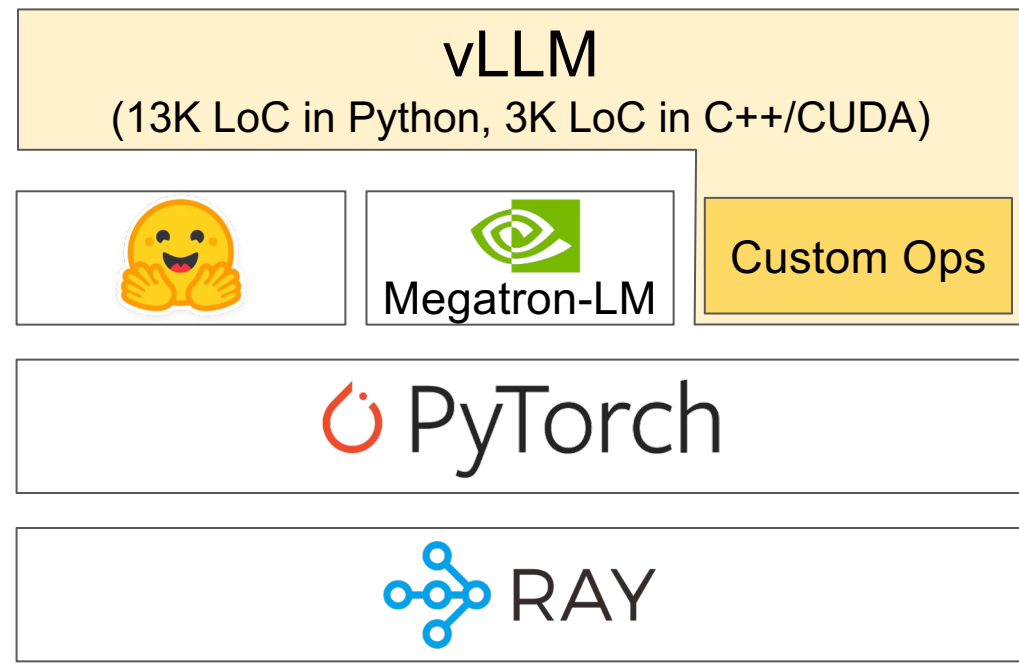
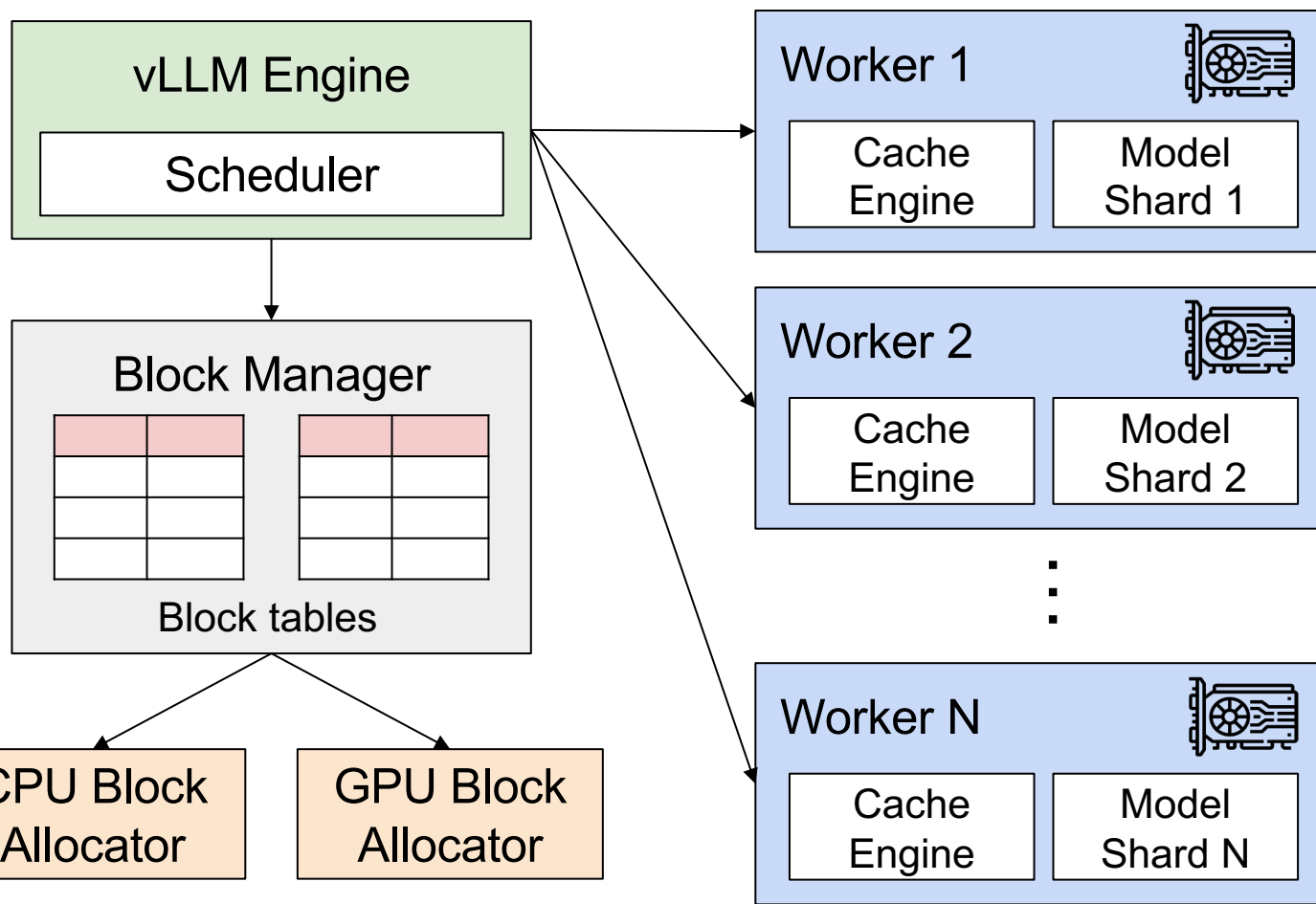
Single-level block table

- Block table is tiny compared to the actual data.

Preemption & Recovery

- Request-level preemption
- Recomputation-based recovery

vLLM Distributed System Architecture & Implementation



Evaluation – Settings

Metric: Serving throughput

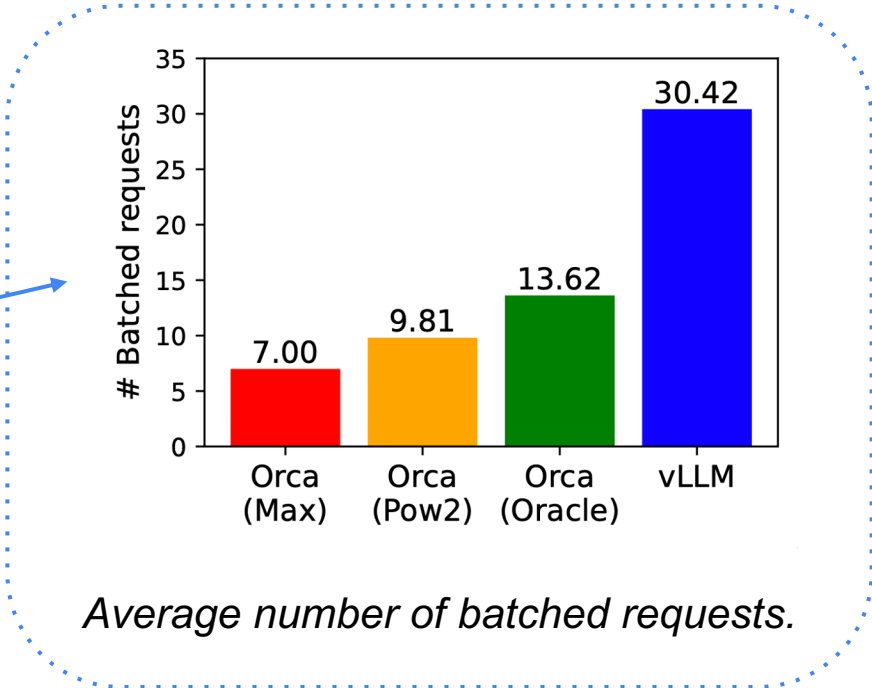
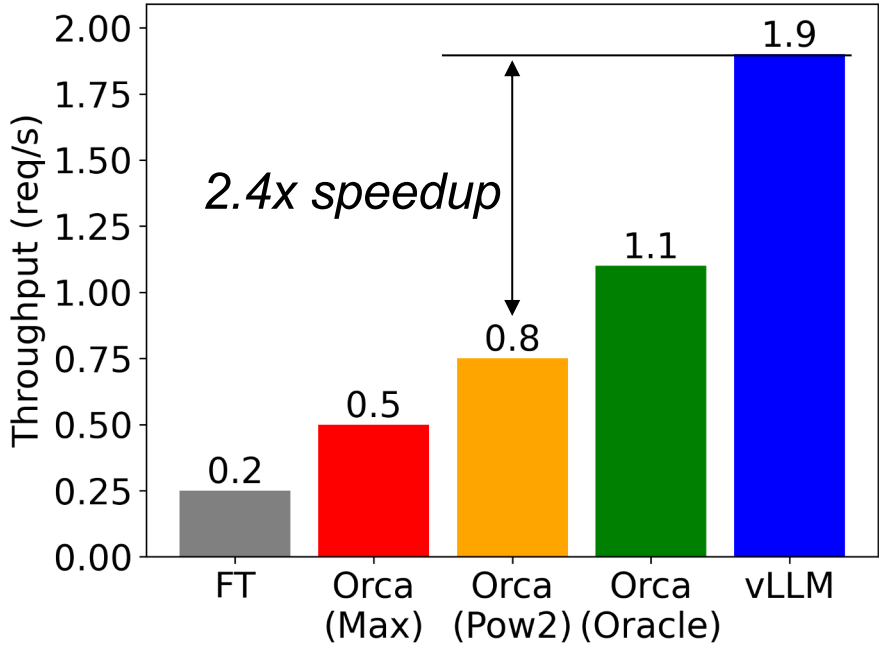
Input/Output Length Distribution

- Alpaca dataset (instruction-following)
- ShareGPT dataset (conversation)

Baselines

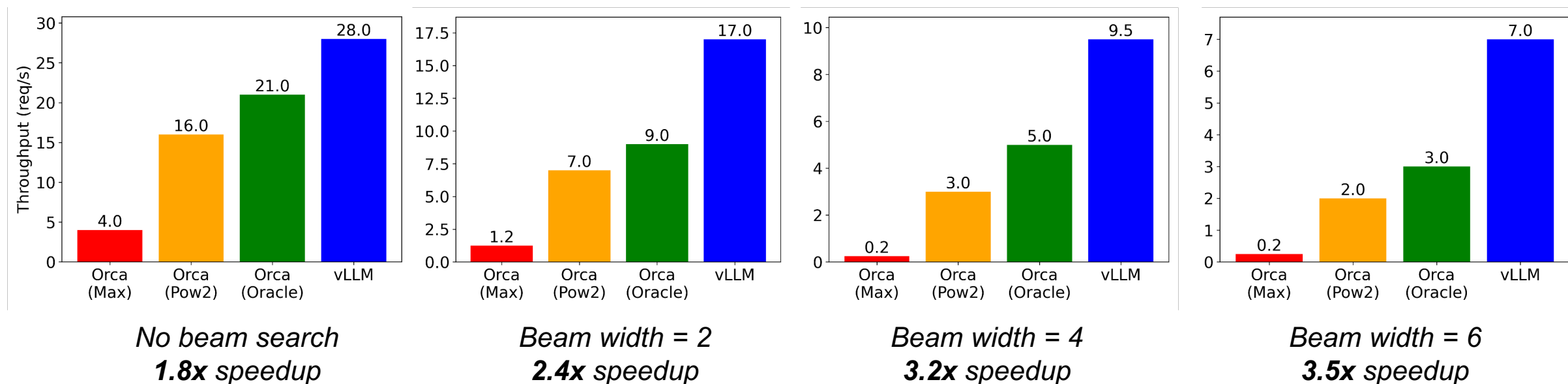
- NVIDIA FasterTransformer (FT)
- Orca
 - Oracle: No over-reserve and know exact output lengths.
 - Pow2: Over-reserve the space for outputs by at most 2x.
 - Max: Over-reserve to the maximum possible output length.

Throughput – Greedy Decoding



OPT-13B on 1xA100 40G with ShareGPT trace.

Throughput – Beam Search



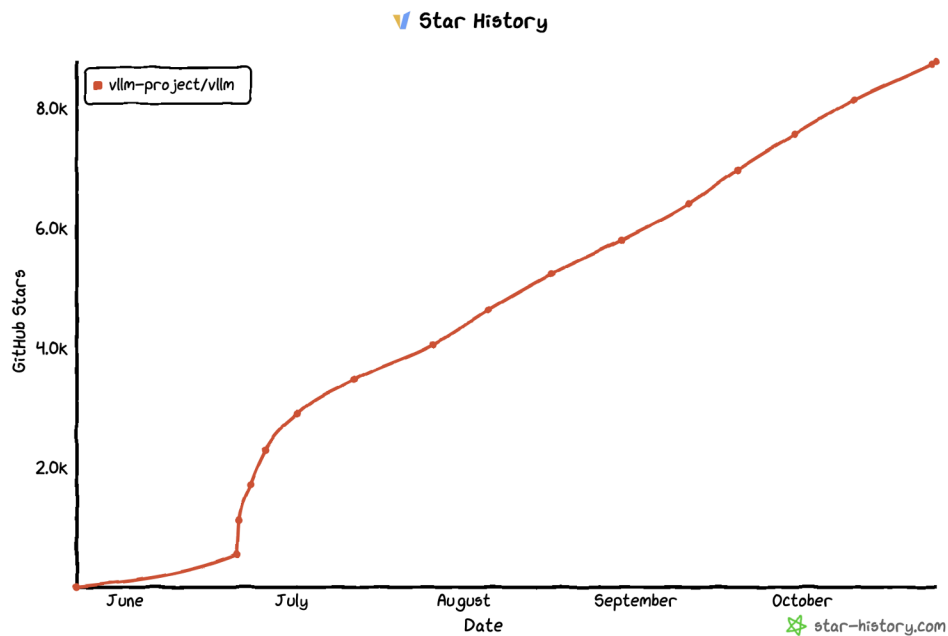
*OPT-13B on 1xA100 40G with Alpaca trace.
Speedup: vLLM v.s. Orca(Pow2)*

vLLM Open-Source Adoption

 github.com/vllm-project/vllm

 `$ pip install vllm`

 **8.8K Stars**  **95+ Contributors**



Open-Source Projects

 **lm-sys/FastChat**  **allenai/open-instruct**

Companies

 **anyscale**





 **C3.ai**

 **databricks**

 **Google Cloud**

 **Lepton AI**

 **Meta**

 **Microsoft**

 **Modal**

 **nVIDIA**

 **Replicate**



 **Twelve Labs**



...

vLLM

- Improve memory efficiency by **2.5x–5x** by **reducing fragmentation** and **enabling sharing** with *PagedAttention*
- Outperform SOTA by **up to 4x** in terms of **serving throughput**
- A vibrant open-source project that gets widely adopted



<https://github.com/vllm-project/vllm>



<https://vllm.ai>



<https://arxiv.org/abs/2309.06180>



<https://discord.gg/jz7wjKhh6g>

Memory Management in Modern Computer Systems

- Memory Abstraction
 - NSDI'14 FaRM
 - SOSPP'23 vLLM
- Demand paging: remote memory over network
 - NSDI'17 InfiniSwap
 - OSDI'20 AIFM
- Demand paging: memory swapping between GPU memory and host memory
 - OSDI'20 PipeSwitch
 - NSDI'23 TGS

Efficient Memory Disaggregation with Infiniswap

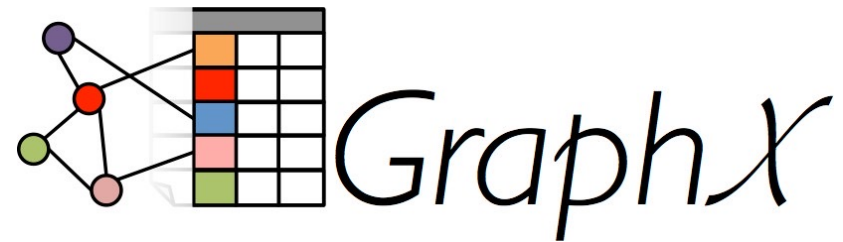
Juncheng Gu, Youngmoon Lee, Yiwen Zhang,
Mosharaf Chowdhury, Kang G. Shin



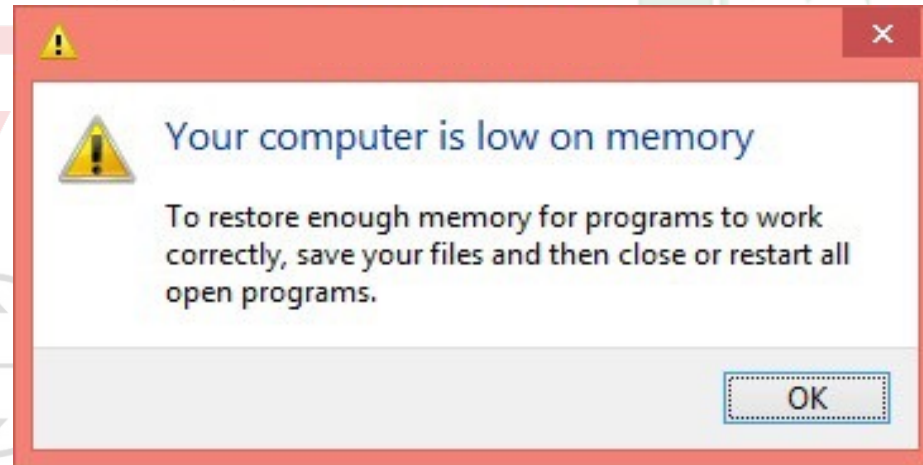
Agenda

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

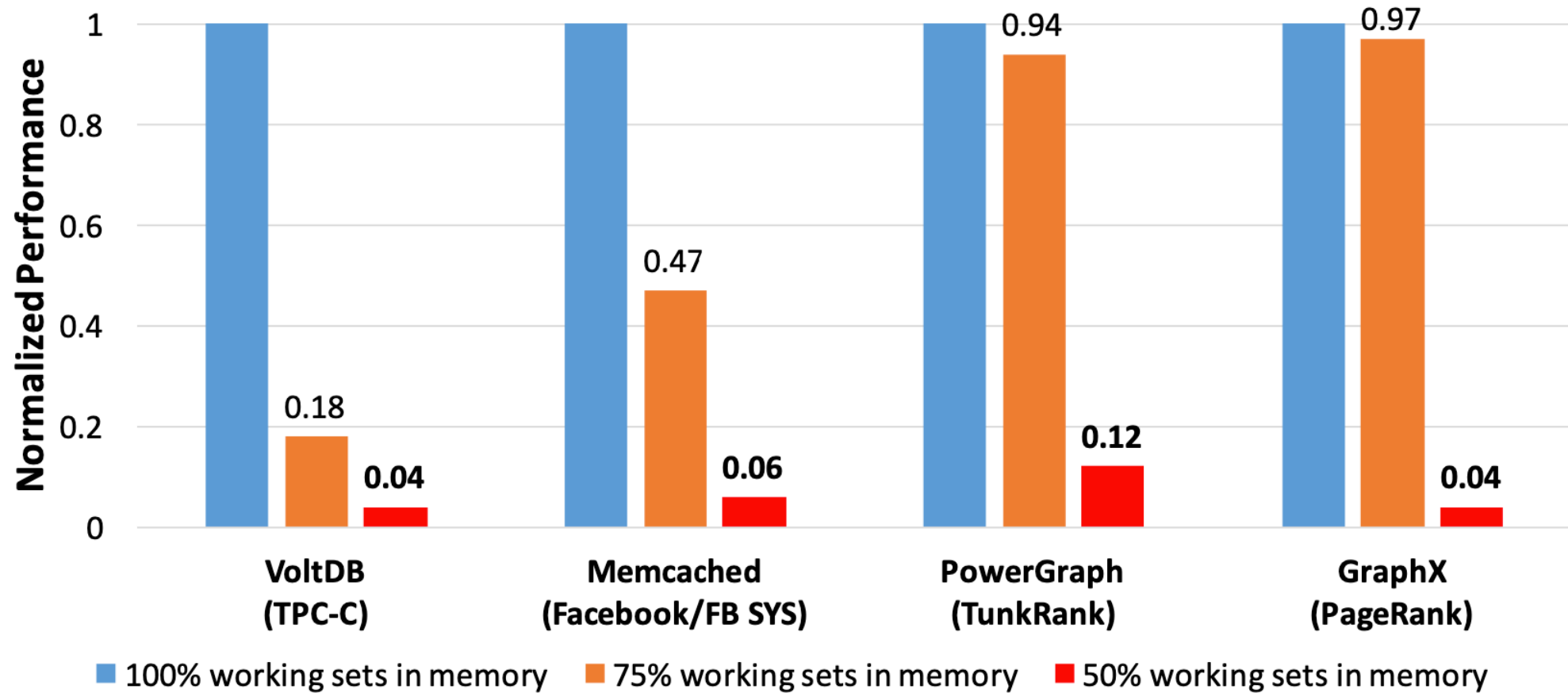
Memory-intensive applications



Memory-intensive applications

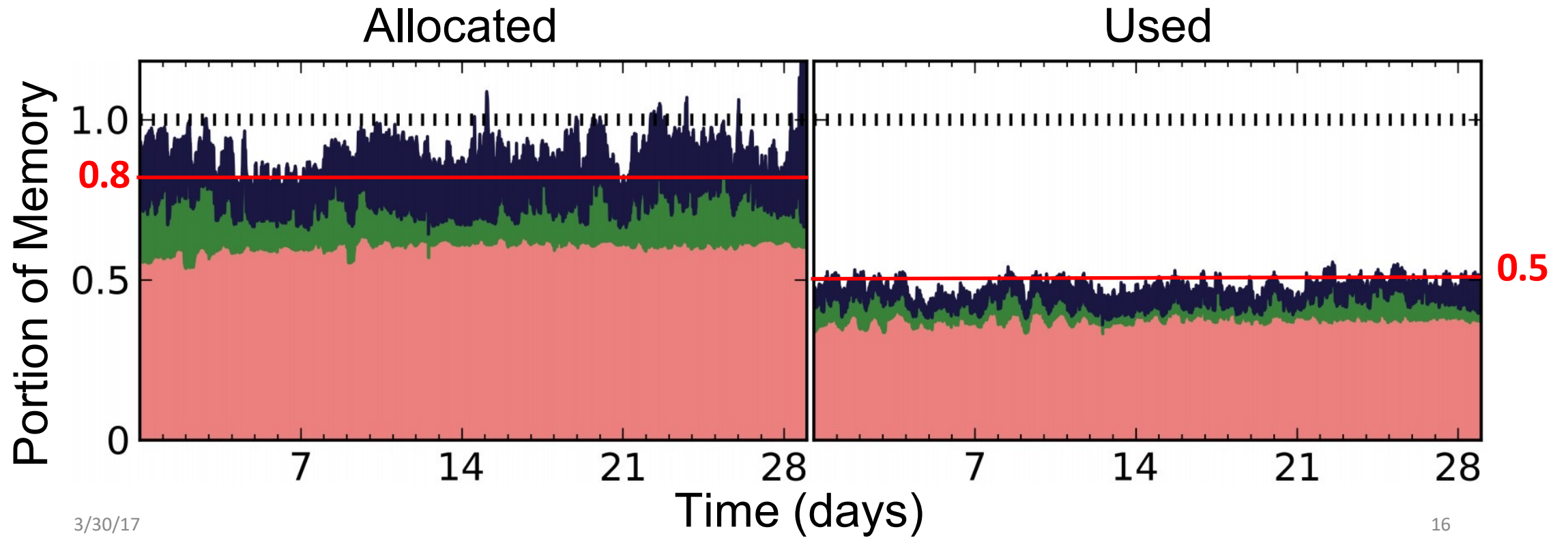


Performance degradation



Memory underutilization

- Google Cluster Analysis^[1]



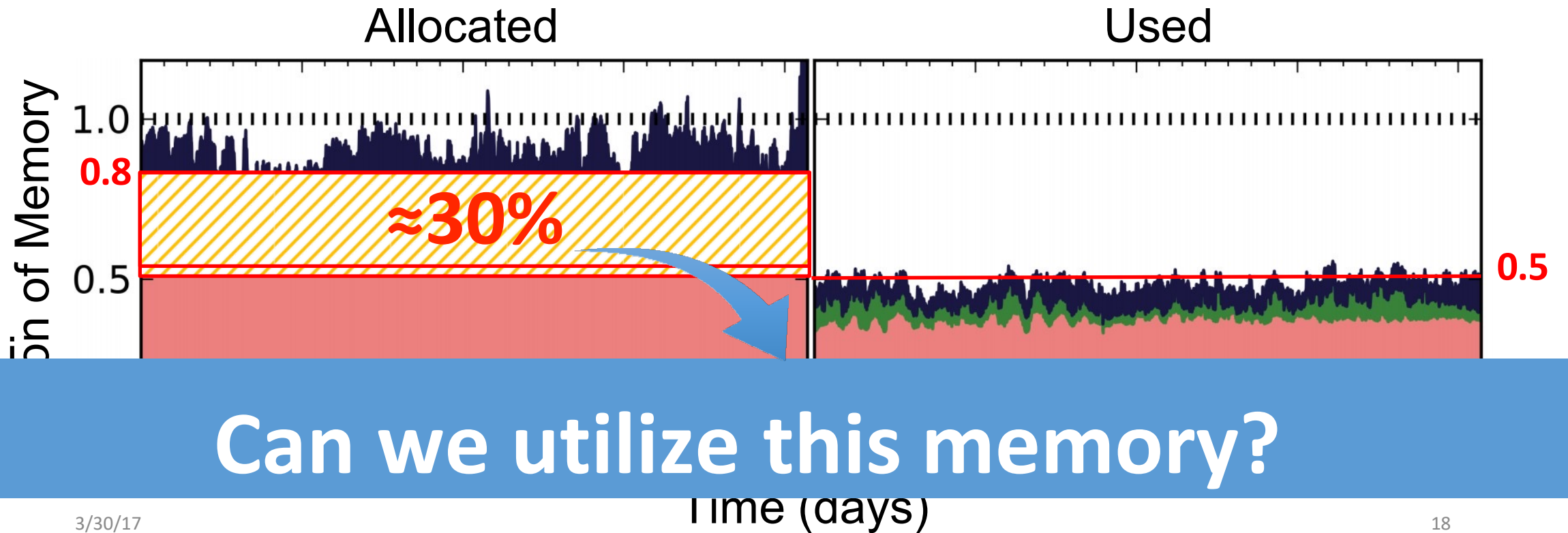
3/30/17

16

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

Memory underutilization

- Google Cluster Analysis^[1]

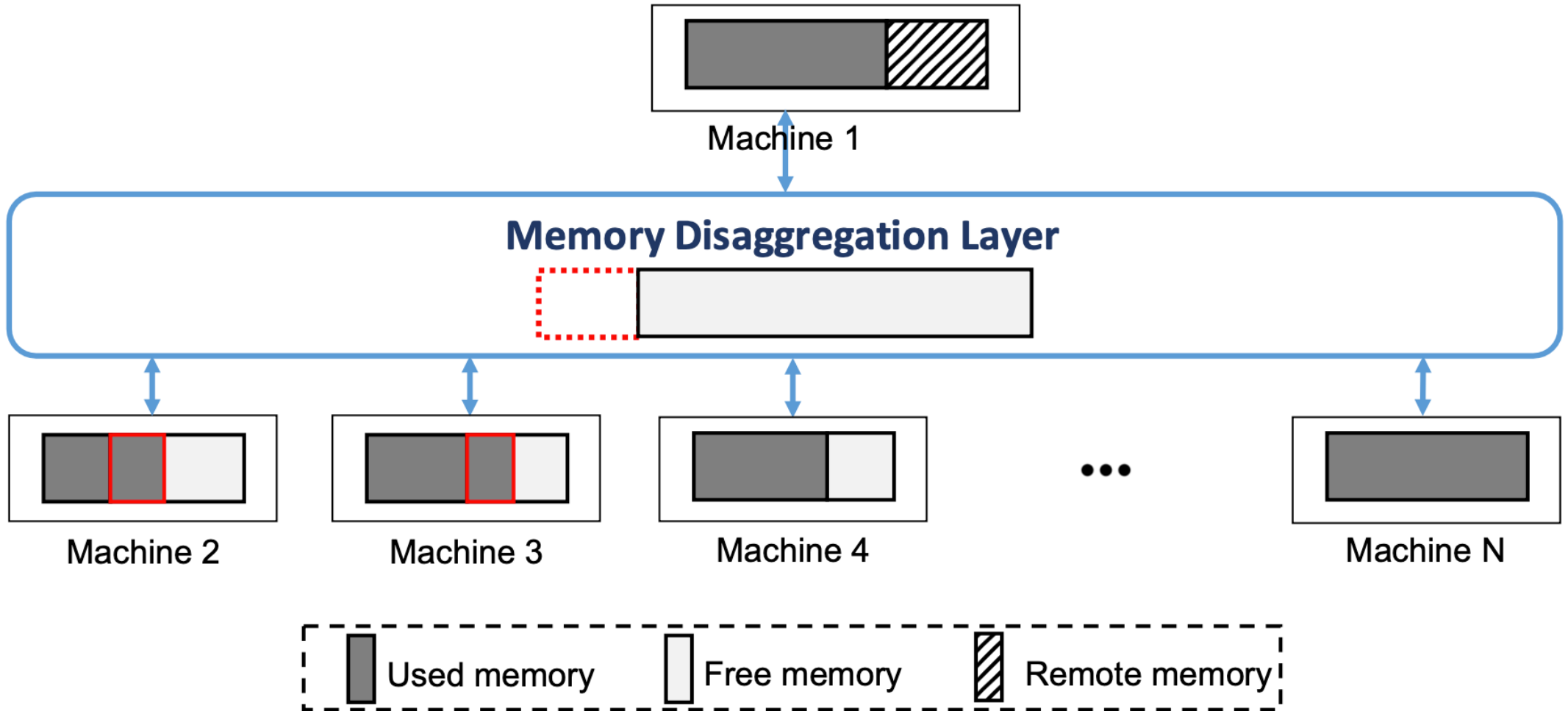


3/30/17

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**

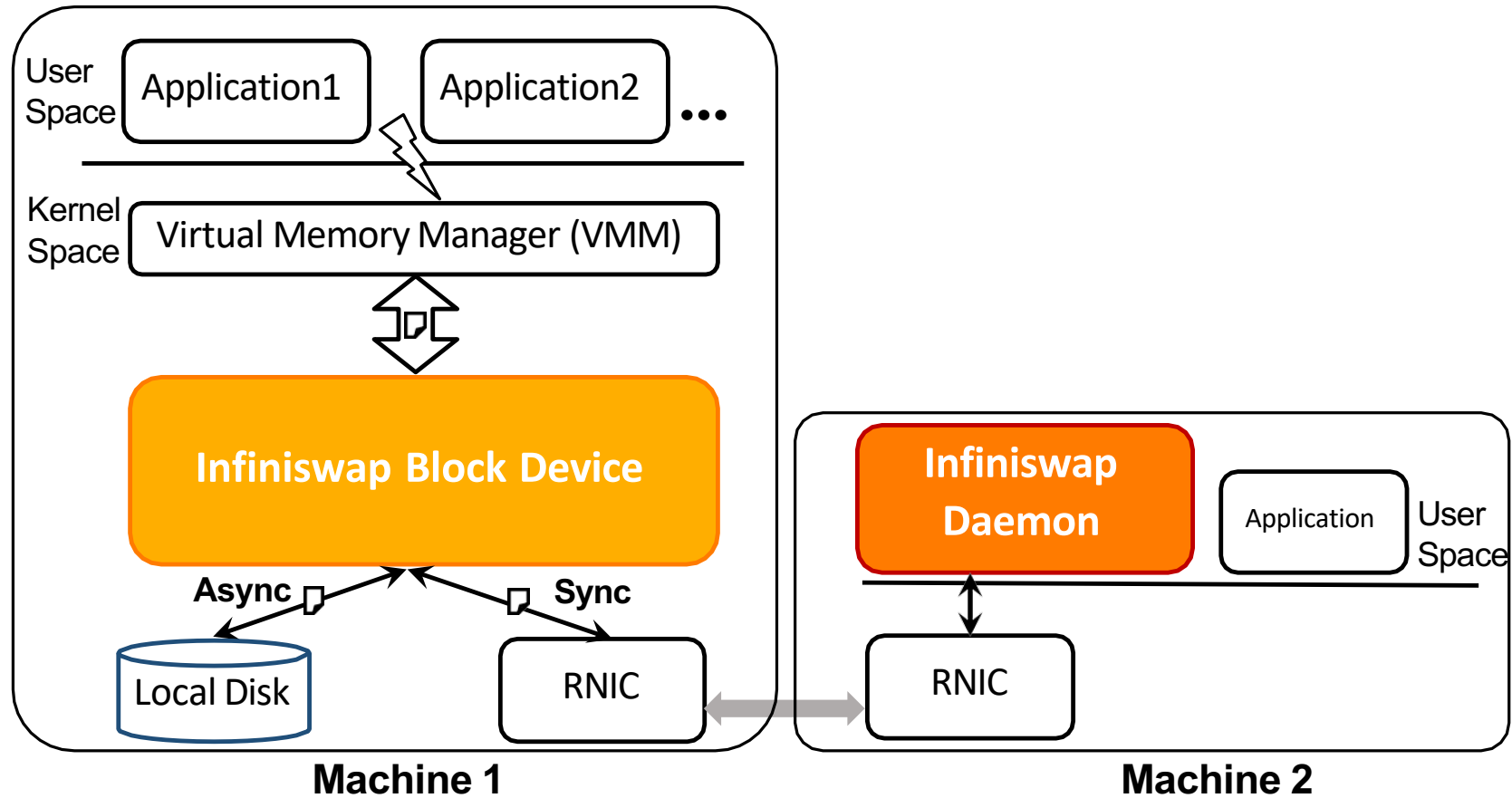
Recent work on memory disaggregation

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

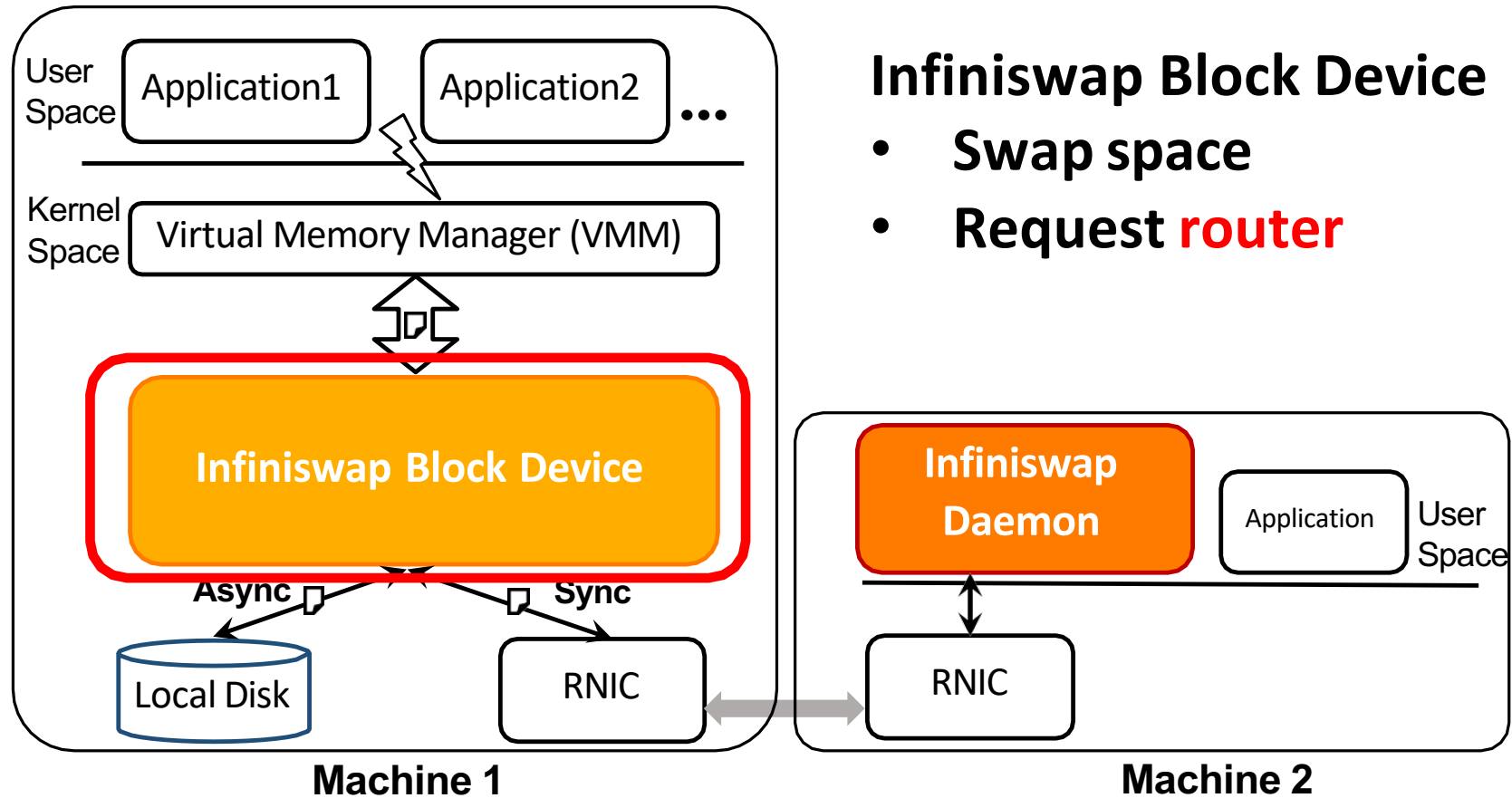
Agenda

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

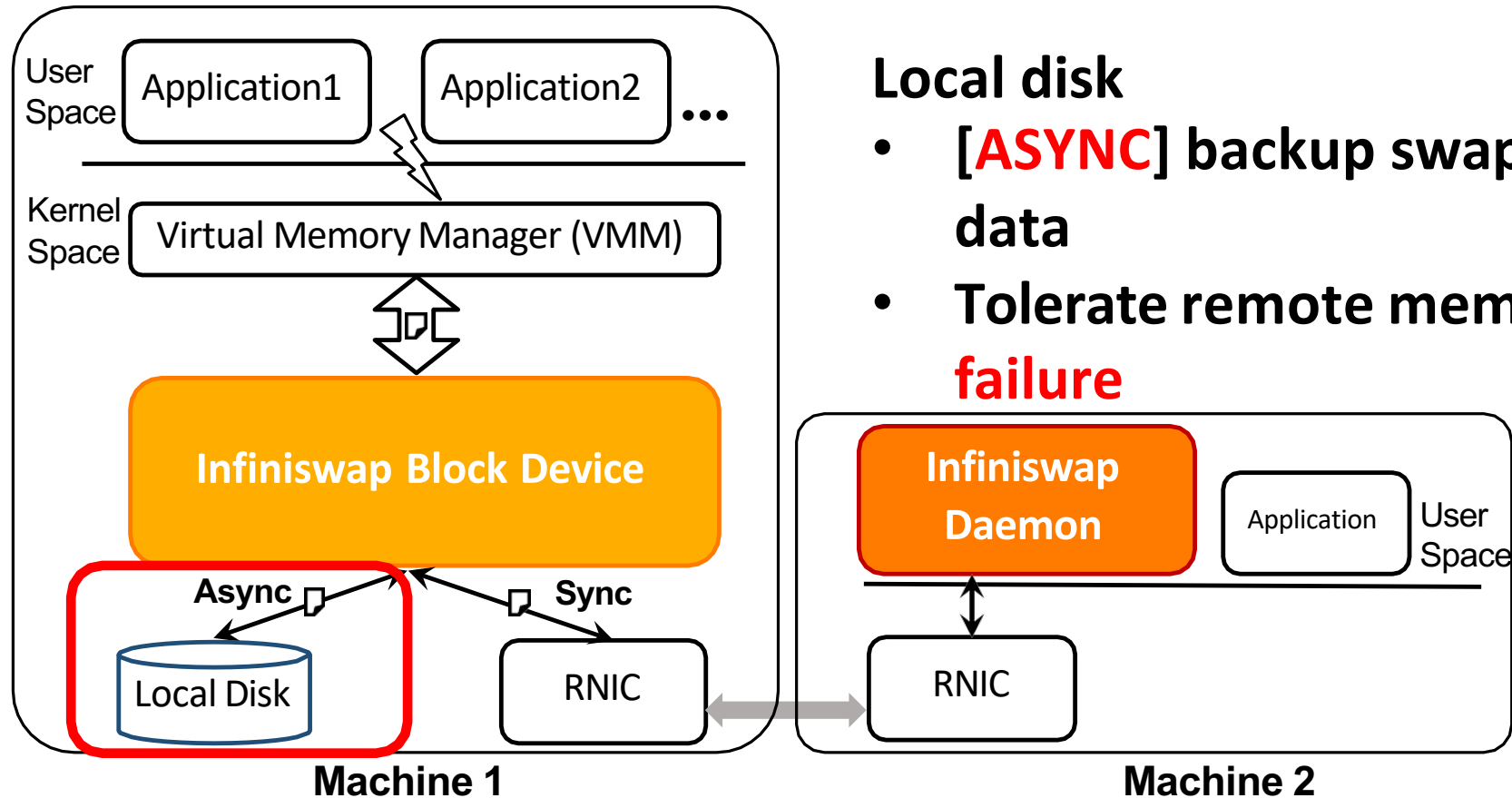
System Overview



System Overview



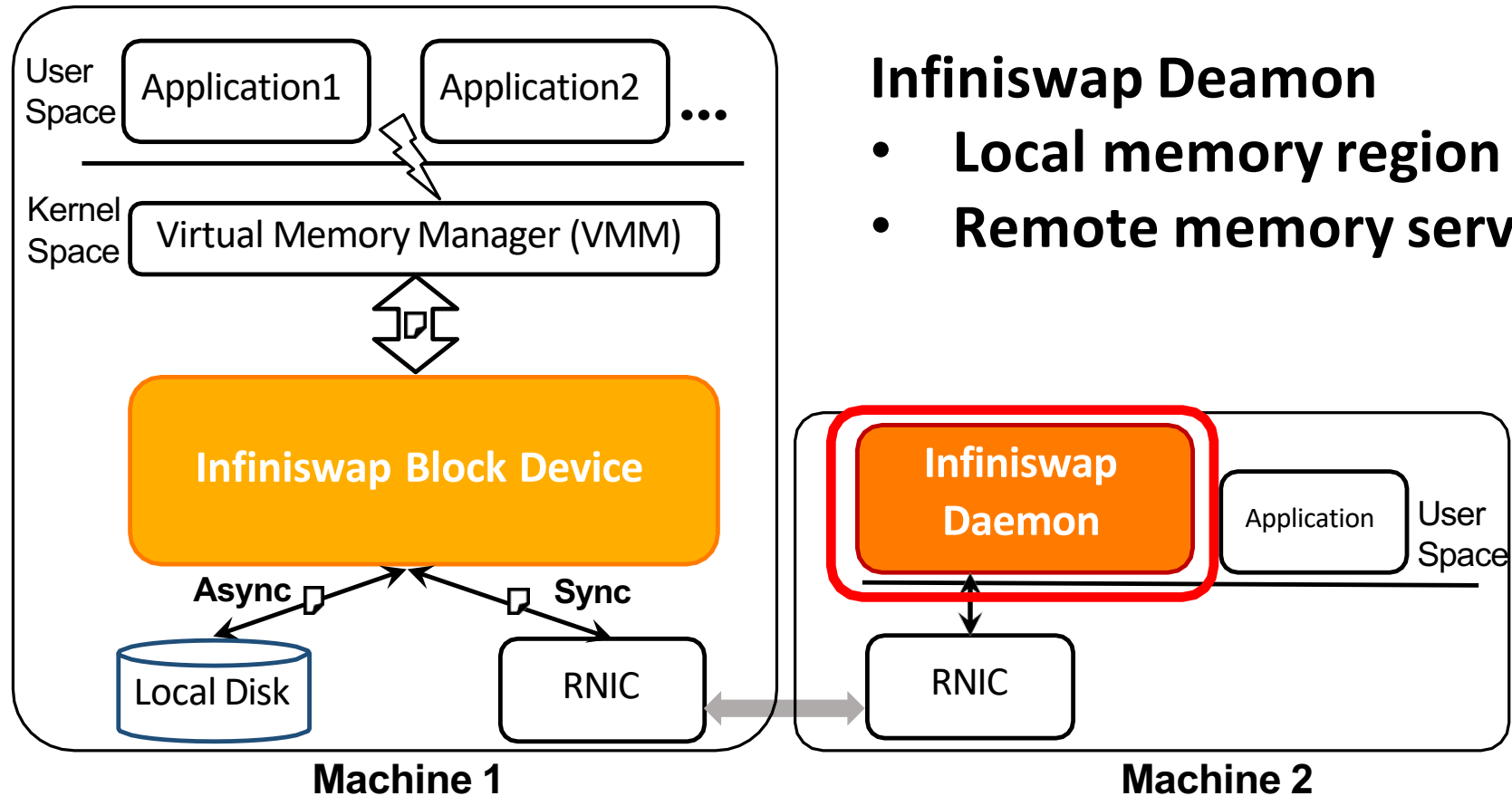
System Overview



Local disk

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

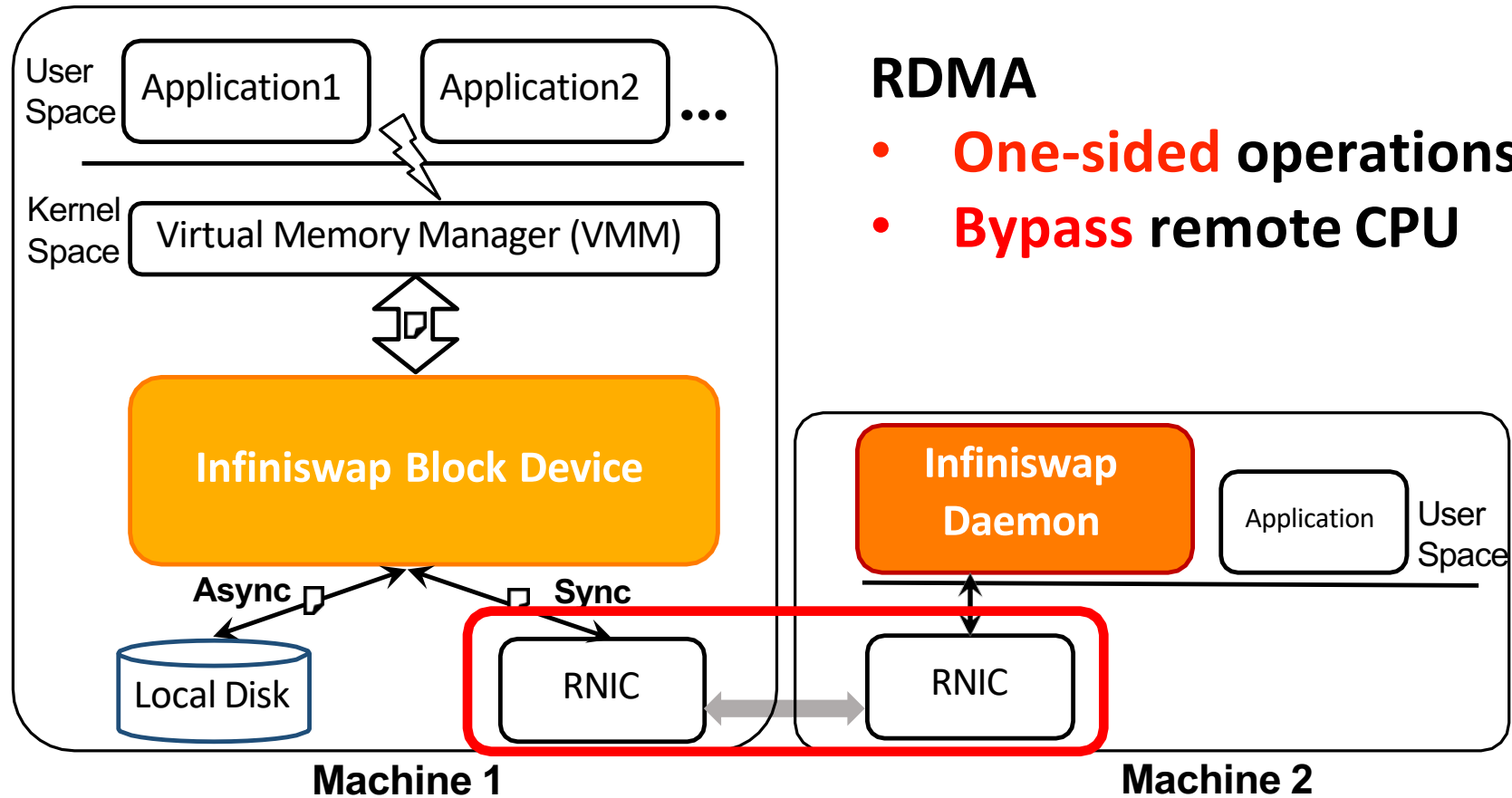
System Overview



Infiniswap Daemon

- Local memory region
- Remote memory service

System Overview



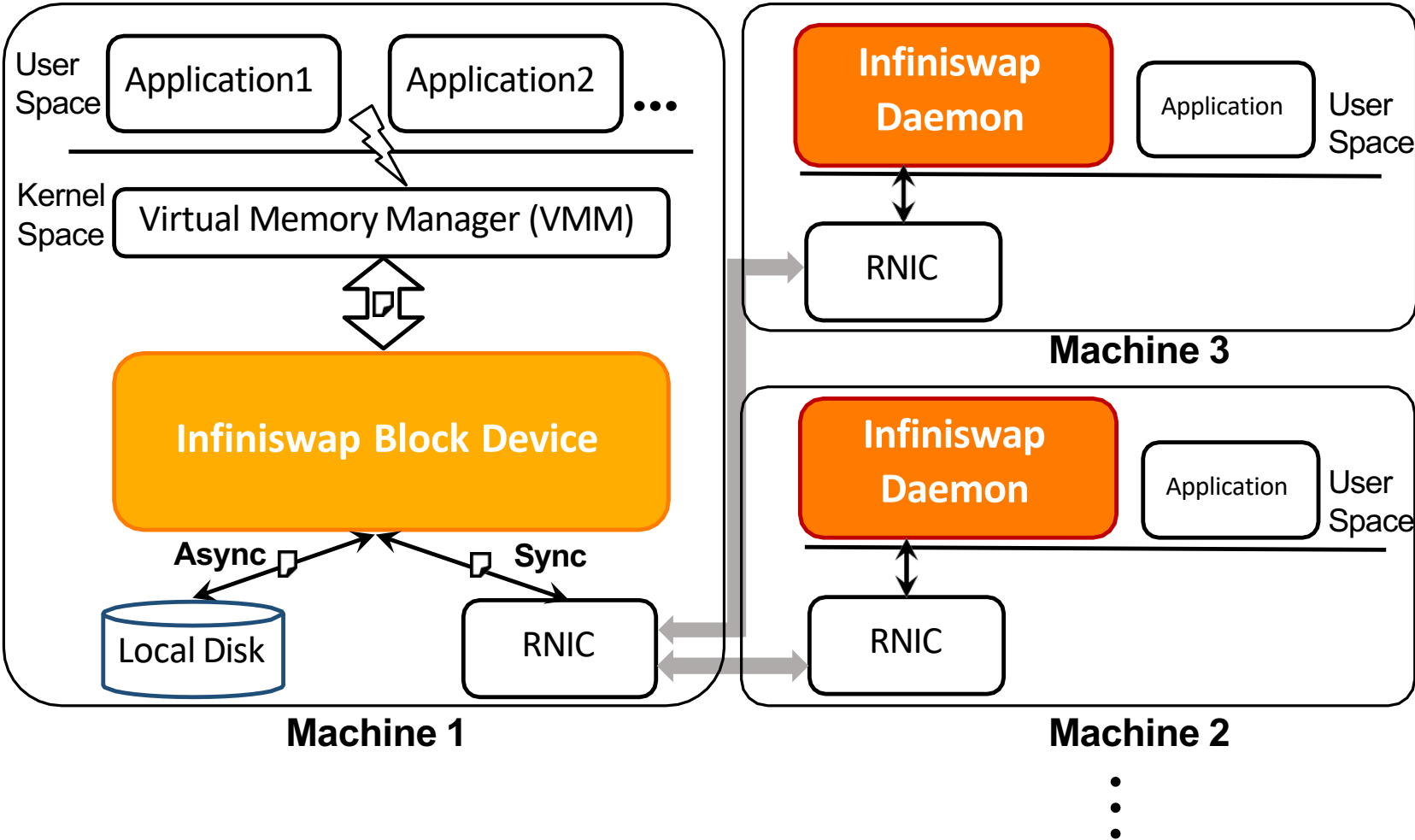
RDMA

- **One-sided** operations
- **Bypass** remote CPU

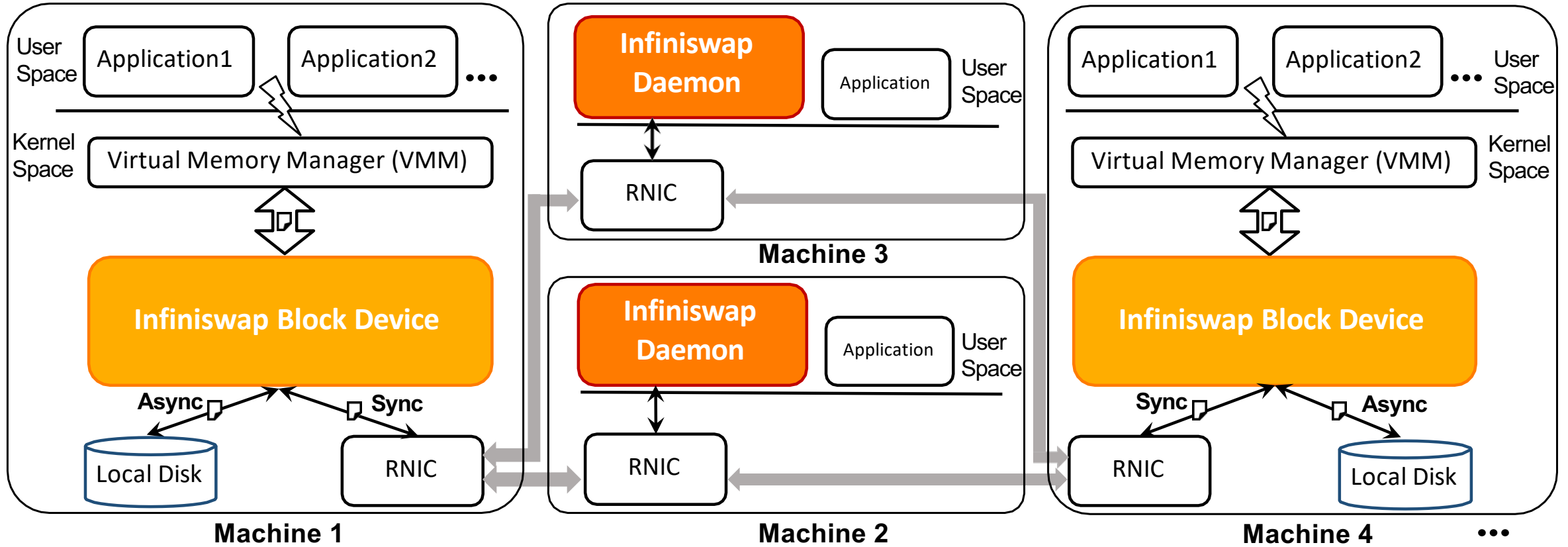
How to meet the design objectives?

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

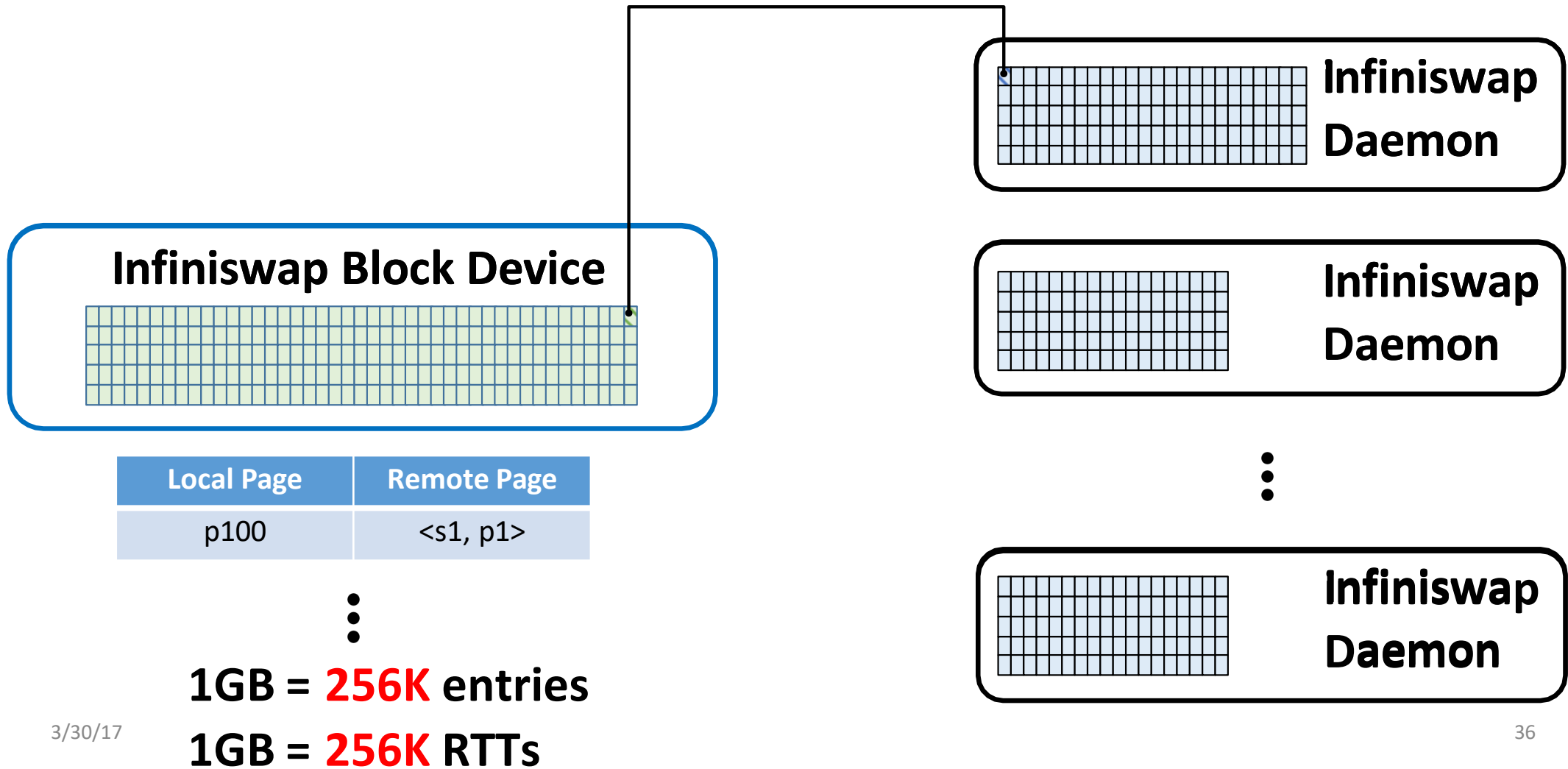
Ideas

Remote paging

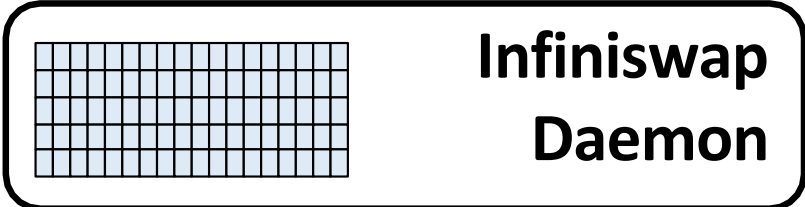
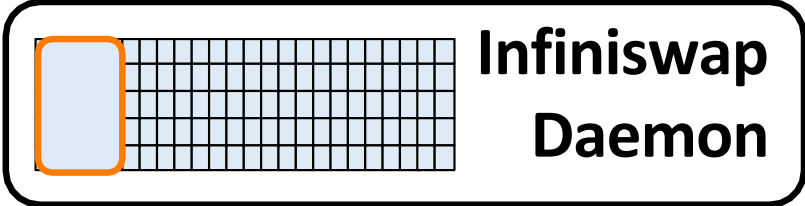
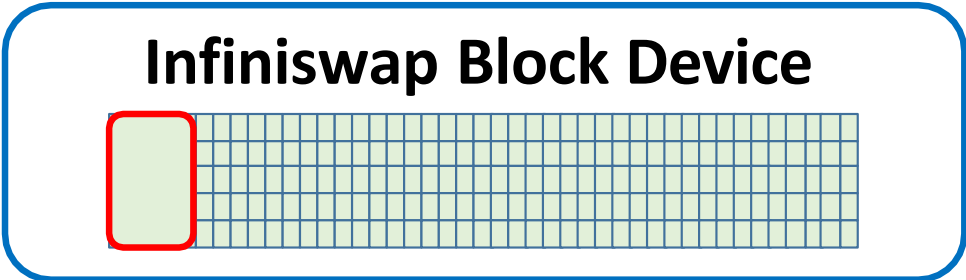
Local backup disk

**Decentralized remote memory
management**

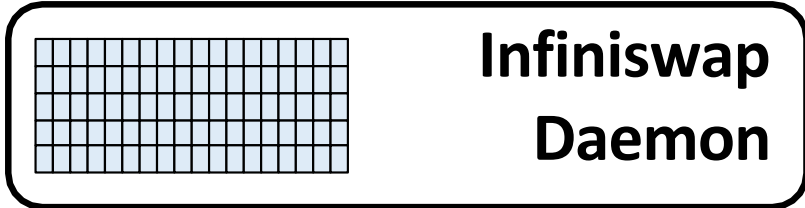
Management unit: memory page?



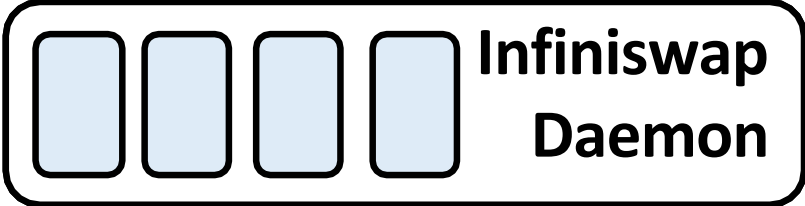
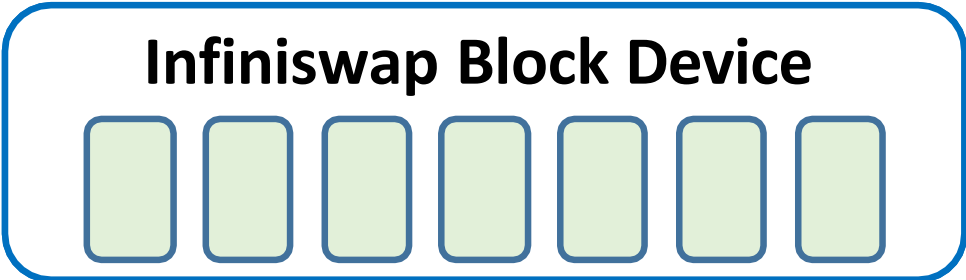
Management unit: memory slab!



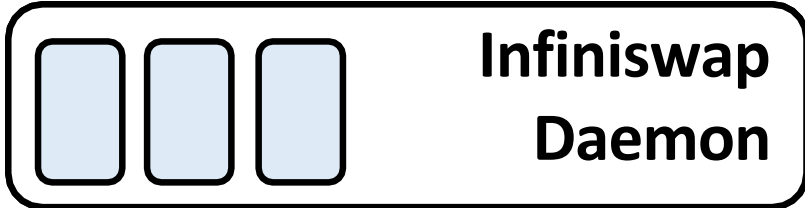
⋮



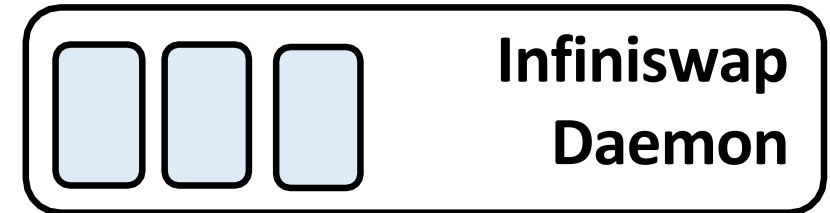
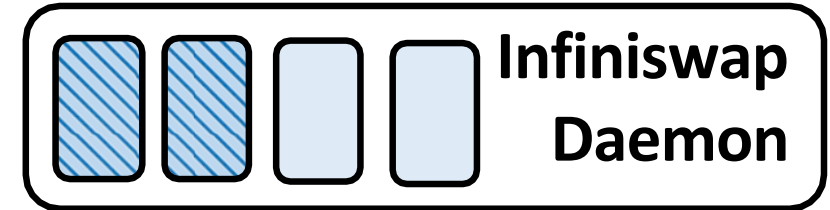
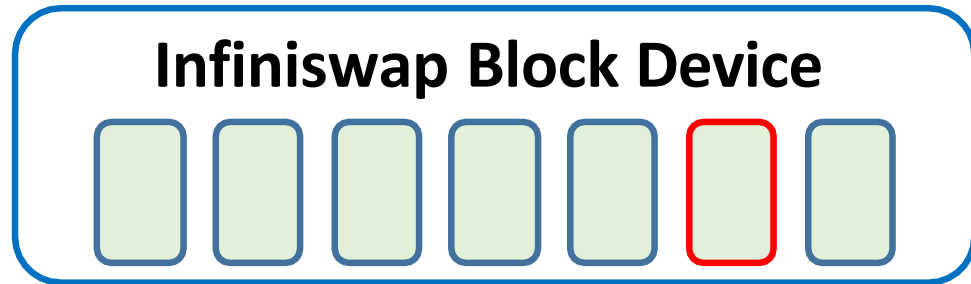
Management unit: memory slab!



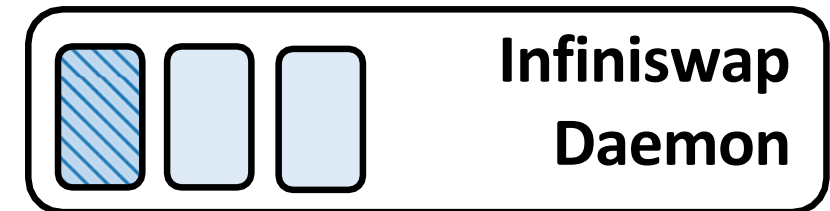
⋮



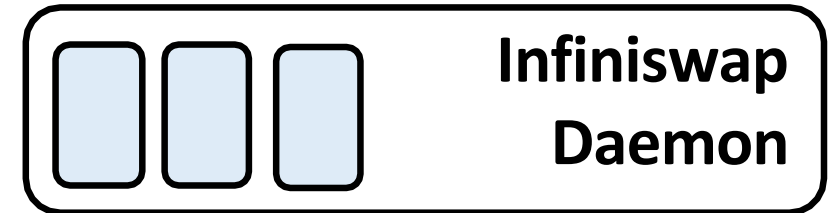
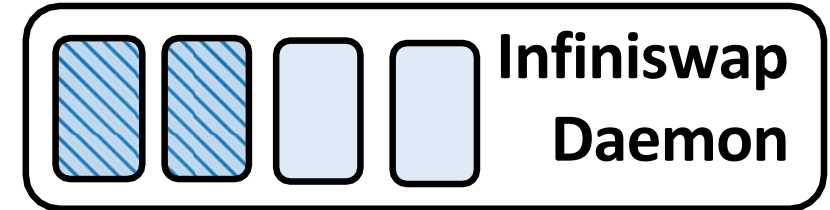
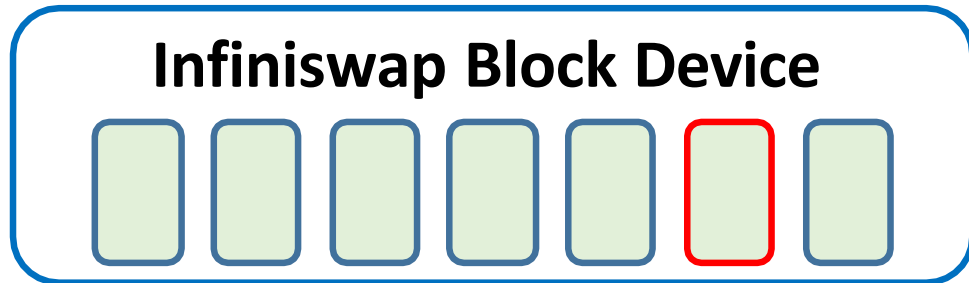
Which remote machine should be selected?



⋮



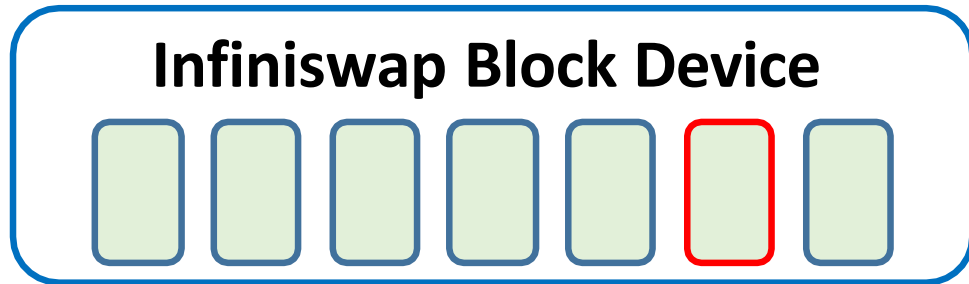
Which remote machine should be selected?



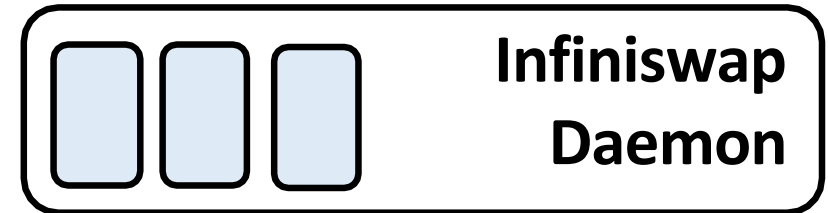
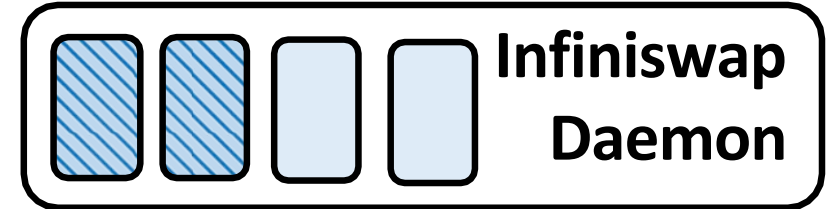
⋮

Goal: **balance** memory utilization

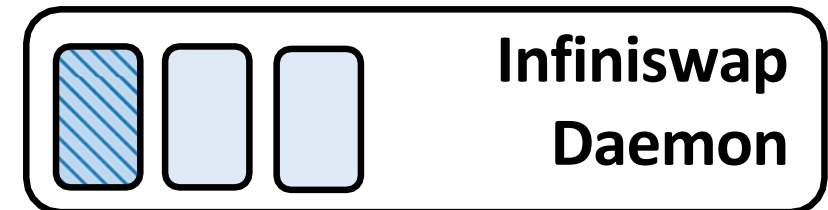
Which remote machine should be selected?



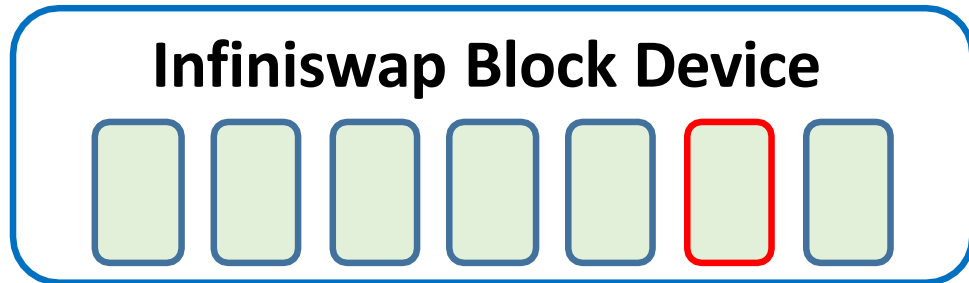
► **Central controller**



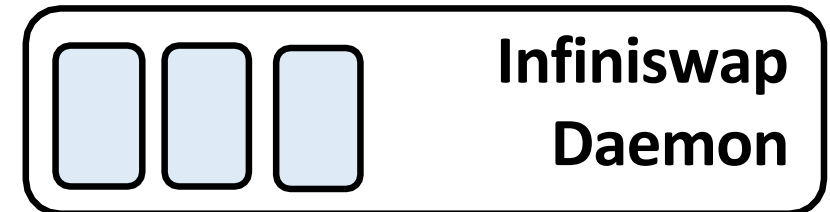
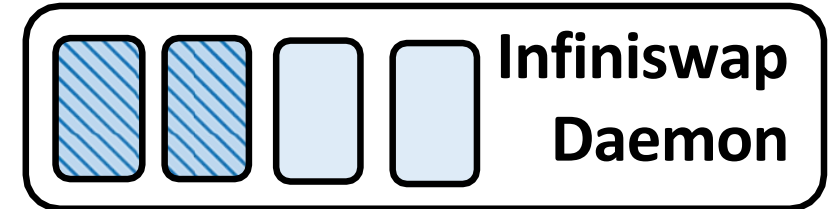
⋮



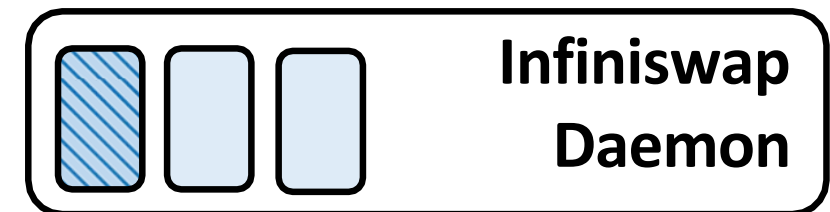
Which remote machine should be selected?



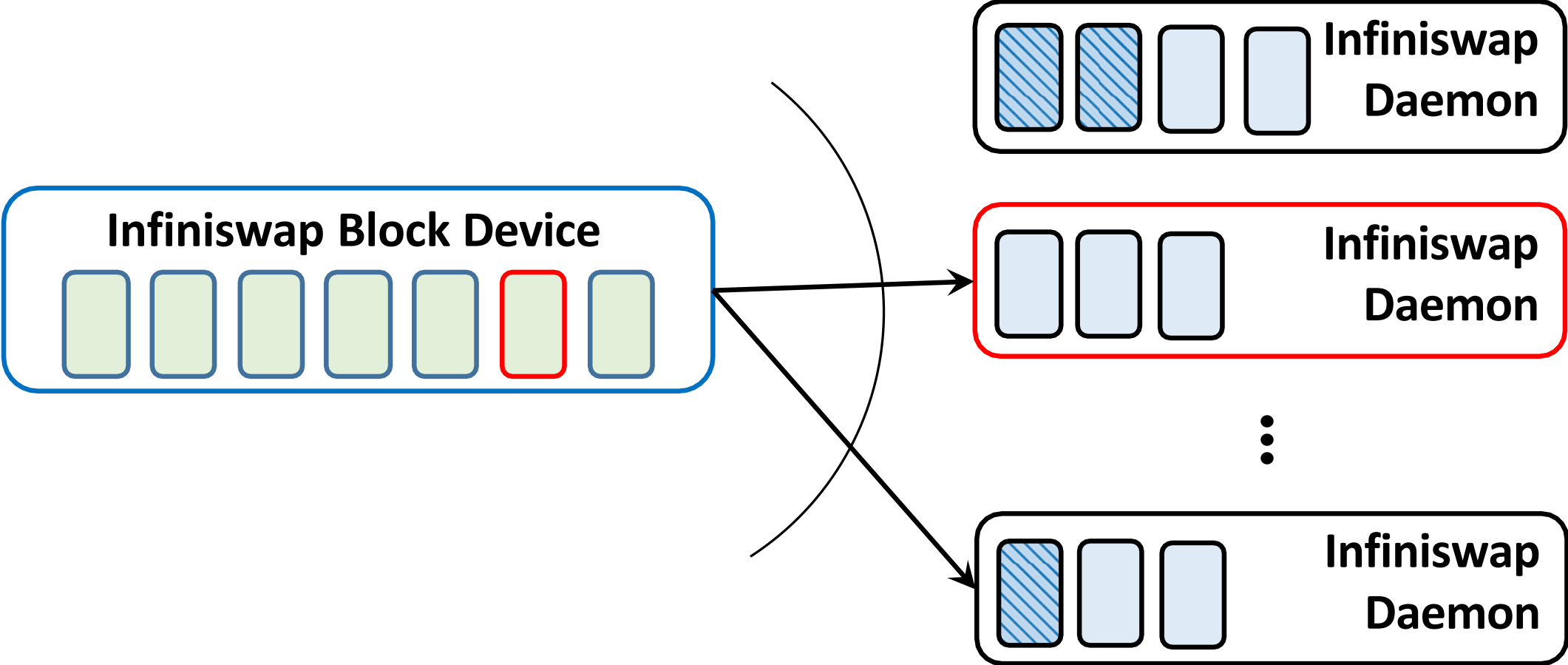
- ~~▶ Central controller~~
- ▶ Decentralized approach



⋮

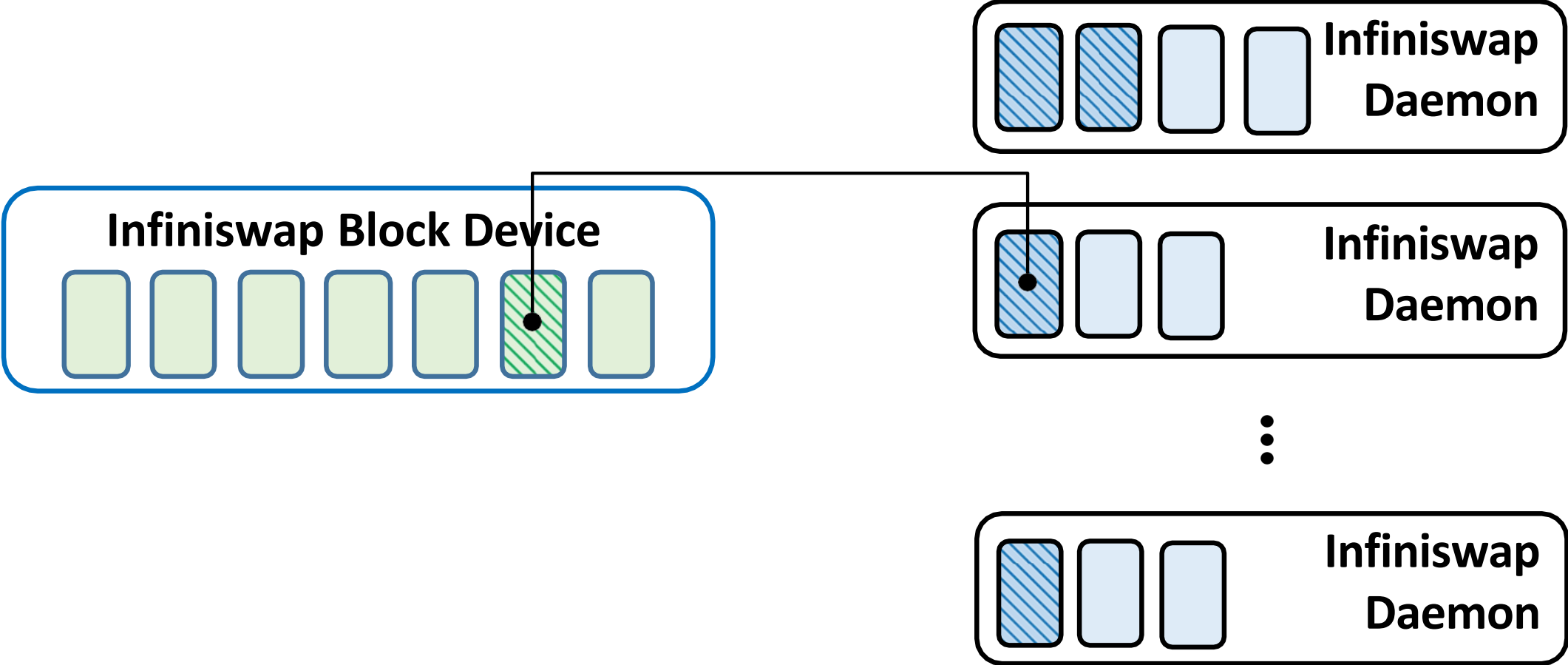


Power of two choices^[1]



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

Power of two choices^[1]



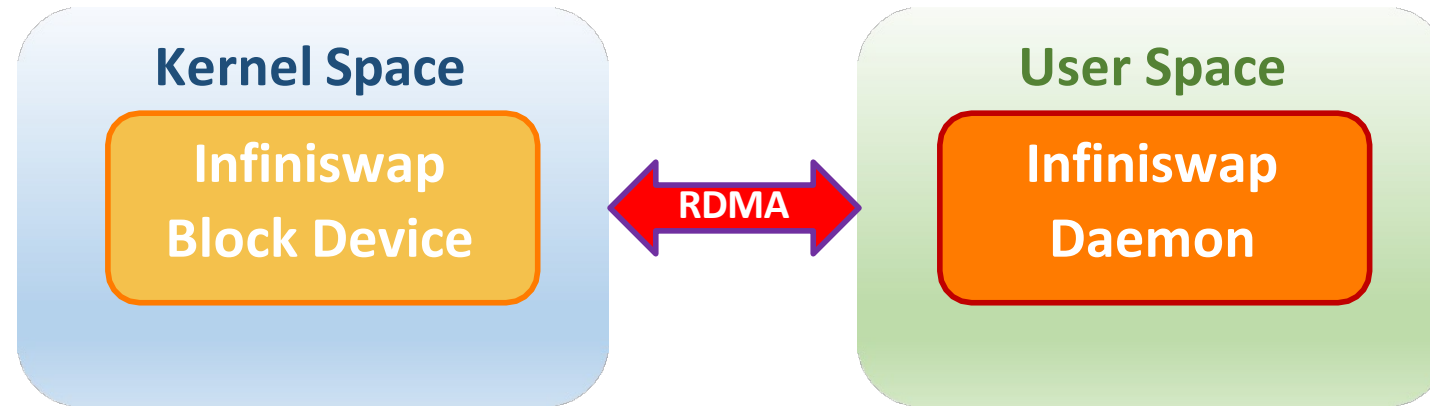
[1] Mitzenmacher, Michael. "The power of two choices in randomized load balancing.", Ph.D. thesis, U.C. Berkeley, 1996

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

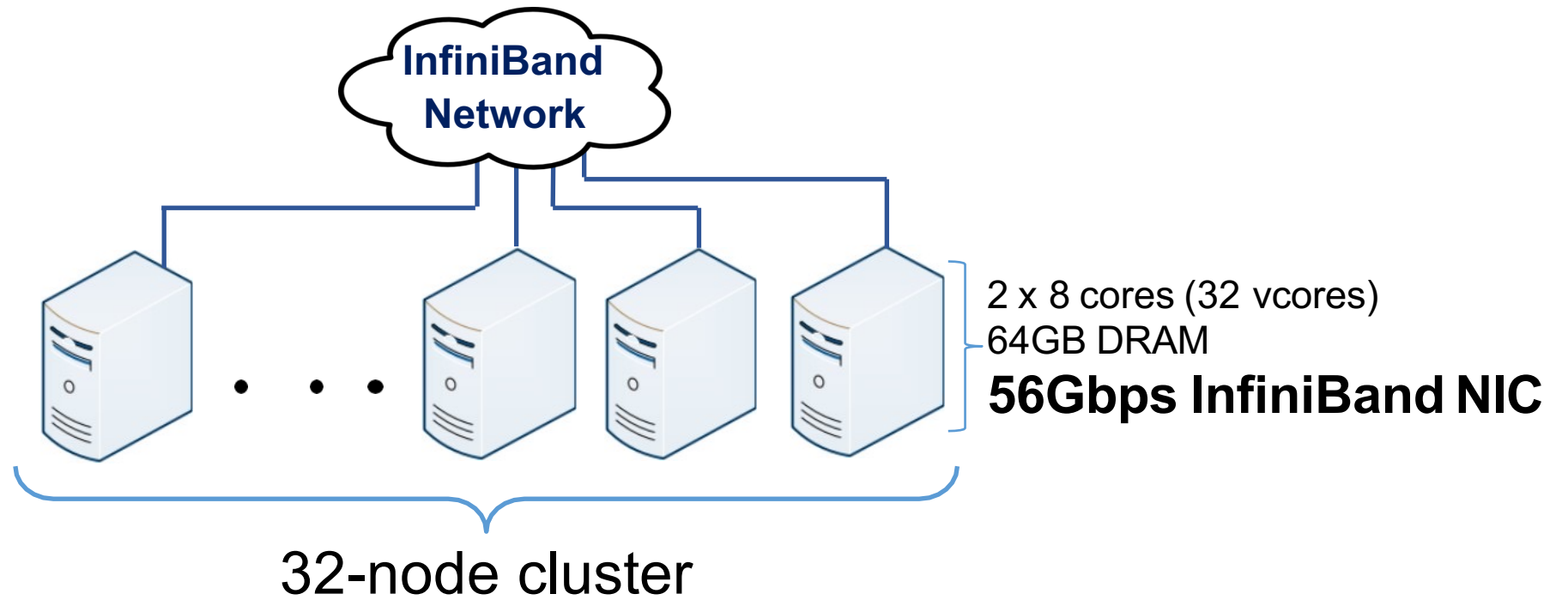
3/30/17

What are we expecting from Infiniswap?

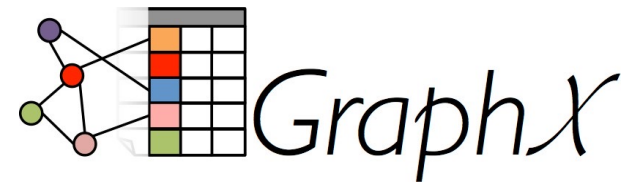
- **Application performance**
- **Cluster memory utilization**
- Network usage
- Eviction overhead
- Fault-tolerance overhead
- Performance as a block device

⋮

Evaluation

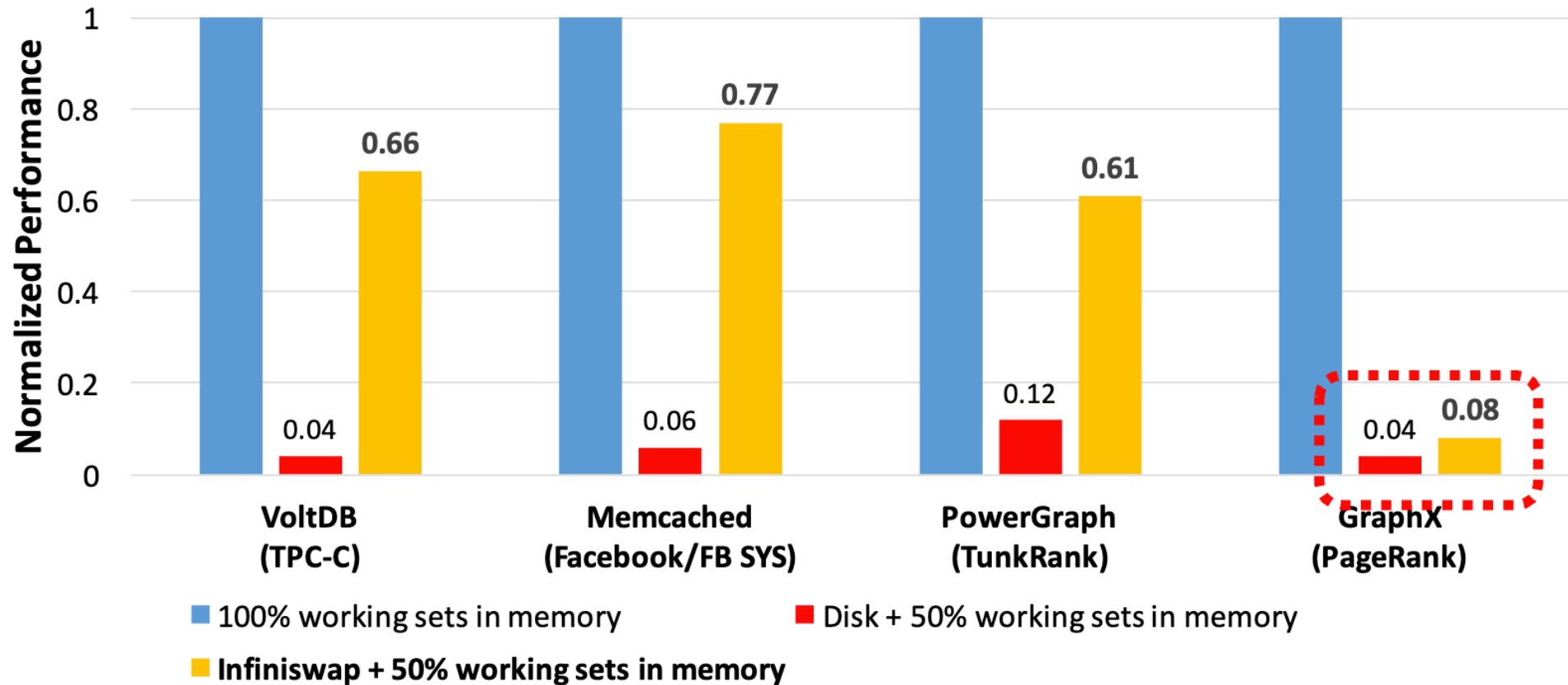


3/30/17



Application performance

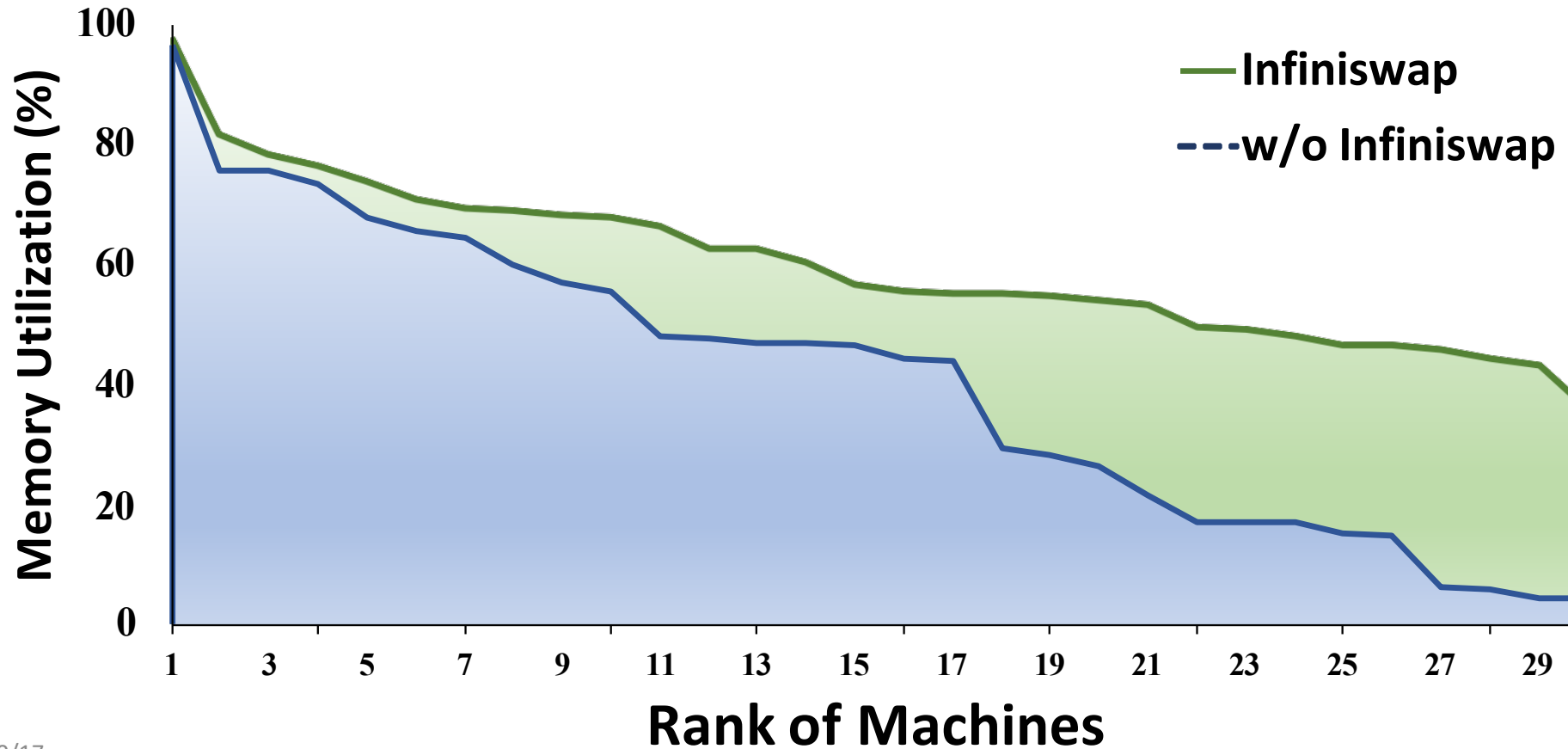
- 50% working sets in memory



- 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
 - SOSp'23 vLLM
- Demand paging: remote memory over network
 - NSDI'17 InfiniSwap
 - OSDI'20 AIFM
- Demand paging: memory swapping between GPU memory and host memory
 - OSDI'20 PipeSwitch
 - NSDI'23 TGS

AIFM: High-Performance, Application-Integrated Far Memory

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

*MIT CSAIL

†Brown University

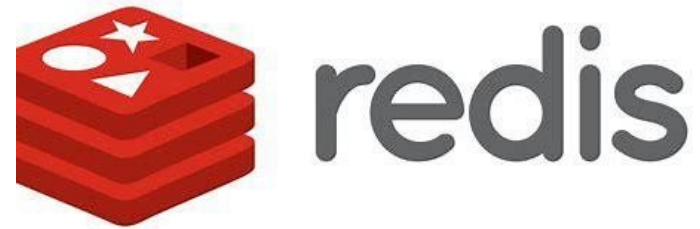
‡VMware Research



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



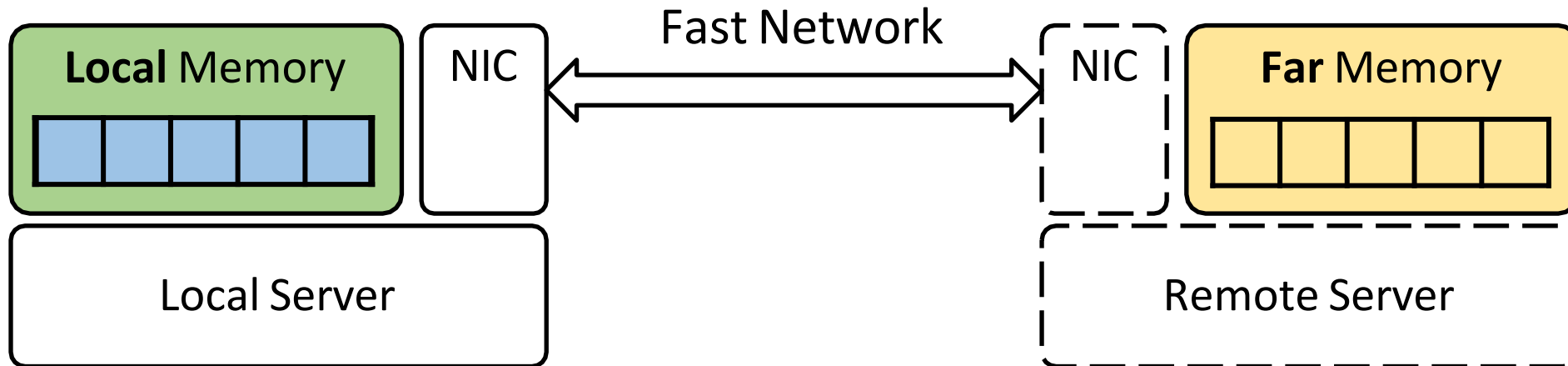
20

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.

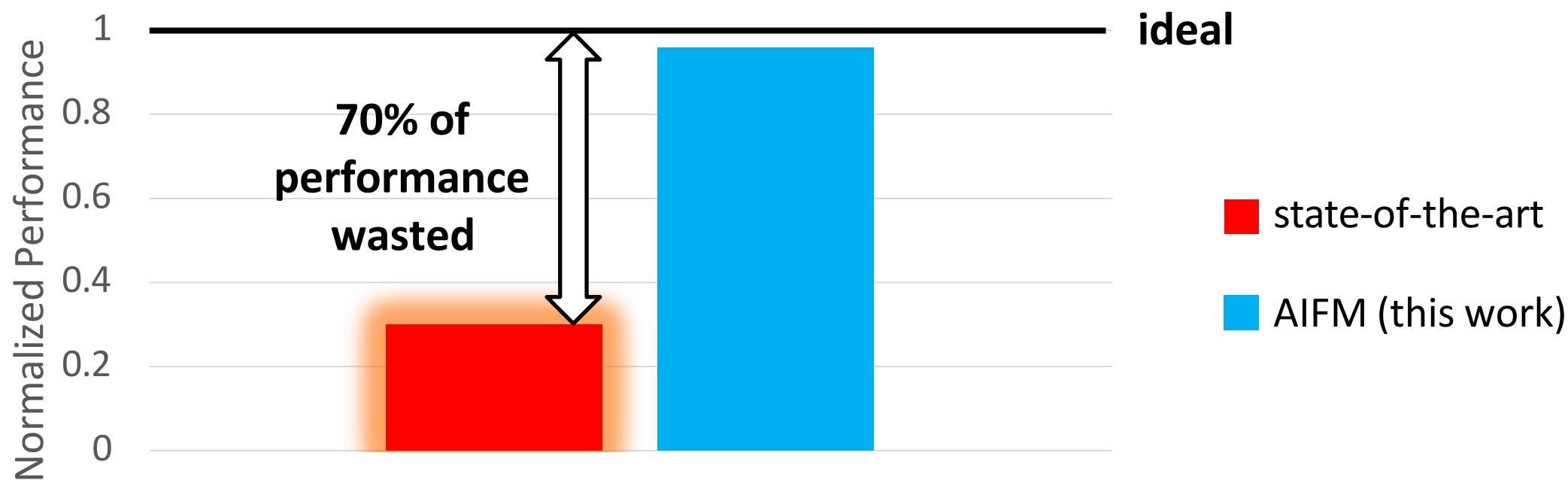
Trending Solution: Far Memory

- 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.

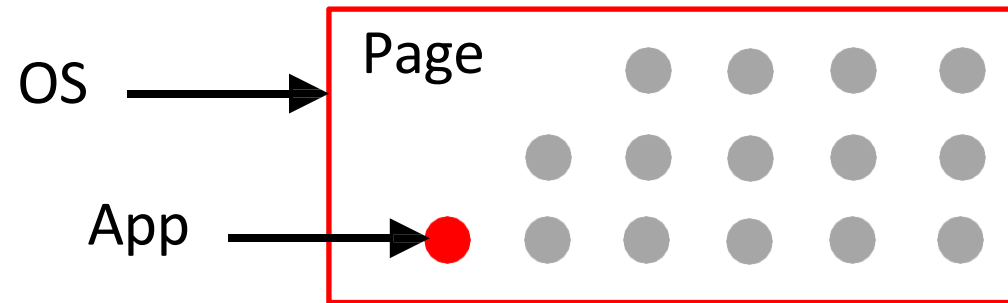


Why Do Existing Systems Waste Performance?

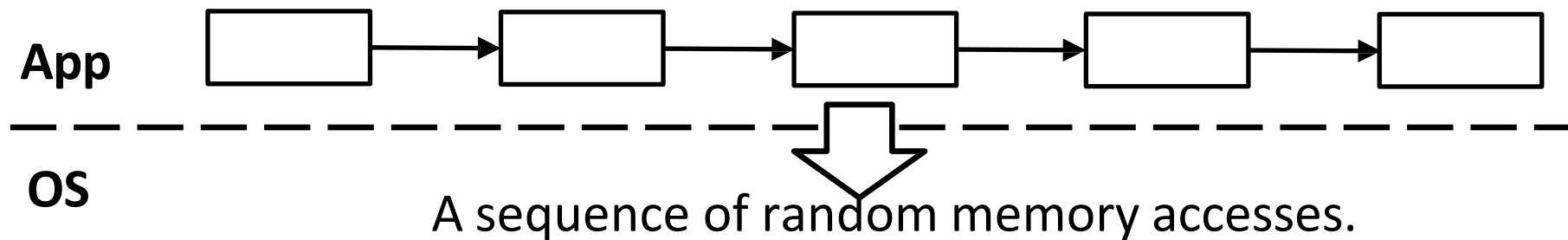
- Problem: based on **OS paging**.
 - Semantic gap.
 - High kernel overheads.

Challenge 1: Semantic Gap

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

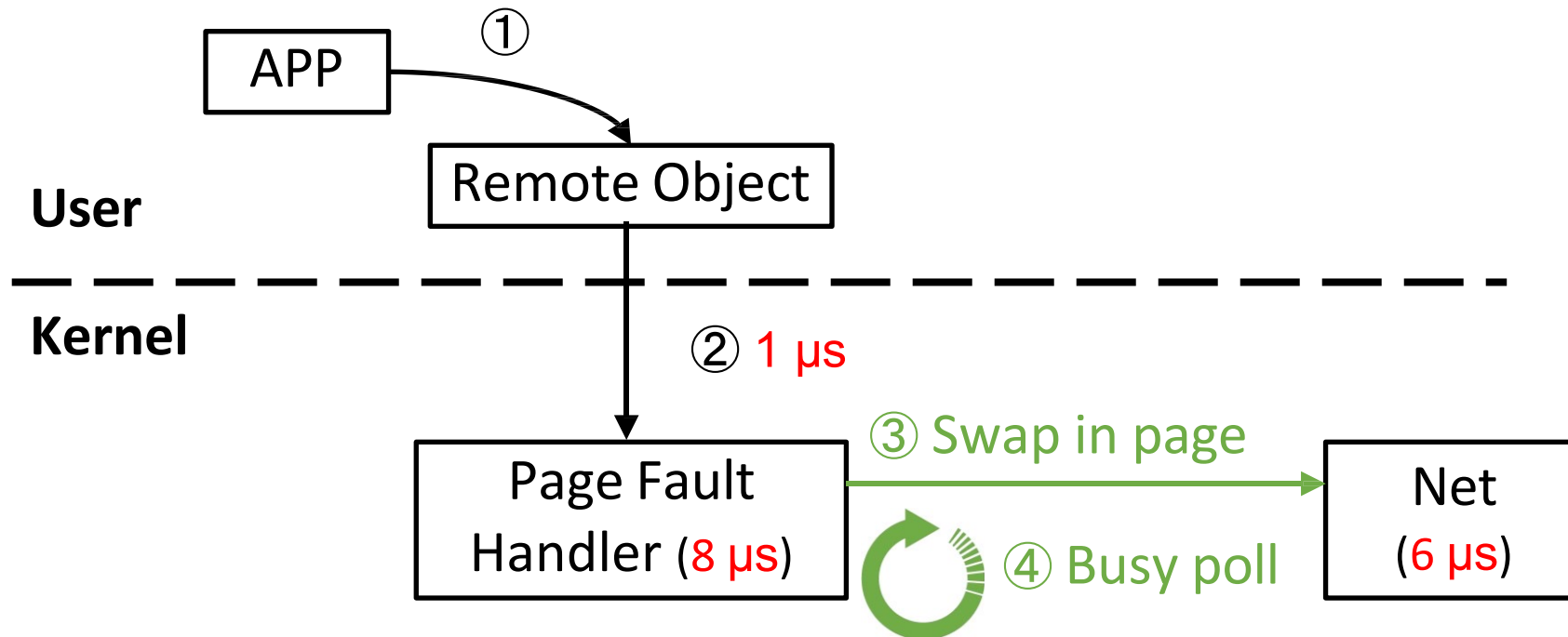


- OS lacks app knowledge → **hard to prefetch, etc.**

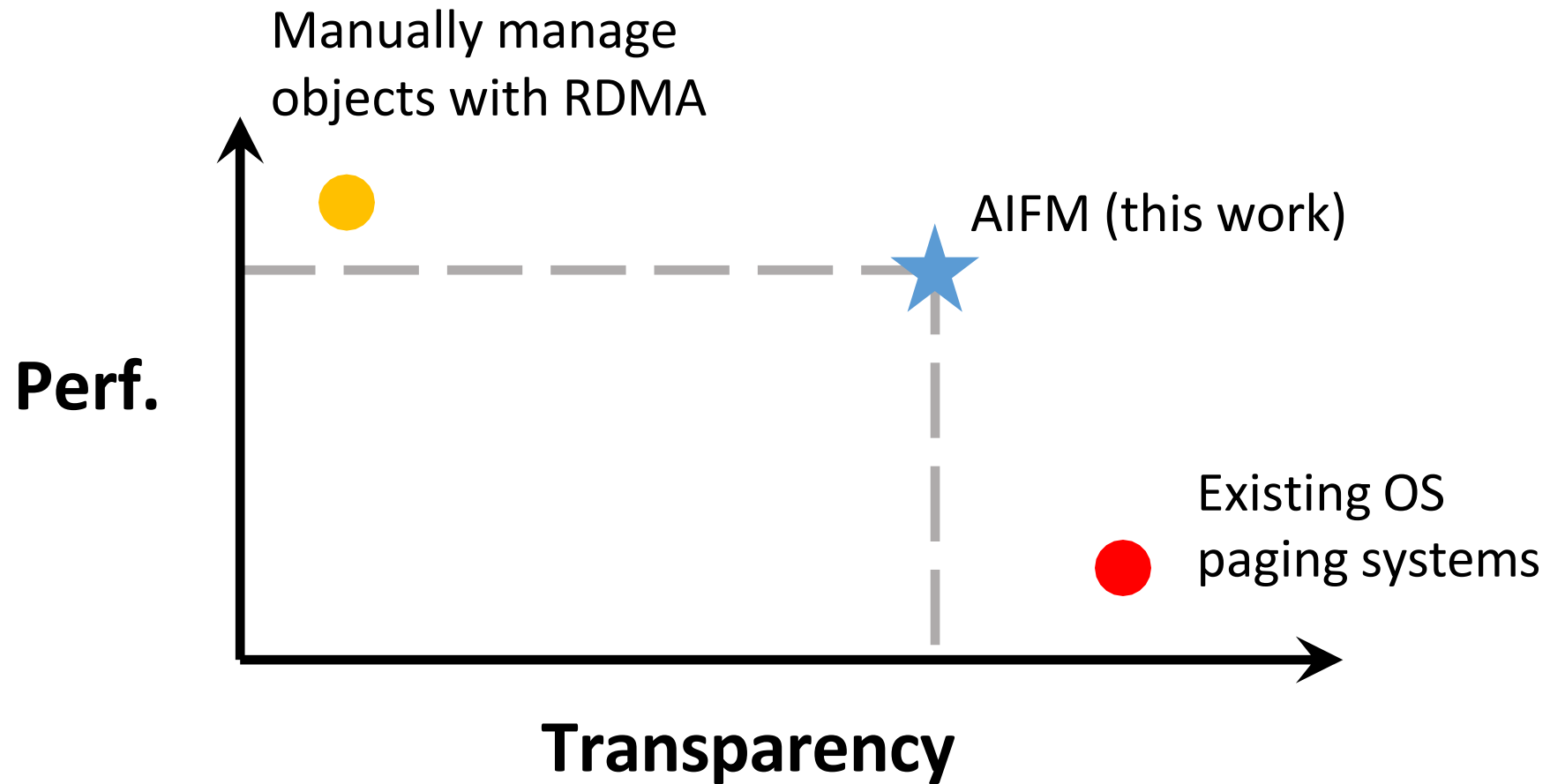


Challenge 2: High Kernel Overheads

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



Design Space



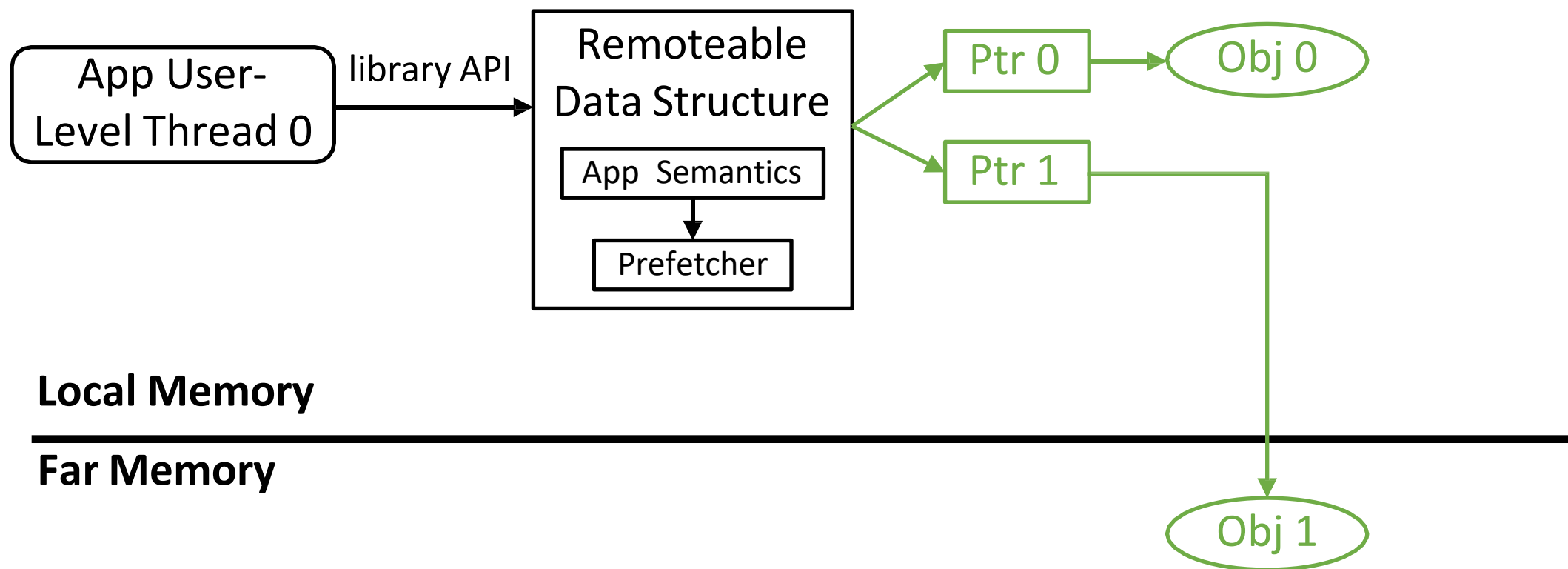
AIFM's Design Overview

➤ Key idea: swap memory using a userspace runtime.

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

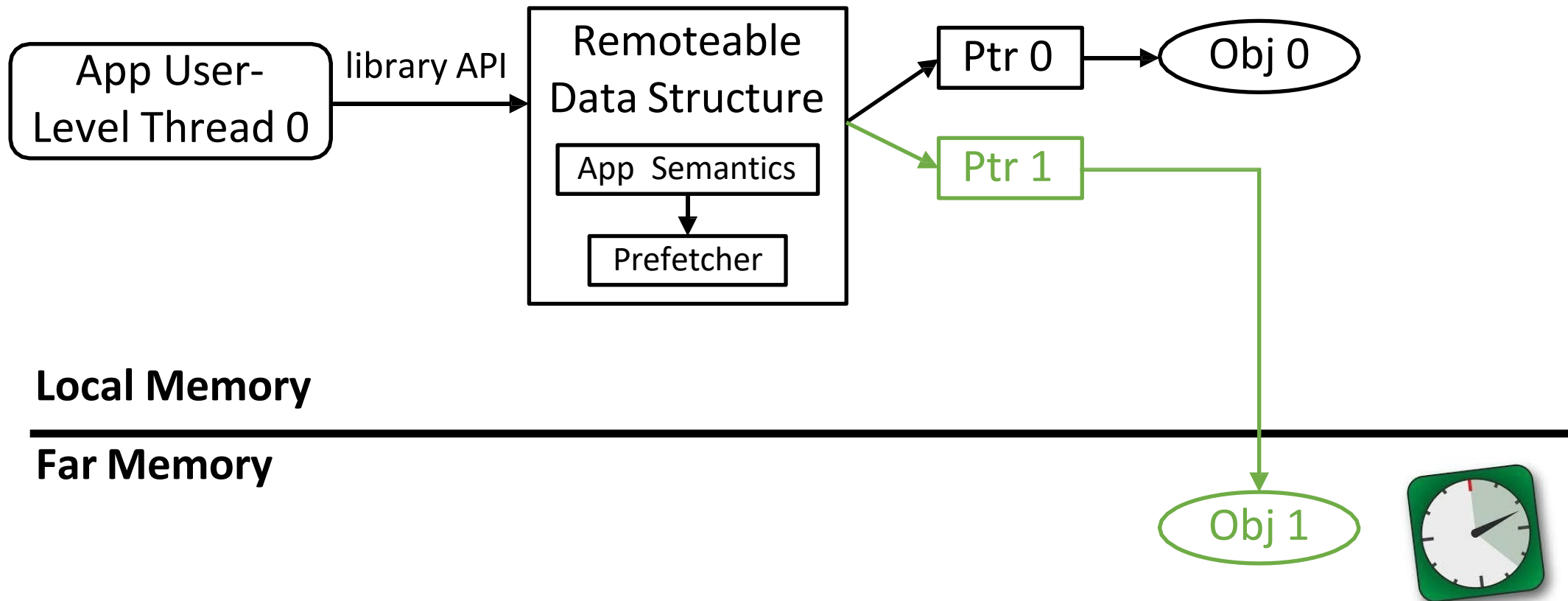
1. Remoteable Data Structure Library

➤ Solved challenge: semantic gap.



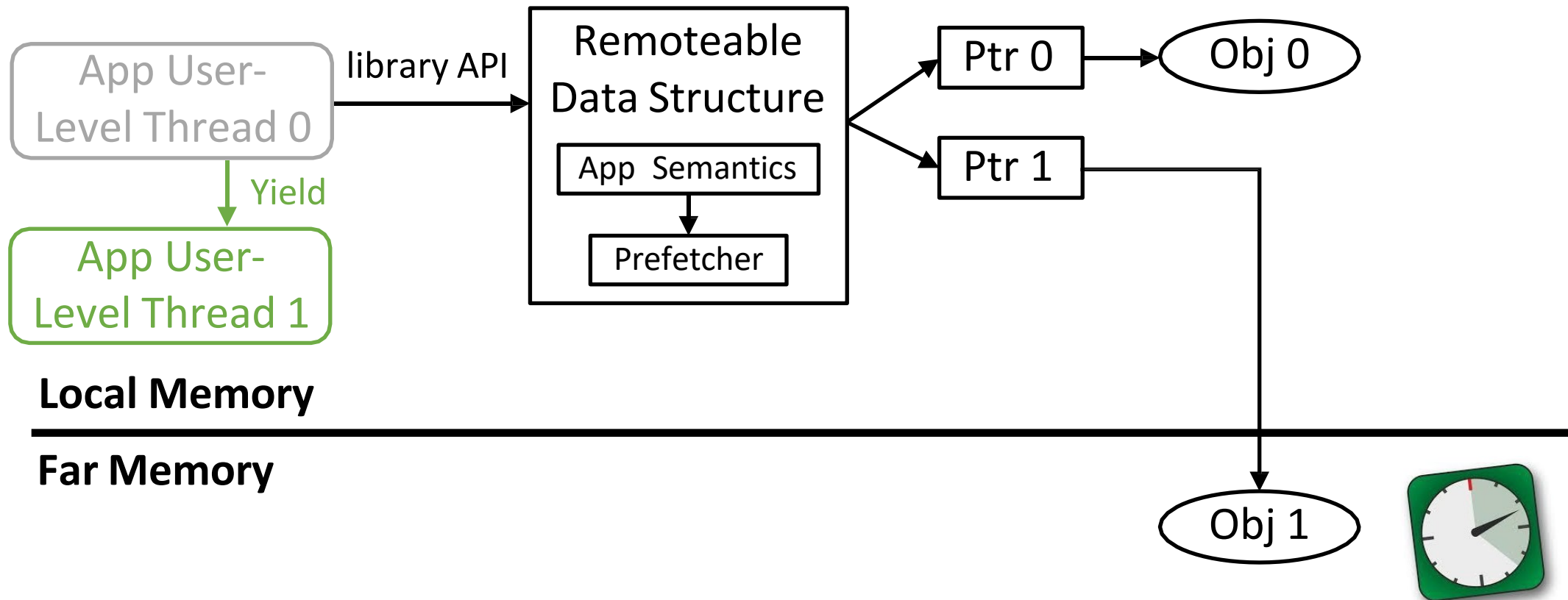
2. Userspace Runtime

➤ Solved challenge: kernel overheads.



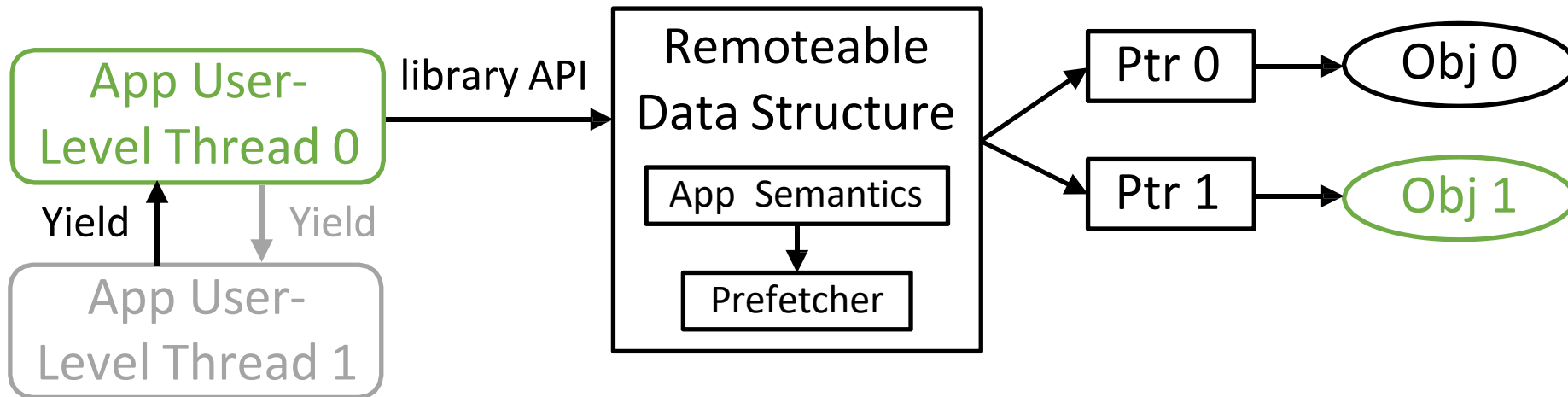
2. Userspace Runtime

➤ Solved challenge: kernel overheads.



2. Userspace Runtime

➤ Solved challenge: kernel overheads.

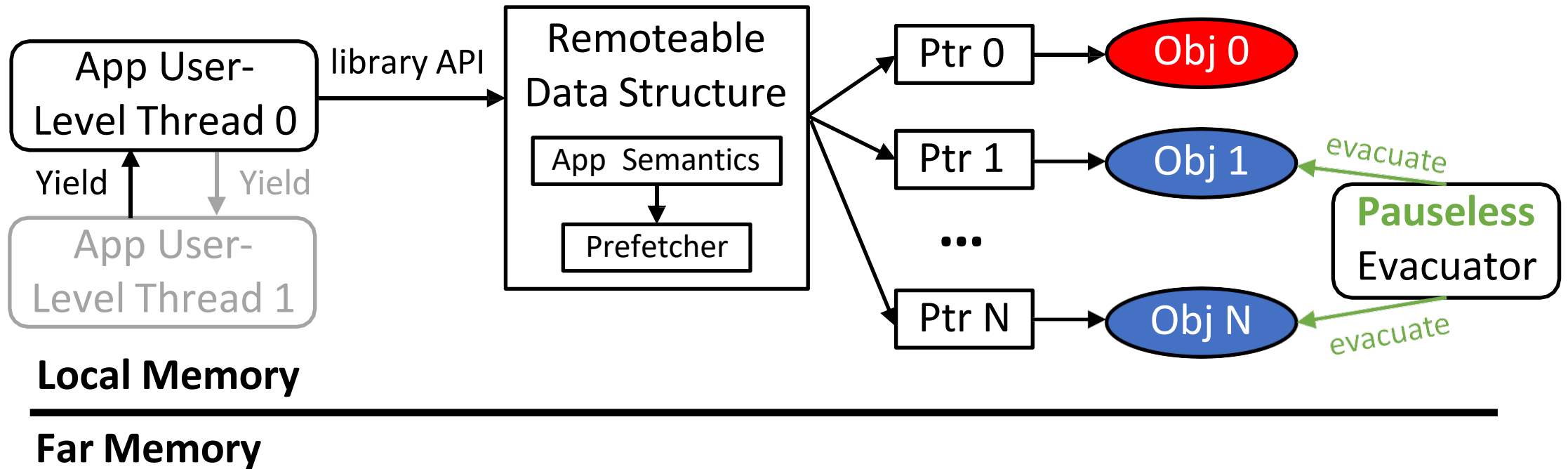


Local Memory

Far Memory

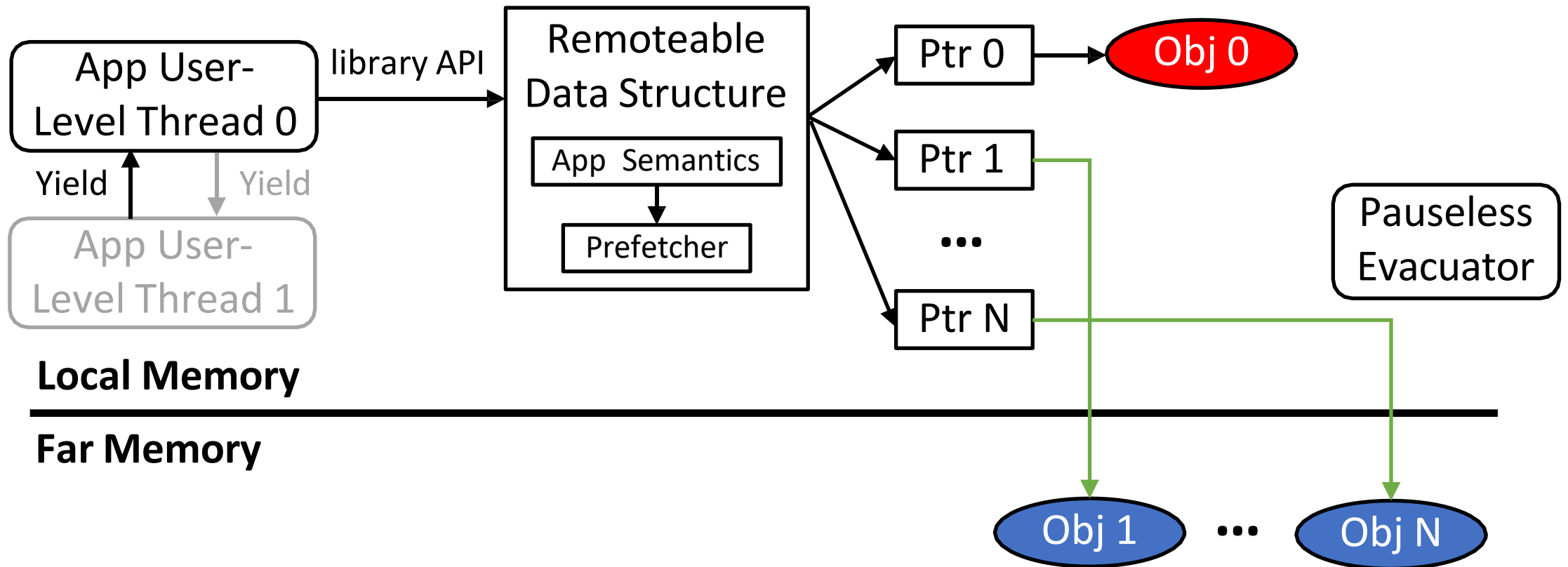
3. Pauseless Evacuator

➤ Solved challenge: impact of memory reclamation.



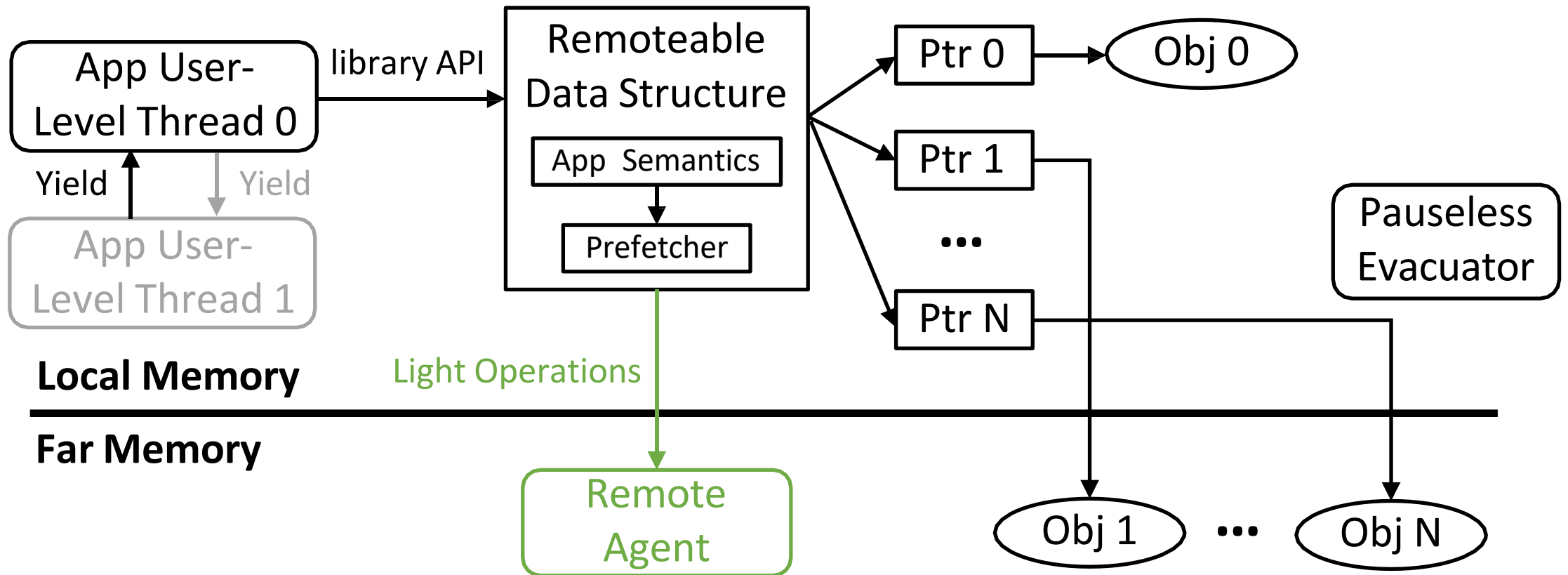
3. Pauseless Evacuator

➤ Solved challenge: impact of memory reclamation.



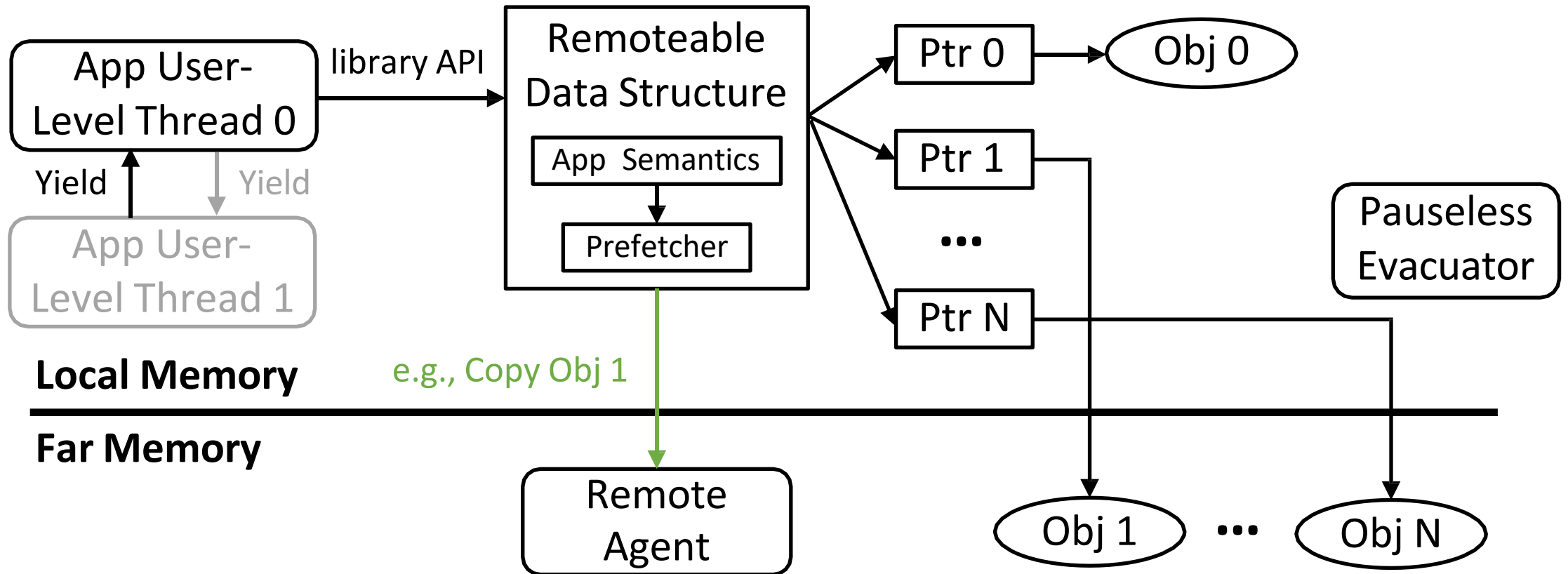
4. Remote Agent

➤ Solved challenge: network BW < DRAM BW.



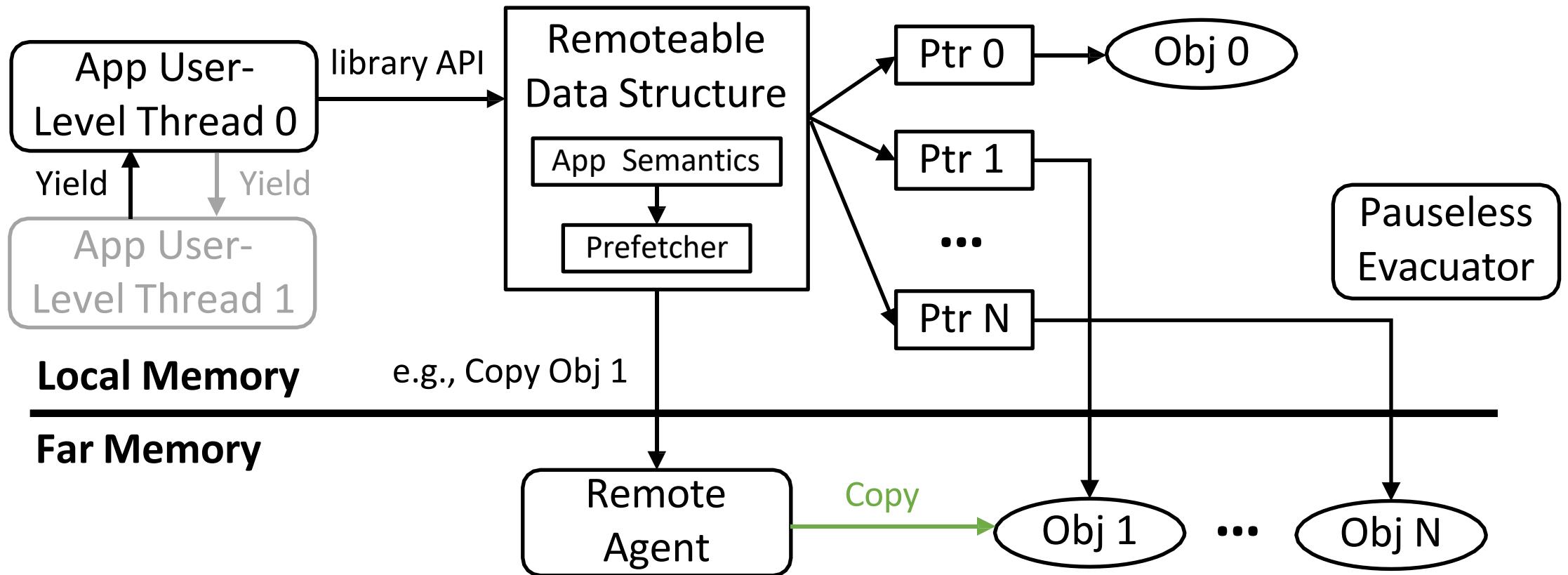
4. Remote Agent

➤ 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.

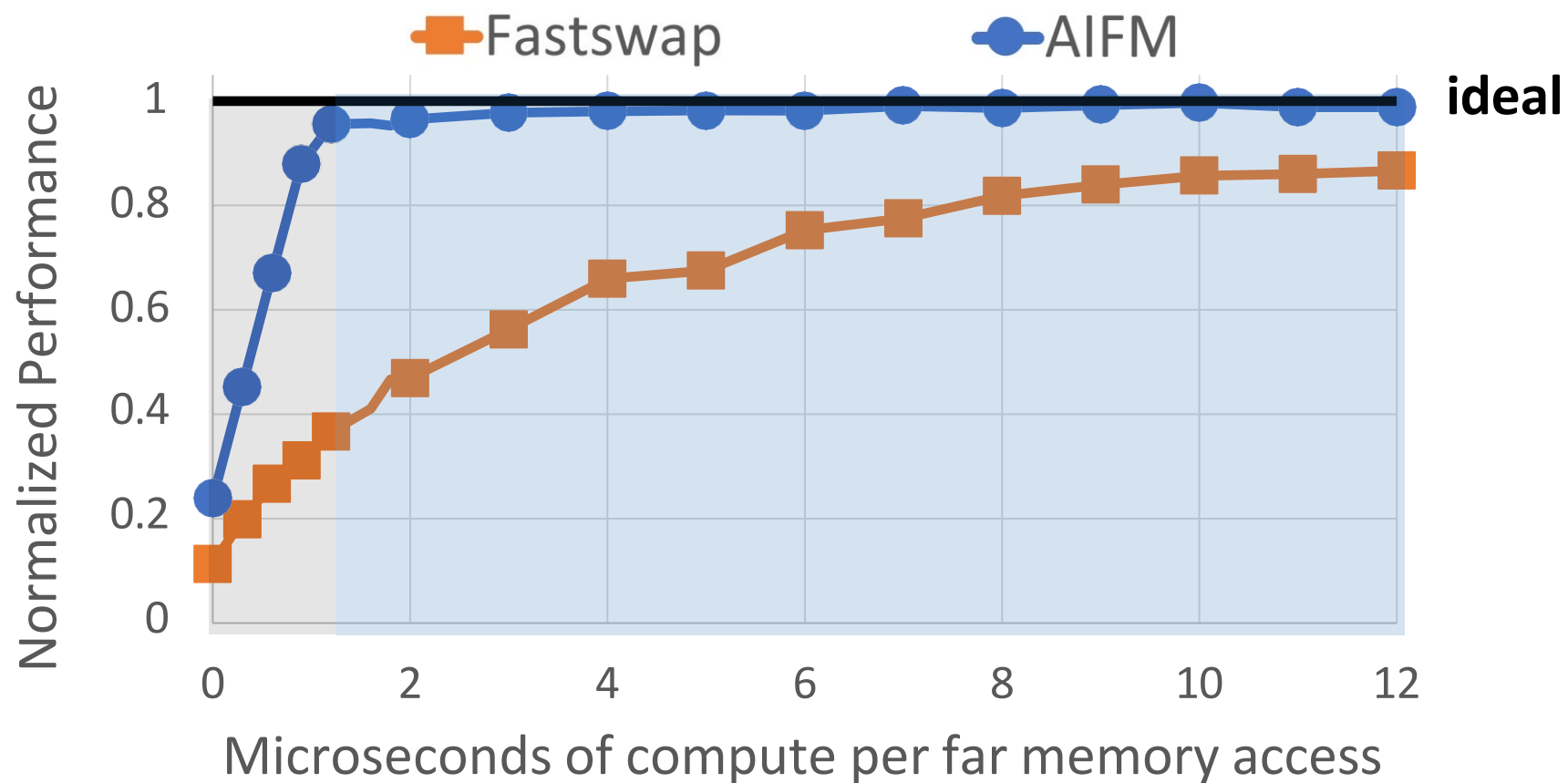
```
    return ret;
```

```
}
```

Implementation

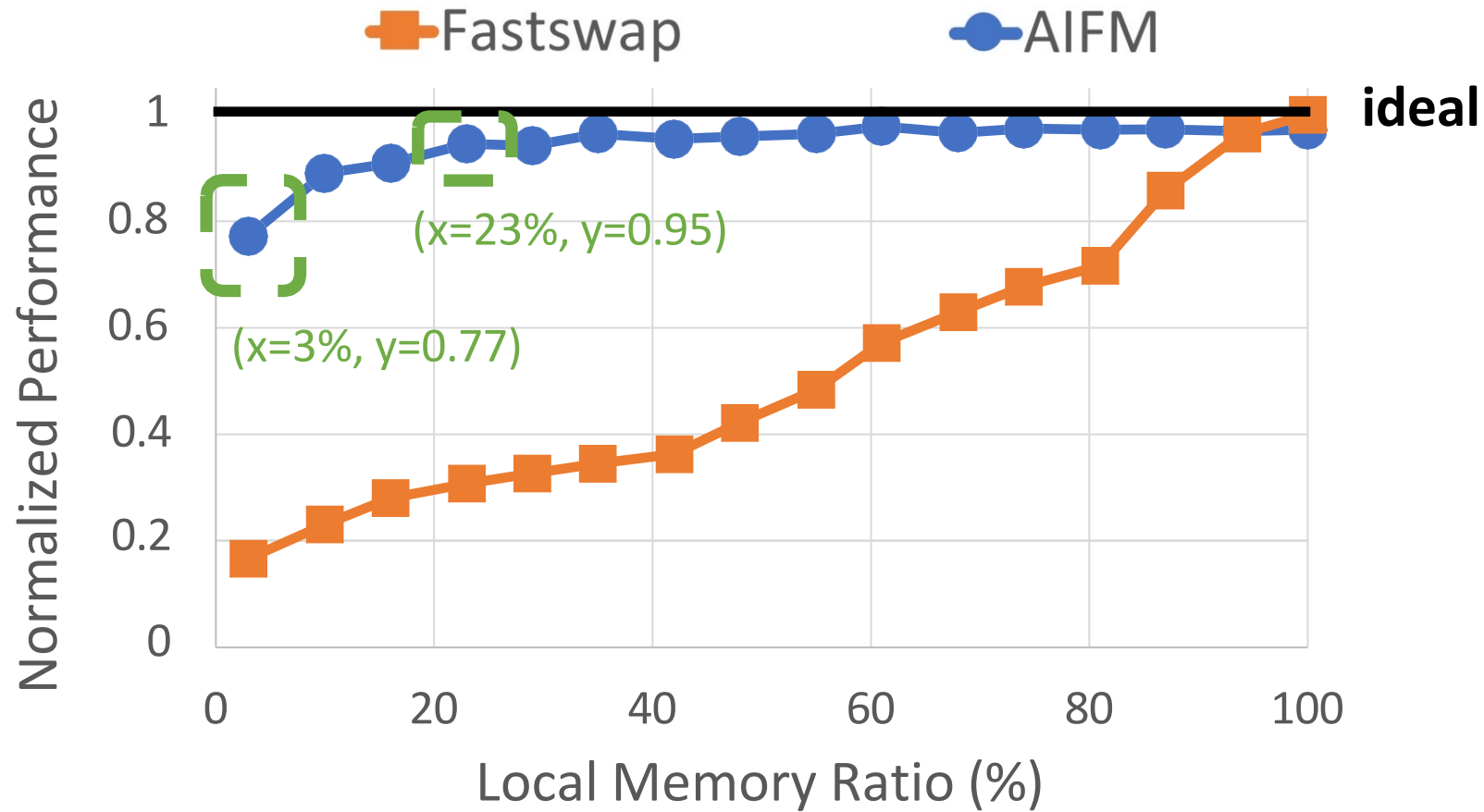
- Implemented 6 data structures.
 - Array, List, Hashtable, Vector, Stack, and Queue.
- Runtime is built on top of Shenango [NSDI' 19].
- TCP far-memory backend.
- LoC: 6.5K (runtime) + 5.5K (data structures) + 0.8K (Shenango)

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

Memory Management in Modern Computer Systems

- Memory Abstraction
 - NSDI'14 FaRM
 - SOSp'23 vLLM
- Demand paging: remote memory over network
 - NSDI'17 InfiniSwap
 - OSDI'20 AIFM
- Demand paging: memory swapping between GPU memory and host memory
 - OSDI'20 PipeSwitch
 - NSDI'23 TGS

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



Training

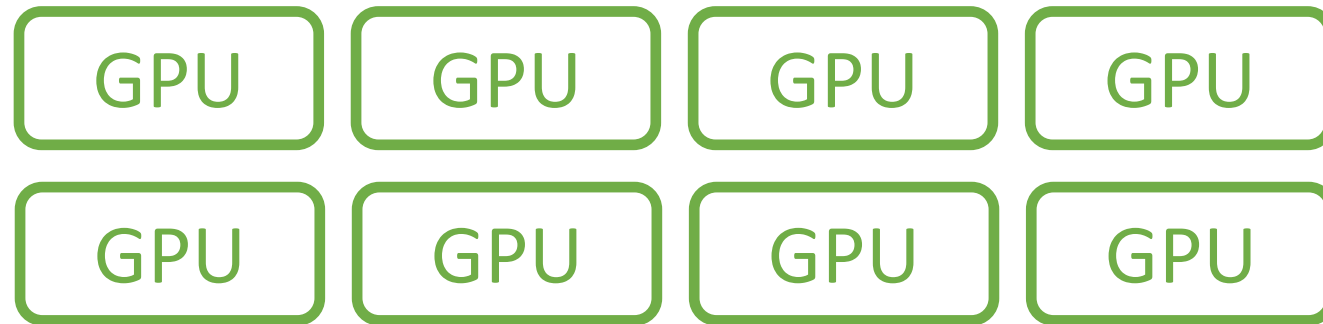
High throughput



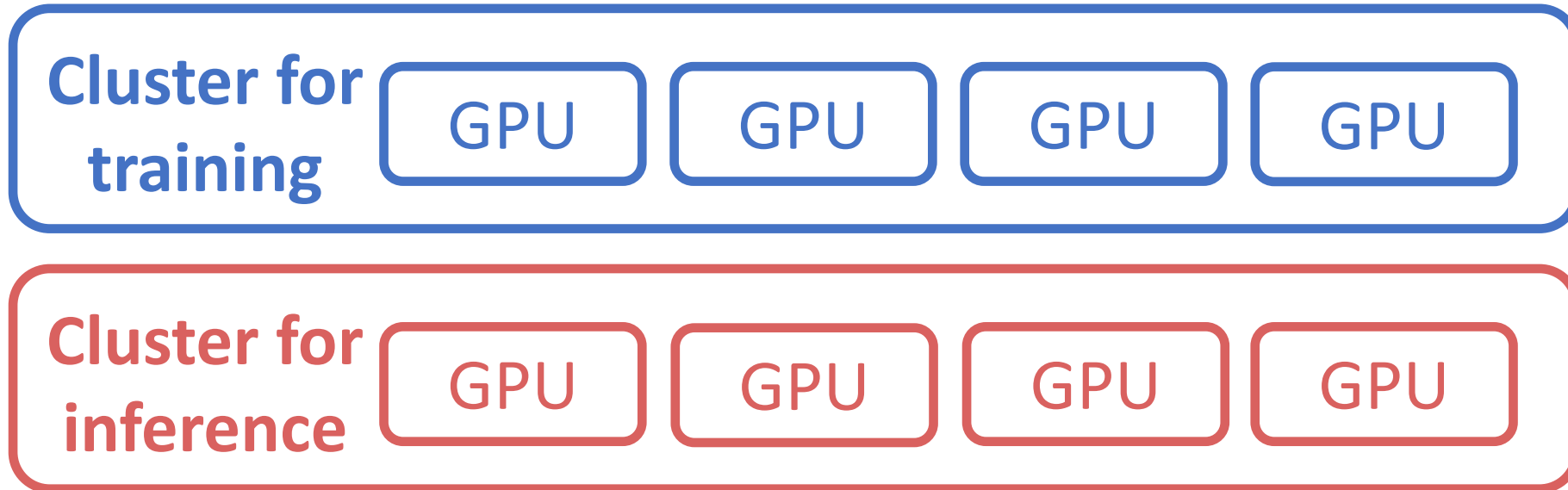
Inference

Low latency

GPUs clusters for DL workloads

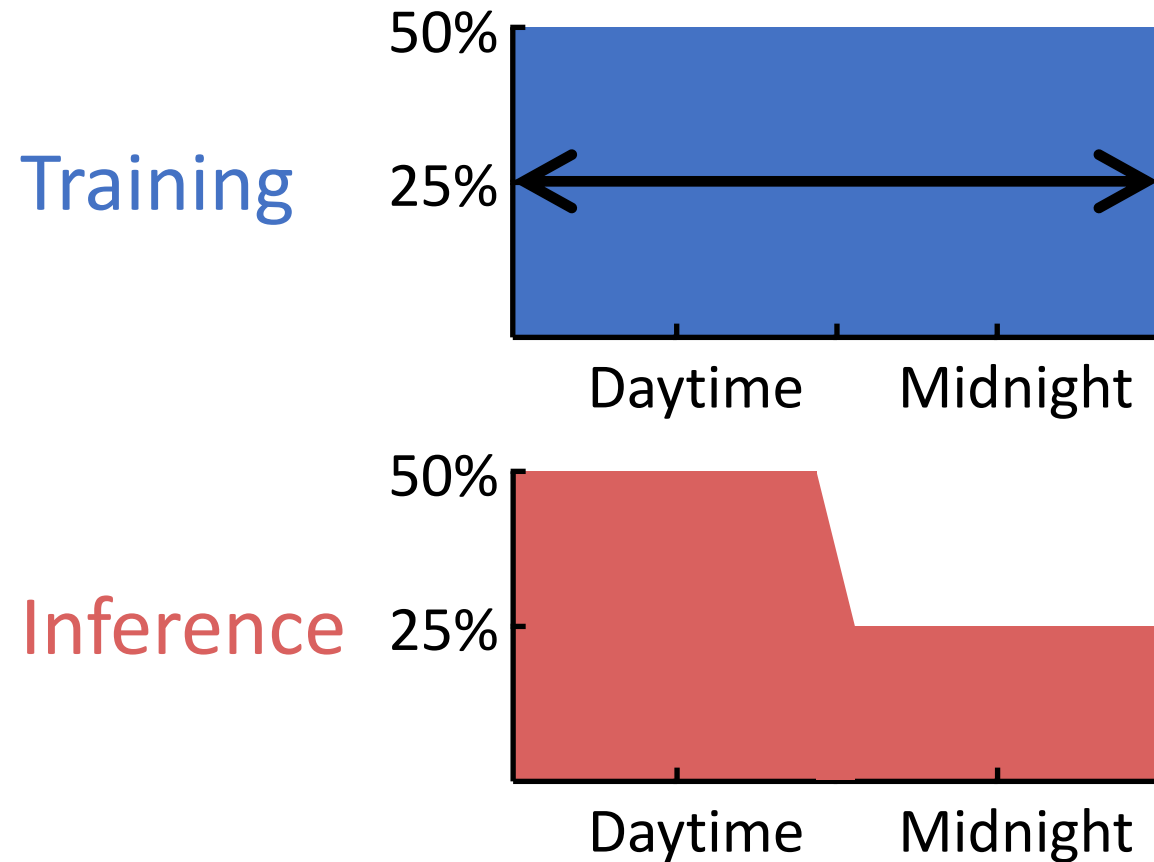


Separate clusters for training and inference

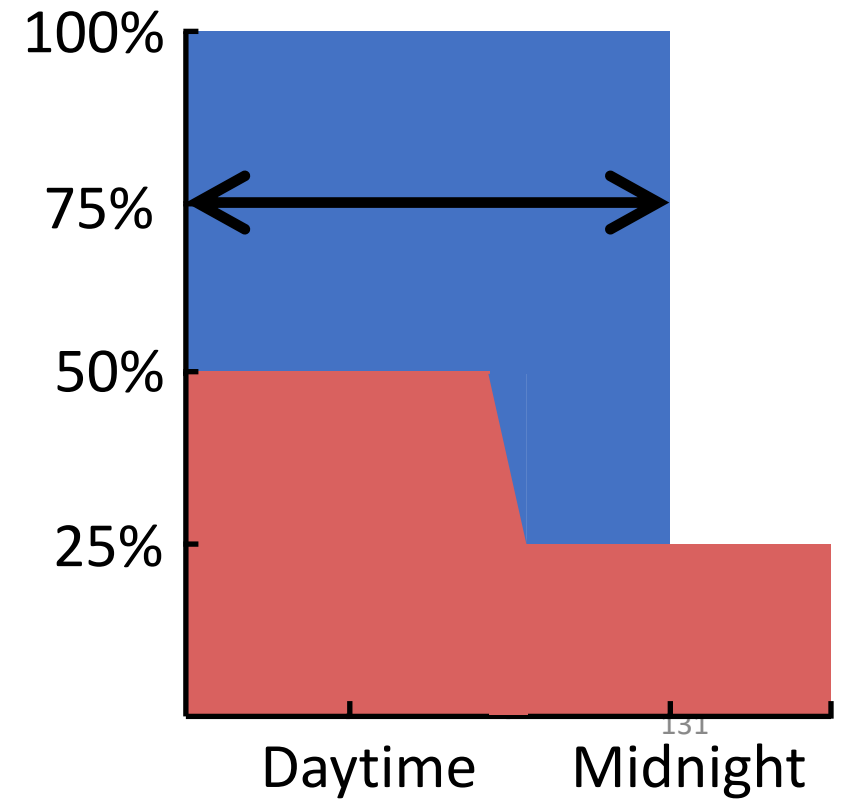


Utilization of GPU clusters is low

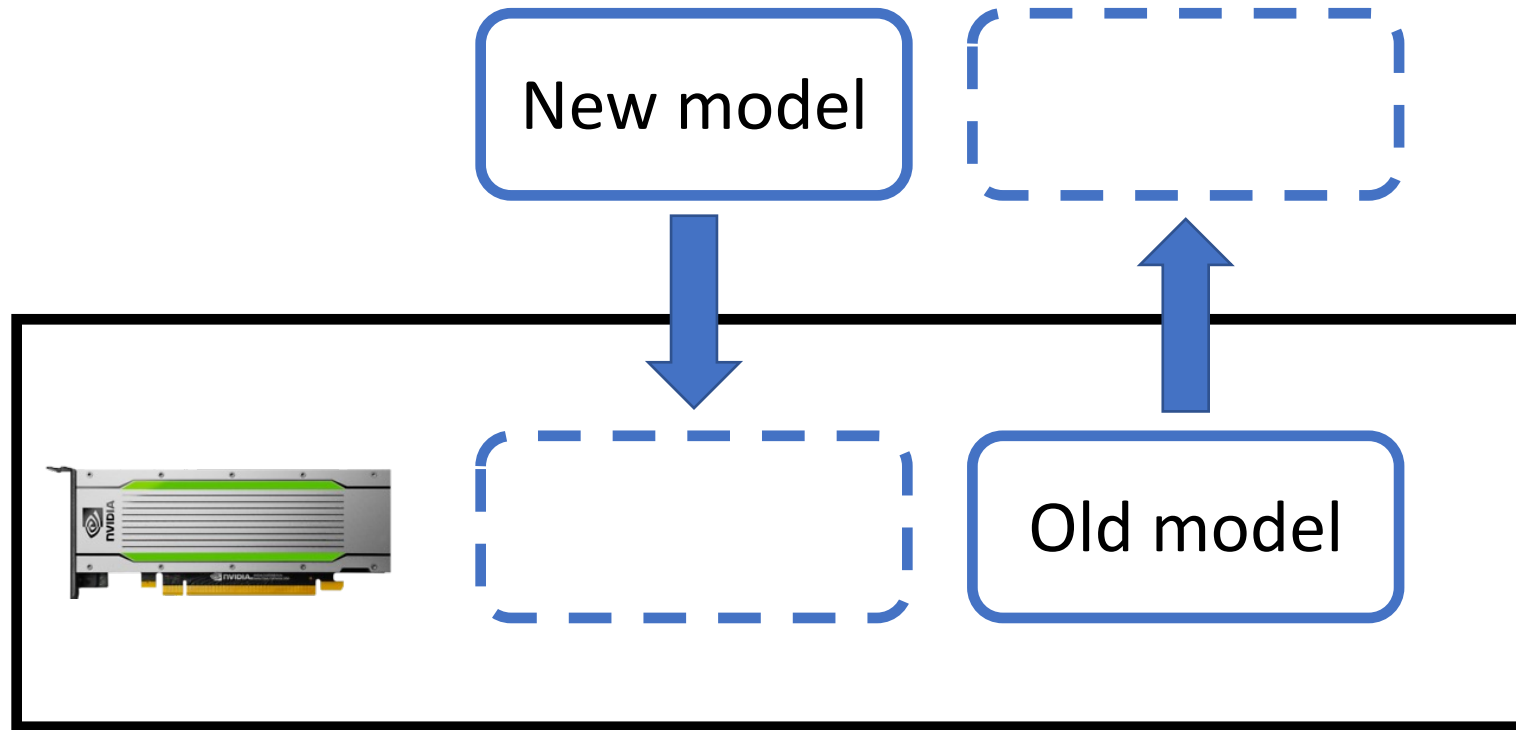
Today: separate clusters



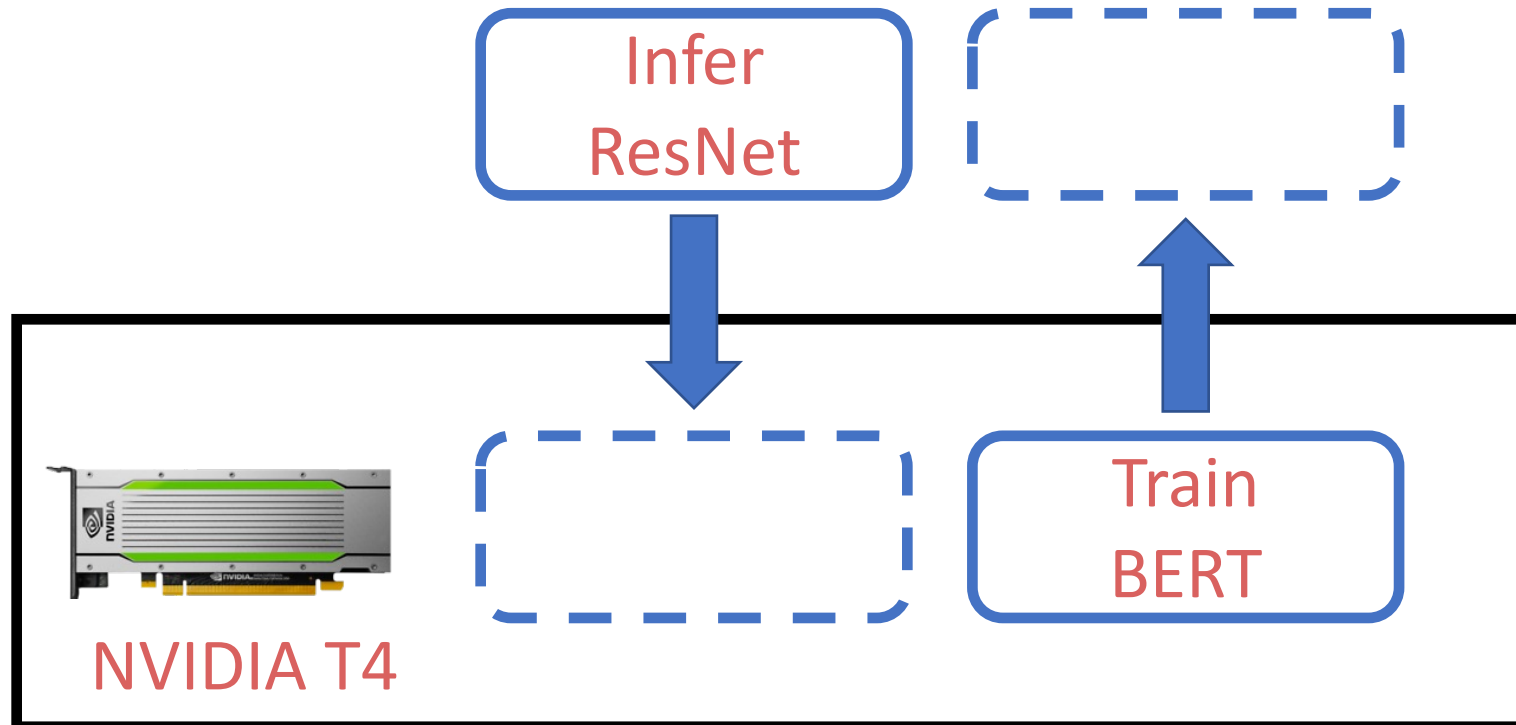
Ideal: shared clusters



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

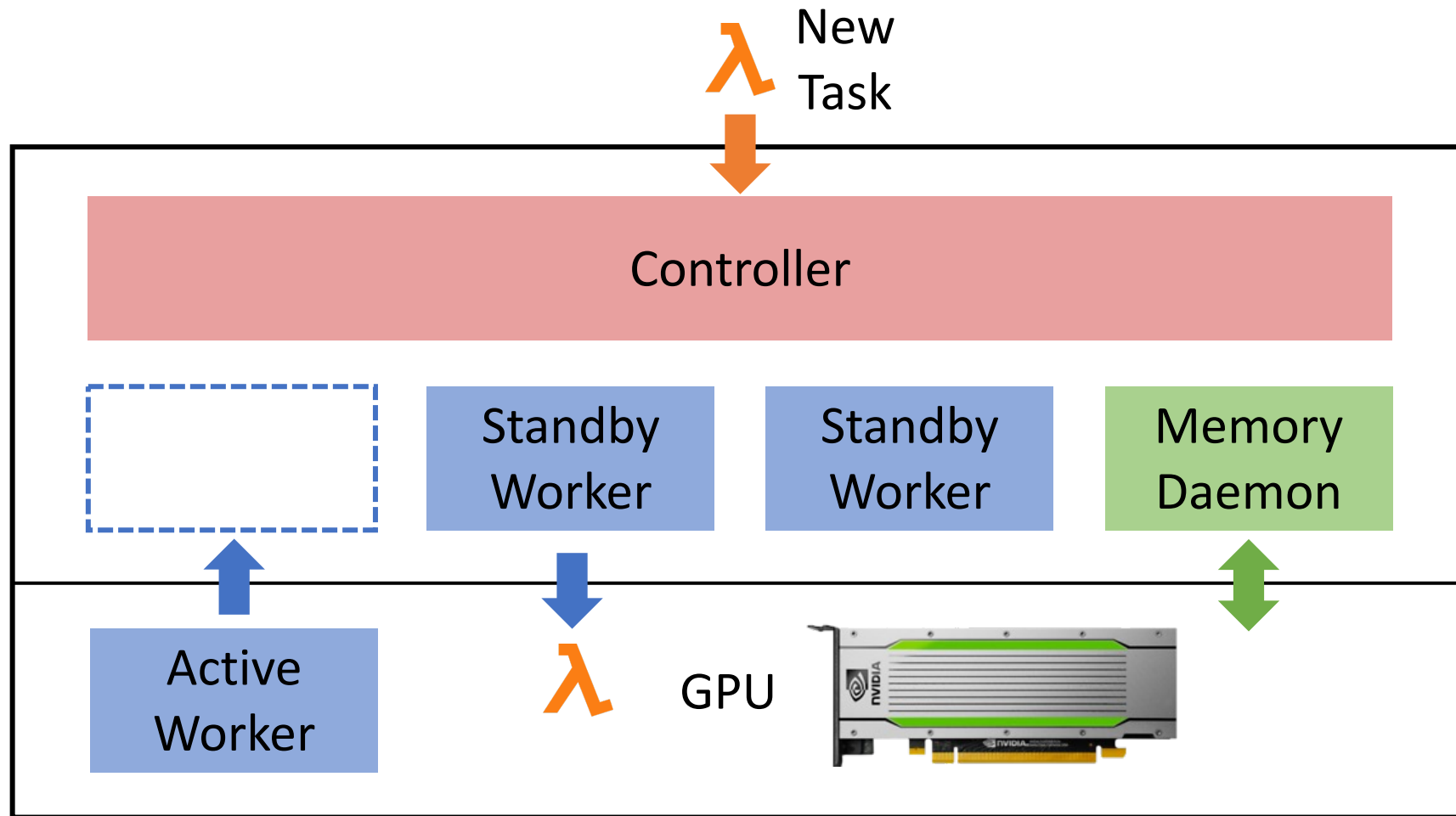
Goal: fast context switching



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

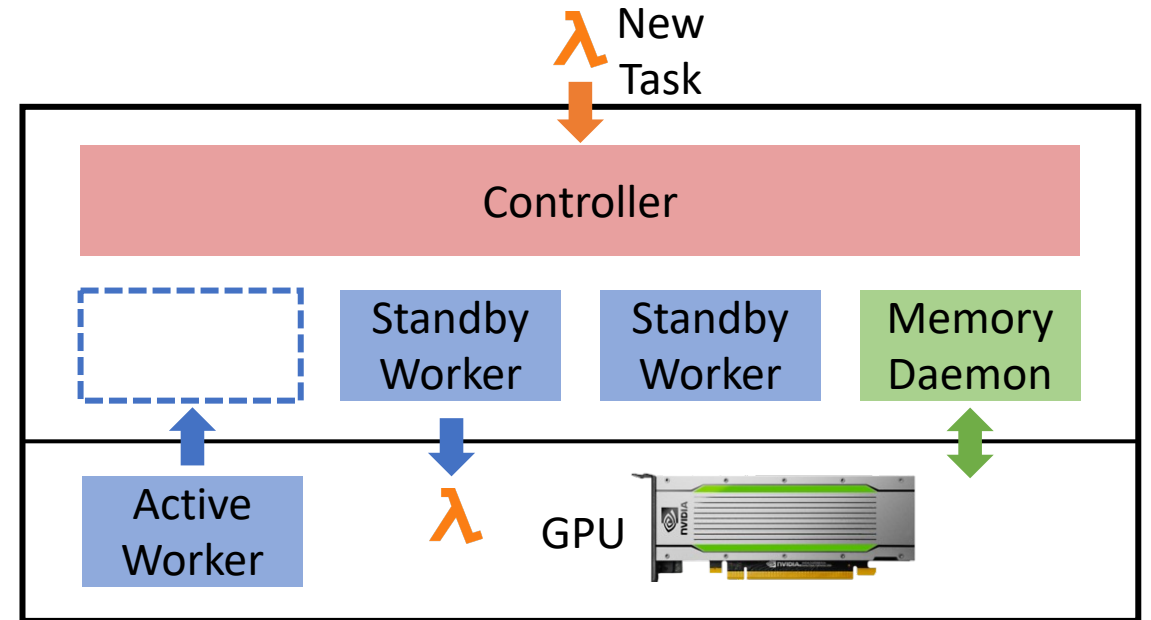
Latency: 6s

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

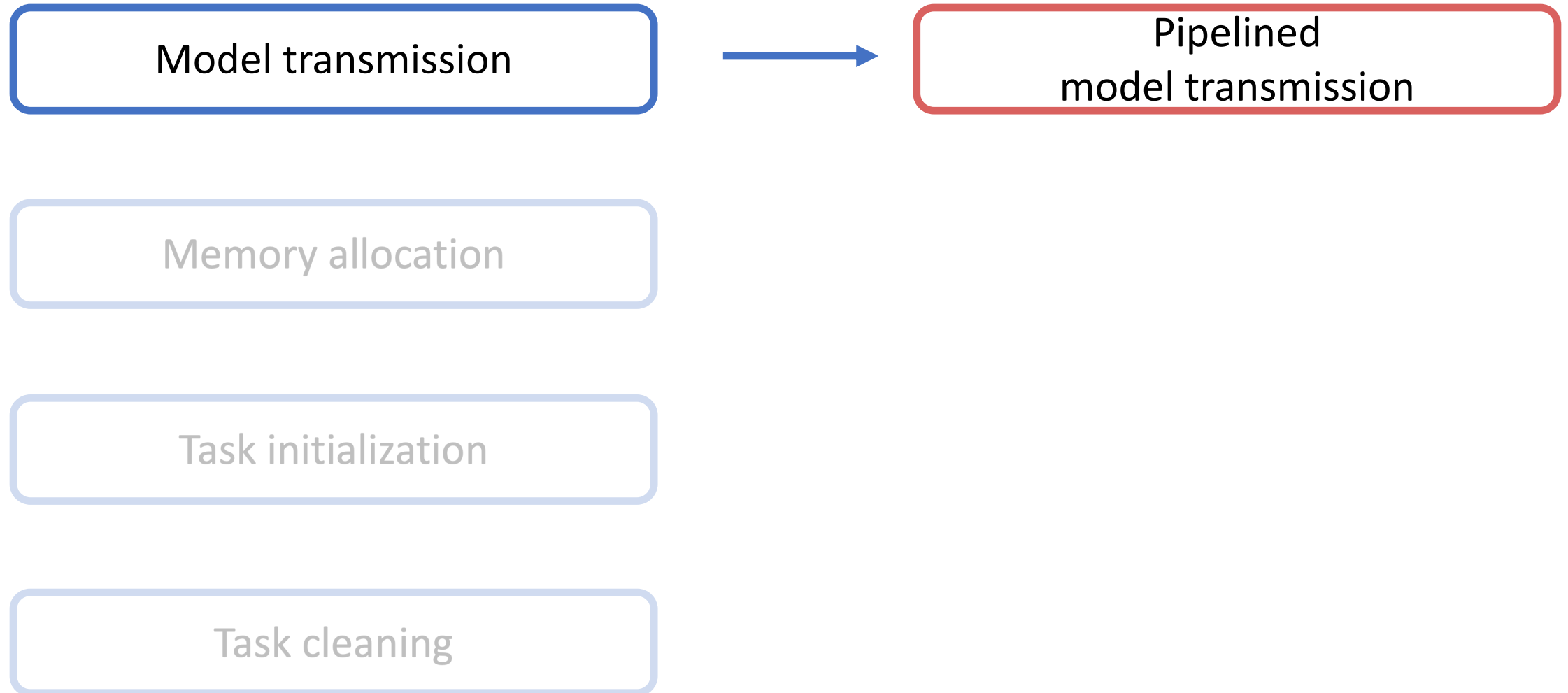
Model transmission

Memory allocation

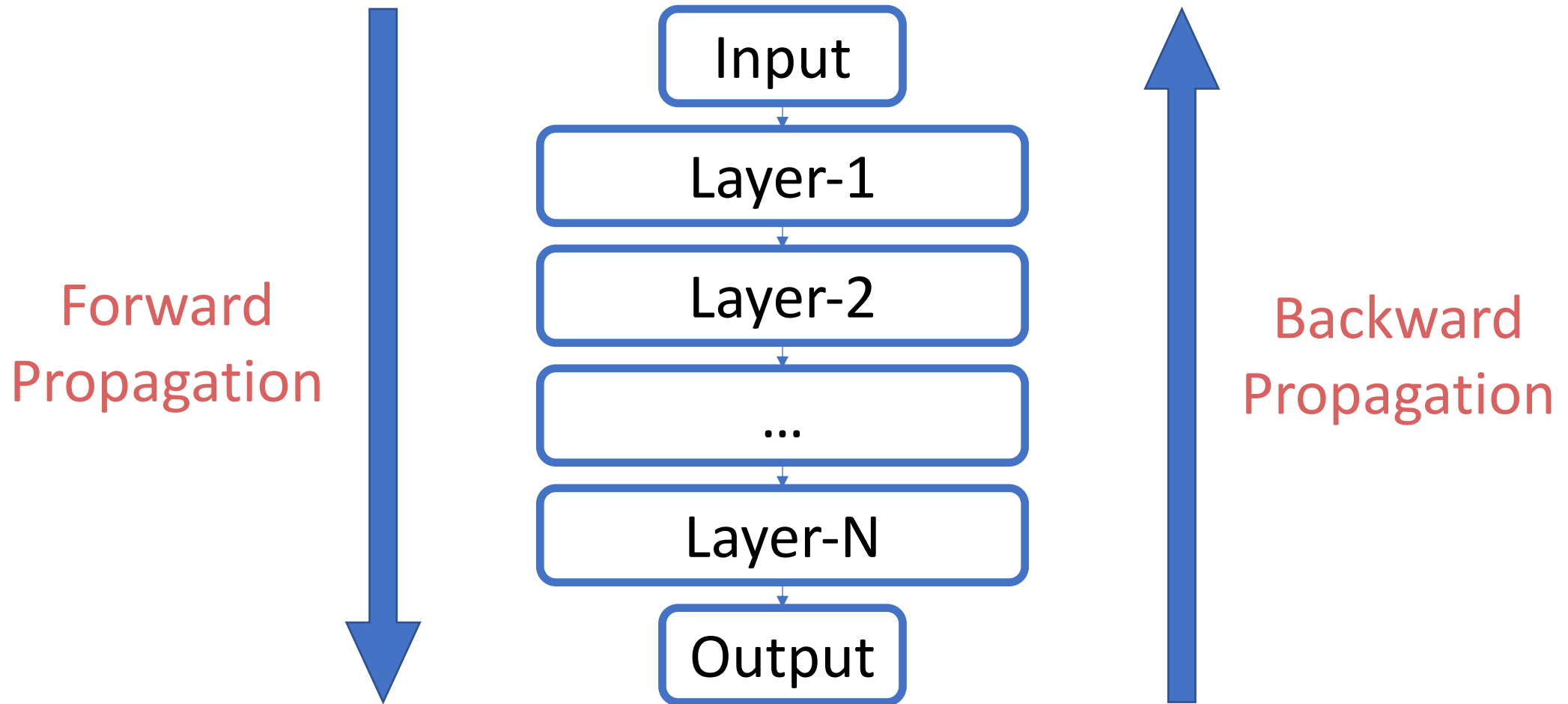
Task initialization

Task cleaning

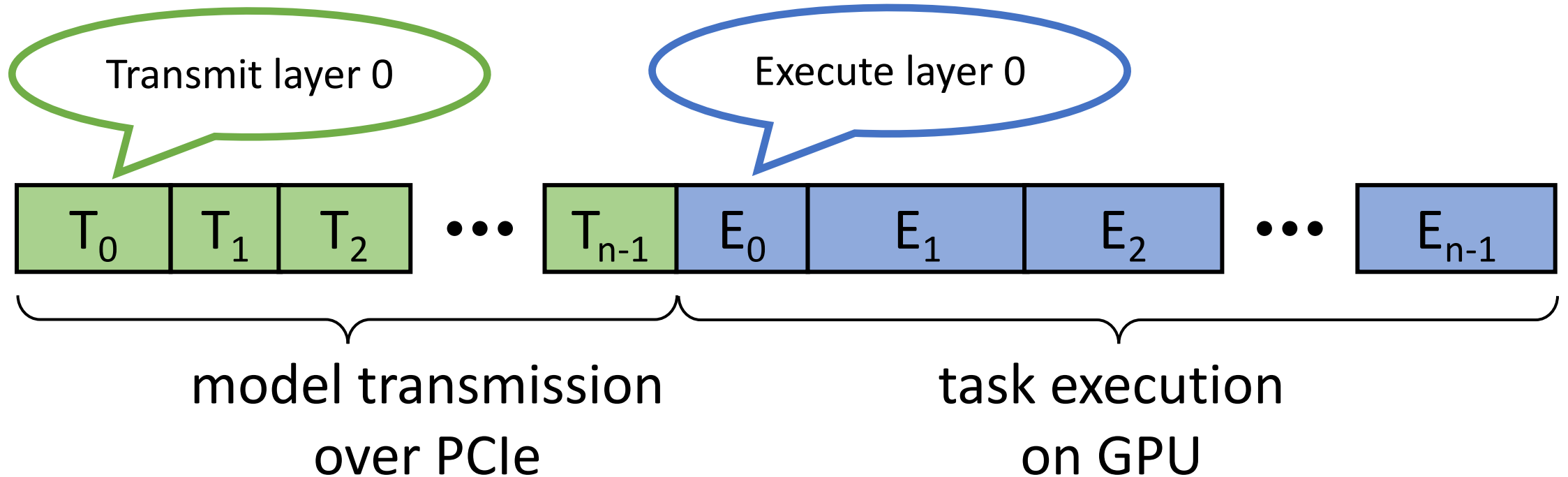
How to reduce the overhead?



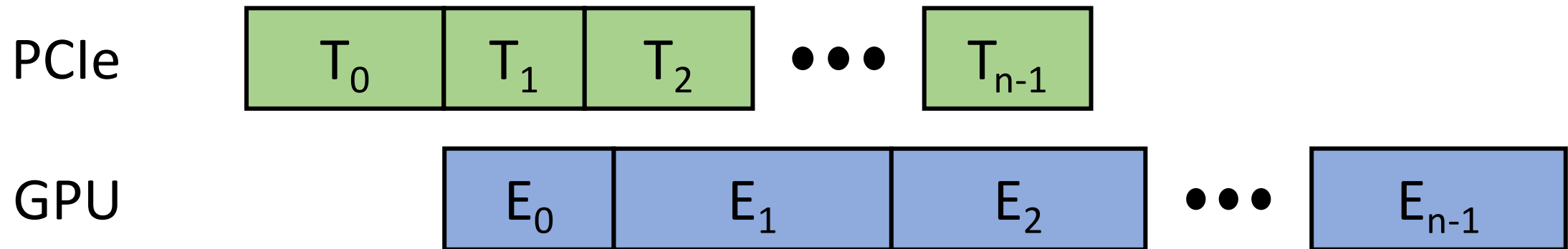
DL models have layered structures



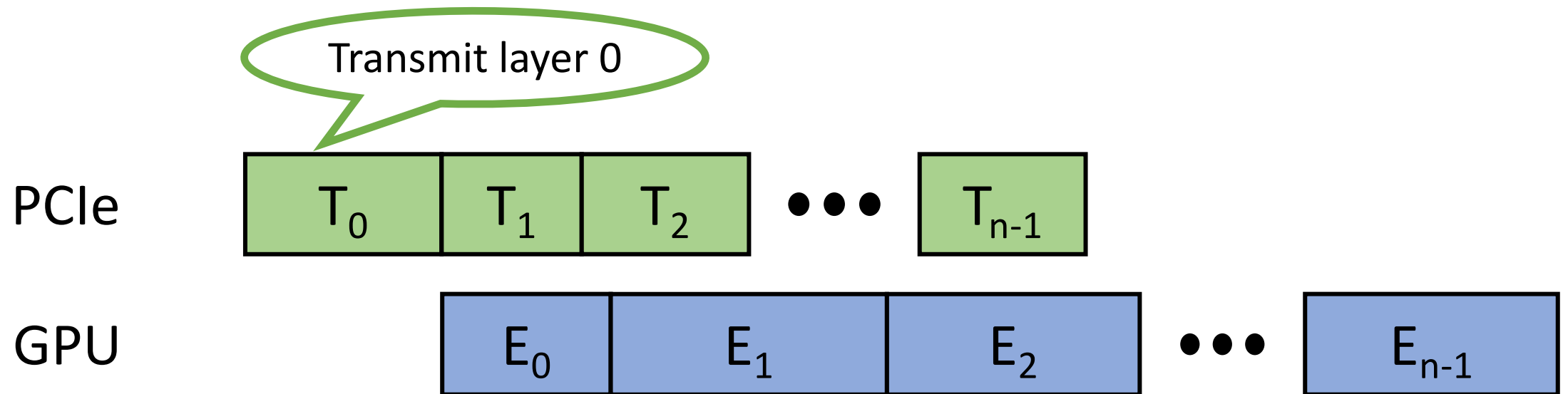
Sequential model transmission and execution



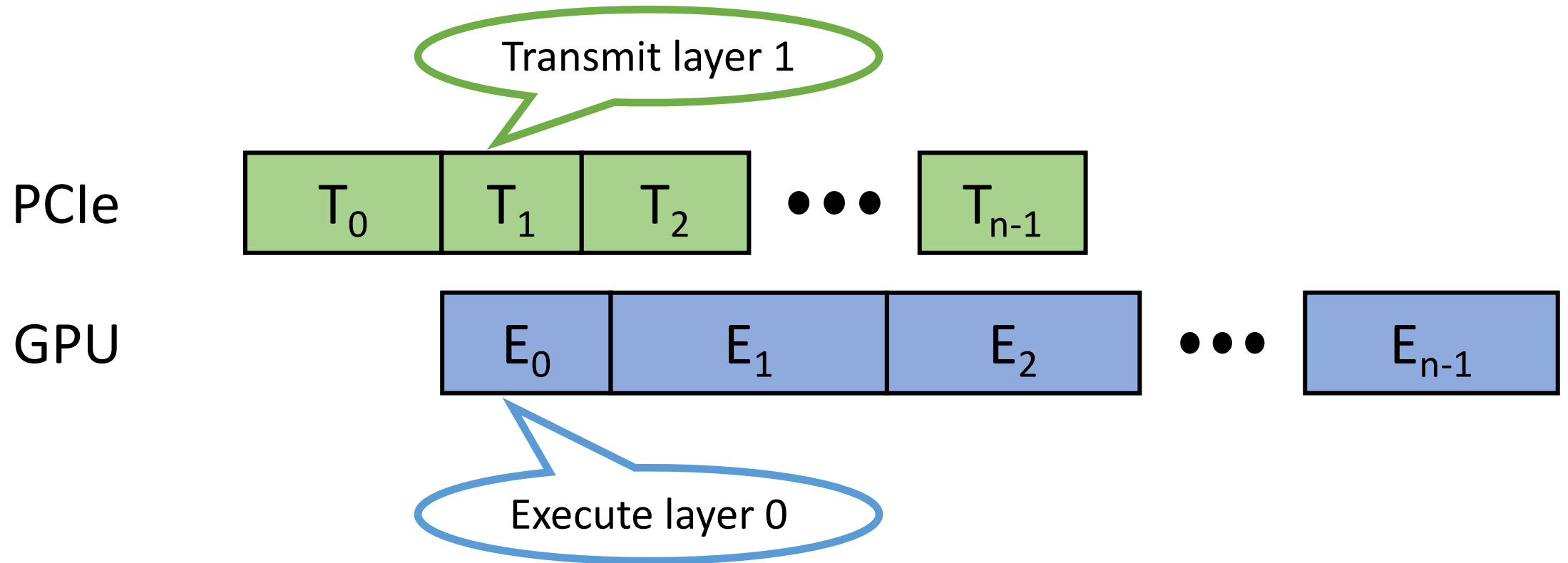
Pipelined model transmission and execution



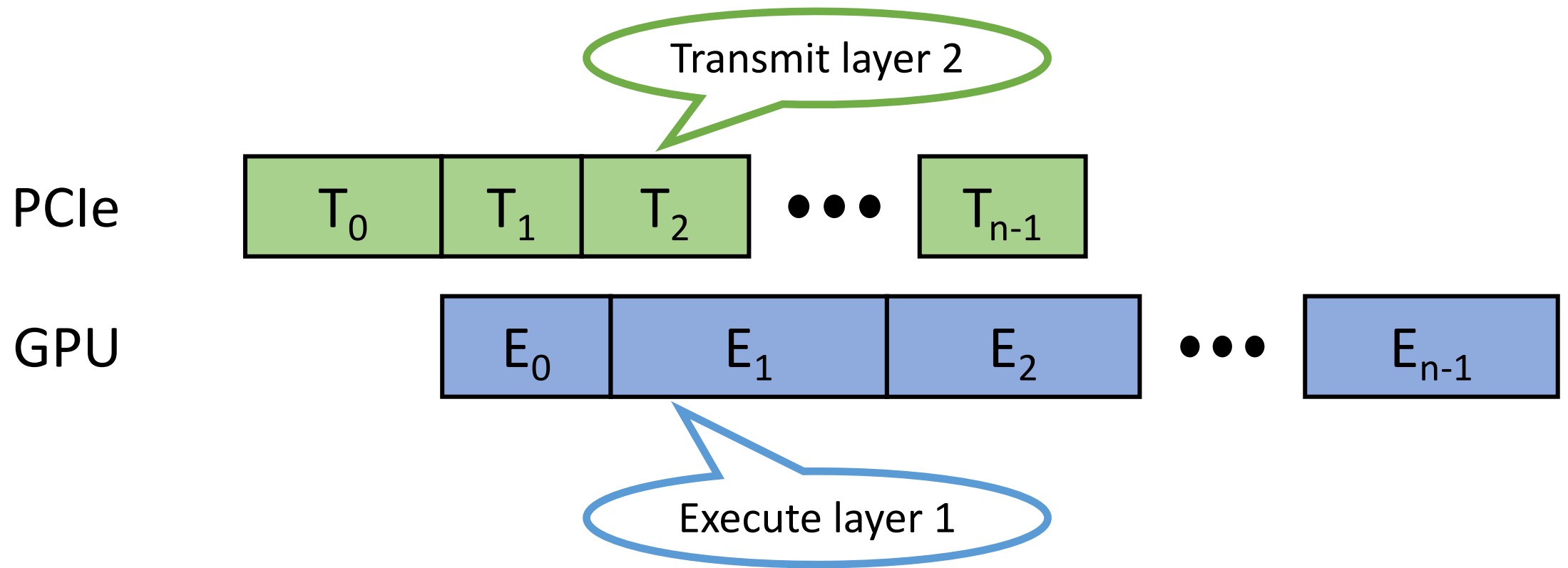
Pipelined model transmission and execution



Pipelined model transmission and execution



Pipelined model transmission and execution



Pipelined model transmission and execution

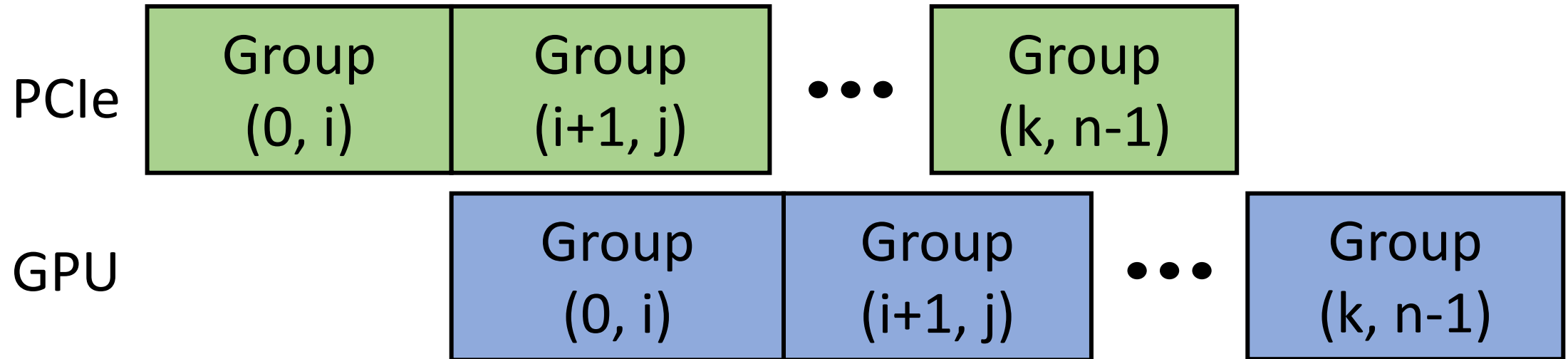
PCIe

GPU

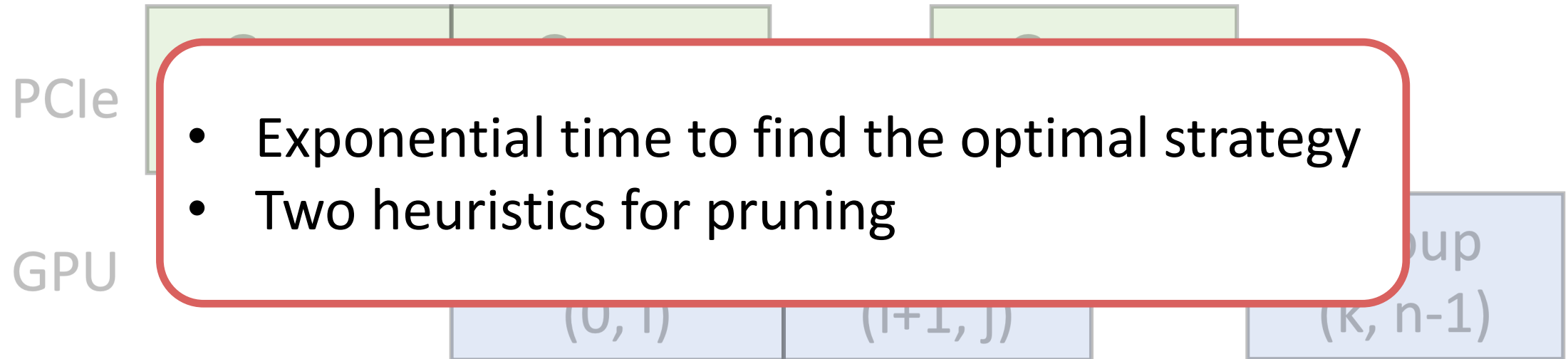
1. Multiple calls to PCIe;
2. Synchronize transmission and execution.

E_{n-1}

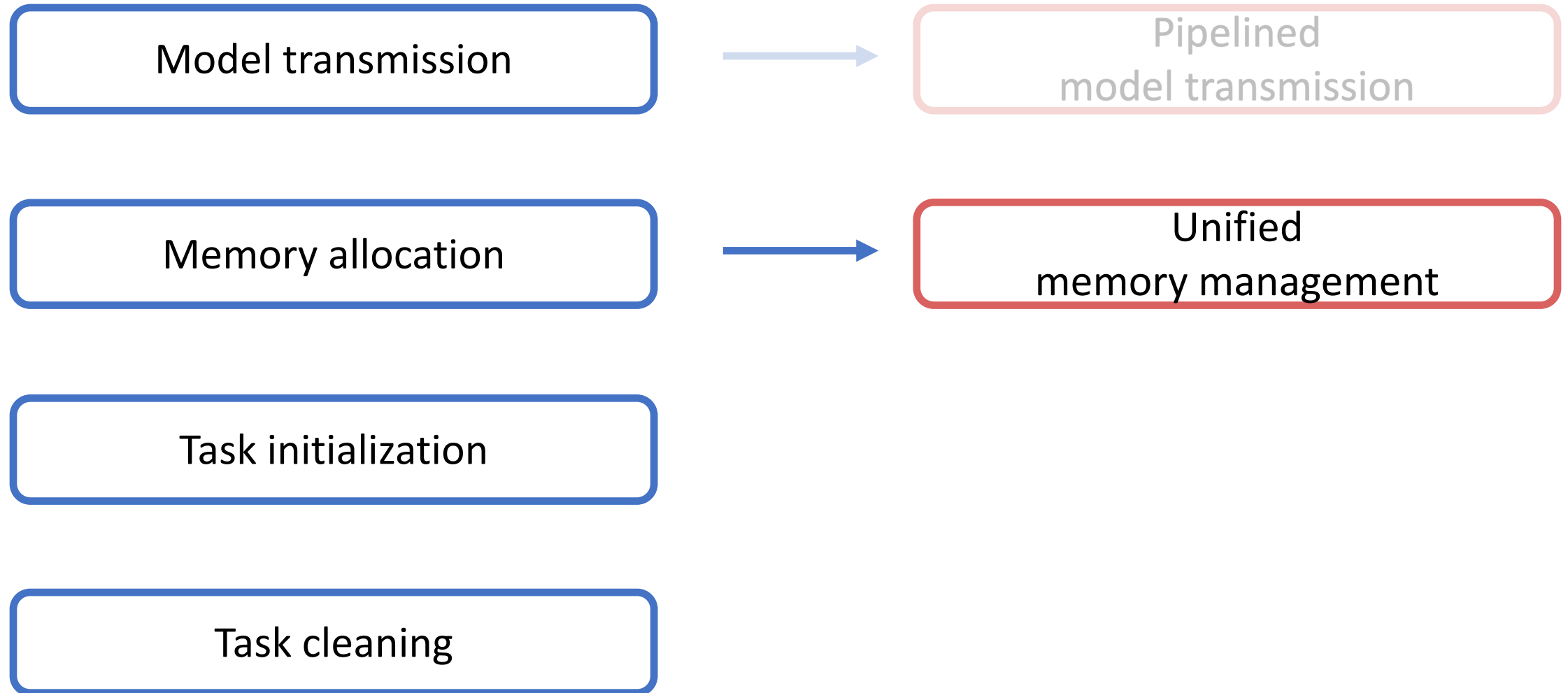
Pipelined model transmission and execution



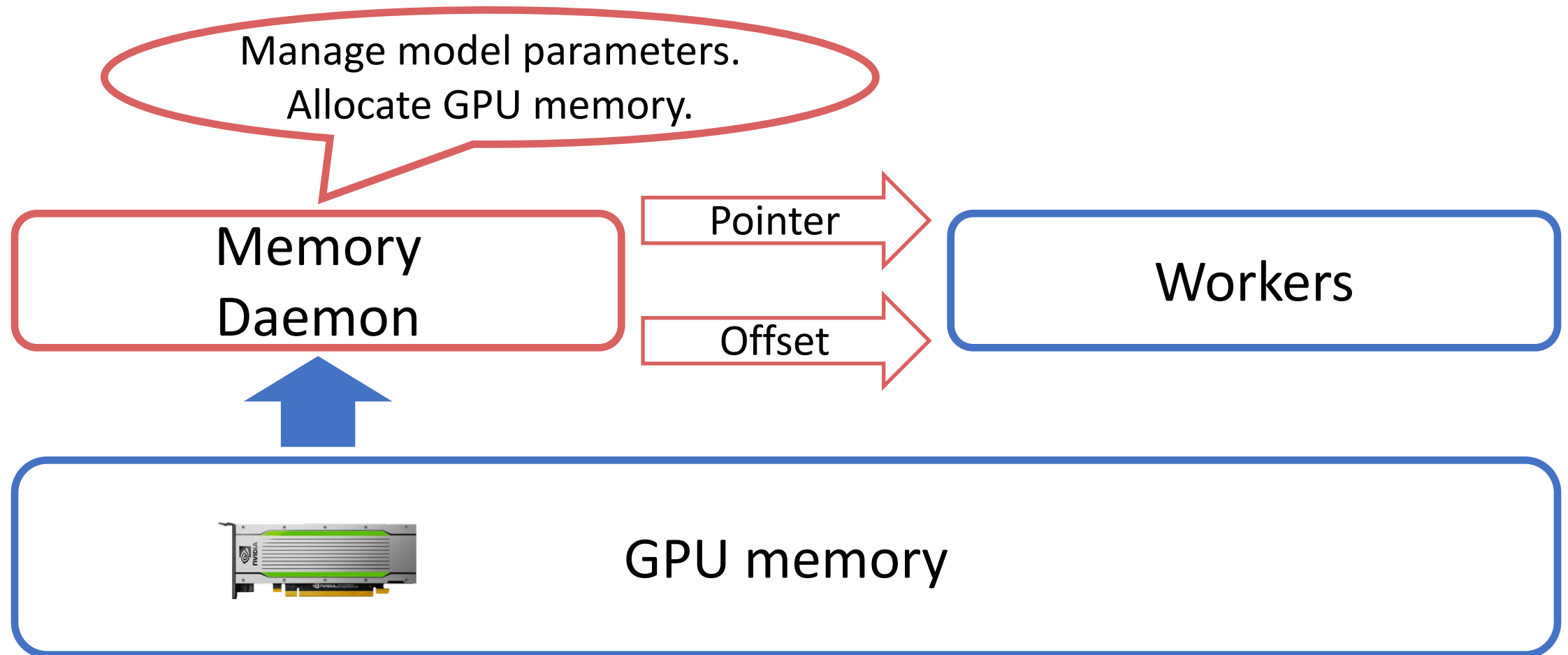
Pipelined model transmission and execution



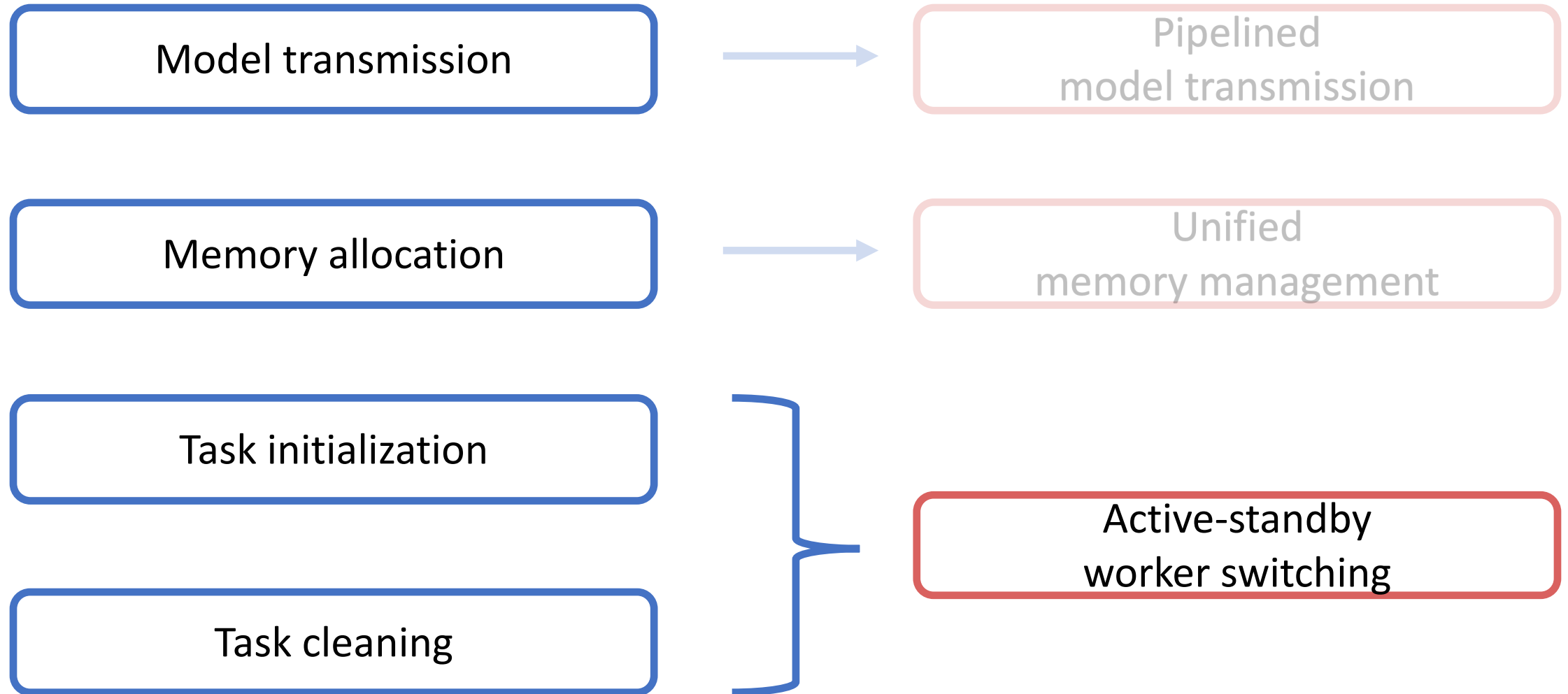
How to reduce the overhead?



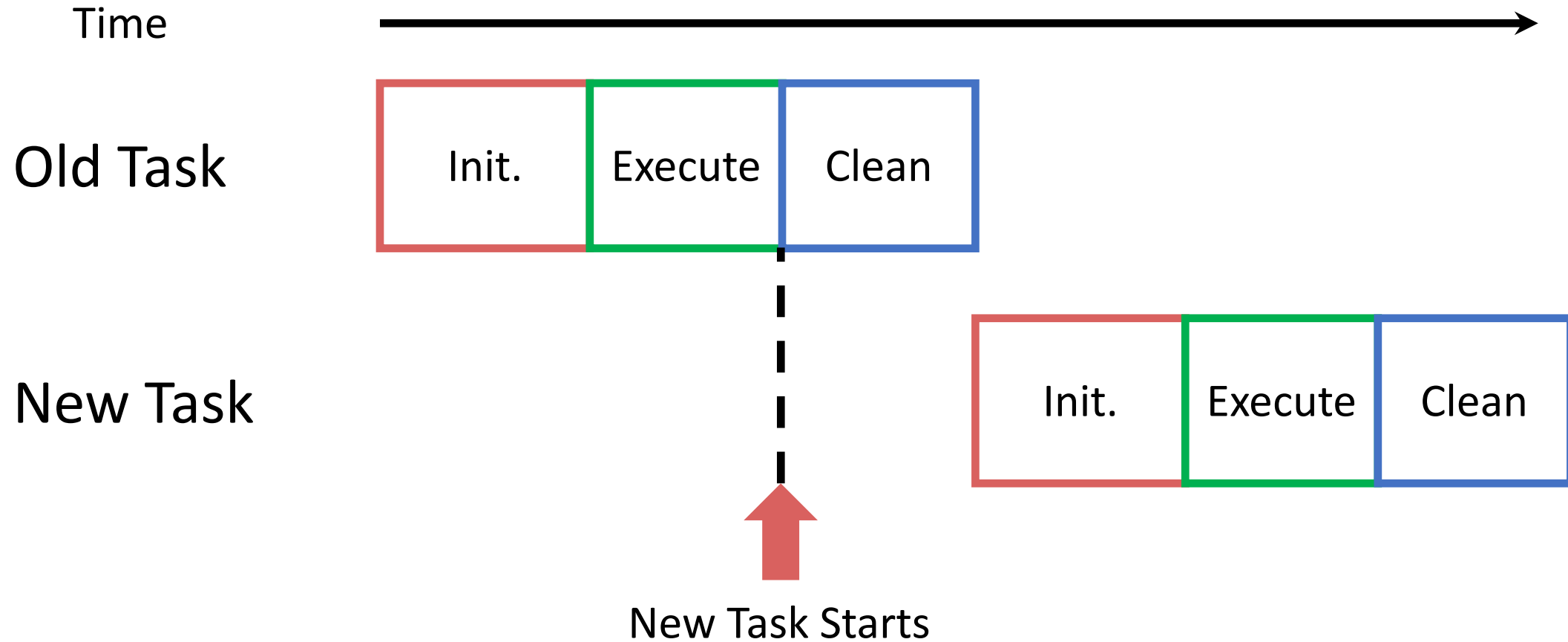
Unified memory management



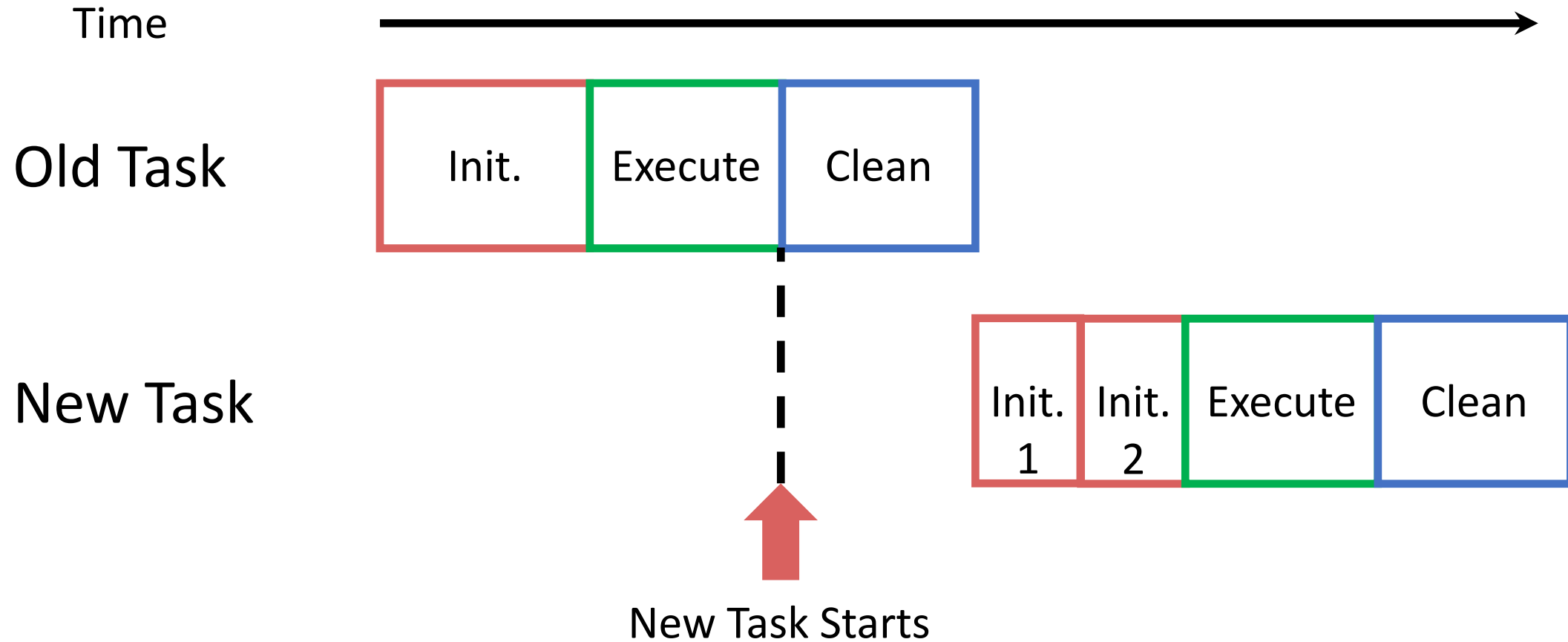
How to reduce the overhead?



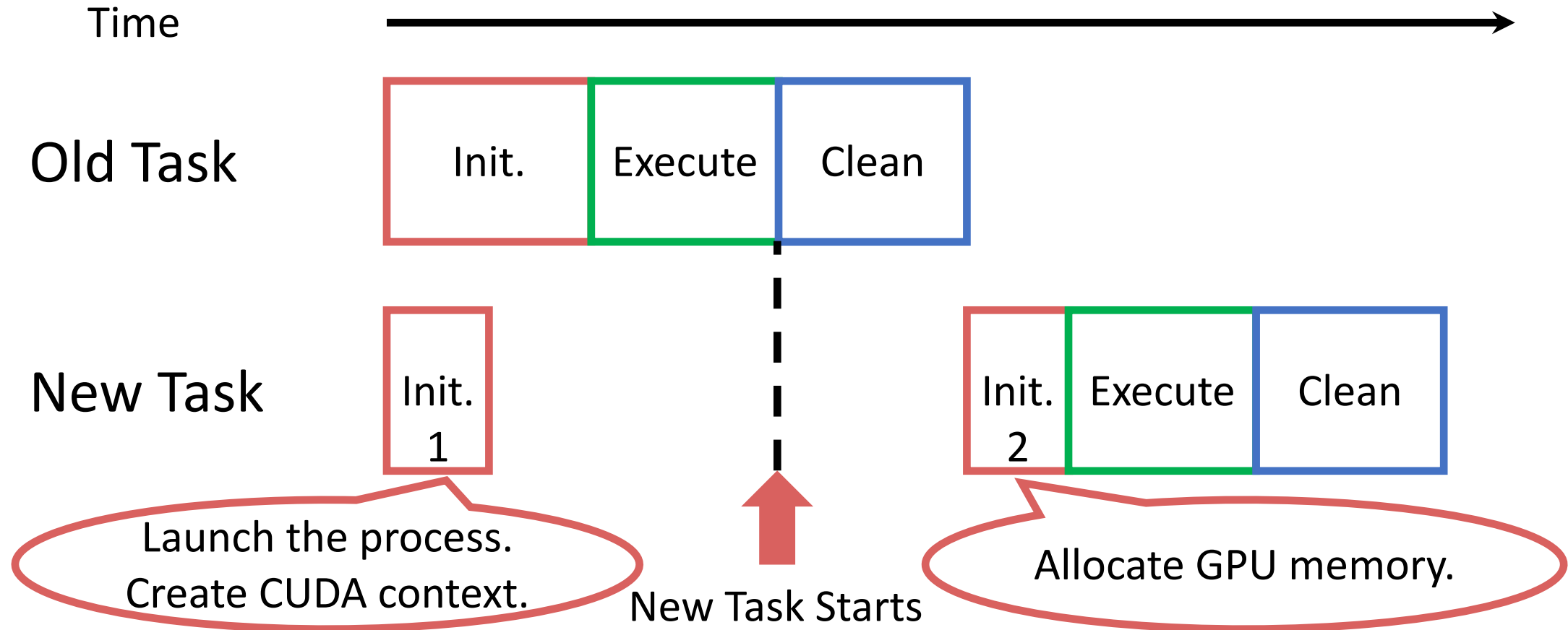
Active-standby worker switching



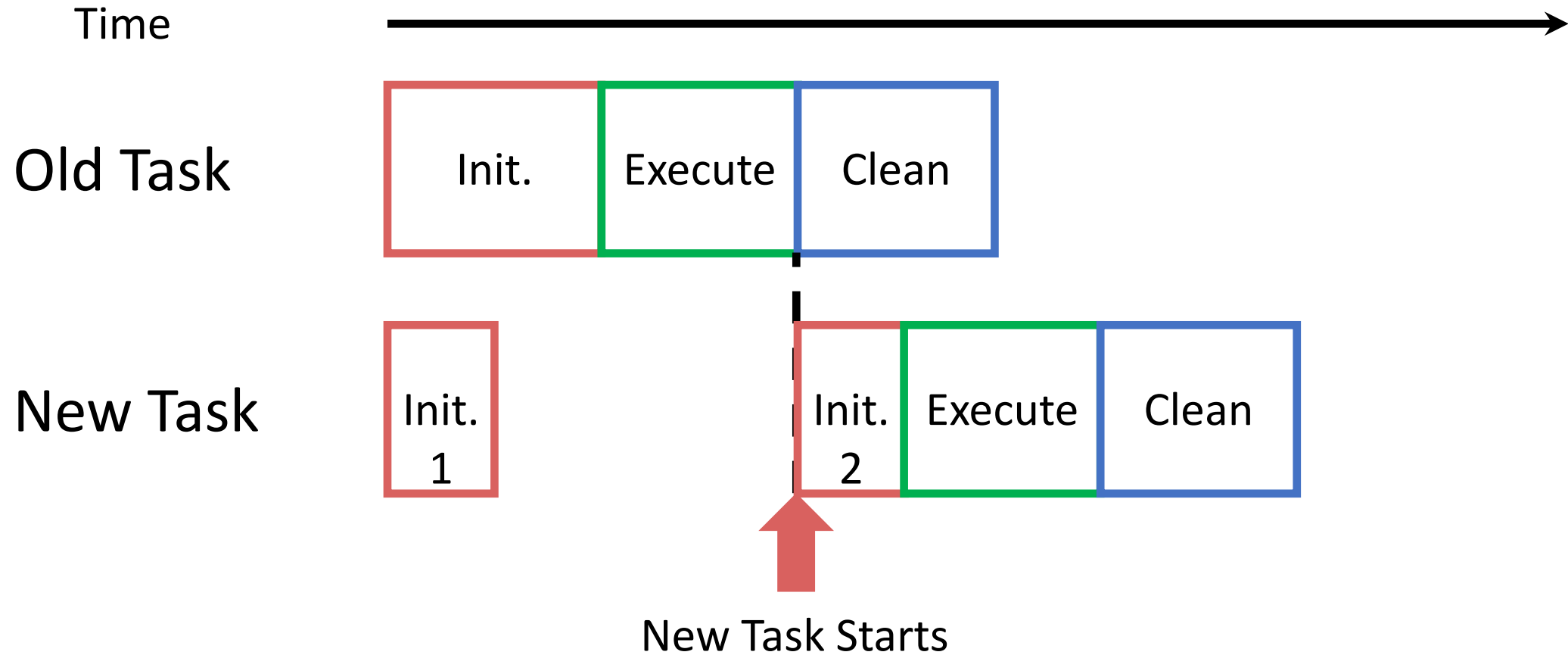
Active-standby worker switching



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

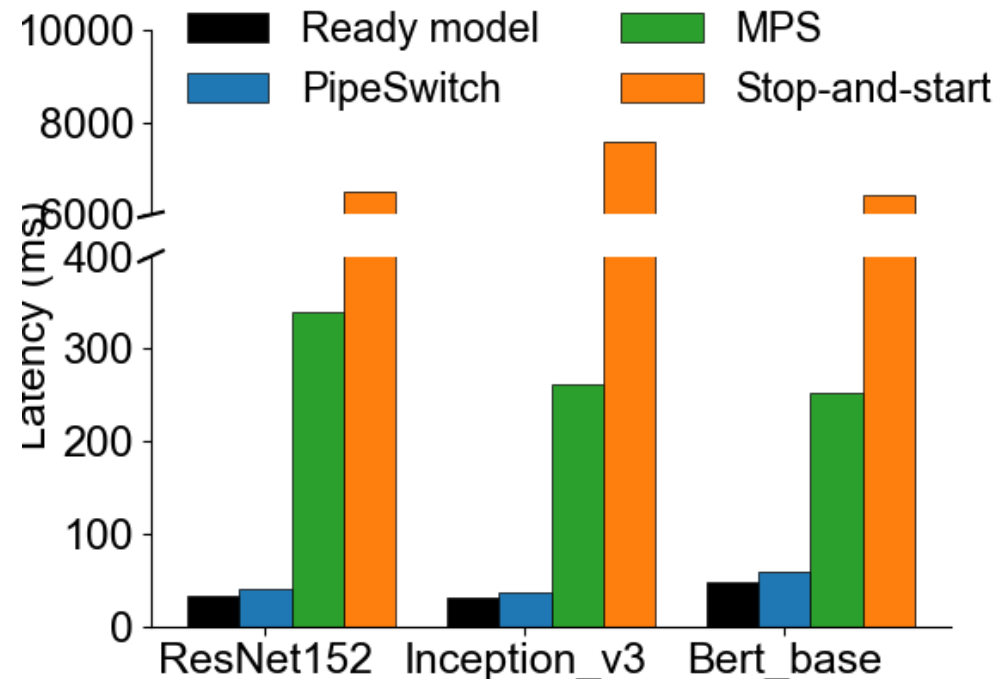
- Can PipeSwitch satisfy SLOs?
- Can PipeSwitch provide high utilization?
- How well do the design choices of PipeSwitch work?

Evaluation

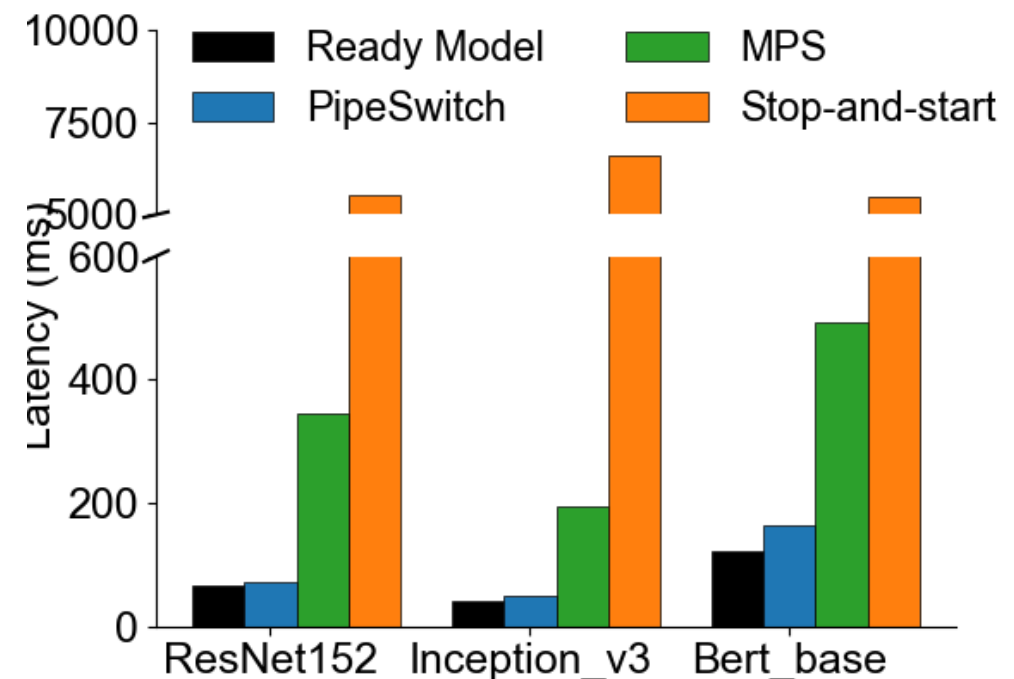
- Can PipeSwitch satisfy SLOs?
- Can PipeSwitch provide high utilization?
- How well do the design choices of PipeSwitch work?

PipeSwitch satisfies SLOs

NVIDIA Tesla V100

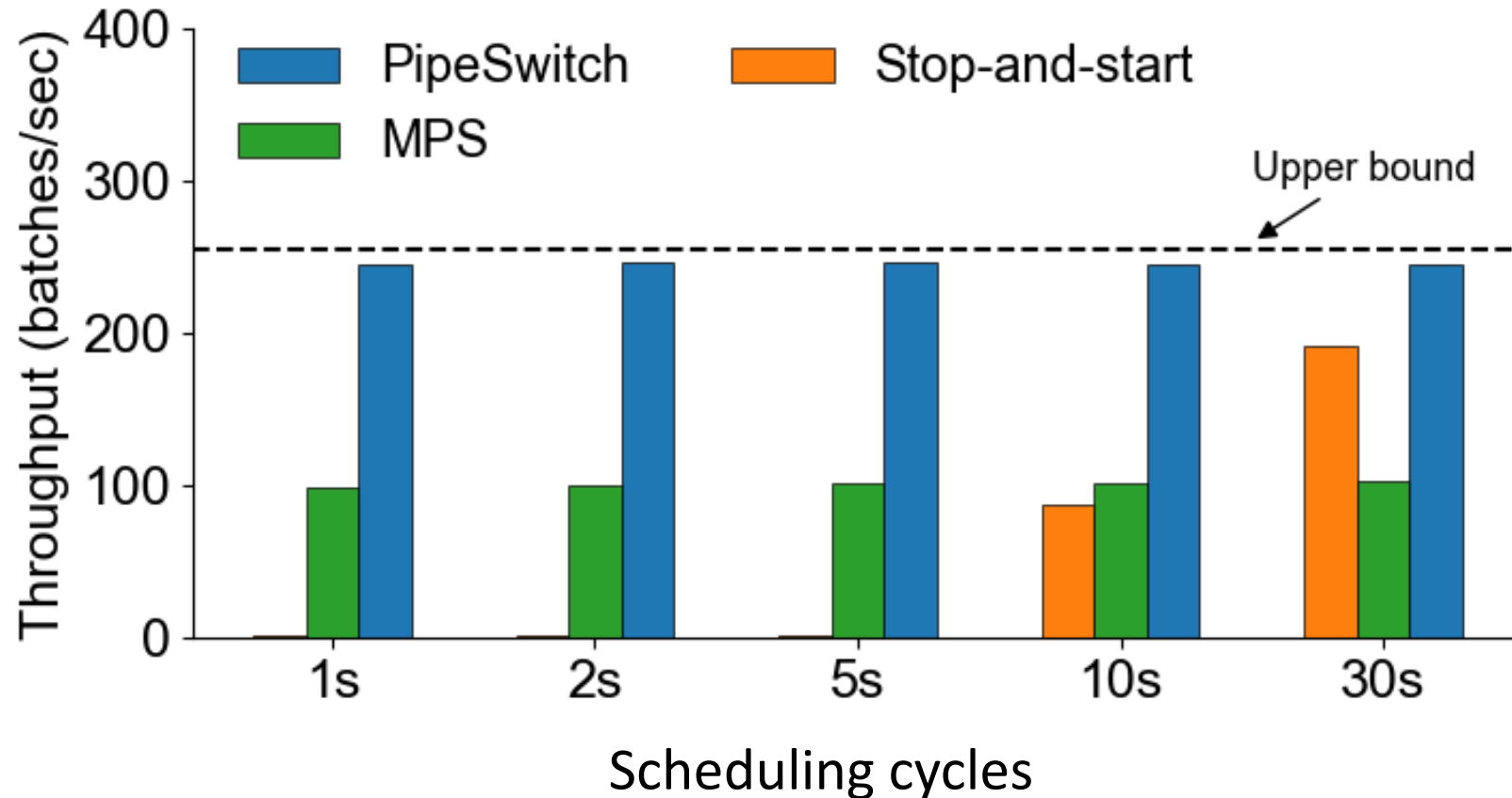


NVIDIA Tesla T4



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

Memory Management in Modern Computer Systems

- Memory Abstraction
 - NSDI'14 FaRM
 - SOSp'23 vLLM
- Demand paging: remote memory over network
 - NSDI'17 InfiniSwap
 - OSDI'20 AIFM
- Demand paging: memory swapping between GPU memory and host memory
 - OSDI'20 PipeSwitch
 - NSDI'23 TGS



Transparent GPU Sharing in Container Clouds for Deep Learning Workloads

Bingyang Wu, Zili Zhang, Zhihao Bai, Xuanzhe Liu, Xin Jin



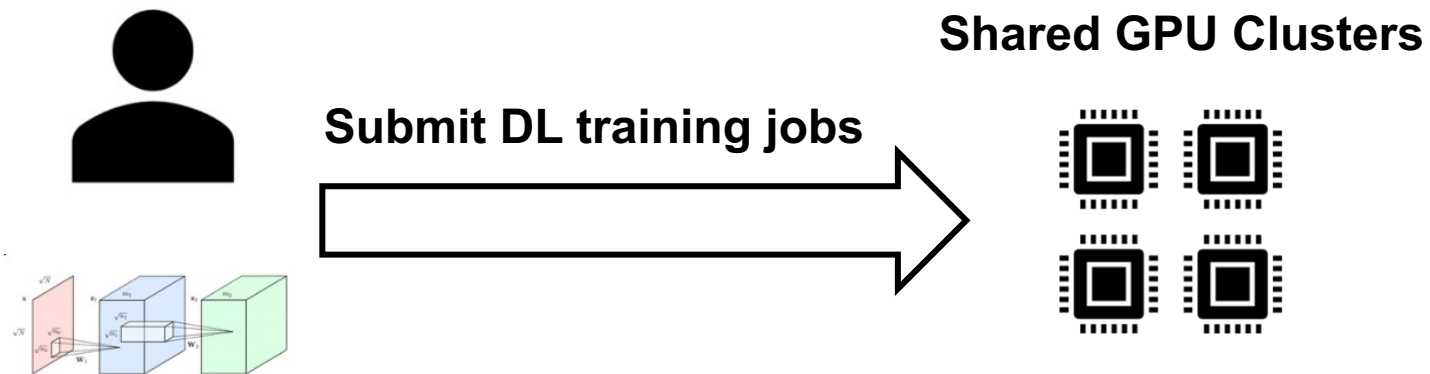
北京大學
PEKING UNIVERSITY



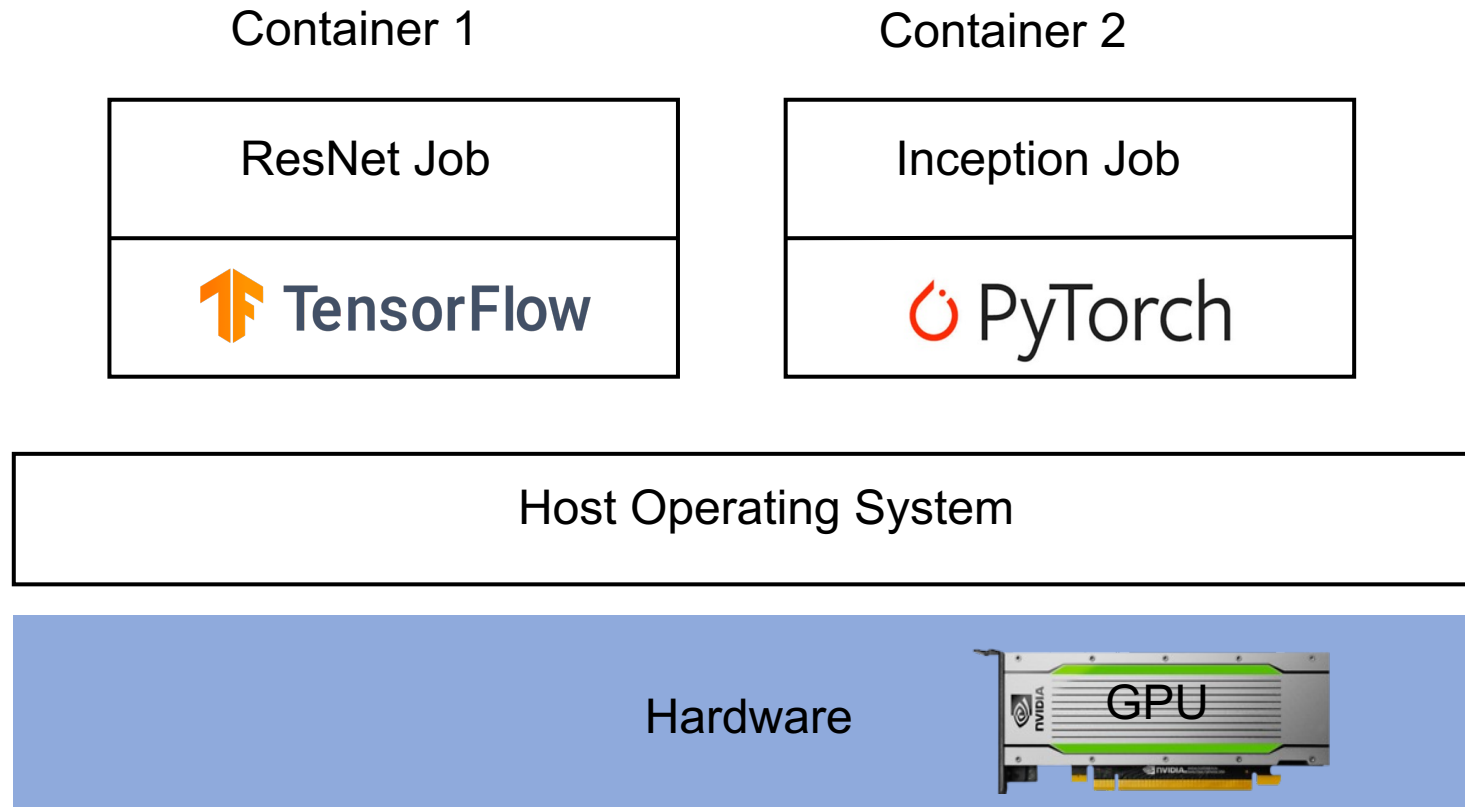
JOHNS HOPKINS
UNIVERSITY

Deep learning training jobs: important workloads in datacenters

- Deep learning is widely used in many applications
 - Recommendation
 - Machine Translation
 - Voice Assistant
 -
- Deep learning models are often trained in **shared GPU clusters**



Deep learning training jobs in container clouds



Low GPU utilization in production

- Microsoft [1]: the average GPU utilization is only 52%
- Alibaba [2]: the median GPU utilization is no more than 10%
- Low GPU utilization is bad
 - Container clouds: idle GPUs are a huge waste
 - Users: longer queueing delay, longer job completion time
- **Root cause:** Each GPU is **statically** assigned to a **single** container

[1] M. Jeon, et al., “Analysis of large-scale multitenant GPU clusters for DNN training workloads,” in *USENIX ATC 2019*.

[2] W. Xiao, et al., “Antman: Dynamic scaling on GPU clusters for deep learning,” in *USENIX OSDI 2020*.

Existing GPU sharing solutions

- **Key idea:** Share GPUs to improve GPU utilization
- Classify DLT jobs into two classes
 - **Production job:** Run without performance degradation
 - **Opportunistic job:** Utilize spare GPU resources to execute
- SOTA solutions:
 - Application-layer solution: AntMan [OSDI' 20]
 - OS-layer solution: NVIDIA MPS, NVIDIA MIG

Application-layer solution: AntMan

- Custom DL framework
 - Modify TensorFlow (~4000 LoC) or PyTorch (~2000 LoC)
- Support GPU compute sharing and GPU memory oversubscription
- **Limitations:** Lack of **Transparency**
 - **Limited use cases:** restricts users to use particular frameworks
 - **Huge operation overhead:** need to maintain custom frameworks

OS-layer solution: NVIDIA MPS

- A software solution for GPU sharing provided by NVIDIA
- Limitations:
 - Low GPU utilization
 - Does not support GPU memory oversubscription
 - Requires application knowledge to properly set the resource limit
 - Weak fault isolation
 - When a job fails, other jobs may be affected and even fails

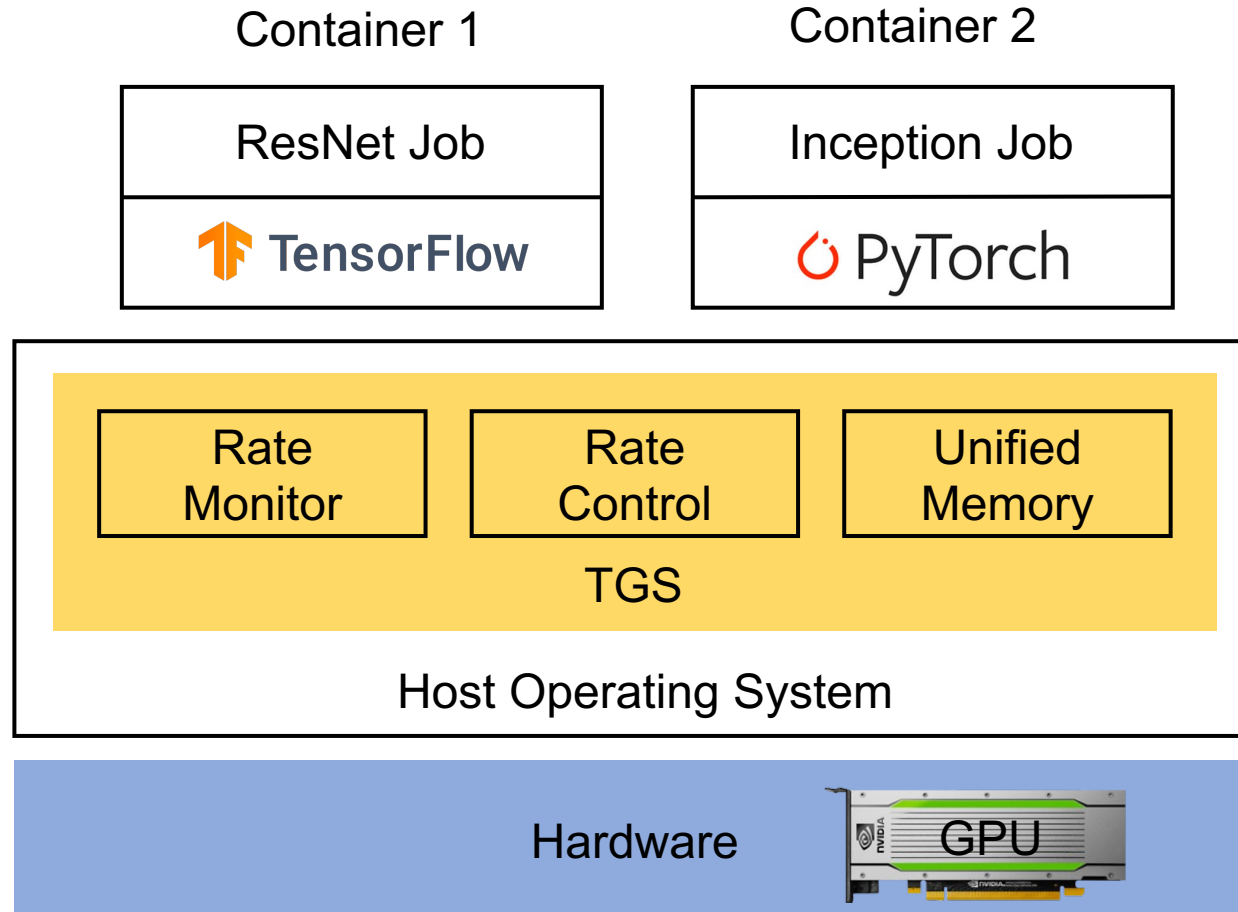
OS-layer solution: NVIDIA MIG

- A recent hardware solution for GPU sharing provided by NVIDIA
- Limitations:
 - Performance isolation
 - Cannot arbitrarily partition a GPU
 - Cannot dynamically change GPU resources
 - Compatibility
 - Only available on a few high-end GPUs
 - Does not support GPU sharing for the multi-GPU instance

A more practical solution: TGS

	AntMan	MPS	MIG	TGS
Transparency		✓	✓	✓
High utilization	✓			✓
Performance isolation	✓	✓	✓	✓
Fault isolation	✓		✓	✓

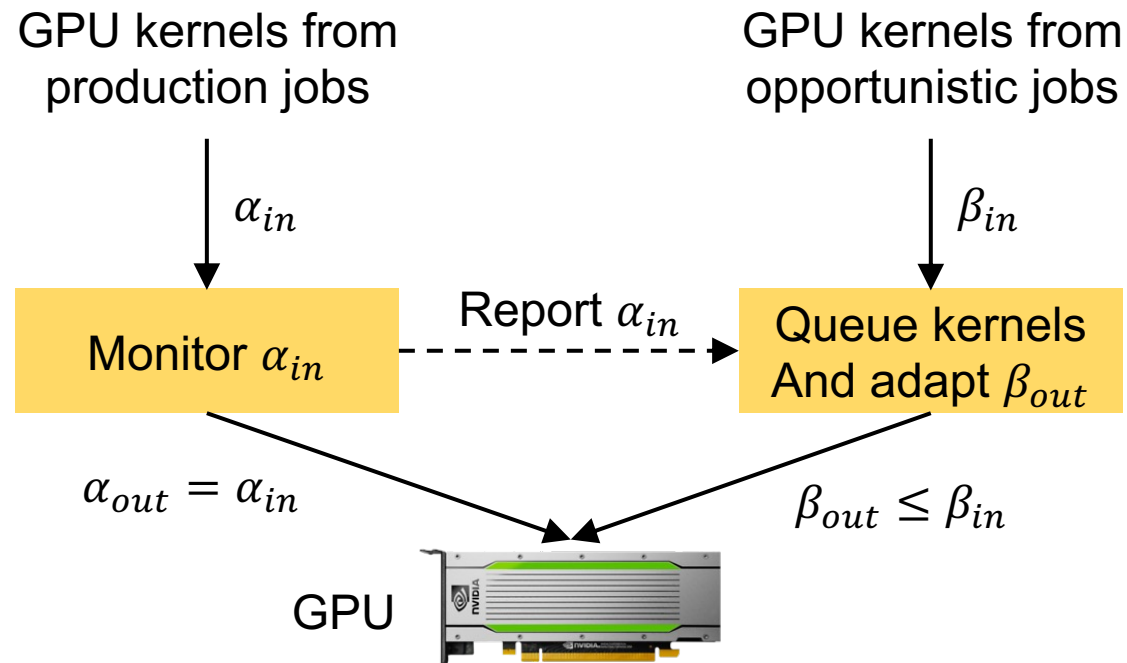
TGS architecture



Sharing GPU compute resources

- Strawman solution: priority scheduling
 - Control the opportunistic job based on the GPU kernel queues
- Low GPU utilization:
 - The state of queues do not reflect the remaining GPU resources

Adaptive rate control of TGS



Sharing GPU memory resources

- **Weak Fault isolation**: total GPU memory consumption may exceed GPU memory capacity and cause OOM
- **Low GPU utilization**: some jobs always claim all GPU memory
- Application-layer technique cannot be used in the OS layer
 - Cannot directly ask DL framework to release unused GPU memory
 - Cannot directly change pointer address from GPU memory to host memory

Transparent unified memory of TGS

- **Key ideas:** leverage CUDA unified memory to transparently unify GPU memory and host memory
- **High GPU utilization:** The actual physical GPU memory is allocated when jobs first access to them
- **Fault isolation:** When GPU memory is oversubscribed, TGS changes virtual memory mapping to evict GPU memory of opportunistic job to host memory

Evaluation setup

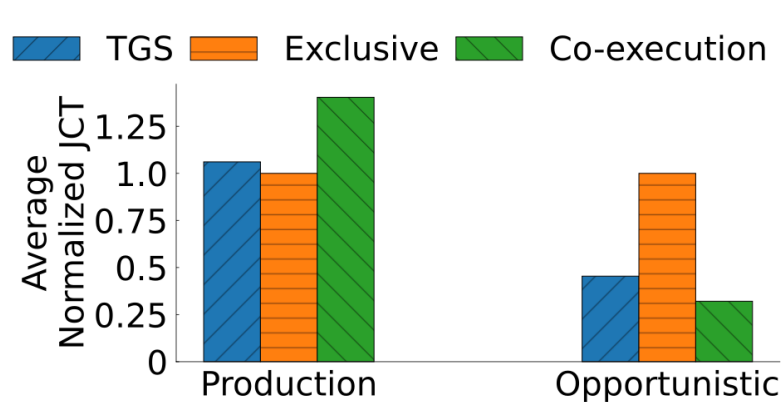
- Implementation: ~3000 LoC C++ & Python
 - Integration with Docker and Kubernetes
- Testbed: NVIDIA A100 GPUs and NVIDIA V100 GPUs
- Trace: Philly Trace from Microsoft [Jeon et al. 2019]
- Models
 - CV: ResNet, ShuffleNet, MobileNet
 - Graph: GCN
 - NLP: Bert, GPT-2
 - Recommendation: DLRM

Evaluation baselines

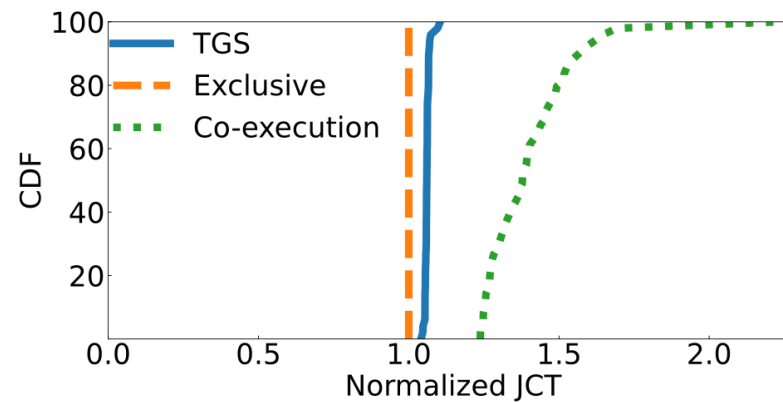
- **TGS**: our work
- **AntMan**: the state-of-the-art application-layer solution
- **MPS**: manually set appropriate limit
- **MIG**: manually set best configuration
- **Exclusive**: give exclusive access to a GPU
- **Co-execution**: share a GPU without any control

Mixed workload job stream

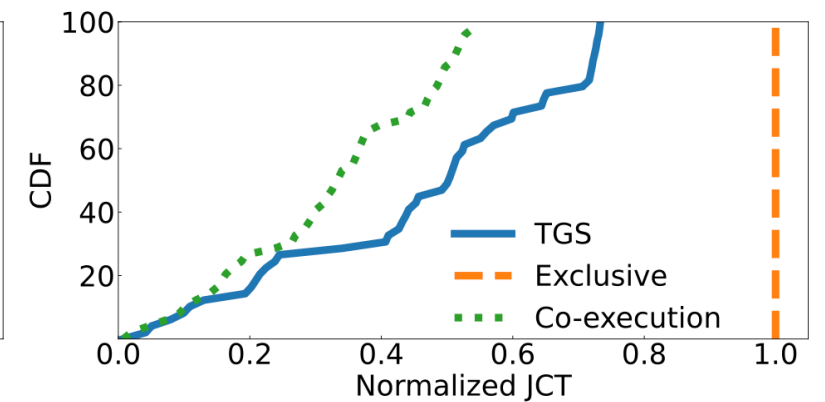
- A job stream contains 50 production jobs and 50 opportunistic jobs
- Opportunistic jobs: 52% JCT reduction compared to Exclusive
- Production jobs: 21% JCT reduction compared to Co-execution



(a) Average JCT.



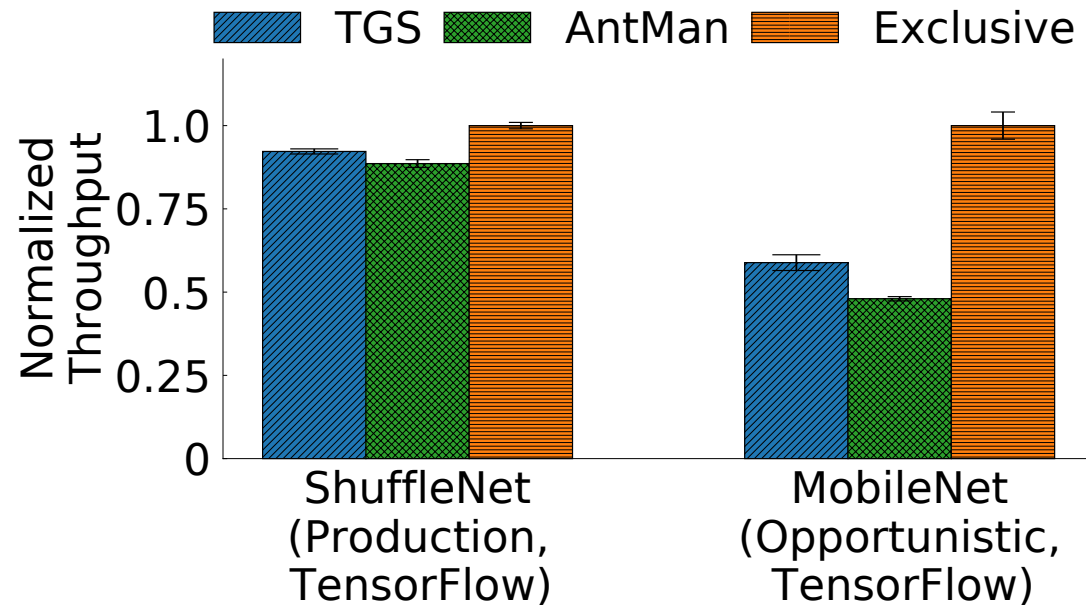
(b) CDF of production jobs.



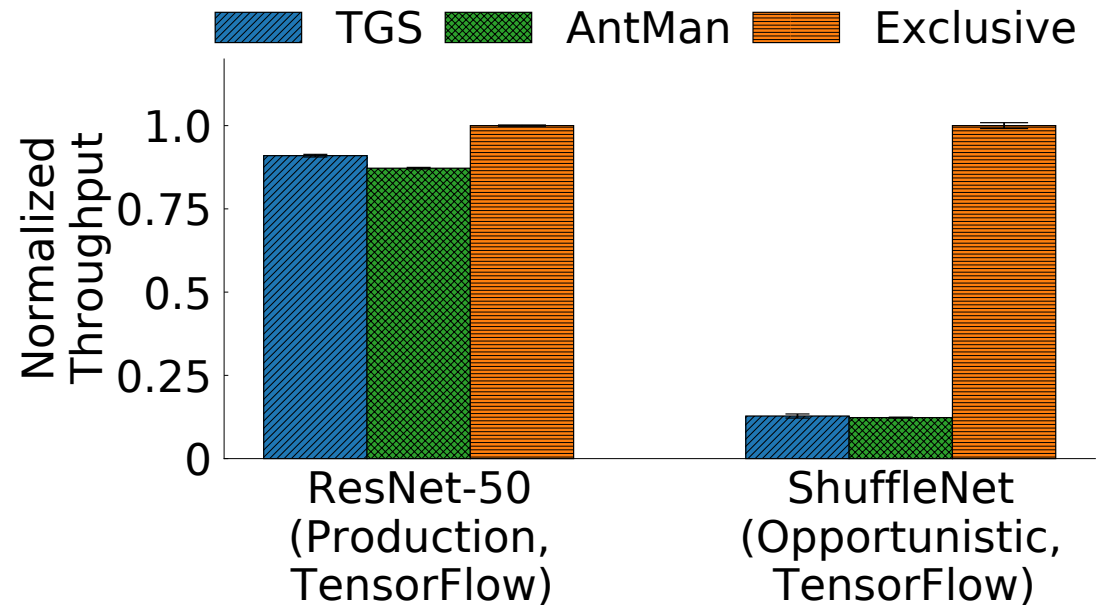
(c) CDF of opportunistic jobs.

Comparison with AntMan

- Achieve comparable performance in different contention scenarios
- Provide transparency without sacrificing performance



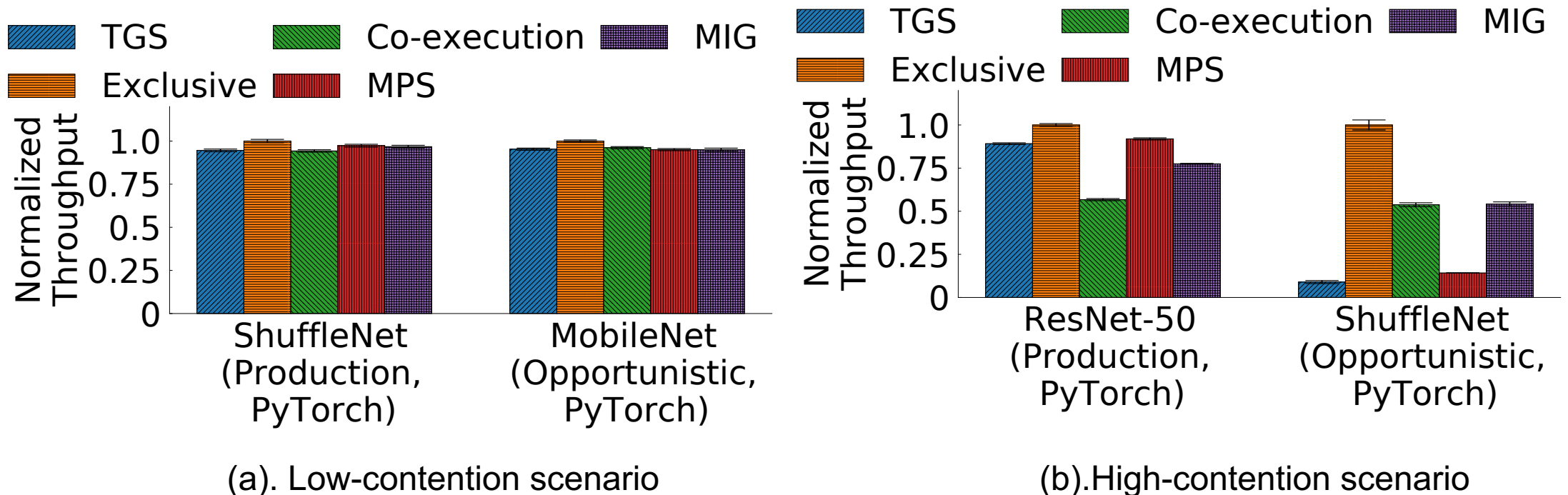
(a). Low-contention scenario



(b). High-contention scenario

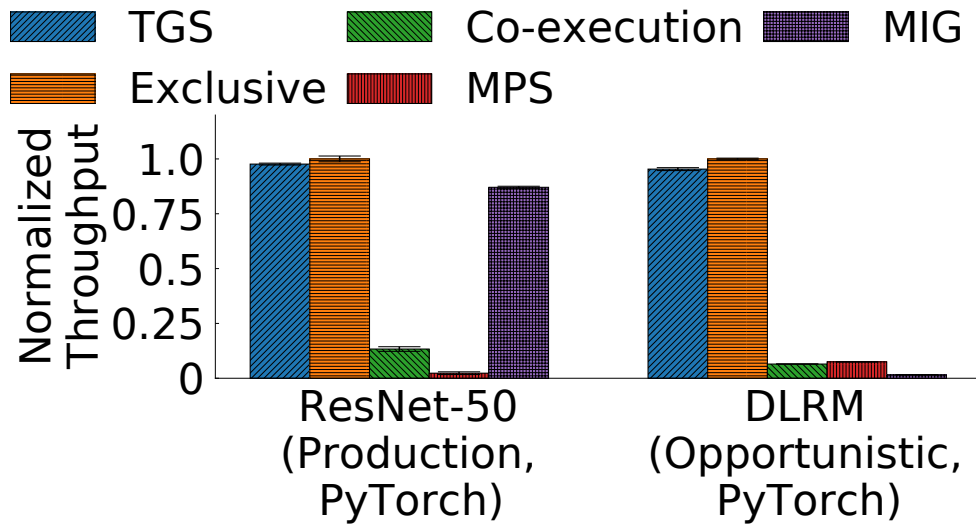
Adaptive rate control of TGS

- TGS protects productions job with little overhead, while providing remaining GPU resources to opportunistic jobs

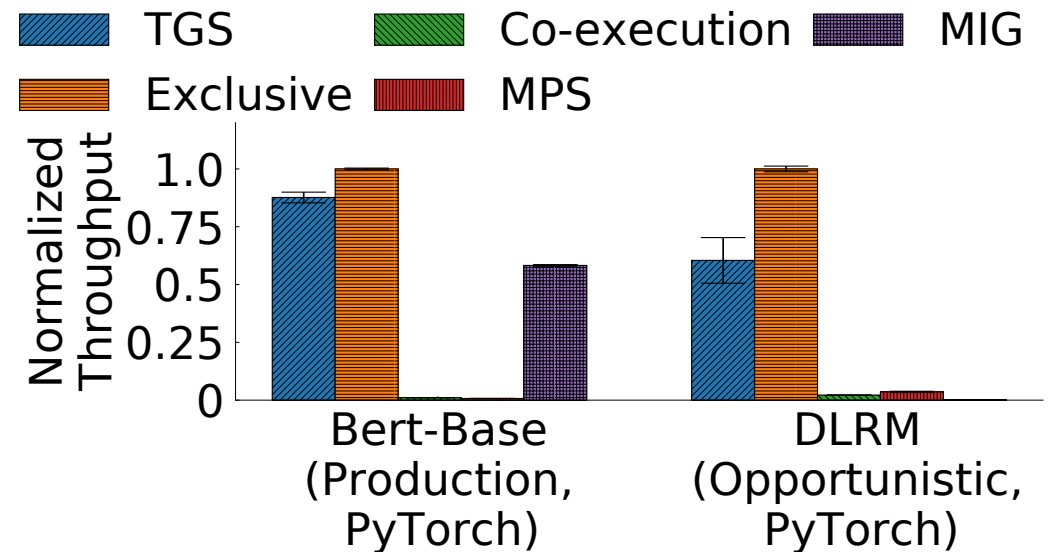


Transparent unified memory of TGS

- TGS protects production jobs under GPU memory oversubscription
- $15\times$ throughput improvement compared to MPS



(a). Low-contention scenario



(b). High-contention scenario

More experiments in our paper

- System overhead
- Convergence of TGS in different scenarios
 - Convergence of the rate control under dynamic job arrival
 - Convergence of the rate control under dynamic resource usage
- Supporting different DL frameworks
- GPU sharing for large model training

Conclusion

- TGS provides transparent GPU sharing to DL training in container clouds with four important properties:
 - Transparency
 - Performance isolation
 - High GPU utilization
 - Fault isolation
- TGS improves the throughput of the opportunistic job by up to $15\times$ compared to the existing OS-layer solution MPS



bingyangwu@pku.edu.cn

Thanks!

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 memory management techniques?
- Can you come up with new ideas?