Operating Systems (Honor Track)

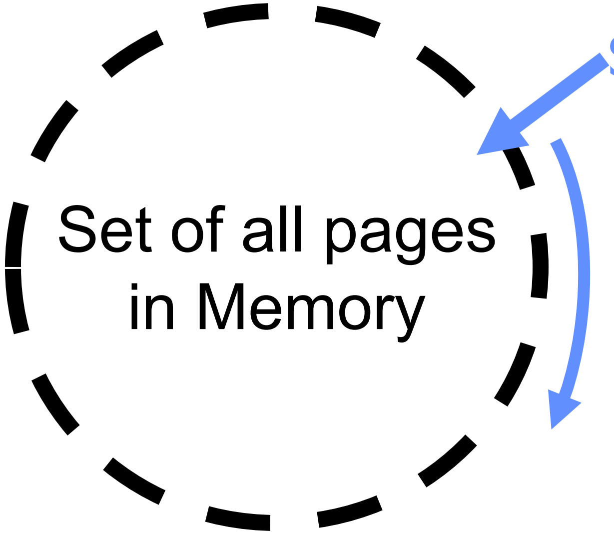
Memory 4: Demand Paging & Memory Management in Modern Computer Systems

Xin Jin

Spring 2022

Acknowledgments: Ion Stoica, Berkeley CS 162

Recap: Approximating LRU: Clock Algorithm



Single Clock Hand:

Advances only on page fault!
Check for pages not used recently
Mark pages as not used recently

- Clock Algorithm: Arrange physical pages in circle with single clock hand
  - Approximate LRU (*approximation to approximation to MIN*)
  - Replace an old page, not the oldest page
- Details:
  - Hardware "use" bit per physical page (called "accessed" in Intel architecture):
    - » Hardware sets use bit on each reference
    - » If use bit isn't set, means not referenced in a long time
  - On page fault:
    - » Advance clock hand (not real time)
    - » Check use bit: 1→ used recently; clear and leave alone
0→ selected candidate for replacement

Recap: N^{th} Chance version of Clock Algorithm

- N^{th} chance algorithm: Give page N chances
  - OS keeps counter per page: # sweeps
  - On page fault, OS checks use bit:
    - » 1 → clear use and also clear counter (used in last sweep)
    - » 0 → increment counter; if count=N, replace page
  - Means that clock hand has to sweep by N times without page being used before page is replaced
- How do we pick N?
  - Why pick large N? Better approximation to LRU
    - » If N ~ 1K, really good approximation
  - Why pick small N? More efficient
    - » Otherwise might have to look a long way to find free page
- What about "modified" (or "dirty") pages?
  - Takes extra overhead to replace a dirty page, so give dirty pages an extra chance before replacing?
  - Common approach:
    - » Clean pages, use N=1
    - » Dirty pages, use N=2 (and write back to disk when N=1)

Group Discussion

- Topic: Clock algorithm variations
  - Do we really need a hardware-supported "modified" bit?
  - Do we really need a hardware-supported "use" bit?

- Discuss in groups of two to three students
  - Each group chooses a leader to summarize the discussion
  - In your group discussion, please do not dominate the discussion, and give everyone a chance to speak

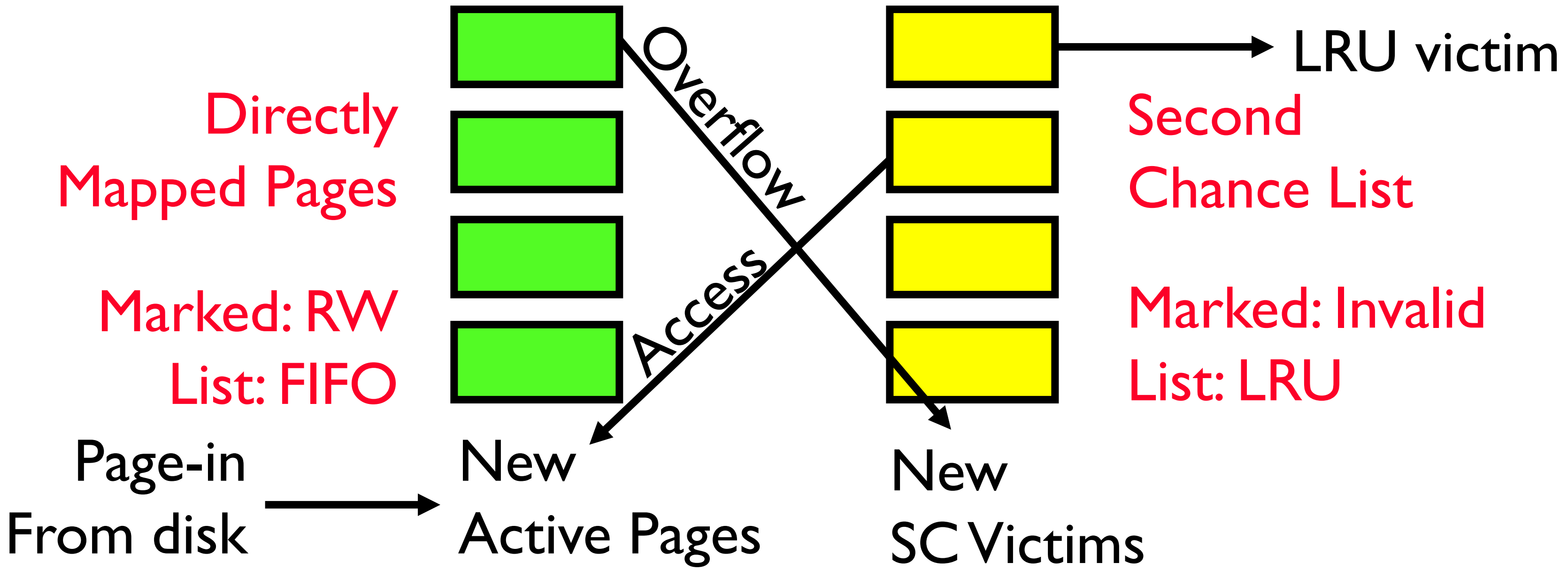
Clock Algorithms Variations

- Do we really need hardware-supported "modified" bit?
  - No. Can emulate it using read-only bit
    - » Need software DB of which pages are allowed to be written (needed this anyway)
    - » We will tell MMU that pages have more restricted permissions than the actually do to force page faults (and allow us notice when page is written)
  - Algorithm (Clock-Emulated-M):
    - » Initially, mark all pages as read-only ($W \rightarrow 0$), even writable data pages.
      Further, clear all software versions of the "modified" bit $\rightarrow$ 0 (page not dirty)
    - » Writes will cause a page fault. Assuming write is allowed, OS sets software "modified" bit $\rightarrow$ 1, and marks page as writable ($W \rightarrow 1$).
    - » Whenever page written back to disk, clear "modified" bit $\rightarrow$ 0, mark read-only

Clock Algorithms Variations (continued)

- Do we really need a hardware-supported "use" bit?
  - No. Can emulate it similar to above (e.g. for read operation)
    - » Kernel keeps a "use" bit and "modified" bit for each page
  - Algorithm (Clock-Emulated-Use-and-M):
    - » Mark all pages as invalid, even if in memory.
      Clear emulated "use" bits → 0 and "modified" bits → 0 for all pages (not used, not dirty)
    - » Read or write to invalid page traps to OS to tell use page has been used
    - » OS sets "use" bit → 1 in software to indicate that page has been "used".
      Further:
      1) If read, mark page as read-only, W→0 (will catch future writes)
      2) If write (and write allowed), set "modified" bit → 1, mark page as writable (W→1)
    - » When clock hand passes, reset emulated "use" bit → 0 and mark page as invalid again
    - » Note that "modified" bit left alone until page written back to disk
- Remember, however, clock is just an approximation of LRU!
  - Can we do a better approximation, given that we have to take page faults on some reads and writes to collect use information?
  - Need to identify an old page, not oldest page!
  - Answer: second chance list

Second-Chance List Algorithm (VAX/VMS)



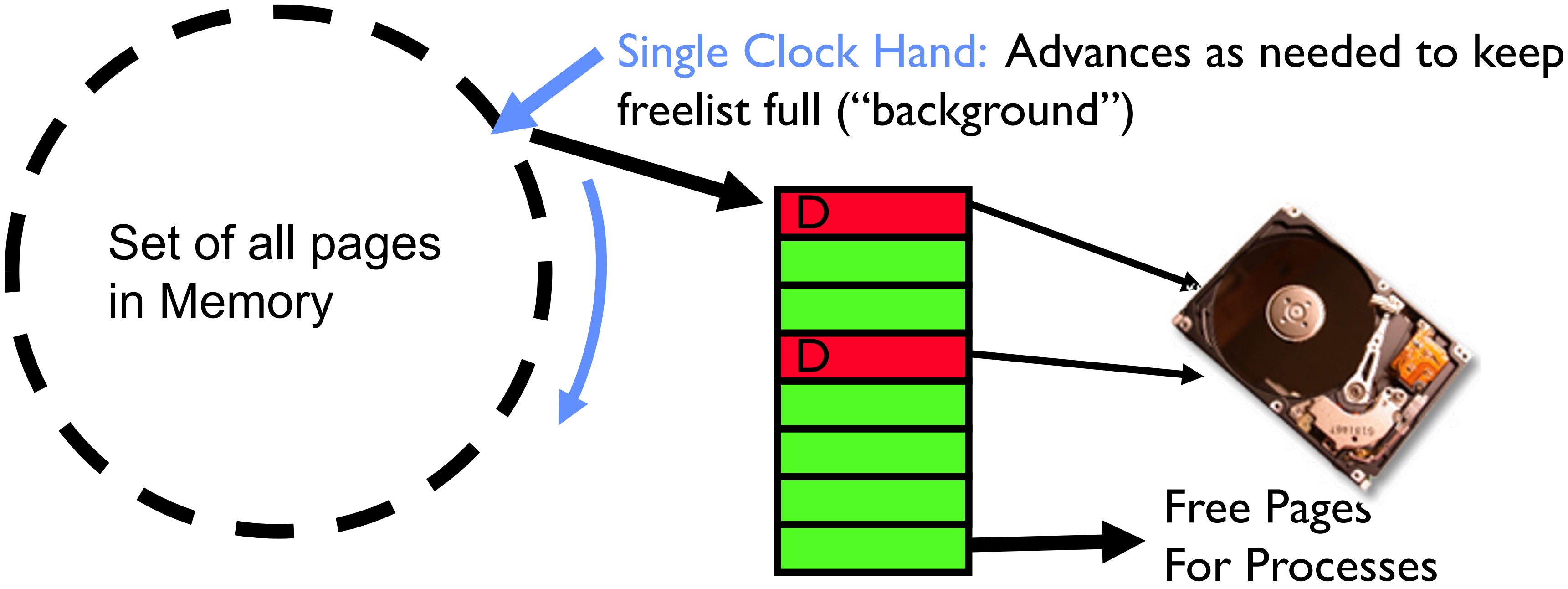
- Split memory in two: Active list (RW), SC list (Invalid)
- Access pages in Active list at full speed
- Otherwise, Page Fault
  - Always move overflow page from end of Active list to front of Second-chance list (SC) and mark invalid
  - Desired Page in SC List: move to it front of Active list, mark it RW
  - Not in SC list: page in to front of Active list, mark RW; page out LRU victim at end of SC list

Second-Chance List Algorithm (continued)

- How many pages for second chance list?
  - If 0 $\Rightarrow$ FIFO
  - If all $\Rightarrow$ LRU, but page fault on every page reference
- Pick intermediate value. Compared to FIFO:
  - Pro: Few disk accesses (page only goes to disk if unused for a long time)
  - Con: Increased overhead trapping to OS (software / hardware tradeoff)

- History: The VAX architecture did not include a "use" bit.
  Why did that omission happen???
  - Strecker (architect) asked OS people, they said they didn't need it, so didn't implement it
  - He later got blamed, but VAX did OK anyway

Free List



- Keep set of free pages ready for use in demand paging
  - Freelist filled in background by Clock algorithm or other technique ("Pageout daemon")
  - Dirty pages start copying back to disk when enter list
- Like VAX second-chance list
  - If page needed before reused, just return to active set
- Advantage: faster for page fault
  - Can always use page (or pages) immediately on fault

Reverse Page Mapping (Sometimes called “Coremap”)

- When evicting a page frame, how to know which PTEs to invalidate?
  - Hard in the presence of shared pages (forked processes, shared memory, …)
- Reverse mapping mechanism must be very fast
  - Must hunt down all page tables pointing at given page frame when freeing a page
  - Must hunt down all PTEs when seeing if pages “active”
- Implementation options:
  - For every page descriptor, keep linked list of page table entries that point to it
    - » Management nightmare – expensive
  - Linux: Object-based reverse mapping
    - » Link together memory region descriptors instead (much coarser granularity)
    - » E.g., program code and files mapped in with mmap()

Allocation of Page Frames (Memory Pages)

- How do we allocate memory among different processes?
  - Does every process get the same fraction of memory? Different fractions?
  - Should we completely swap some processes out of memory?
- Each process needs *minimum* number of pages
  - Want to make sure that all processes that are loaded into memory can make forward progress
  - Example: IBM 370 – 6 pages to handle SS MOVE instruction:
    - » instruction is 6 bytes, might span 2 pages
    - » 2 pages to handle *from*
    - » 2 pages to handle *to*
- Possible Replacement Scopes:
  - Global replacement – process selects replacement frame from set of all frames; one process can take a frame from another
  - Local replacement – each process selects from only its own set of allocated frames

Fixed/Priority Allocation

- Equal allocation (Fixed Scheme):
  - Every process gets same amount of memory
  - Example: 100 frames, 5 processes $\rightarrow$ process gets 20 frames
- Proportional allocation (Fixed Scheme)
  - Allocate according to the size of process
  - Computation proceeds as follows:

    s_i = size of process p_i and $S = \sum s_i$

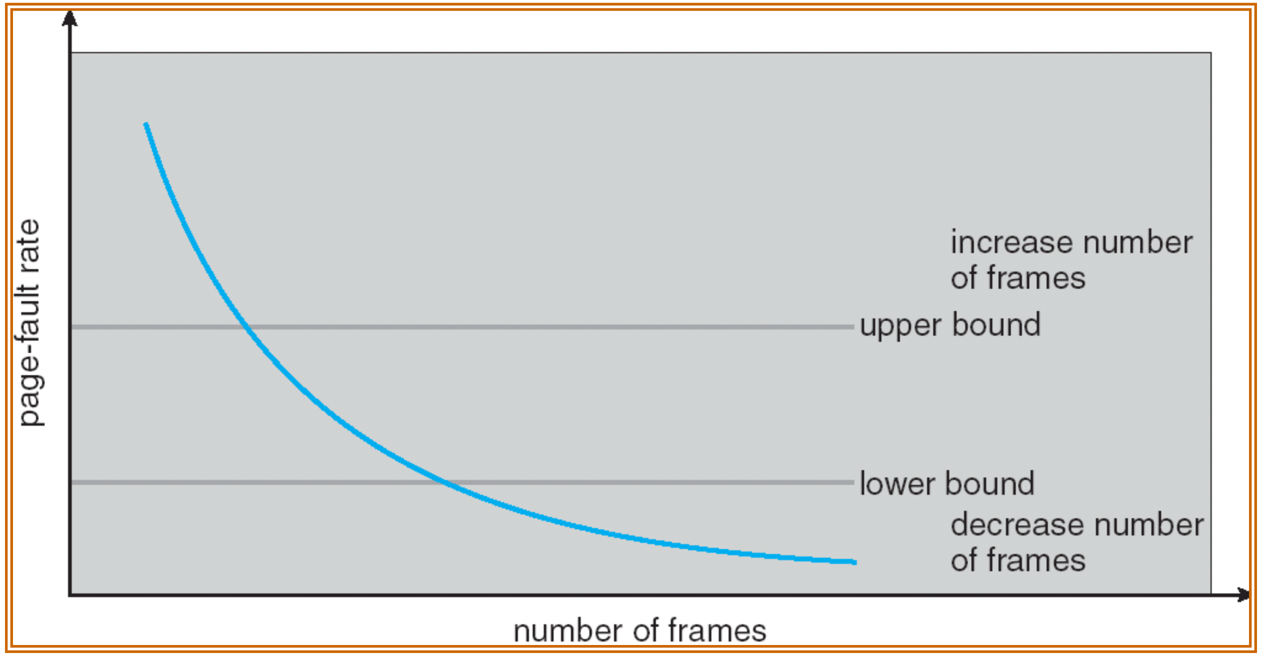
    m = total number of physical frames in the system

    a_i = (allocation for p_i) $= \frac{s_i}{S} \times m$
- Priority Allocation:
  - Proportional scheme using priorities rather than size
    - » Same type of computation as previous scheme
  - Possible behavior: If process p_i generates a page fault, select for replacement a frame from a process with lower priority number

- Perhaps we should use an adaptive scheme instead???
  - What if some application just needs more memory?

Page-Fault Frequency Allocation

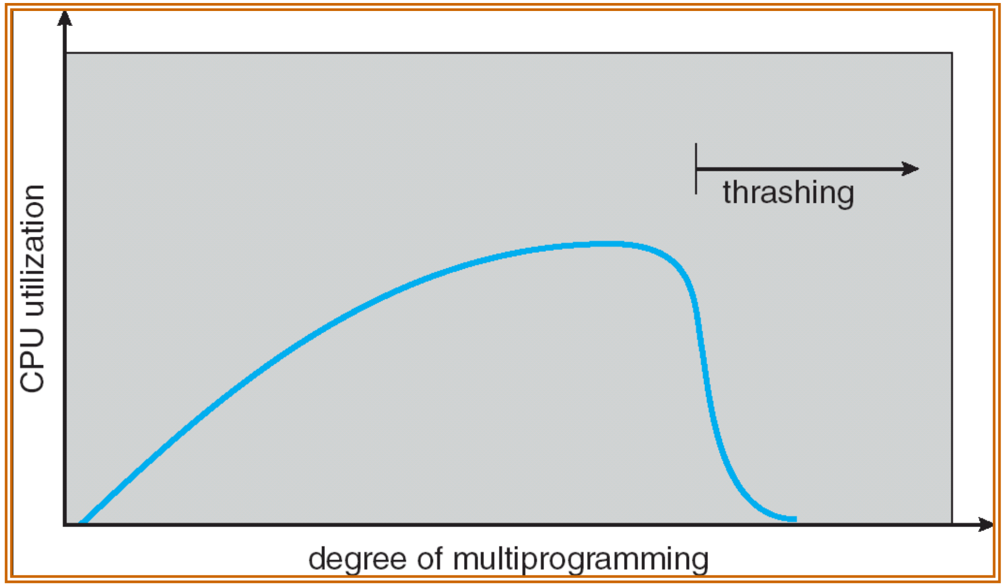
- Can we reduce Capacity misses by dynamically changing the number of pages/application?



- Establish "acceptable" page-fault rate
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame
- Question: What if we just don't have enough memory?

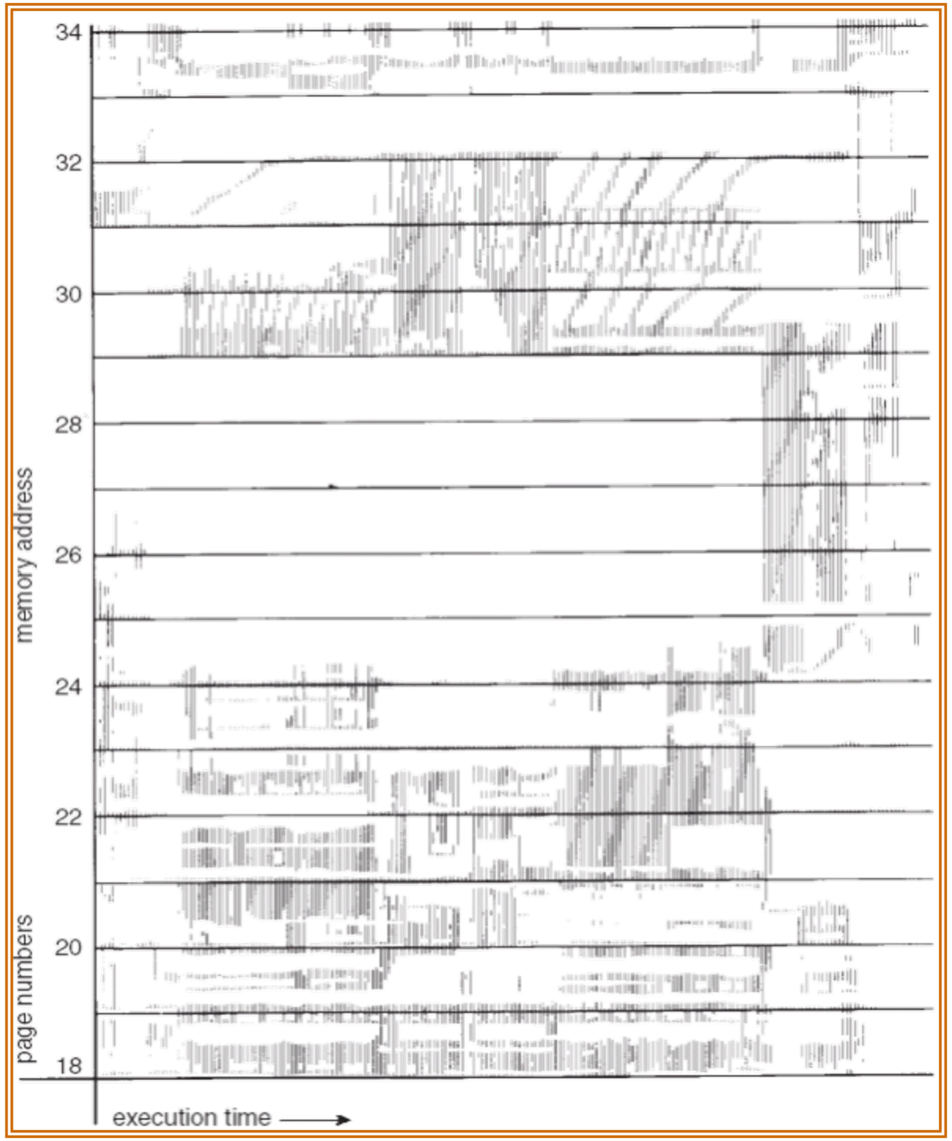
Thrashing

- If a process does not have "enough" pages, the page-fault rate is very high.
  This leads to:
  - low CPU utilization
  - operating system spends most of its time swapping to disk
- Thrashing ≡ a process is busy swapping pages in and out with little or no actual progress
- Questions:
  - How do we detect Thrashing?
  - What is best response to Thrashing?

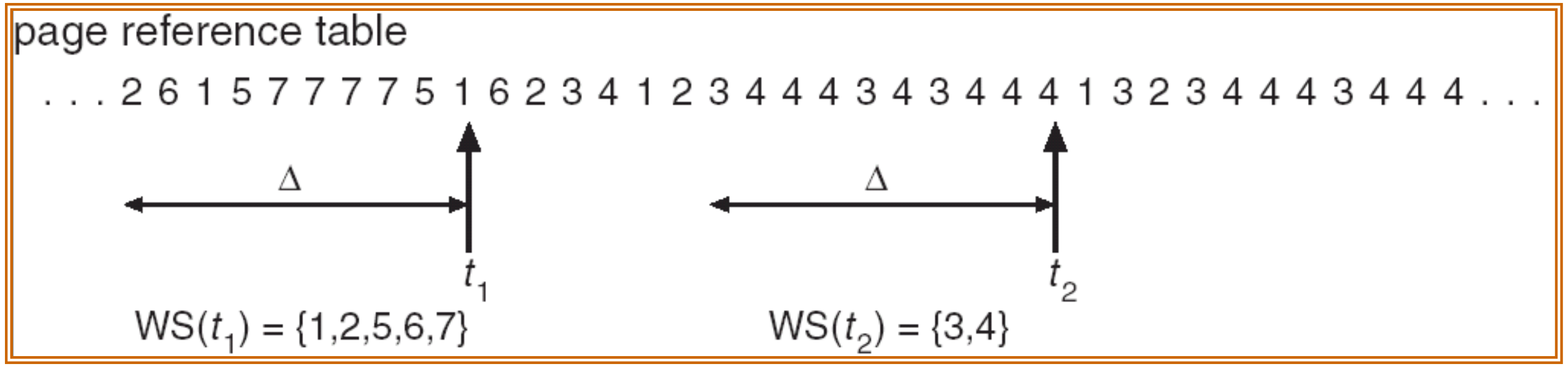


Locality In A Memory-Reference Pattern

- Program Memory Access Patterns have temporal and spatial locality
  - Group of Pages accessed along a given time slice called the "Working Set"
  - Working Set defines minimum number of pages for process to behave well
- Not enough memory for Working Set $\Rightarrow$ Thrashing
  - Better to swap out process?



Working-Set Model



- $\Delta \equiv$ working-set window $\equiv$ fixed number of page references
  - Example: 10,000 instructions
- WSi (working set of Process Pi) = total set of pages referenced in the most recent Δ (varies in time)
  - if Δ too small will not encompass entire locality
  - if Δ too large will encompass several localities
  - if $\Delta = \infty \Rightarrow$ will encompass entire program
- $D = \Sigma|WSi| \equiv$ total demand frames
- if $D > m \Rightarrow$ Thrashing
  - Policy: if $D > m$, then suspend/swap out processes
  - This can improve overall system behavior by a lot!

What about Compulsory Misses?

- Recall that compulsory misses are misses that occur the first time that a page is seen
  – Pages that are touched for the first time
  – Pages that are touched after process is swapped out/swapped back in
- Clustering:
  – On a page-fault, bring in multiple pages “around” the faulting page
  – Since efficiency of disk reads increases with sequential reads, makes sense to read several sequential pages
- Working Set Tracking:
  – Use algorithm to try to track working set of application
  – When swapping process back in, swap in working set

Summary

- Second-Chance List algorithm: Yet another approximate LRU
  - Divide pages into two groups, one of which is truly LRU and managed on page faults.
- Working Set:
  - Set of pages touched by a process recently
- Thrashing: a process is busy swapping pages in and out
  - Process will thrash if working set doesn’t fit in memory
  - Need to swap out a process

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

Microsoft Research

FaRM: Fast Remote Memory

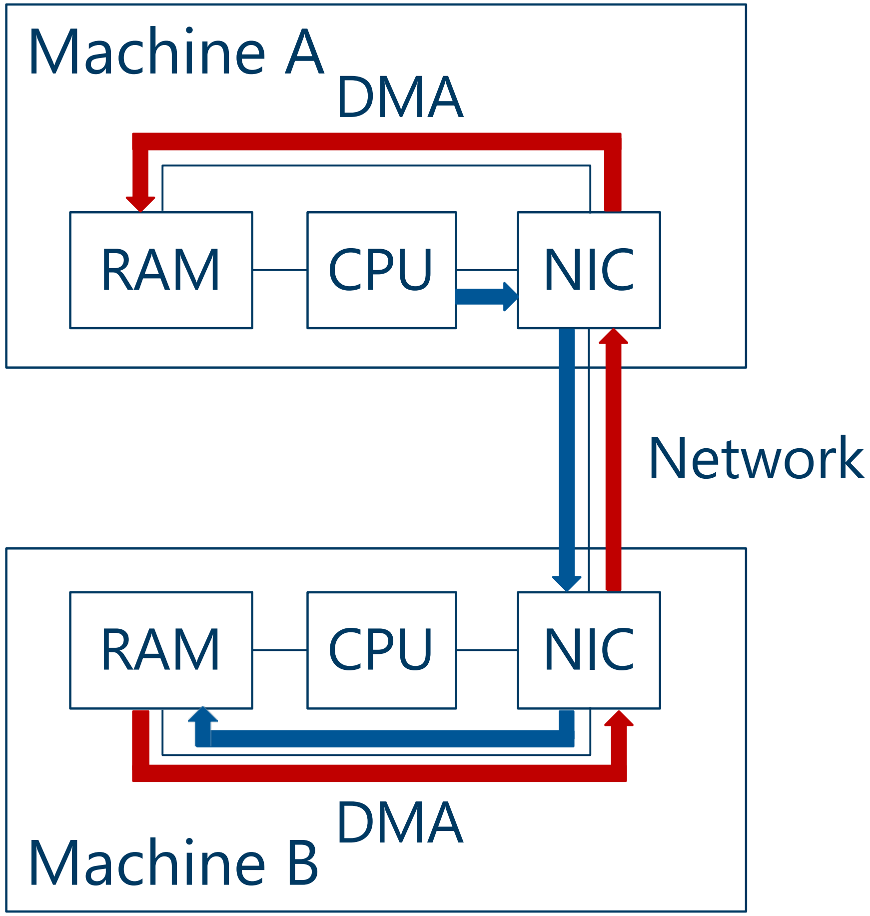
Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, Miguel Castro

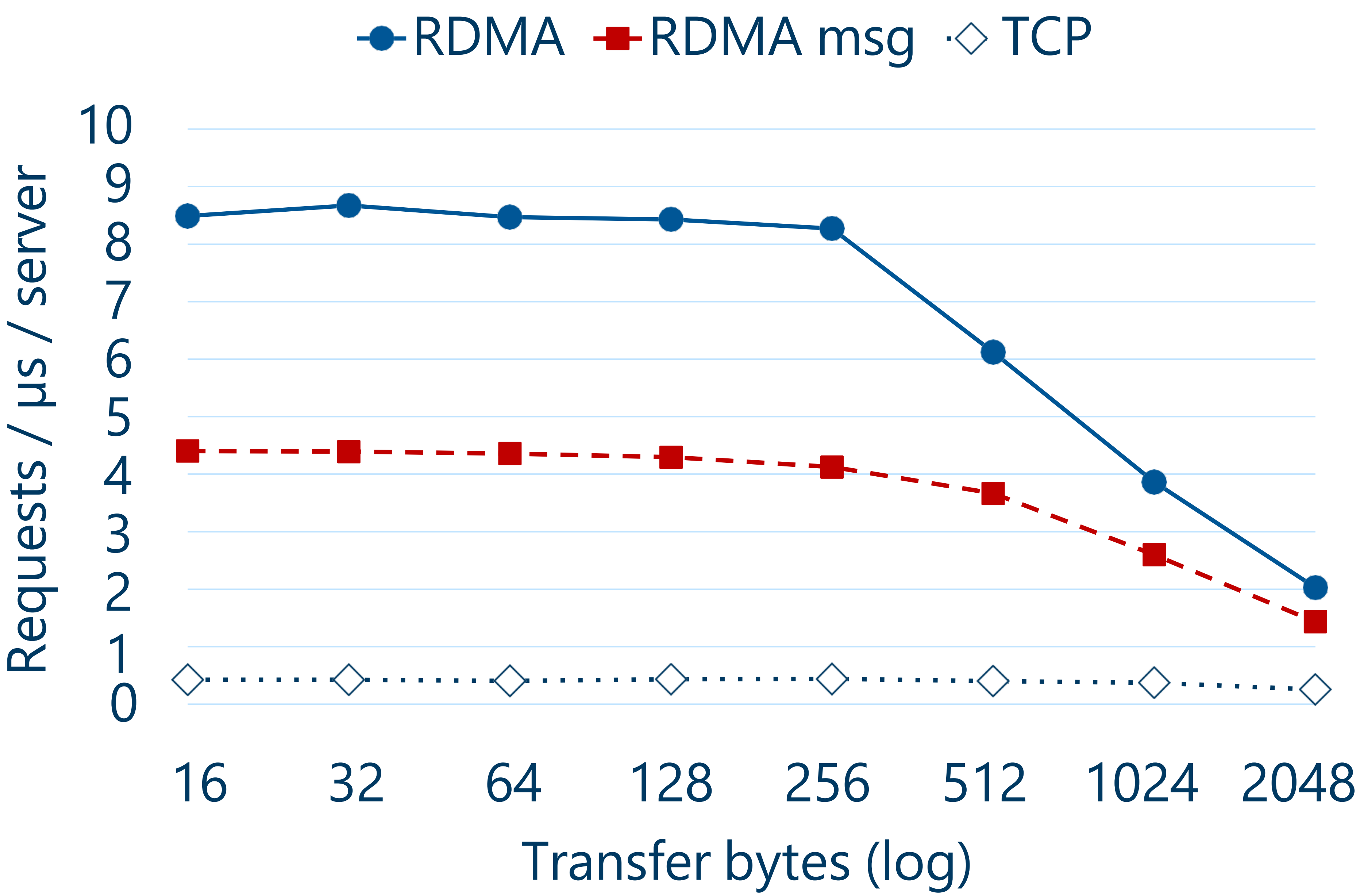
Hardware trends

- 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 μs latency
  - RDMA primitives

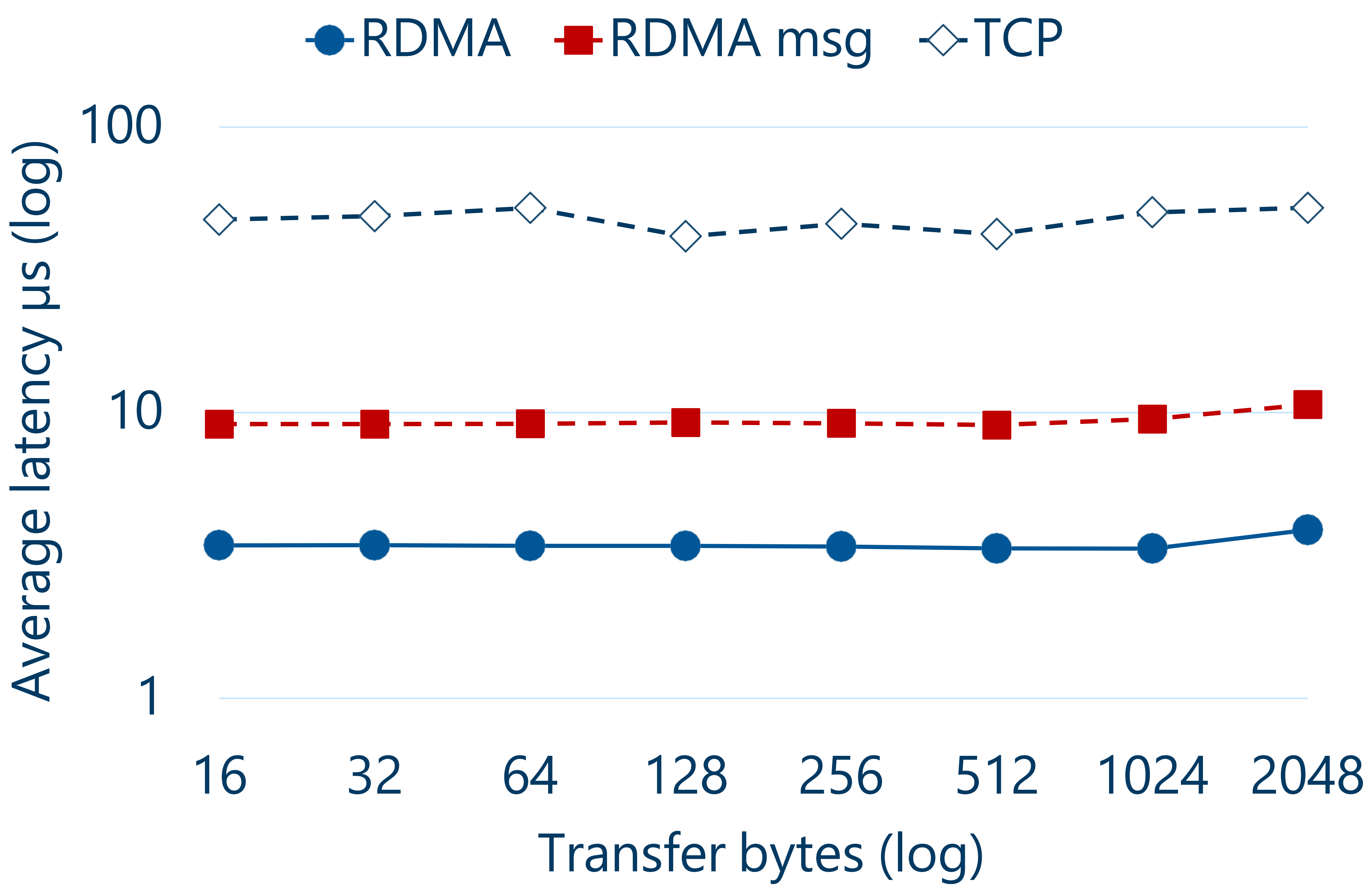
Remote direct memory access

- 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





RDMA
RDMA msg
TCP
Requests / µs / server
10
9
8
7
6
5
4
3
2
1
0
16
32
64
128
256
512
1024
2048
Transfer bytes (log)



RDMA
RDMA msg
TCP
Average latency μs (log)
100
10
1
16
32
64
128
256
512
1024
2048
Transfer bytes (log)

Applications

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

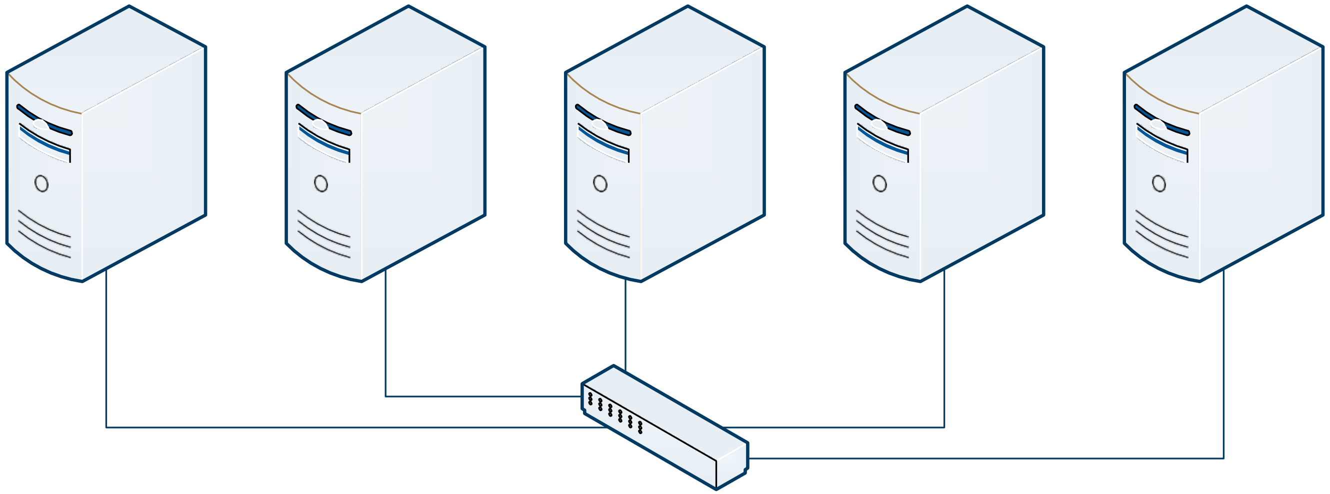
How to program a modern cluster?

We have:

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

Desirable:

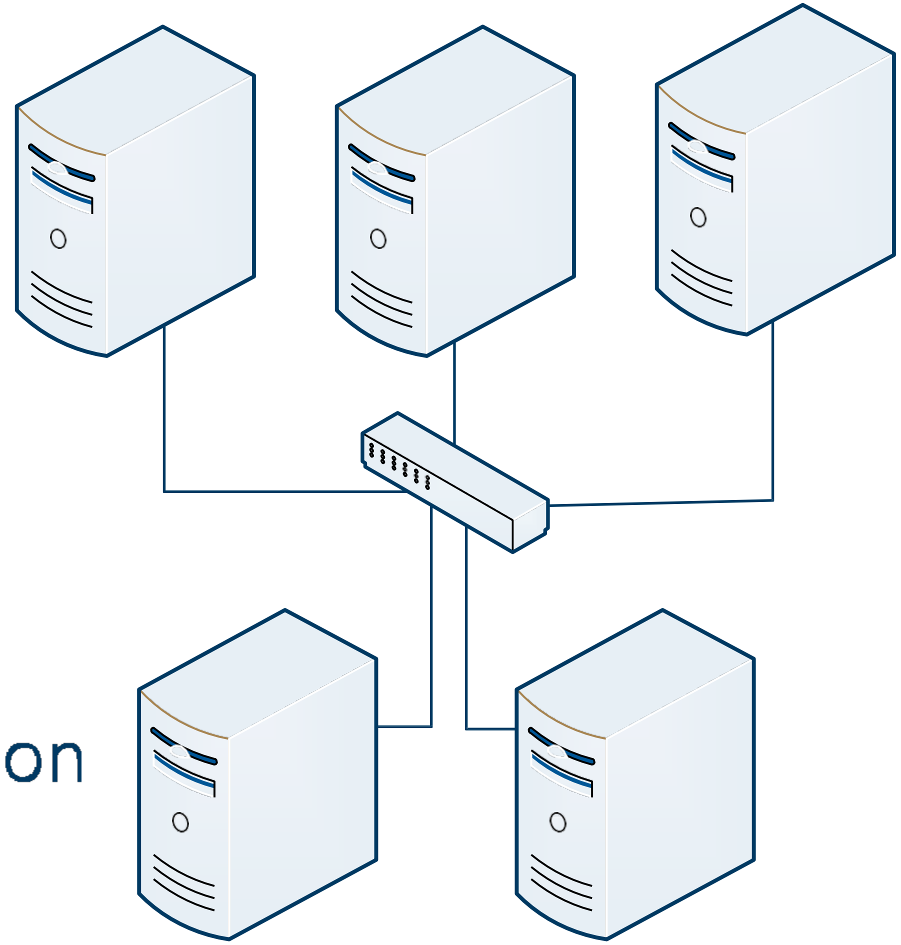
- Keep data in memory
- Access data using RDMA
- Collocate data and computation



Traditional model

Servers: store data

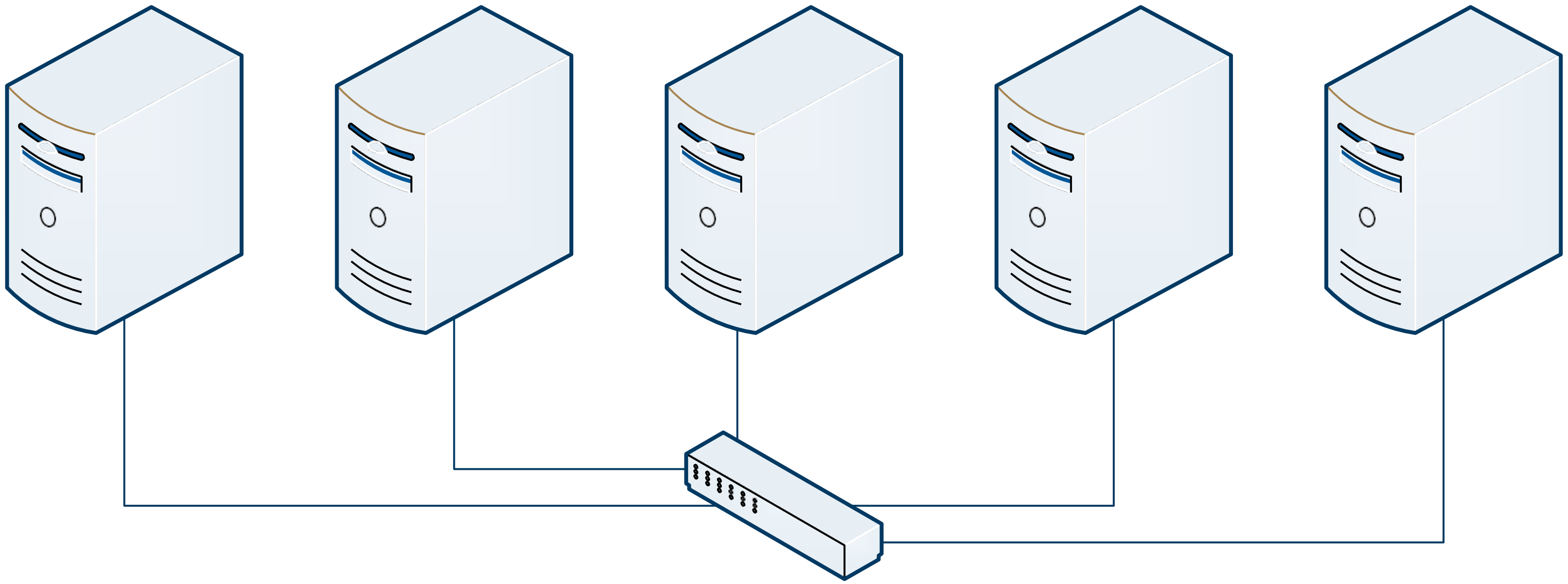
Clients: execute application



Symmetric model

Access to local memory is much faster

Server CPUs are mostly idle with RDMA

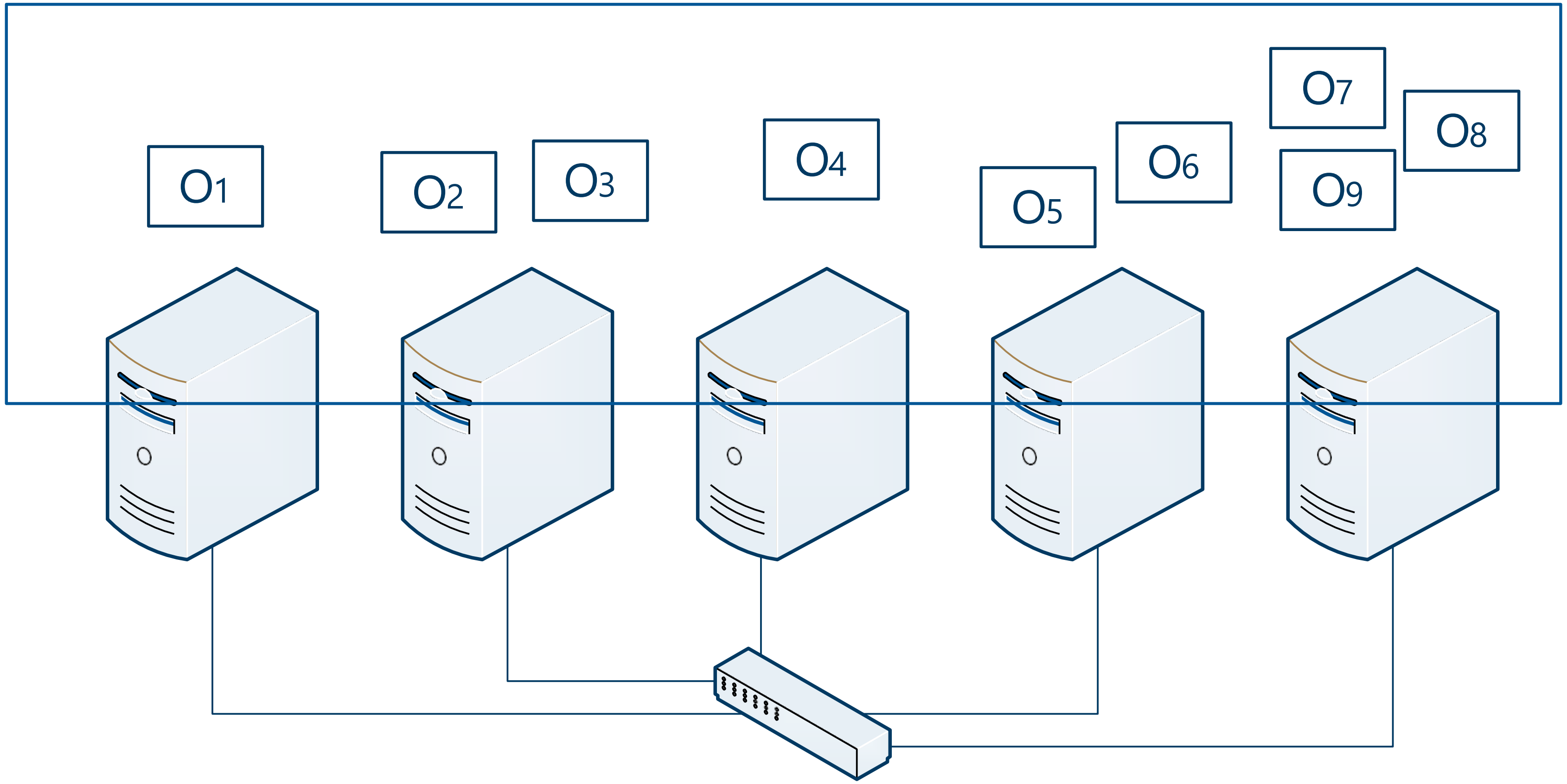


Machines store data and execute application

Shared address space

Supports direct RDMA of objects

Programmability a welcome bonus

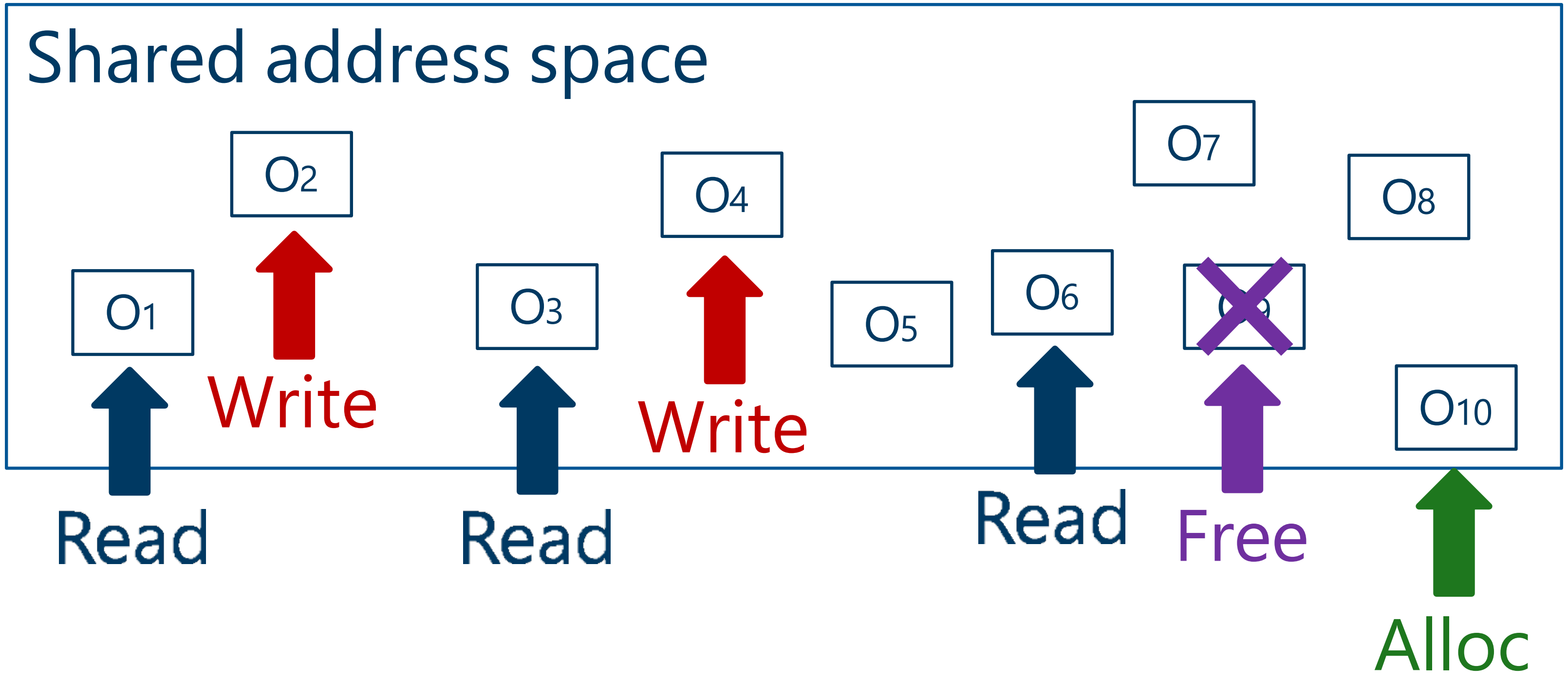


Shared address space

General primitive

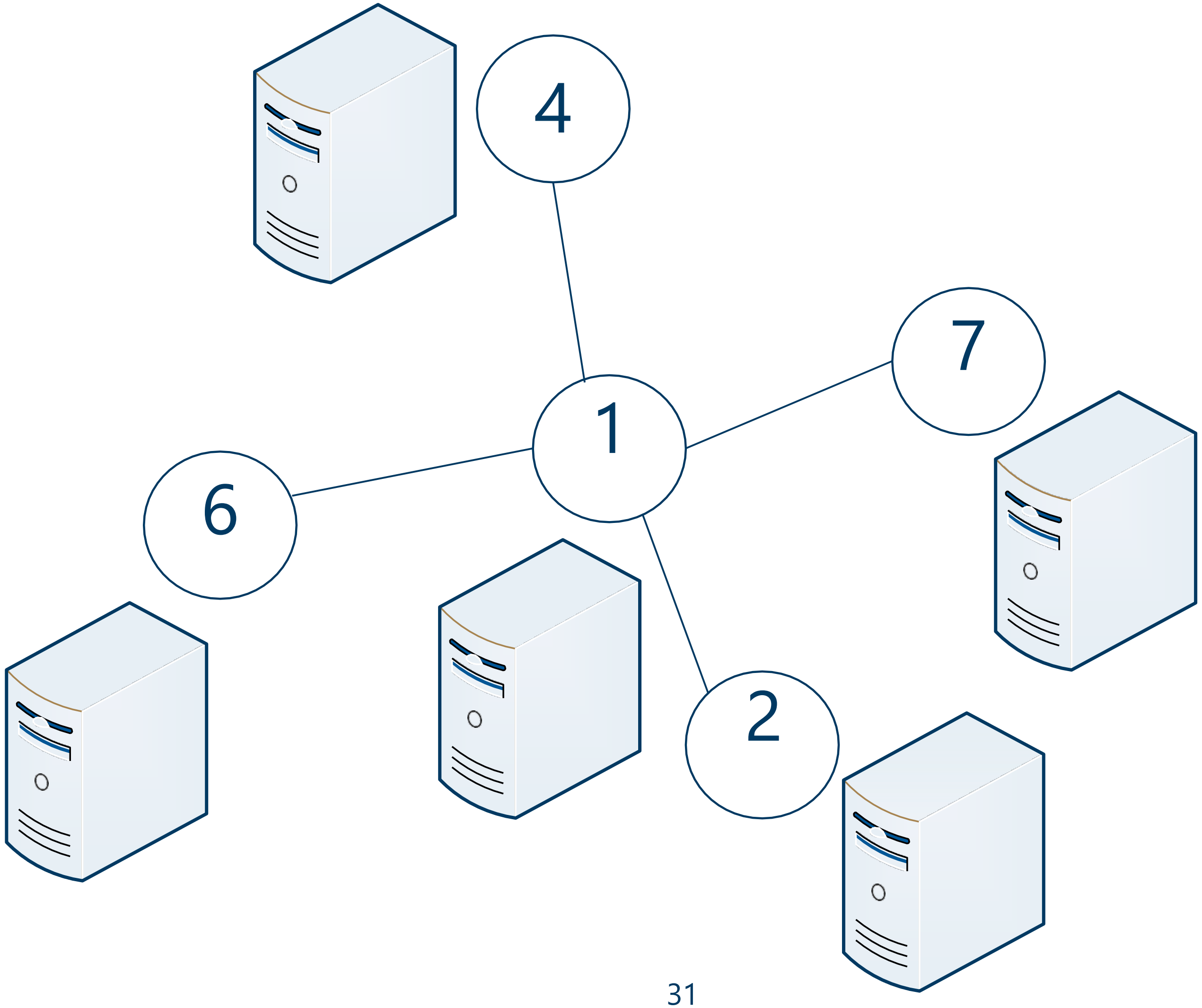
Strong consistency: serializability

Transparent:
- location
- concurrency
- failures



Atomic execution of multiple operations

Optimizations: locality awareness

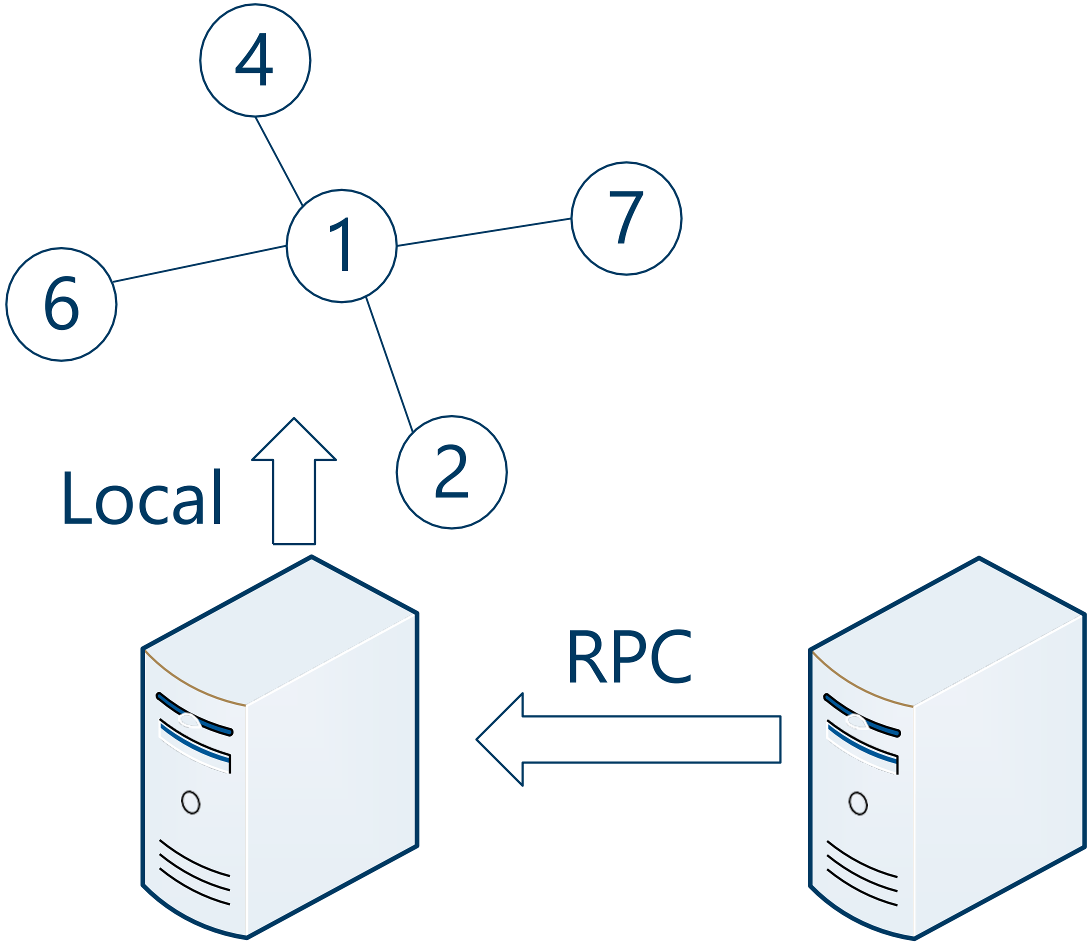


Optimizations: locality awareness

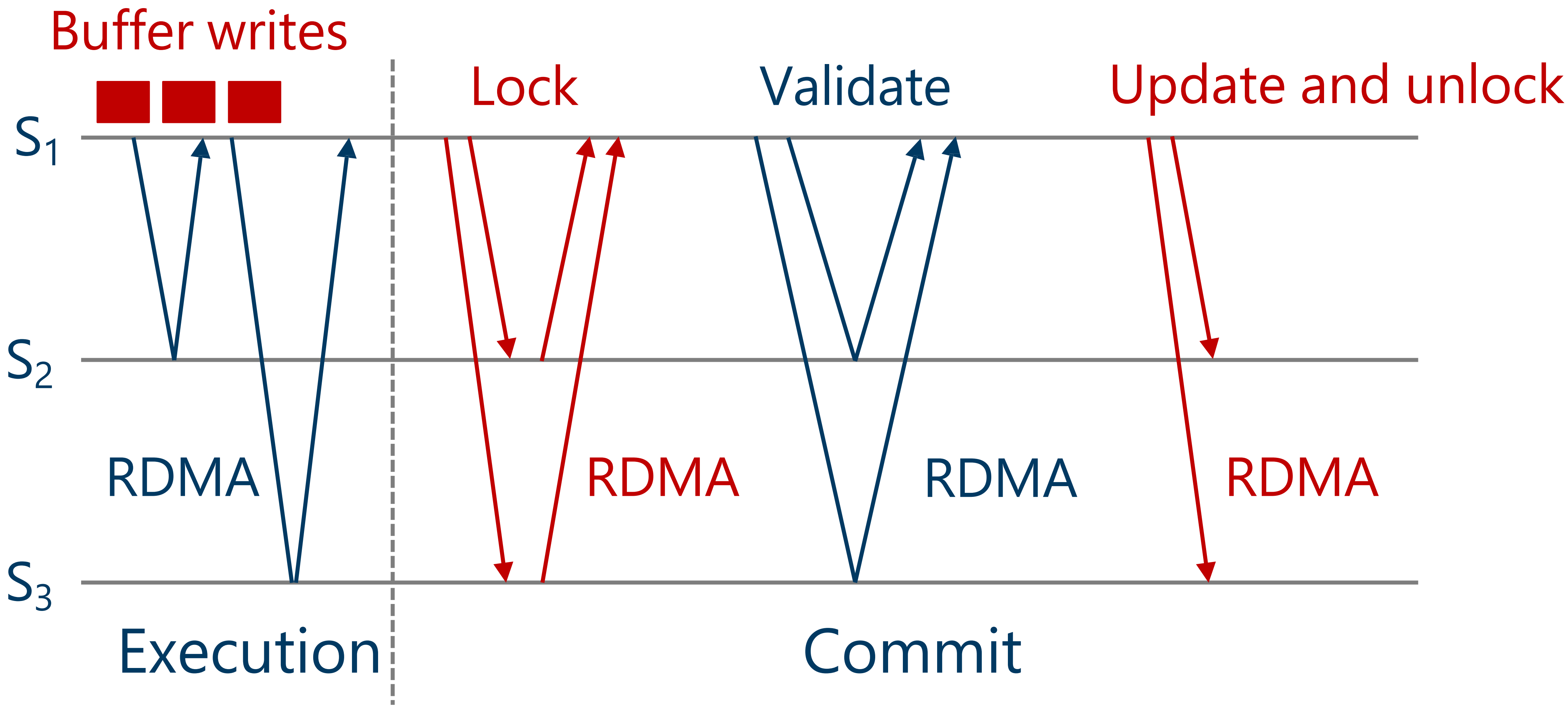
Collocate data accessed together

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 μs 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

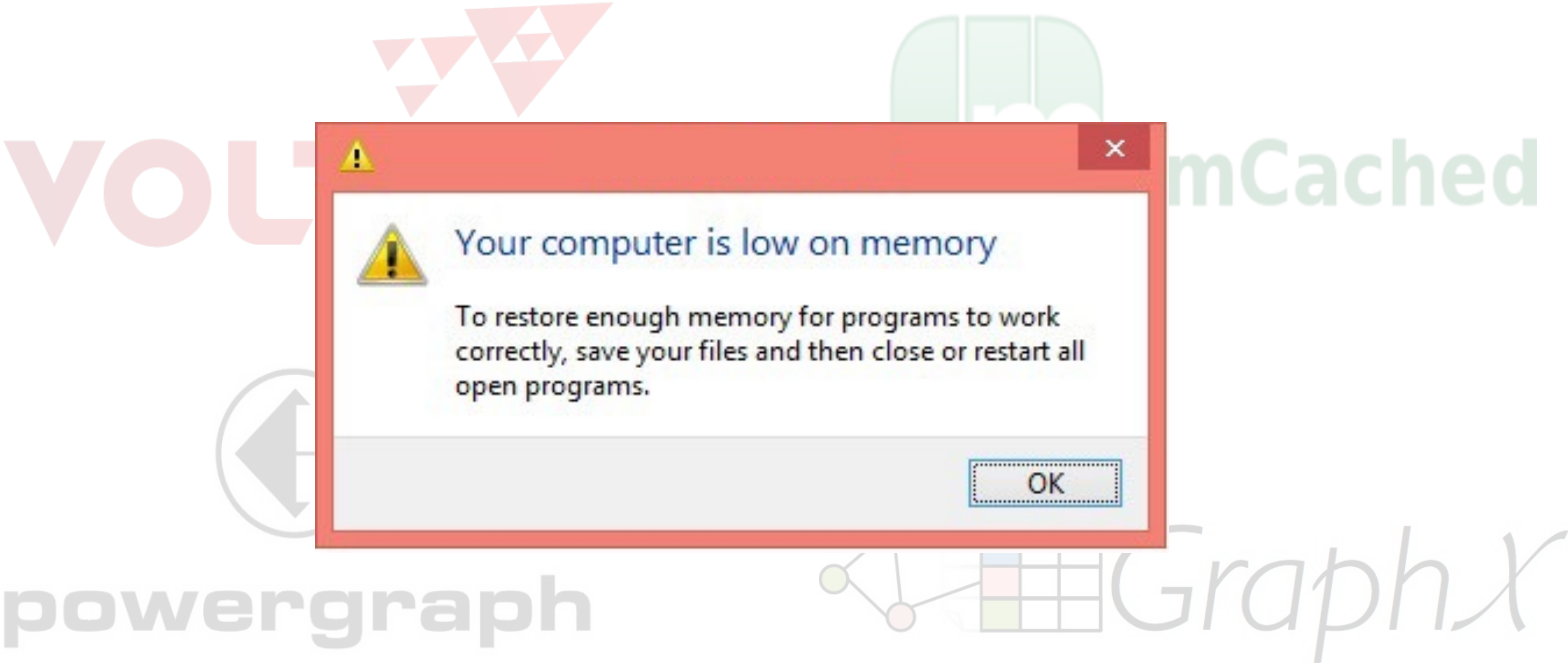
UNIVERSITY OF MICHIGAN

Agenda

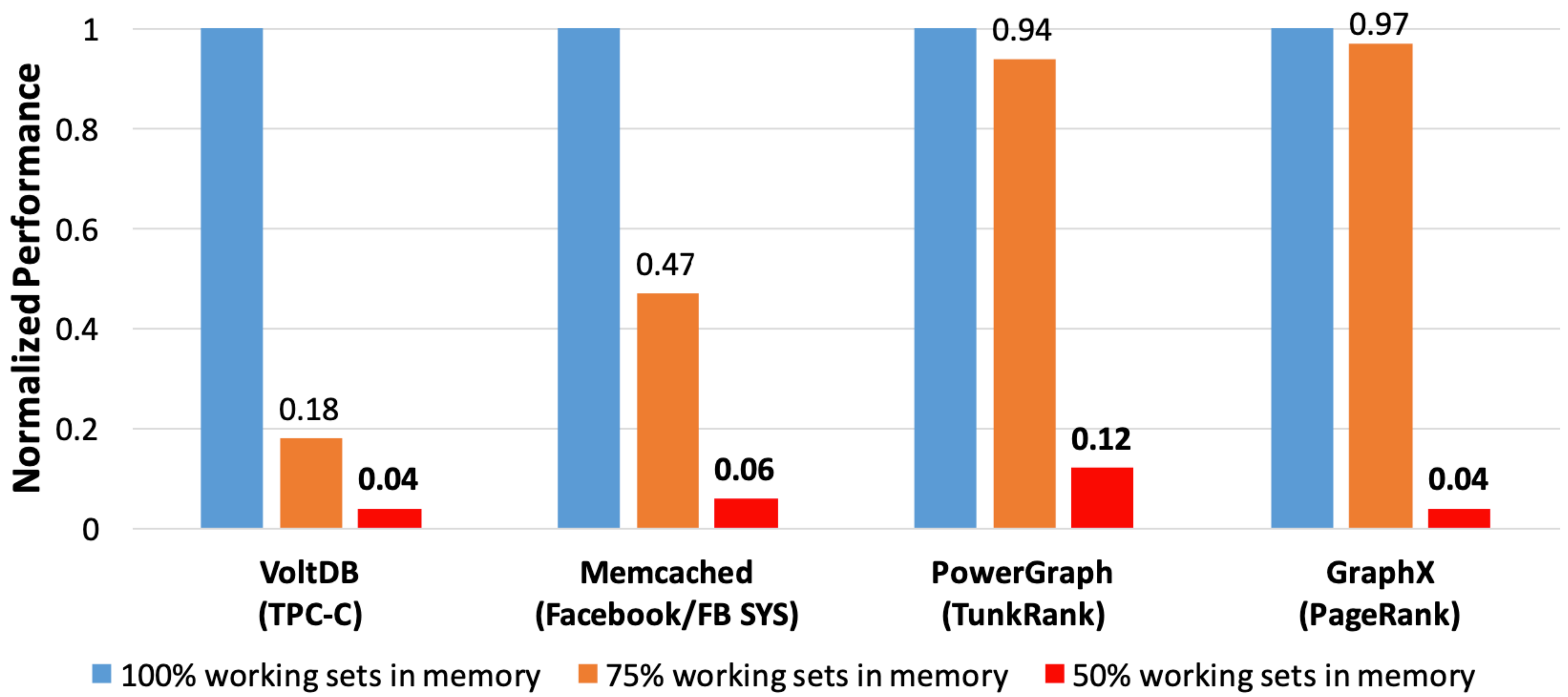
- **Motivation and related work**
- Design and system overview
- Implementation and evaluation
- Future work and conclusion

Memory-intensive applications

Memory-intensive applications

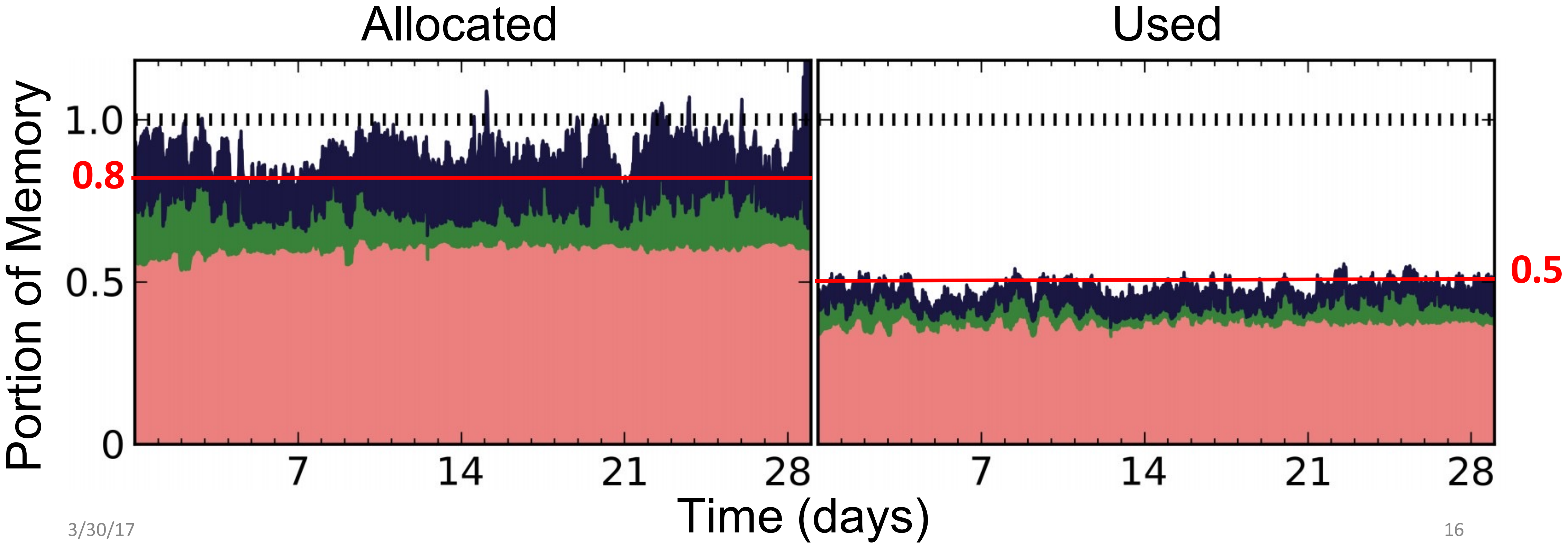


Performance degradation



Memory underutilization

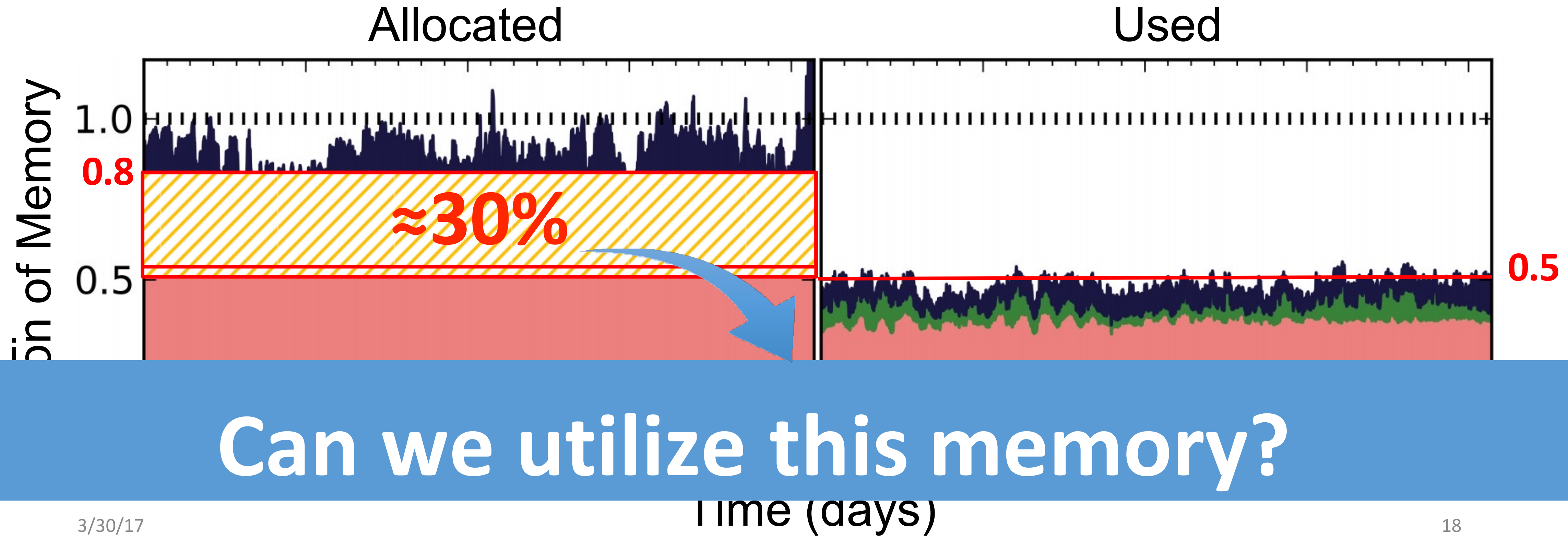
- Google Cluster Analysis[1]



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

Memory underutilization

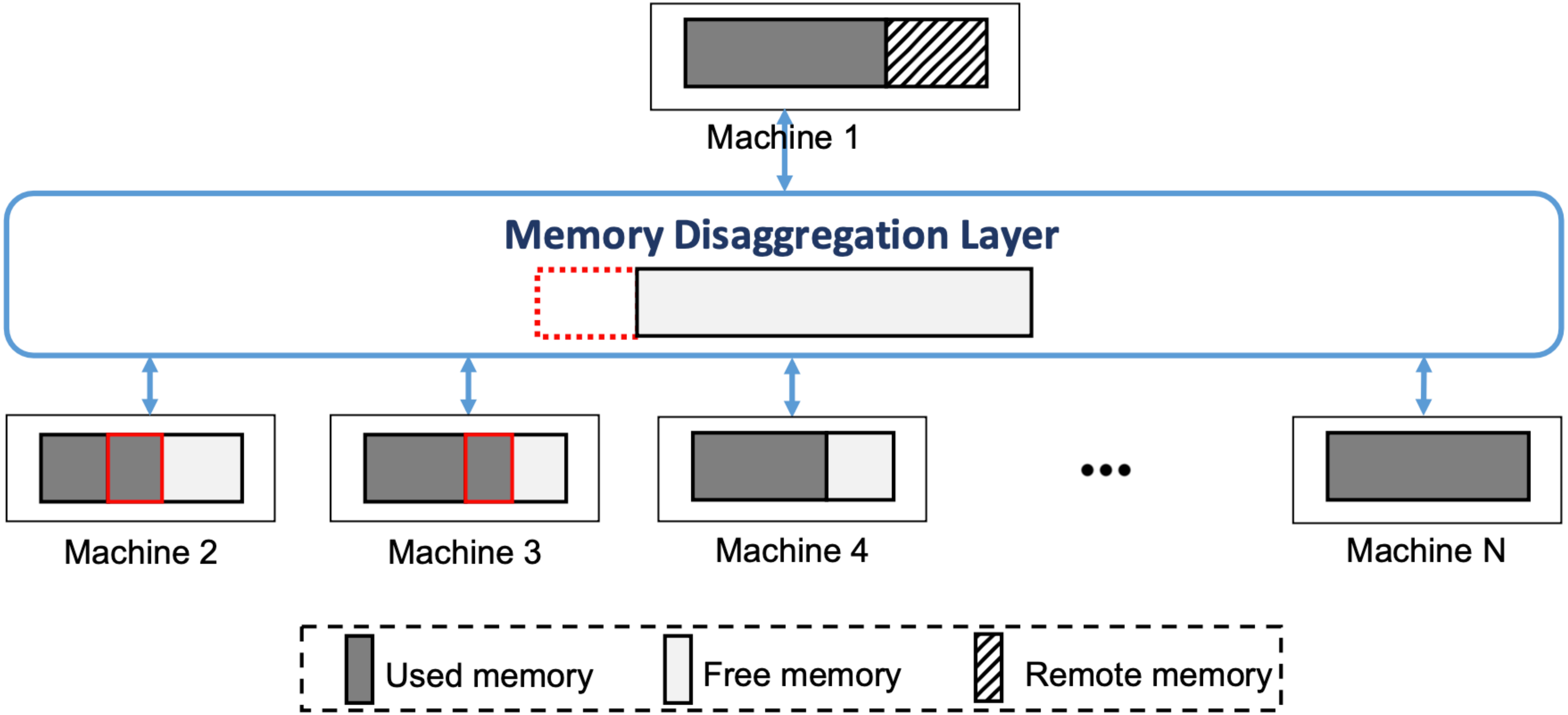
- Google Cluster Analysis[1]



Can we utilize this memory?

[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

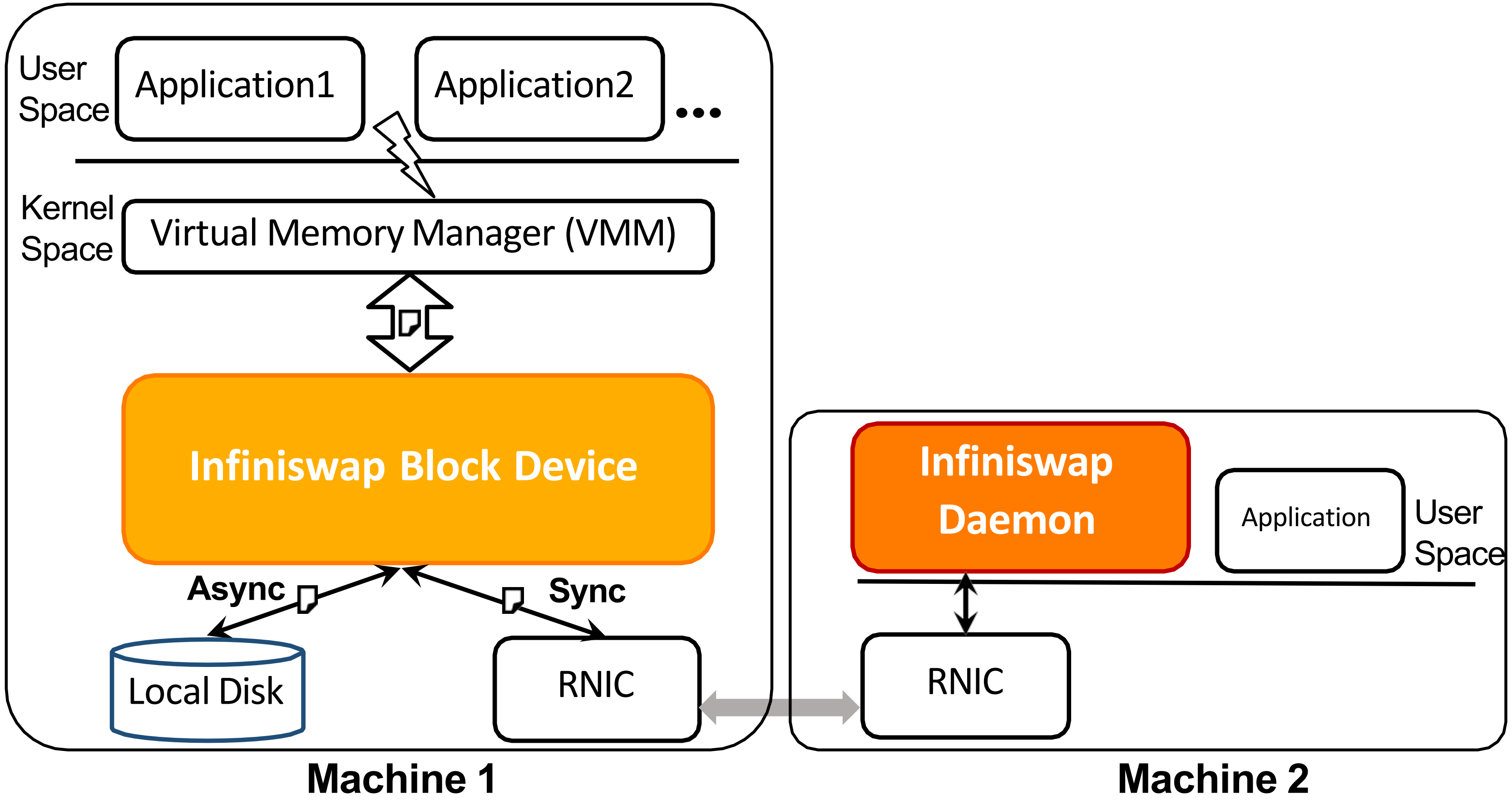
	No HW design	No app modification	Fault-tolerance	Scalability
Memory Blade[ISCA'09]	✗	✓	✓	✓
HPBD[CLUSTER'05] / **NBDX**[1]	✓	✓	✗	✗
RDMA key-value service (e.g. HERD[SIGCOMM'14], FaRM[NSDI'14])	✓	✗	✓	✓
Intel Rack Scale Architecture (RSA)[2]	✗	✓	✓	✓
Infiniswap	✓	✓	✓	✓

1 https://github.com/accelio/NBDX
2 http://www.intel.com/content/www/us/en/architecture-and-technology/rack-scale-design-overview.html

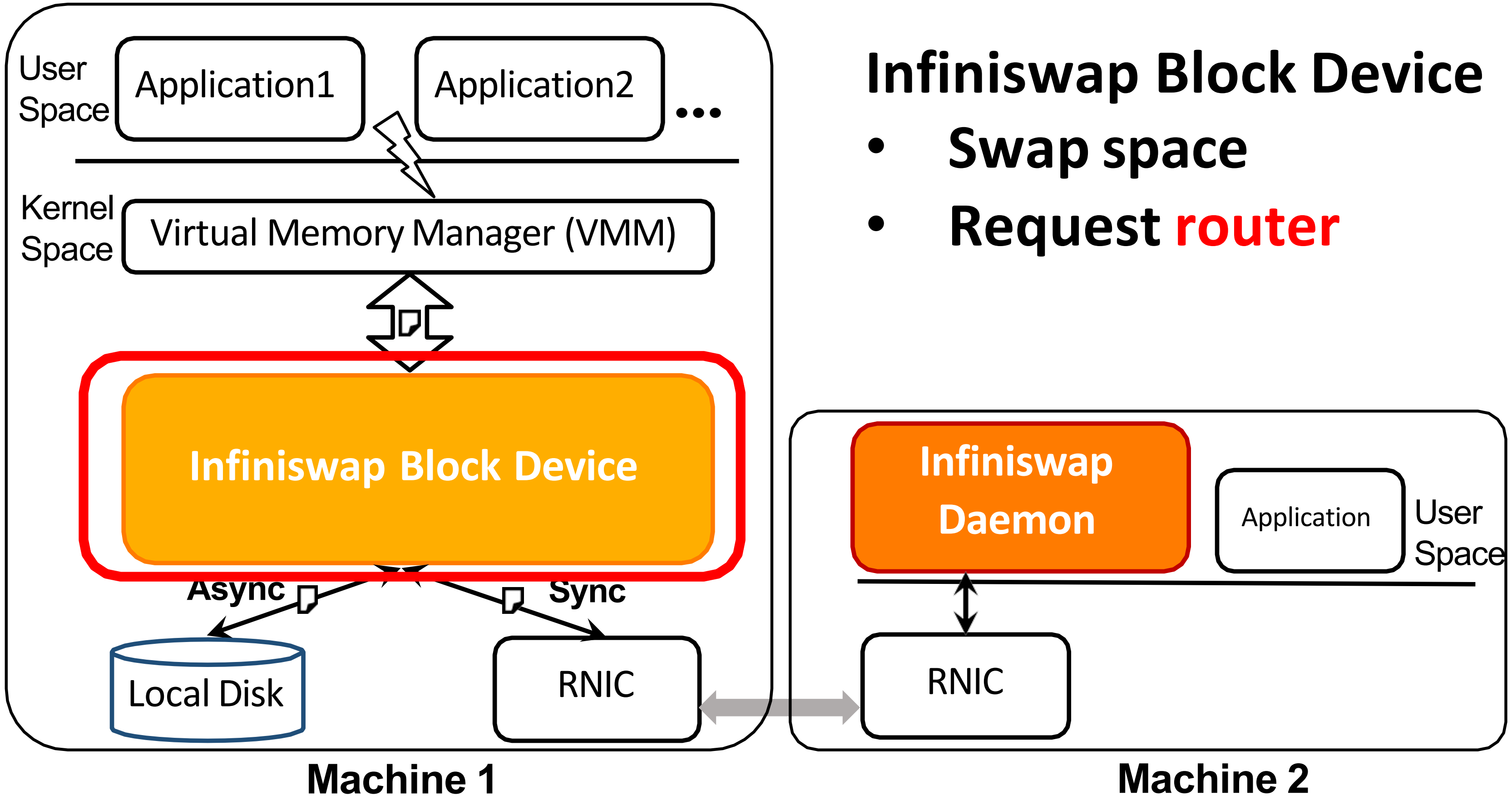
Agenda

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

System Overview



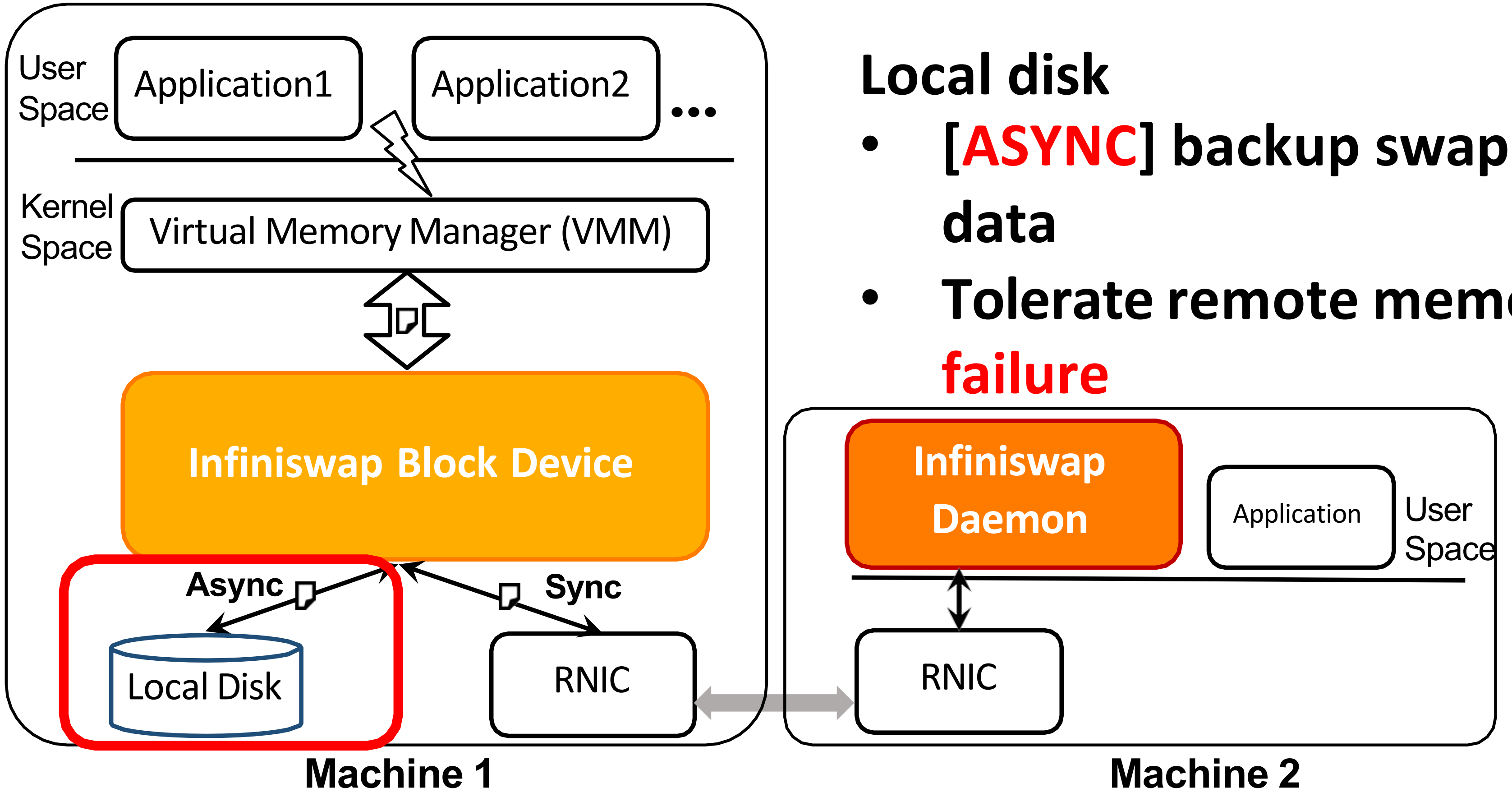
System Overview



Infiniswap Block Device

- **Swap space**
- **Request router**

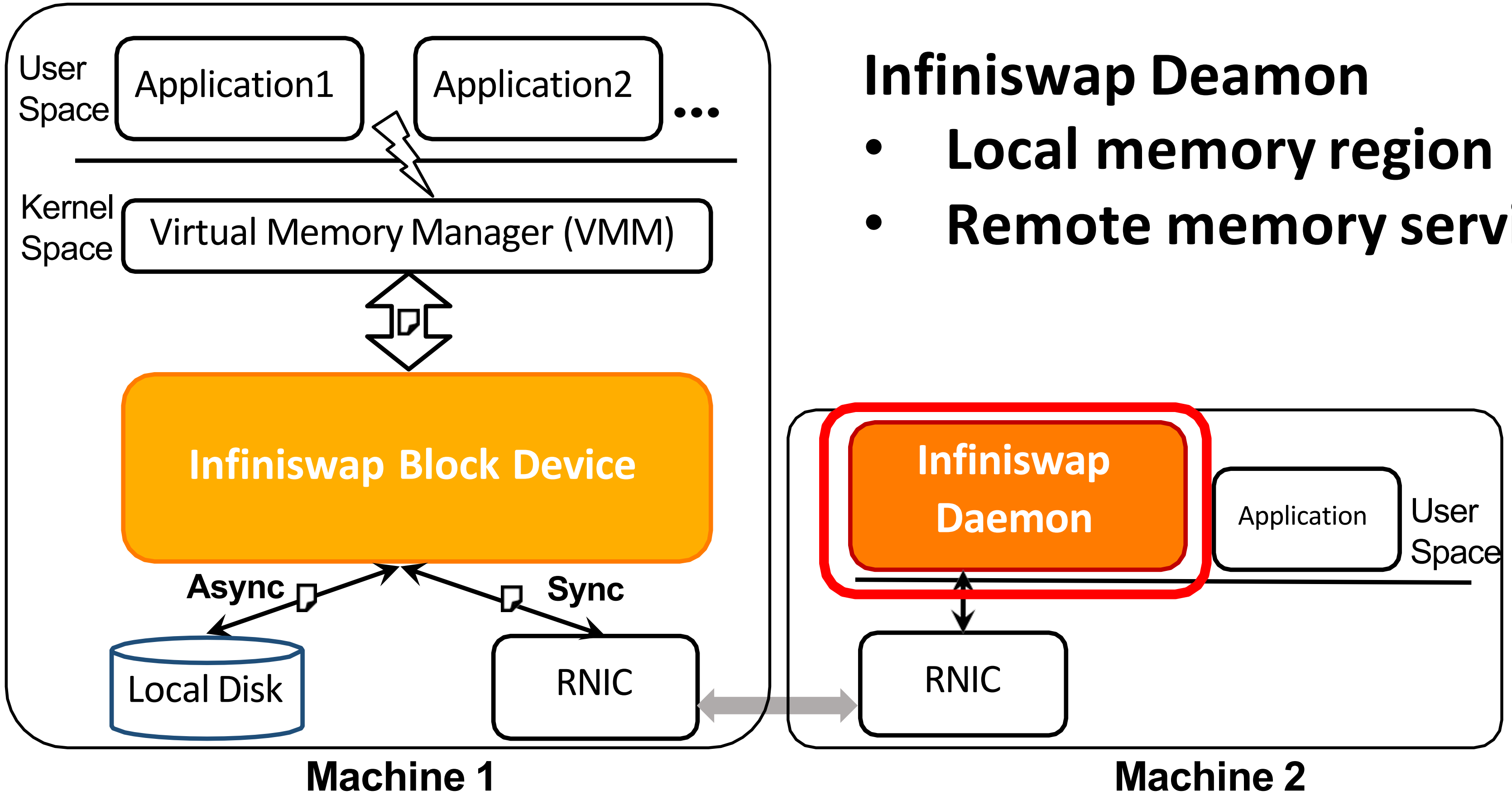
System Overview



Local disk

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

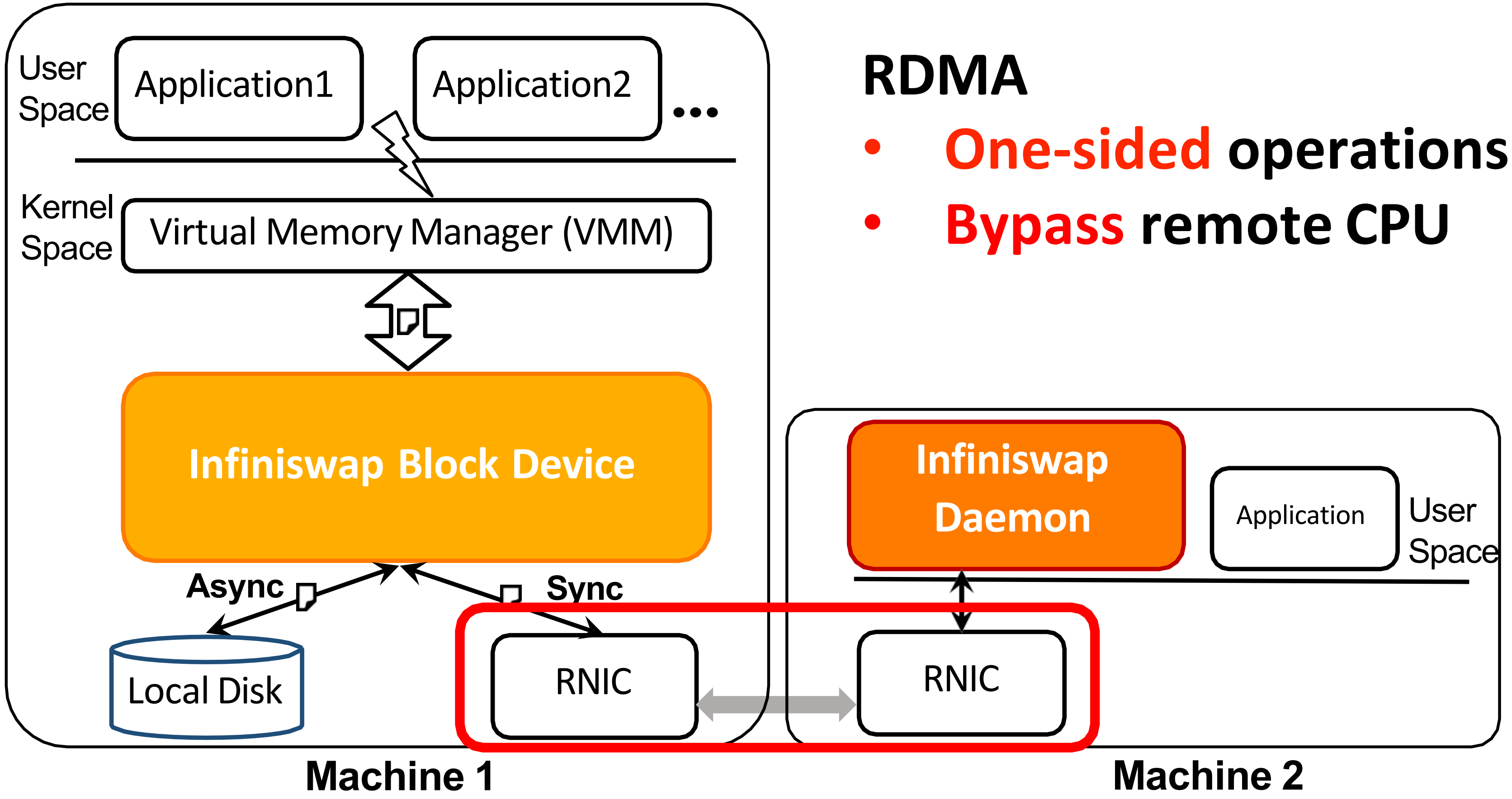
System Overview



Infiniswap Deamon

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

System Overview



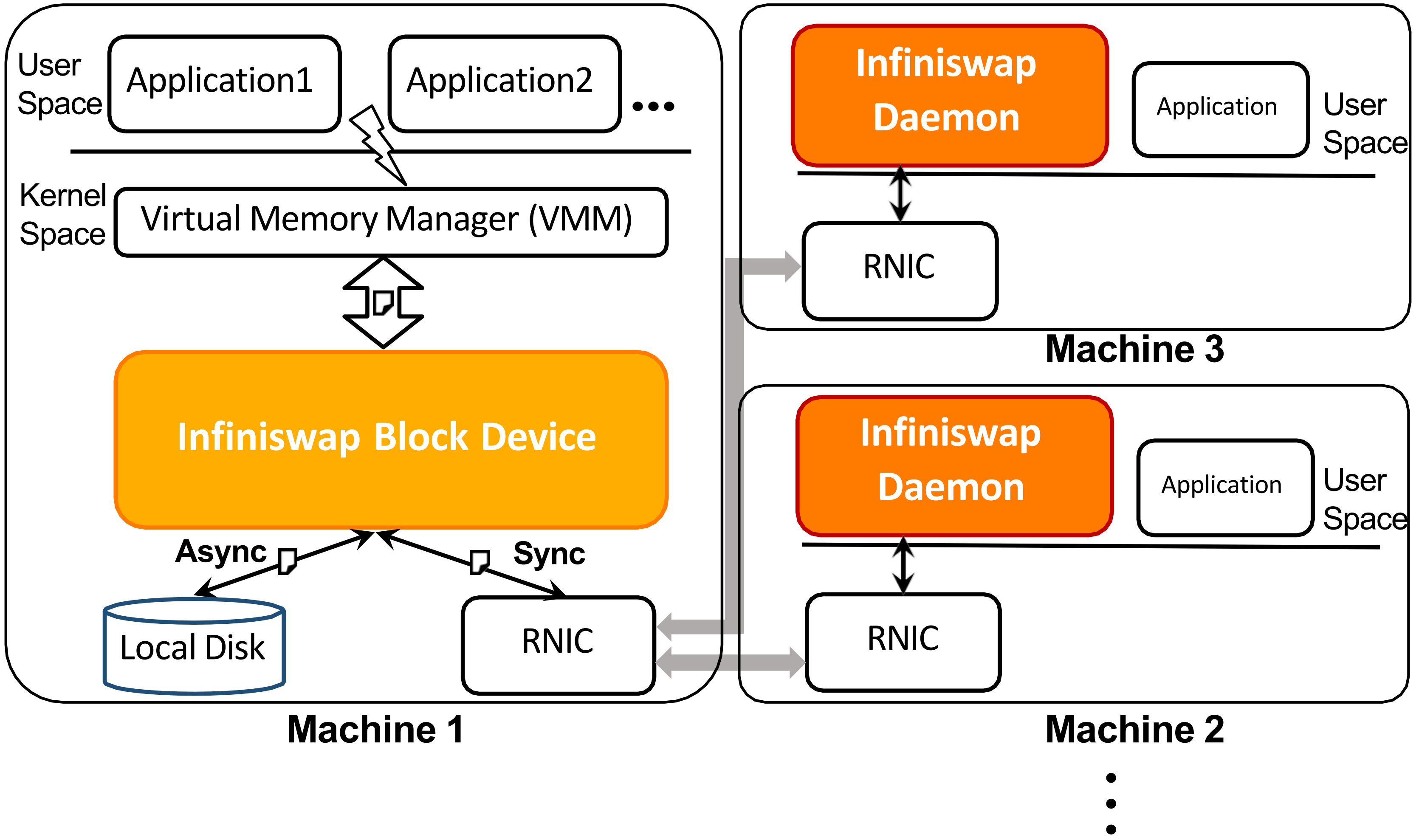
RDMA

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

How to meet the design objectives?

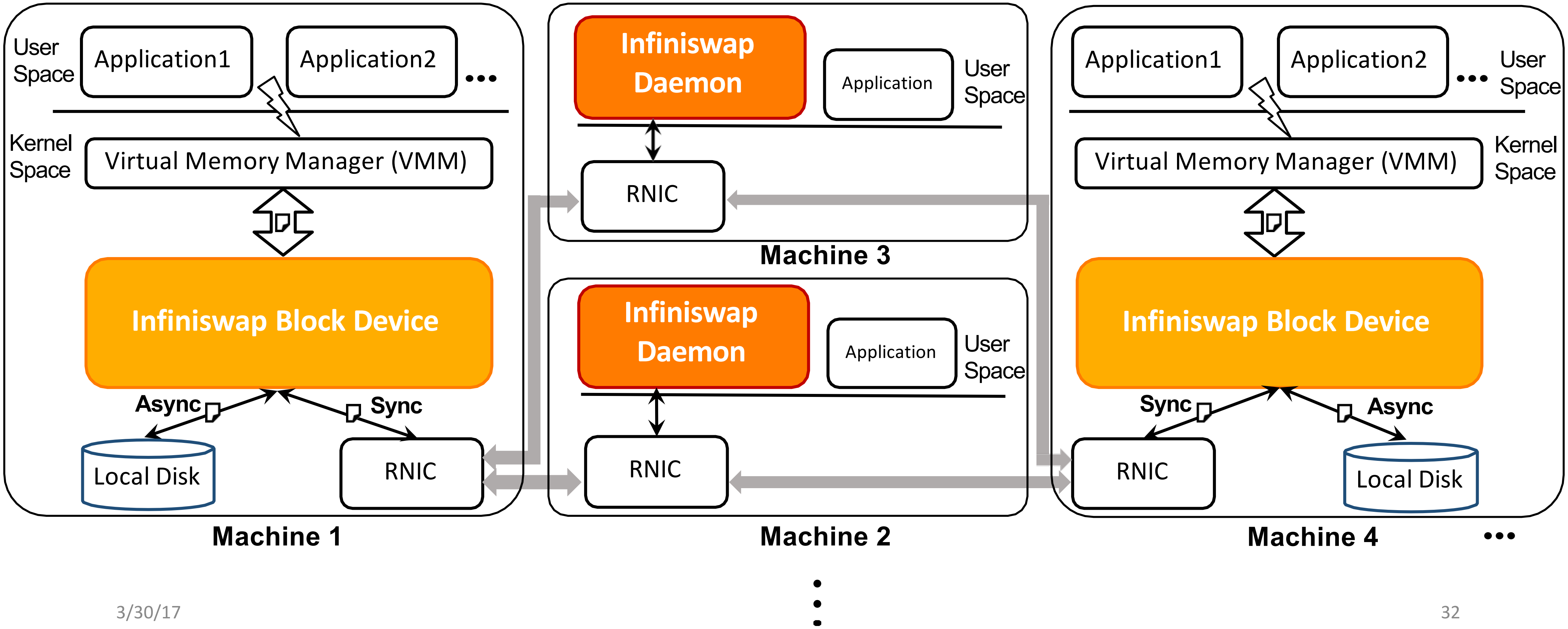
Objectives	Ideas
No hardware design	Remote paging
No application modification	
Fault-tolerance	Local backup disk

One-to-many



User Space
Application1
Application2
...
Kernel Space
Virtual Memory Manager (VMM)
Infiniswap Block Device
Async
Sync
Local Disk
RNIC
Machine 1
Infiniswap Daemon
Application
User Space
RNIC
Machine 3
Infiniswap Daemon
Application
User Space
RNIC
Machine 2

Many-to-many



User Space
Application1
Application2
Kernel Space
Virtual Memory Manager (VMM)
Infiniswap Block Device
Async
Sync
Local Disk
RNIC
Machine 1
Infiniswap Daemon
Application
User Space
RNIC
Machine 3
Infiniswap Daemon
Application
User Space
RNIC
Machine 2
Application1
Application2
User Space
Virtual Memory Manager (VMM)
Kernel Space
Infiniswap Block Device
Sync
Async
RNIC
Local Disk
Machine 4

Many-to-many

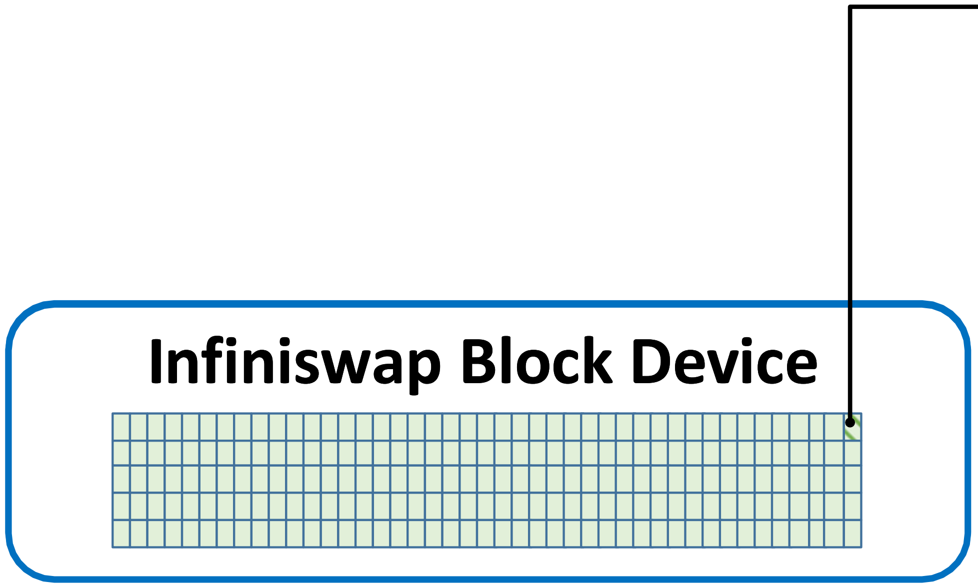
How to scale remote memory?

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

How to meet the design objectives?

Objectives	Ideas
No hardware design	Remote paging
No application modification	
Fault-tolerance	Local backup disk
Scalability	**Decentralized** remote memory management

Management unit: memory page?

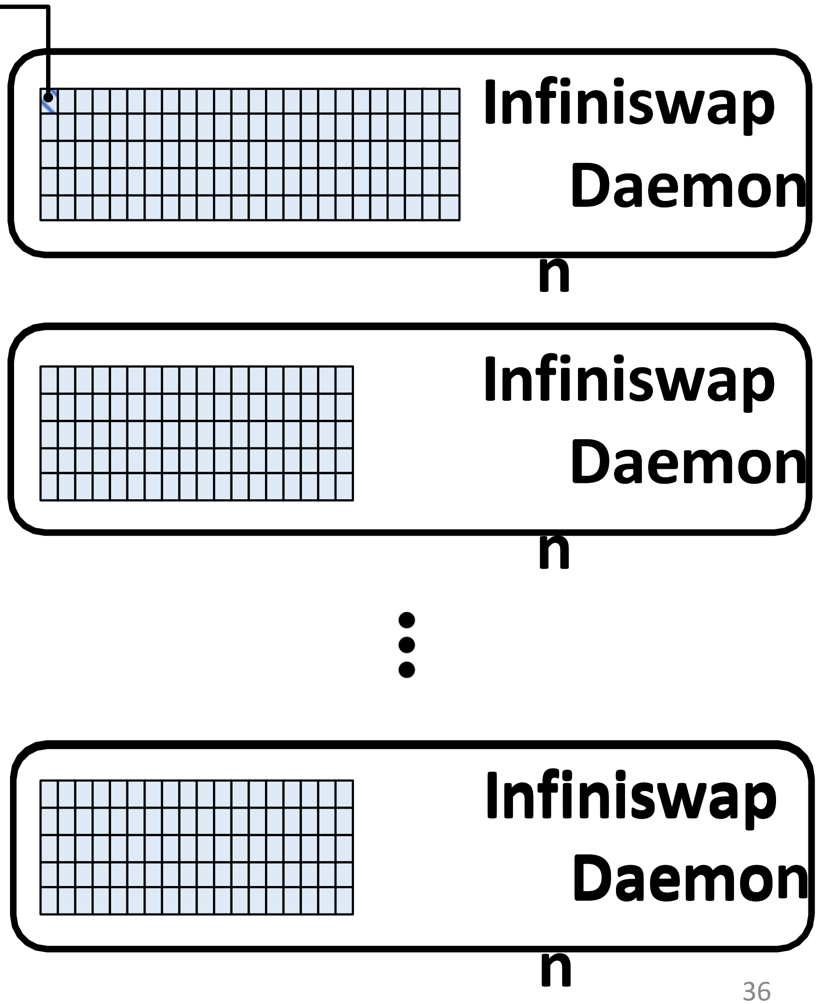


Local Page	Remote Page
p100	<s1, p1>

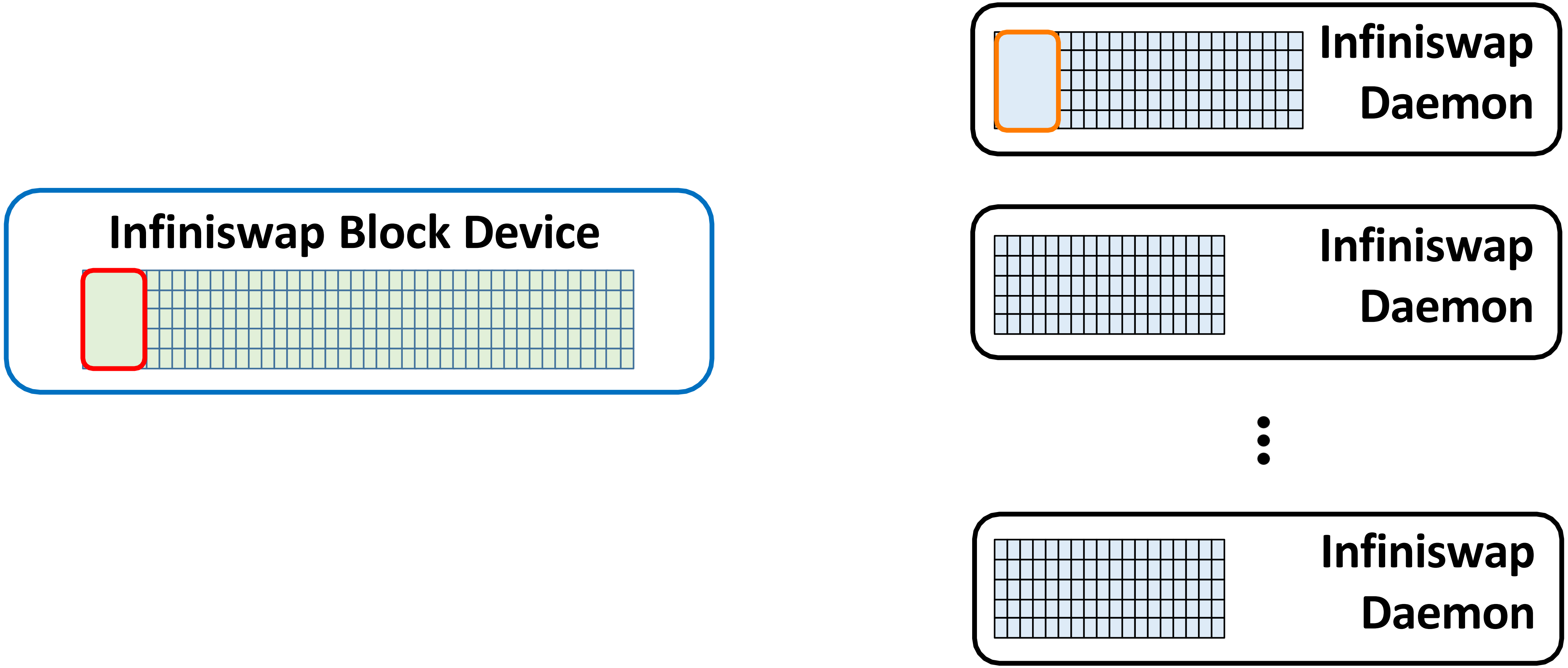
⋮

1GB = 256K entries

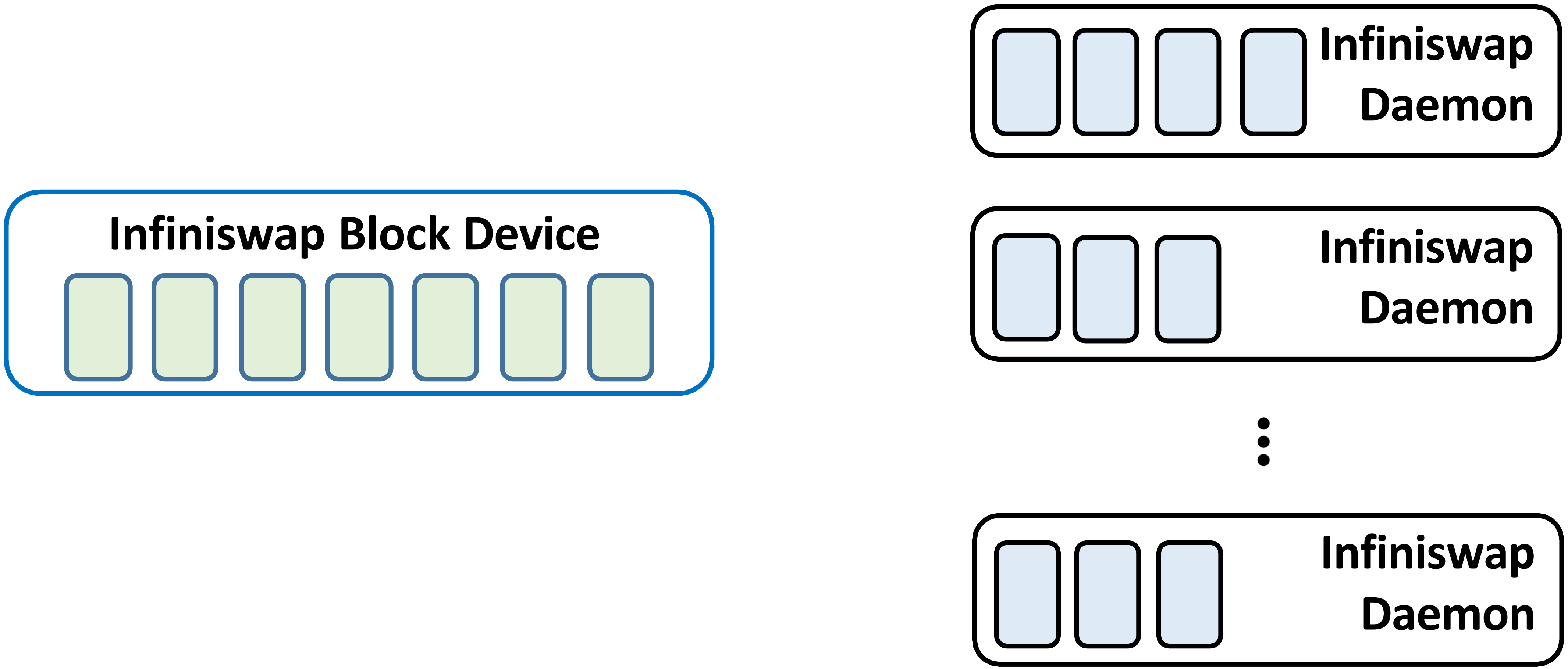
1GB = 256K RTTs



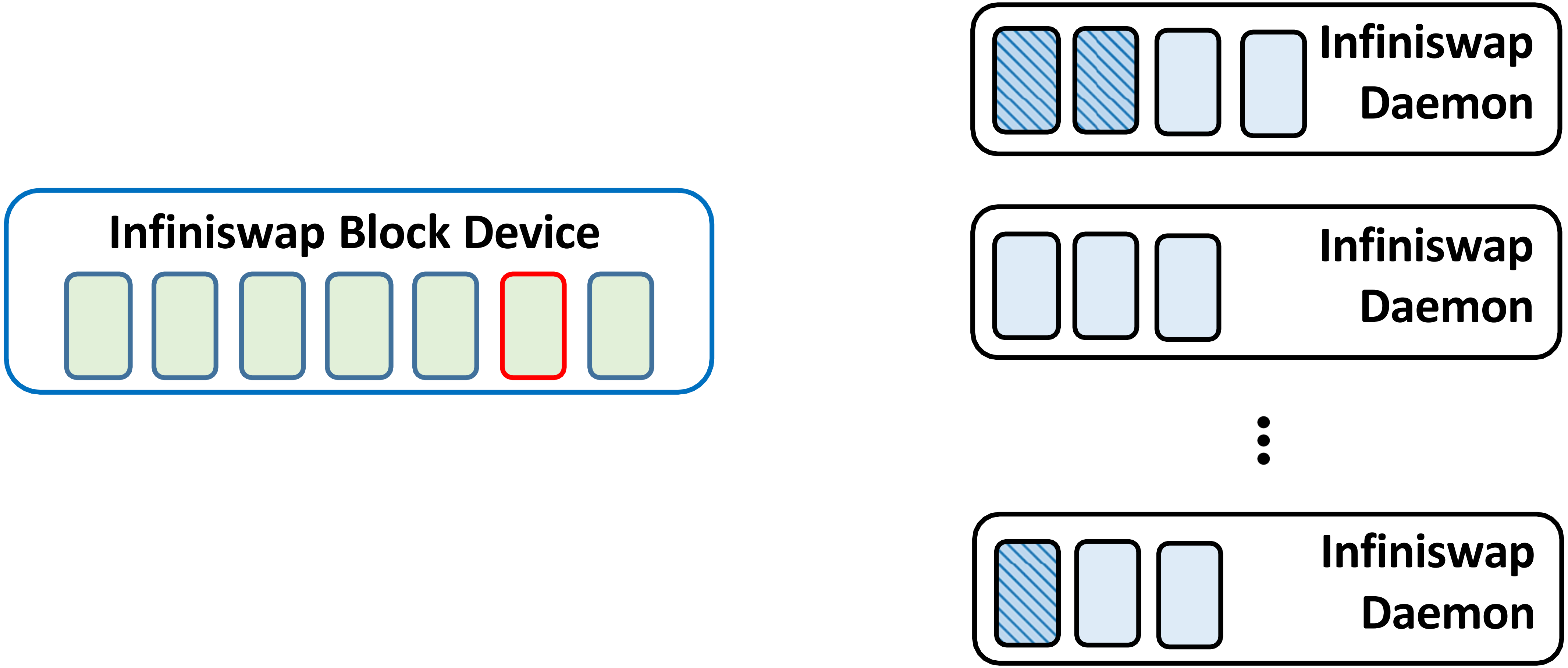
Management unit: memory slab!



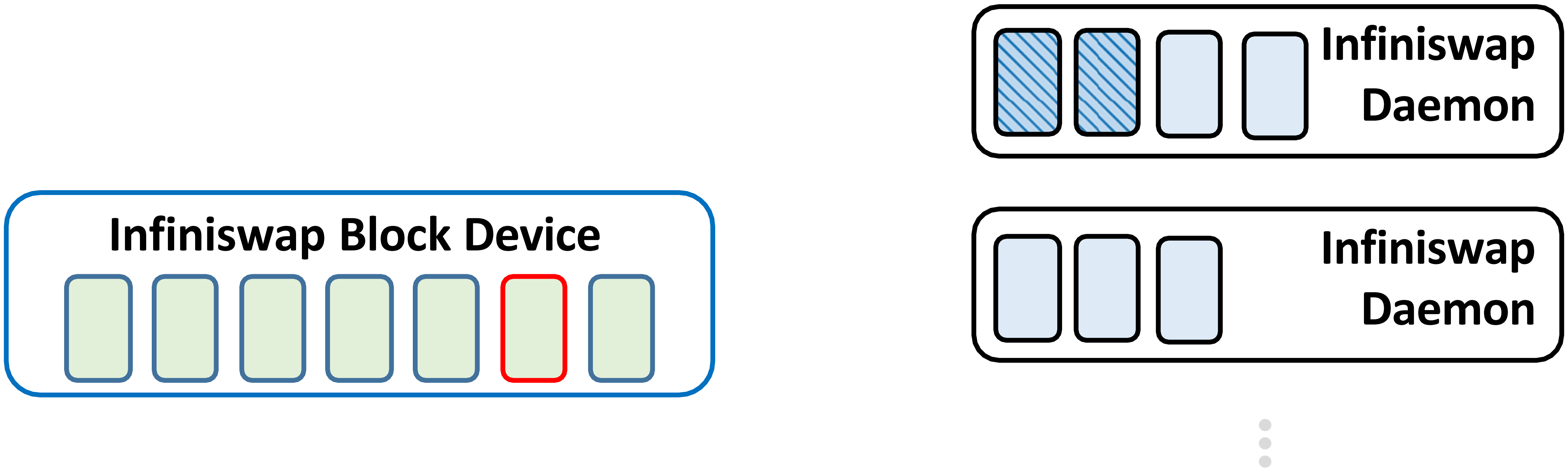
Management unit: memory slab!



Which remote machine should be selected?

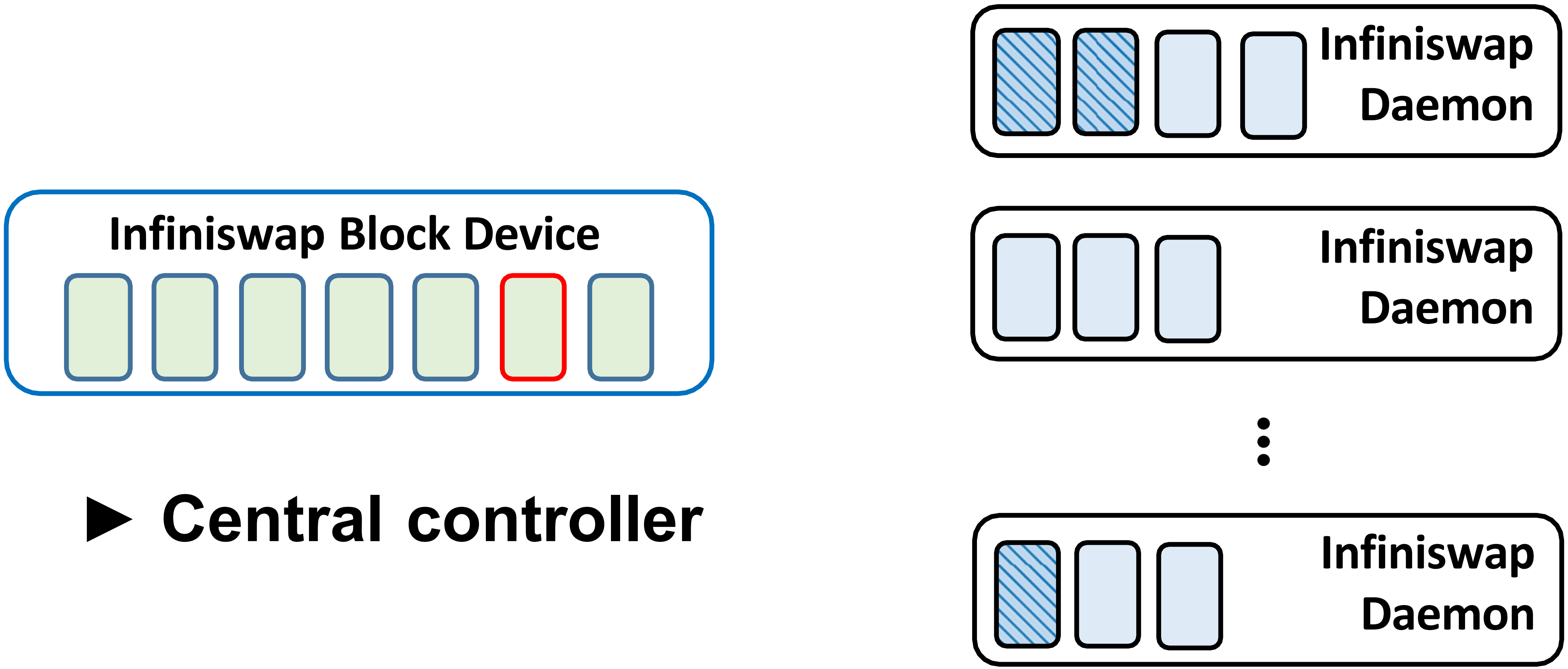


Which remote machine should be selected?



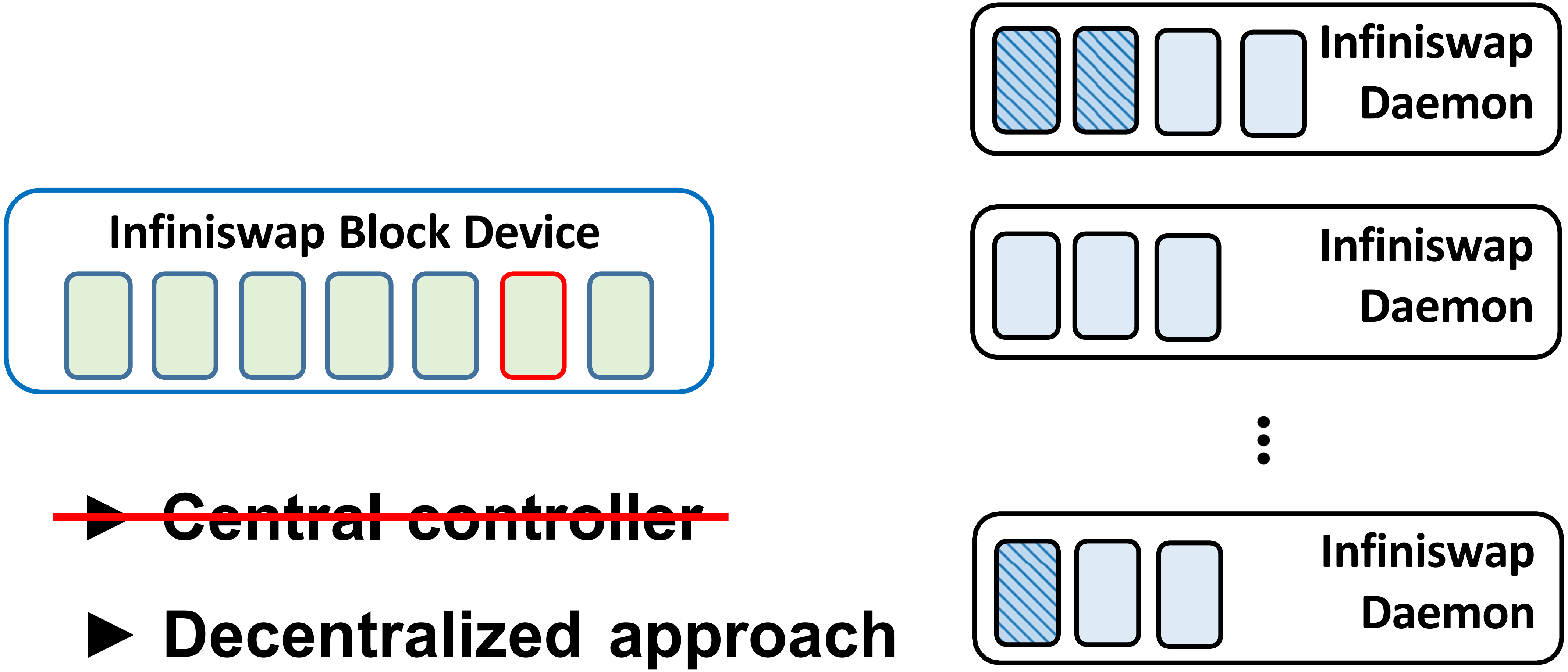
Goal: balance memory utilization

Which remote machine should be selected?



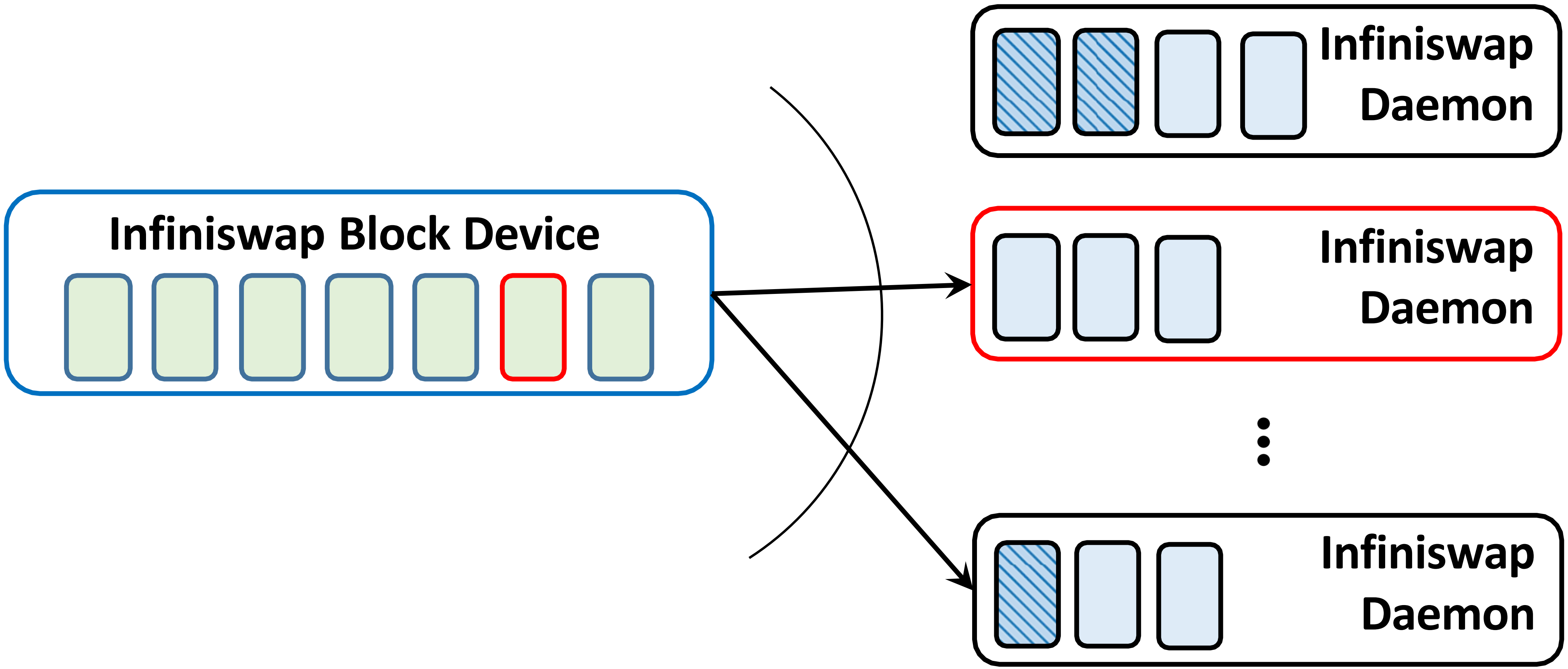
► Central controller

Which remote machine should be selected?



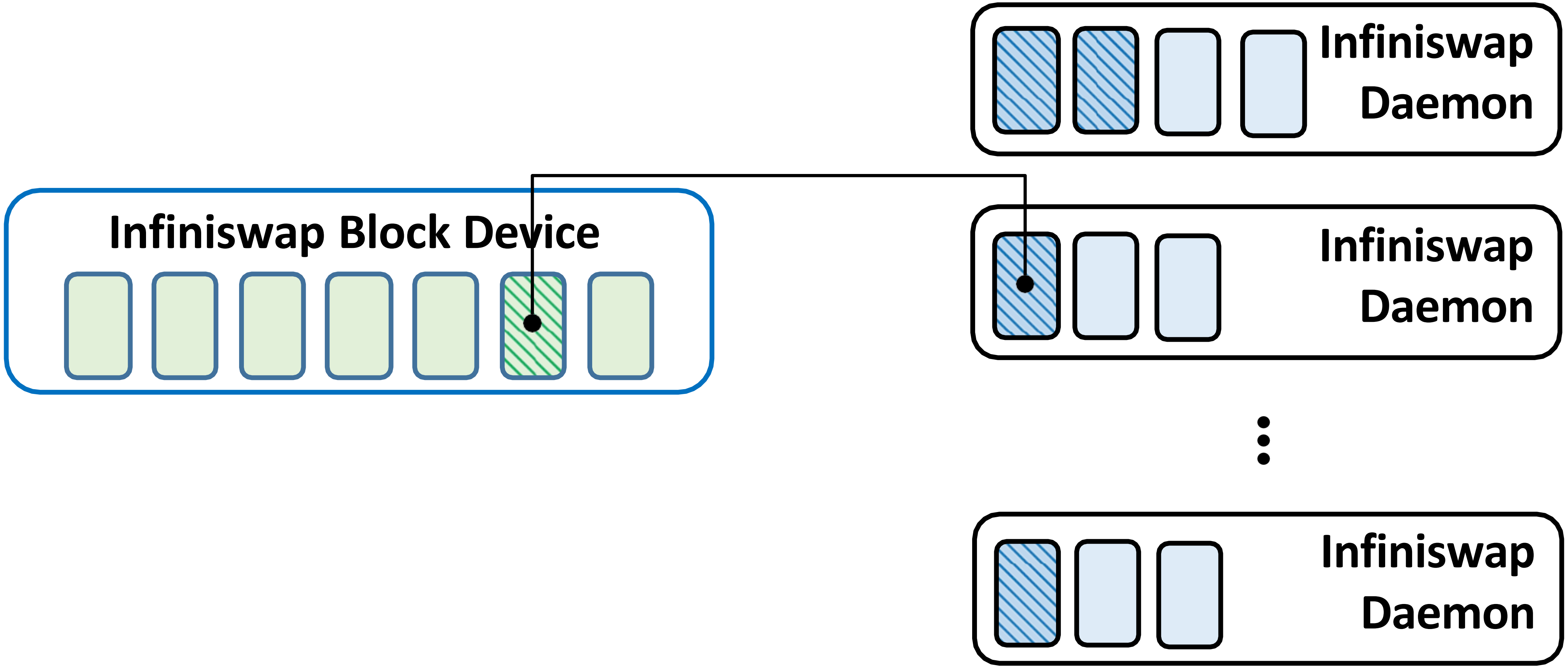
- ~~Central controller~~
- Decentralized approach

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]

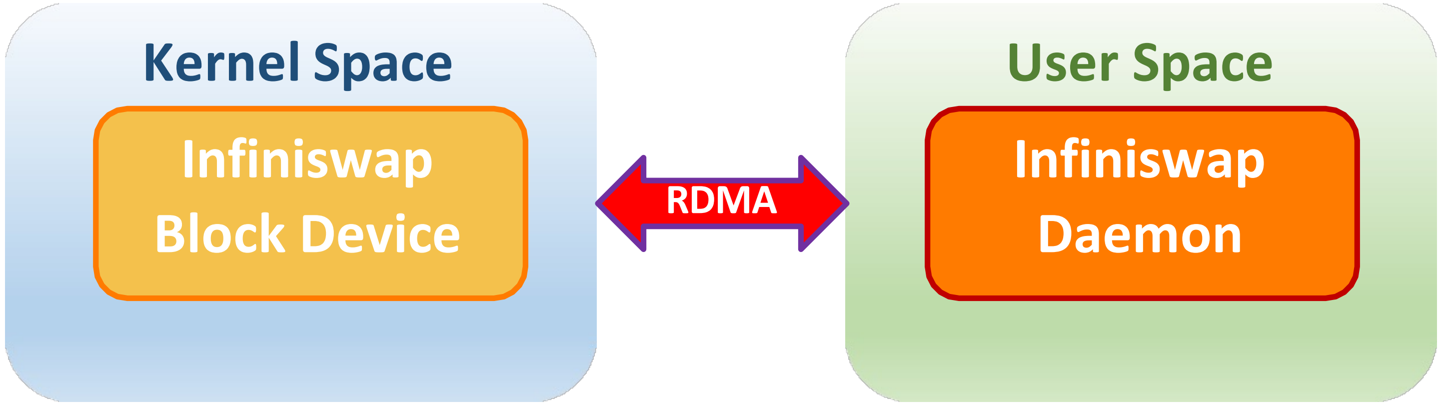


[1] Mitzenmacher, Michael. "The power of two choices in randomized load balancing.", Ph.D. thesis, U.C. Berkeley, 1996

Agenda

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

Implementation



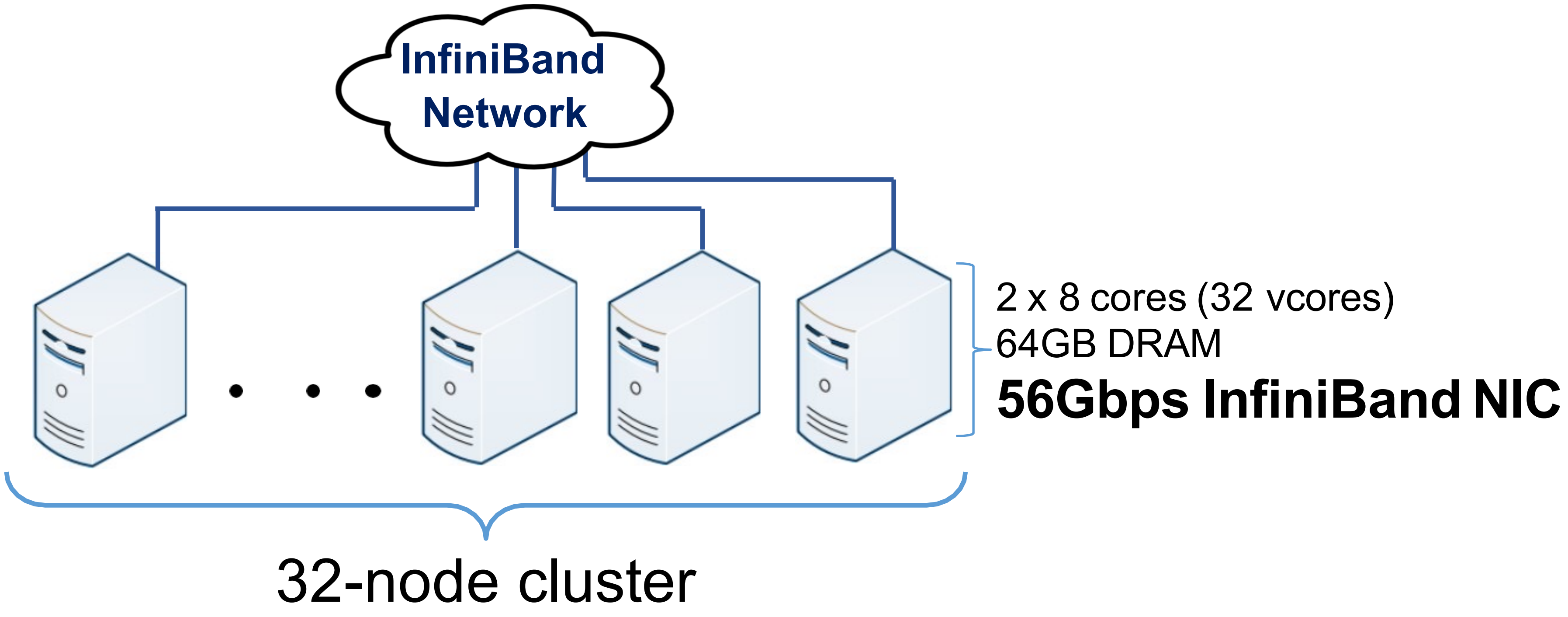
- **Connection Management**
  - **One** RDMA connection per active block device - daemon pair
- **Control Plane**
  - **SEND, RECV**
- **Data Plane**
  - **One-sided RDMA READ, WRITE**

What are we expecting from Infiniswap?

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

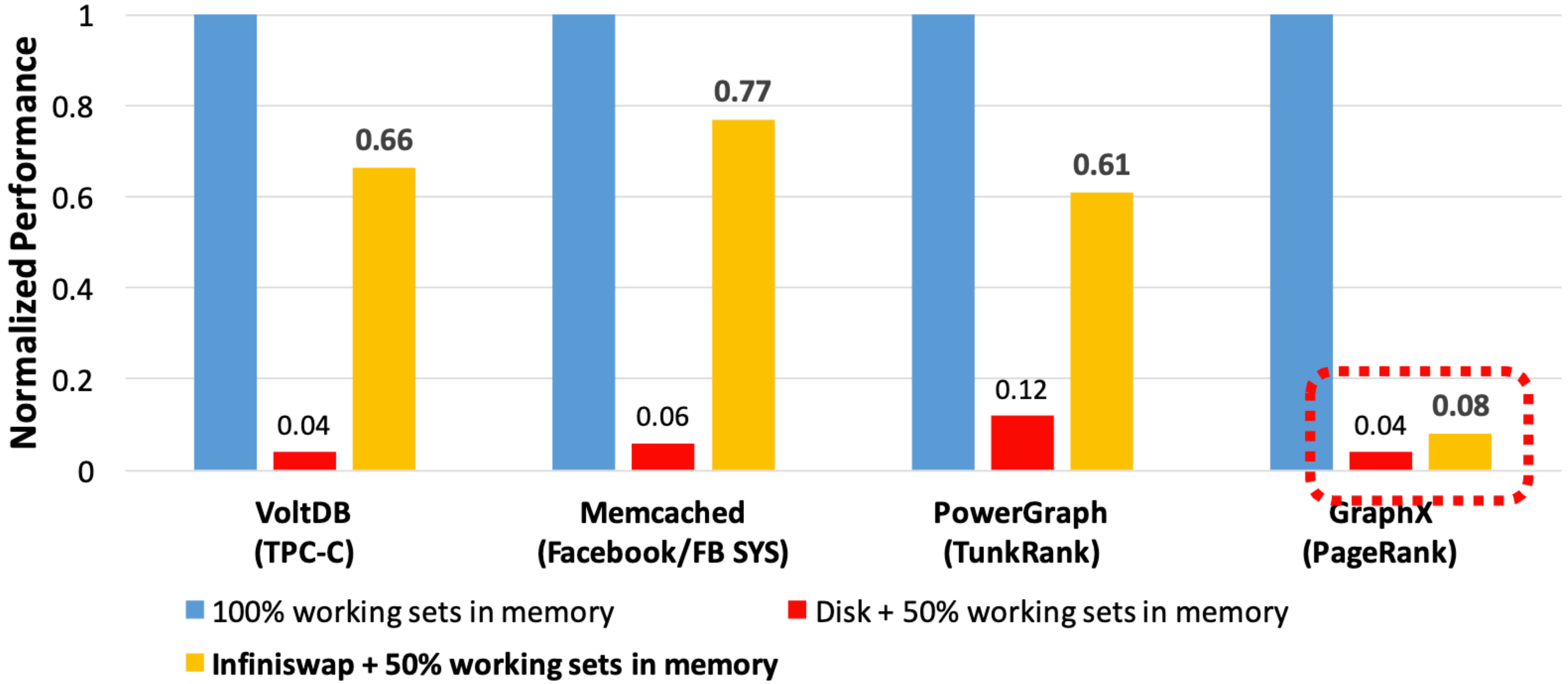
⋮

Evaluation


InfiniBand
Network
2 x 8 cores (32 vcores)
64GB DRAM
56Gbps InfiniBand NIC
32-node cluster

Application performance

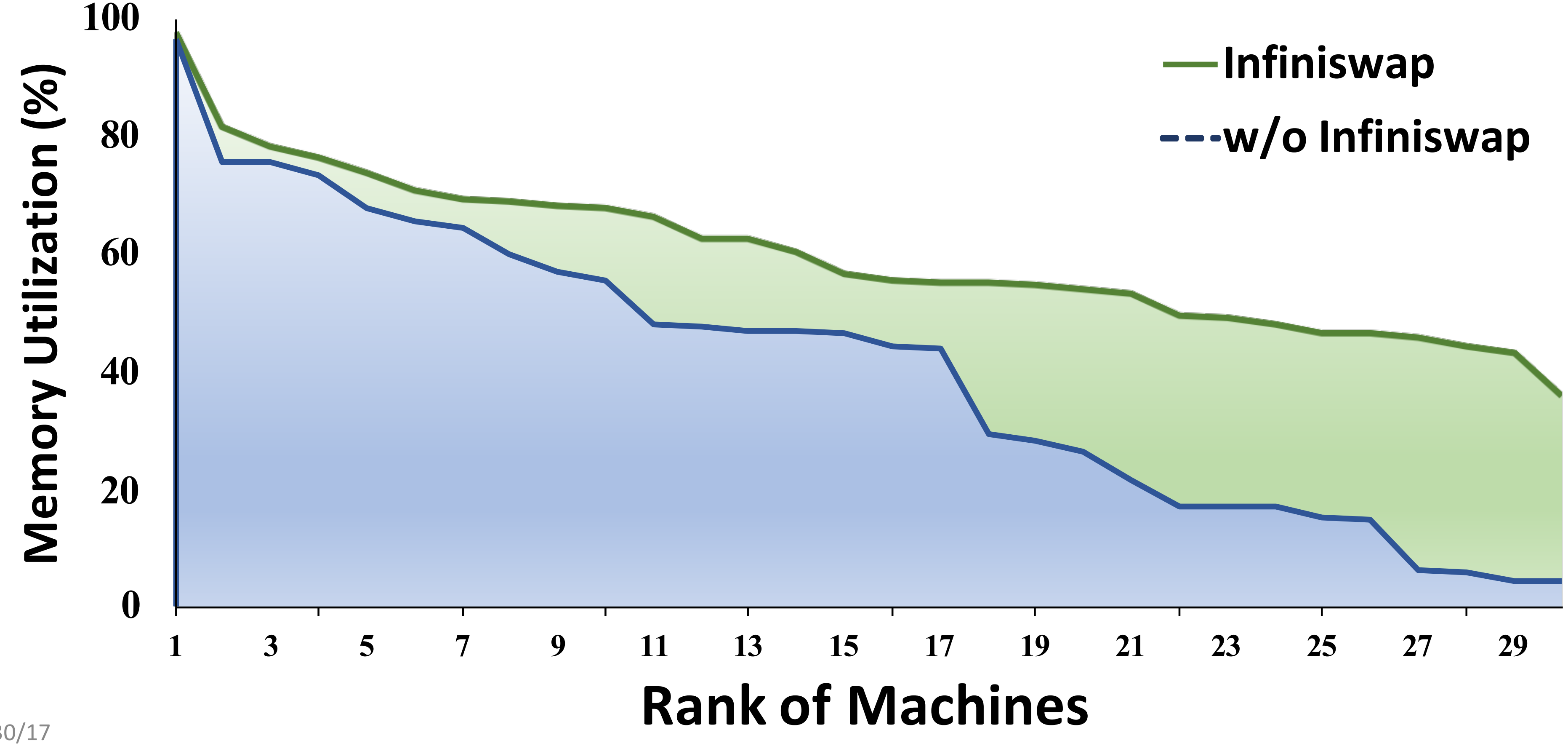
- 50% working sets in memory



- Application performance is improved by 2-16x

Cluster memory utilization

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



- 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

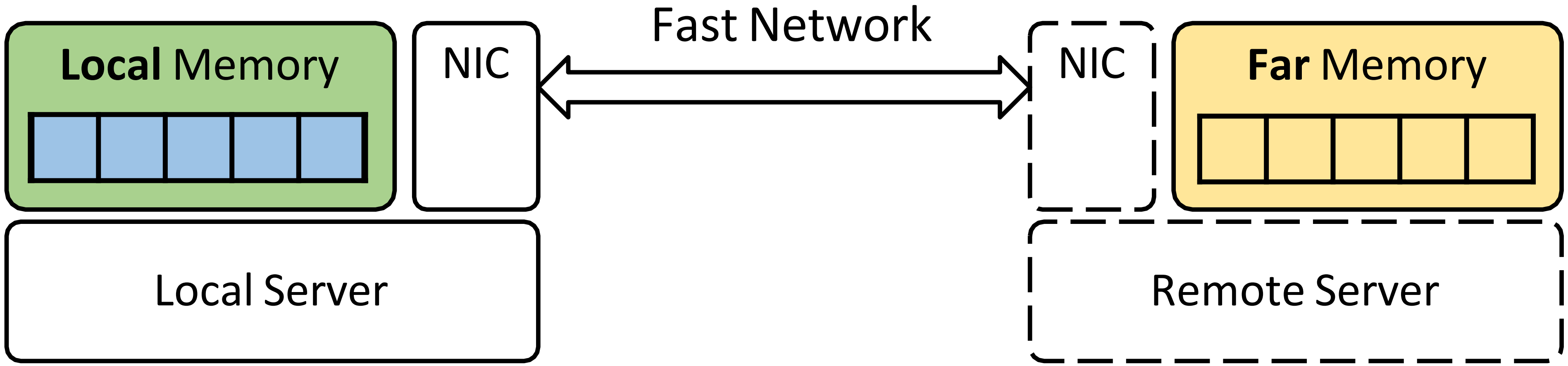
30

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

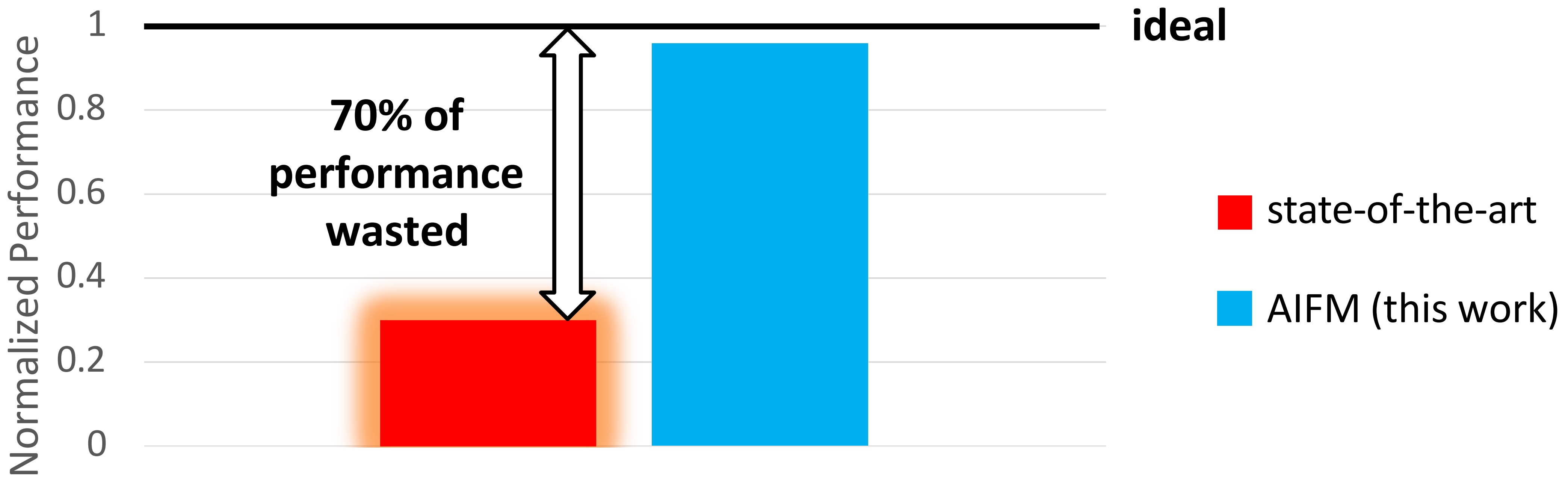
- Leverage the idle memory of remote servers (with fast network).



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.

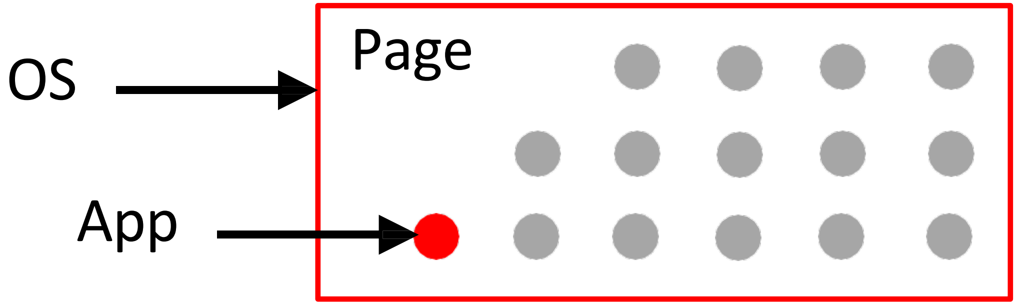


Why Do Existing Systems Waste Performance?

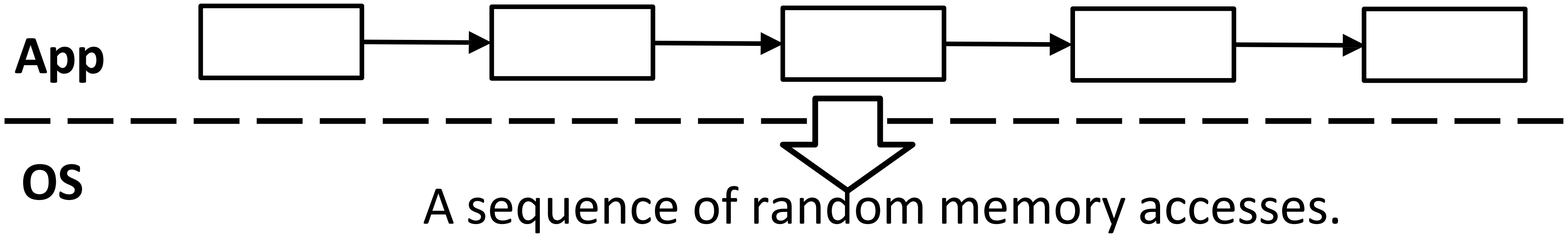
- Problem: based on **OS paging**.
  - Semantic gap.
  - High kernel overheads.

Challenge 1: Semantic Gap

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



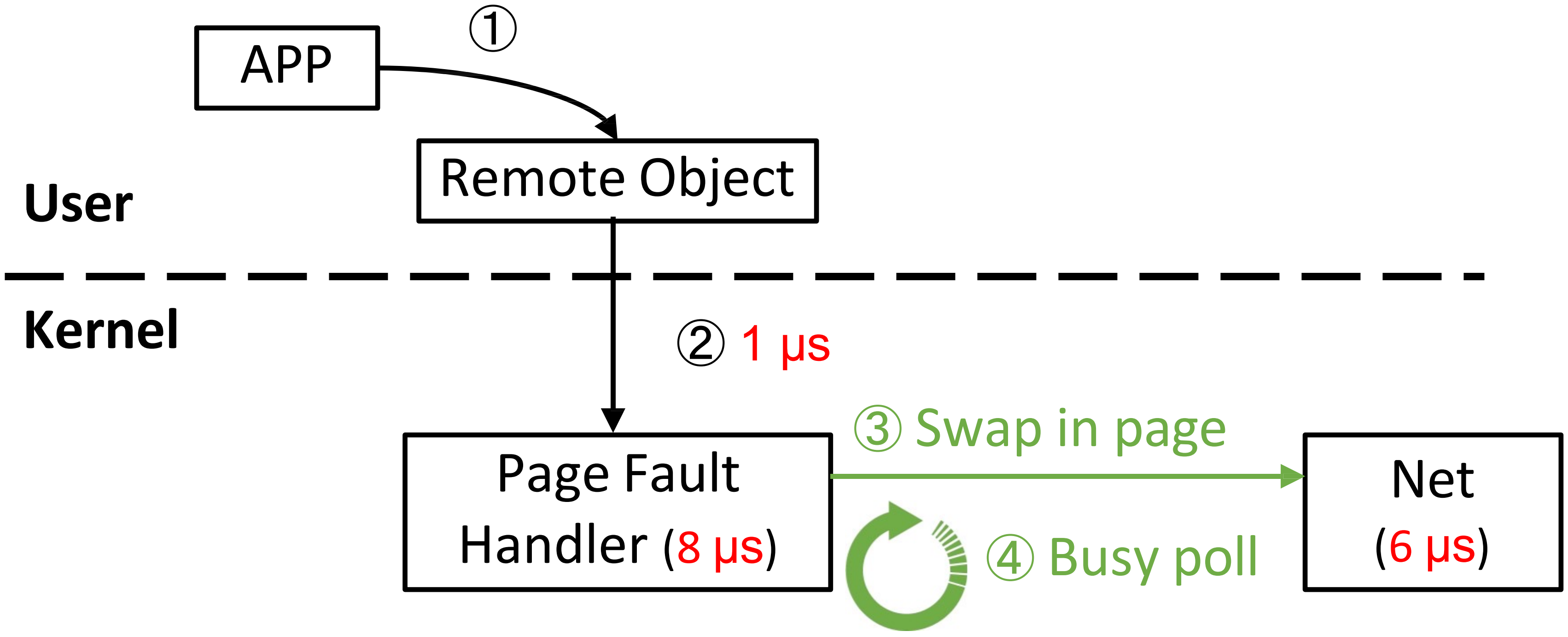
➢ OS lacks app knowledge ➔ **hard to prefetch, etc.**



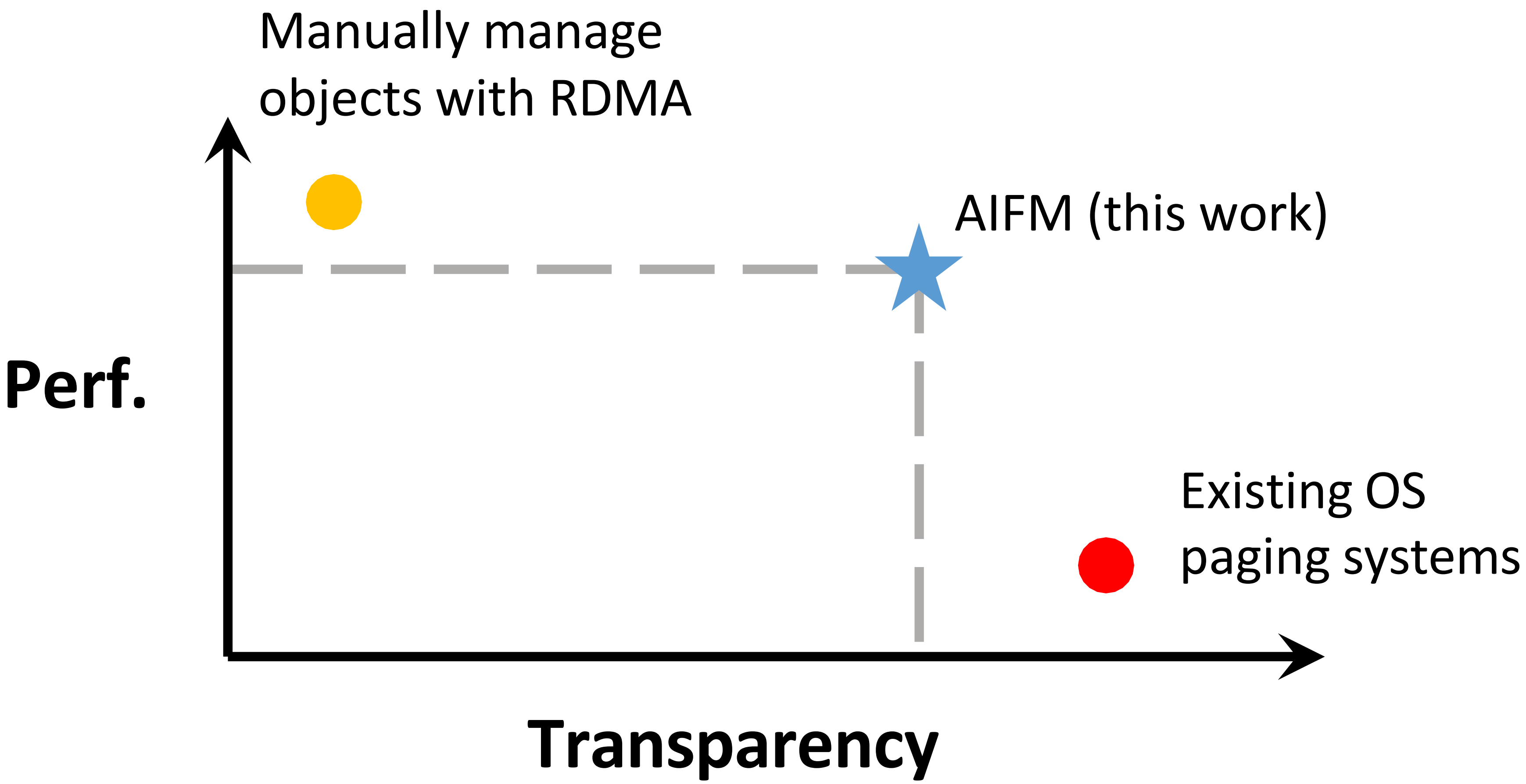
Challenge 2: High Kernel Overheads

- **Expensive page faults.**

➢ Busy Polling for in-kernel net I/O ➔ **burn CPU cycles**.



Design Space



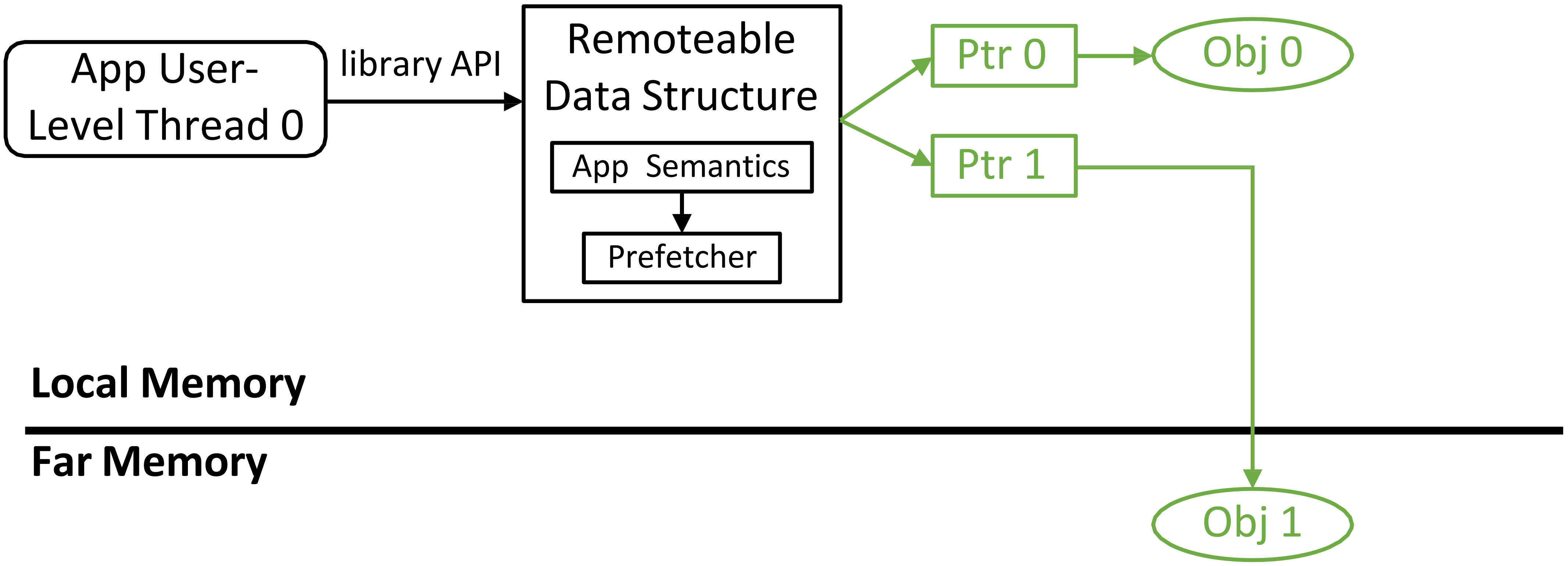
AIFM’s Design Overview

- 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

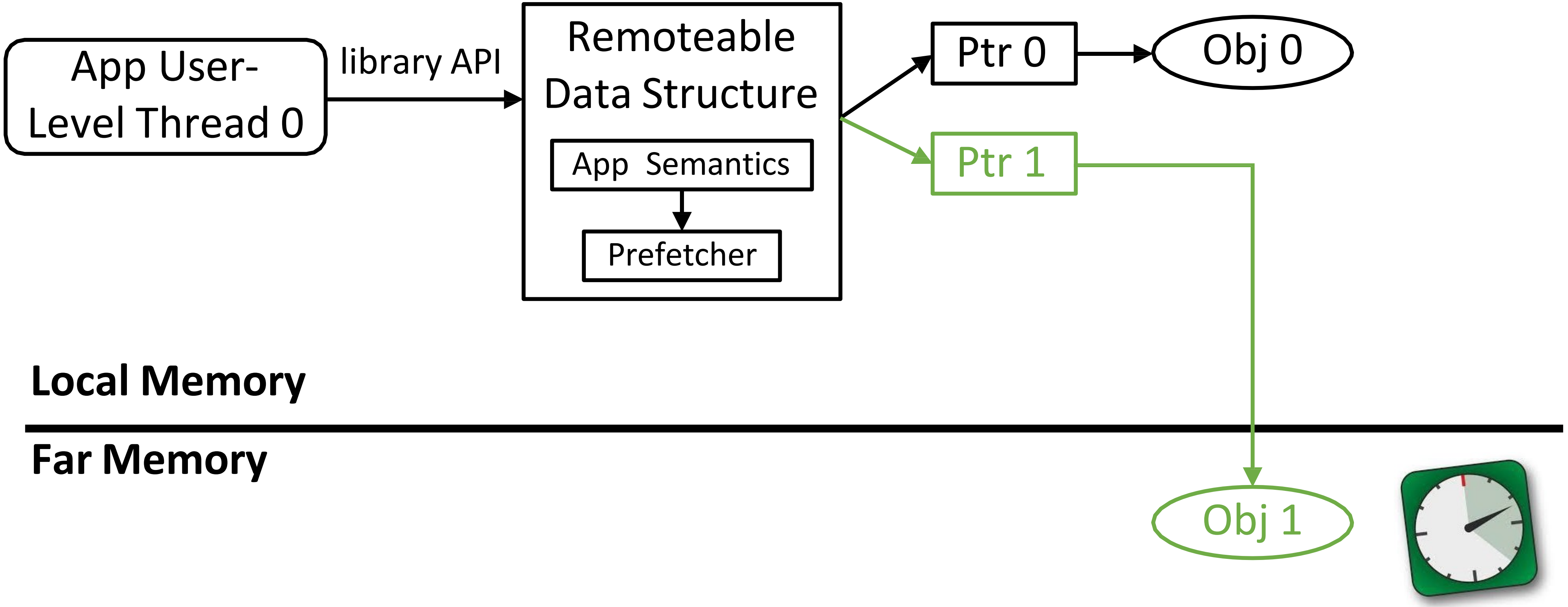
1. Remoteable Data Structure Library

- Solved challenge: semantic gap.



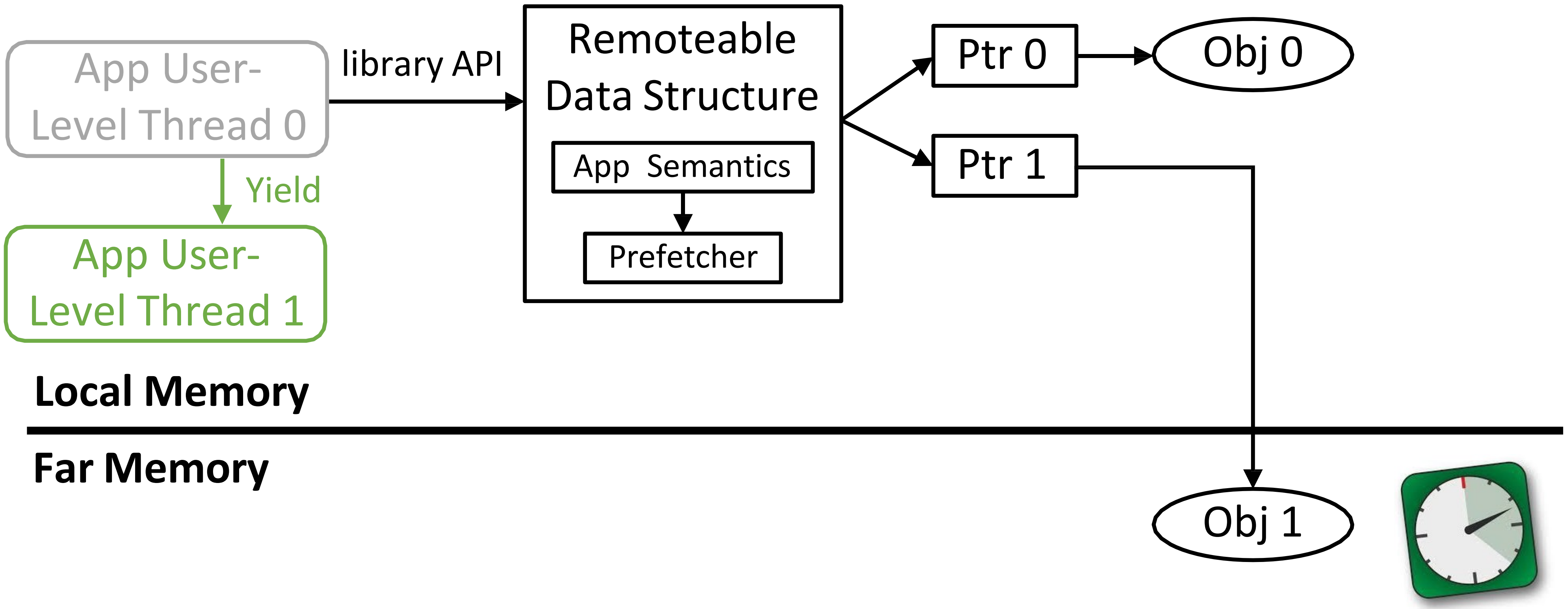
2. Userspace Runtime

- Solved challenge: kernel overheads.



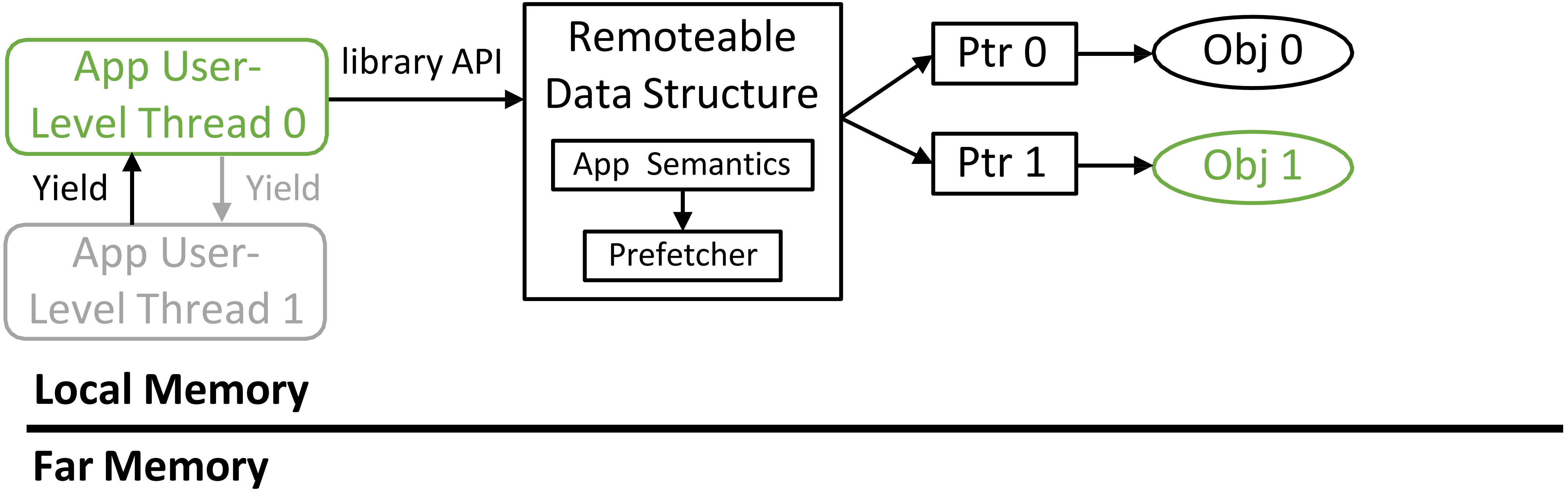
2. Userspace Runtime

- Solved challenge: kernel overheads.



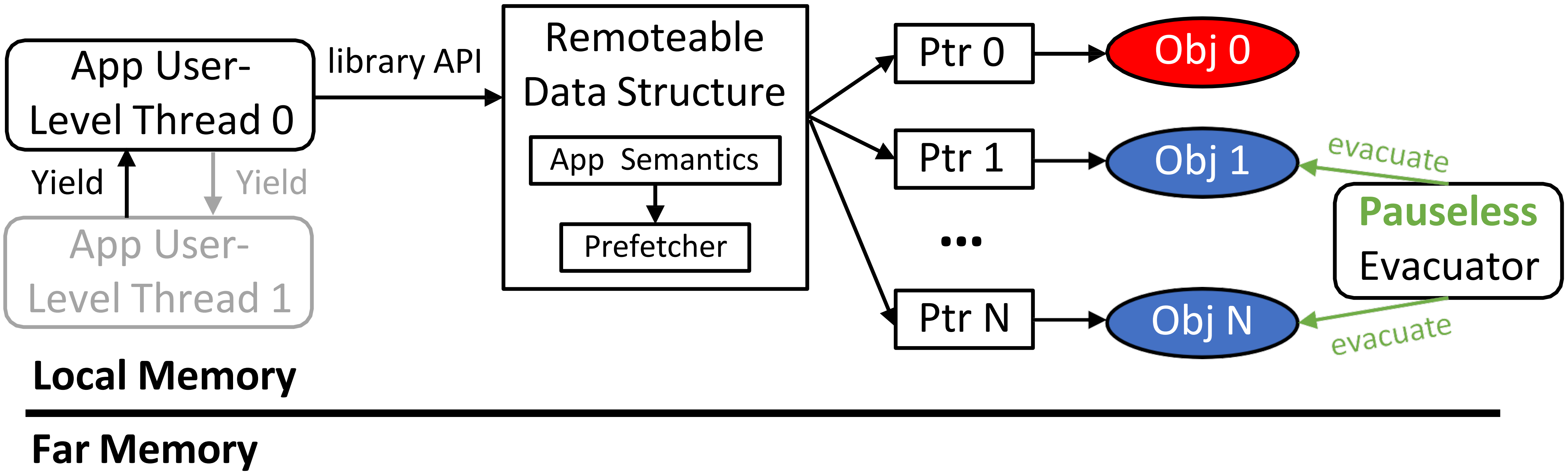
2. Userspace Runtime

- Solved challenge: kernel overheads.



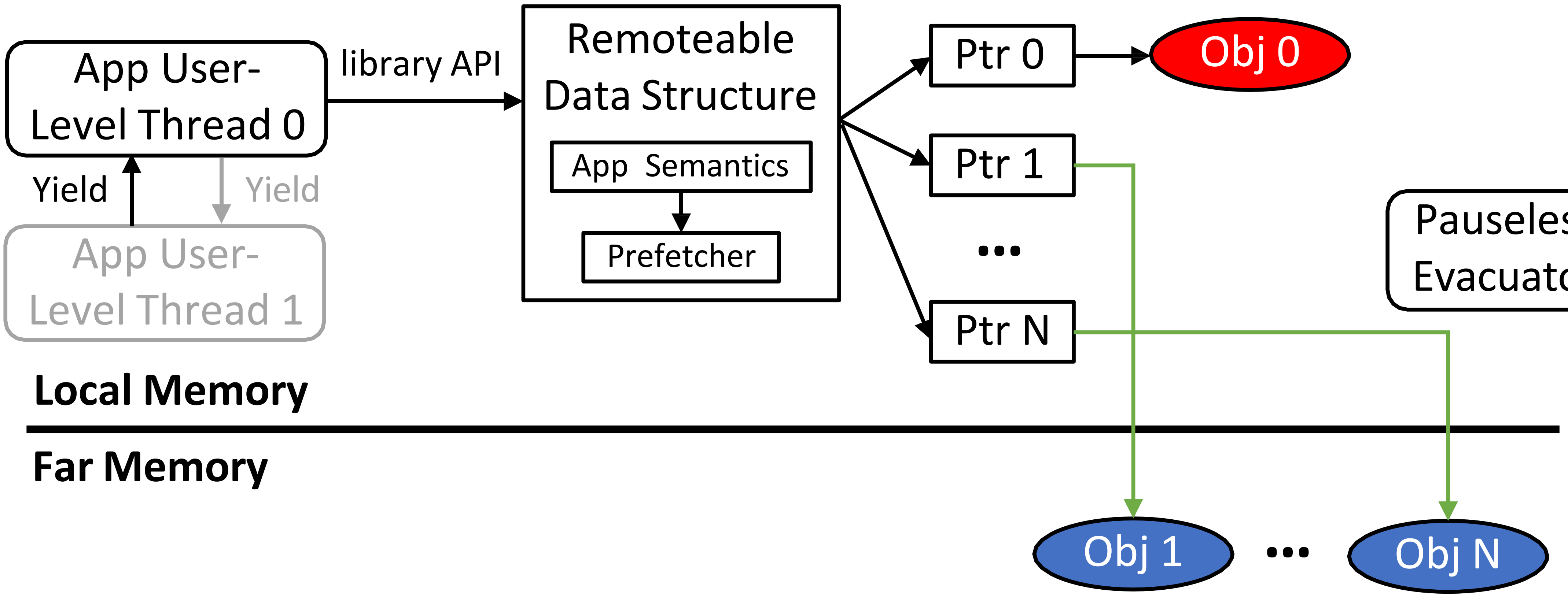
3. Pauseless Evacuator

➢ Solved challenge: impact of memory reclamation.



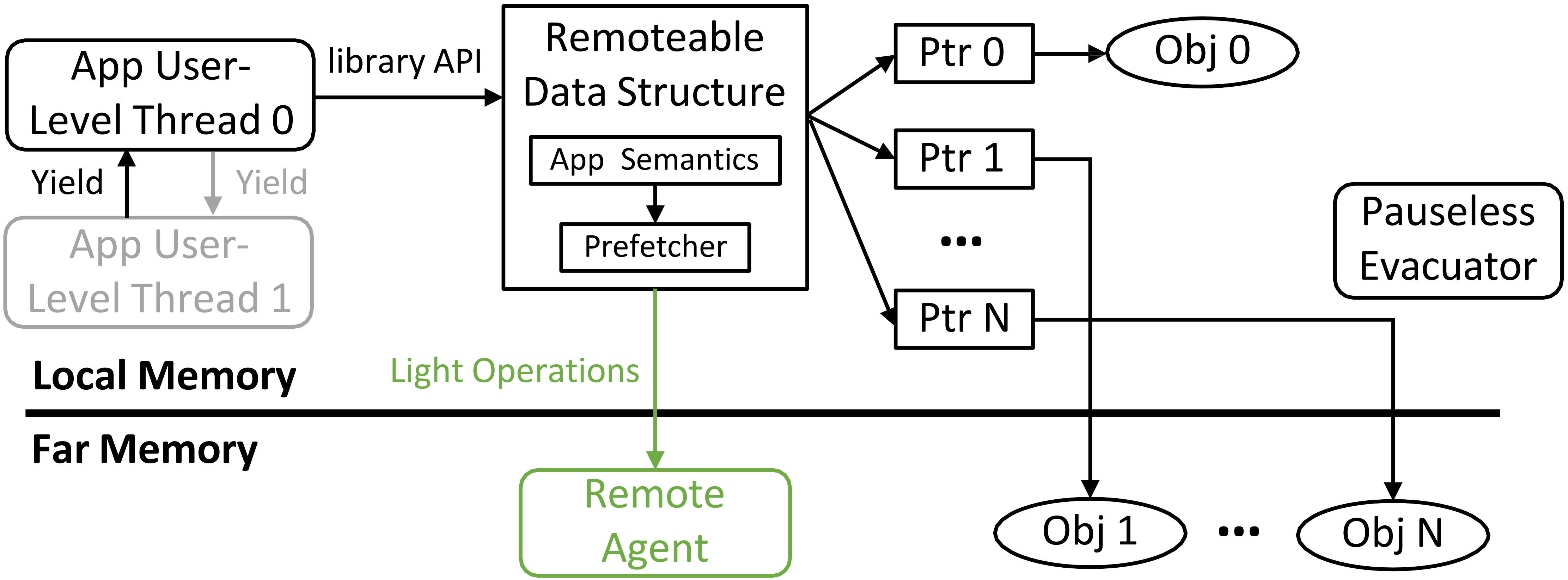
3. Pauseless Evacuator

- Solved challenge: impact of memory reclamation.



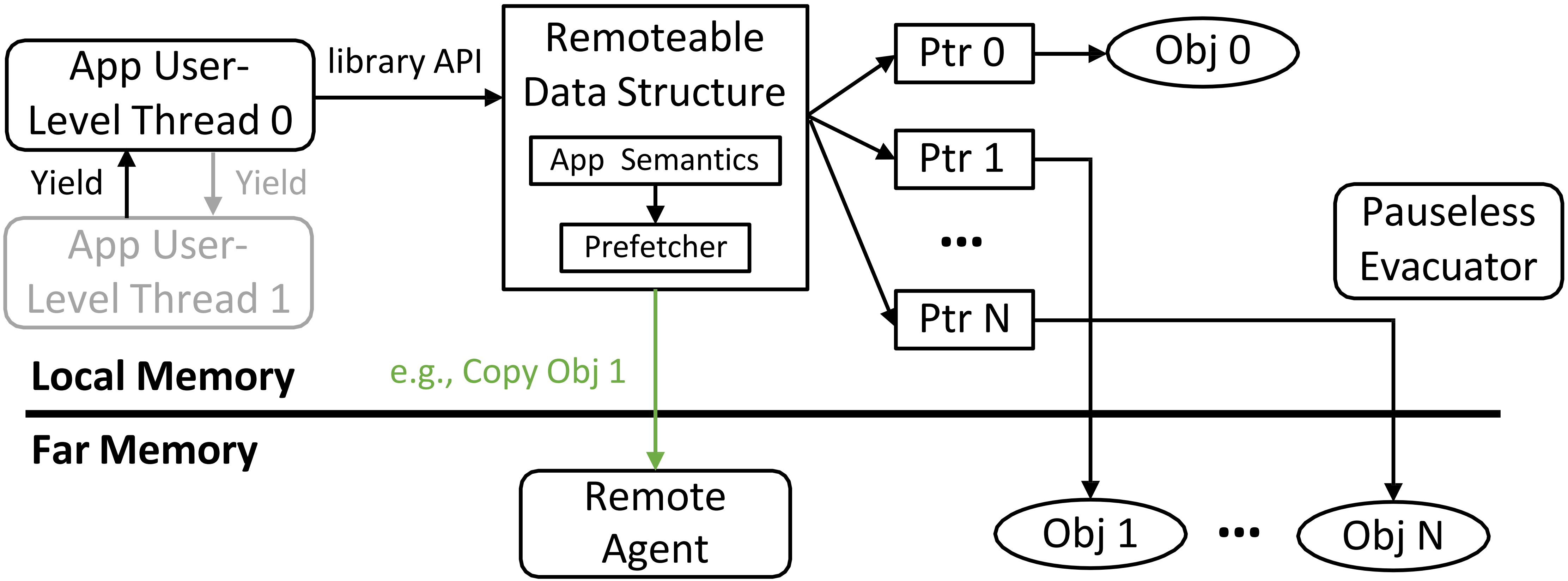
4. Remote Agent

➢Solved challenge: network BW < DRAM BW.



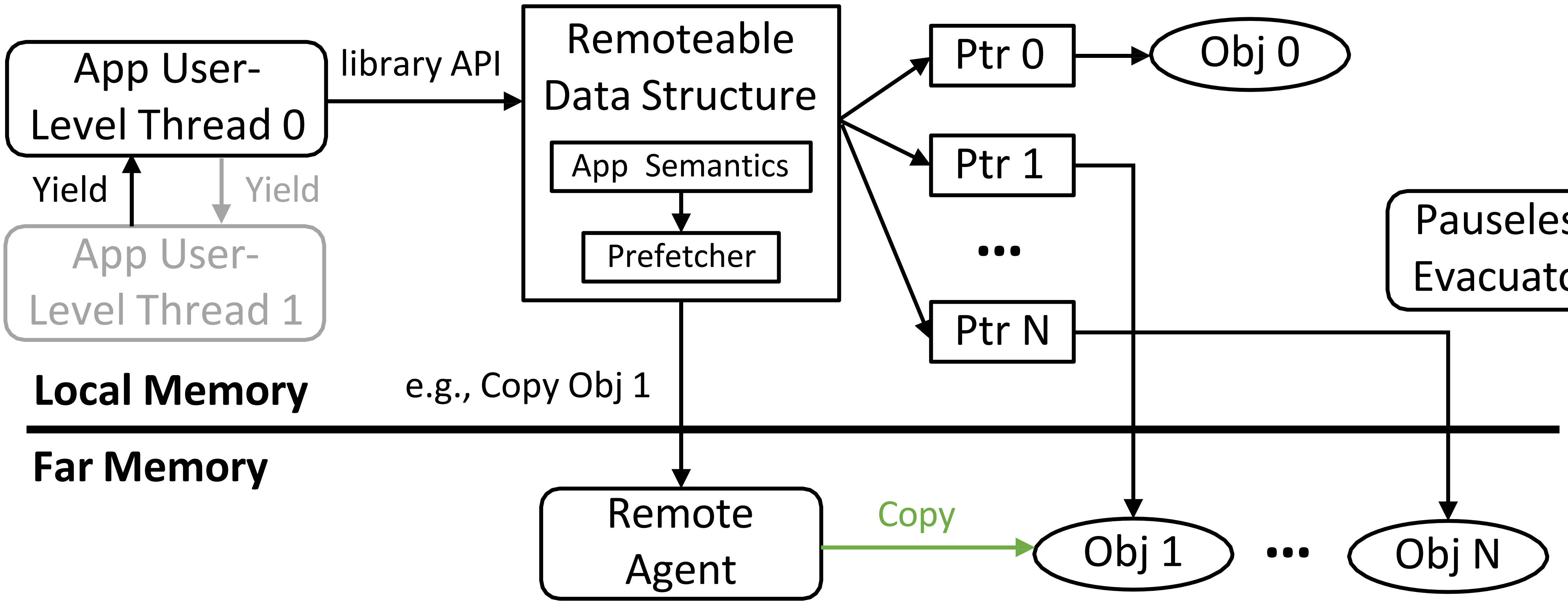
4. Remote Agent

➢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

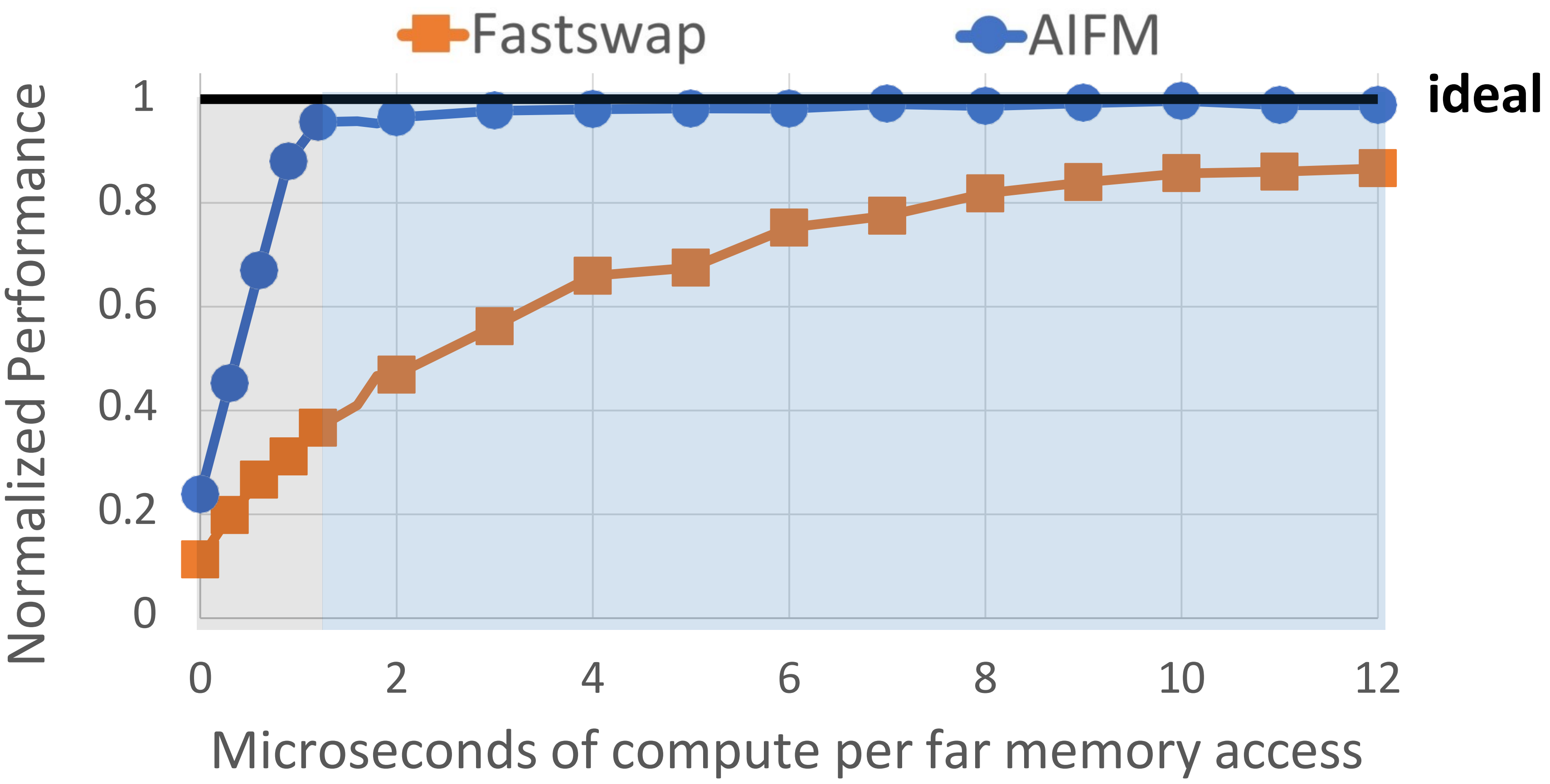
```
RemHashTable<key_t, int> hashtable;
RemArray<LargeData> arr;

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.

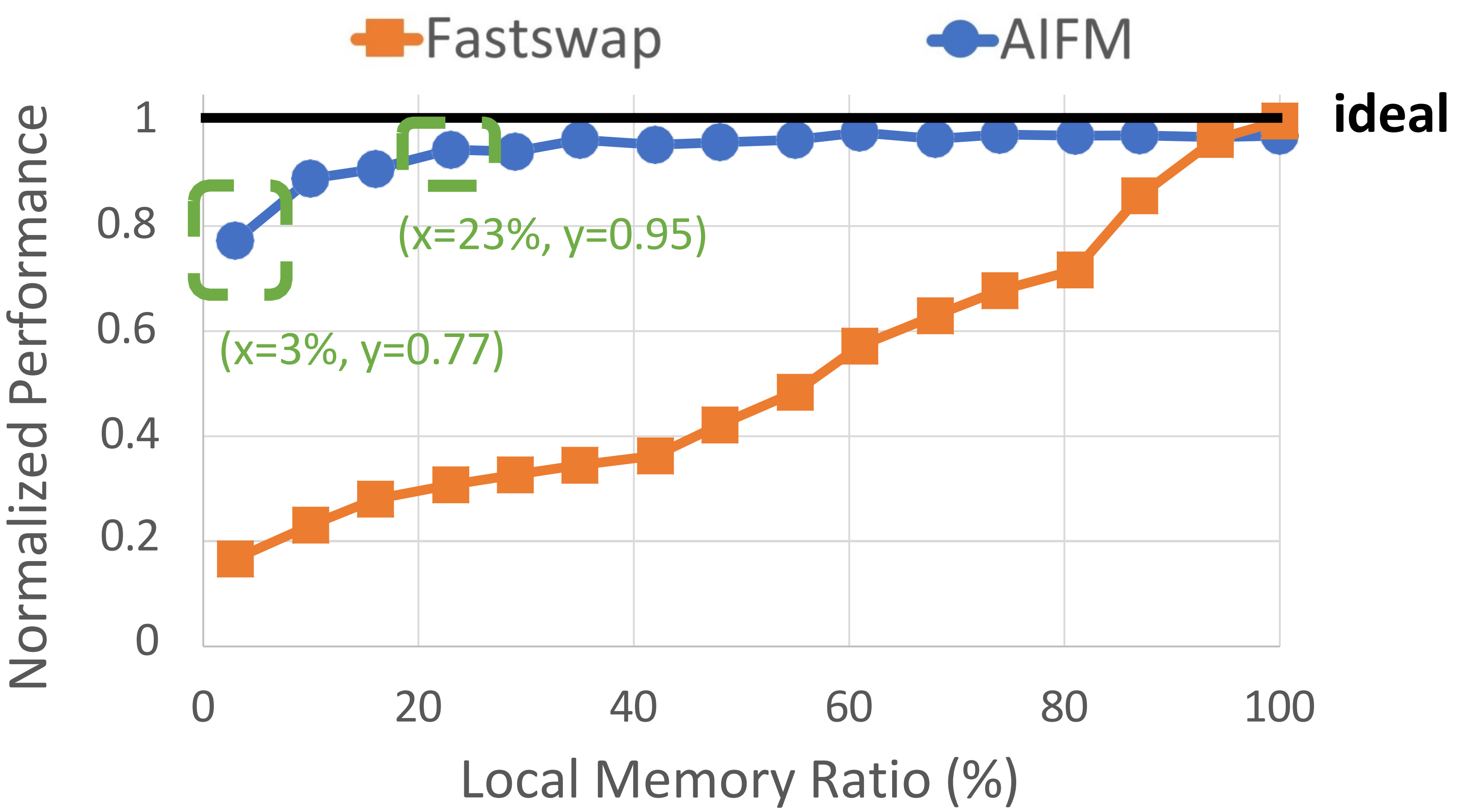
➢ LoC: 6.5K (runtime) + 5.5K (data structures) + 0.8K (Shenango)

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
- 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

Johns Hopkins University

ByteDance

Deep learning powers intelligent applications in many domains

Training and inference

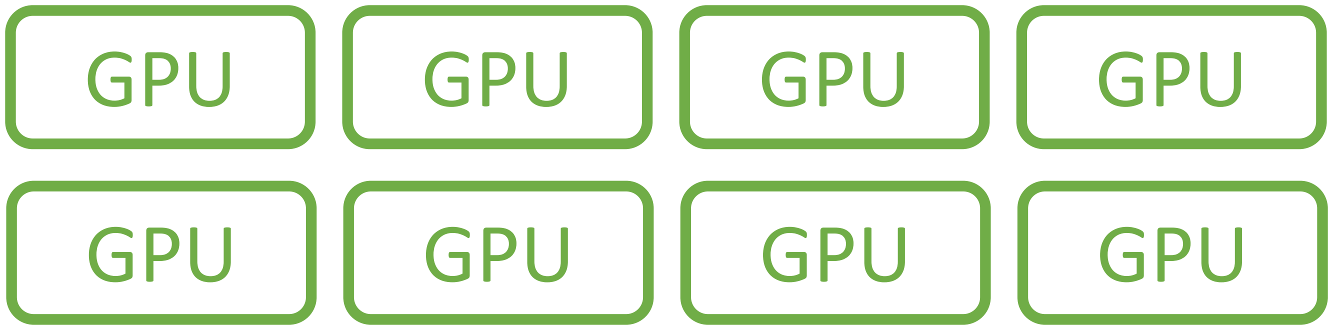
Training

High throughput

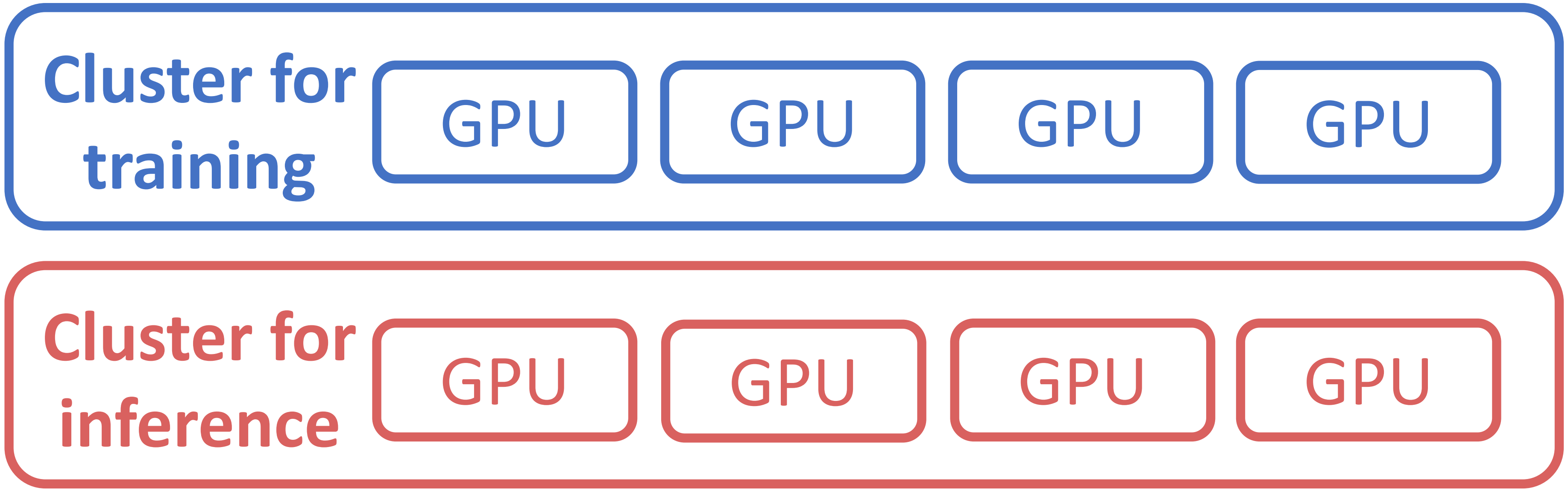
Inference

Low latency

GPUs clusters for DL workloads

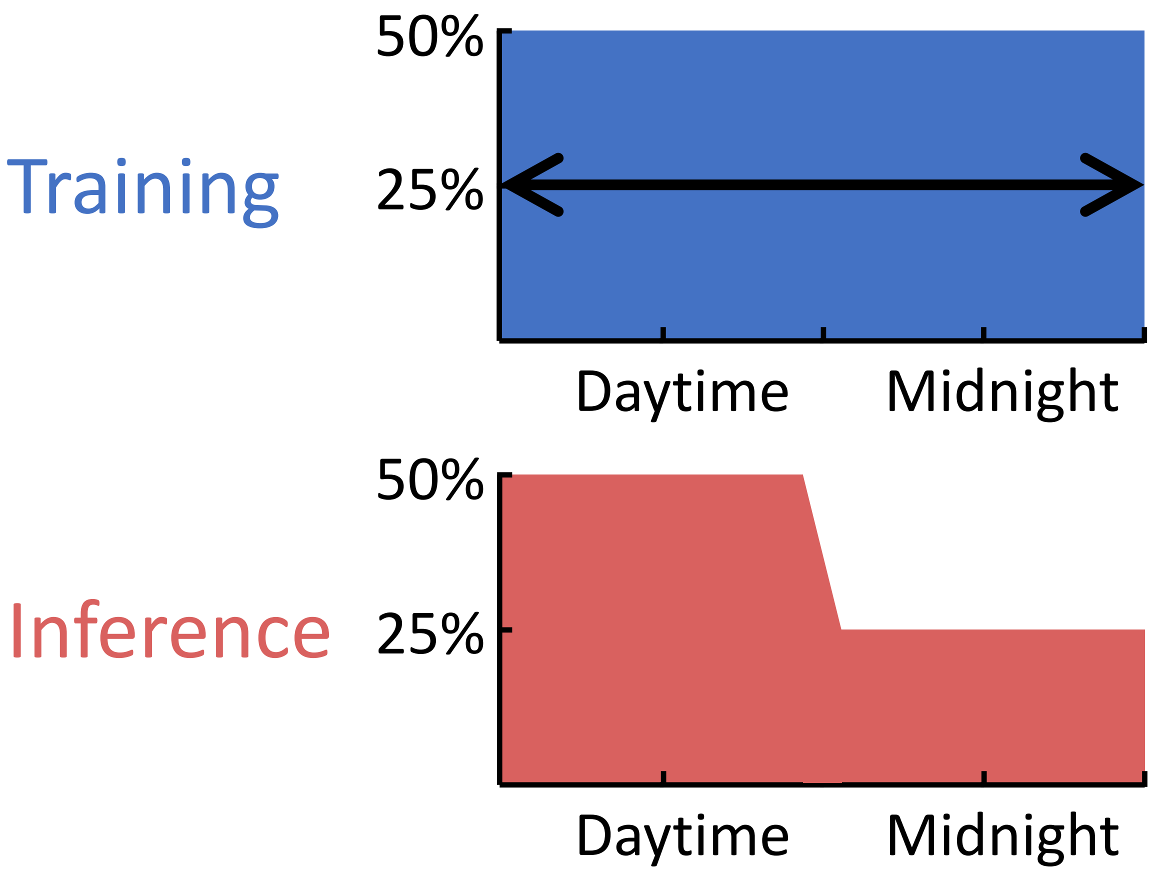


Separate clusters for training and inference

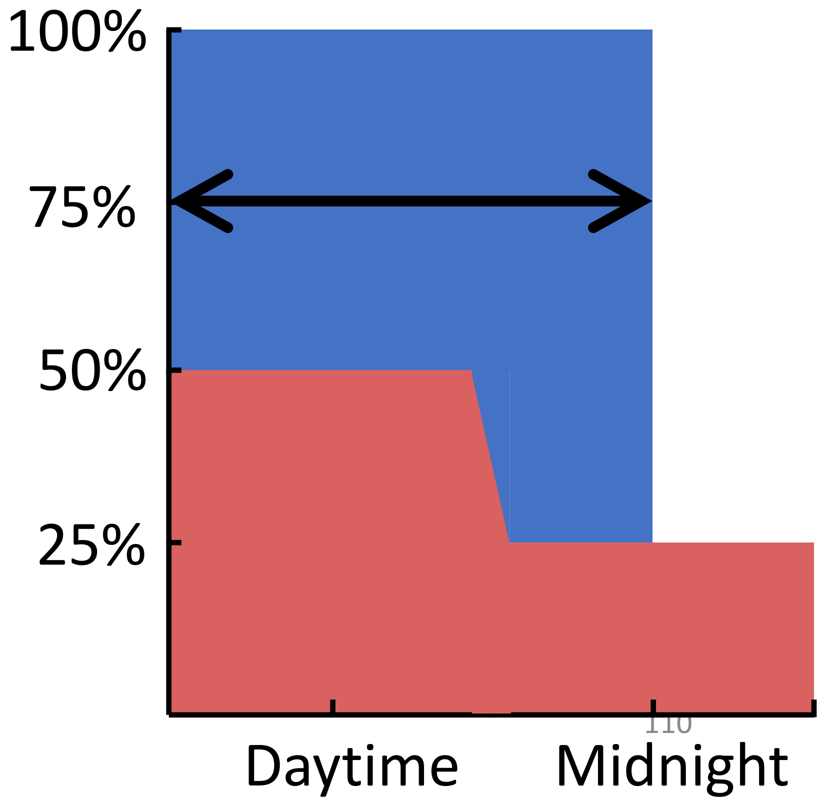


Utilization of GPU clusters is low

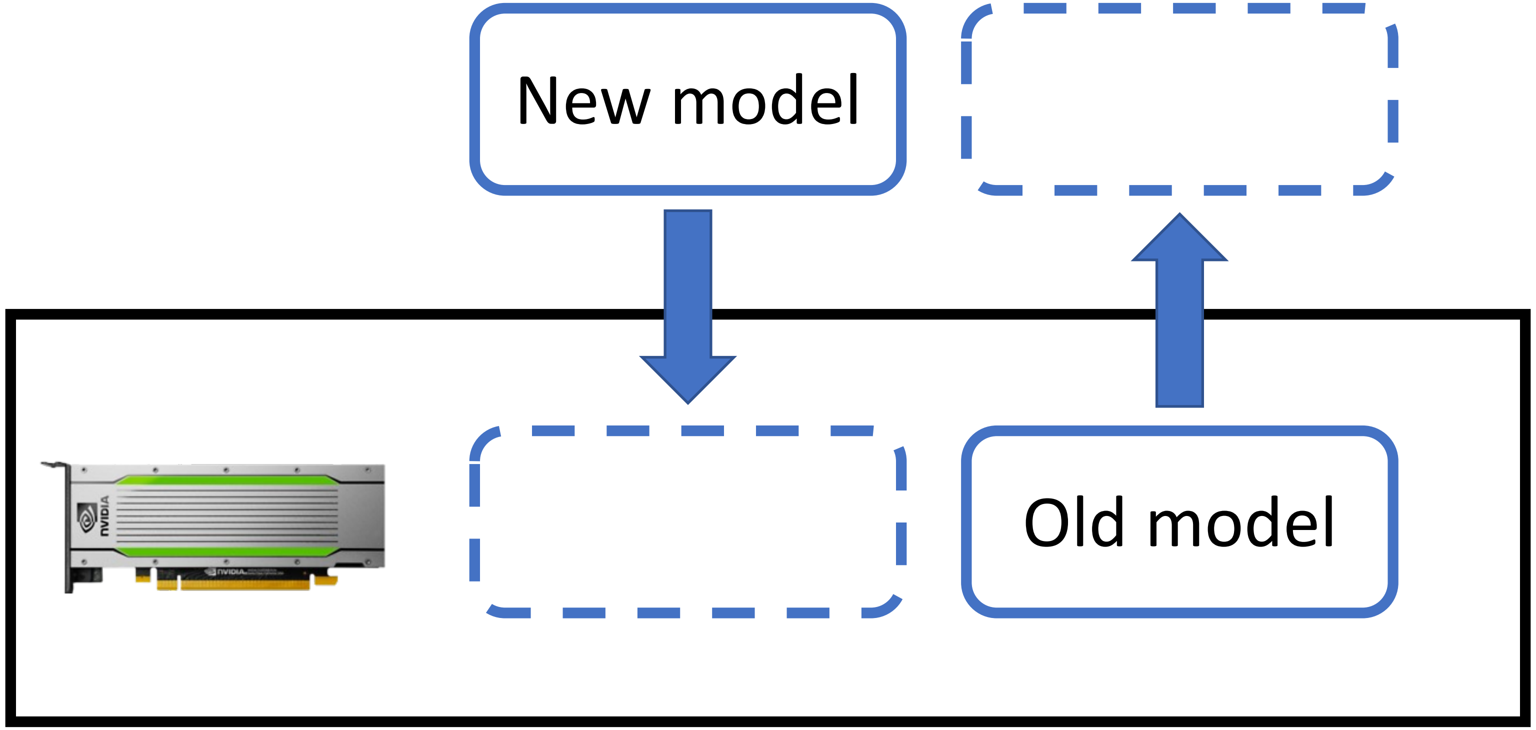
Today: separate clusters



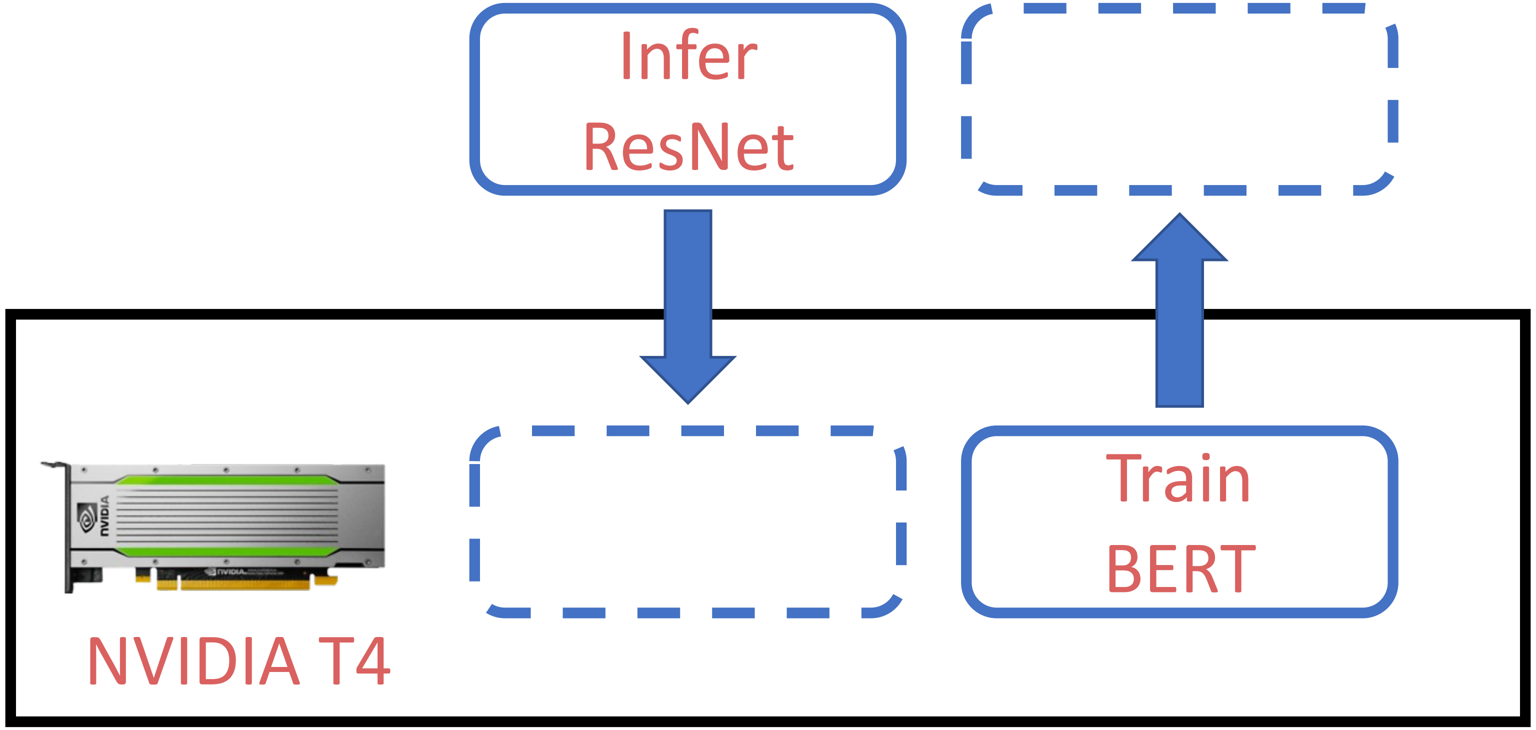
Ideal: shared clusters



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

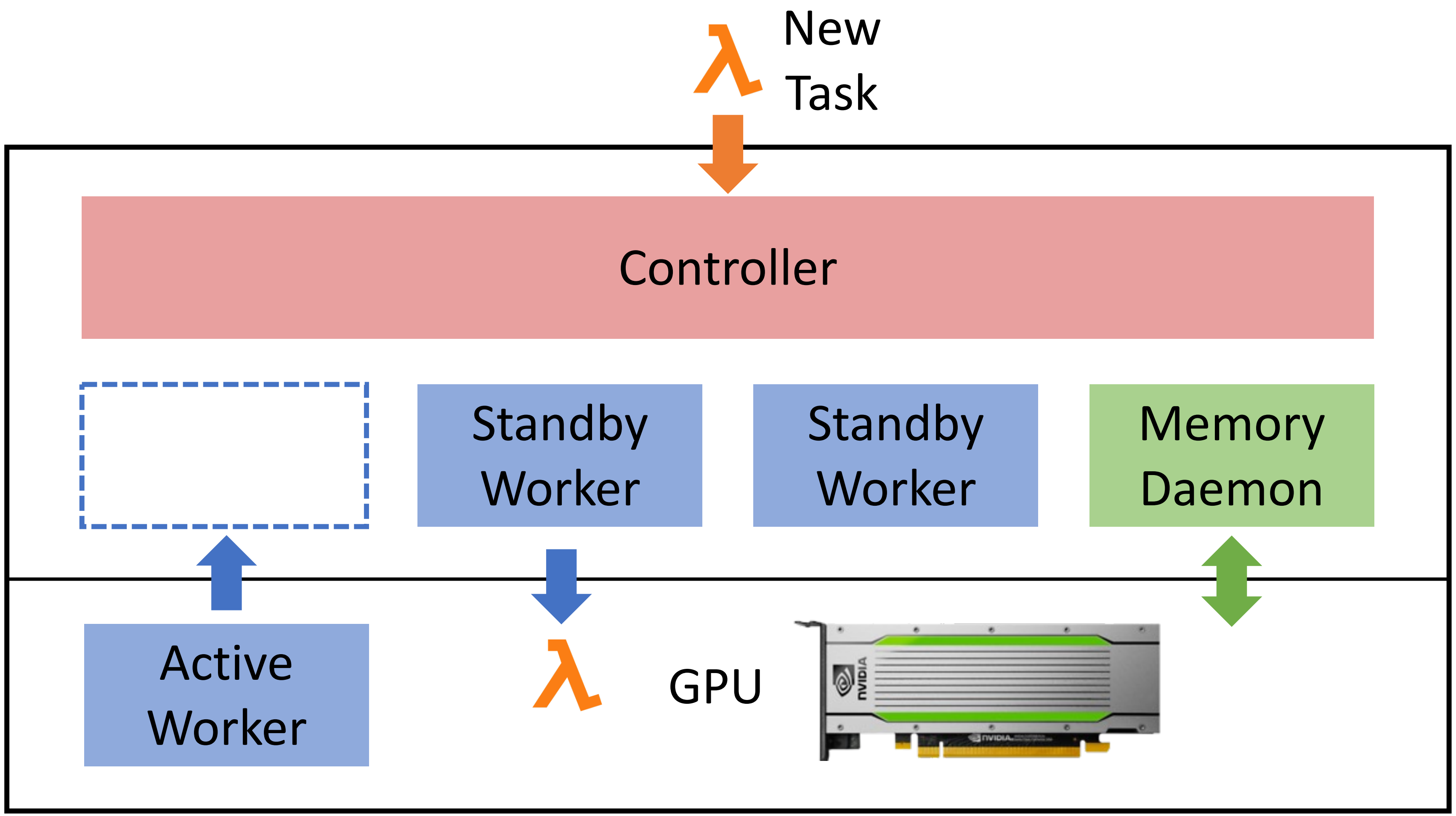
Latency: 6s

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

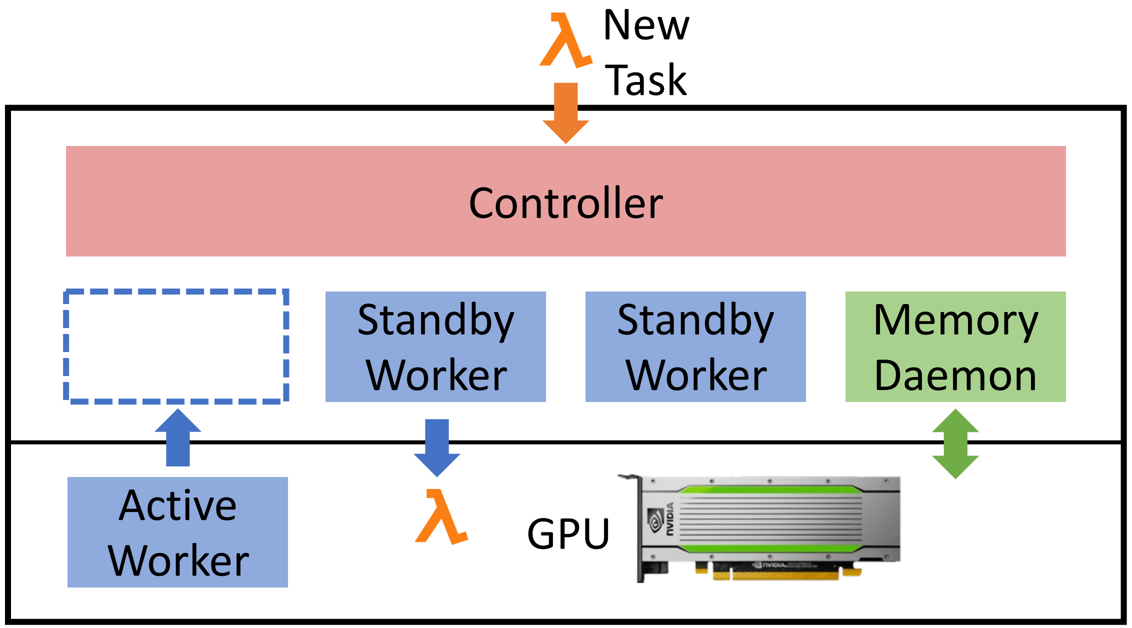
Latency: 6s

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?

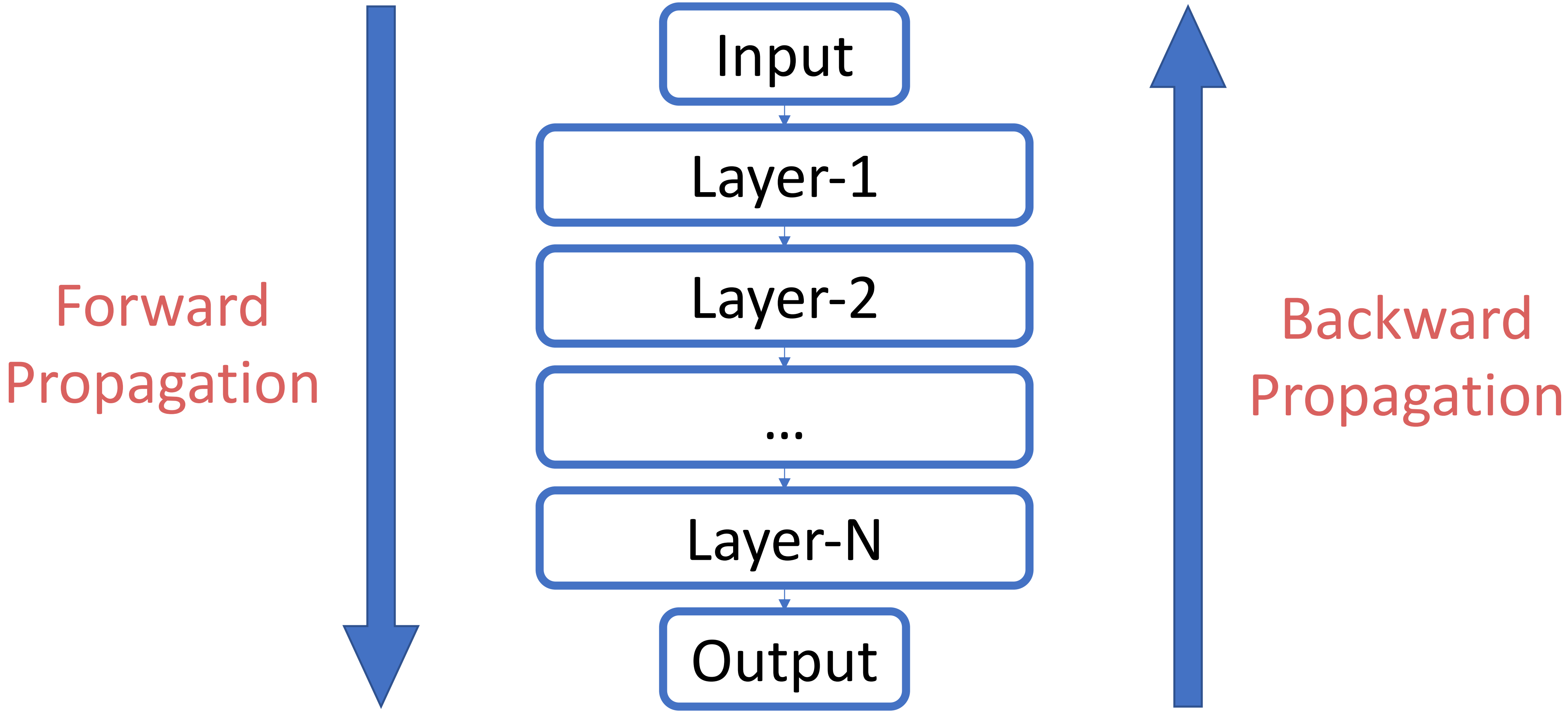
Model transmission → Pipelined model transmission

Memory allocation

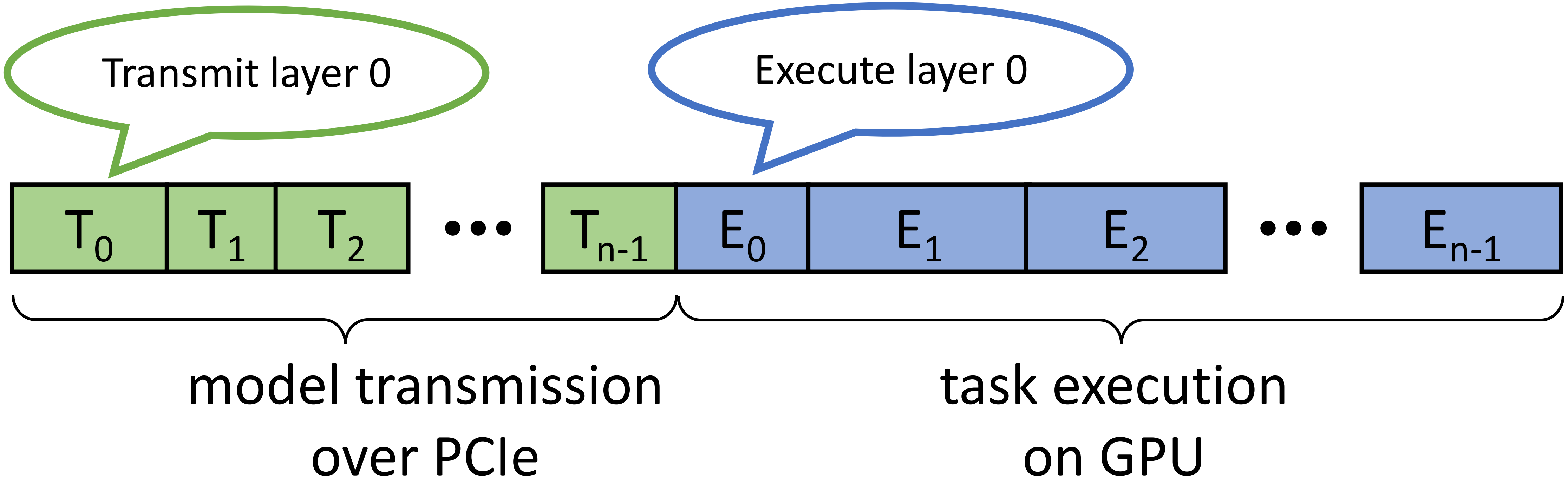
Task initialization

Task cleaning

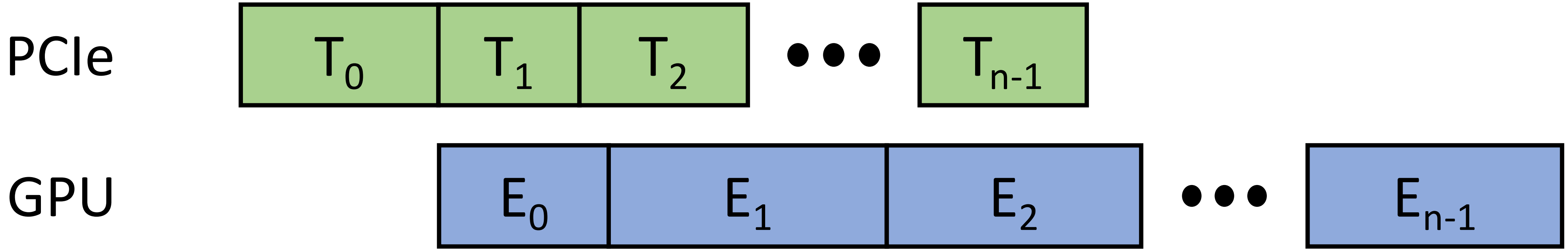
DL models have layered structures



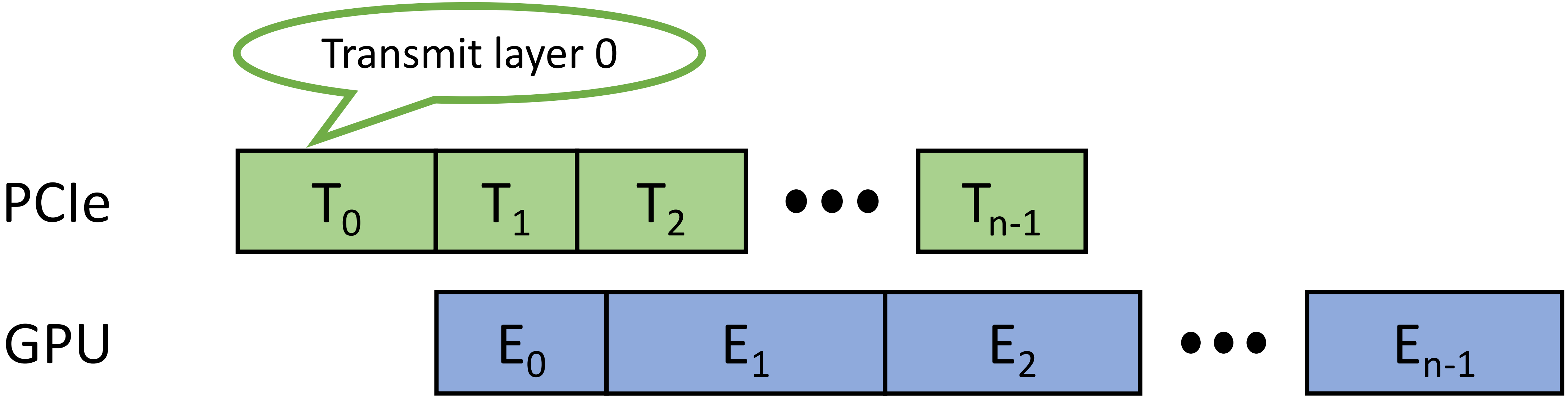
Sequential model transmission and execution



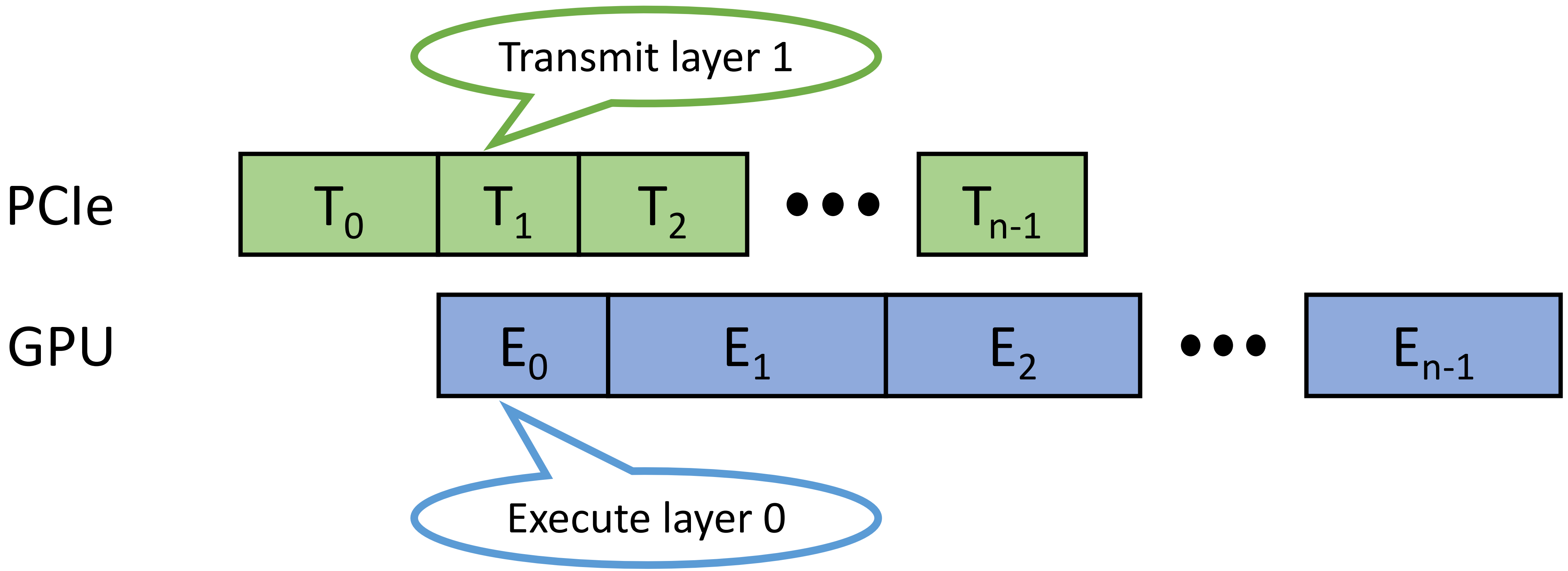
Pipelined model transmission and execution



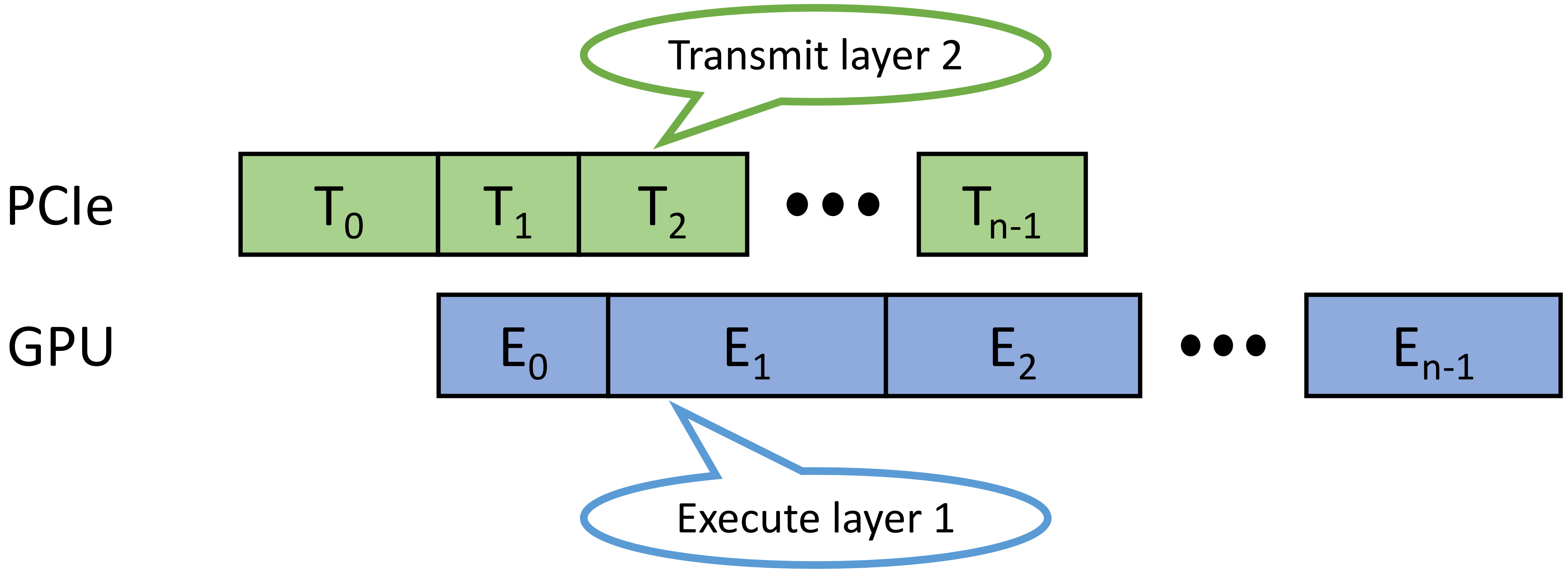
Pipelined model transmission and execution



Pipelined model transmission and execution



Pipelined model transmission and execution



Pipelined model transmission and execution

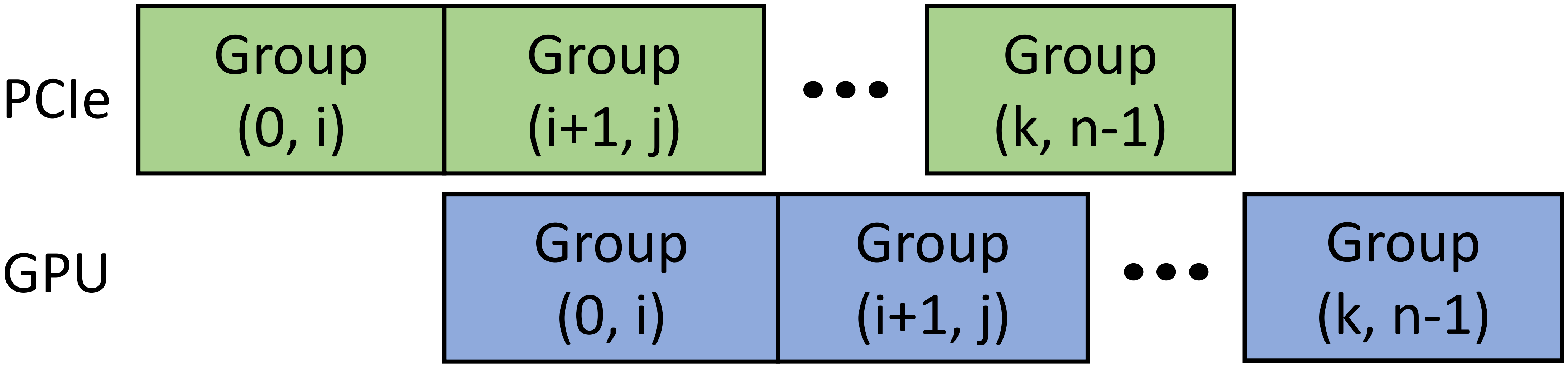
PCIe

GPU

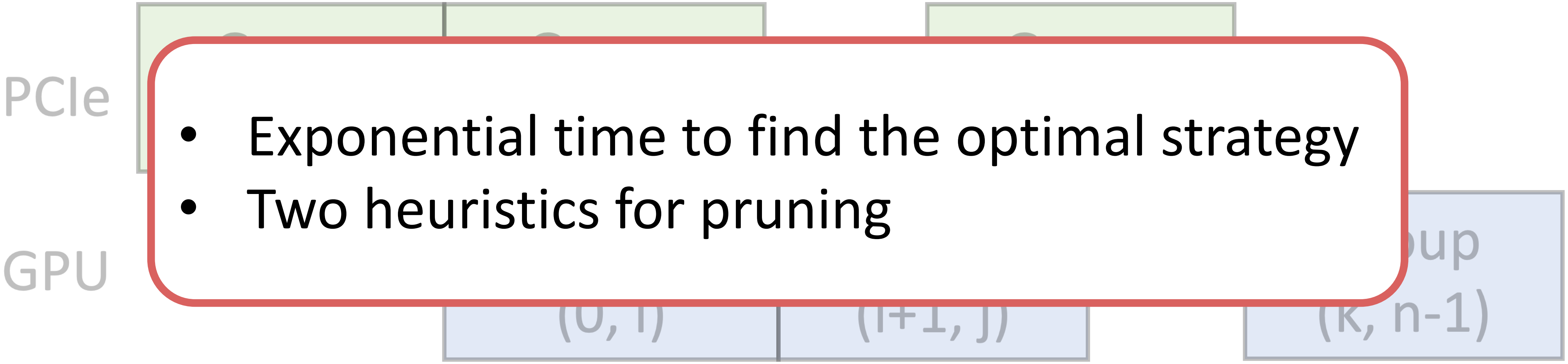
1. Multiple calls to PCIe;
2. Synchronize transmission and execution.

E_{n-1}

Pipelined model transmission and execution



Pipelined model transmission and execution



- Exponential time to find the optimal strategy
- Two heuristics for pruning

How to reduce the overhead?

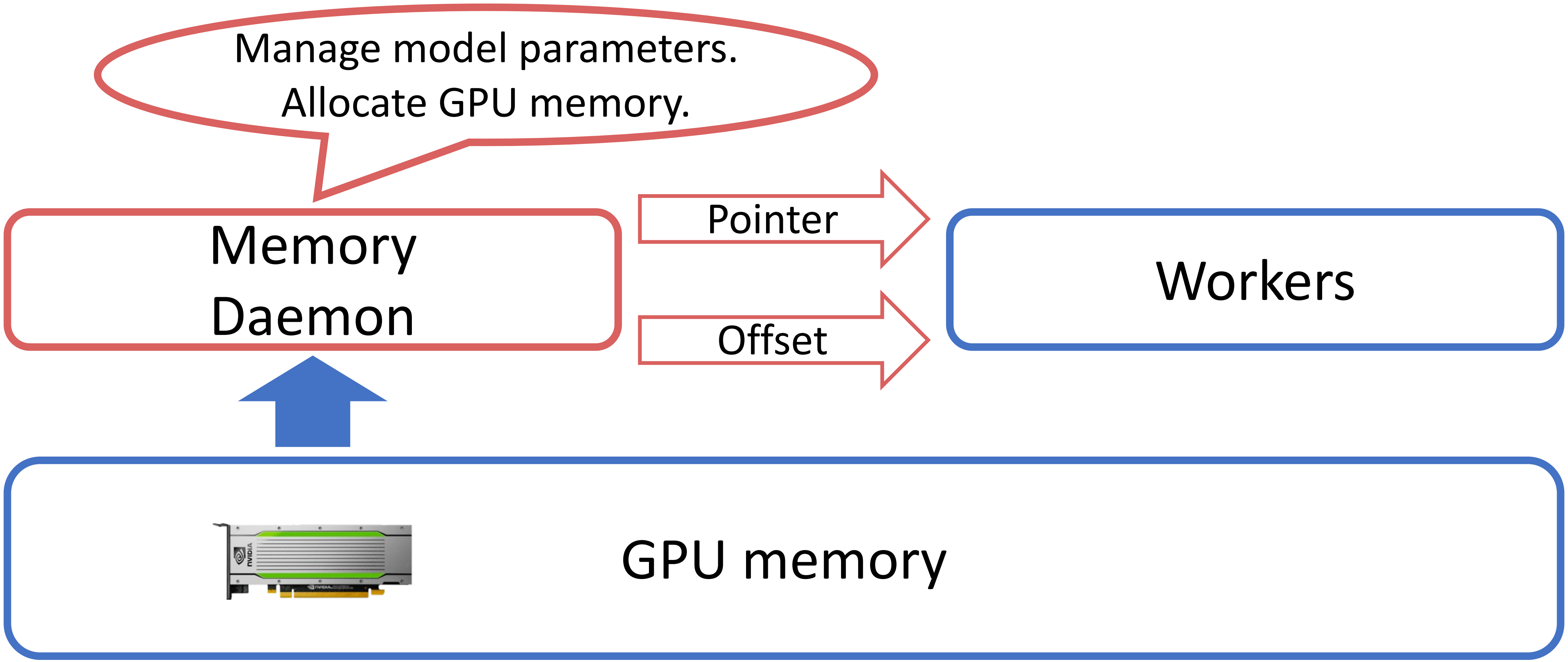
Model transmission → Pipelined model transmission

Memory allocation → Unified memory management

Task initialization

Task cleaning

Unified memory management



How to reduce the overhead?

Model transmission → Pipelined model transmission

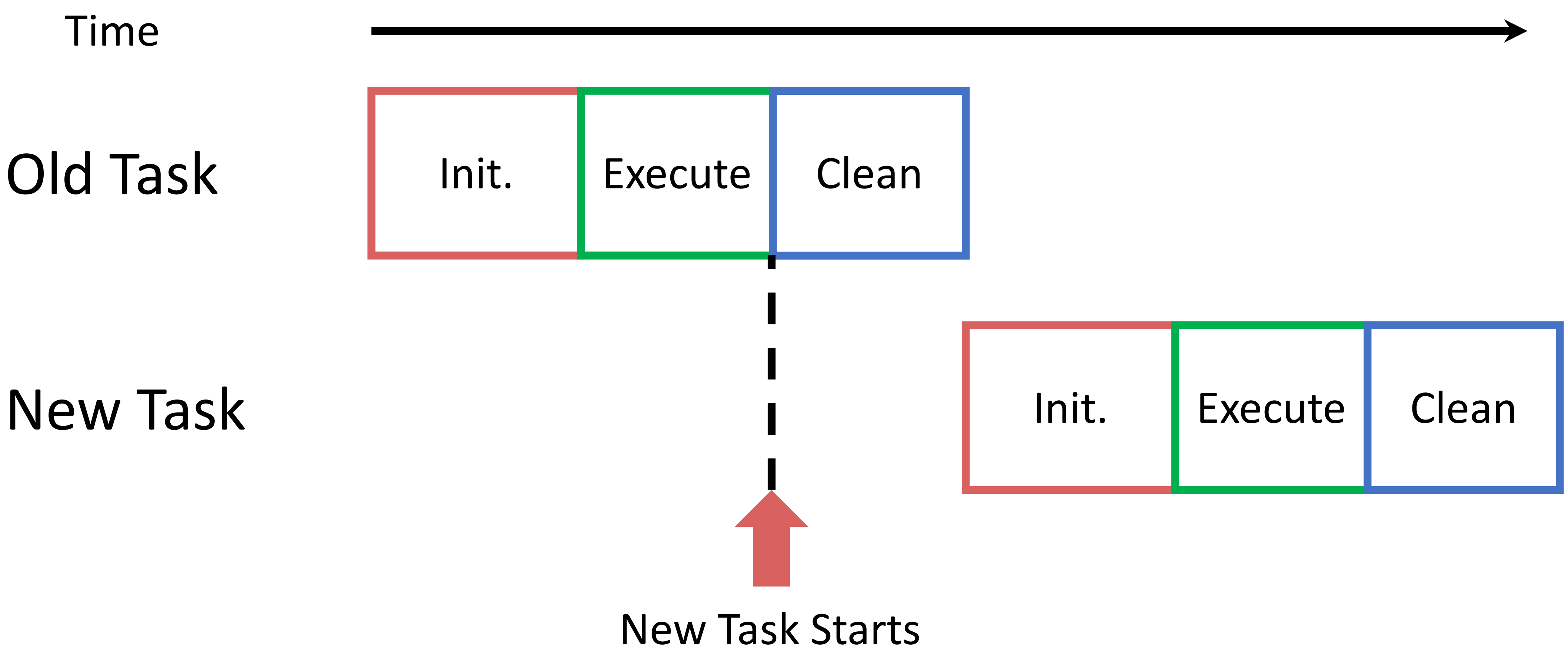
Memory allocation → Unified memory management

Task initialization

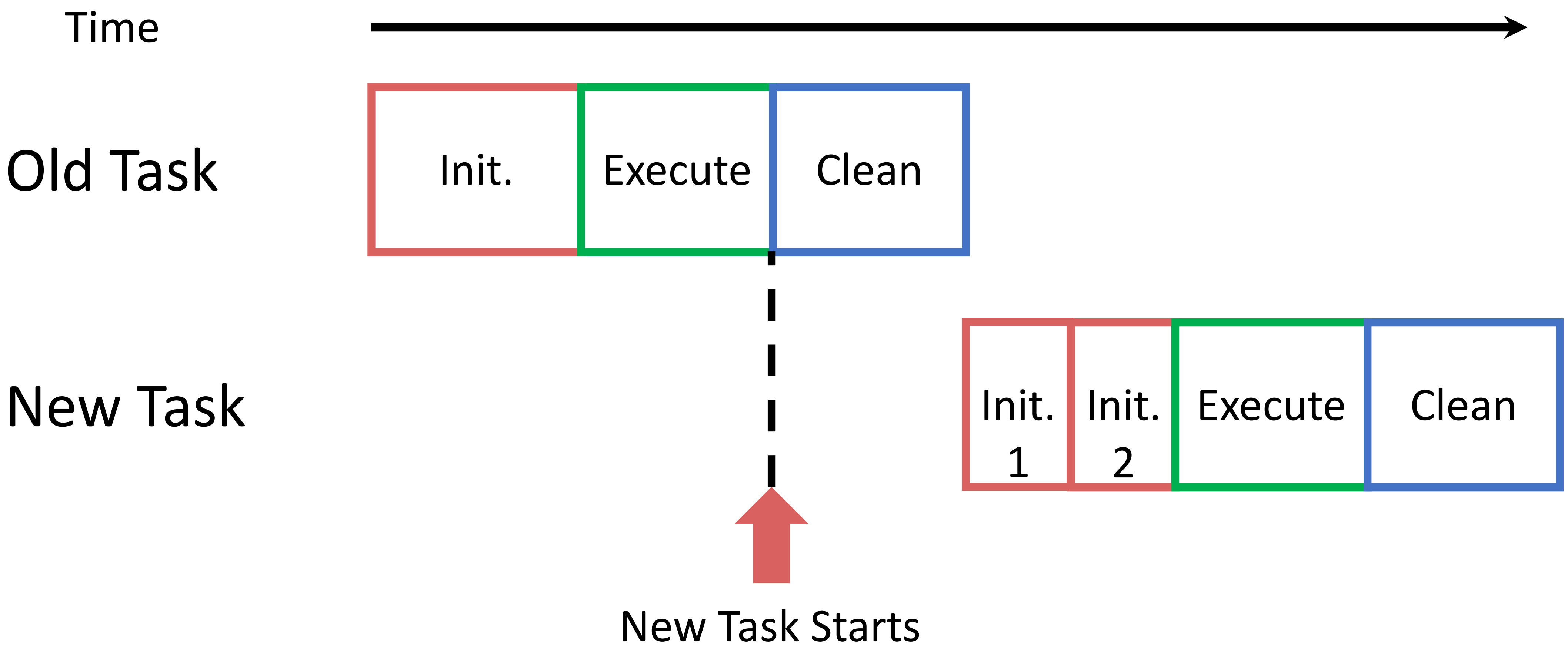
Task cleaning

} Active-standby worker switching

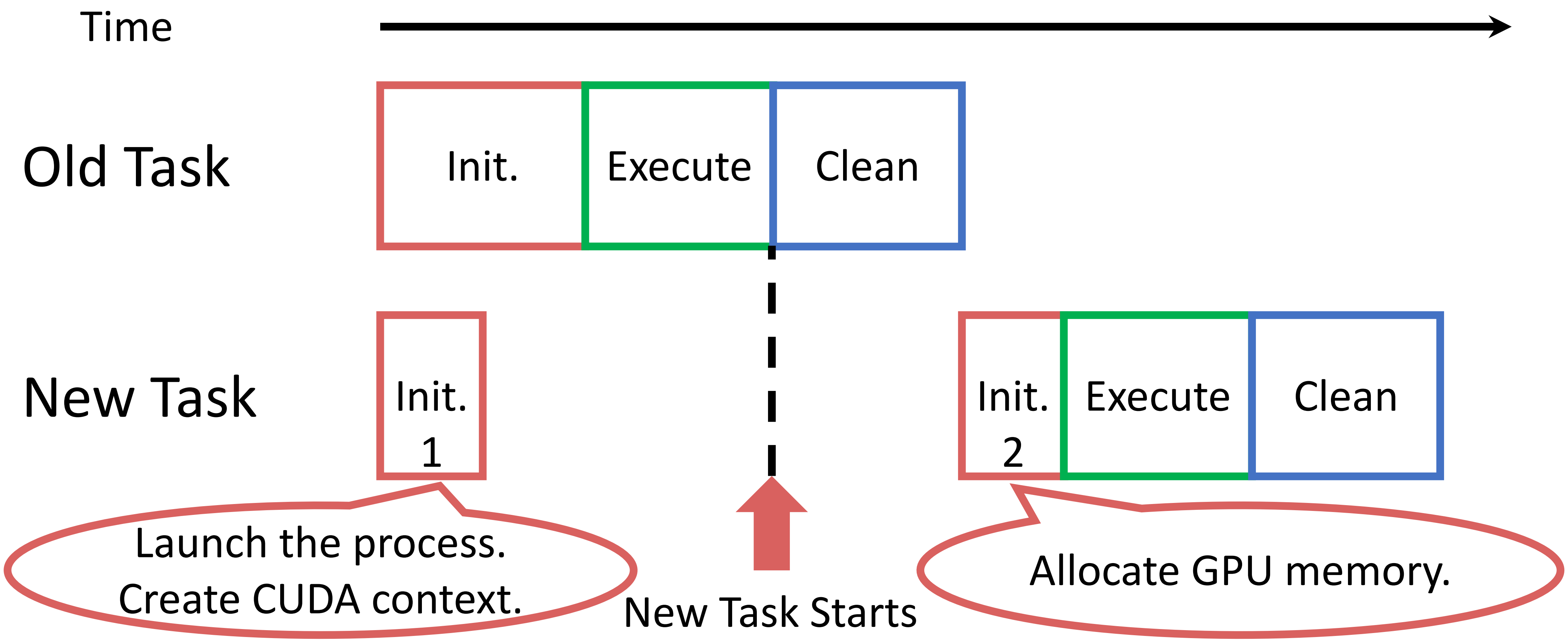
Active-standby worker switching



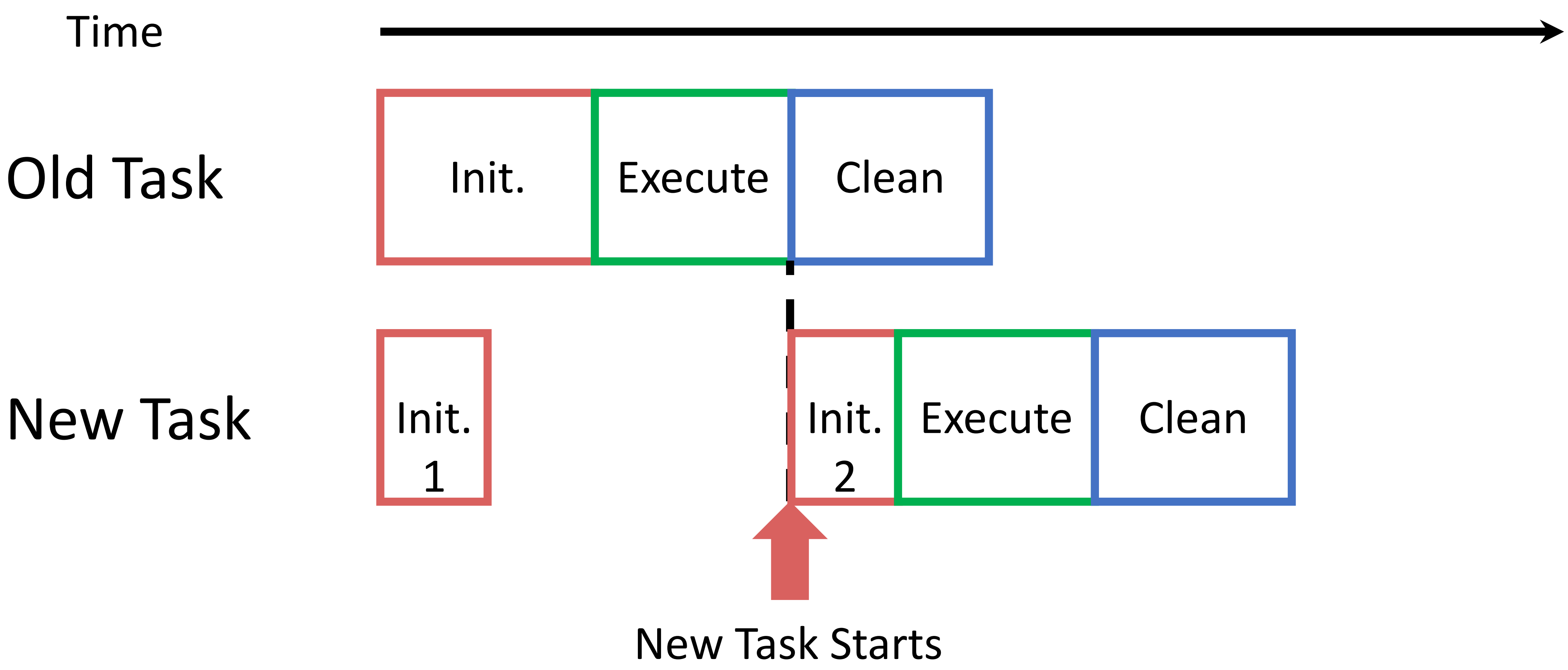
Active-standby worker switching



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

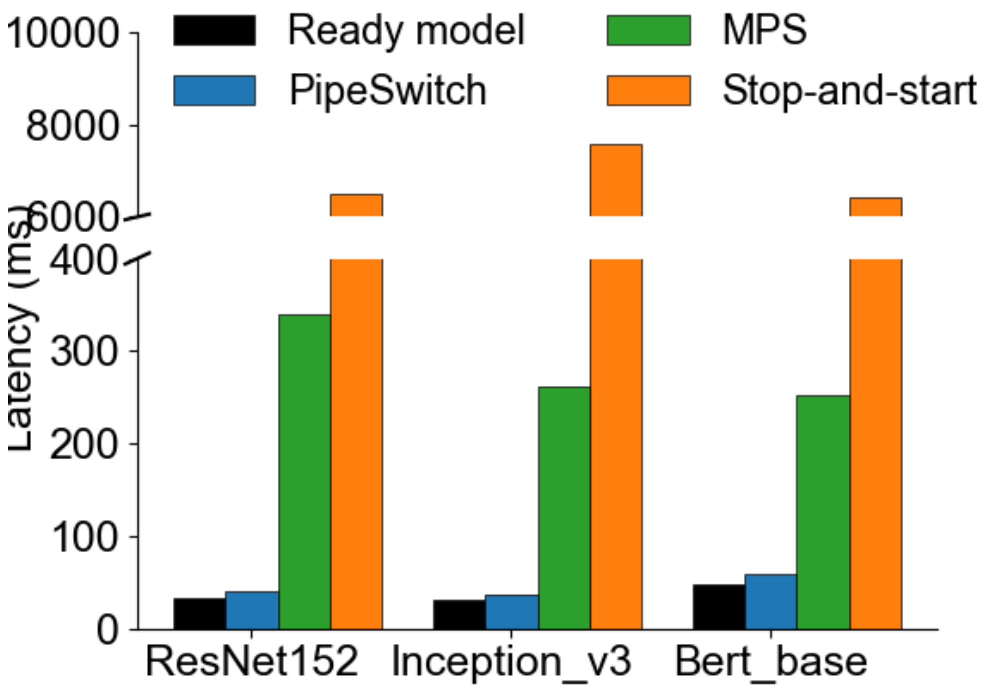
- Can PipeSwitch satisfy SLOs?
- Can PipeSwitch provide high utilization?
- How well do the design choices of PipeSwitch work?

Evaluation

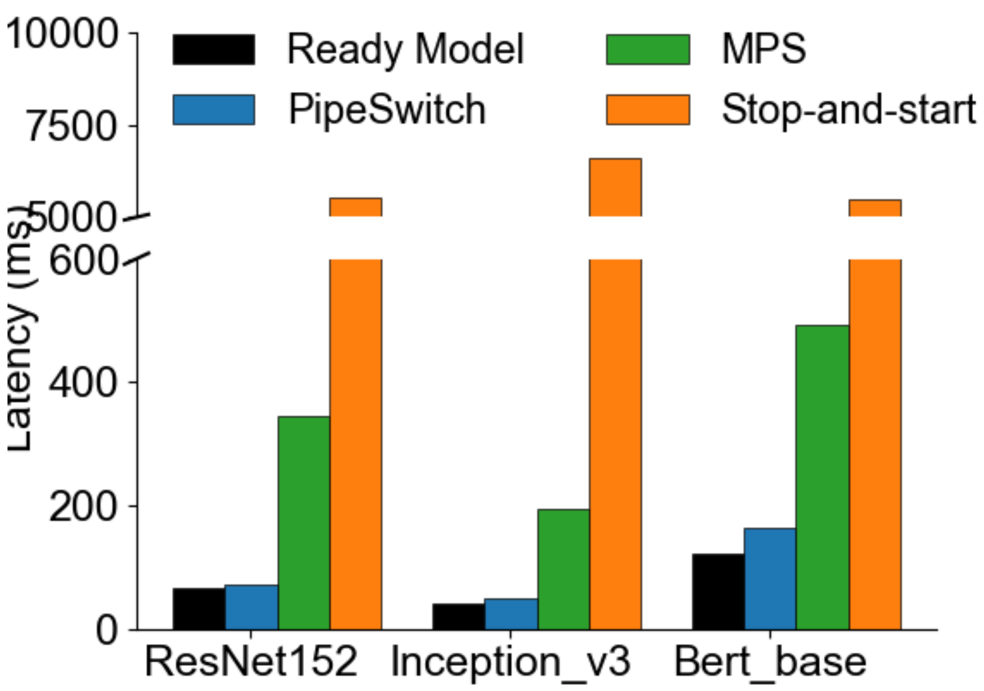
- Can PipeSwitch satisfy SLOs?
- Can PipeSwitch provide high utilization?
- How well do the design choices of PipeSwitch work?

PipeSwitch satisfies SLOs

NVIDIA Tesla V100

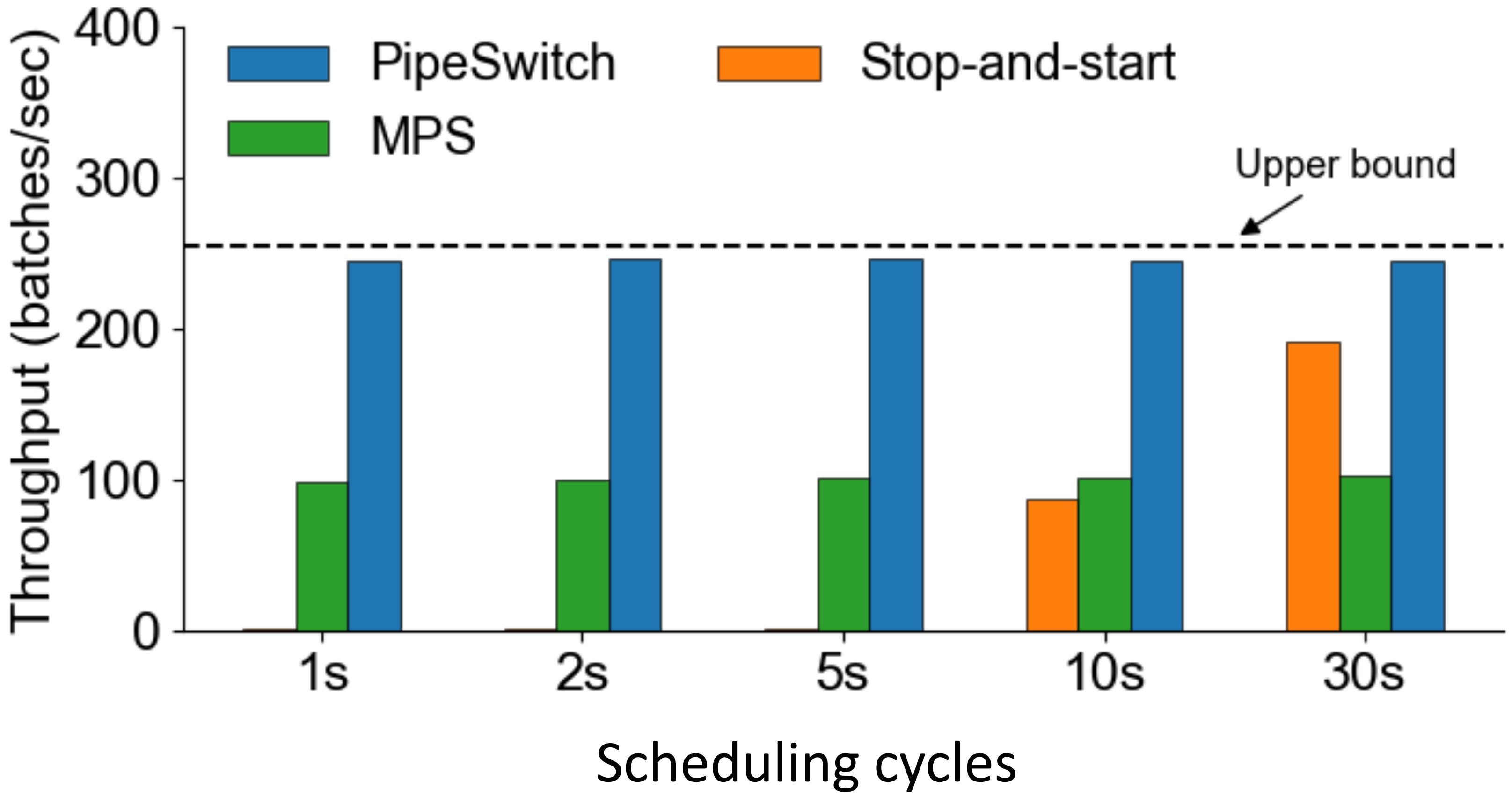


NVIDIA Tesla T4



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