6.5810: Virtualization + Dune

Adam Belay <abelay@mit.edu>

Logistics

- Sign up for paper presentations
  - Due date has been updated to Sunday (9/18) to give more time
- If you haven’t already, create a CloudLab account

Plan for today

1. Dune:

- Exposes privileged CPU features, normally used to build kernels, safely to userspace

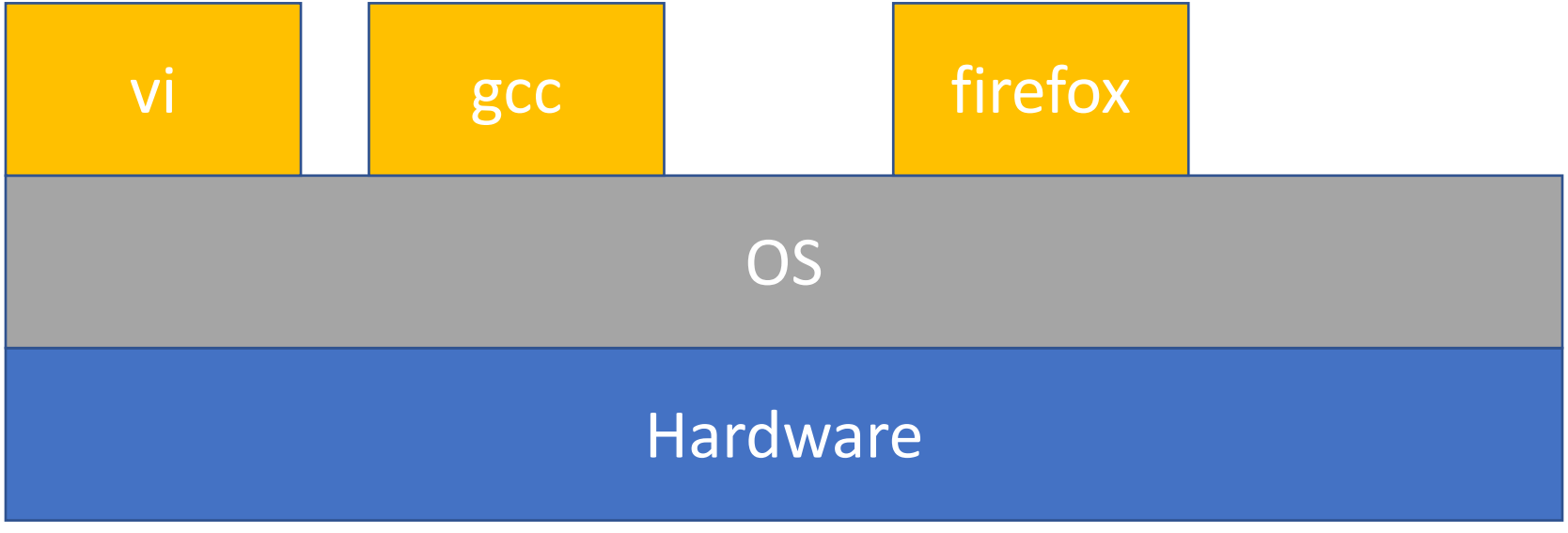
2. IO Virtualization:

- Allows programs to communicate with I/O hardware (e.g., networking and storage) directly and safely (not violating memory isolation)

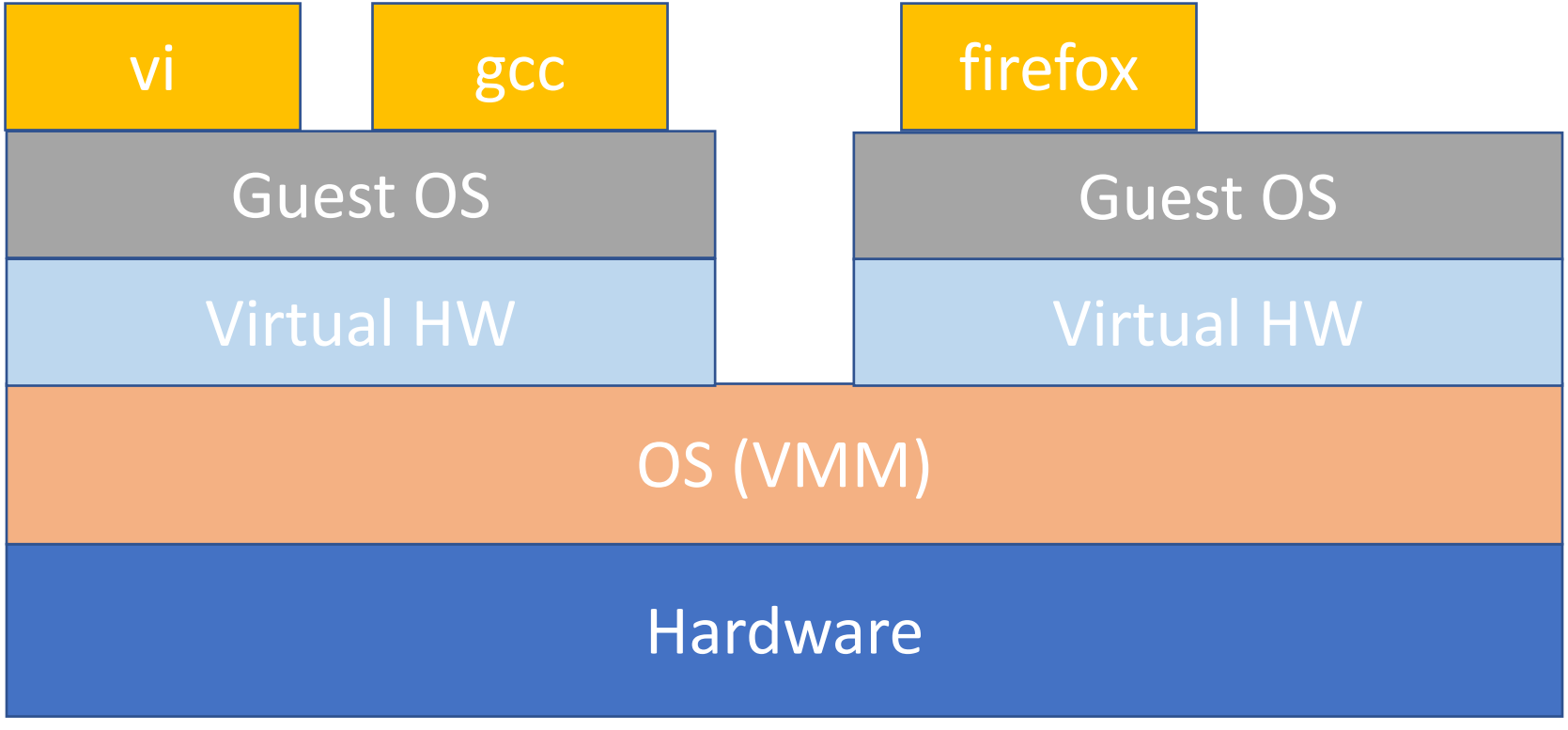
3. AIFM:

- Application-integrated far memory

Recap: Process Architecture



Recap: VM Architecture



- What if the process abstraction looked just like HW?

Comparing a process and HW

Process

- Non privileged registers and instructions
- Virtual memory
- Signals
- File system and sockets

Hardware

- All registers and instructions
- Virt. mem. and MMU
- Traps and interrupts
- I/O devices and DMA

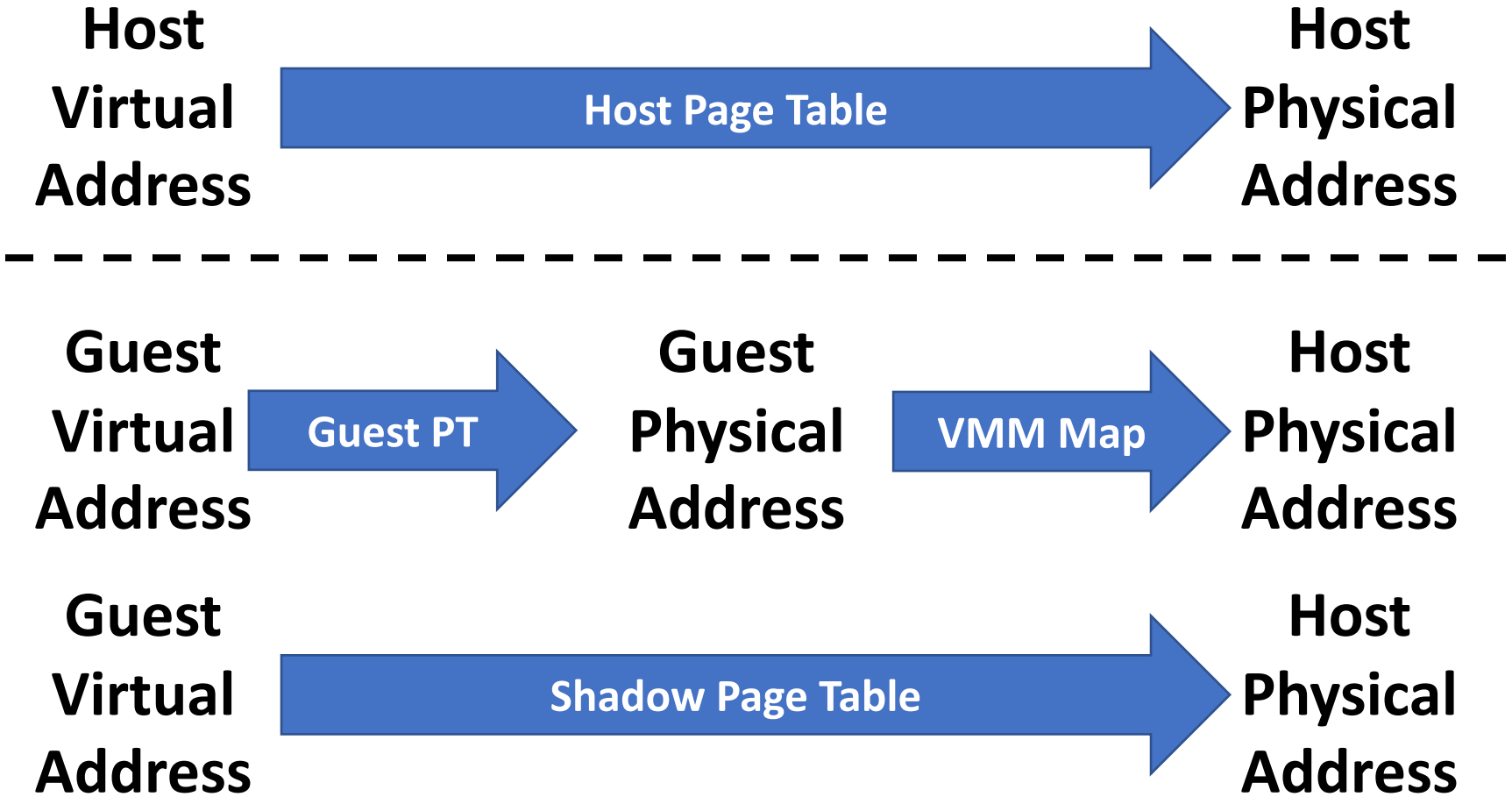
Can a CPU be virtualized?

Requirements to be “virtualizable” defined by Popek and Goldberg in 1974:

1. **Fidelity**: Software on the VMM executes identically to its execution on hardware, barring timing effects.
2. **Performance**: An overwhelming majority of guest instructions are executed by the hardware without the intervention of the VMM.
3. **Safety**: The VMM manages all hardware resources.

Memory virtualization

Memory virtualization



Why can’t the VMM let the VM guest kernel program the page table directly?

Trap-and-emulate is not possible on x86

Two problems:

1. Some instructions behave differently in user mode instead of trapping
2. Some registers leak state that reveals if the CPU is running in usermode

- Violates ***fidelity*** property
- Risc-V doesn’t have this problem!

Two possible solutions

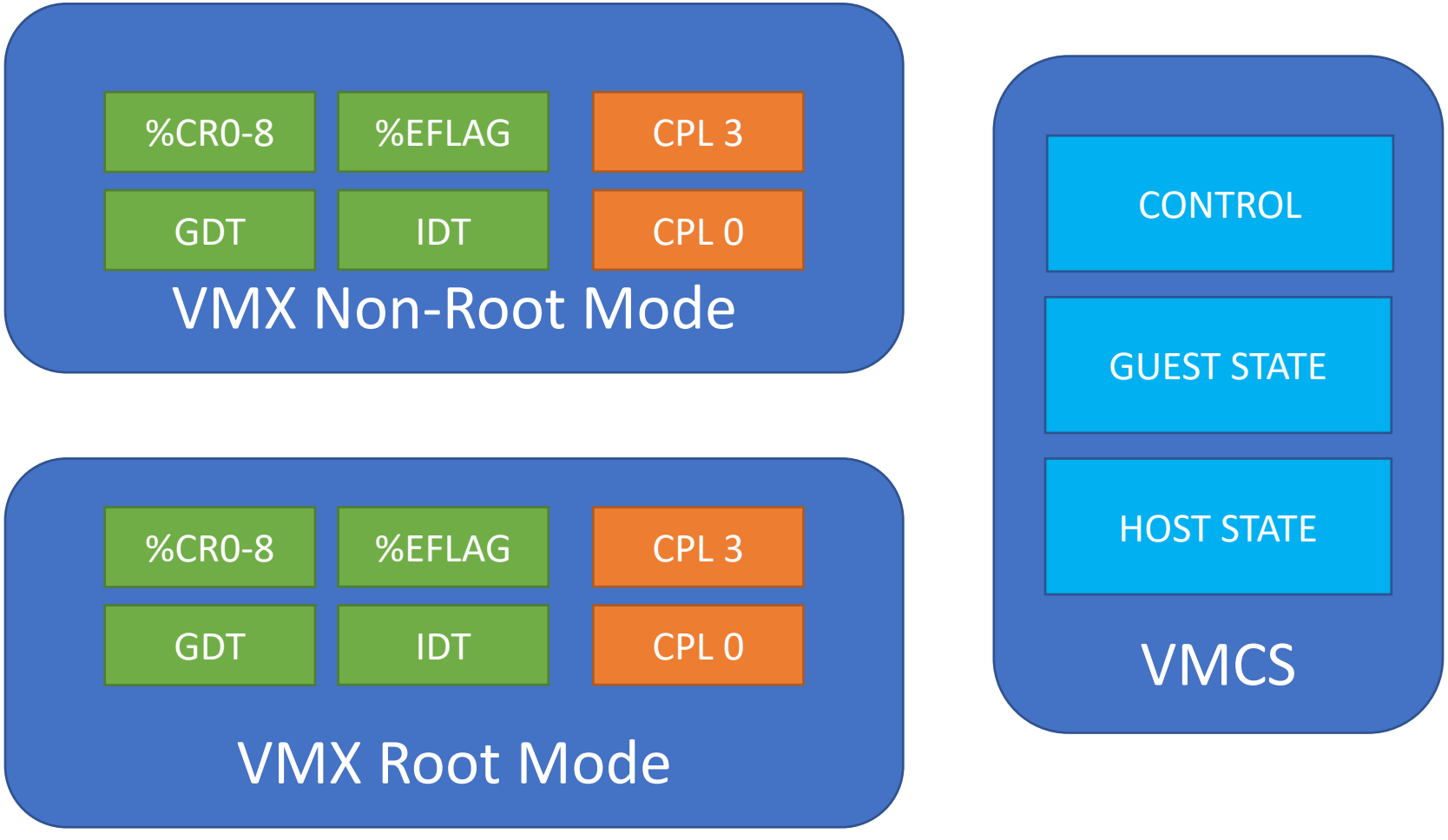
1. Binary translation
   - Rewrite offending instructions to behave correctly

2. Hardware virtualization
   - CPU maintains shadow state internally and directly executes privileged guest instructions

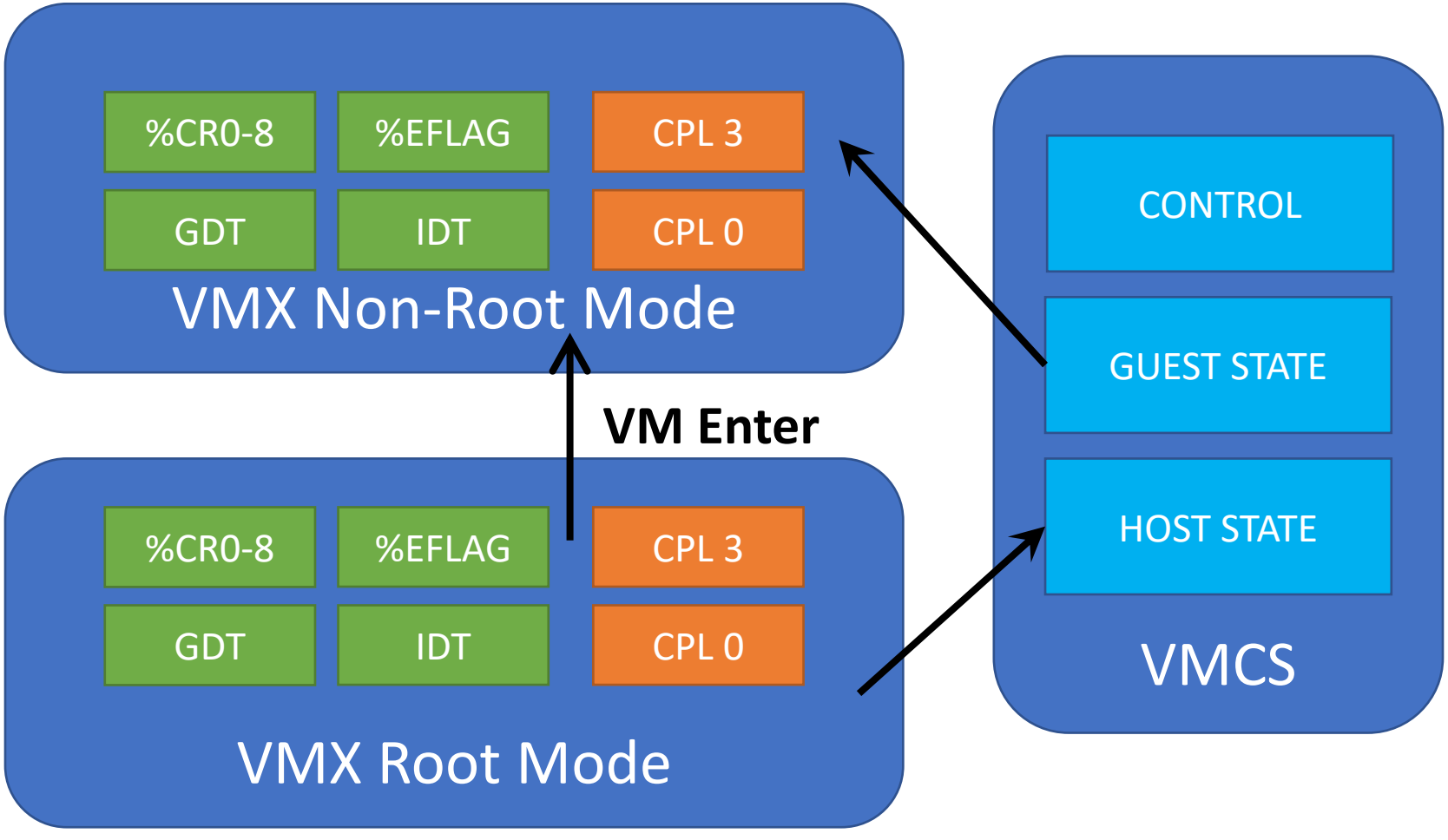
Intel VT-x

- Makes x86 hardware “virtualizable” under Popek and Goldberg definition
- Goal: **Direct execution** of most privileged instructions
- Introduces two CPU modes, kind of like ring protection
  - VMX Root Mode: For running VMM (host)
  - VMX Non-root Mode: For running VMs (guest)
  - But each mode has its own rings (user/kernel)
- In-memory structure called VMCS stores privileged register state and control flags

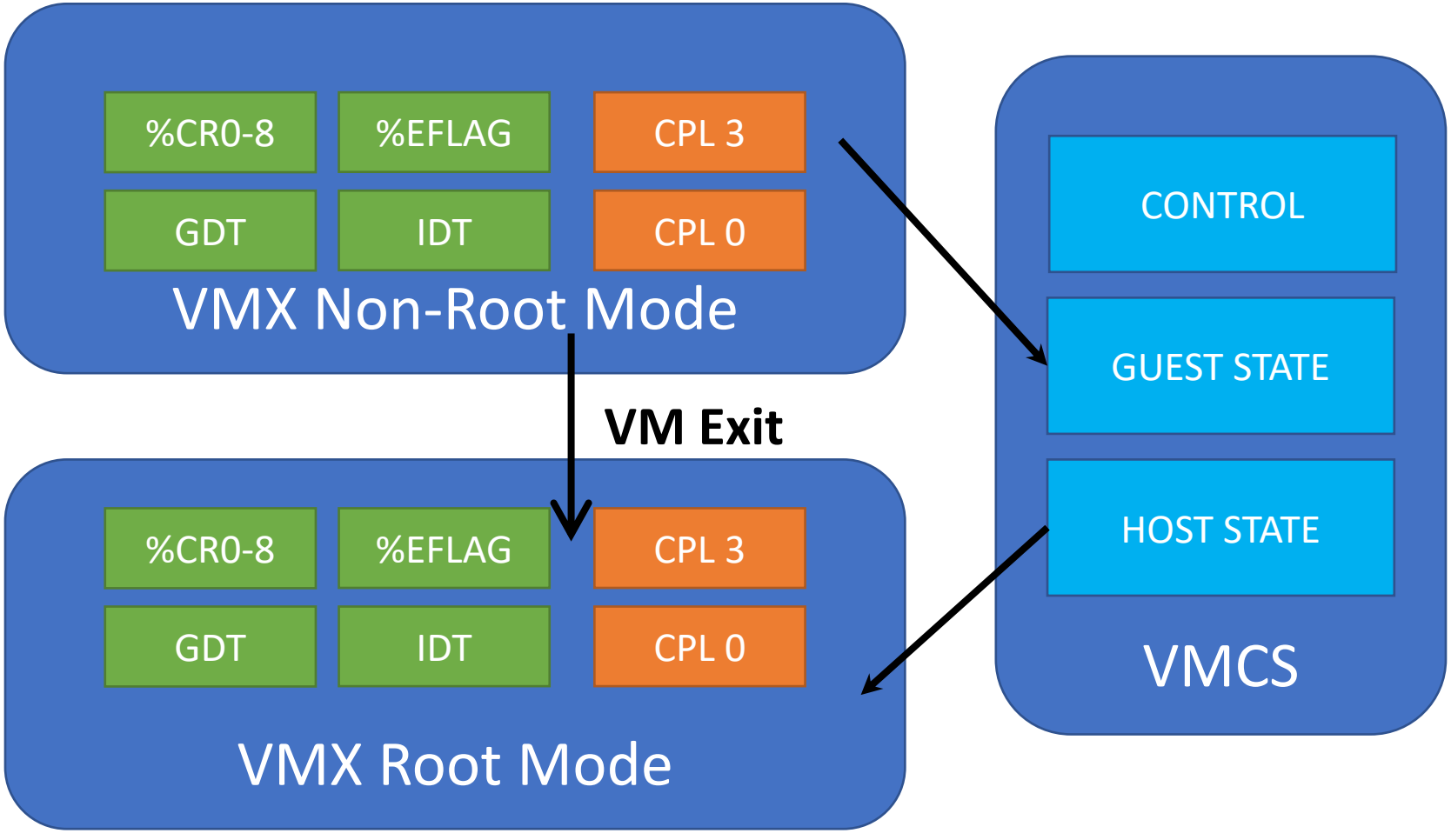
Intel VT-x


%CR0-8
%EFLAG
CPL 3
GDT
IDT
CPL 0
VMX Non-Root Mode
%CR0-8
%EFLAG
CPL 3
GDT
IDT
CPL 0
VMX Root Mode
CONTROL
GUEST STATE
HOST STATE
VMCS

Intel VT-x: VM Enter



Intel VT-x: VM Exit

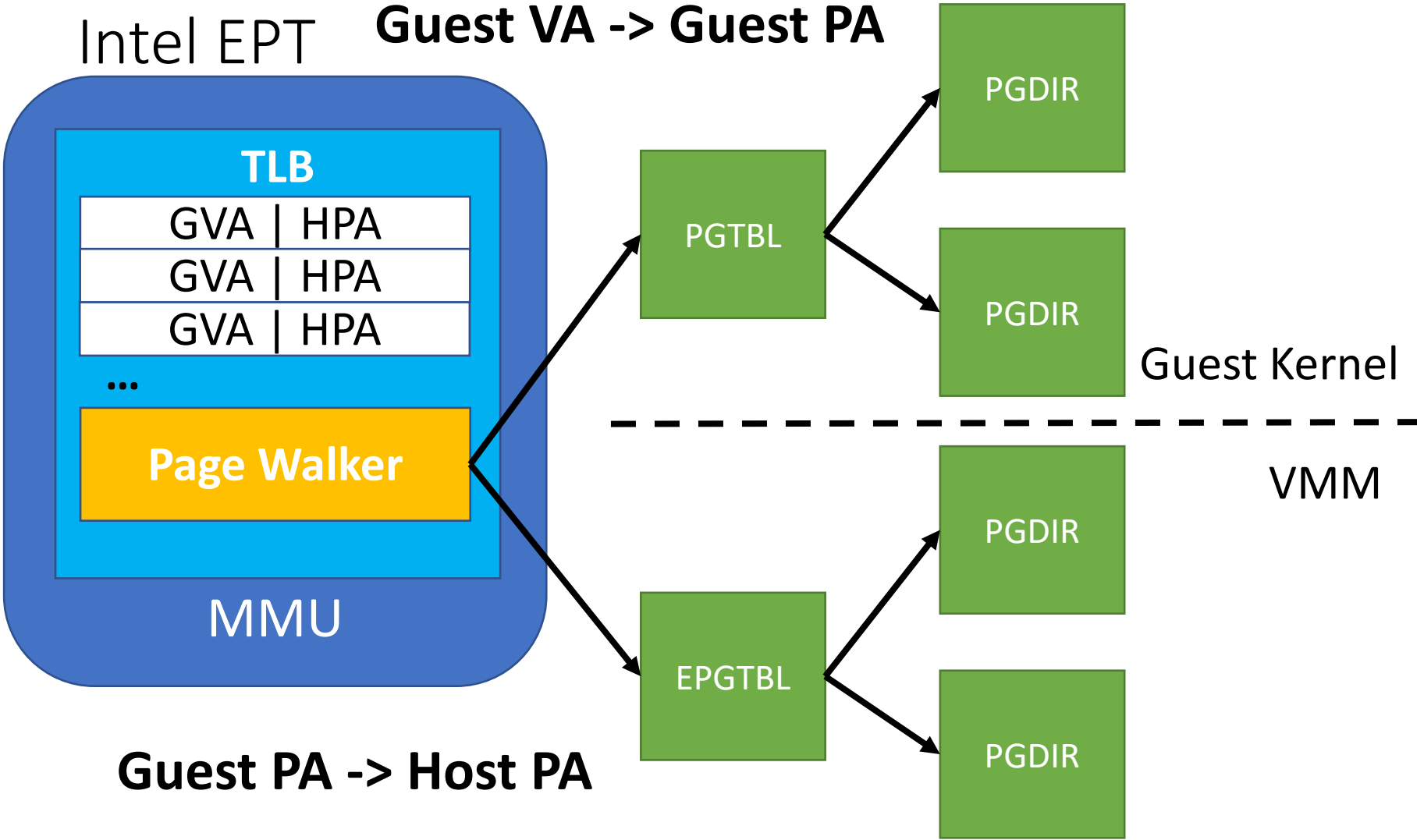


VM Enter and VM Exit

- Transitions between VMX Root Mode and VMX Non-root Mode
- VM Exit
  - VMCALL instruction, EPT Page Faults, some trap and emulate (configured in VMCS)
- VM Enter
  - VMLAUNCH instruction: Enter VMX Non-root Mode for a new VMCS
  - VMRESUME instruction: Enter VMX Non-root Mode for the last VMCS (faster)
- Typical VM Exit/Enter is ~200 cycles on modern HW

Intel EPT (nested paging)

- Goal: **Direct execution** of guest page table interactions
  - Reads and write to page table in memory
  - mov %eax, %cr3, INVLPG, etc.
- Idea: Maintain two layers of paging translation
  - Normal page table: Guest-virtual to guest-physical
  - EPT: guest-physical to host-physical

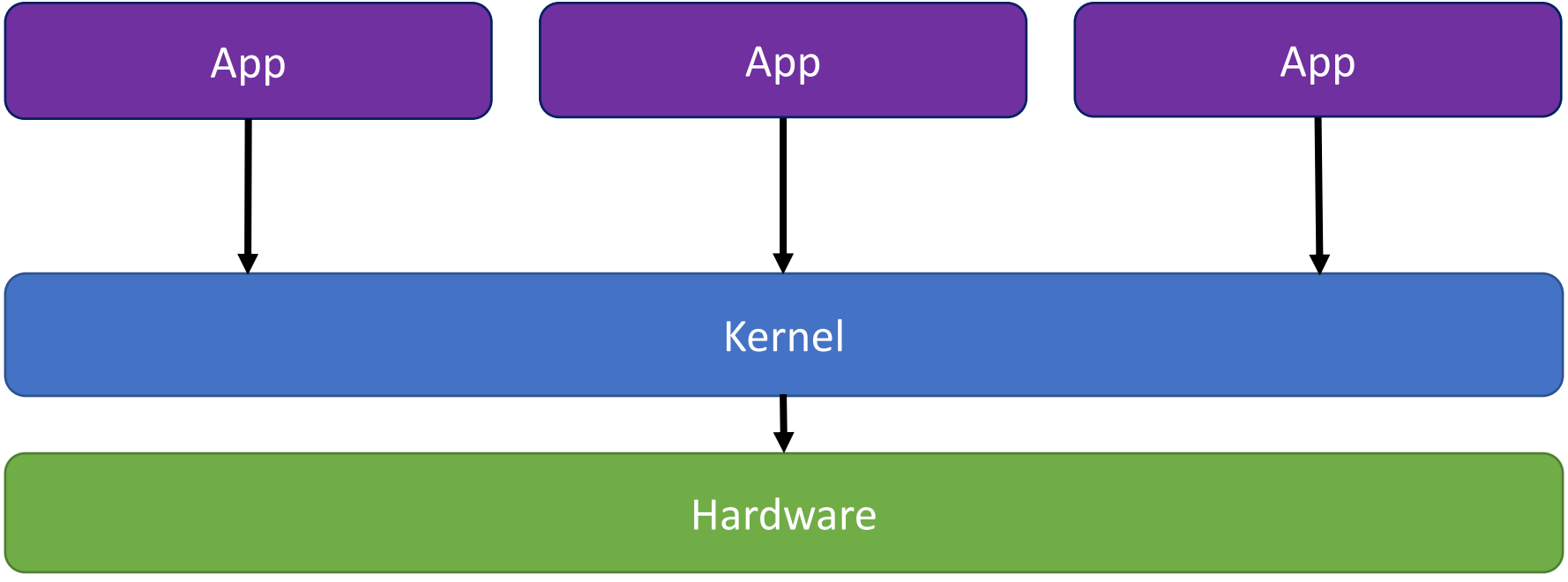

Intel EPT
Guest VA -> Guest PA
TLB
GVA | HPA
GVA | HPA
GVA | HPA
...
Page Walker
MMU
PGTBL
PGDIR
PGDIR
Guest Kernel
VMM
EPGTBL
PGDIR
PGDIR
Guest PA -> Host PA

Q: What’s faster EPT or Shadow Page Tables?

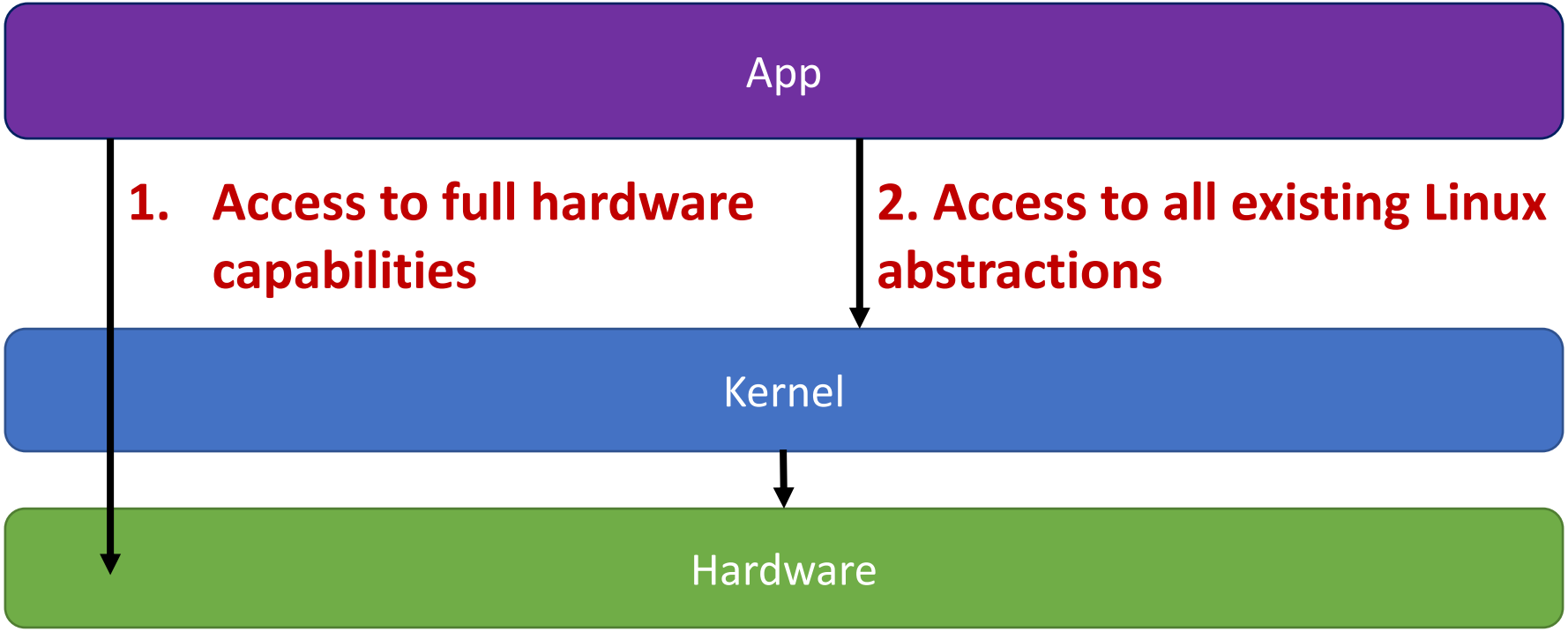
Big picture

- Direct execution reduces overhead
  - Avoids VM exits, trap-and-emulate, binary translation
- Enabled by microarchitectural changes:
  - Intel VT-x: direct execution of most privileged instructions (e.g. IDT, GDT, ring protection, EFLAG, etc.)
  - Intel EPT: direct execution of page table manipulation

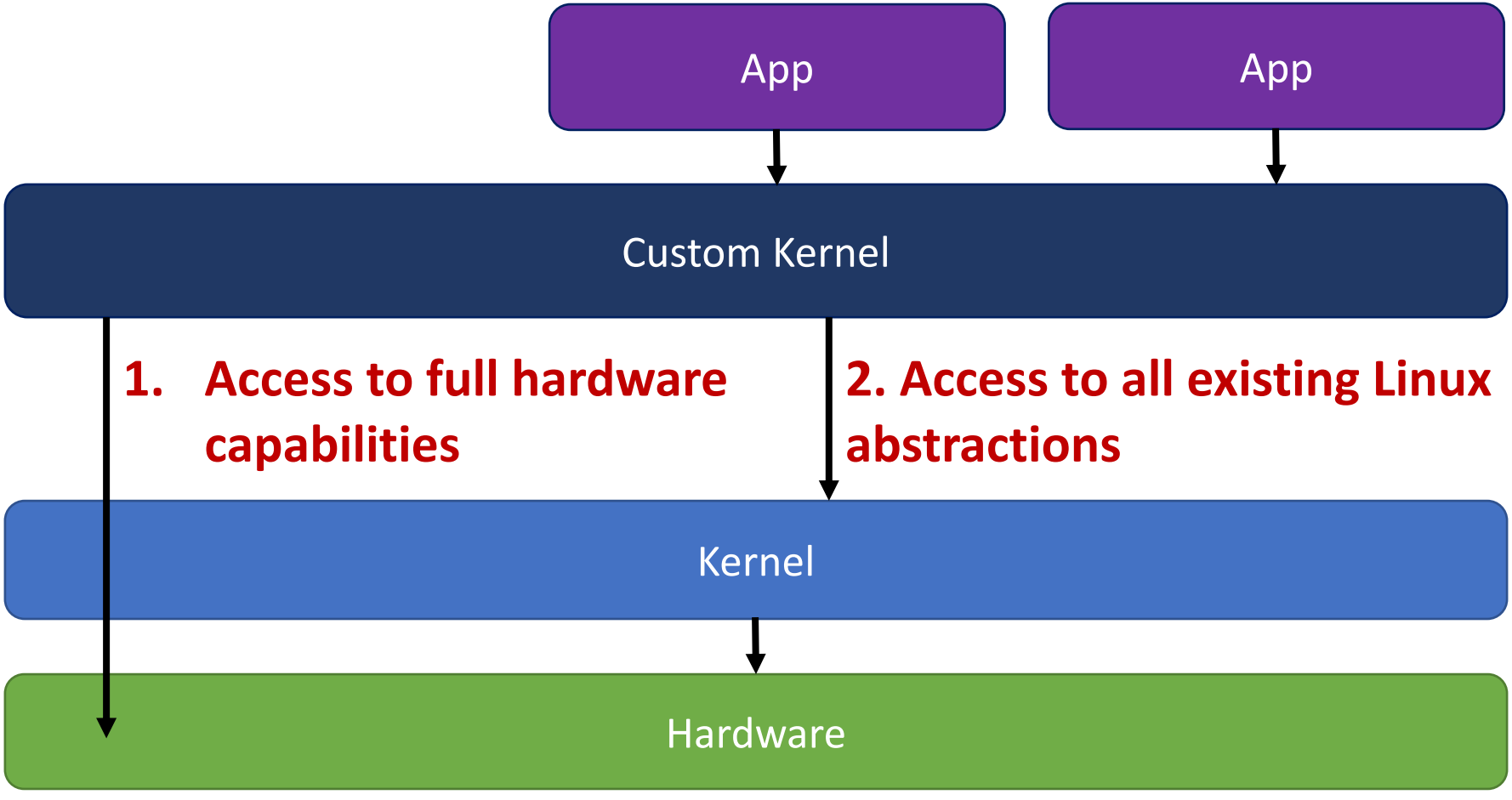
Operating systems today



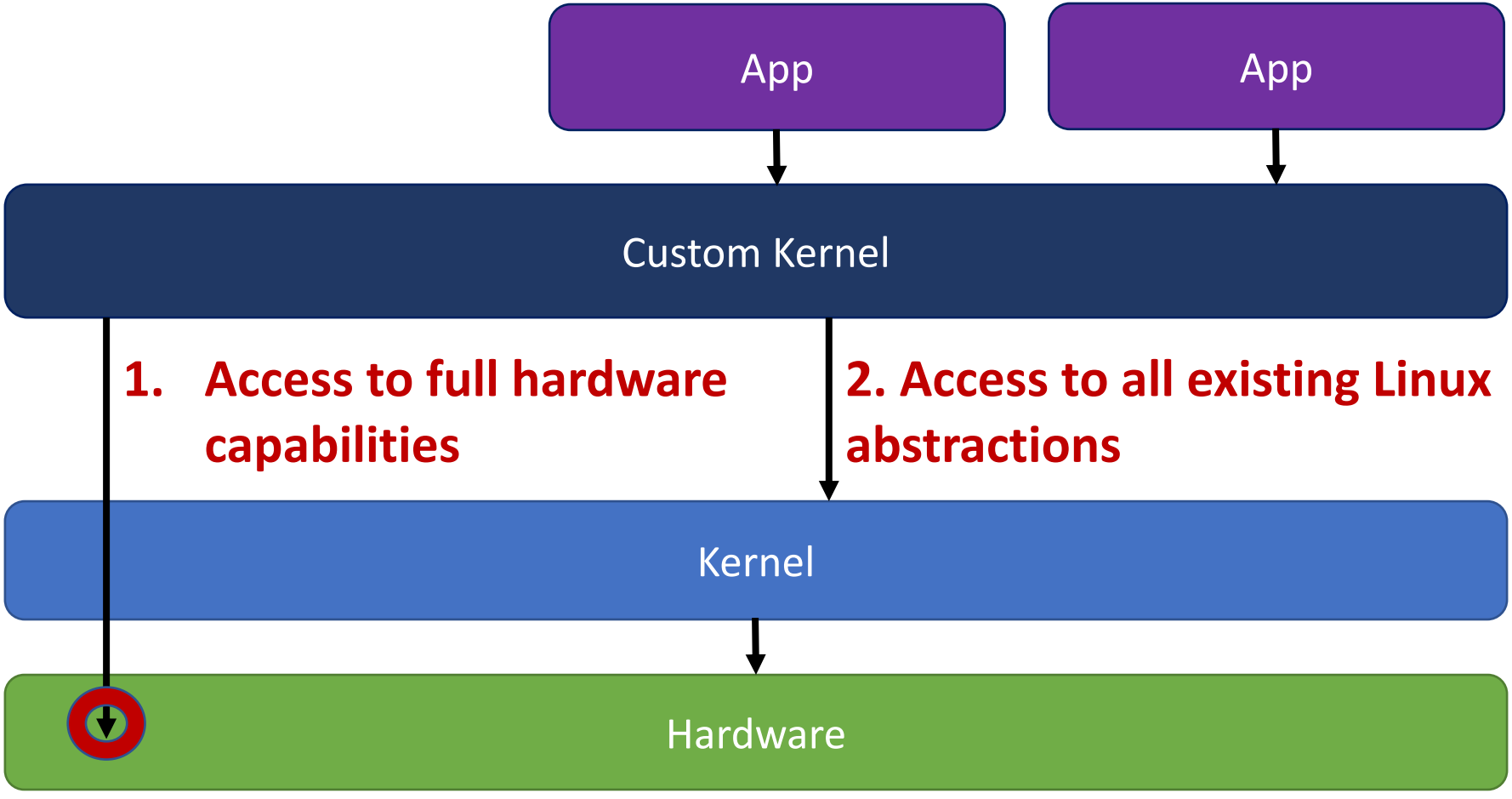
What if you could give a process access to raw hardware?



Could build new OS on top of Linux



But still must maintain process isolation



Dune

- Key Idea: Use VT-x, EPT, etc. to support Linux processes instead of virtual machines
- Dune is loadable kernel module, makes it possible for an ordinary Linux process to switch to “Dune mode”
- Dune mode processes can run along side ordinary processes.

A dune process

- Is still a process
  - has memory, can make Linux system calls, is fully isolated, etc.
- But isolated with VT-x Non-root mode
  - Rather than with CPL=3 and page table protections
- memory protection via EPT
  - Dune configures EPT so process can only access the same physical pages it would normally have access to

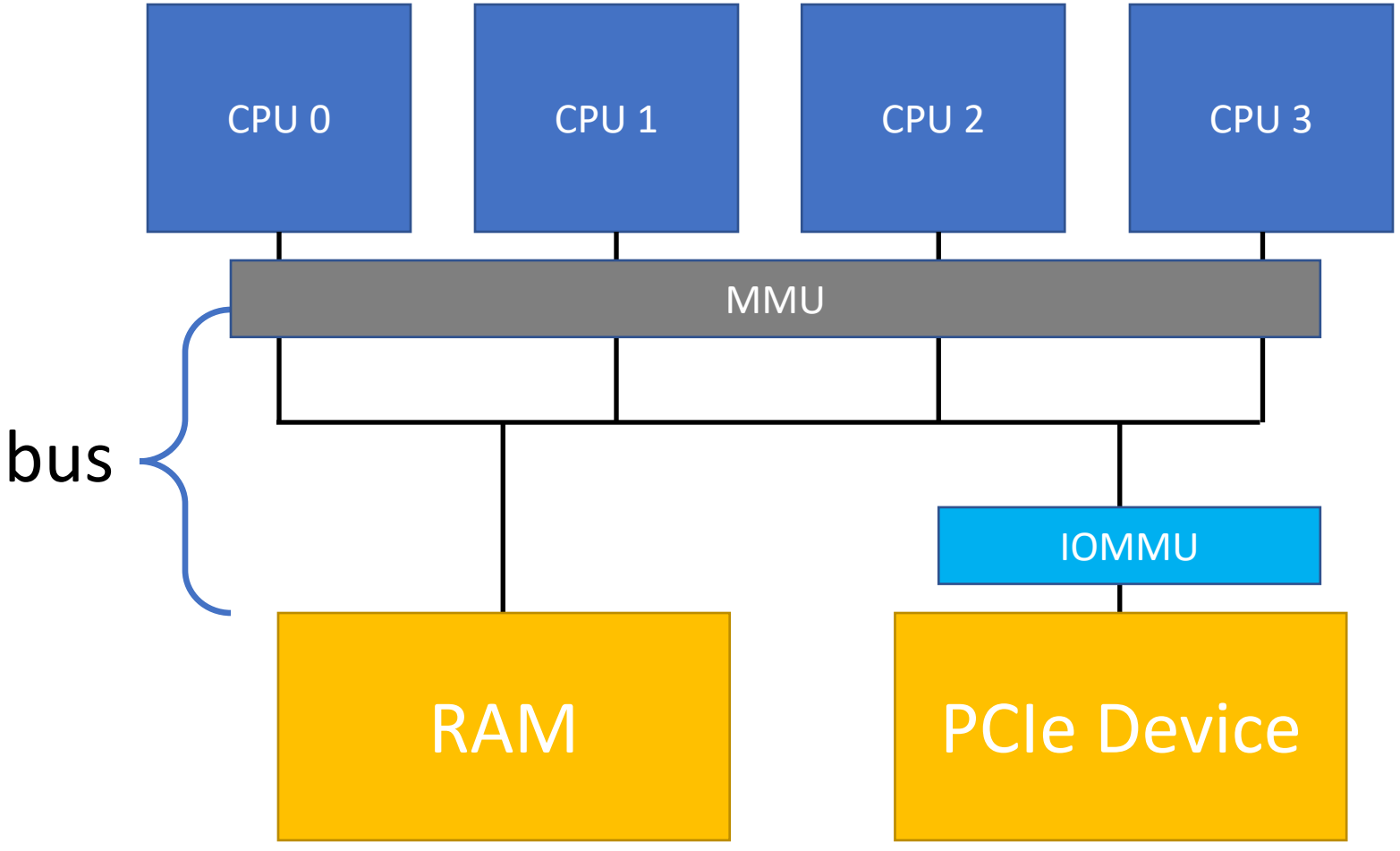
Why isolate a process with VT-x?

- Process can access all of Linux environment while also directly executing most privileged instructions
- User code now runs at CPL 0
- Process can manage its own page table via %CR3
- Fast exceptions (e.g. page faults) via shadow IDT
  - Kernel crossings eliminated
- Can run sandboxed code at CPL 3
  - So process can act like a kernel!

SR-IOV + IOMMU

- Goal: Allows **direction execution** of I/O device access
- Challenge #1: How to partition a single device into multiple instances
- Challenge #2: How to prevent DMA from overwriting memory belonging to VMM or another guest

IOMMU



Major challenges in practice

1. Memory: Traditionally must be pinned (cannot be swapped)
   - Pinning and unpinning is very expensive
2. Completions: Must be busy-polled (wastes CPU) or interrupt driven (high overhead)
   - Caladan offers a solution to this problem (next week)
3. Scalability: SR-IOV is HW intensive, IOTLB is limited in size
   - Today's IO devices degrade in performance beyond a certain scale

Conclusion

- VT-x and EPT enable direct execution of guest instructions
- Dune implements processes with VT-x and EPT rather than ordinary ring protection

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

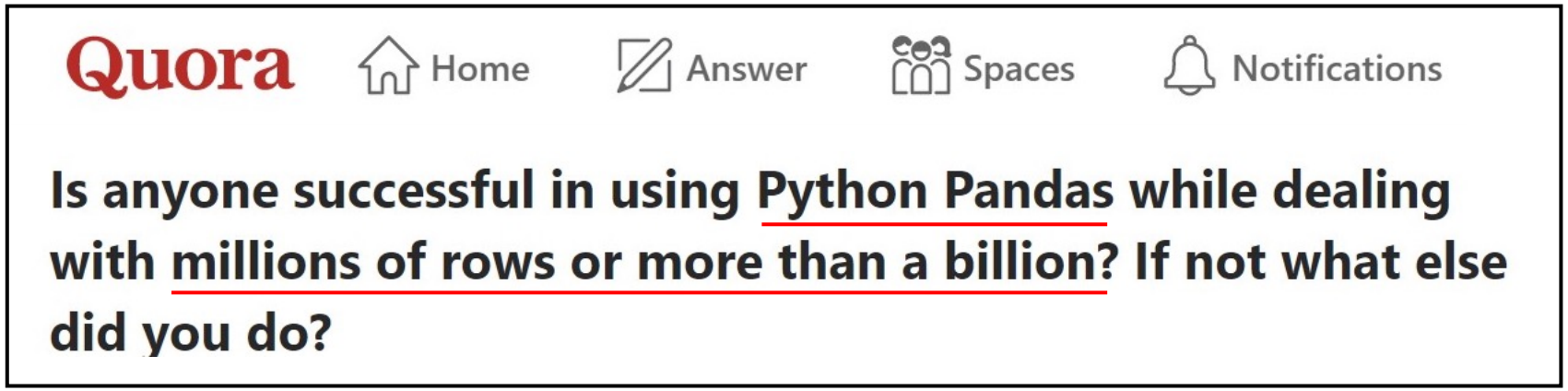
Memory Is Inelastic

- Limited by the server physical boundary.

Memory Is Inelastic

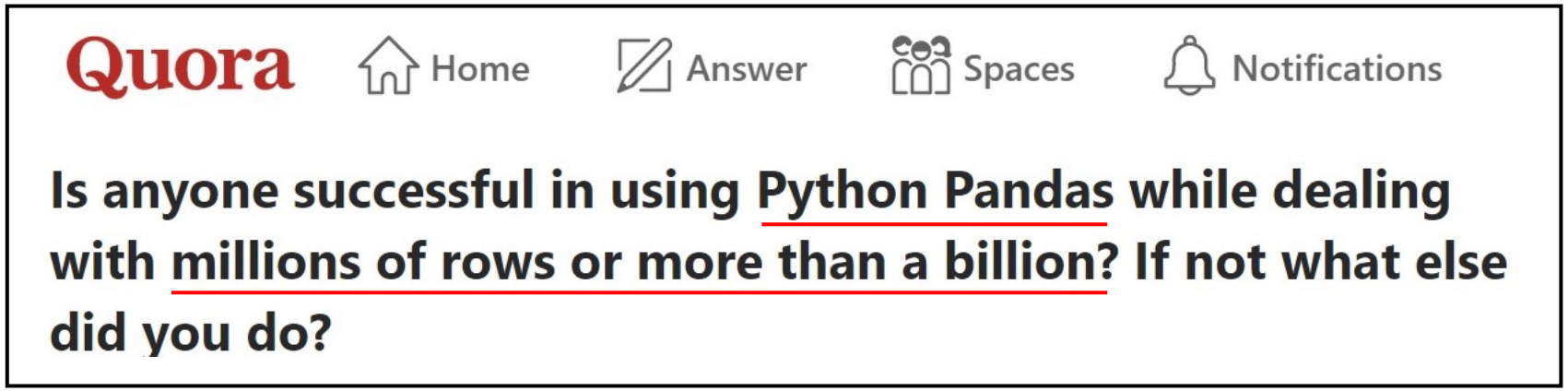
- Limited by the server physical boundary.

➢Applications cannot overcommit memory.



Memory Is Inelastic

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



➢Expensive solution: overprovision memory for peak usage.

Recent Far-Memory Systems

- Leverage the idle memory of remote servers.

Recent Far-Memory Systems

- Leverage the idle memory of remote servers.

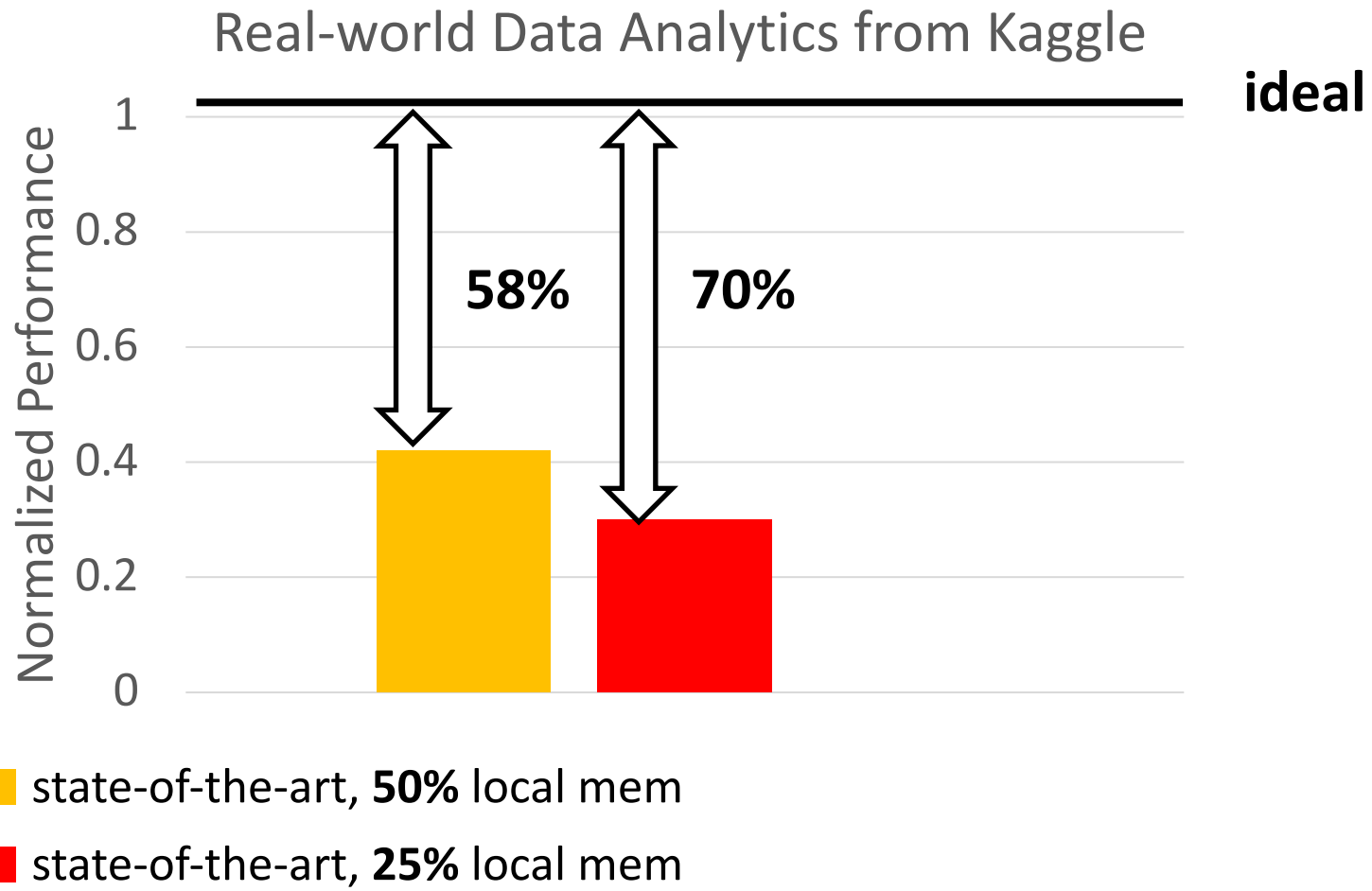
➢Enabler: narrowed Net/DRAM performance gap.

Recent Far-Memory Systems

- Leverage the idle memory of remote servers.
- Enabler: narrowed Net/DRAM performance gap.

➢Built on top of **OS paging (swap)**.

Does OS-Paging Systems Perform Well?



Why OS-Paging Systems Suffer?

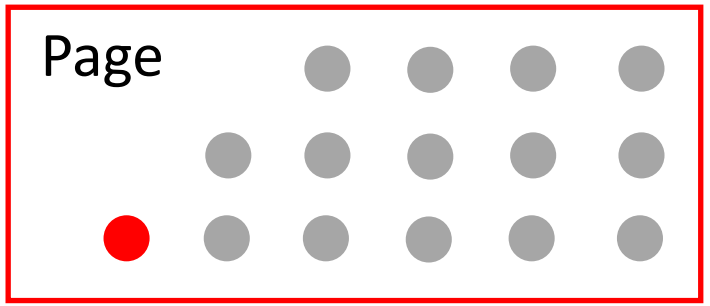
- Goal: transparent, no app code modification.

Why OS-Paging Systems Suffer?

- Goal: transparent, no app code modification.
- Require to use OS to manage virtual memory pages.
  - Semantic gap.
  - High kernel overheads.

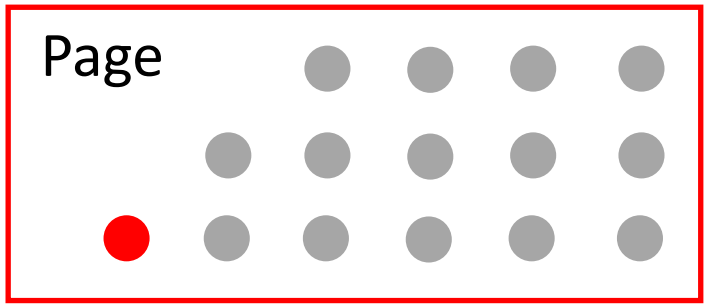
Semantic Gap (in Paging Systems)

- Page granularity. Example: R/W amplification.



Semantic Gap (in Paging Systems)

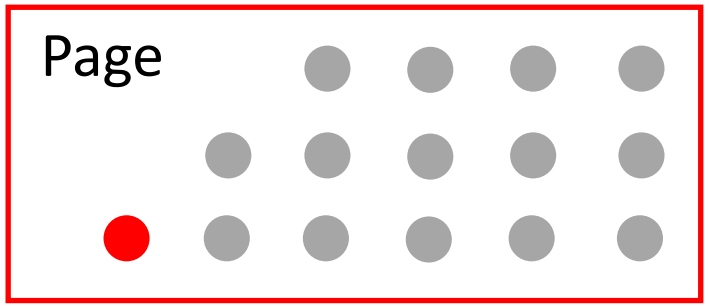
- Page granularity. Example: R/W amplification.



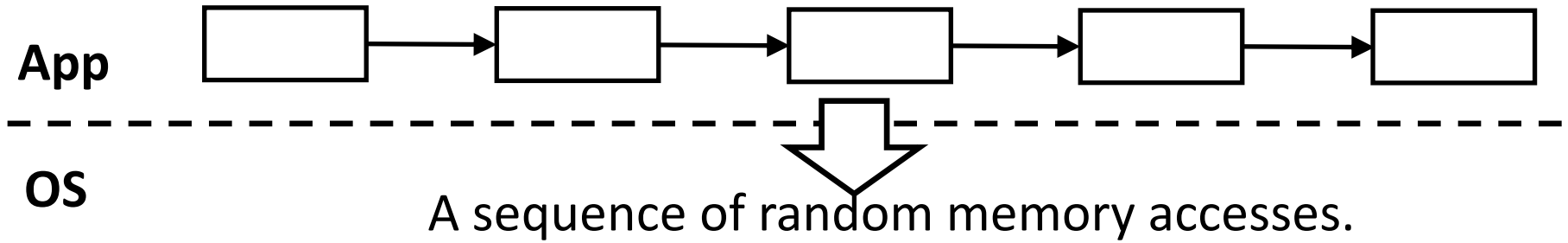
➢OS lacks app knowledge. Example: hard to prefetch.

Semantic Gap (in Paging Systems)

- Page granularity. Example: R/W amplification.



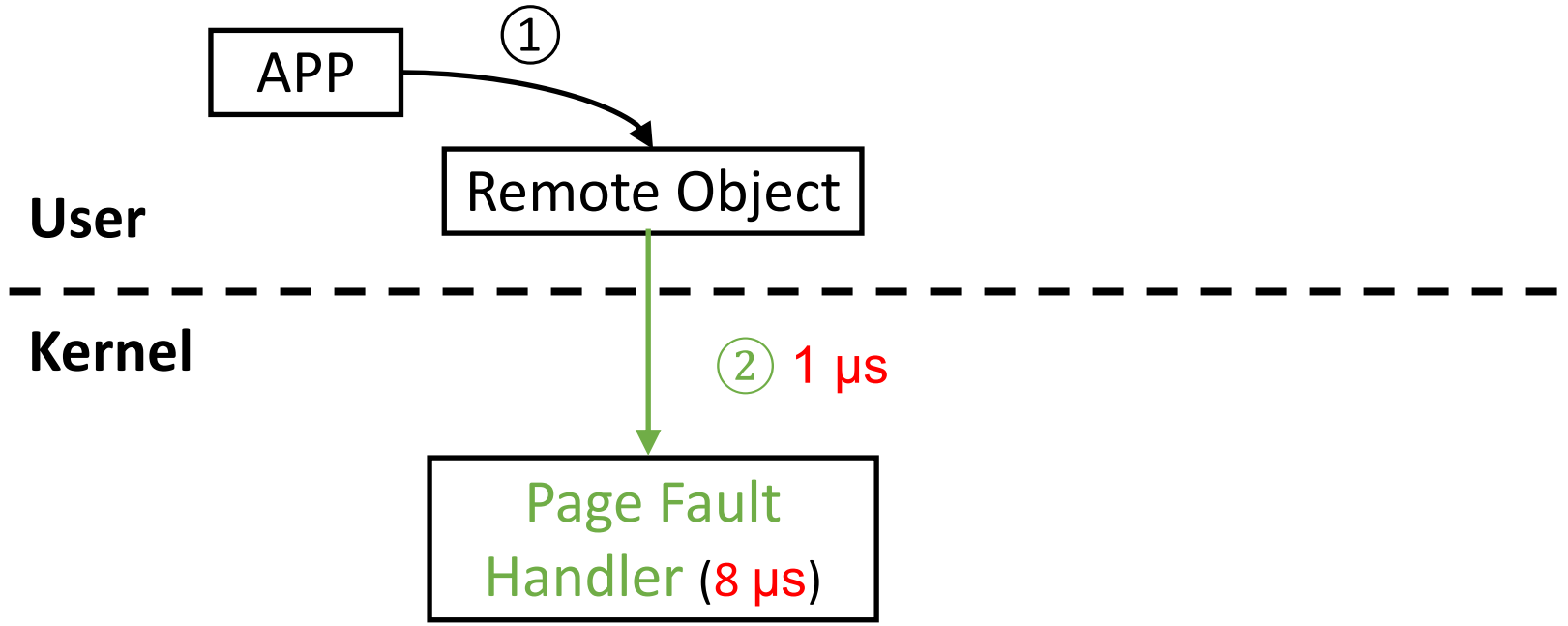
➢OS lacks app knowledge. Example: hard to prefetch.



High Kernel Overheads (in Paging Systems)

High Kernel Overheads (in Paging Systems)

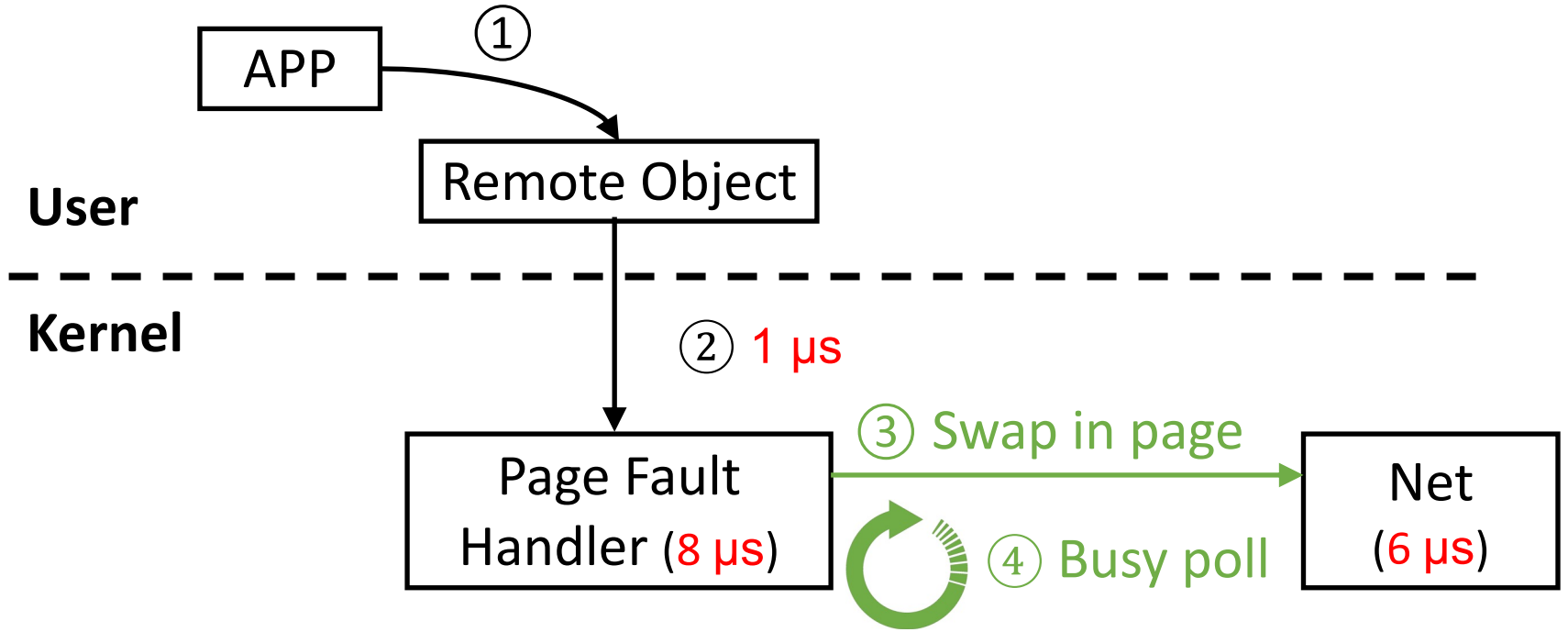
➢Use expensive page faults.



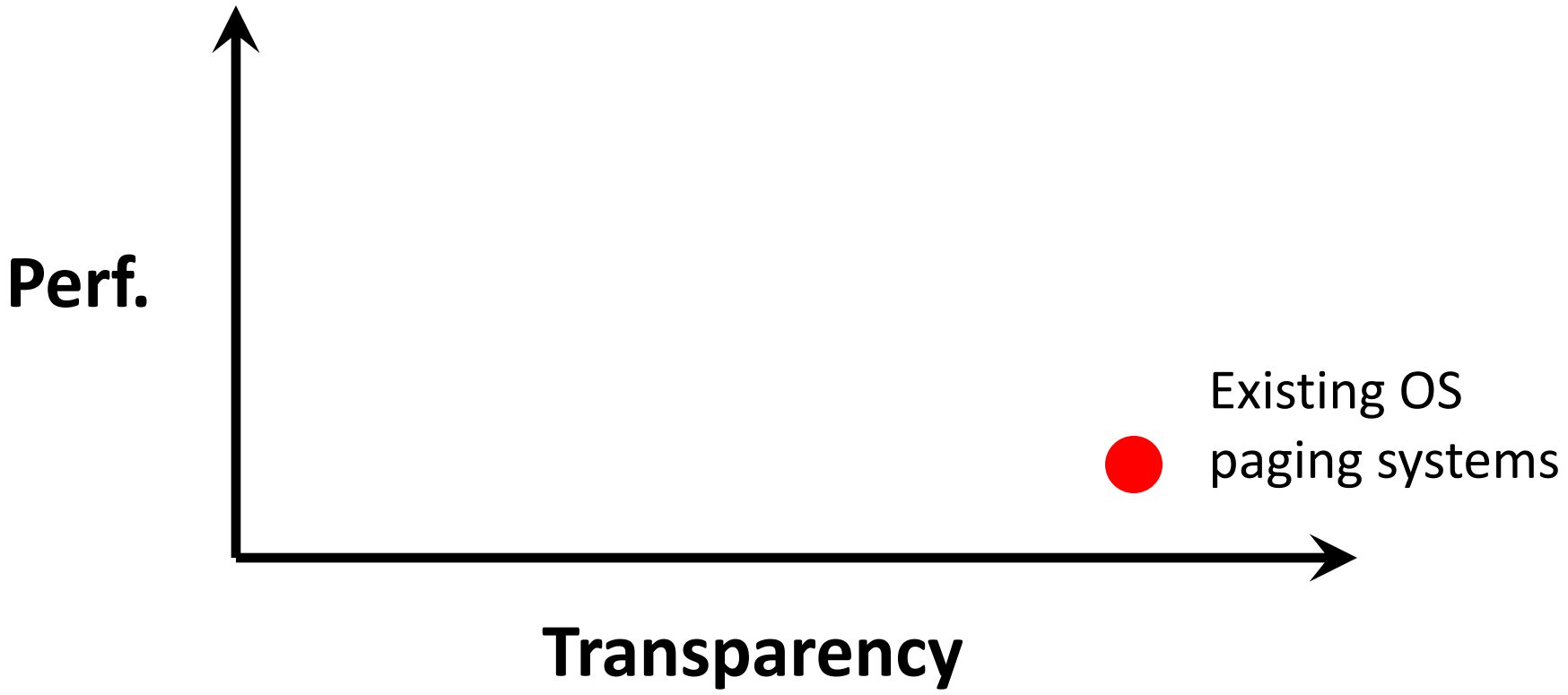
High Kernel Overheads (in Paging Systems)

- Use expensive page faults.

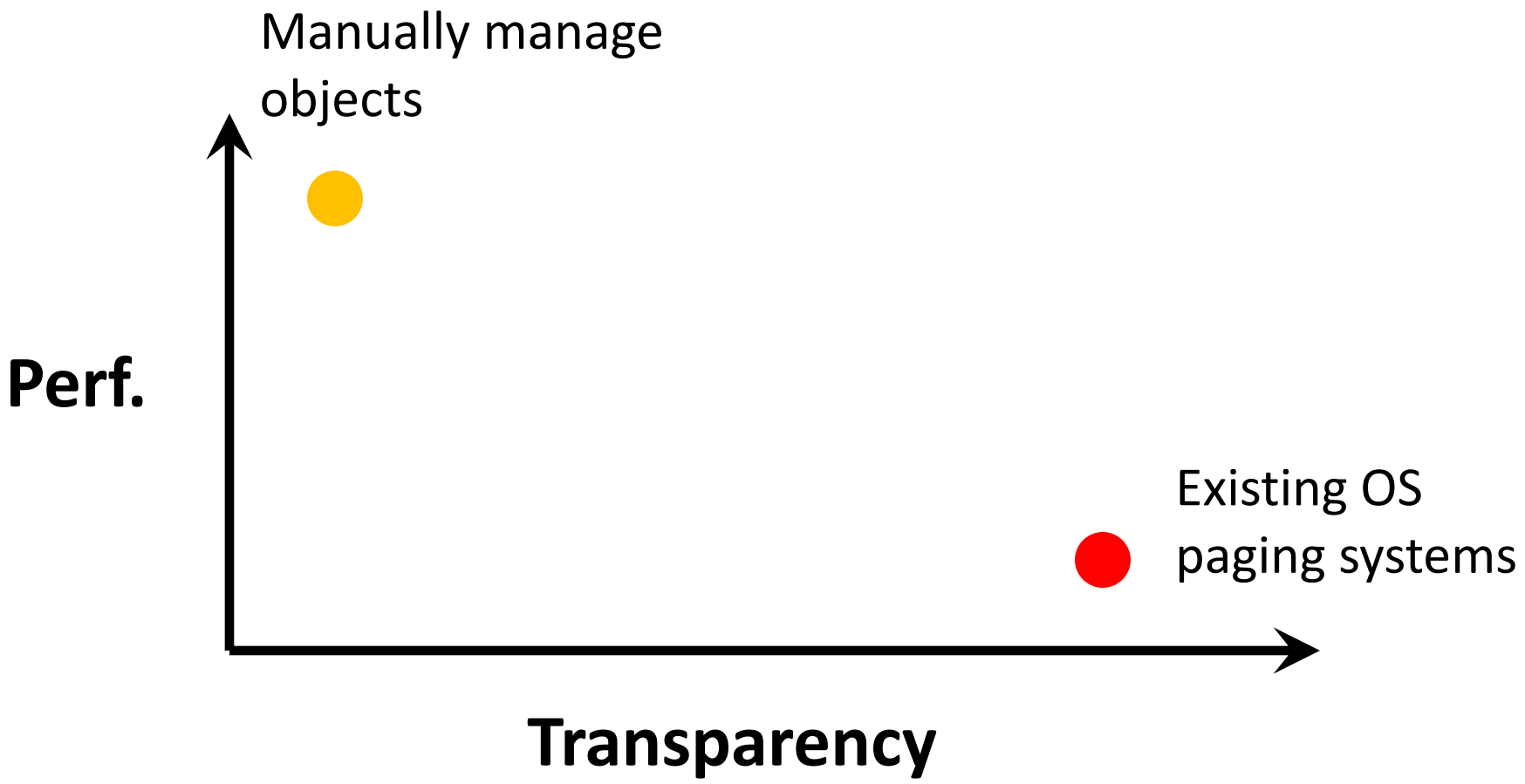
➢Use polling for in-kernel net I/O → burn CPU cycles.



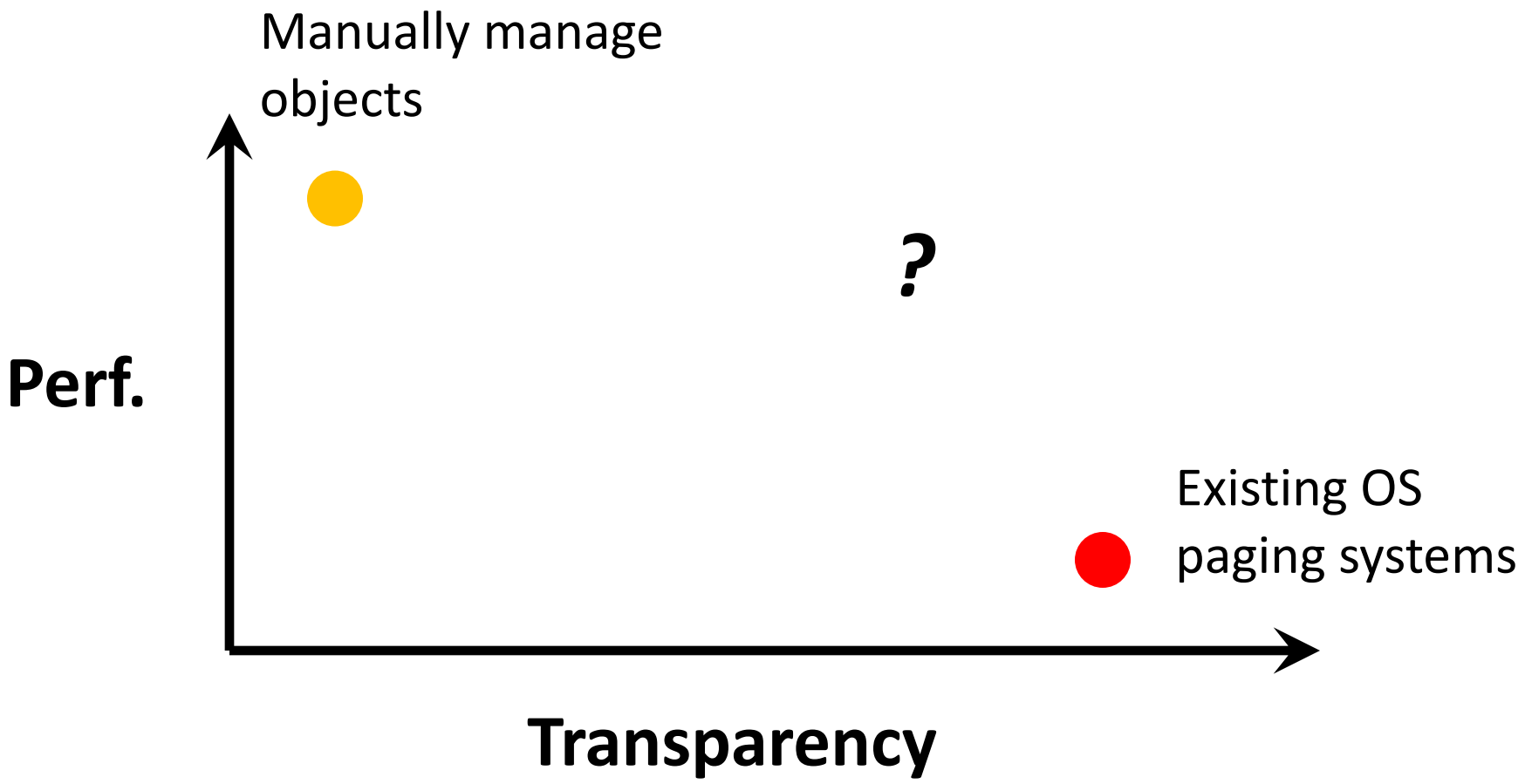
Design Space



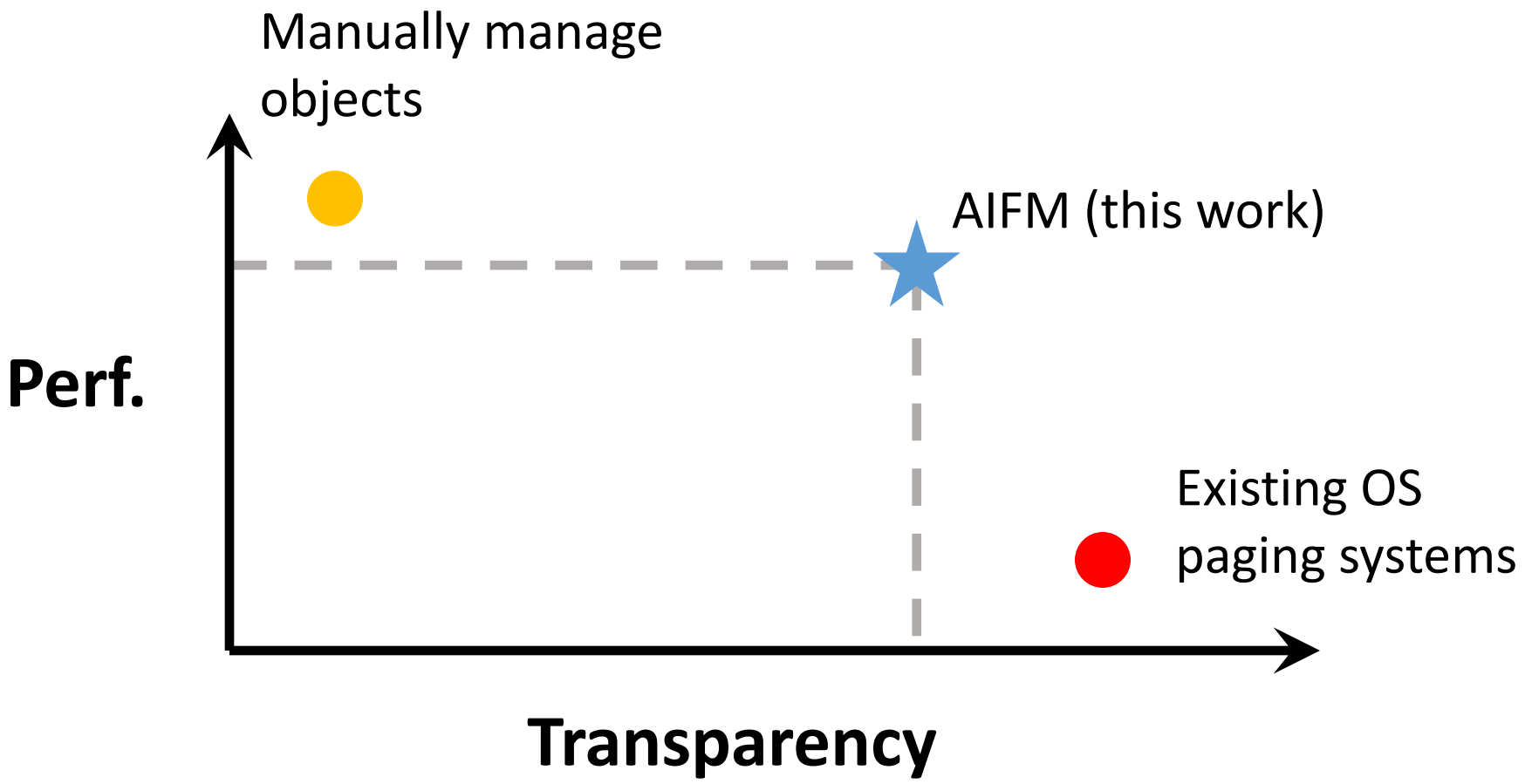
Design Space



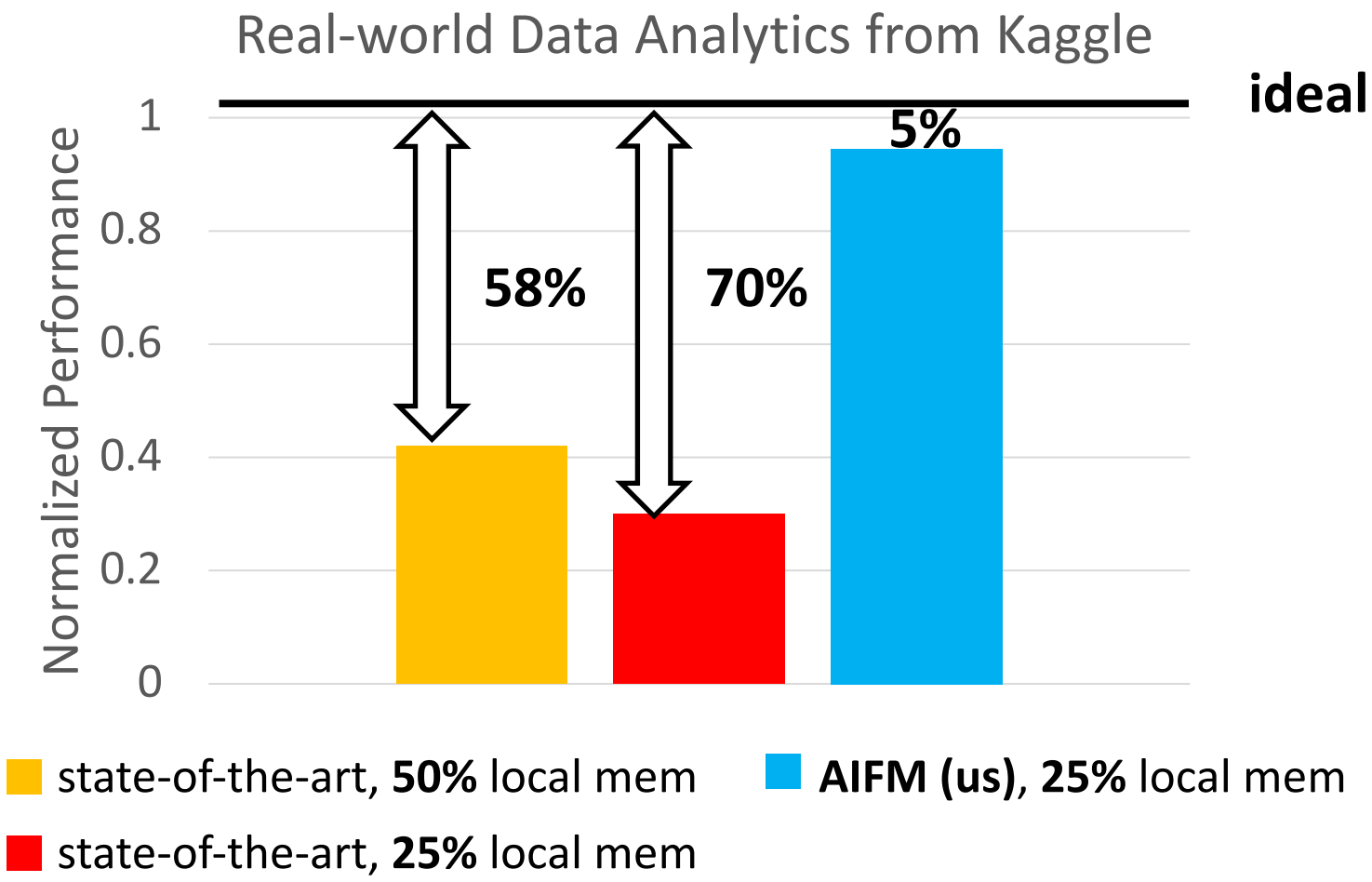
Design Space



Design Space



How Does AIFM Perform?



AIFM's Approach

- AIFM: **Application-Integrated** Far Memory.

AIFM's Approach

➢AIFM: **Application-Integrated** Far Memory.

Existing OS Paging Systems

- Semantic gap
  - Page granularity
  - No data structure knowledge
- High kernel overheads
  - Page faults on accessing remote objs
  - Busy polling for net I/O

AIFM

AIFM's Approach

➢AIFM: **Application-Integrated** Far Memory.

Existing OS Paging Systems

- Semantic gap
  - Page granularity
  - No data structure knowledge
- High kernel overheads
  - Page faults on accessing remote objs
  - Busy polling for net I/O

AIFM

- Use data structure lib API to bridge gap

AIFM's Approach

➢AIFM: **Application-Integrated** Far Memory.

Existing OS Paging Systems

- Semantic gap
  - Page granularity
  - No data structure knowledge
- High kernel overheads
  - Page faults on accessing remote objs
  - Busy polling for net I/O

AIFM

- Use data structure lib API to bridge gap
  - Object granularity

AIFM's Approach

➢AIFM: **Application-Integrated** Far Memory.

Existing OS Paging Systems

- Semantic gap
  - Page granularity
  - No data structure knowledge
- High kernel overheads
  - Page faults on accessing remote objs
  - Busy polling for net I/O

AIFM

- Use data structure lib API to bridge gap
  - Object granularity
  - Full data structure knowledge

AIFM's Approach

➢AIFM: **Application-Integrated** Far Memory.

Existing OS Paging Systems	**AIFM**

• Semantic gap
  • Page granularity
  • No data structure knowledge
	• Use data structure lib API to bridge gap
	  • Object granularity
	  • Full data structure knowledge

• High kernel overheads
  • Page faults on accessing remote objs
  • Busy polling for net I/O
	• Userspace runtime that swaps in/out objs

AIFM's Approach

➢AIFM: **Application-Integrated** Far Memory.

Existing OS Paging Systems

- Semantic gap
  - Page granularity
  - No data structure knowledge
- High kernel overheads
  - Page faults on accessing remote objs
  - Busy polling for net I/O

AIFM

- Use data structure lib API to bridge gap
  - Object granularity
  - Full data structure knowledge
- Userspace runtime that swaps in/out objs
  - Function calls on accessing remote objs

AIFM's Approach

➢AIFM: **Application-Integrated** Far Memory.

Existing OS Paging Systems

- Semantic gap
  - Page granularity
  - No data structure knowledge
- High kernel overheads
  - Page faults on accessing remote objs
  - Busy polling for net I/O

AIFM

- Use data structure lib API to bridge gap
  - Object granularity
  - Full data structure knowledge
- Userspace runtime that swaps in/out objs
  - Function calls on accessing remote objs
  - Context switch for net I/O

AIFM in Action

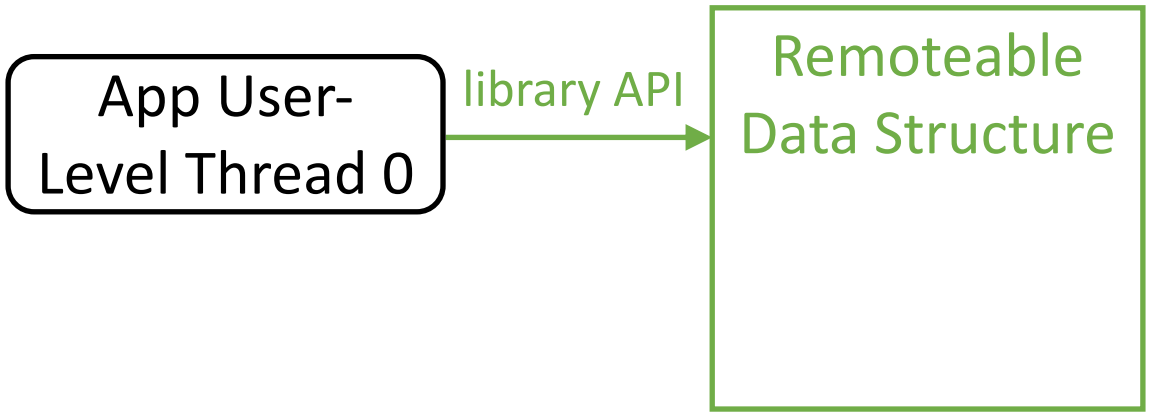
App User-Level Thread 0

Local Memory

Far Memory

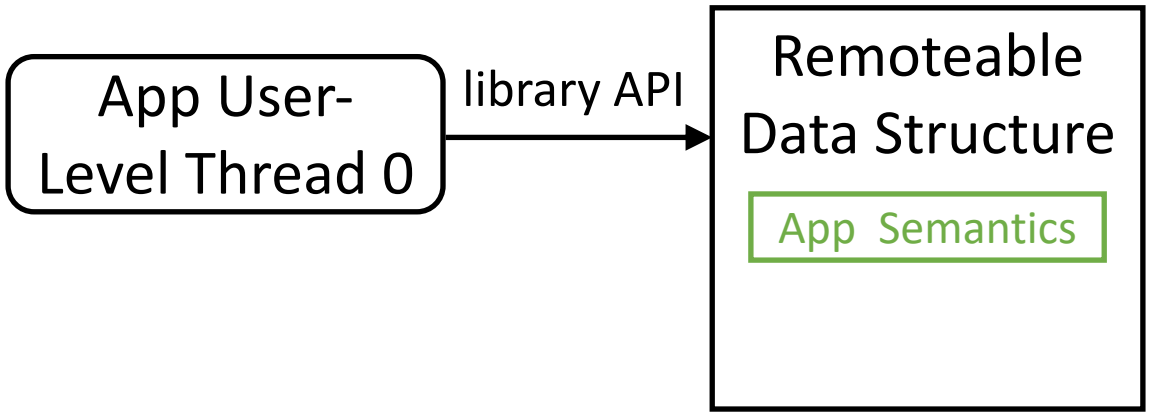
1. Remoteable Data Structure Library

- Solved challenge: semantic gap.



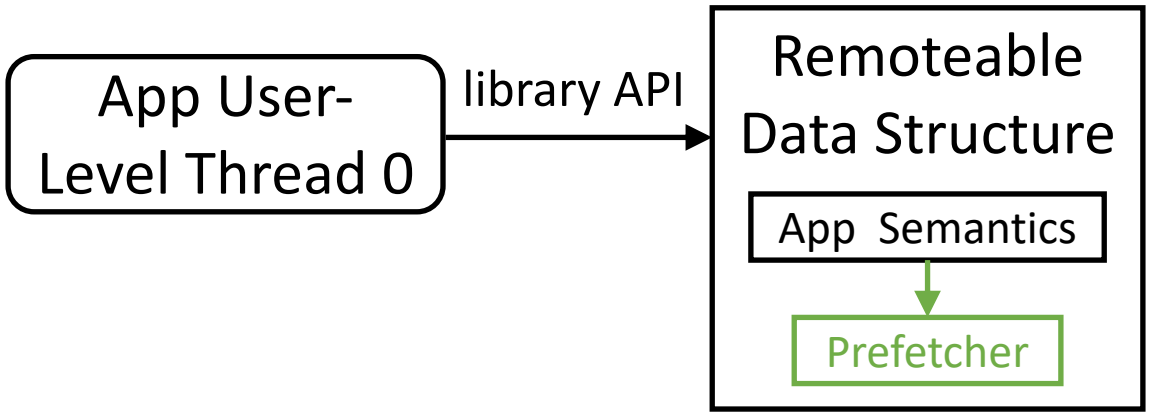
1. Remoteable Data Structure Library

➢Solved challenge: semantic gap.



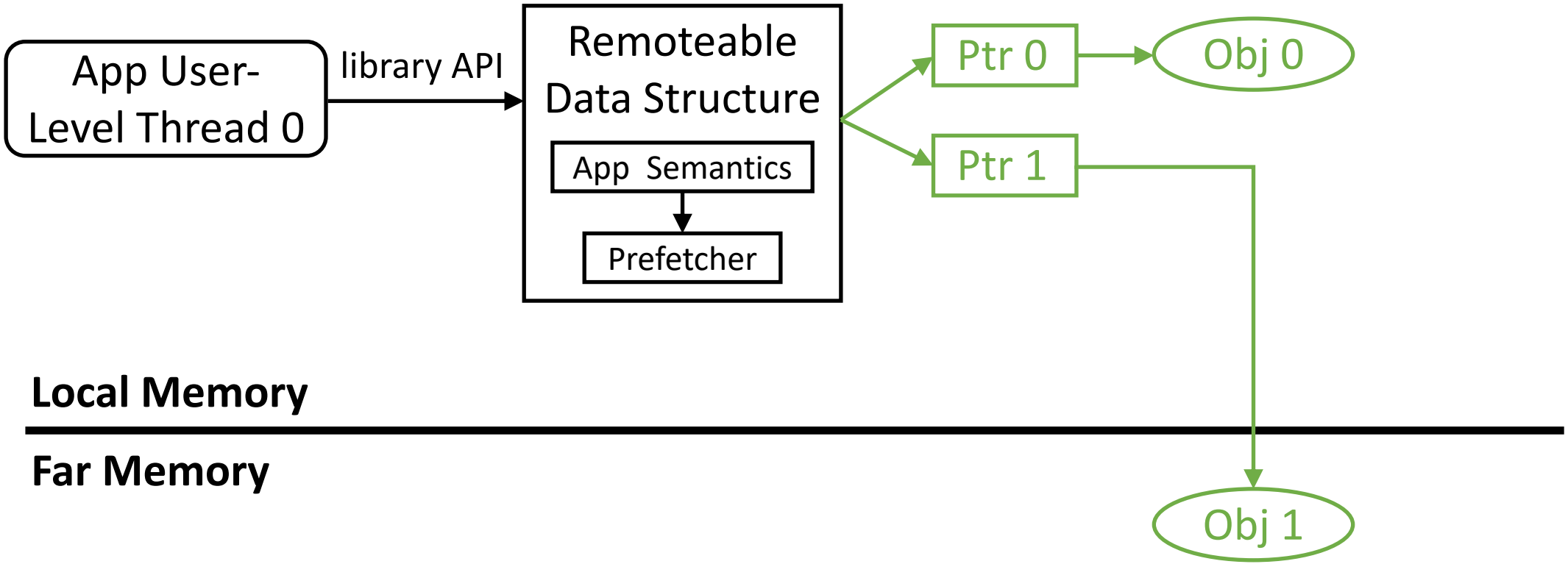
1. Remoteable Data Structure Library

➢Solved challenge: semantic gap.



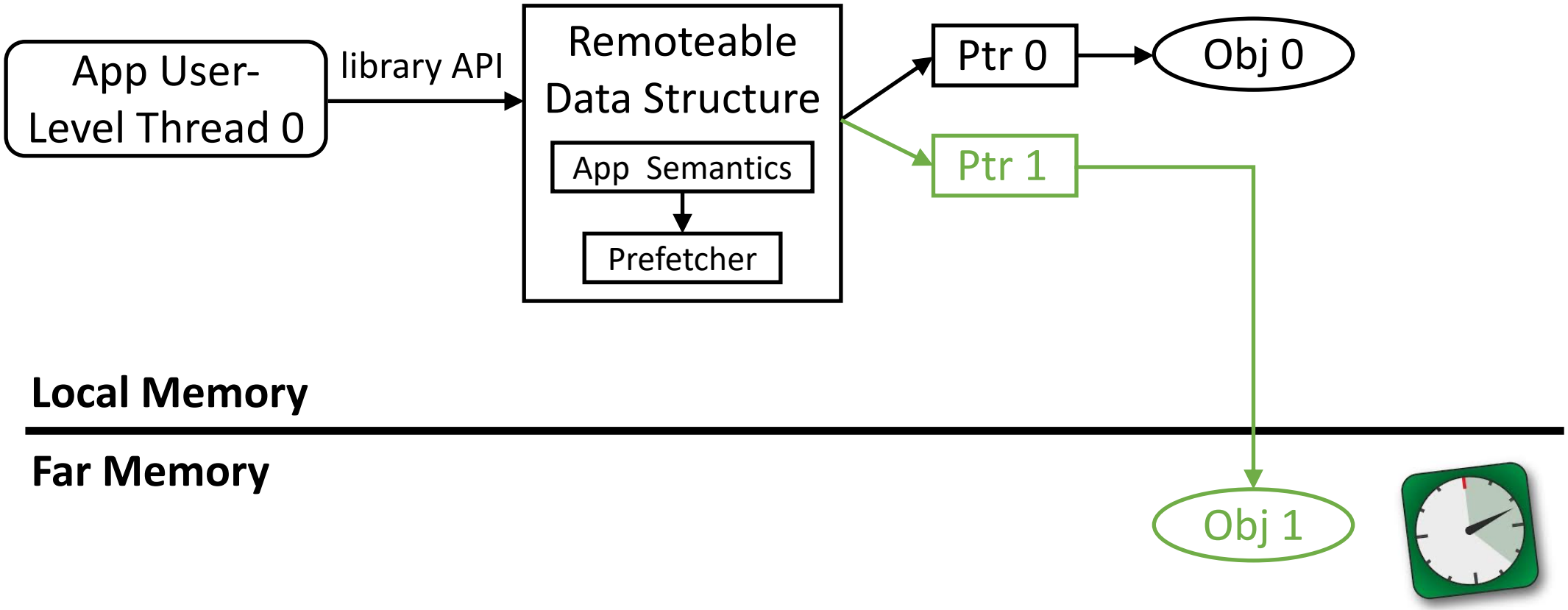
1. Remoteable Data Structure Library

- Solved challenge: semantic gap.



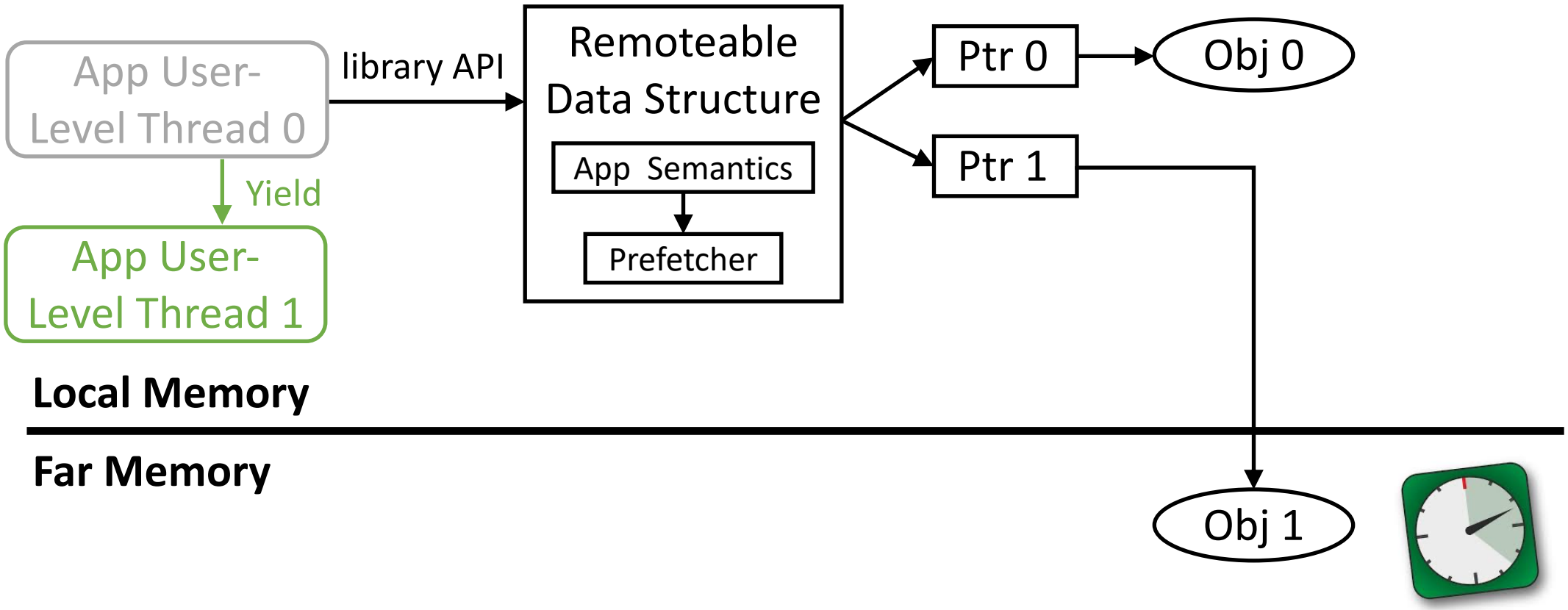
2. Userspace Runtime

➢Solved challenge: kernel overheads.



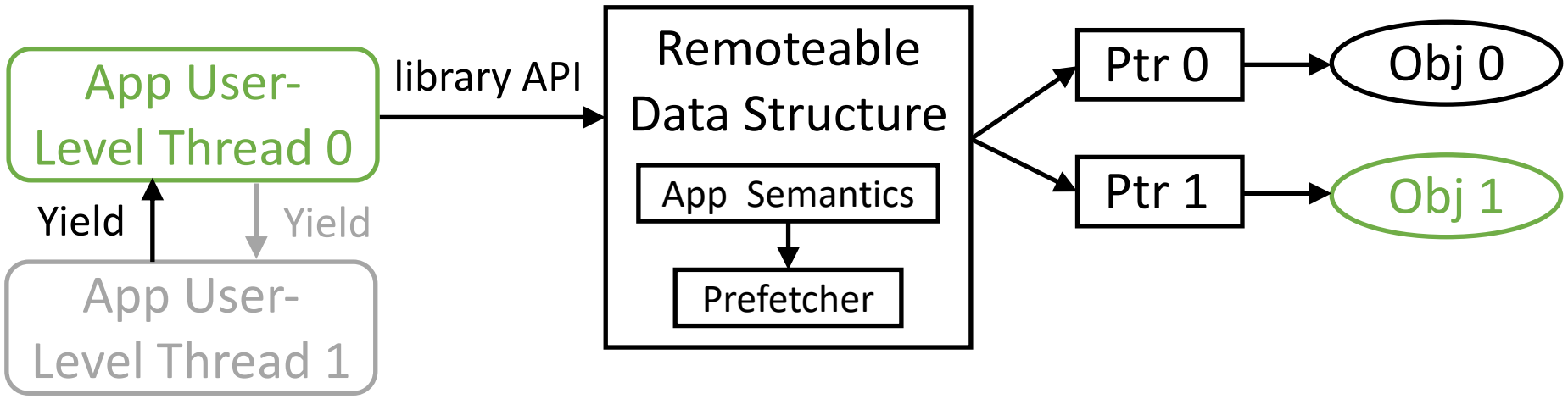
2. Userspace Runtime

➢Solved challenge: kernel overheads.



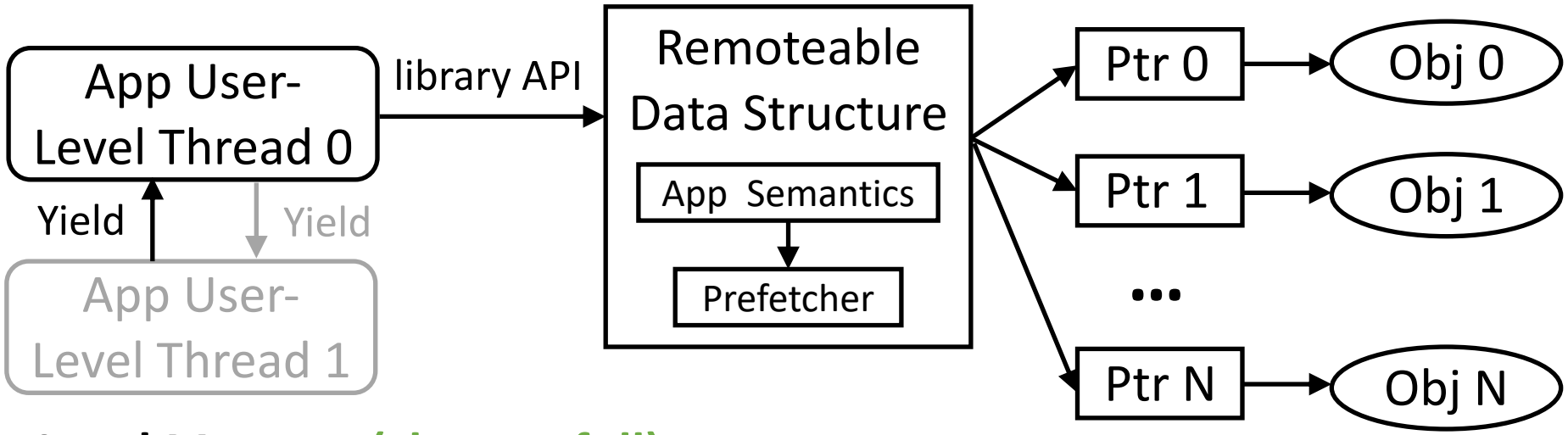
2. Userspace Runtime

➢Solved challenge: kernel overheads.



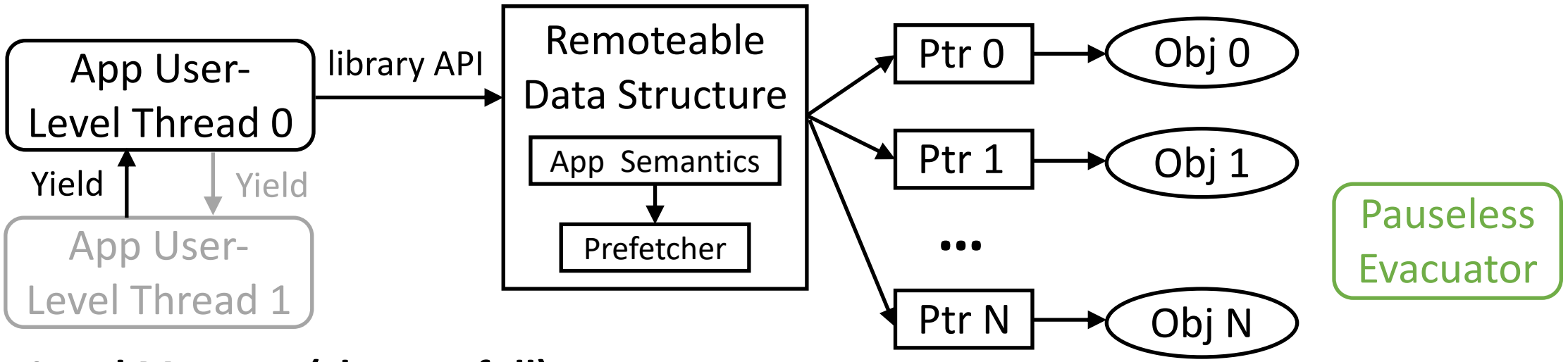
3. Pauseless Evacuator

➢Solved challenge: impact of memory reclamation.



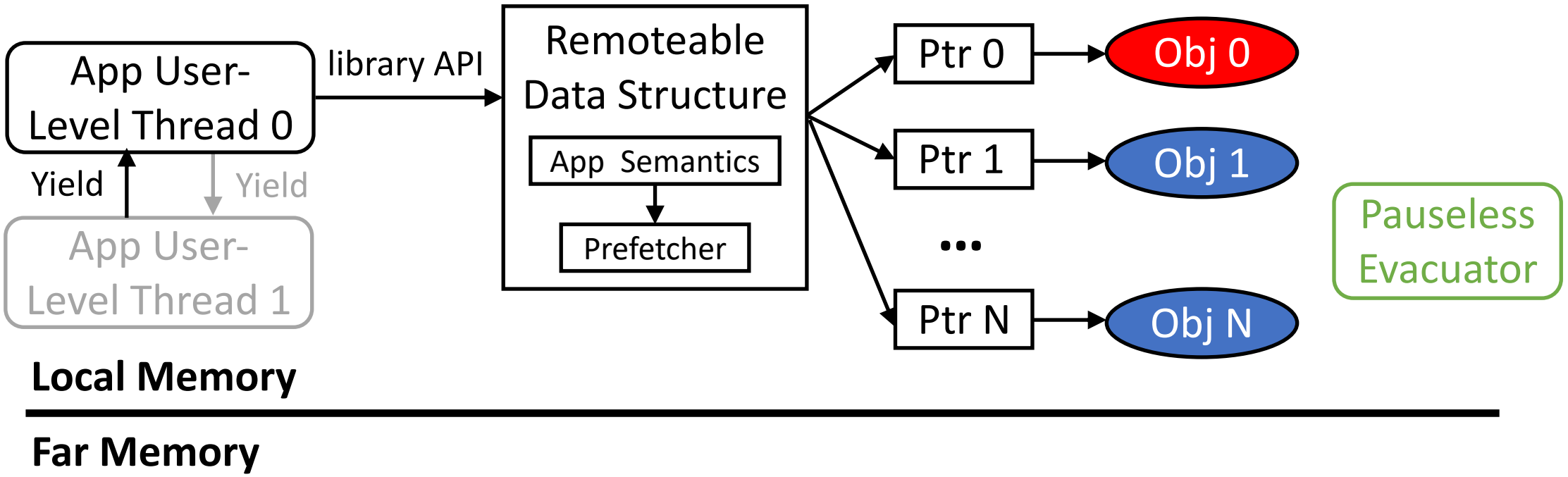
3. Pauseless Evacuator

➢Solved challenge: impact of memory reclamation.



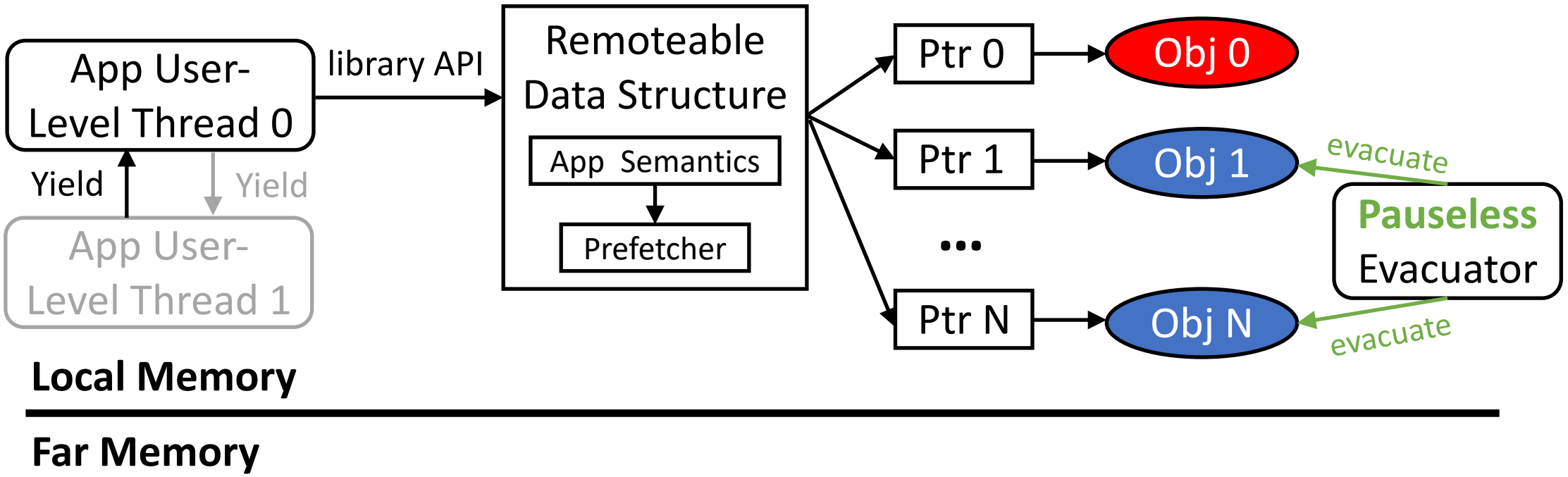
3. Pauseless Evacuator

➢Solved challenge: impact of memory reclamation.



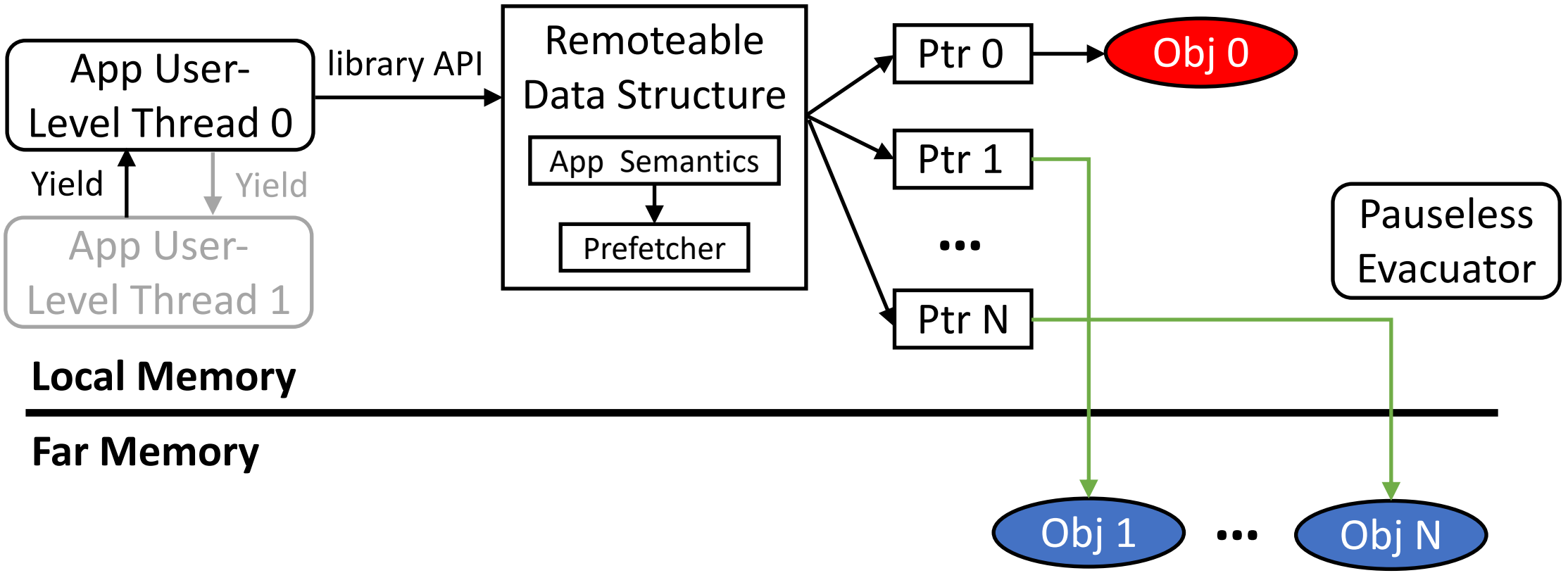
3. Pauseless Evacuator

➢Solved challenge: impact of memory reclamation.



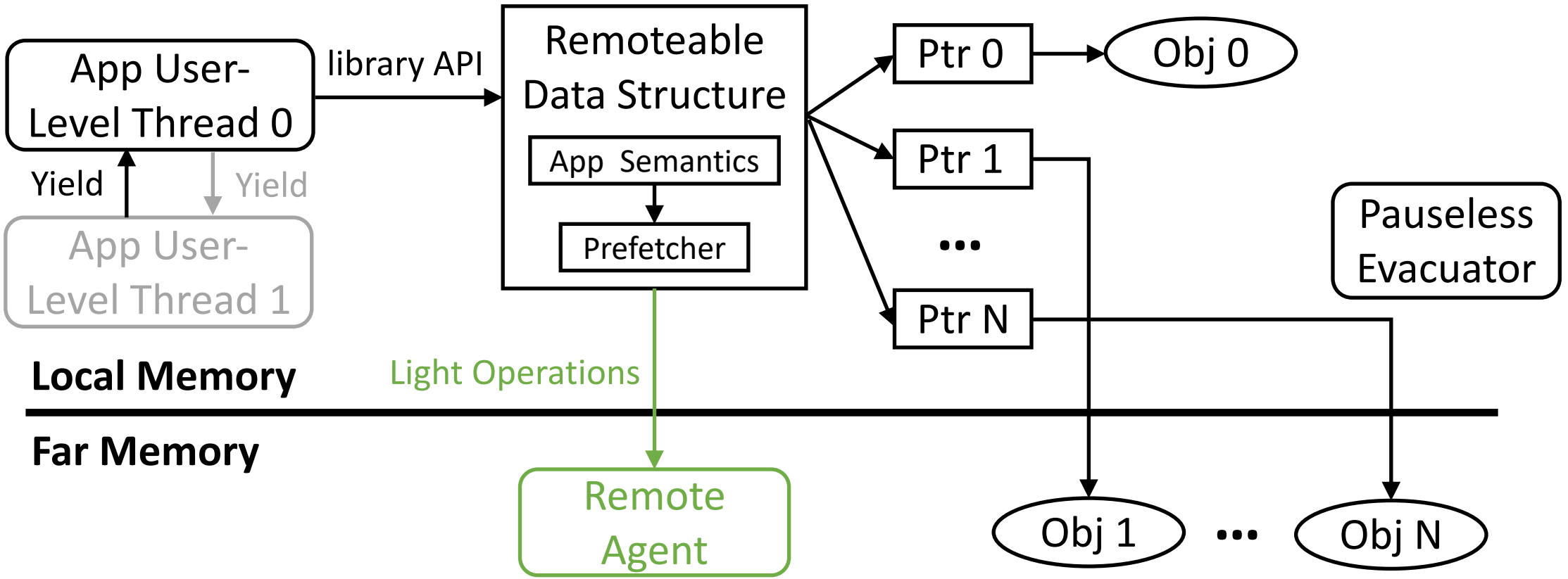
3. Pauseless Evacuator

➢Solved challenge: impact of memory reclamation.



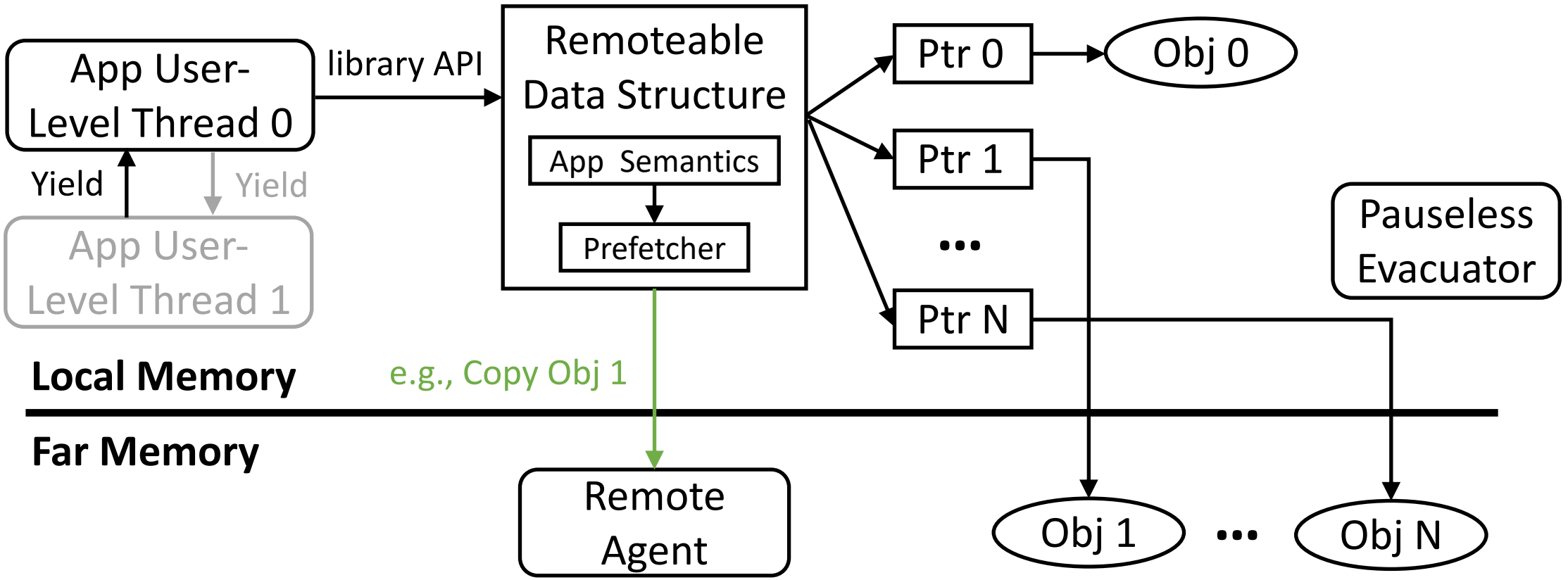
4. Remote Agent

➢Solved challenge: network BW < DRAM BW.



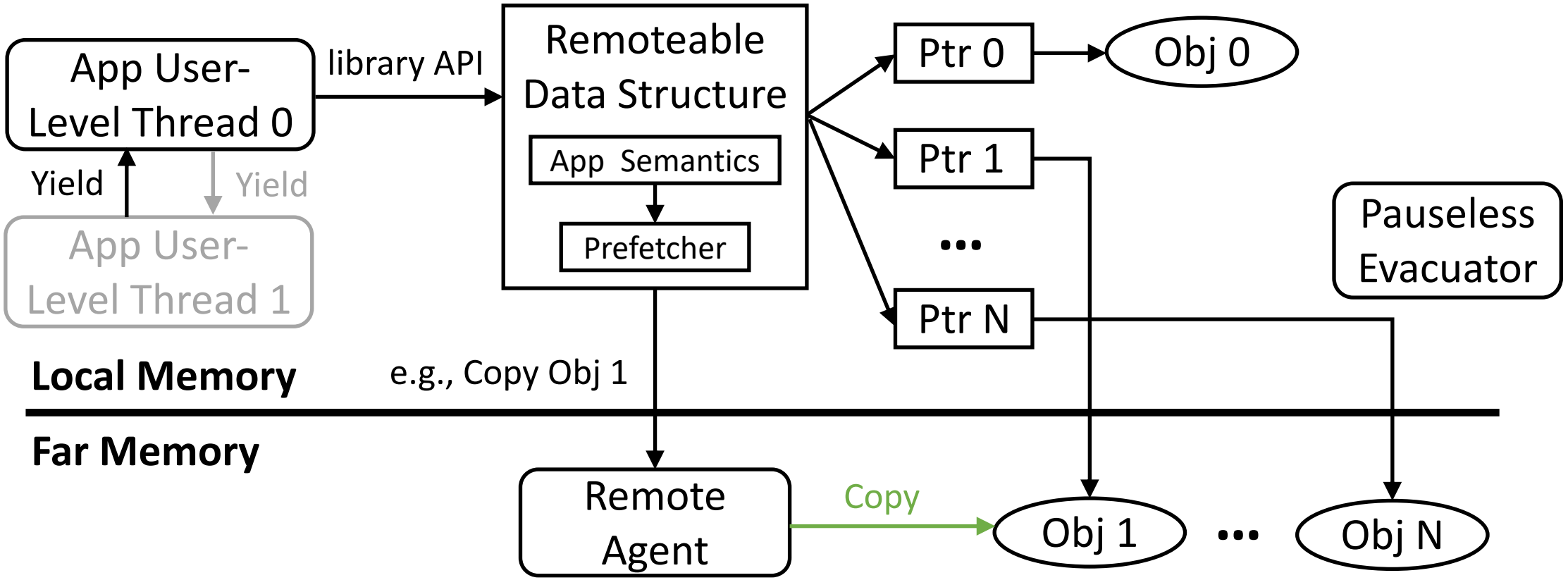
4. Remote Agent

➢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) {

        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) {
        DerefScope scope;
        sum += hashtable.at(key, scope);
    }
    DerefScope scope;
    LargeData ret = arr.at(sum, scope);
    return ret;
}
```

Ensure the objects being accessed will not be moved by the evacuator.

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) {
        DerefScope scope;
        sum += hashtable.at(key, scope);
    }
    DerefScope scope;
    LargeData ret = arr.at</*don't cache*/ true>(sum, scope);
    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);
   }
   DerefScope scope;
   LargeData ret = arr.at</*don't cache*/ true>(sum, scope);
   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 KV pairs.
  }
  DerefScope scope;
  LargeData ret = arr.at</*don't cache*/ true>(sum, scope);
  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 KV pairs.
  }
  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.

Implementation

- Implemented 6 data structures.
  - Array, List, Hashtable, Vector, Stack, and Queue.

➢AIFM runtime is built on top of Shenango [NSDI’ 19]

Implementation

- Implemented 6 data structures.
  - Array, List, Hashtable, Vector, Stack, and Queue.
- AIFM runtime is built on top of Shenango [NSDI' 19]

➢TCP-based far memory backend.

Implementation

- Implemented 6 data structures.
  - Array, List, Hashtable, Vector, Stack, and Queue.
- AIFM runtime is built on top of Shenango [NSDI' 19]
- TCP-based far memory backend.

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

Evaluation

- Setup: 1 compute server + 1 far memory server, 25 GbE.

Evaluation

- Setup: 1 compute server + 1 far memory server, 25 GbE.
- How does AIFM
  - ➢… perform on applications with different compute intensities?

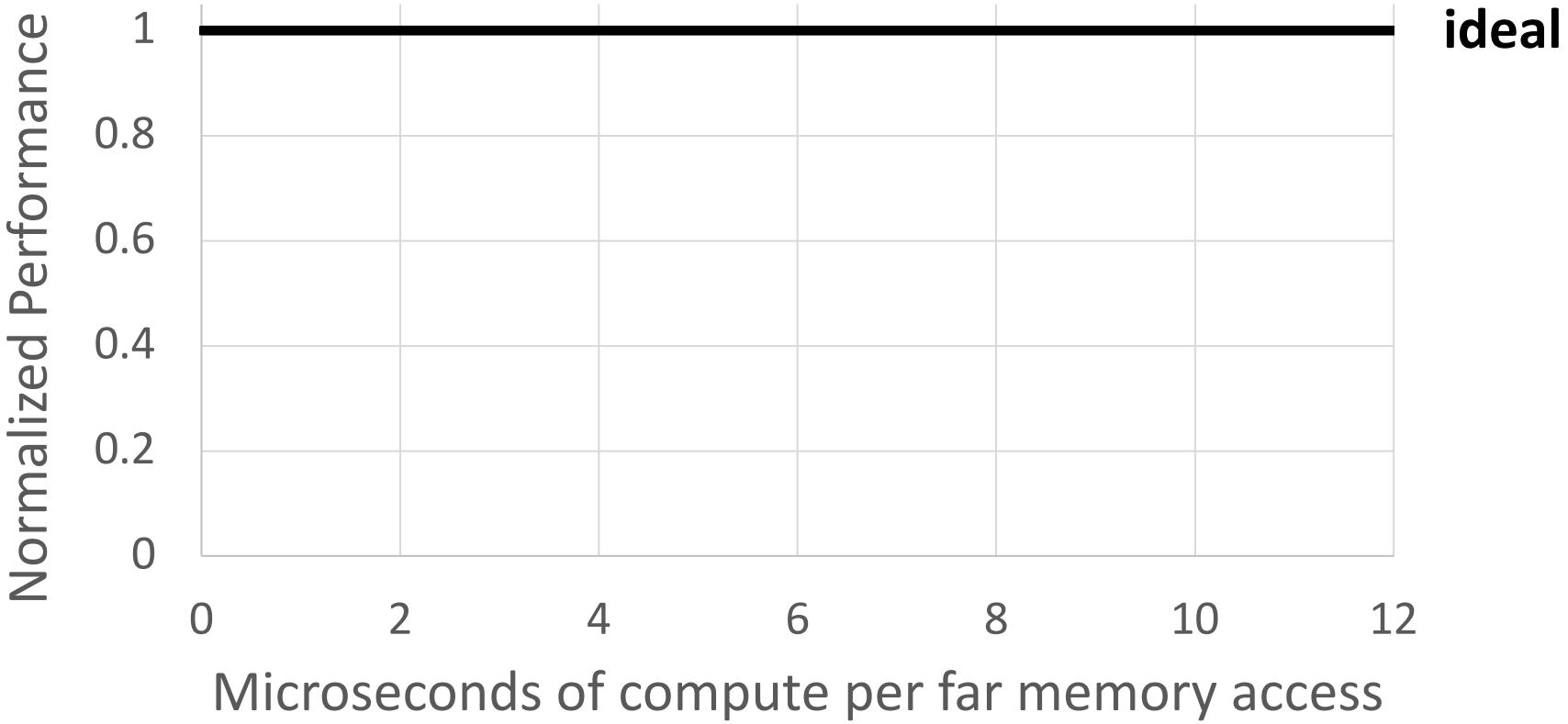
Evaluation

- Setup: 1 compute server + 1 far memory server, 25 GbE.
- How does AIFM
  - … perform on applications with different compute intensities?
  - ➢ … compare to the local-only (ideal) system?

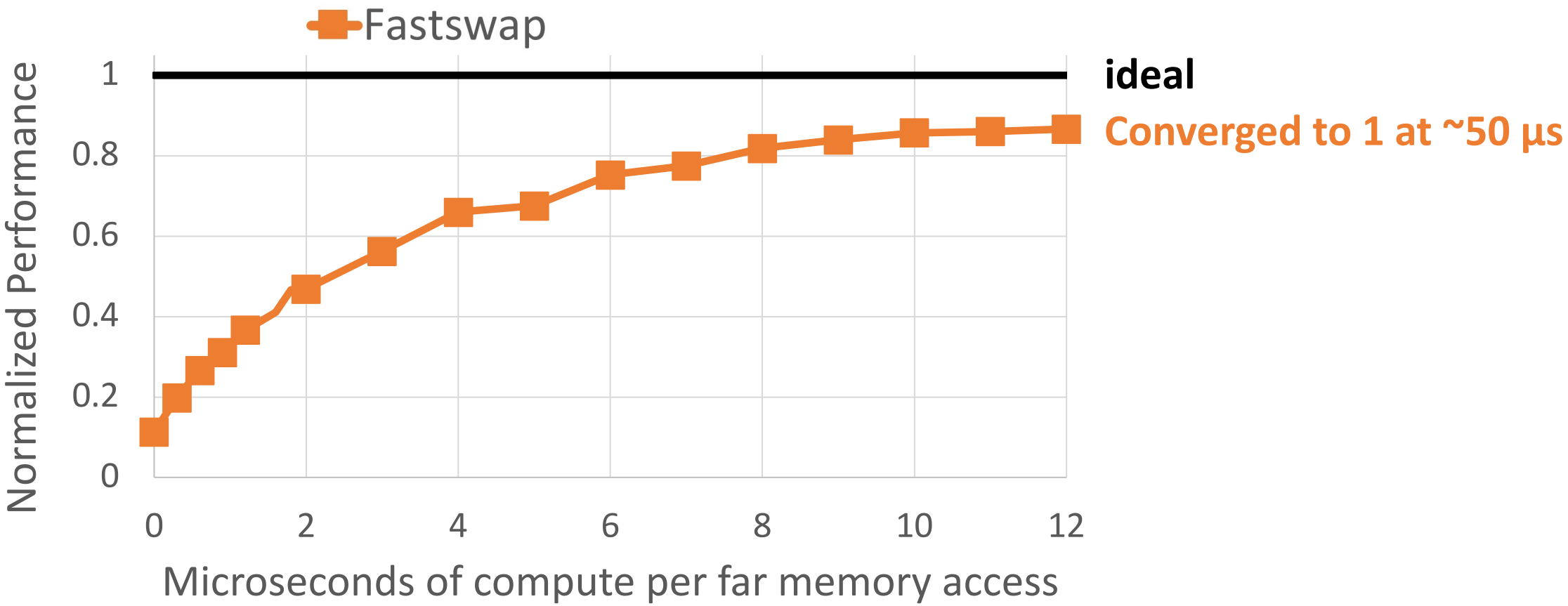
Evaluation

- Setup: 1 compute server + 1 far memory server, 25 GbE.
- How does AIFM
  - … perform on applications with different compute intensities?
  - … compare to the local-only (ideal) system?
  - ➢… compare to the state-of-the-art paging system, Fastswap [EuroSys' 20]?

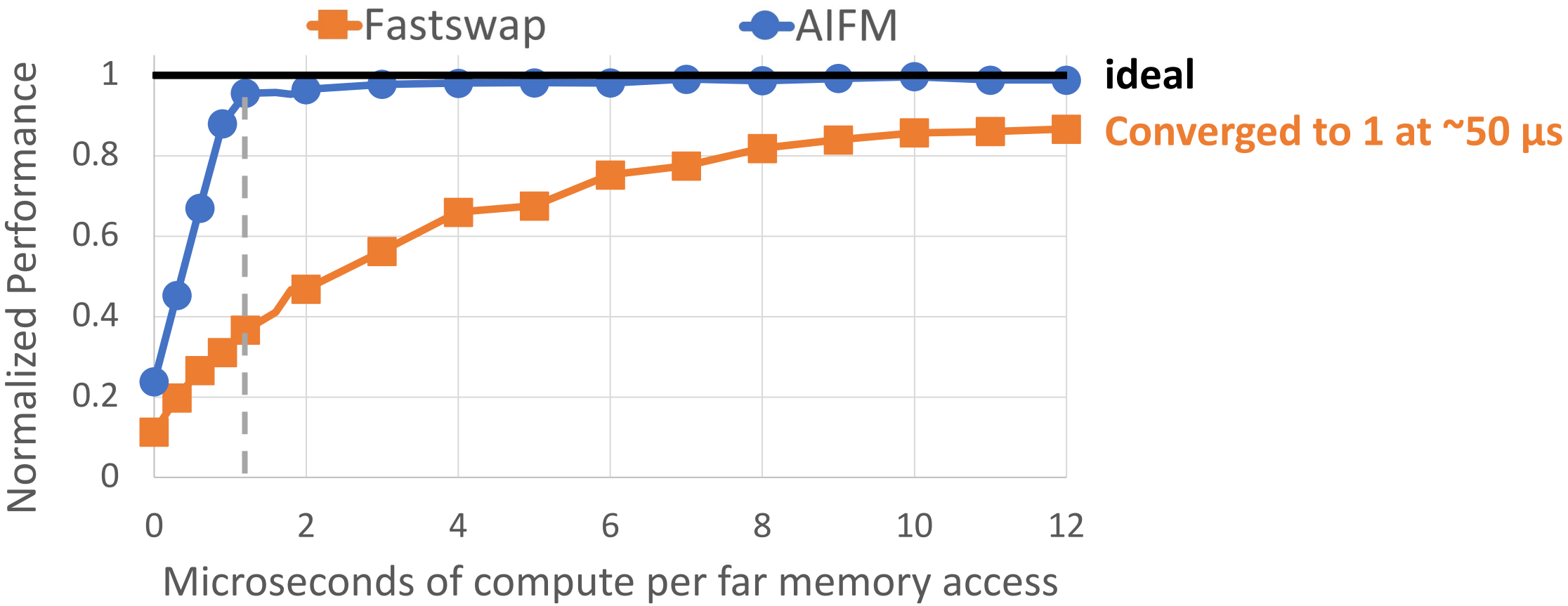
Performance on Different Compute Intensities



Performance on Different Compute Intensities

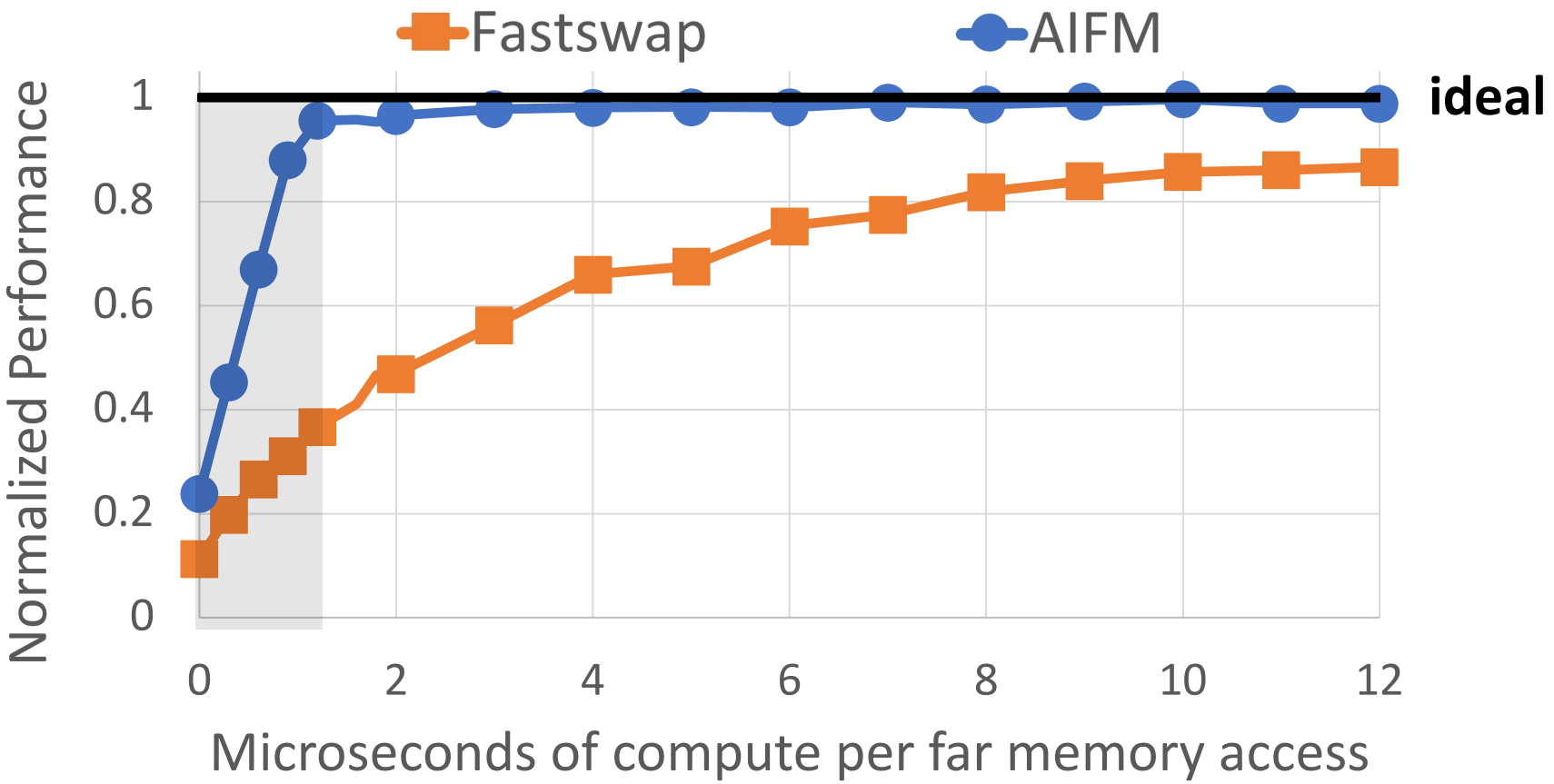


Performance on Different Compute Intensities

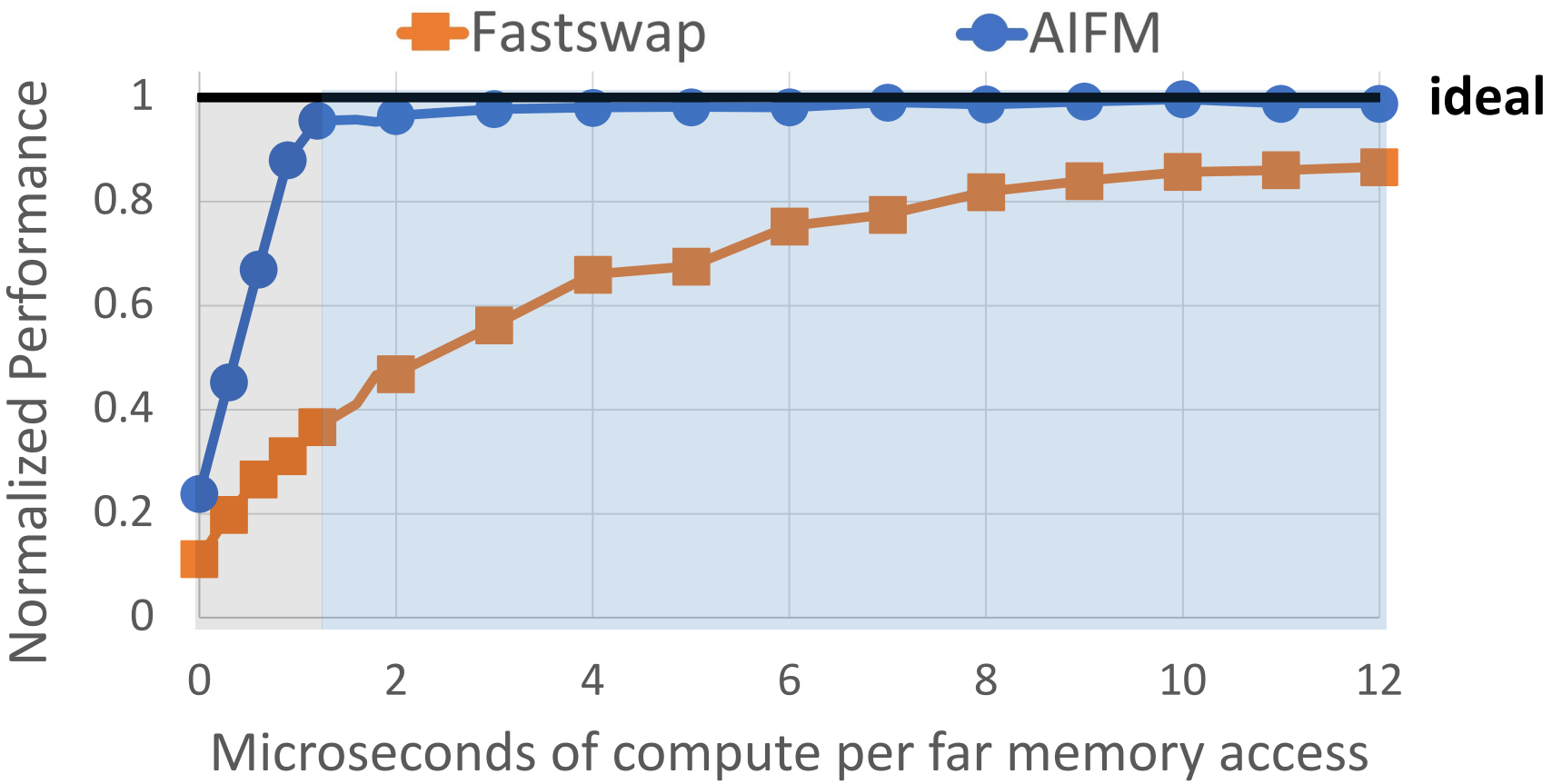


AIFM hides far memory latency with moderate compute.

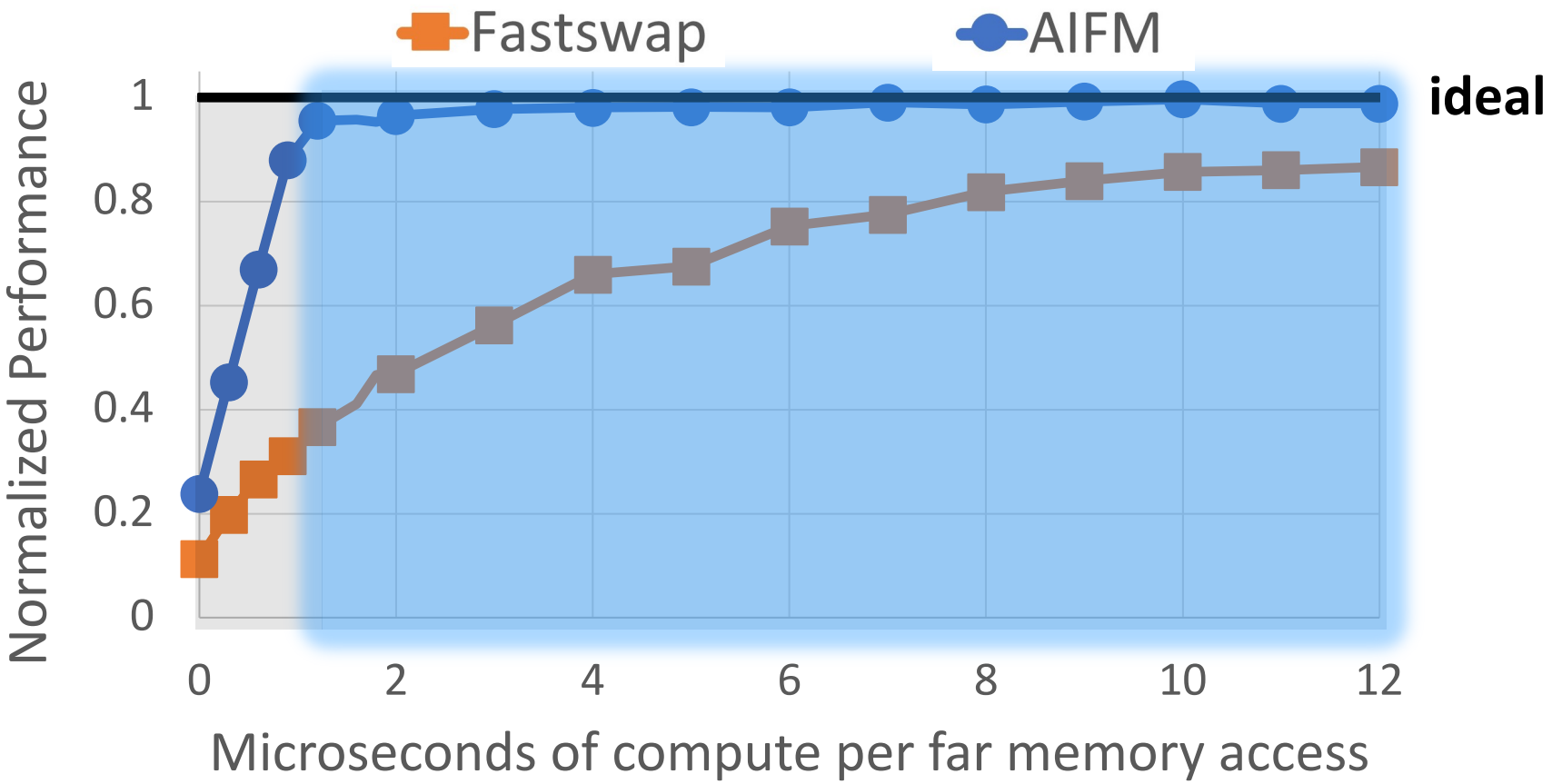
Performance on Different Compute Intensities



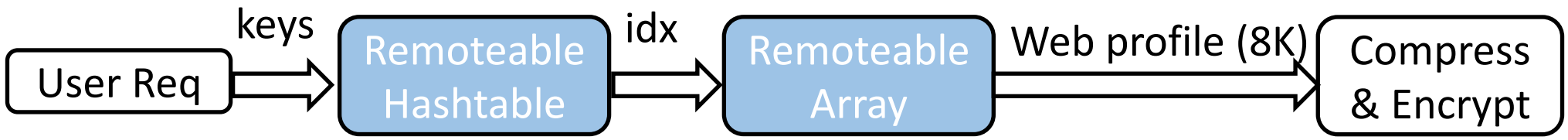
Performance on Different Compute Intensities



Performance on Different Compute Intensities

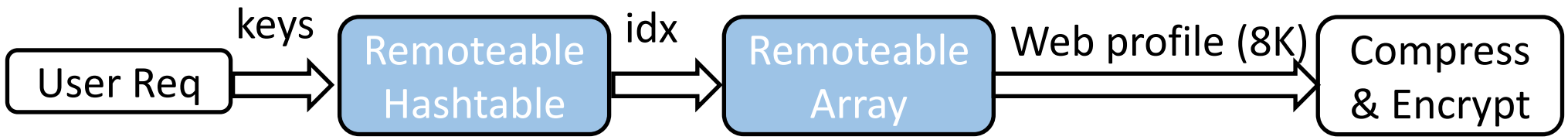


Synthetic Web Frontend



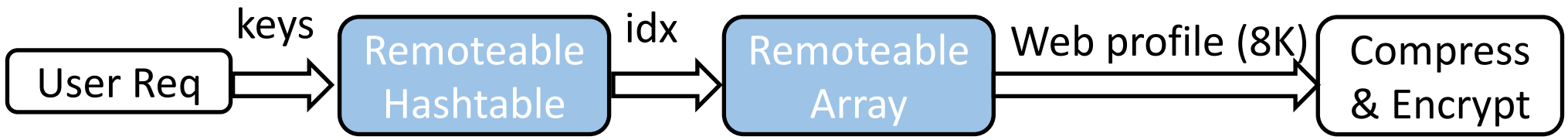
Synthetic Web Frontend

- Object-level caching.



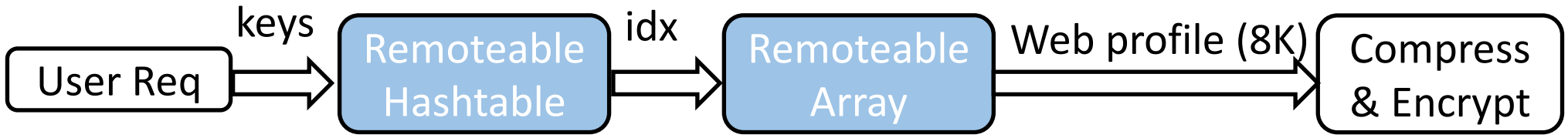
Synthetic Web Frontend

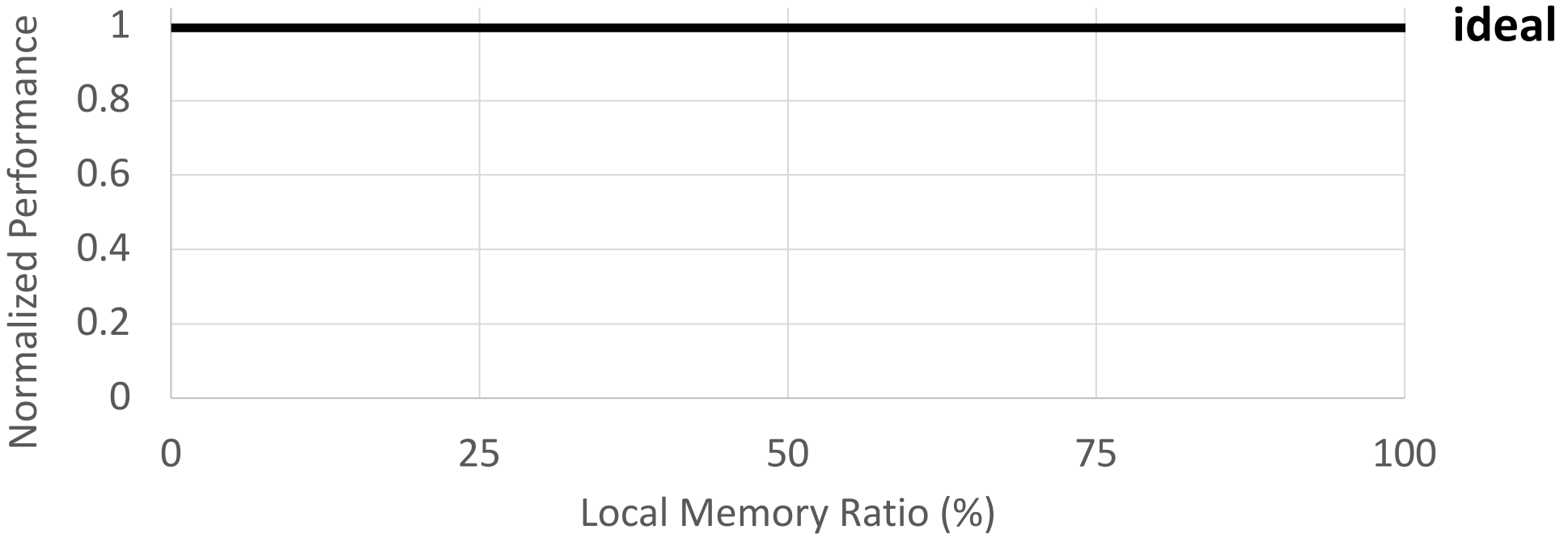
- Object-level caching. ➢ Avoid polluting local mem



Synthetic Web Frontend

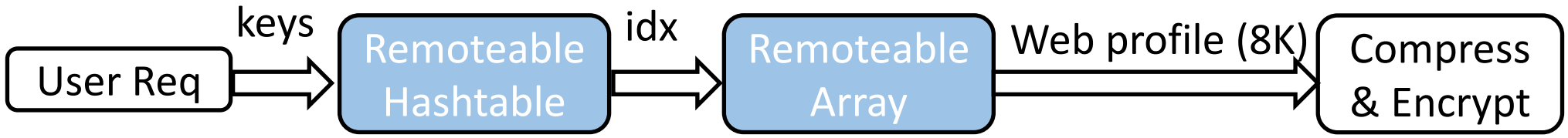
- Object-level caching.
- Avoid polluting local mem

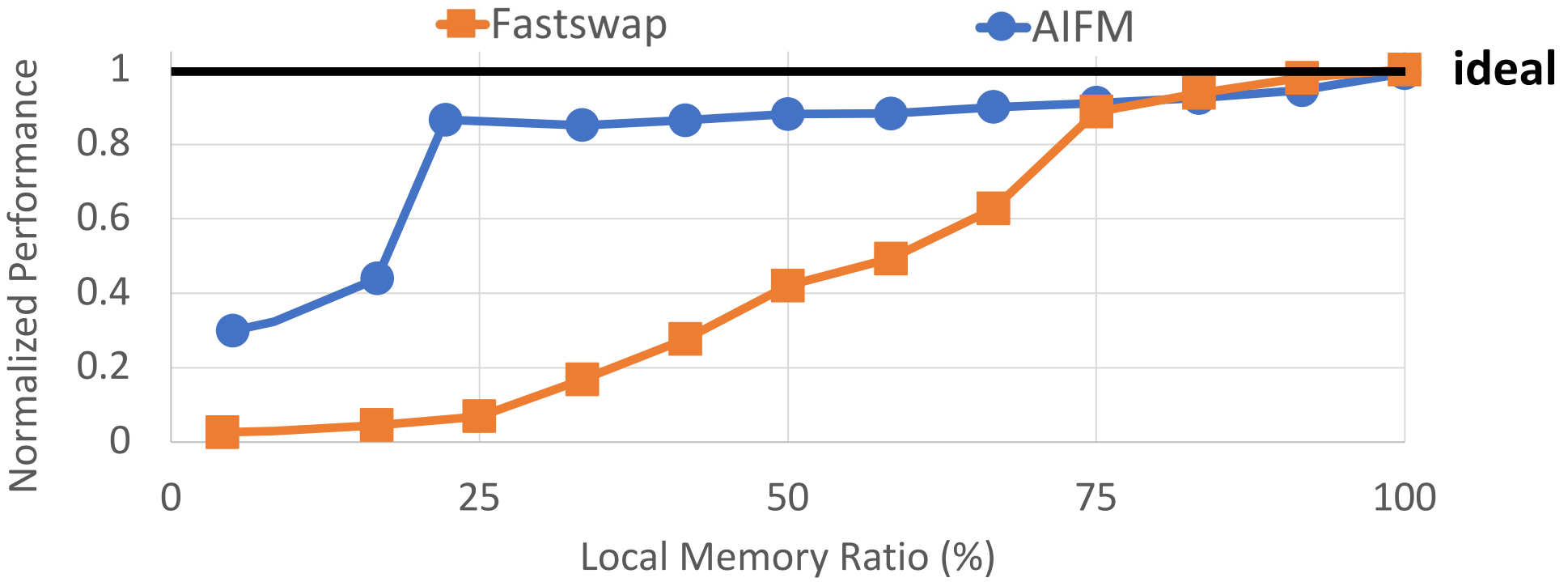




Synthetic Web Frontend

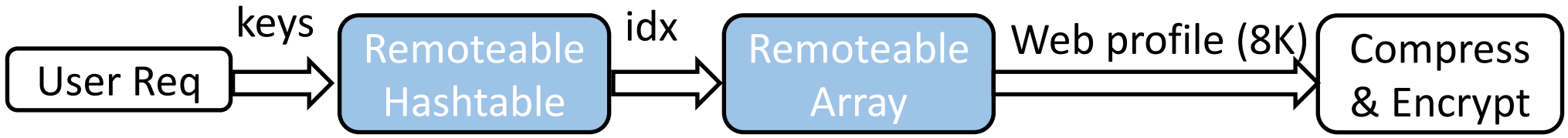
- Object-level caching.
- Avoid polluting local mem

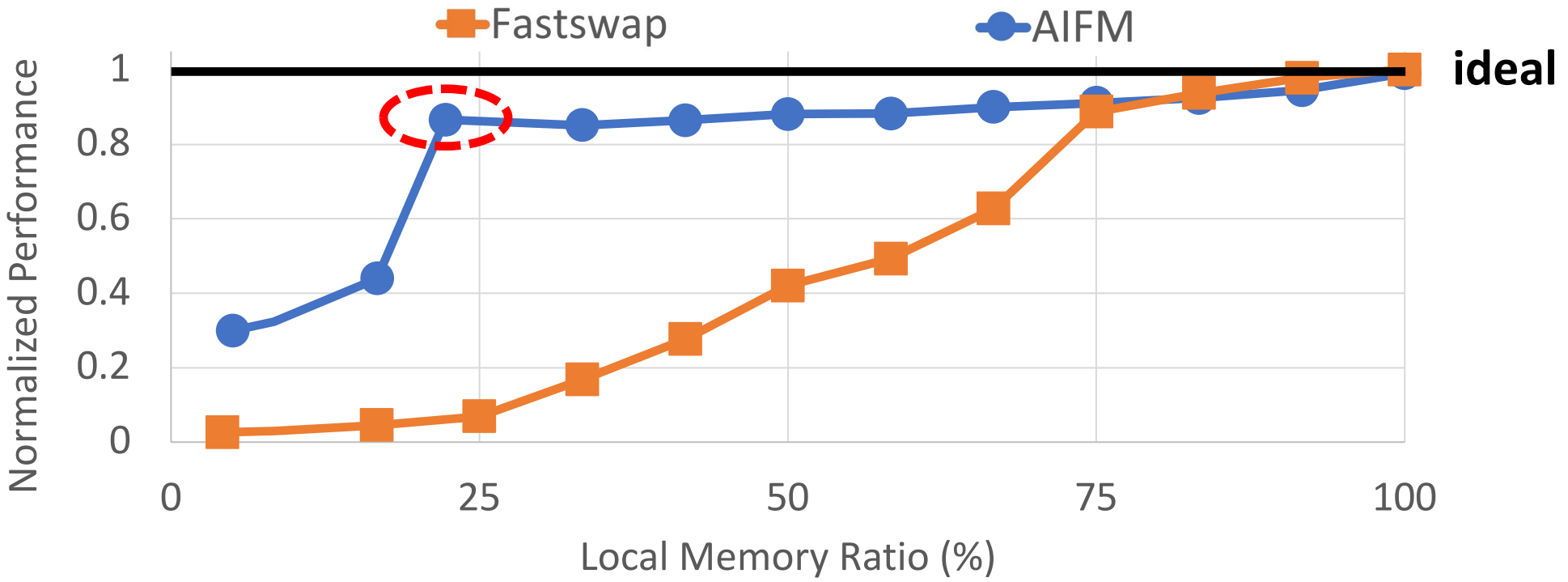




Synthetic Web Frontend

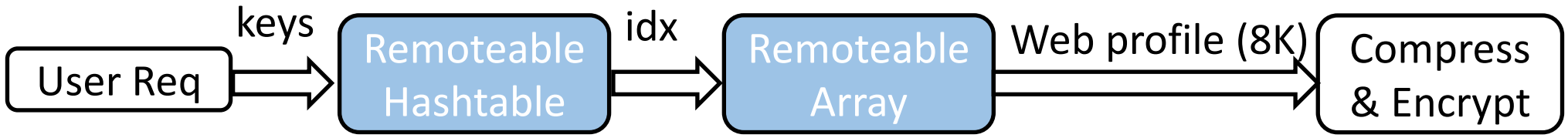
- Object-level caching.
- Avoid polluting local mem

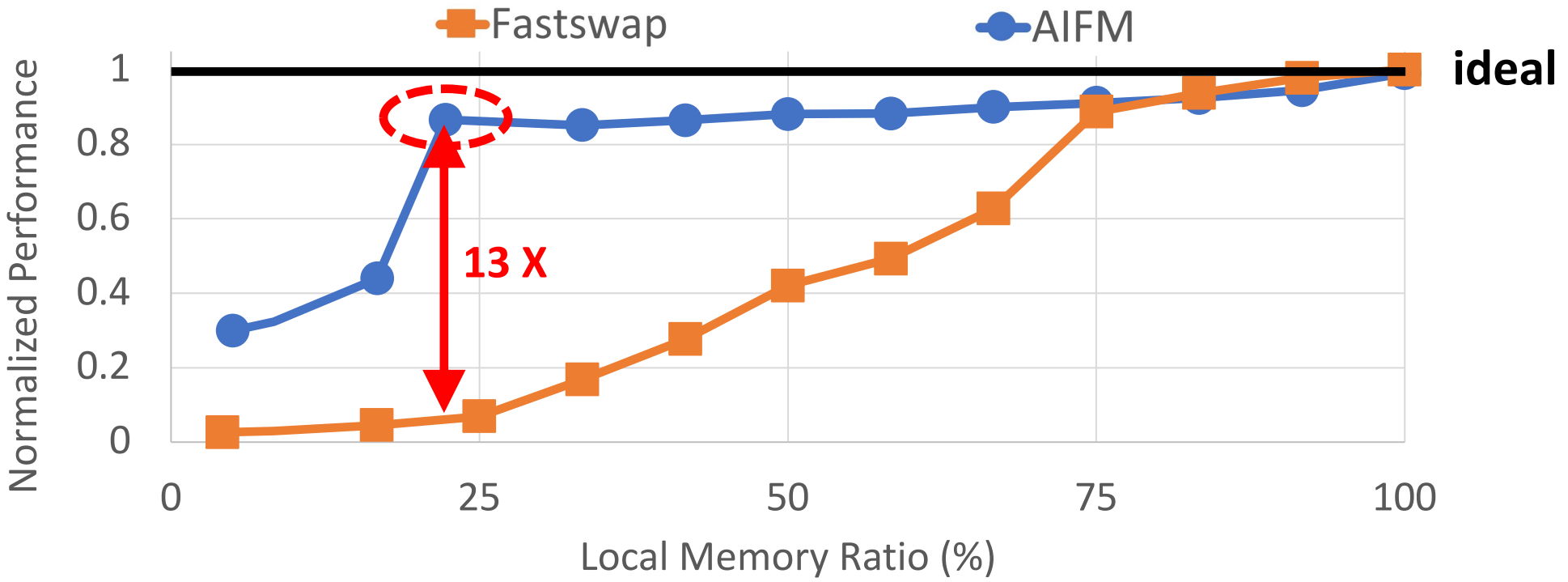




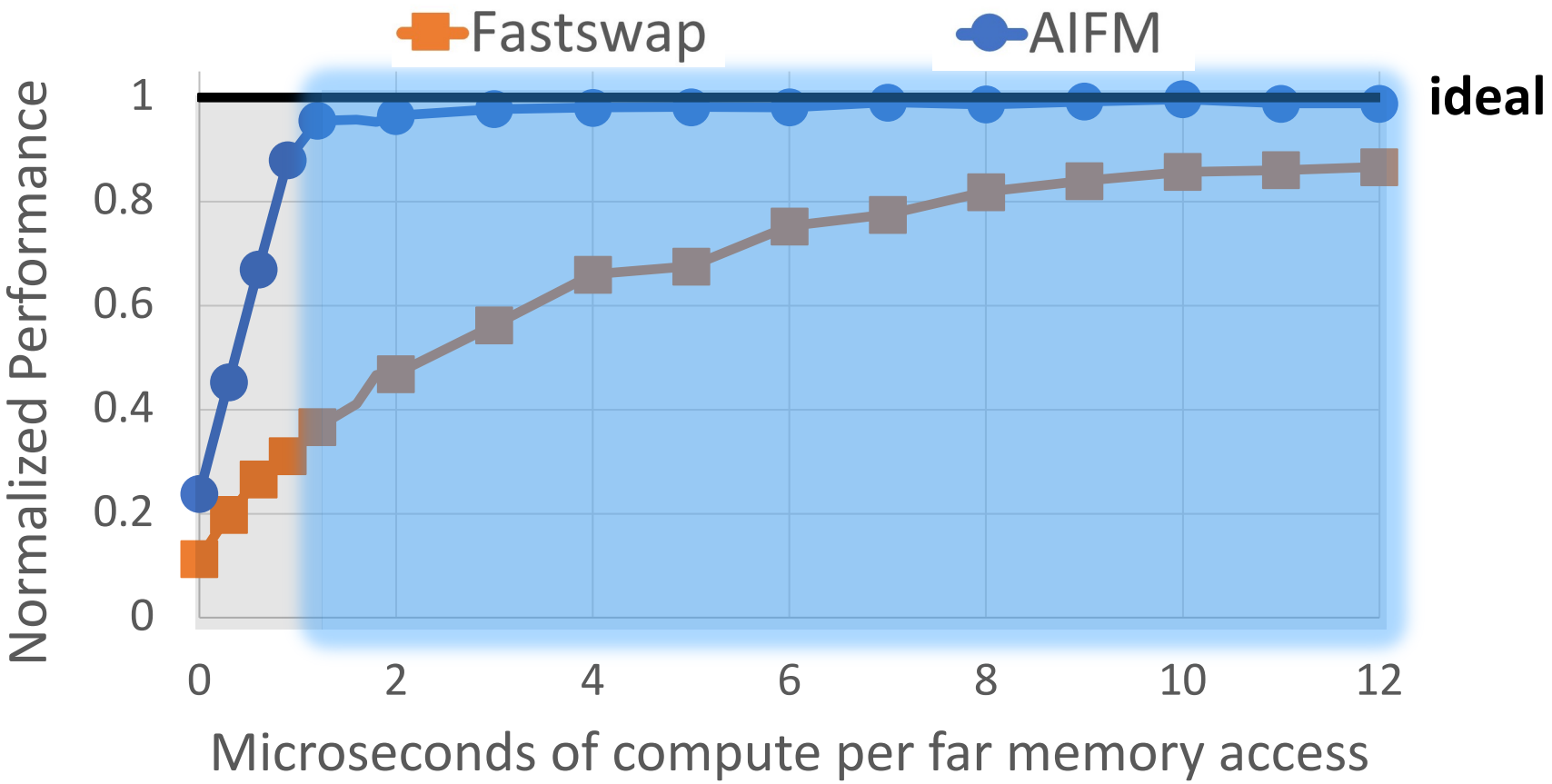
Synthetic Web Frontend

- Object-level caching.
- Avoid polluting local mem

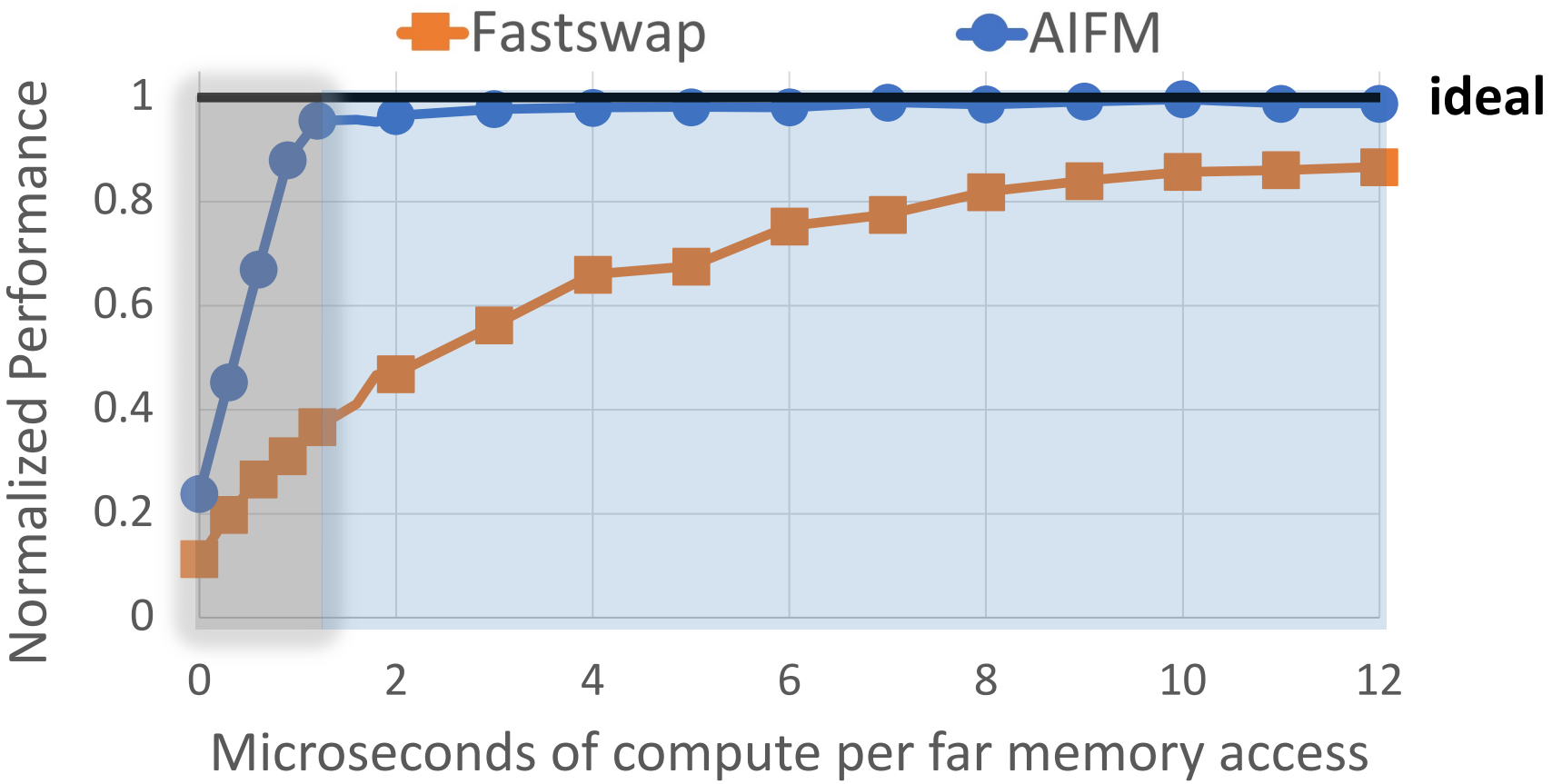




Performance on Different Compute Intensities



Performance on Different Compute Intensities



NYC Taxi Analysis (C++ DataFrame)

- DataFrame: data analytical framework, similar to Python Pandas.

NYC Taxi Analysis (C++ DataFrame)

- DataFrame: data analytical framework, similar to Python Pandas.

➢Real Kaggle workload: use DataFrame to explore trip dimensions.

  - Working set size = 31 GB.
  - Modify 1.4K LoC (out of 24.3K LoC), five person-days.

NYC Taxi Analysis (C++ DataFrame)

- DataFrame: data analytical framework, similar to Python Pandas.
- Real Kaggle workload: use DataFrame to explore trip dimensions.
  - Working set size = 31 GB.
  - Modify 1.4K LoC (out of 24.3K LoC), five person-days.

➢Relatively low compute intensity → Unable to hide far-mem latency.

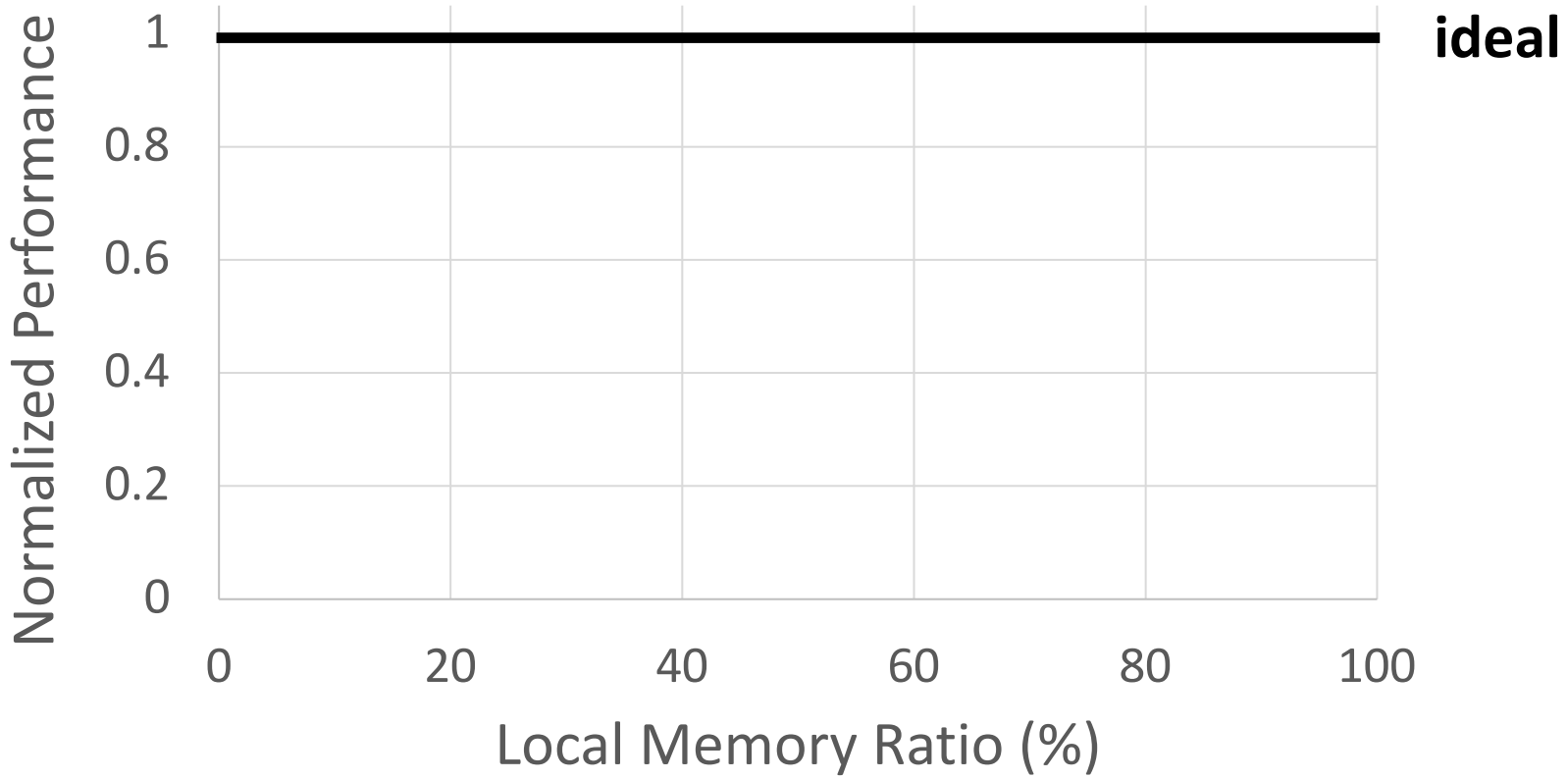
NYC Taxi Analysis (C++ DataFrame)

- DataFrame: data analytical framework, similar to Python Pandas.
- Real Kaggle workload: use DataFrame to explore trip dimensions.
  - Working set size = 31 GB.
  - Modify 1.4K LoC (out of 24.3K LoC), five person-days.
- Relatively low compute intensity → Unable to hide far-mem latency.

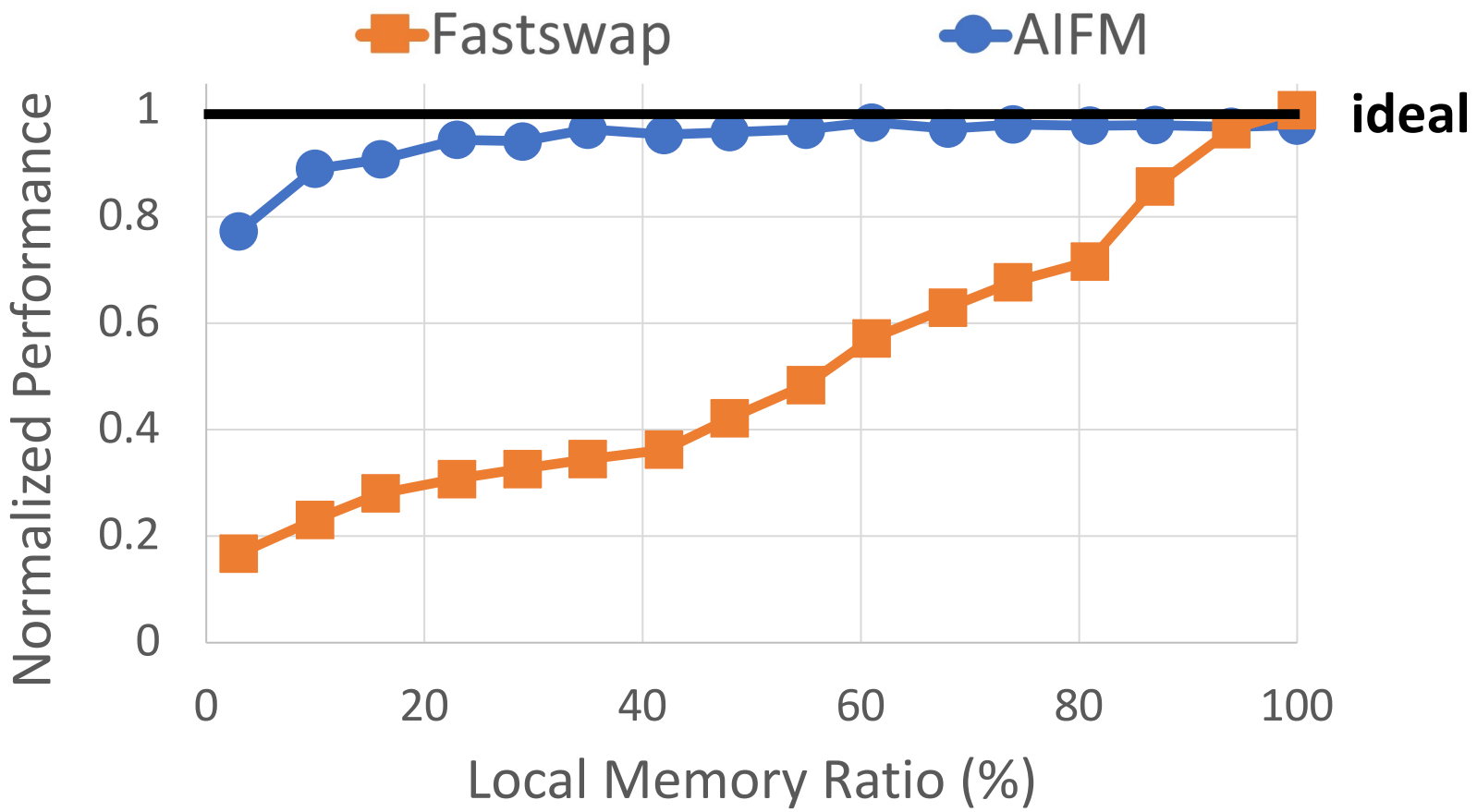
➢Keep complex operations locally and **offload** very light operations.

  - Significantly reduce expensive data transfer over network.

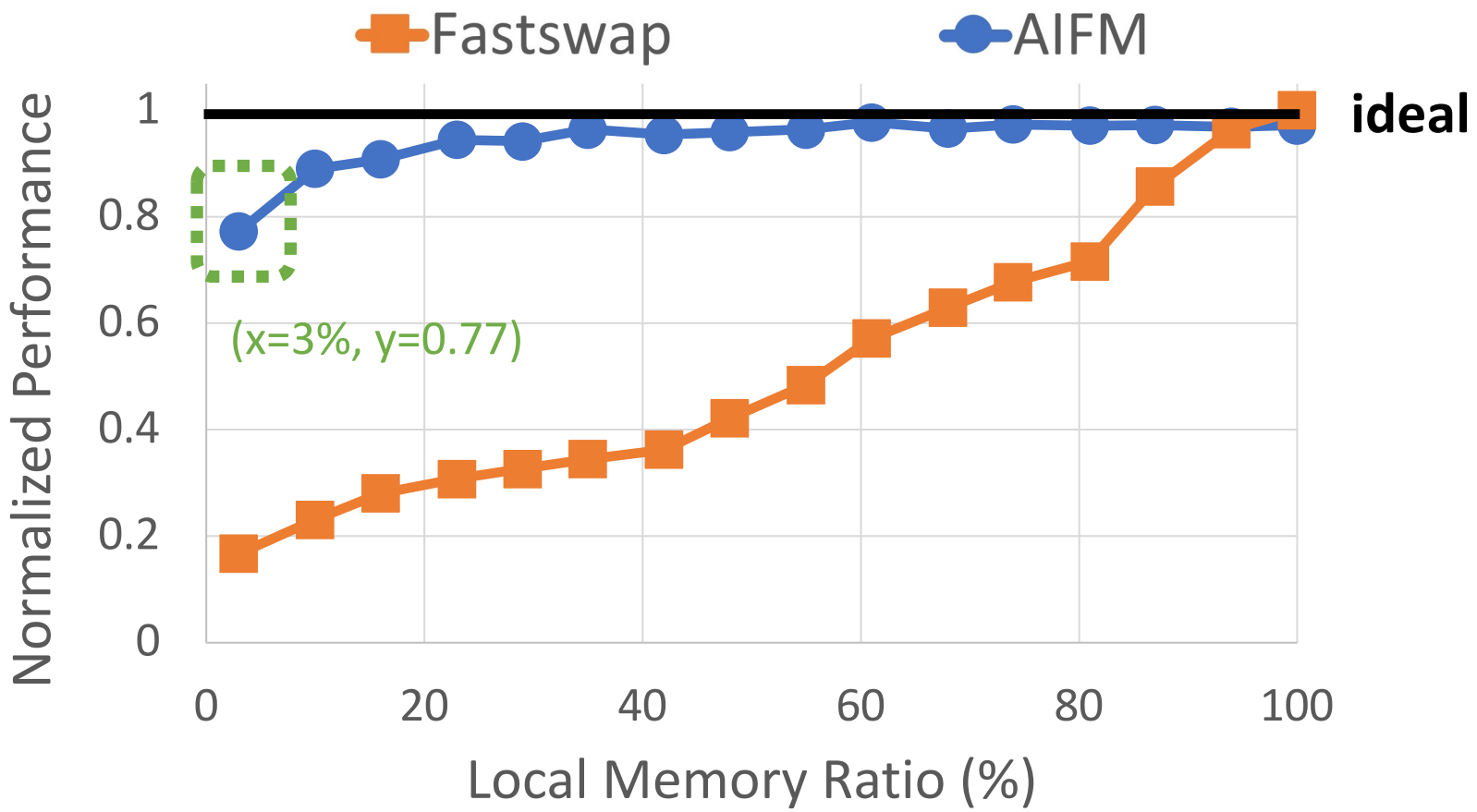
NYC Taxi Analysis (C++ DataFrame)



NYC Taxi Analysis (C++ DataFrame)

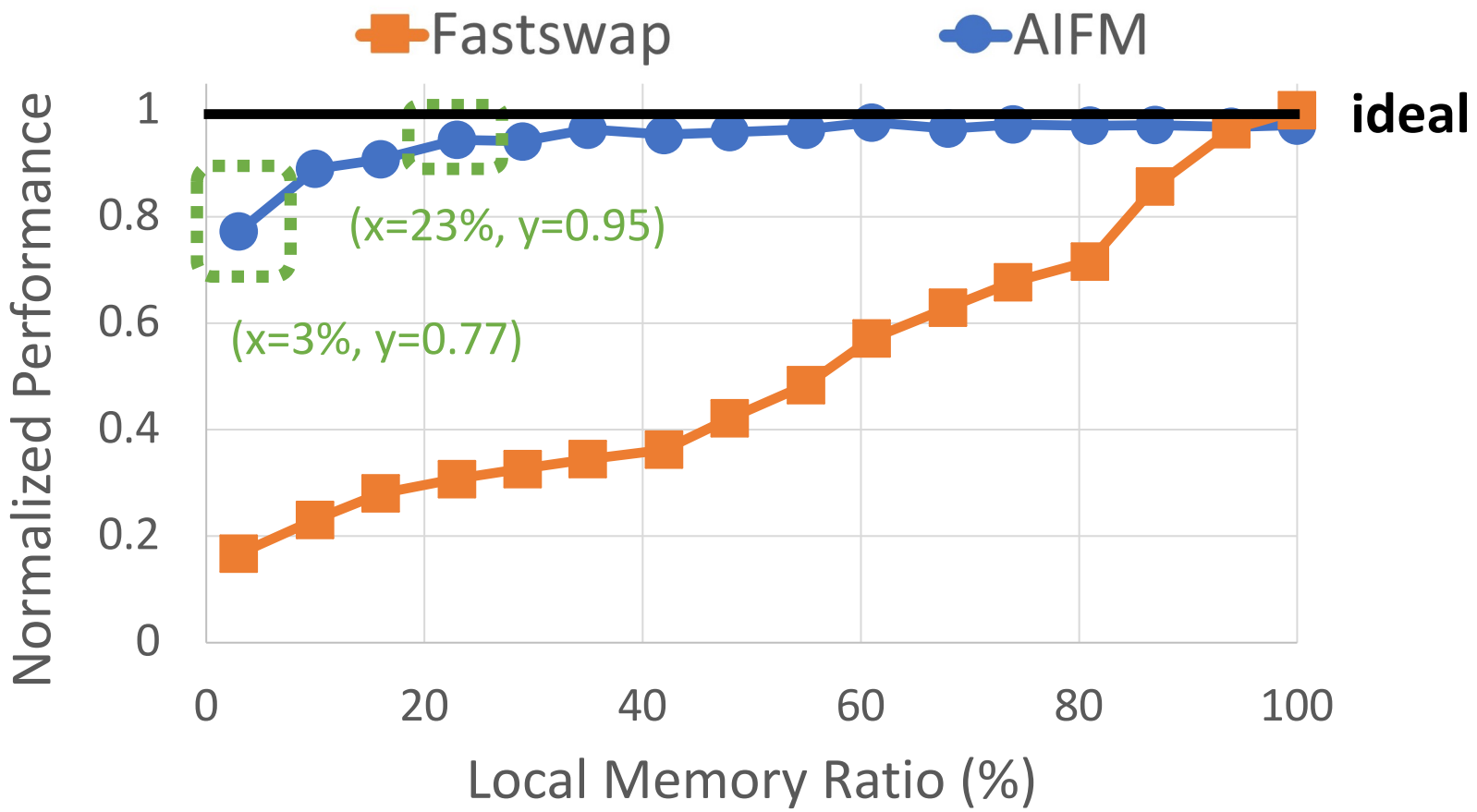


NYC Taxi Analysis (C++ DataFrame)



AIFM achieves near-ideal performance with small local memory.

NYC Taxi Analysis (C++ DataFrame)



AIFM achieves near-ideal performance with small local memory.

Conclusion

- AIFM: Application-Integrated Far Memory.

Conclusion

- AIFM: Application-Integrated Far Memory.

➢Key idea: swap memory using a userspace runtime.

Conclusion

- AIFM: Application-Integrated Far Memory.
- Key idea: swap memory using a userspace runtime.
  - Data Structure Library: captures application semantics.

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.

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.

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