Splinter: Bare-Metal Extensions for Multi-Tenant Low-Latency Storage

Chinmay Kulkarni, Sara Moore, Mazhar Naqvi, Tian Zhang, Robert Ricci, and Ryan Stutsman

University of Utah

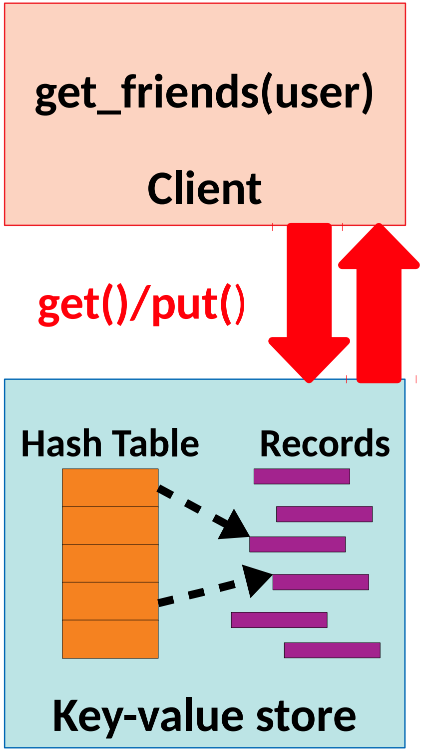
Introduction

- **Kernel-bypass** key-value stores offer **< 10µs** latency, **> Mops/s** throughput
  - Fast because they're just dumb?

- **Problem:** Leverage performance → **share** between tenants
- **Problem:** Apps require rich data models. Ex: Facebook's TAO
  - Implement using gets & puts? → **Data movement, client stalls**
  - Push code to key-value store? → **Isolation costs limit density**

- **Splinter: Multi-tenant** key-value store that code can be pushed to
  - Tenants push type- & memory-safe code written in **Rust** at runtime
  - **> 1000** tenants/server, **3.5 Million** ops/s, **9µs** median latency

Richer Data Models Come At A Price



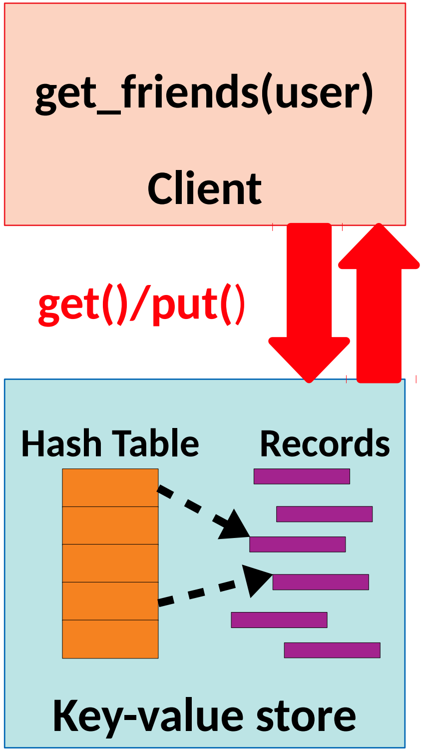
Apps require rich data models in addition to performance

- Ex: Social graphs, Decision trees etc.

Key-value stores trade-off data model for performance

- Simple get()’s & put()’s over key-value pairs

Richer Data Models Come At A Price



Apps require rich data models in addition to performance

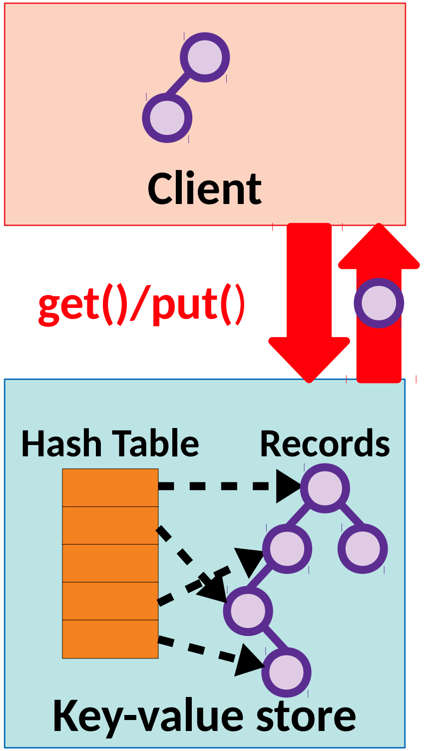
- Ex: Social graphs, Decision trees etc.

Key-value stores trade-off data model for performance

- Simple get()'s & put()'s over key-value pairs

Thinner data model → Better performance
But do applications benefit?

Extra Round-Trips (RTTs) Hurt Latency & Utilization



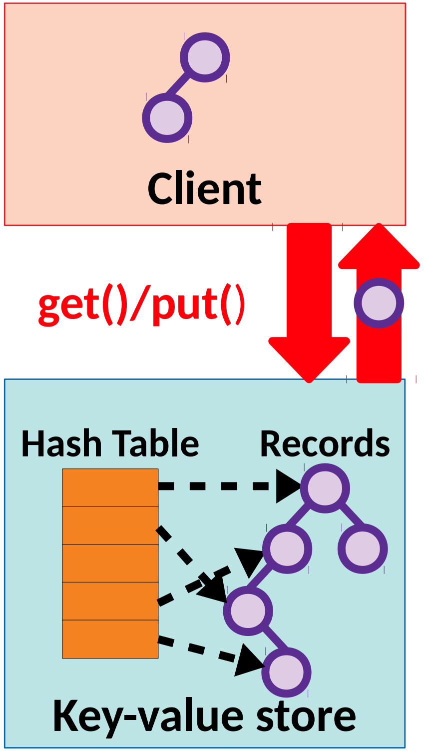
Example: Traverse tree with N nodes using gets

- One get() at each level of the tree → **O(log N) RTTs**
- Control flow depends on data → **Client stalls during get()**

Network RTTs, dispatch are the main bottleneck ~10µs

- 1.5µs inside the server

Extra Round-Trips (RTTs) Hurt Latency & Utilization



Example: Traverse tree with N nodes using gets

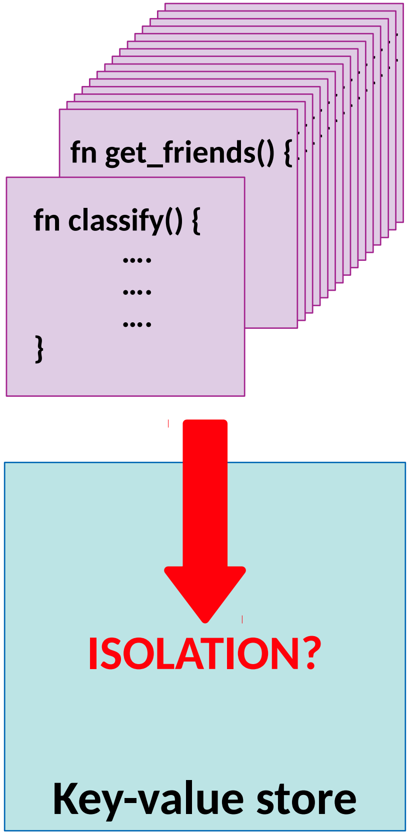
- One get() at each level of the tree → **O(log N) RTTs**
- Control flow depends on data → **Client stalls during get()**

Network RTTs, dispatch are the main bottleneck ~10µs

- 1.5µs inside the server

So push code to storage?

Why Not Push Compute To Storage?



RPC Processing Time ~1.5µs

Only native code will do

Context Switches ~1.5µs

Multi-tenancy → Need hardware isolation

What Do We Want From The Storage Layer?

Granularity of compute is steadily decreasing

Virtual machines → Containers → Lambdas

- **Extremely high tenant density**
  - Fine-grained resource allocation; 100s of CPU cycles, Kilobytes of memory

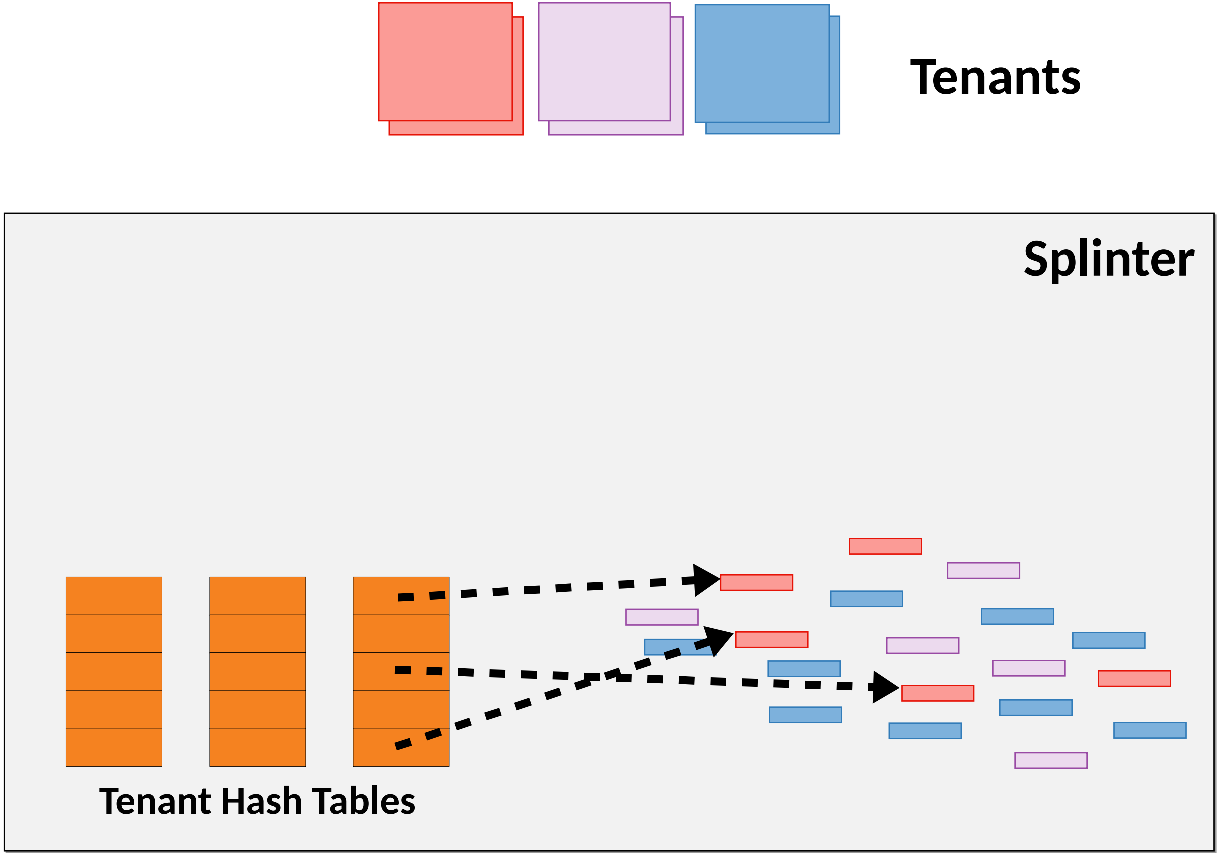
- **Allow tenants to extend data model at runtime**
  - Low overhead isolation between tenants & storage layer

Splinter: A Multi-Tenant Key-Value Store

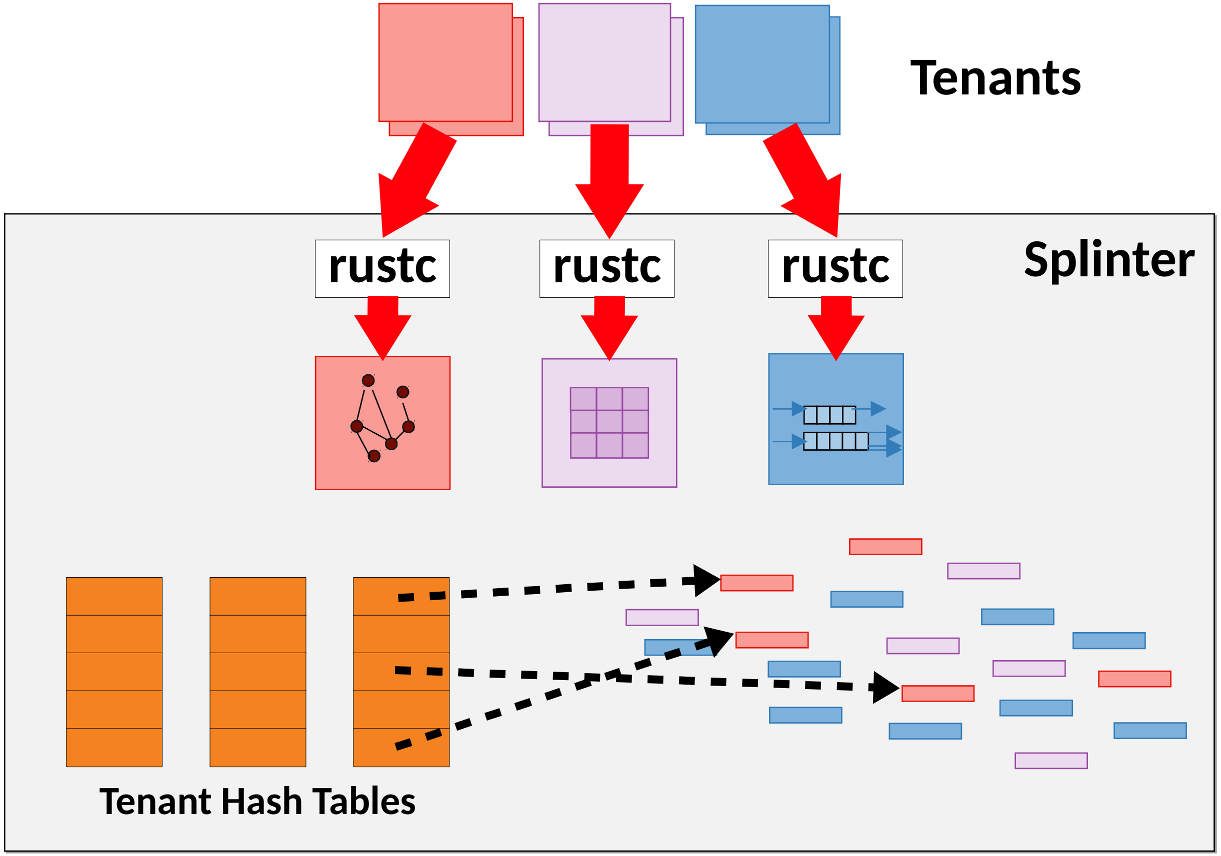
- **Tenants can install and invoke extensions at runtime**
  - Extensions written in **Rust**
  - Rely on type and memory safety for isolation, avoids context switch

- **Implemented in ~9000 lines of Rust**
  - Supports two RPCs **→ install(ext_name) & invoke(ext_name)**
  - Also supports regular get() & put() RPCs **→ “Native” operations**

1000 Foot View Of Splinter



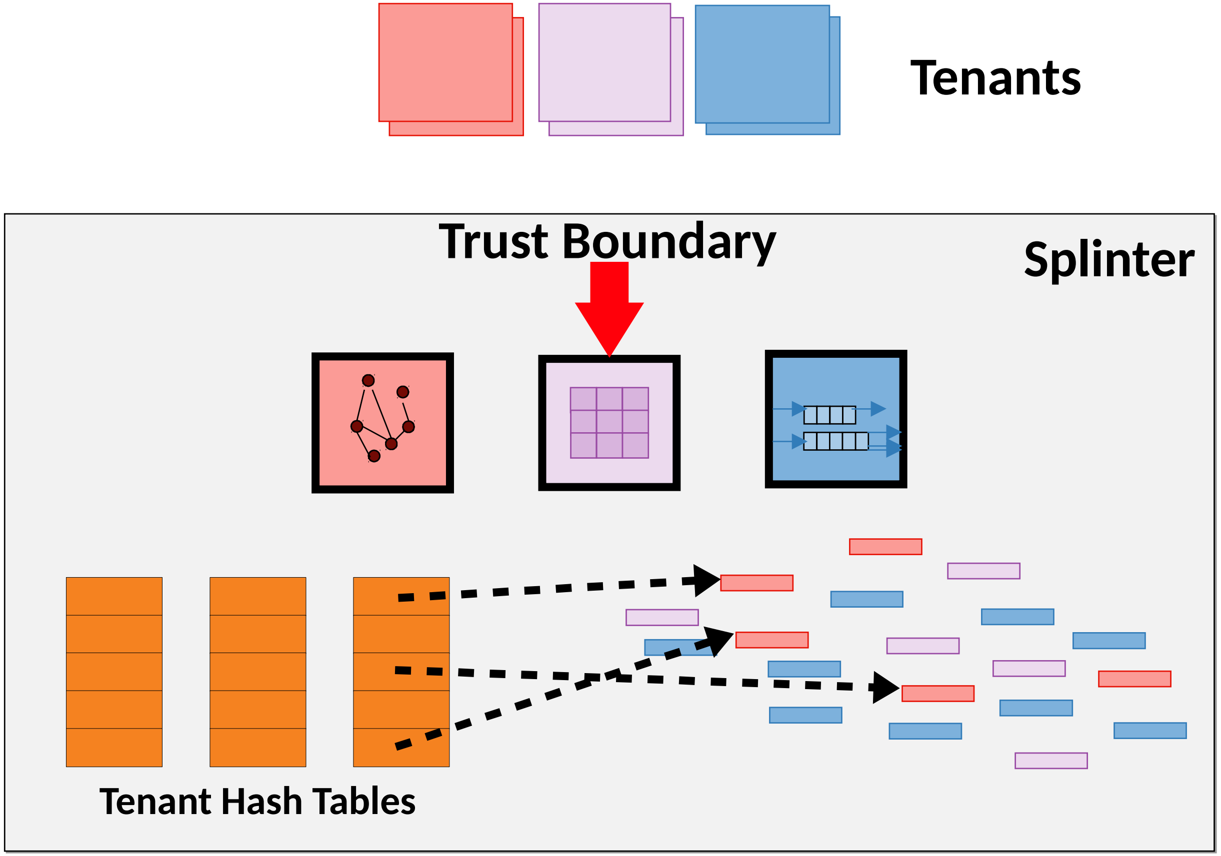
1000 Foot View Of Splinter



Tenants push extensions written in Rust

Splinter compiles, loads extensions Into address-space

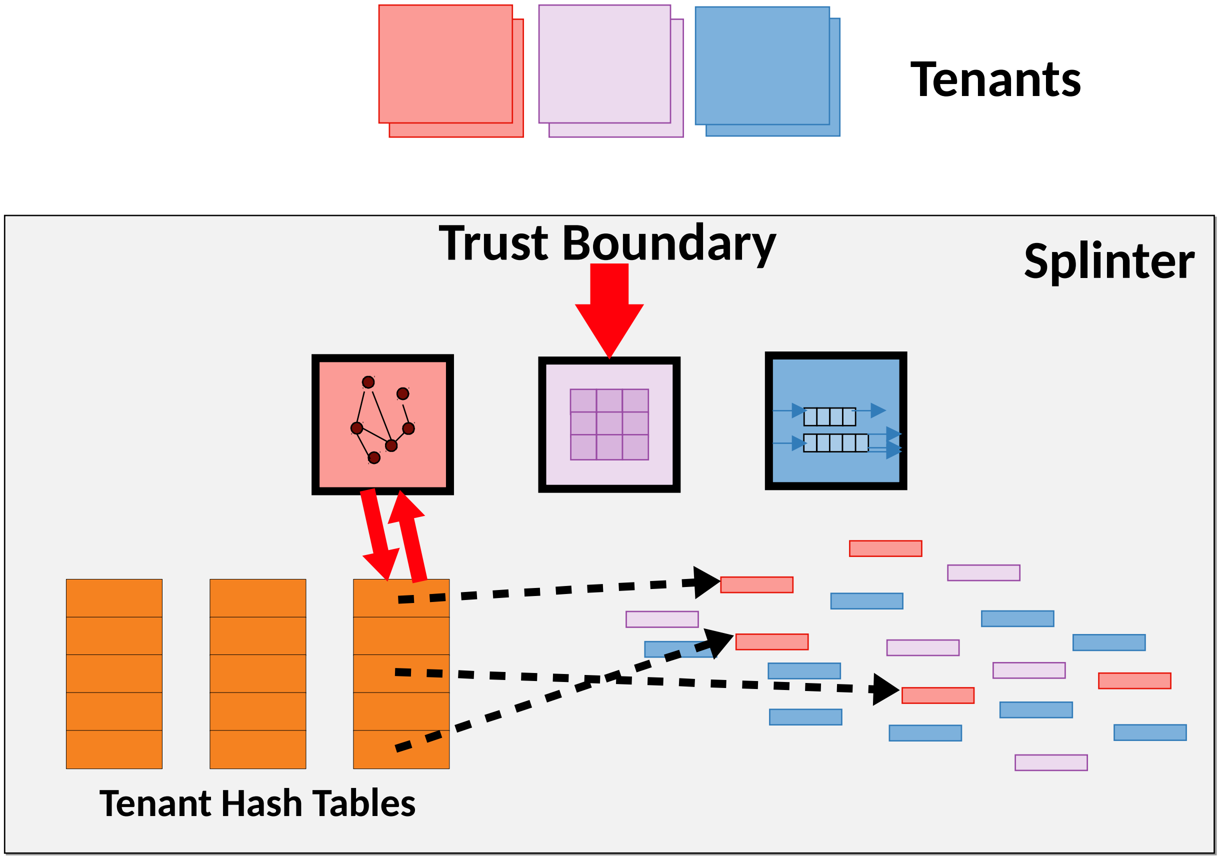
1000 Foot View Of Splinter



Rust provides memory-safety

Extensions do not share state

1000 Foot View Of Splinter



Extensions receive references to records

Each tenant sees a custom key-value store

Simple Aggregation With Splinter

Native mode

Client

1024 Tenants
100 GB Data

Splinter Server

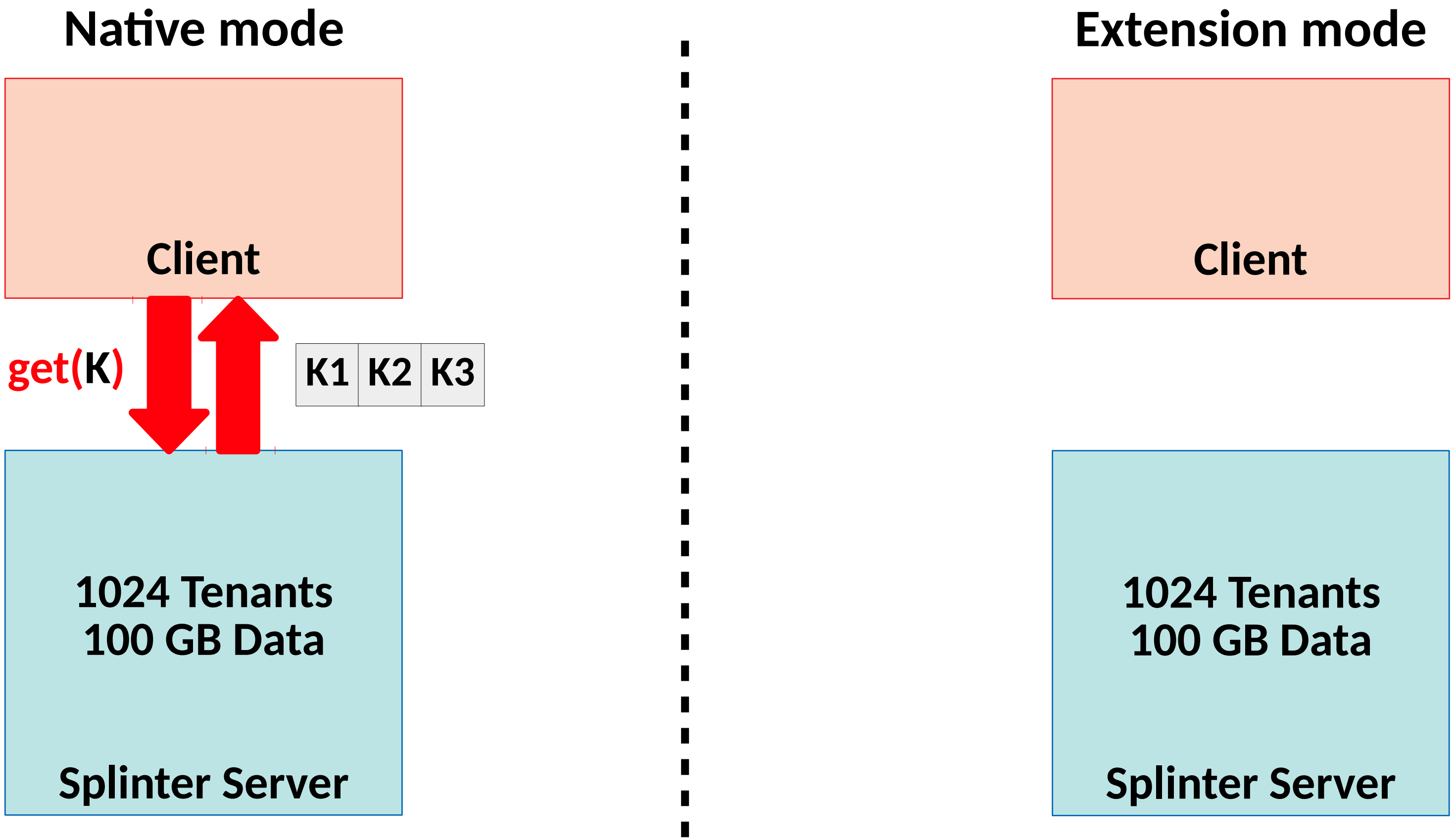
Extension mode

Client

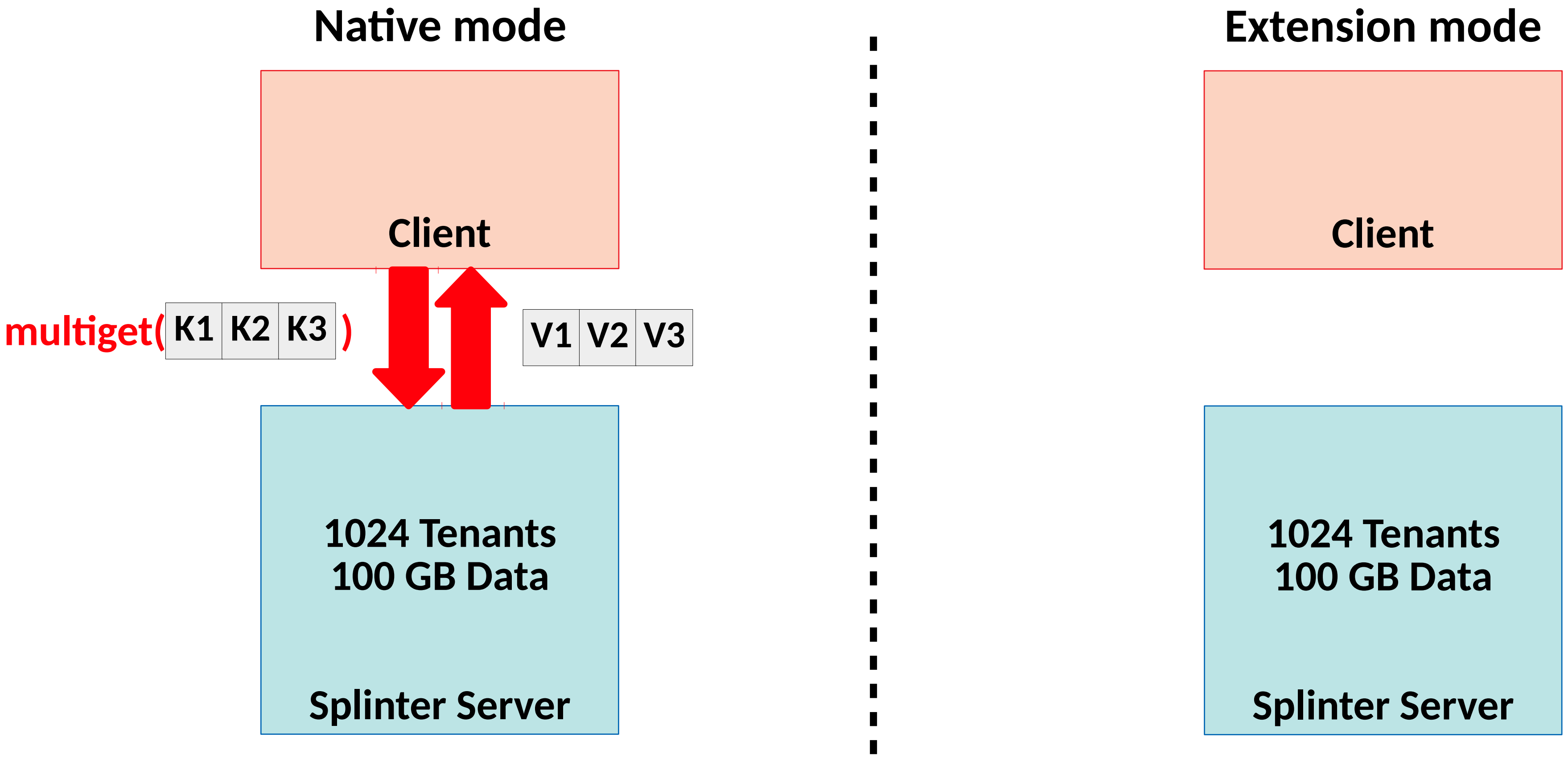
1024 Tenants
100 GB Data

Splinter Server

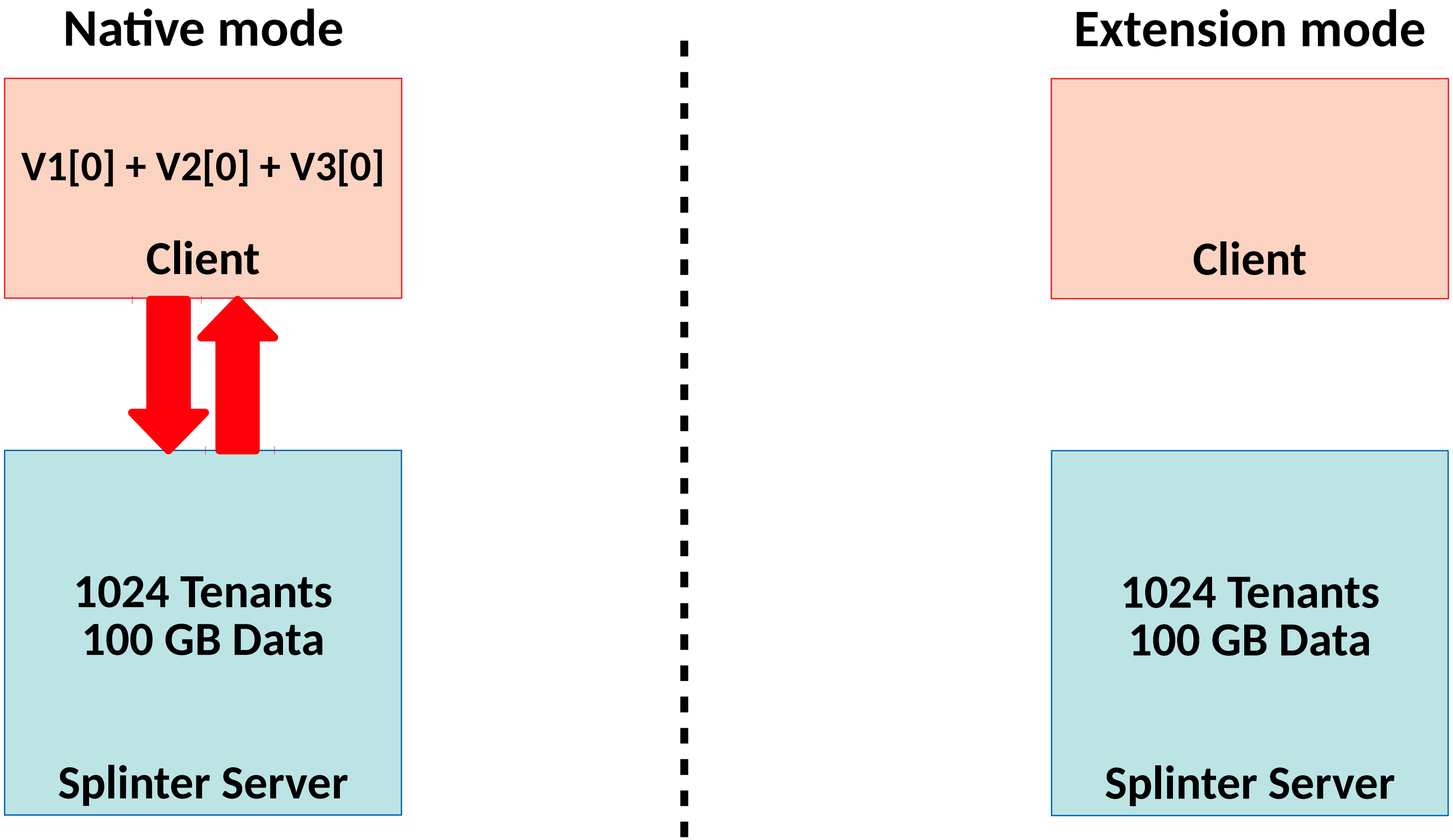
Simple Aggregation With Splinter



Simple Aggregation With Splinter



Simple Aggregation With Splinter



Native mode

V1[0] + V2[0] + V3[0]

Client

1024 Tenants
100 GB Data

Splinter Server

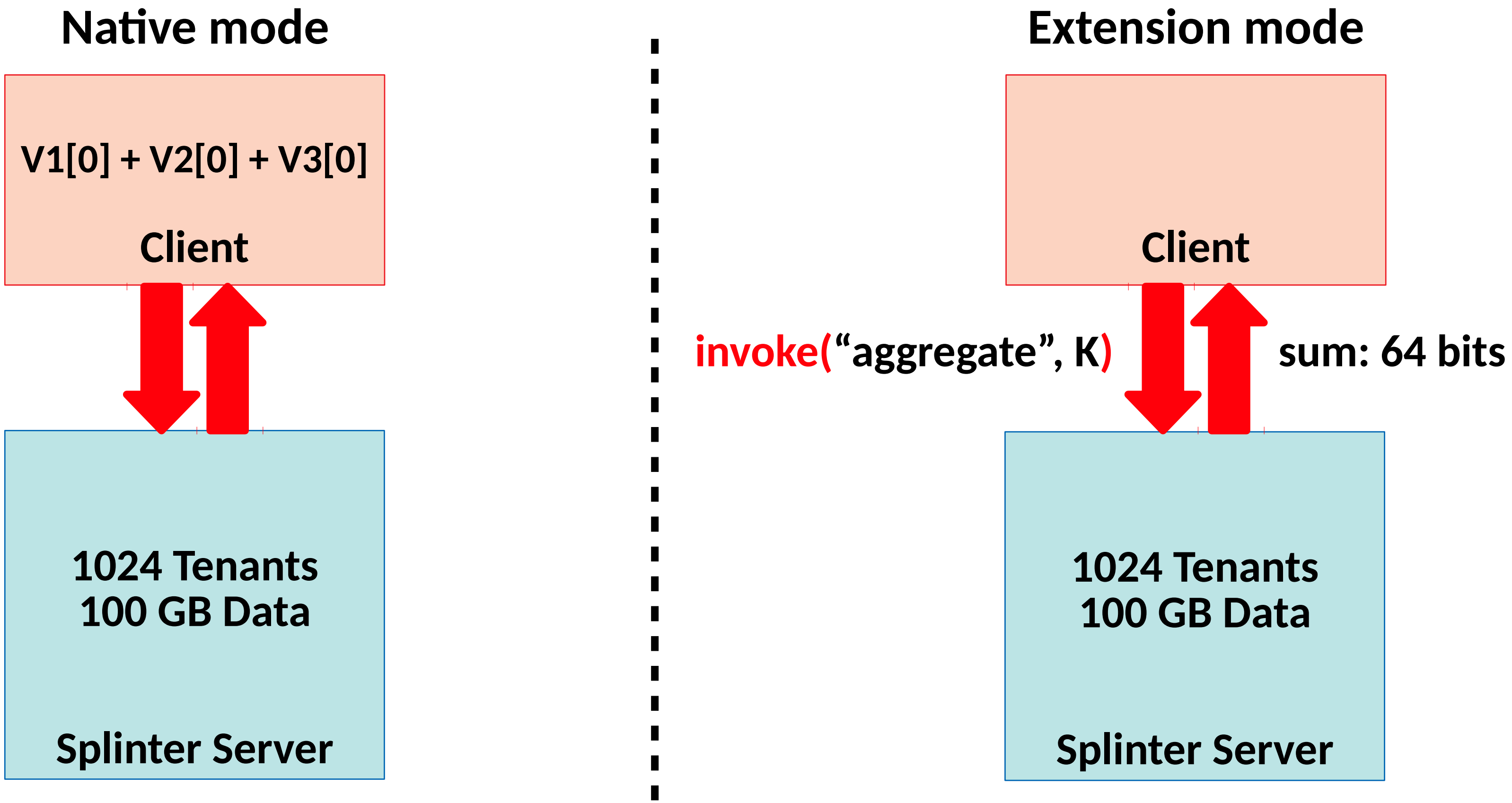
Extension mode

Client

1024 Tenants
100 GB Data

Splinter Server

Simple Aggregation With Splinter



Native mode

V1[0] + V2[0] + V3[0]

Client

1024 Tenants
100 GB Data

Splinter Server

Extension mode

Client

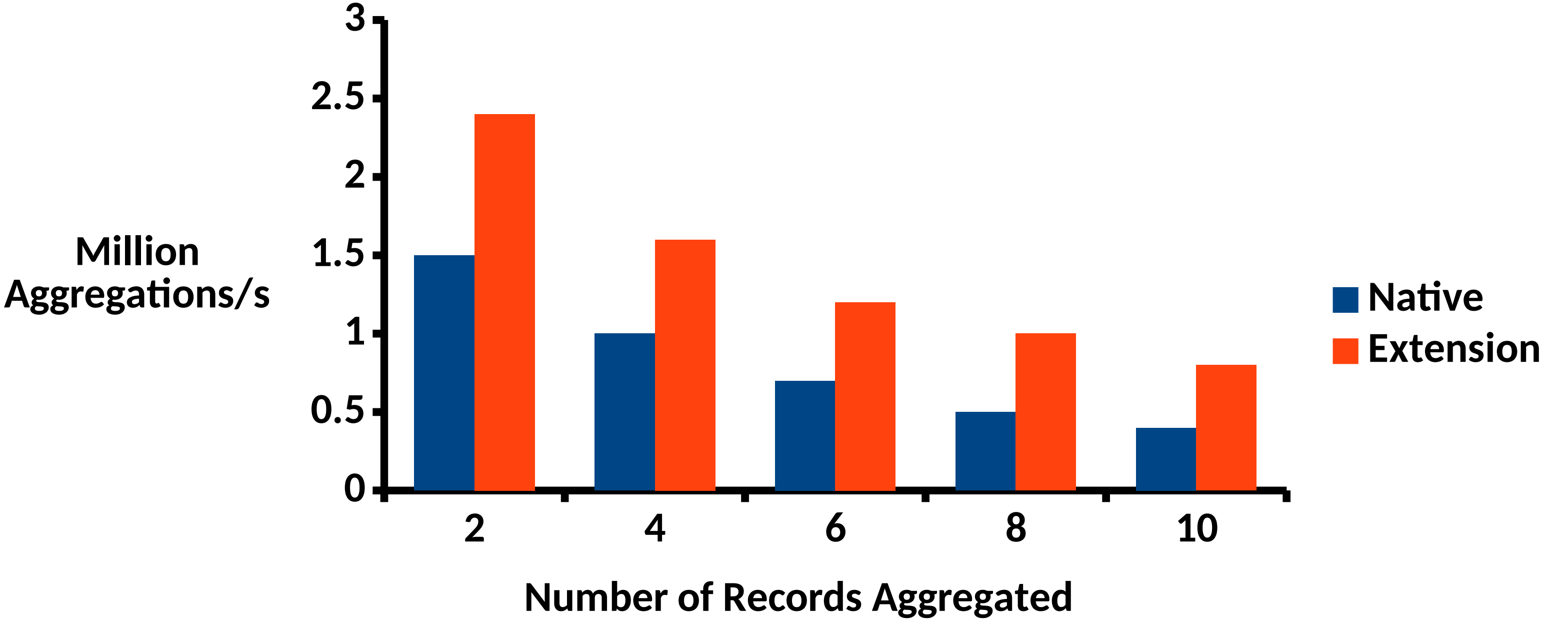
invoke(“aggregate”, K)

sum: 64 bits

1024 Tenants
100 GB Data

Splinter Server

Simple Aggregation With Splinter



Extension Mode → Few RPCs, Less Data movement → Better Throughput

Splinter: Design

- **Tenant Locality And Work Stealing**
  - Avoid cross-core coordination while avoiding hotspots

- **Lightweight Cooperative Scheduling**
  - Prevent long running extensions from starving short running ones

- **Low cost isolation**
  - No forced data copies across trust boundary

Splinter: Design

- **Tenant Locality And Work Stealing**
  - Avoid cross-core coordination while avoiding hotspots

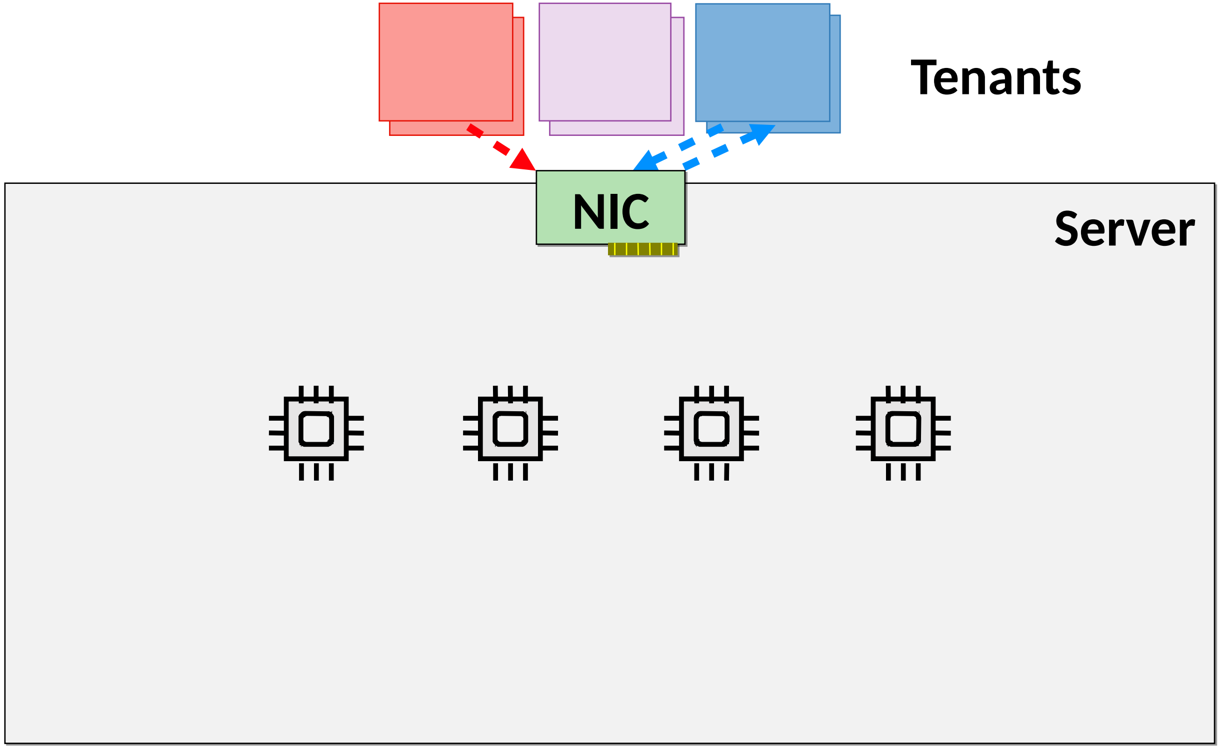
- Lightweight Cooperative Scheduling
  - Prevent long running extensions from starving short running ones

- Low cost isolation
  - No forced data copies across trust boundary

Splinter: Tenant Locality And Work Stealing

Problem: **Quickly dispatch requests to cores, avoid hotspots**

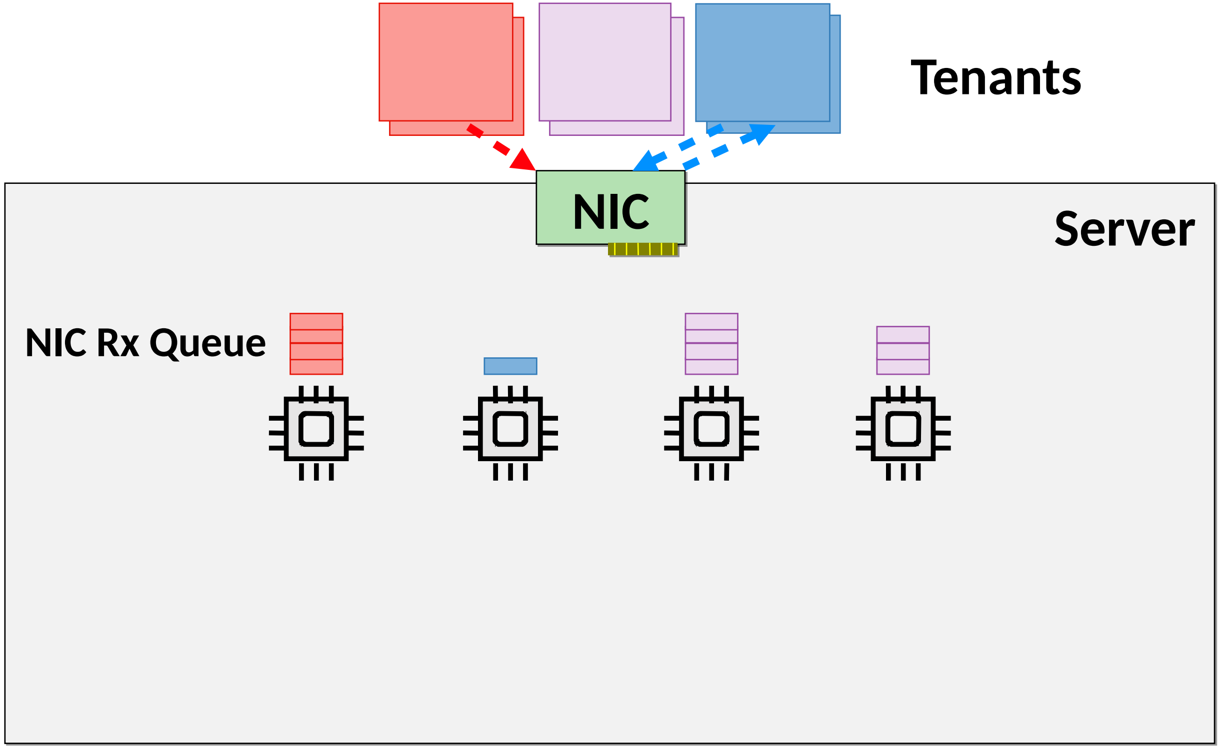
Solution: **NIC routes tenants to cores, cores steal work**



Splinter: Tenant Locality And Work Stealing

Problem: **Quickly dispatch requests to cores, avoid hotspots**

Solution: **NIC routes tenants to cores, cores steal work**

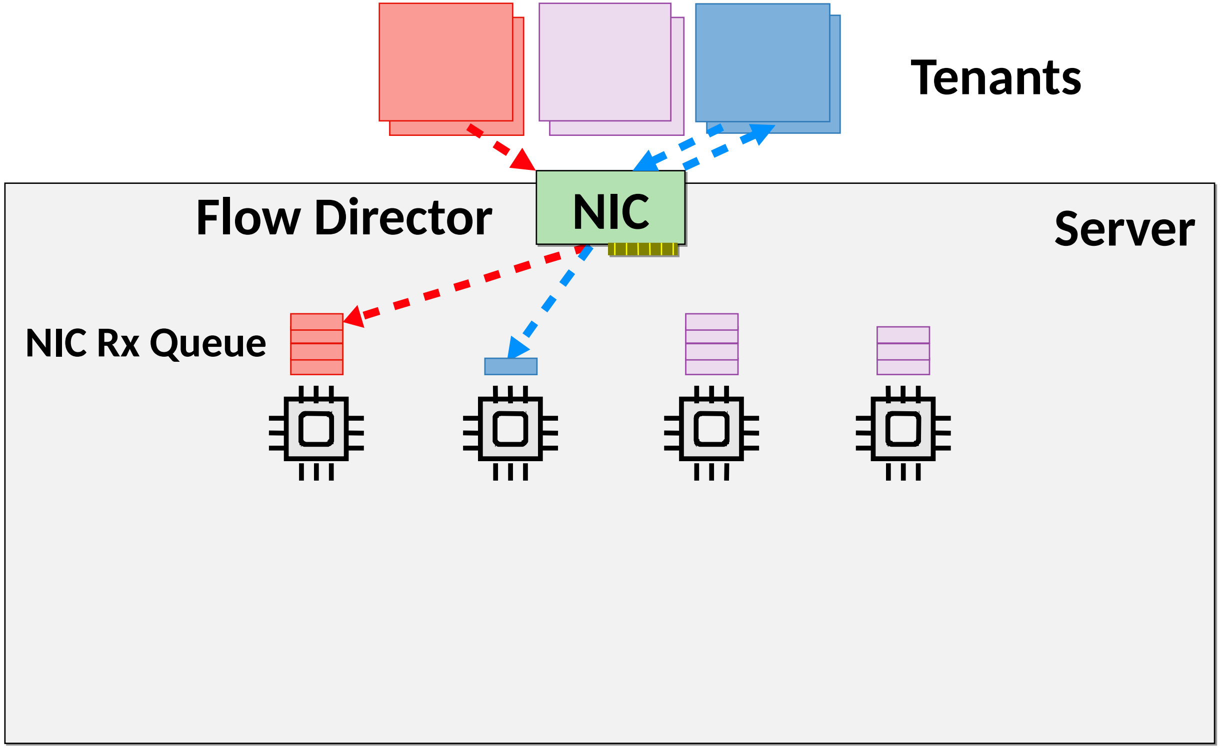


One Rx queue per core

Splinter: Tenant Locality And Work Stealing

Problem: **Quickly dispatch requests to cores, avoid hotspots**

Solution: **NIC routes tenants to cores, cores steal work**

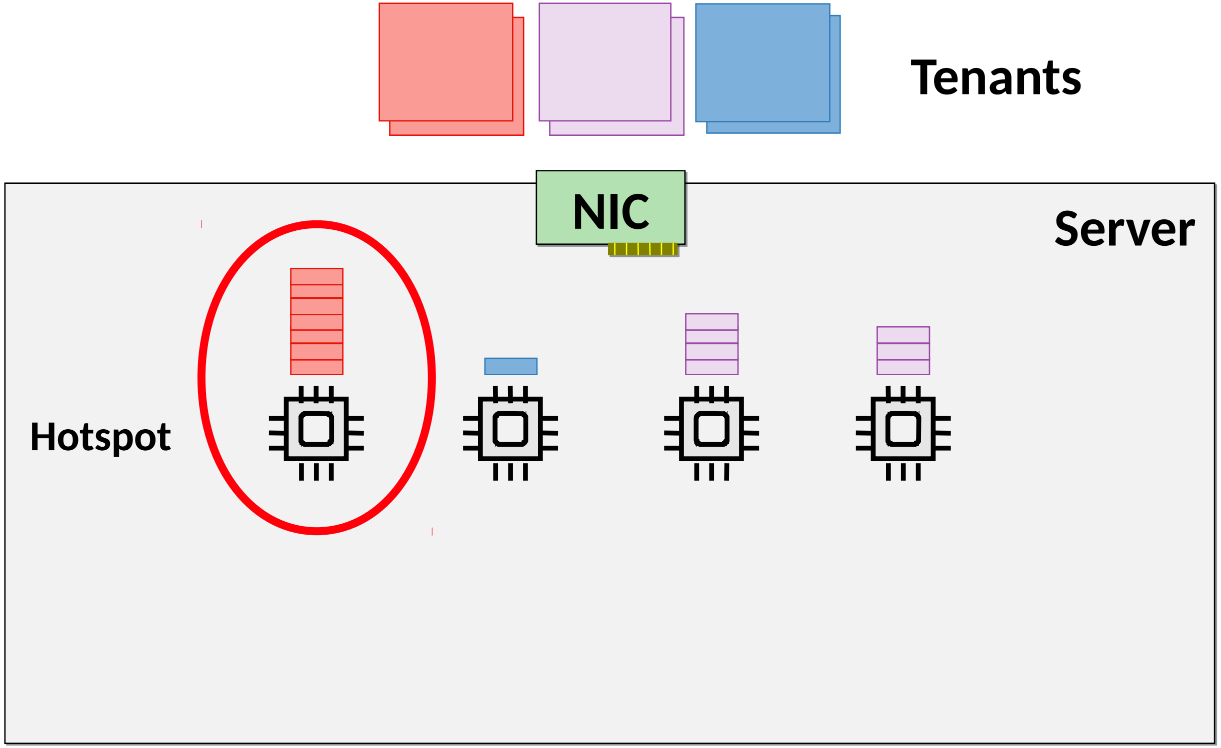


Maintain "Locality" route tenant to queue

Splinter: Tenant Locality And Work Stealing

Problem: **Quickly dispatch requests to cores, avoid hotspots**

Solution: **NIC routes tenants to cores, cores steal work**

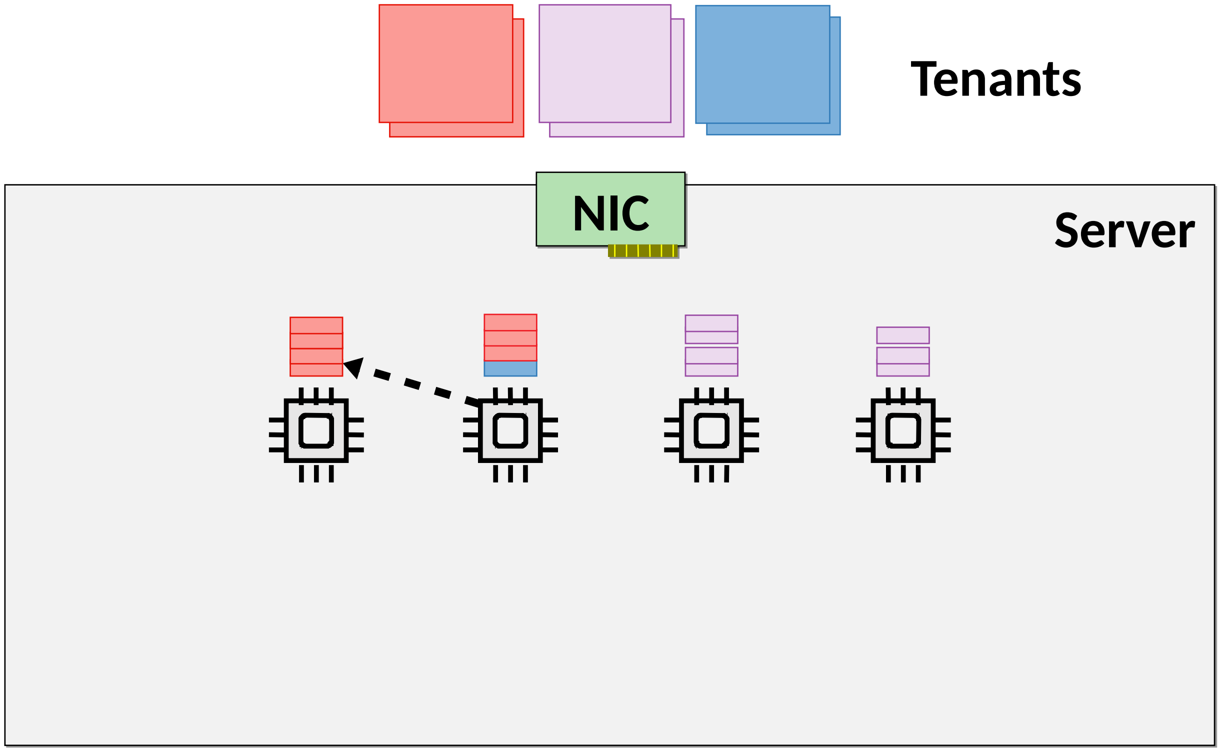


Few active tenants
Many idle tenants

Splinter: Tenant Locality And Work Stealing

Problem: **Quickly dispatch requests to cores, avoid hotspots**

Solution: **NIC routes tenants to cores, cores steal work**



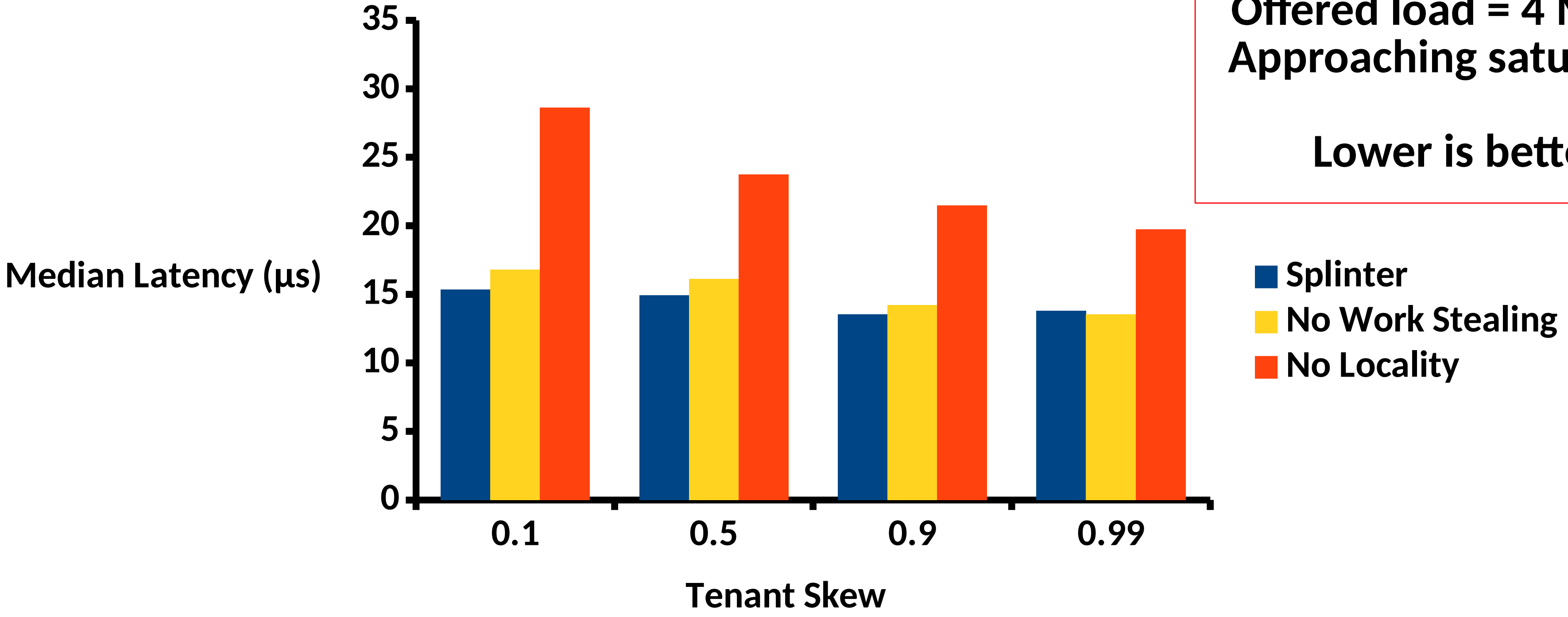
Cores steal from neighboring queue

What are the benefits of tenant locality & work stealing?

Setup:

- 1024 tenants
- Invoke small extension that reads one object

Performance With Tenant Locality & Work Stealing



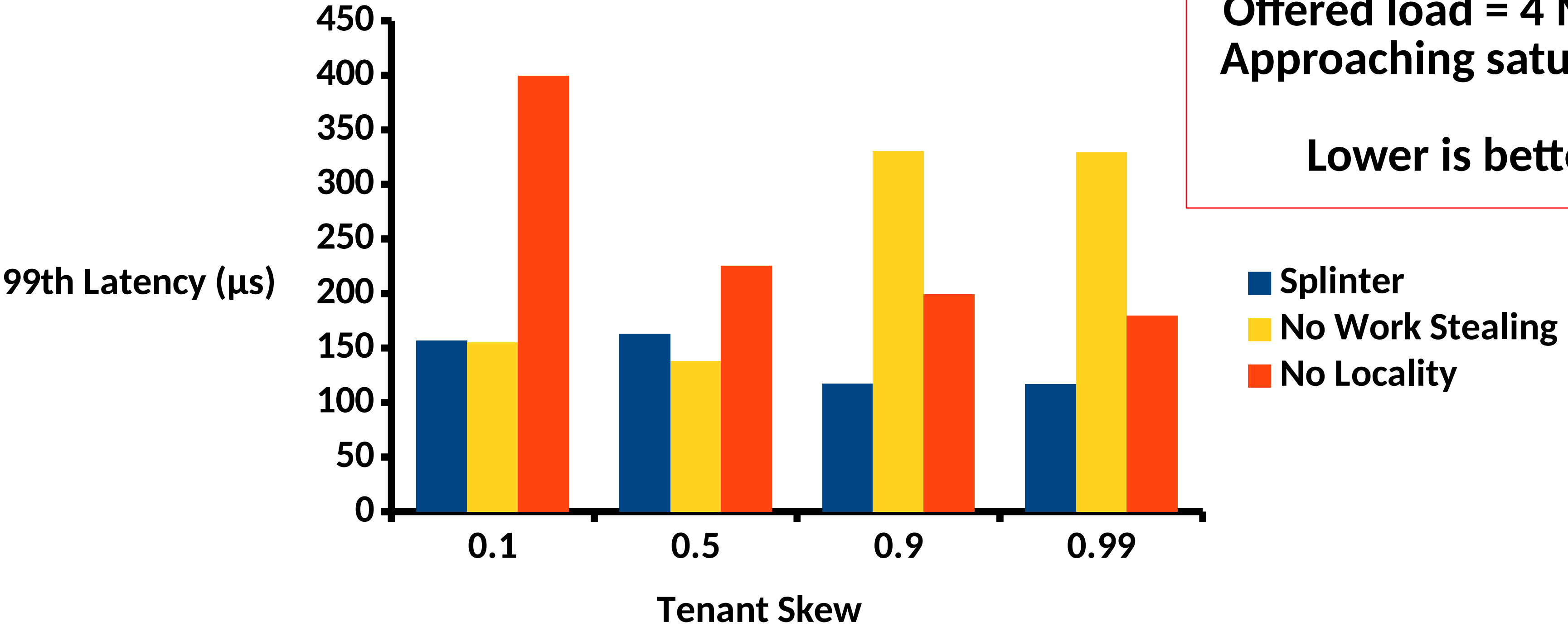
Offered load = 4 Mop/s
Approaching saturation

Lower is better

Higher tenant skew → Fewer active tenants

No Tenant Locality → Poor median Latency

Performance With Tenant Locality & Work Stealing



Offered load = 4 Mop/s
Approaching saturation

Lower is better

Higher tenant skew → Fewer active tenants

No work stealing → Poor tail Latency under high skew

Splinter: Design

- Tenant Locality And Work Stealing
  - Avoid cross-core coordination while avoiding hotspots

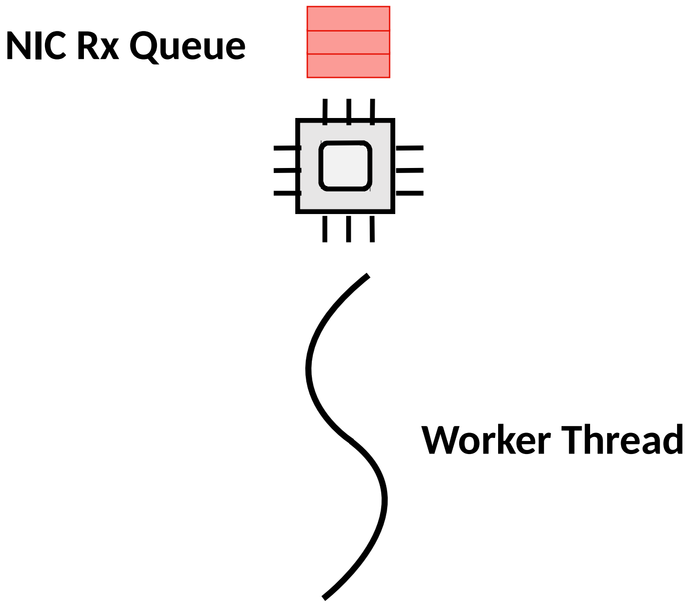
- **Lightweight Cooperative Scheduling**
  - Prevent long running extensions from starving short running ones

- Low cost isolation
  - No forced data copies across trust boundary

Splinter: Lightweight Cooperative Scheduling

Problem: Minimize trust boundary crossing cost

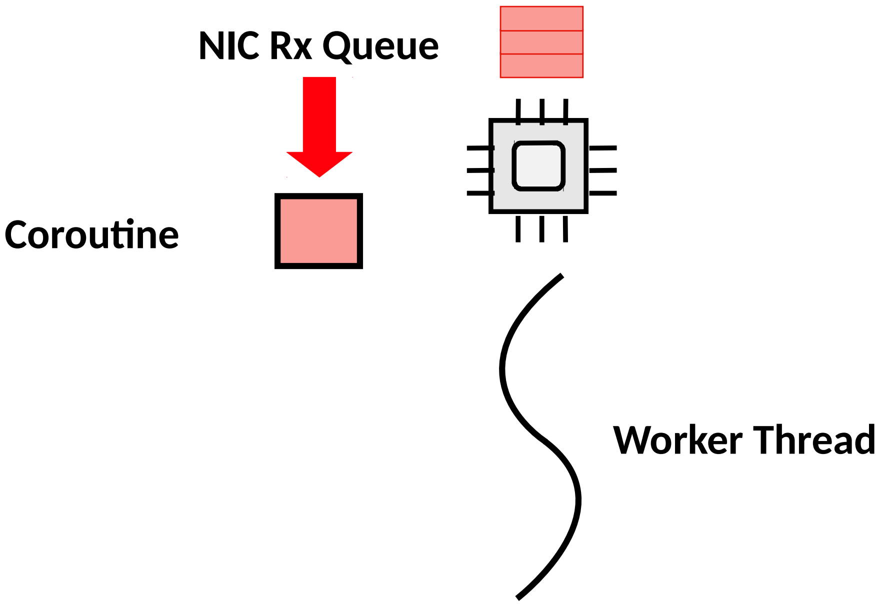
Solution: Run extensions in stackless coroutines



Splinter: Lightweight Cooperative Scheduling

Problem: Minimize trust boundary crossing cost

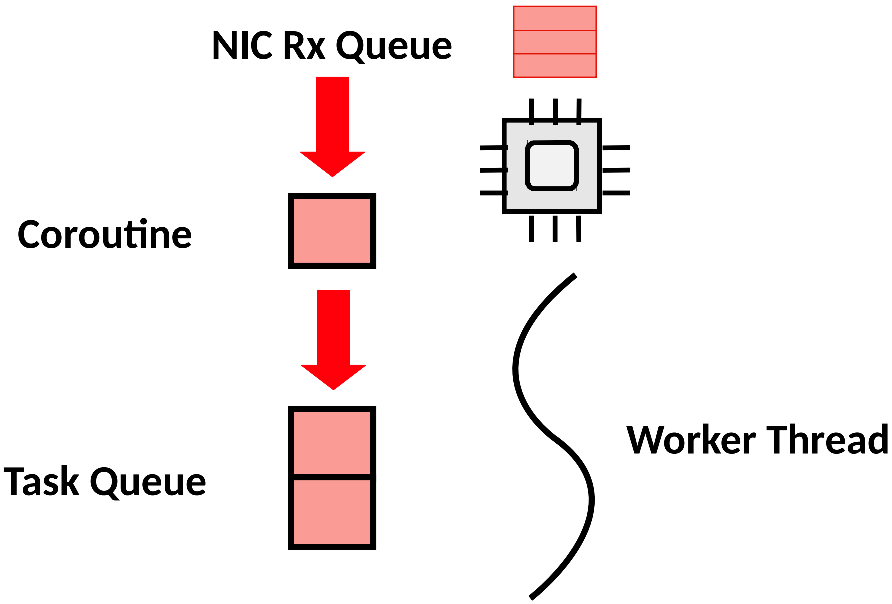
Solution: Run extensions in stackless coroutines



Splinter: Lightweight Cooperative Scheduling

Problem: Minimize trust boundary crossing cost

Solution: Run extensions in stackless coroutines



NIC Rx Queue

Coroutine

Task Queue

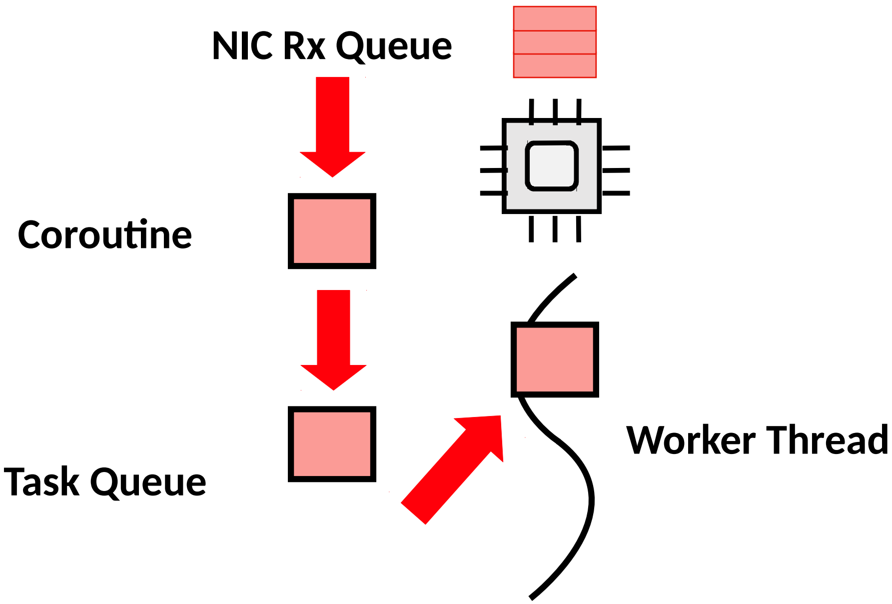
Worker Thread

Dedicated Dispatch task to construct coroutines

Splinter: Lightweight Cooperative Scheduling

Problem: **Minimize trust boundary crossing cost**

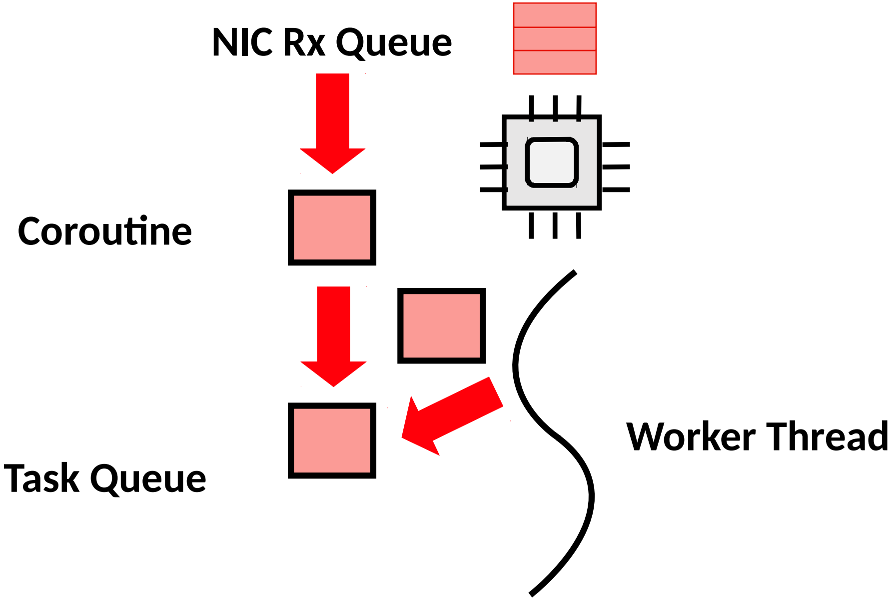
Solution: **Run extensions in stackless coroutines**



Splinter: Lightweight Cooperative Scheduling

Problem: **Minimize trust boundary crossing cost**

Solution: **Run extensions in stackless coroutines**



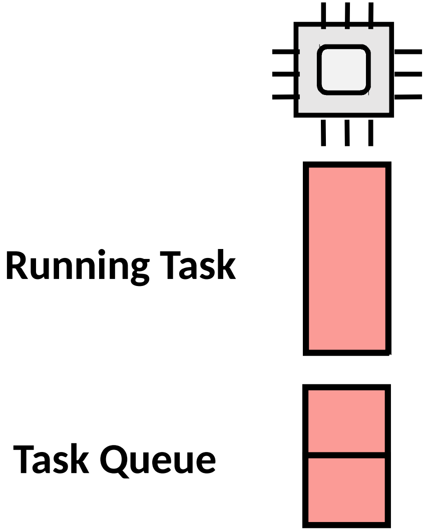
Task switch cost ~10ns

Run extension until it returns

Splinter: Lightweight Cooperative Scheduling

Problem: Long running tasks starve shorter tasks, hurt latency

Solution: Extensions are cooperative, must yield frequently

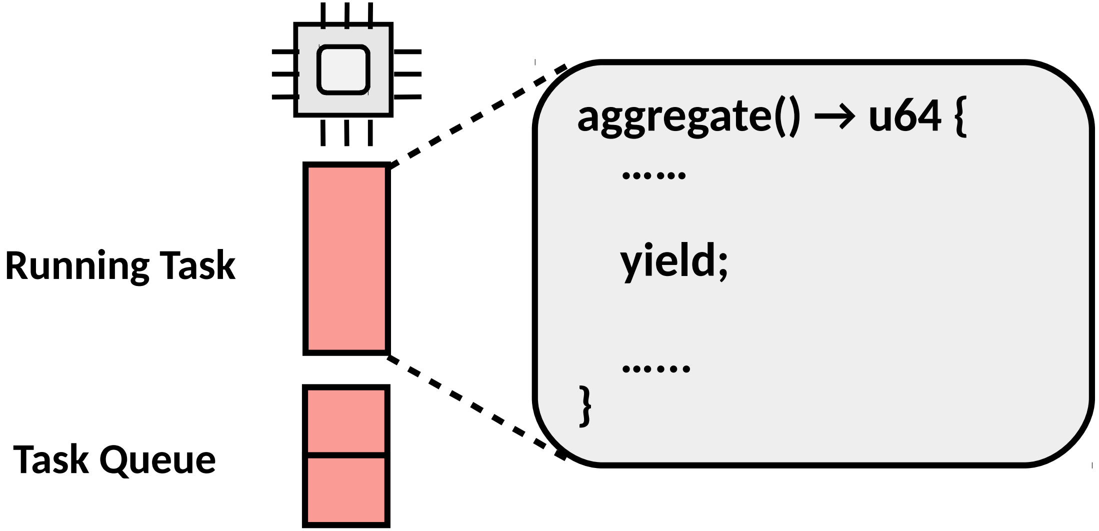


Preemption is too expensive!

Splinter: Lightweight Cooperative Scheduling

Problem: Long running tasks starve shorter tasks, hurt latency

Solution: Extensions are cooperative, must yield frequently



```
aggregate() → u64 {
    ……

    yield;

    ……
}
```

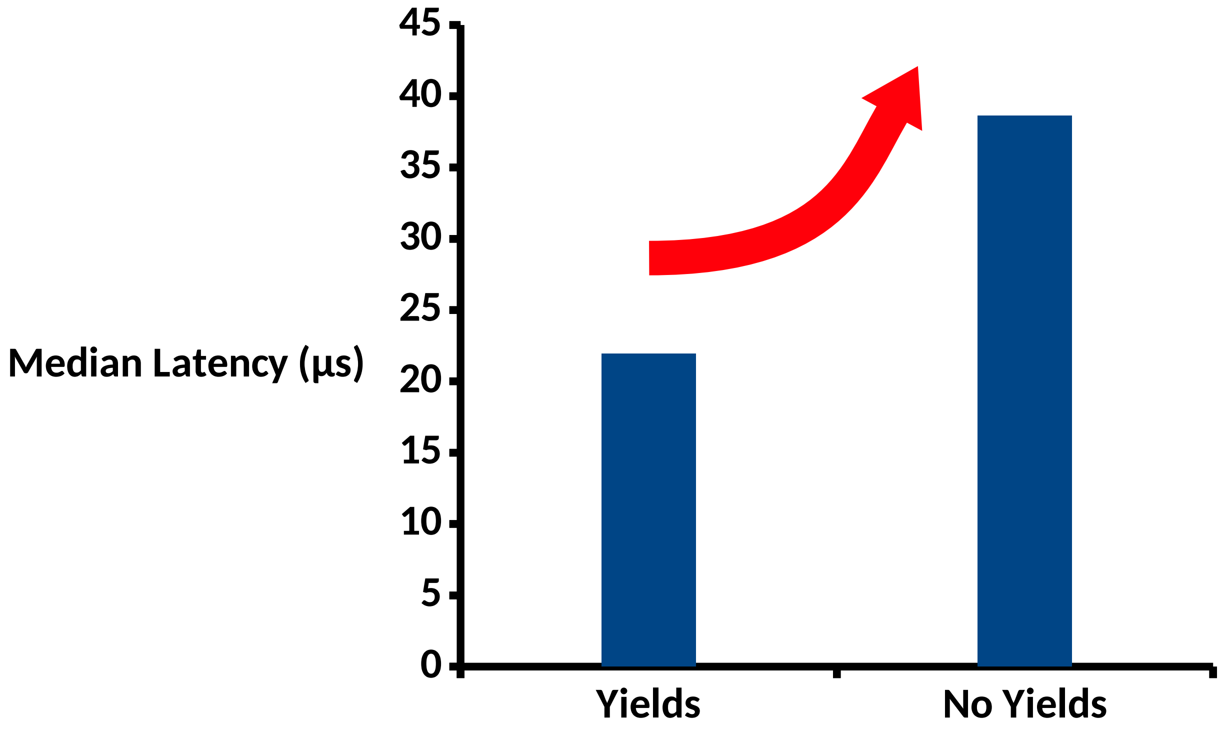
Compiler generates code to save & restore state

What are the benefits of cooperative scheduling?

Setup:

- 1024 tenants
- 85% requests invoke small extension that reads one object
- 15% requests invoke extension that reads 128 objects

Performance With And Without Yields



Offered load = 1 Mop/s

15% long running
Approaching saturation

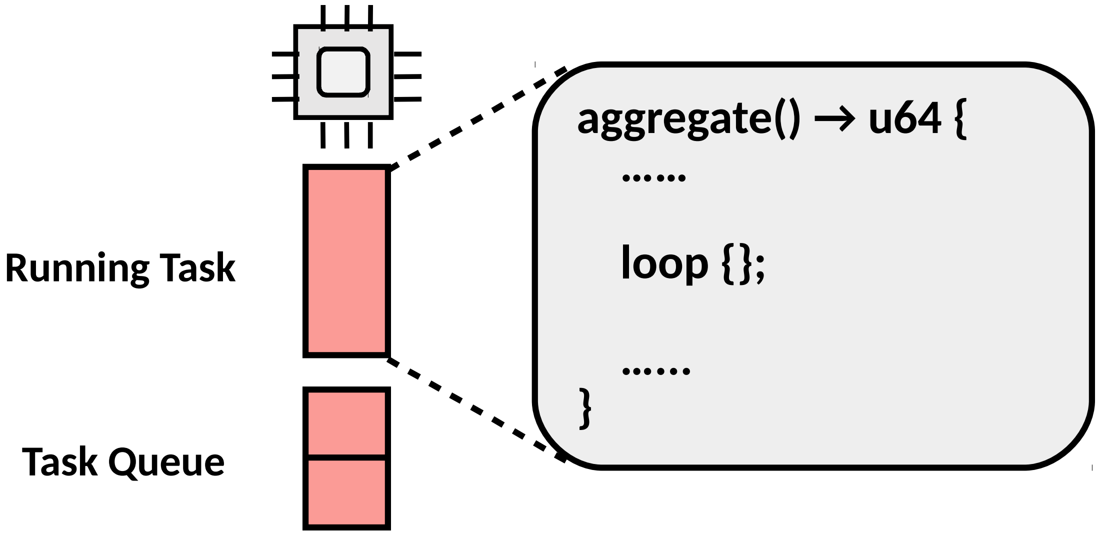
Lower is better

Yield frequently → Better Qos, Less interference

Splinter: Lightweight Cooperative Scheduling

Problem: **Uncooperative extensions**

Solution: **Trusted watchdog core**

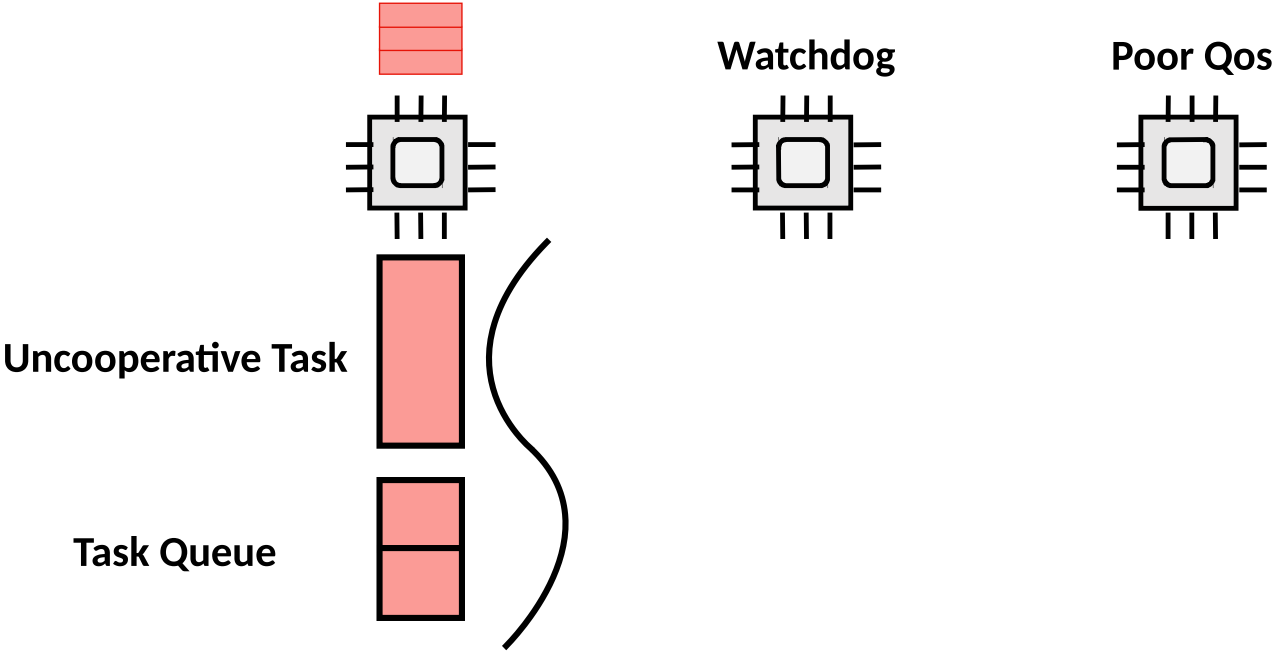


```
aggregate() → u64 {
    ......

    loop {};

    ......
}
```

Splinter: Lightweight Cooperative Scheduling

Problem: **Uncooperative extensions**

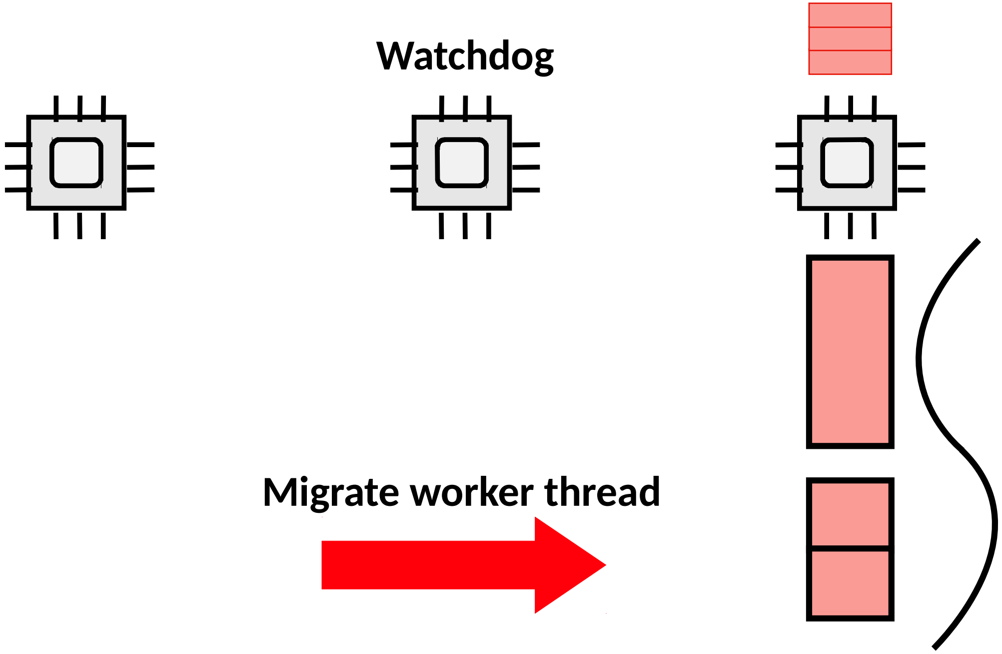
Solution: **Trusted watchdog core**



Splinter: Lightweight Cooperative Scheduling

Problem: **Uncooperative extensions**

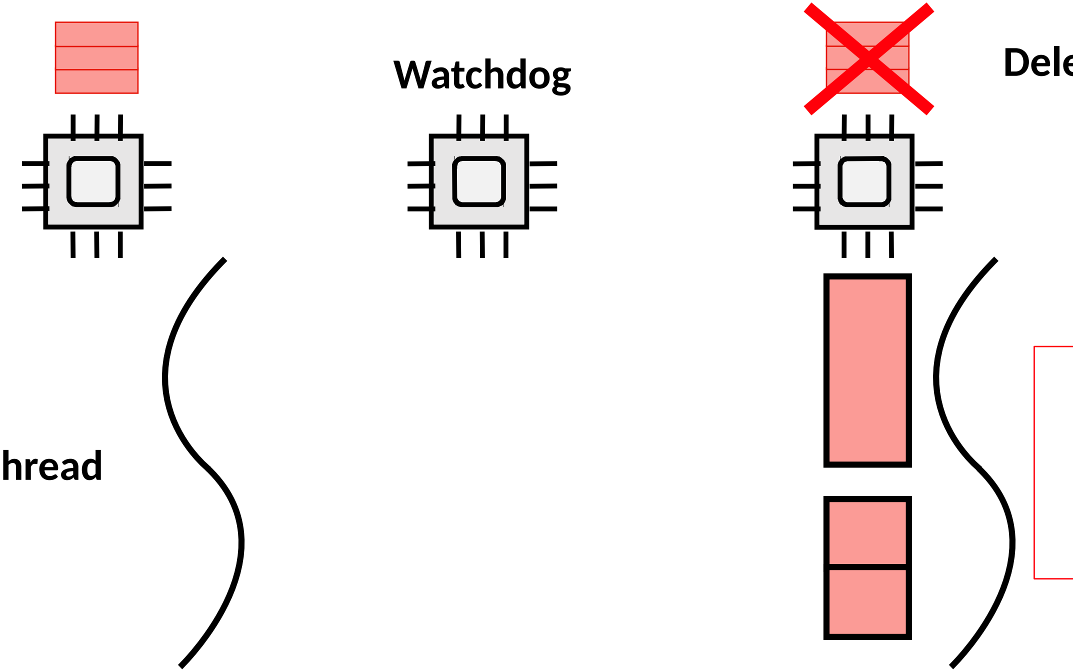
Solution: **Trusted watchdog core**



Splinter: Lightweight Cooperative Scheduling

Problem: Uncooperative extensions

Solution: Trusted watchdog core



Watchdog

Delete Rx queue

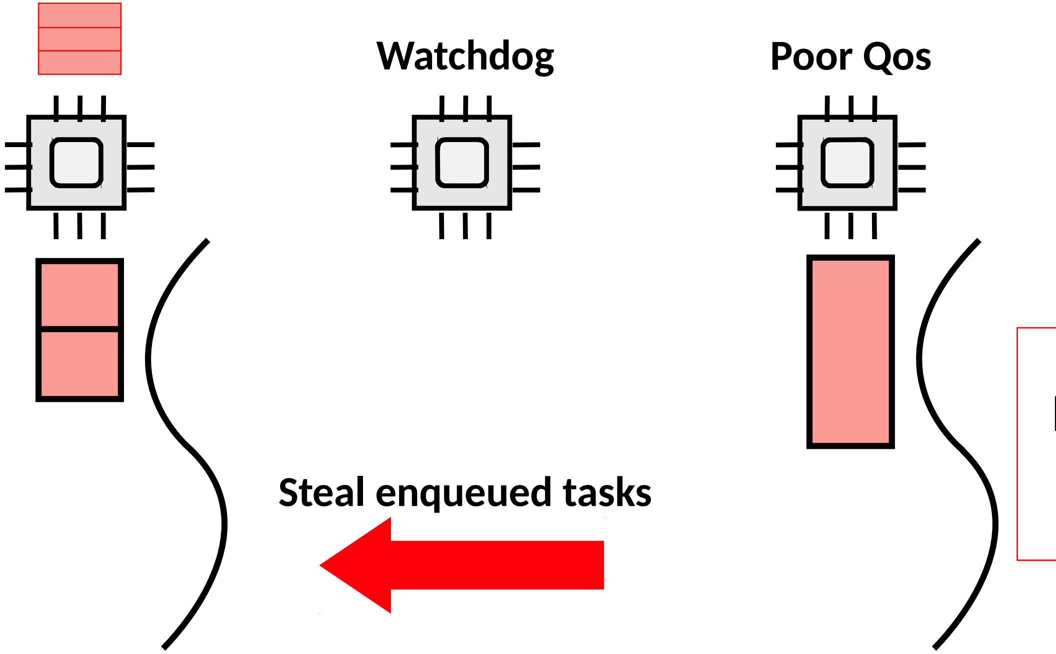
Spawn new worker thread

Extension cannot send/recv packets

Splinter: Lightweight Cooperative Scheduling

Problem: **Uncooperative extensions**

Solution: **Trusted watchdog core**



Kill worker thread when task yields

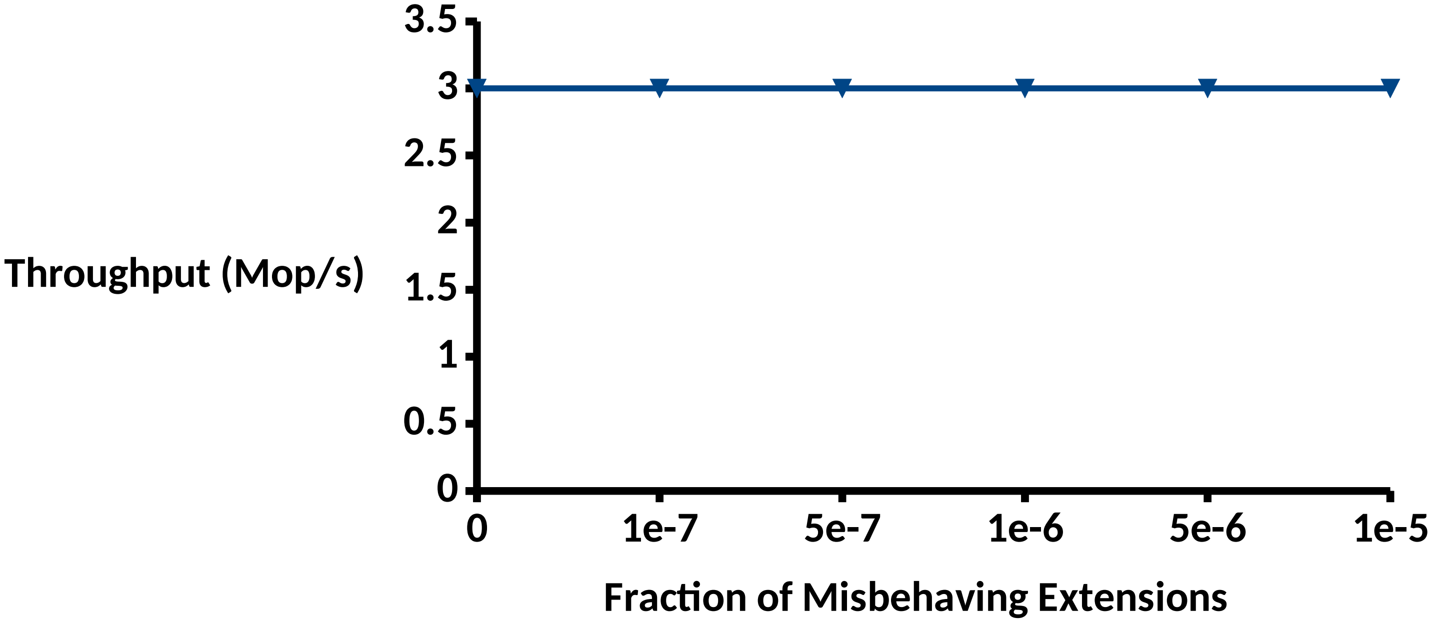
What are the benefits of the watchdog?

Setup:

- 1024 tenants
- Invoke small extension that reads one object

Performance With Misbehavior

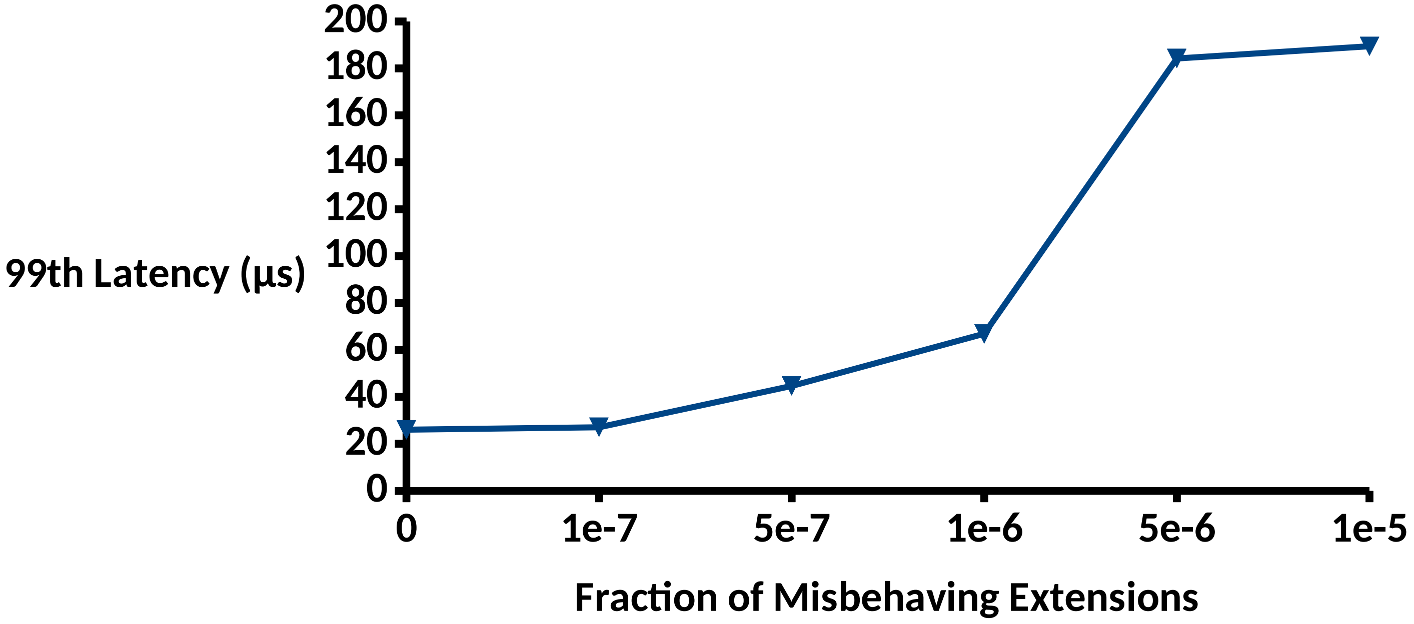
Offered load = 3 Mop/s



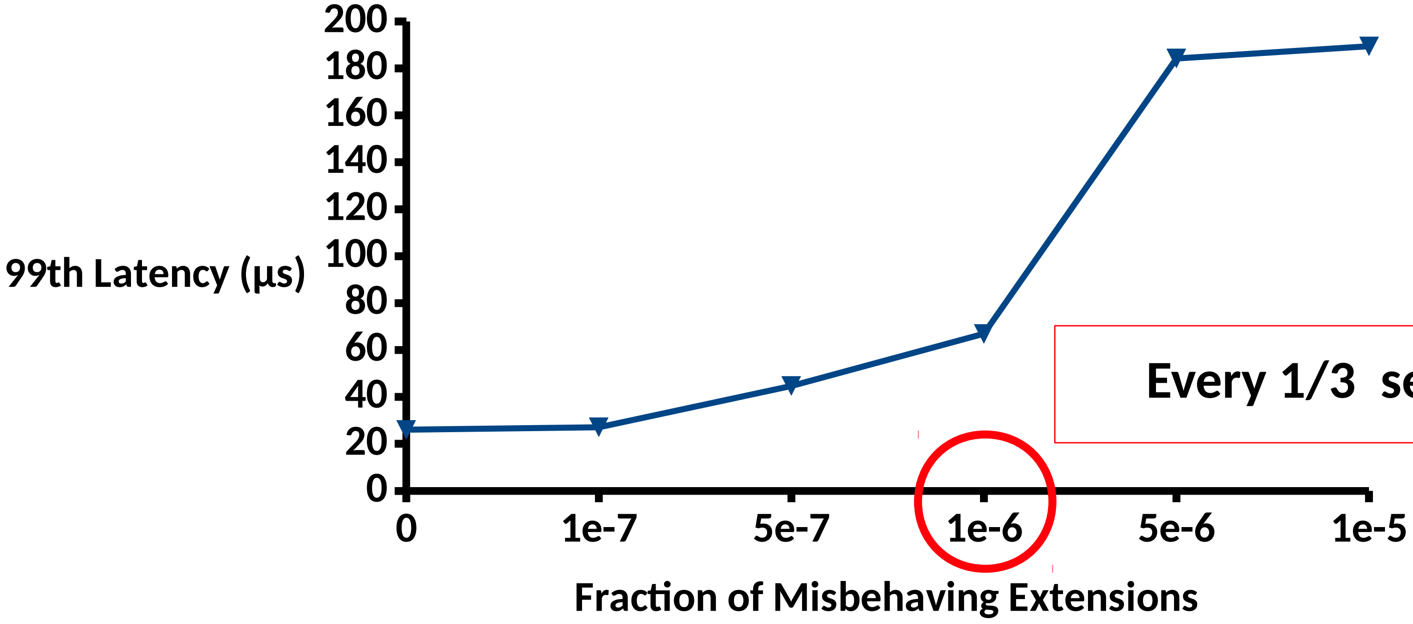
Watchdog → Maintain performance during misbehavior

Performance With Misbehavior

Offered load = 3 Mop/s



Performance With Misbehavior



Offered load = 3 Mop/s

Every 1/3 seconds

Will need tight admission control

Splinter: Design

- Tenant Locality And Work Stealing
  - Avoid cross-core coordination while avoiding hotspots

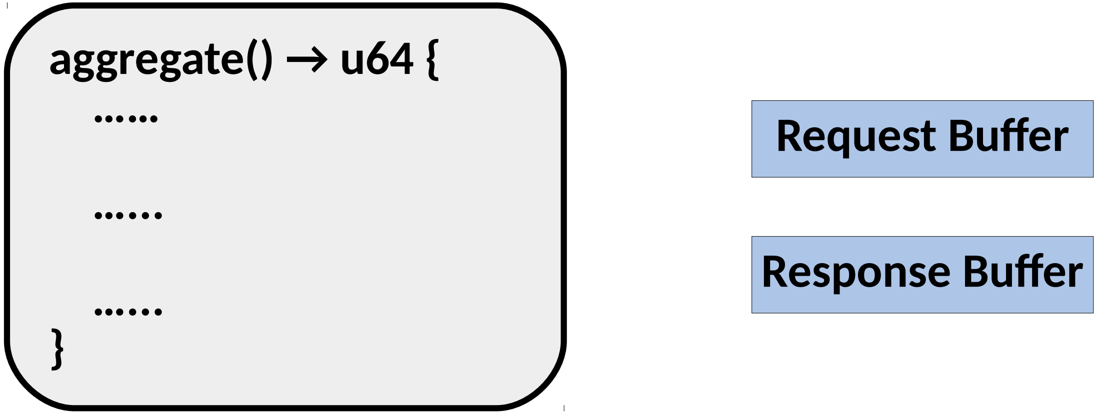
- Lightweight Cooperative Scheduling
  - Prevent long running extensions from starving short running ones

- **Low cost isolation**
  - No forced data copies across trust boundary

Splinter: Low Cost Isolation

Problem: **No forced data copies across trust boundary**

Solution: **Ensure buffers outlast reference lifetime**



```
aggregate() → u64 {
    ……

    ……

    ……
}
```

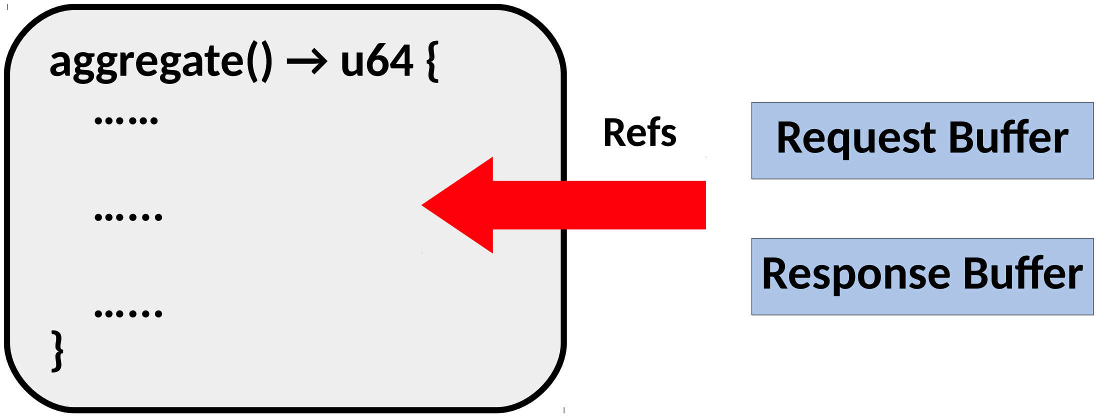
Request Buffer

Response Buffer

Splinter: Low Cost Isolation

Problem: **No forced data copies across trust boundary**

Solution: **Ensure buffers outlast reference lifetime**



```
aggregate() → u64 {
    ……
    ……
    ……
}
```

Refs

Request Buffer

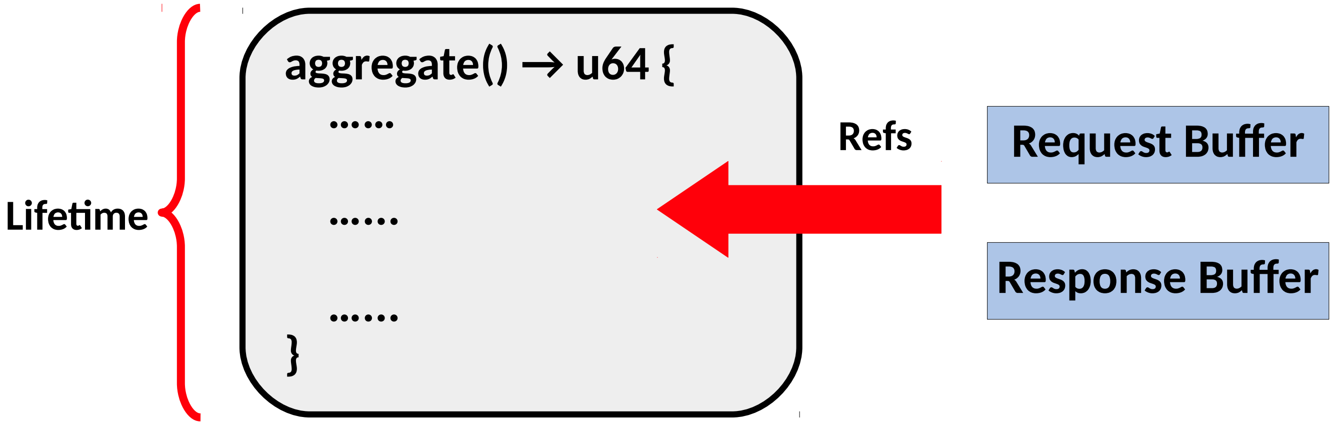
Response Buffer

Send references to extension

Splinter: Low Cost Isolation

Problem: **No forced data copies across trust boundary**

Solution: **Ensure buffers outlast reference lifetime**

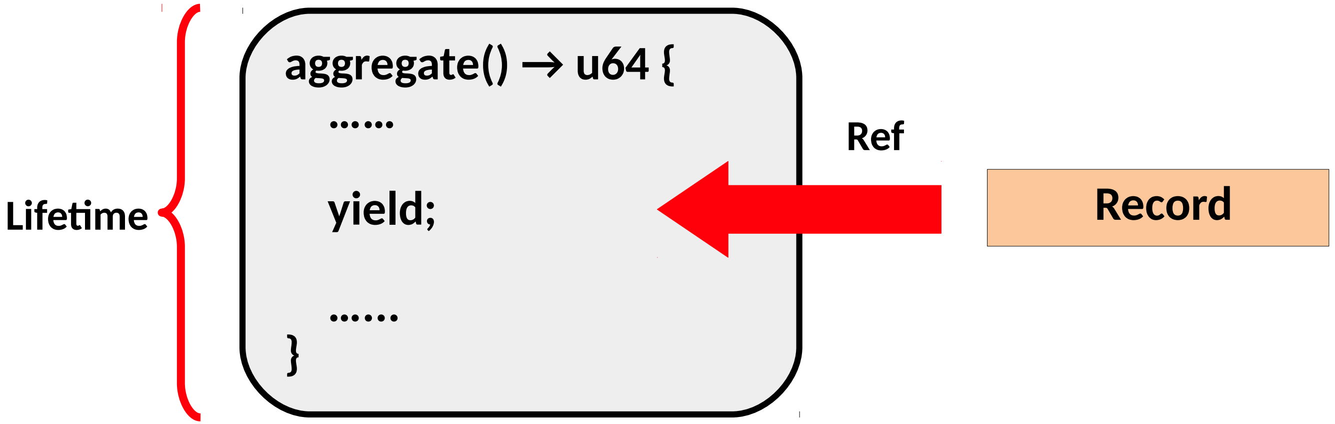


Statically ensure RPC buffers outlast lifetime

Splinter: Low Cost Isolation

Problem: **No forced data copies across trust boundary**

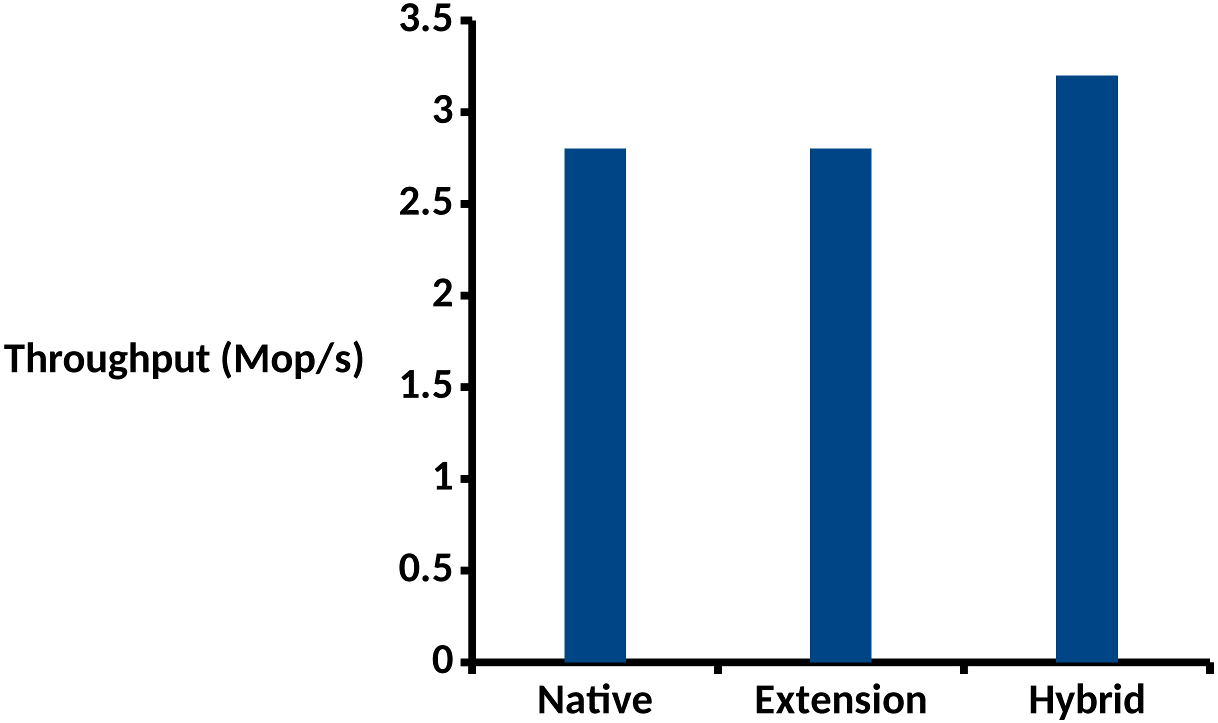
Solution: **Ensure buffers outlast reference lifetime**



Statically ensure record stays stable across yields

Refer to paper

Pushing Facebook’s TAO To Splinter



Hybrid → get() for point ops, extension for dependencies

Best of both worlds!

Related Work

- **Language isolation for kernels – SPIN, Singularity**
  - Low runtime overheads, zero-copy interface

- **Using Rust for memory safety – NetBricks, Tock**
  - Small set of static functions; does not target massive tenant densities

- **Software fault isolation**
  - Requires data copies, page table manipulation

- **Pushing extensions/compute to storage – Malacology, Redis etc**
  - Extensions are usually trusted, SQL not very good for ADTs

Conclusion

- **Kernel-bypass** key-value stores offer **< 10µs** latency, **> Mops/s** throughput
  - Fast because they're just dumb?

- **Problem:** Leverage performance → **share** between tenants
- **Problem:** Apps require rich data models. Ex: Facebook's TAO
  - Implement using gets & puts? → **Data movement, client stalls**
  - Push code to key-value store? → **Isolation costs limit density**

- **Splinter:** **Multi-tenant** key-value store that code can be pushed to
  - Tenants push type- & memory-safe code written in **Rust** at runtime
  - **> 1000** tenants/server, **3.5 Million** ops/s, **9µs** median latency