MittOS

Supporting Millisecond Tail Tolerance with Fast Rejecting SLO-Aware OS Interface

Mingzhe Hao, Huaicheng Li, Michael Hao Tong,
Chrisma Pakha, Riza O. Suminto, Cesar A. Stuardo,
Andrew A. Chien, and Haryadi S. Gunawi

THE UNIVERSITY OF CHICAGO

CERES
Center for Unstoppable Computing

Millisecond Matters!

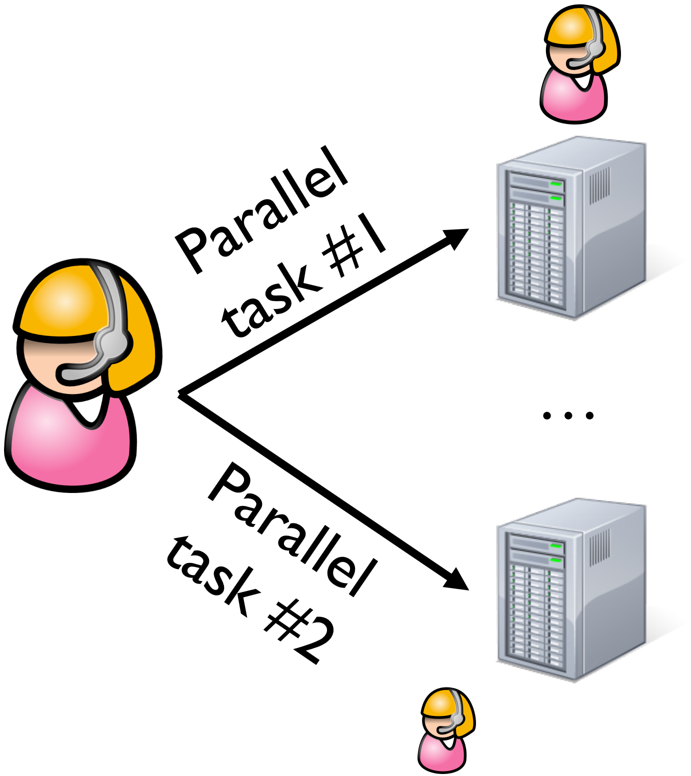
AMAZON: "EVERY 100MS OF LATENCY COSTS 1% IN SALES"

TABB GROUP: "BROKER COULD LOSE AS MUCH AS $4 MILLION IN REVENUES PER MILLISECOND IF ITS ELECTRONIC TRADING PLATFORM WAS ONLY 5MS BEHIND THE COMPETITION"

GOOGLE: "EXTRA 500MS IN SEARCH PAGE GENERATION TIME DROPPED TRAFFIC BY 20%"

Millisecond Tail Latency

Millisecond Tail Latency

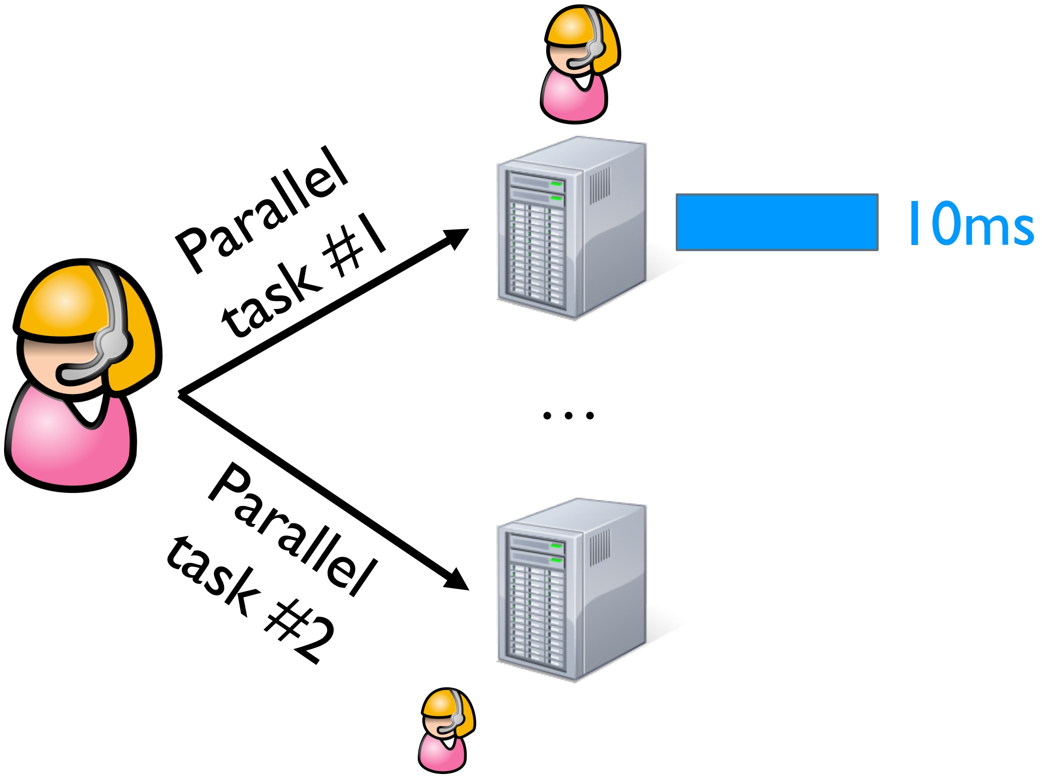


Parallel task #1

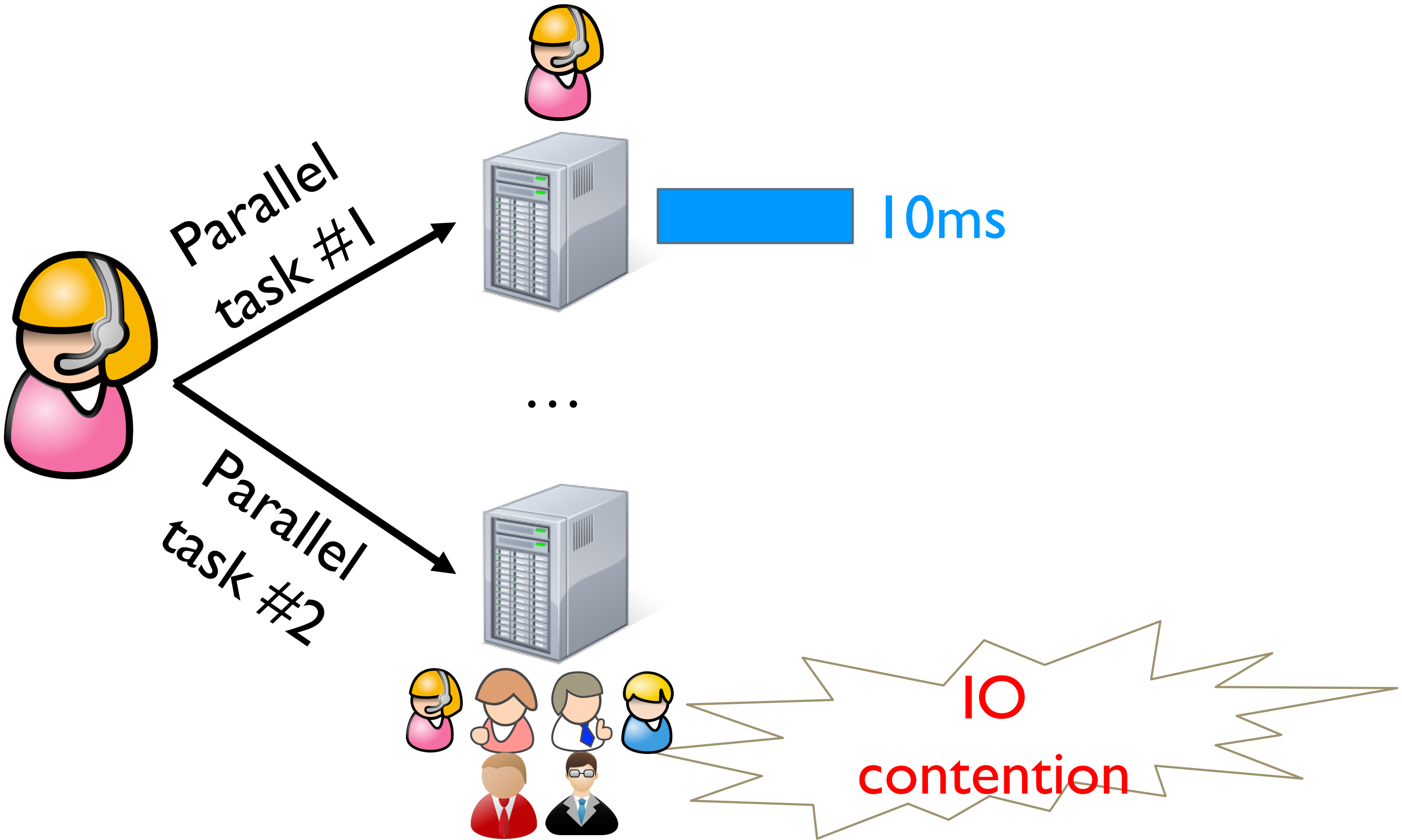
…

Parallel task #2

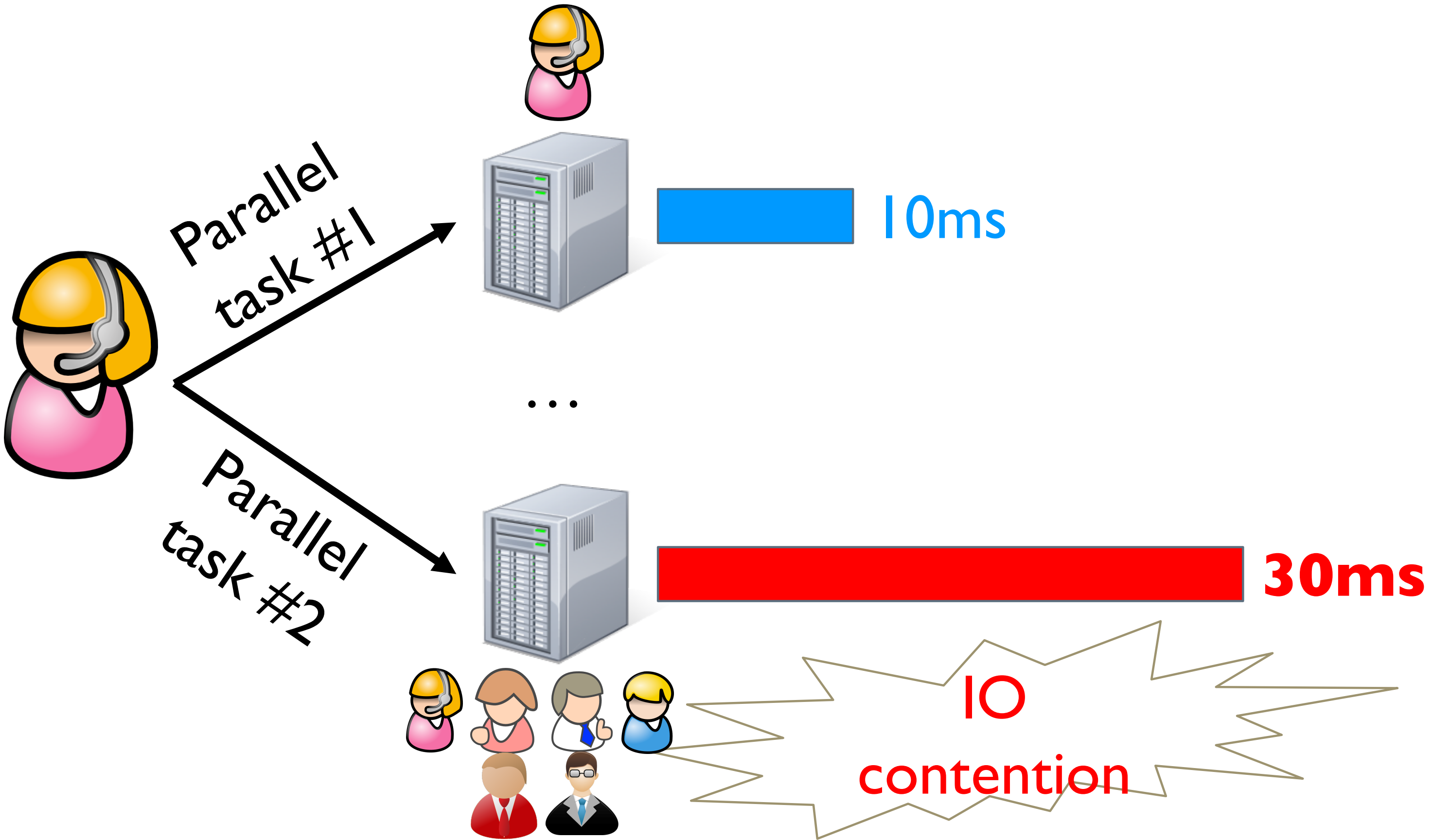
Millisecond Tail Latency



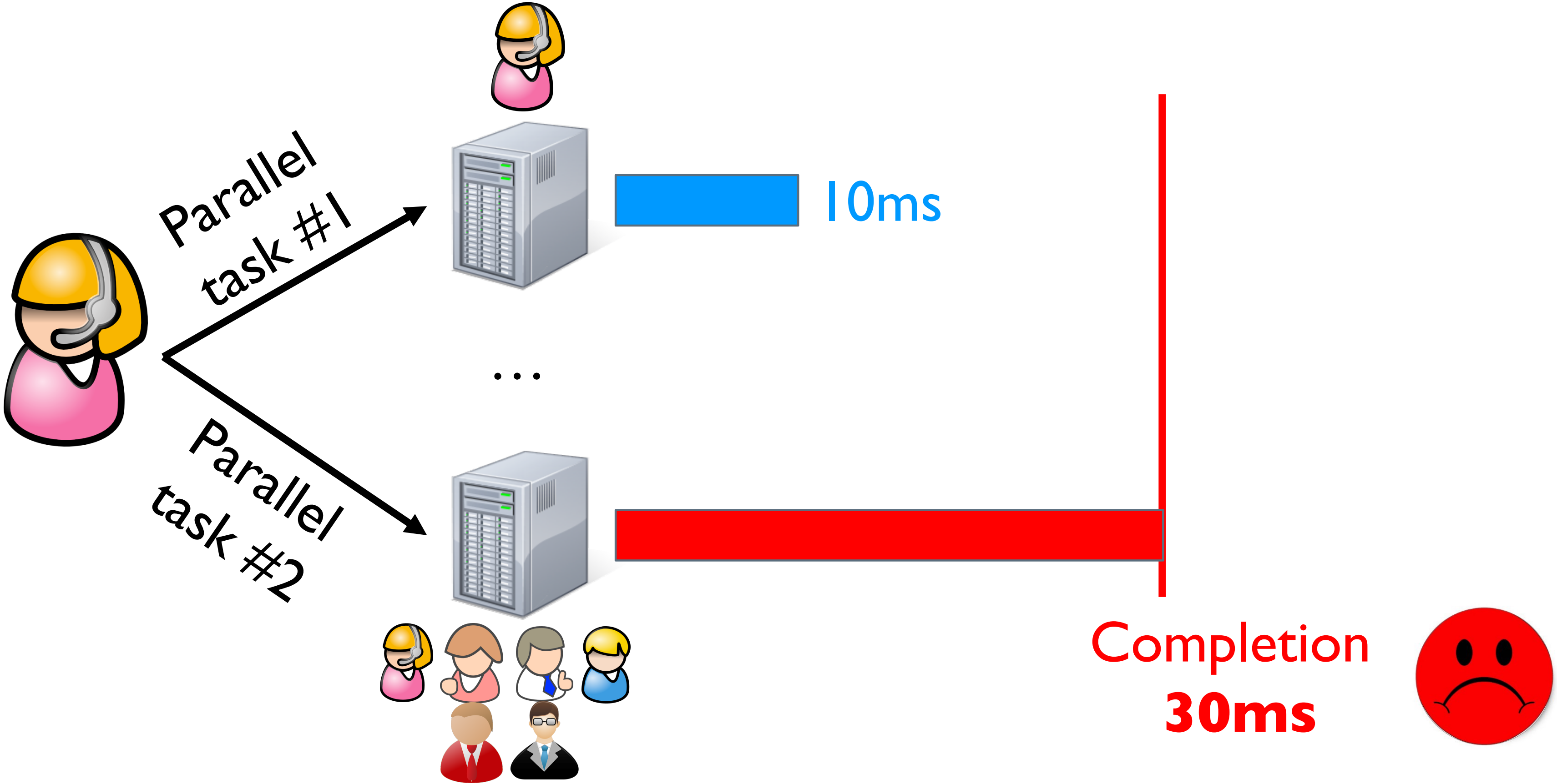
Millisecond Tail Latency



Millisecond Tail Latency



Millisecond Tail Latency



Parallel task #1

Parallel task #2

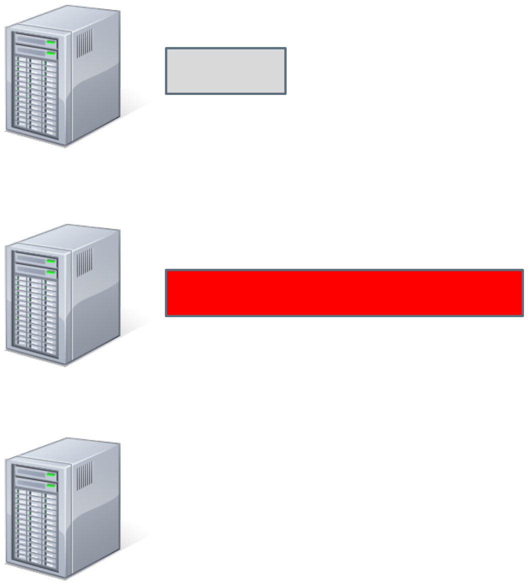
10ms

…

Completion
30ms

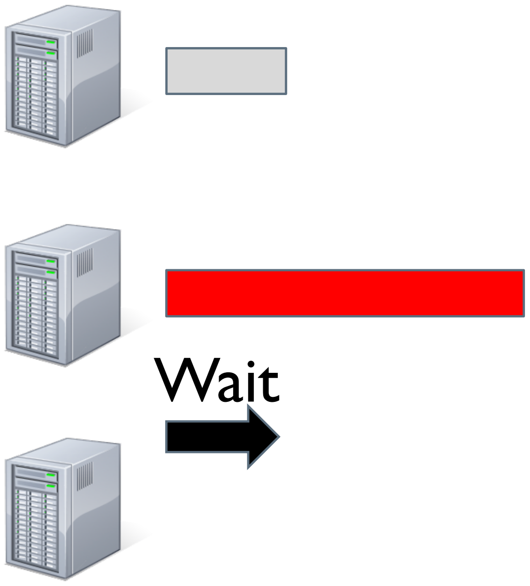
Current Tail-Tolerant Mechanisms

1. Speculation



Current Tail-Tolerant Mechanisms

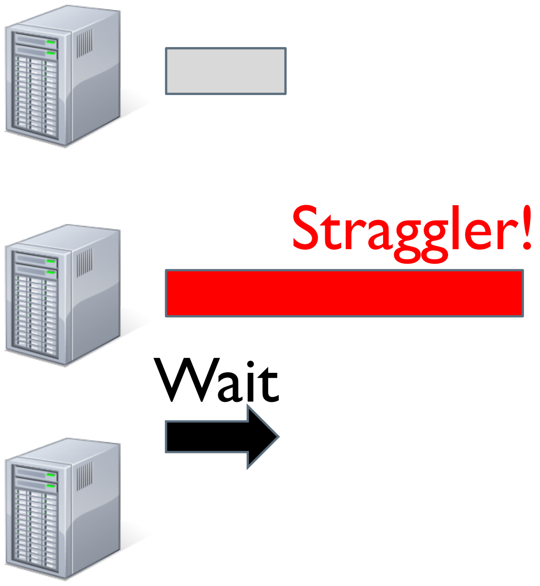
1. Speculation



Wait

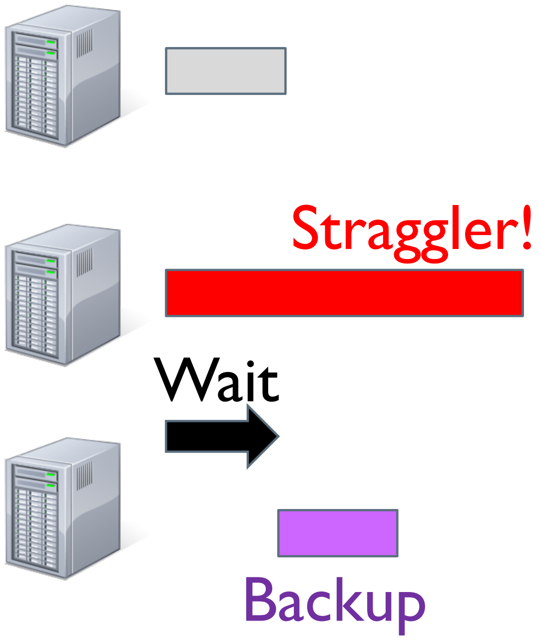
Current Tail-Tolerant Mechanisms

1. Speculation



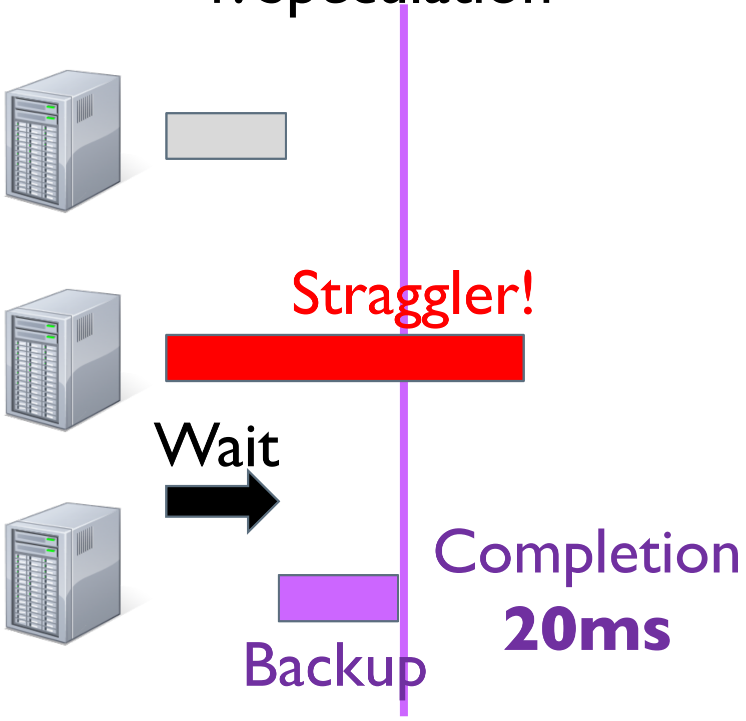
Current Tail-Tolerant Mechanisms

1. Speculation



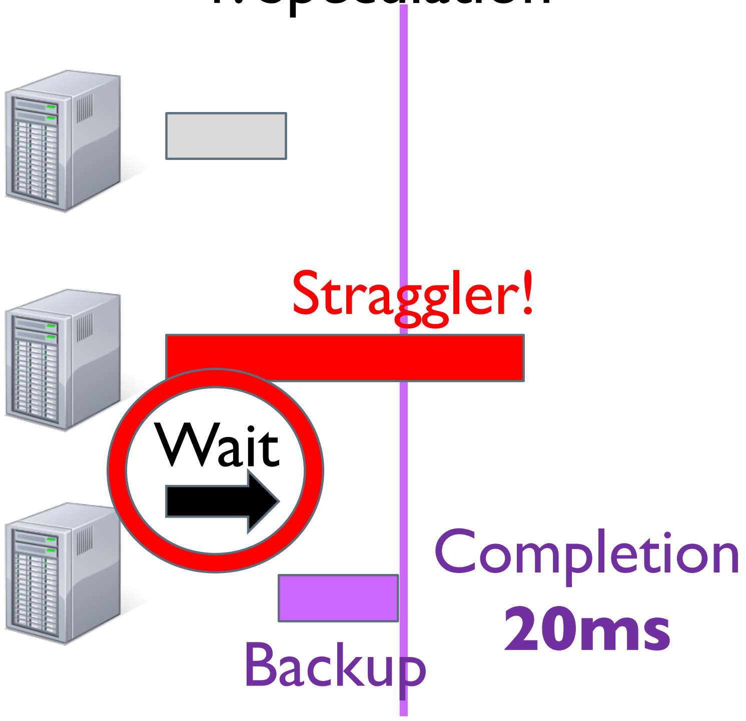
Current Tail-Tolerant Mechanisms

1. Speculation



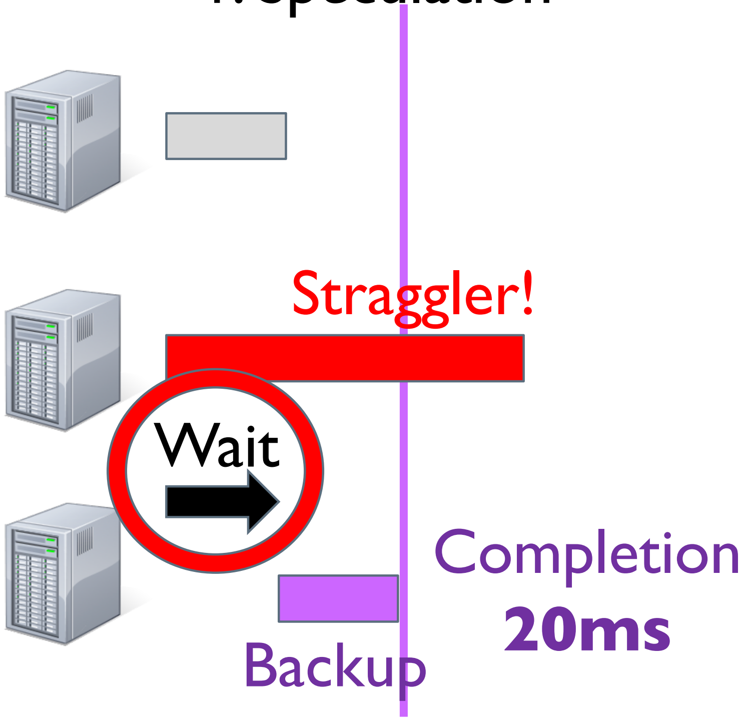
Current Tail-Tolerant Mechanisms

1. Speculation



Current Tail-Tolerant Mechanisms

1. Speculation



Straggler!

Wait

Completion

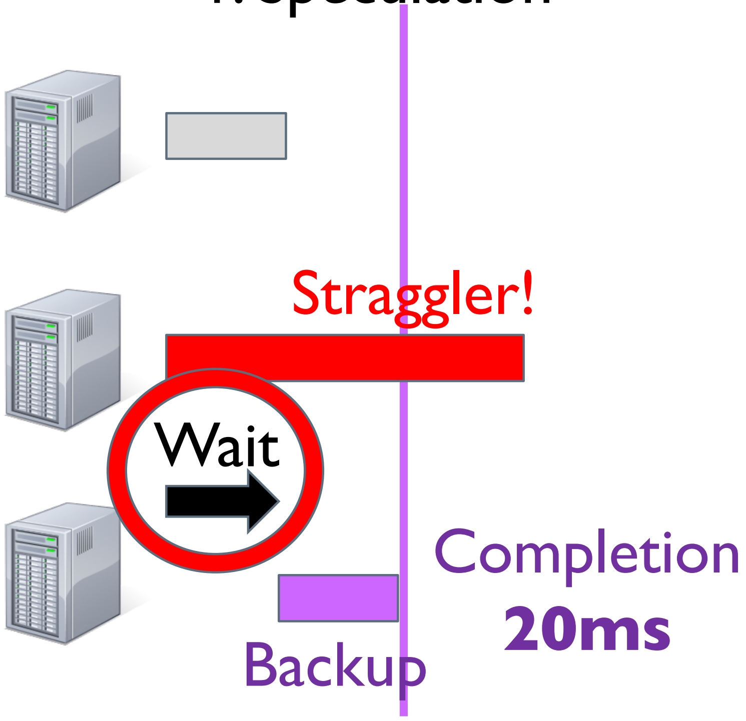
20ms

Backup

2. Cloning
   - Introduces 2x workload

Current Tail-Tolerant Mechanisms

1. Speculation

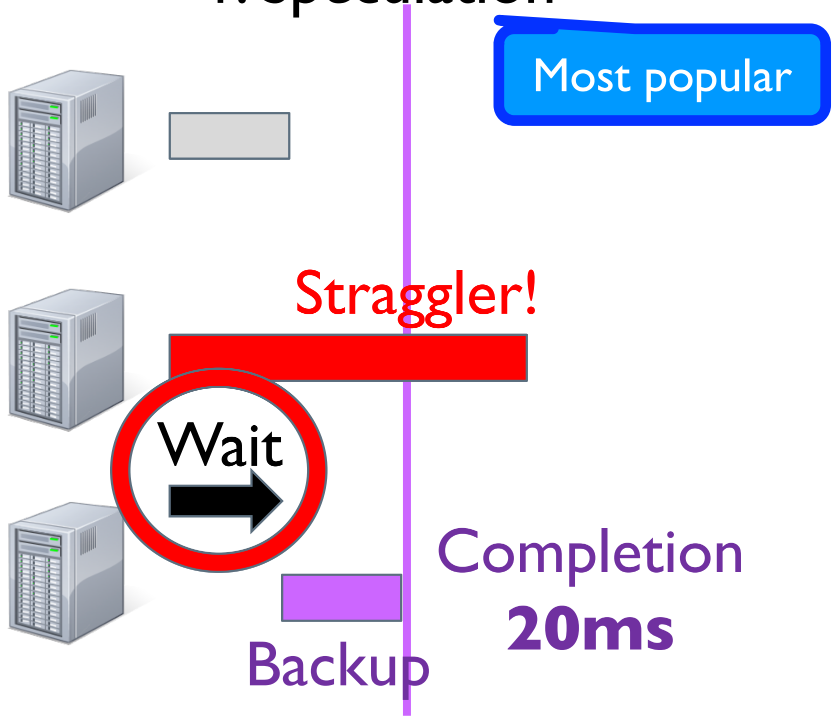


2. Cloning
   - Introduces 2x workload

3. Snitching
   - Does not work when burstiness fluctuates in ms-level

Current Tail-Tolerant Mechanisms

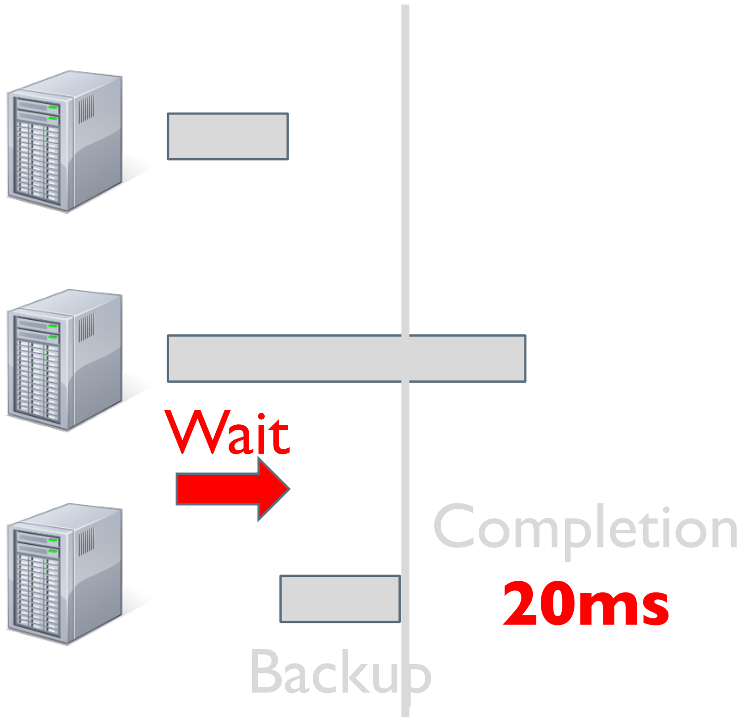
1. Speculation



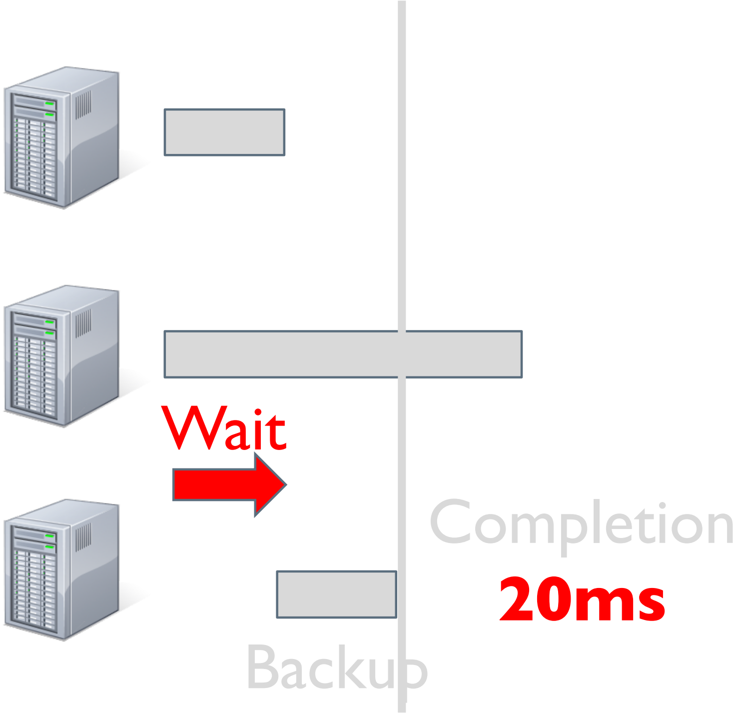
2. Cloning
   - Introduces 2x workload

3. Snitching
   - Does not work when burstiness fluctuates in ms-level

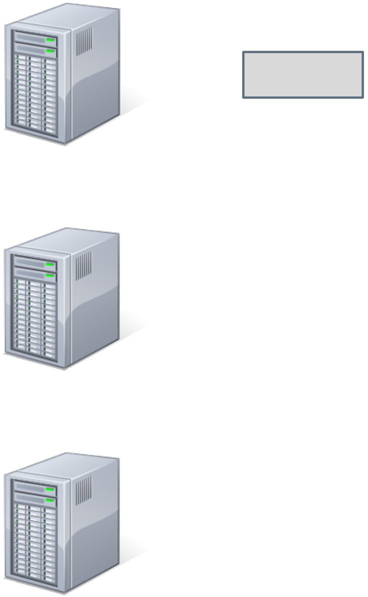
Must Wait!



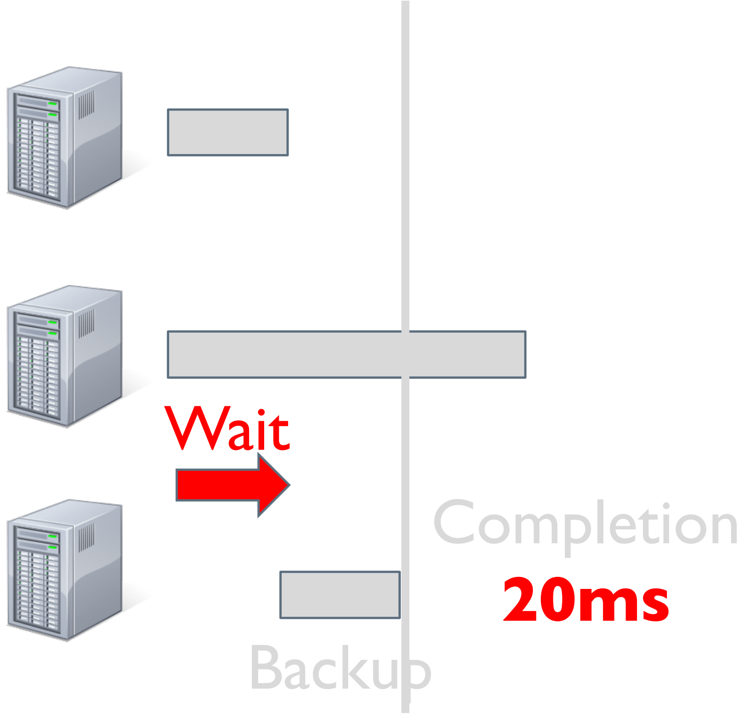
Must Wait!



No Wait ?



Must Wait!



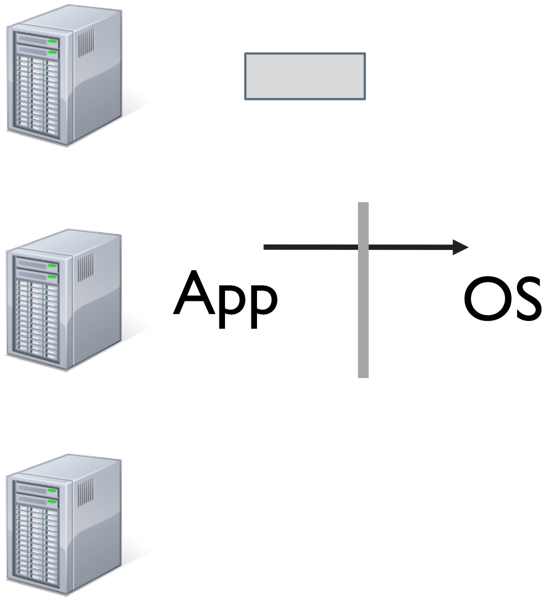
Wait

Completion

20ms

Backup

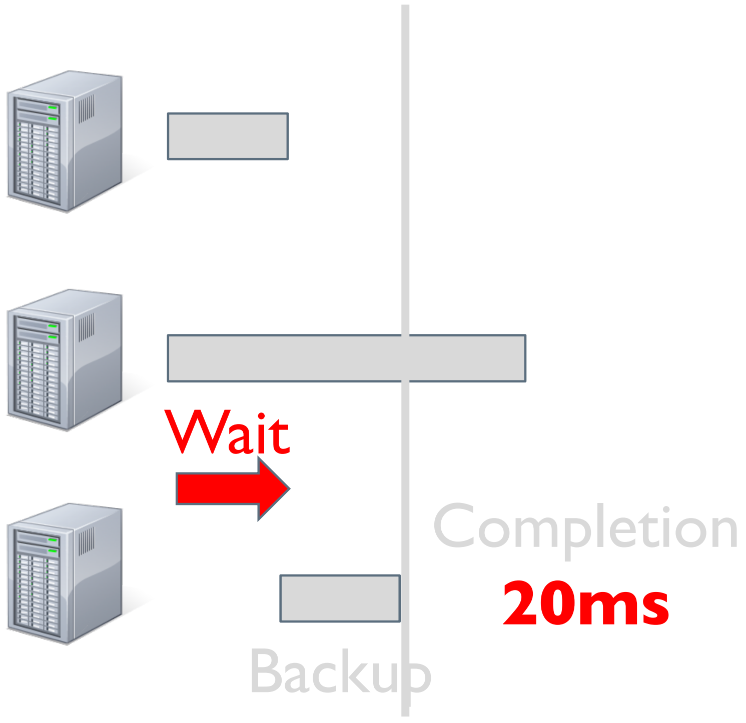
No Wait ?



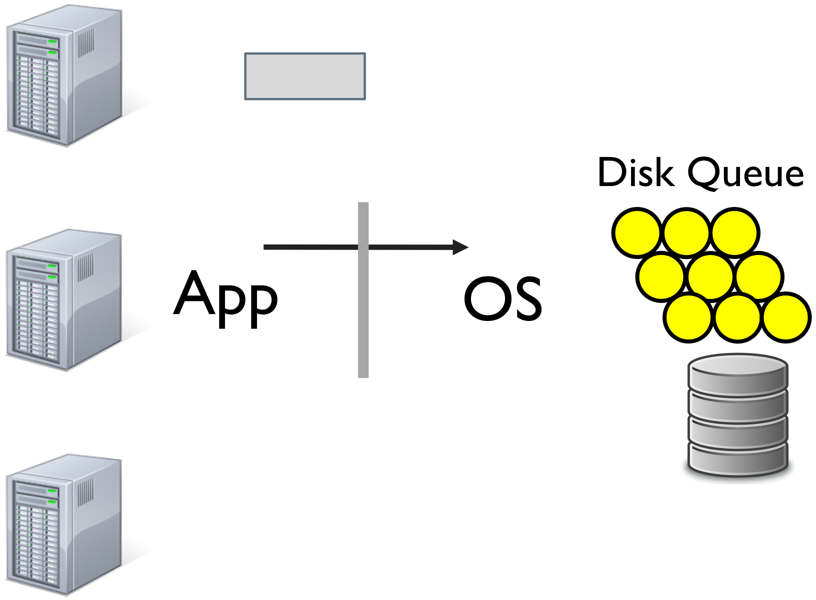
App

OS

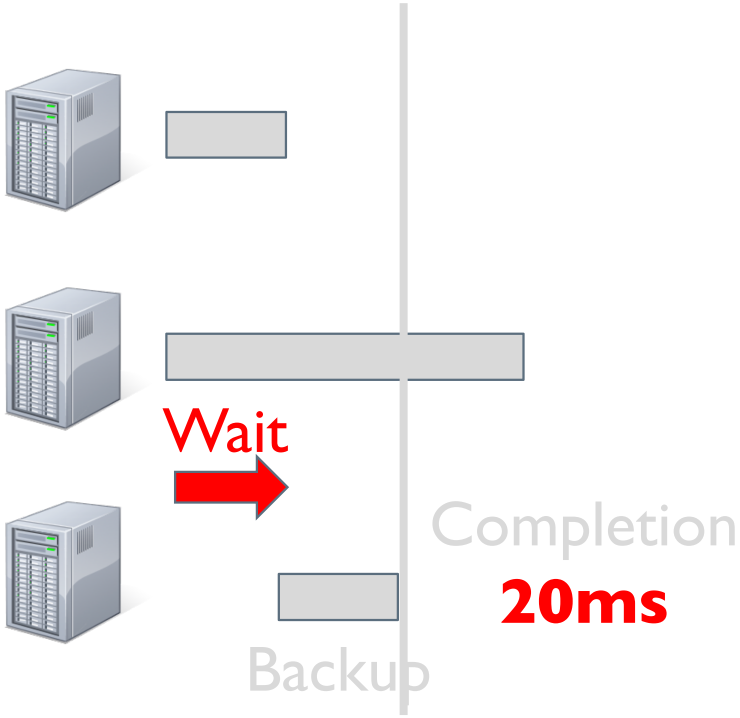
Must Wait!



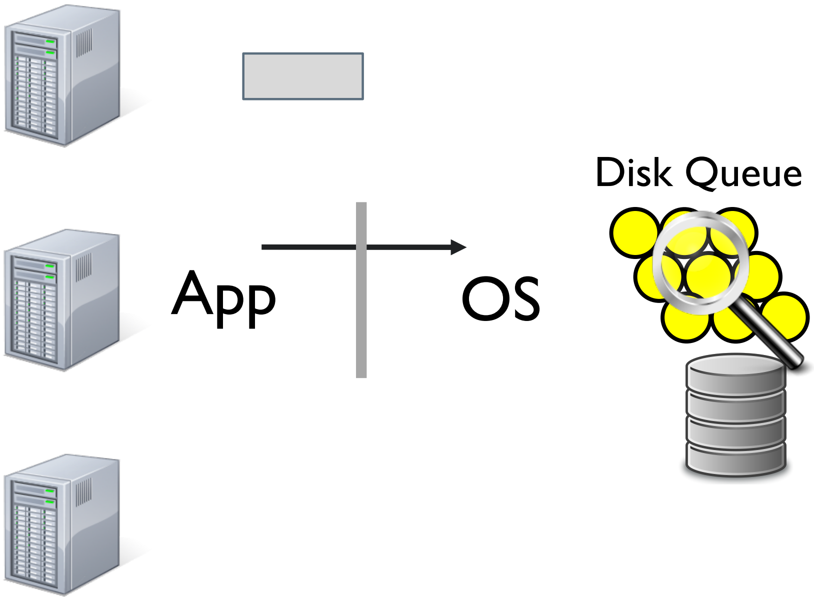
No Wait ?



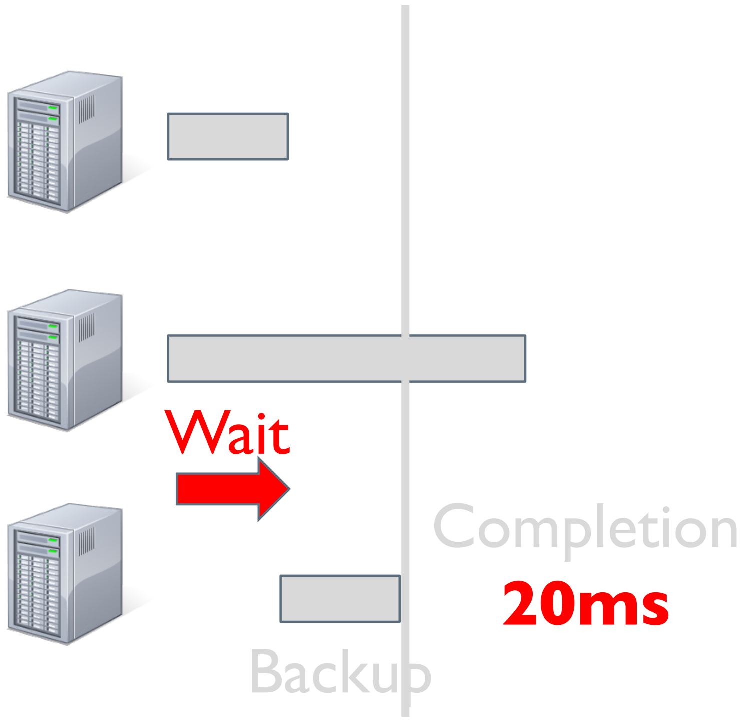
Must Wait!



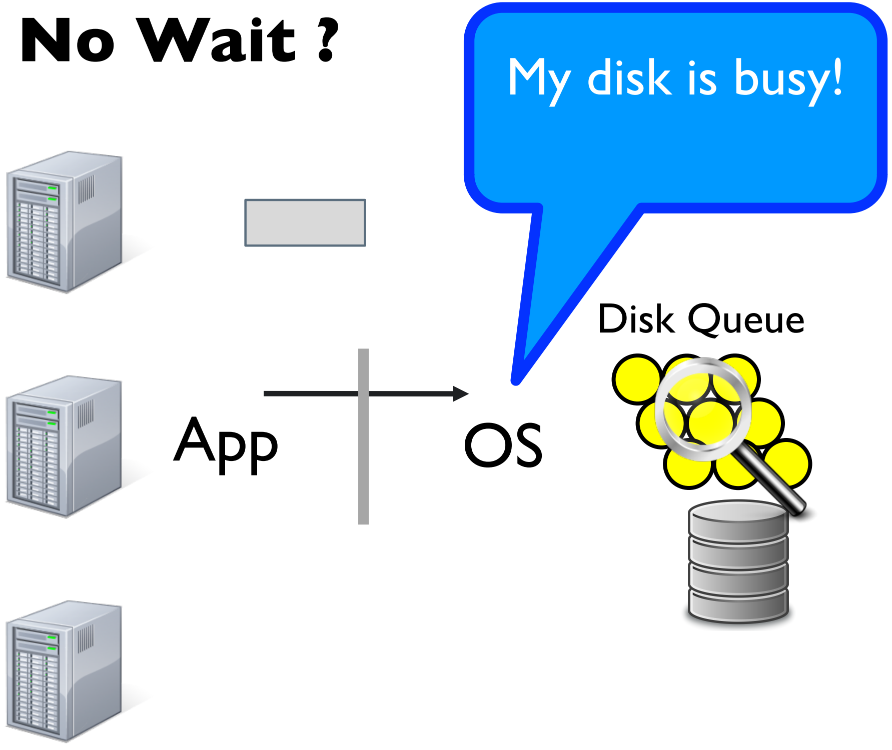
No Wait ?



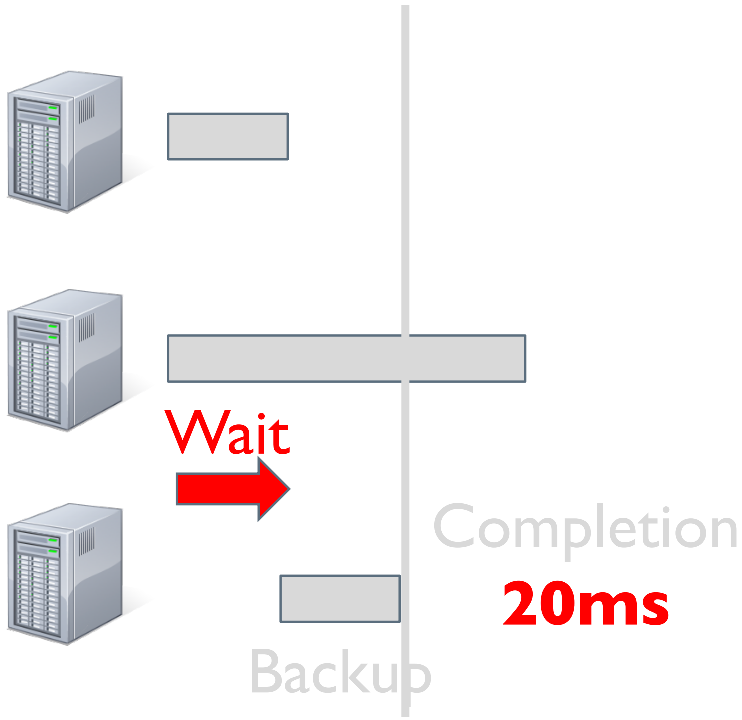
Must Wait!



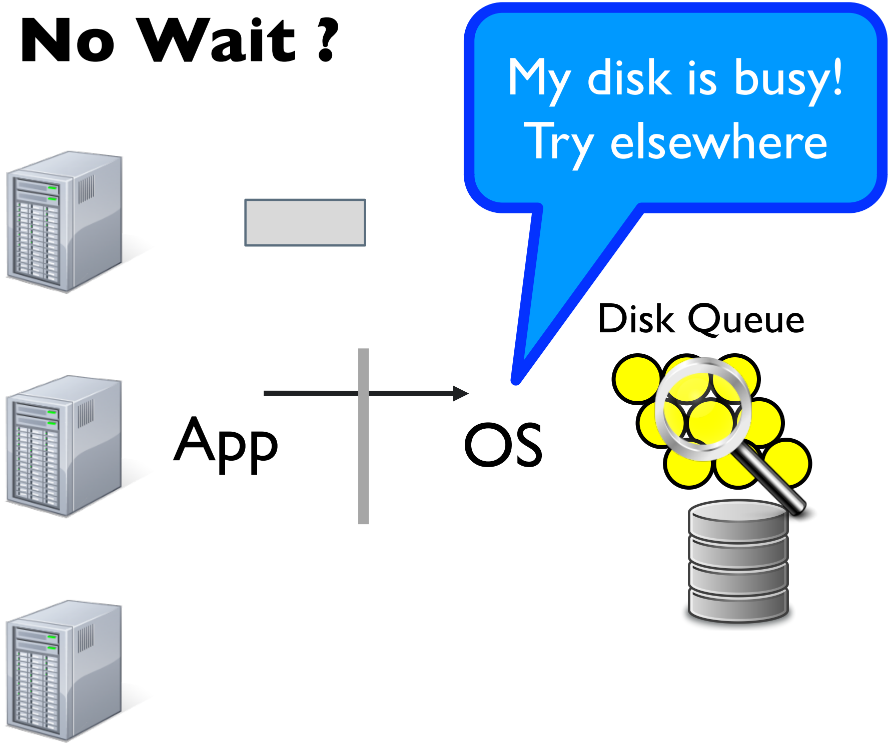
No Wait ?



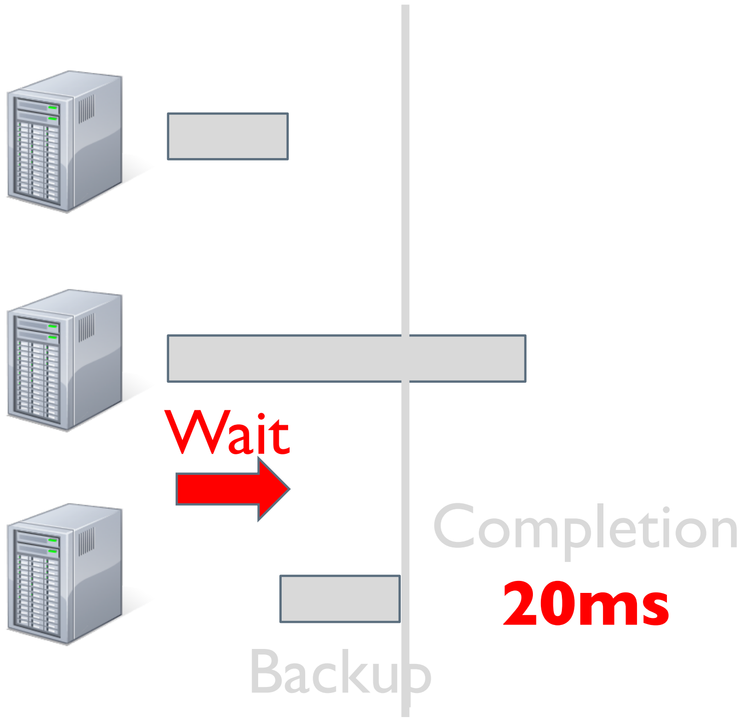
Must Wait!



No Wait ?



Must Wait!



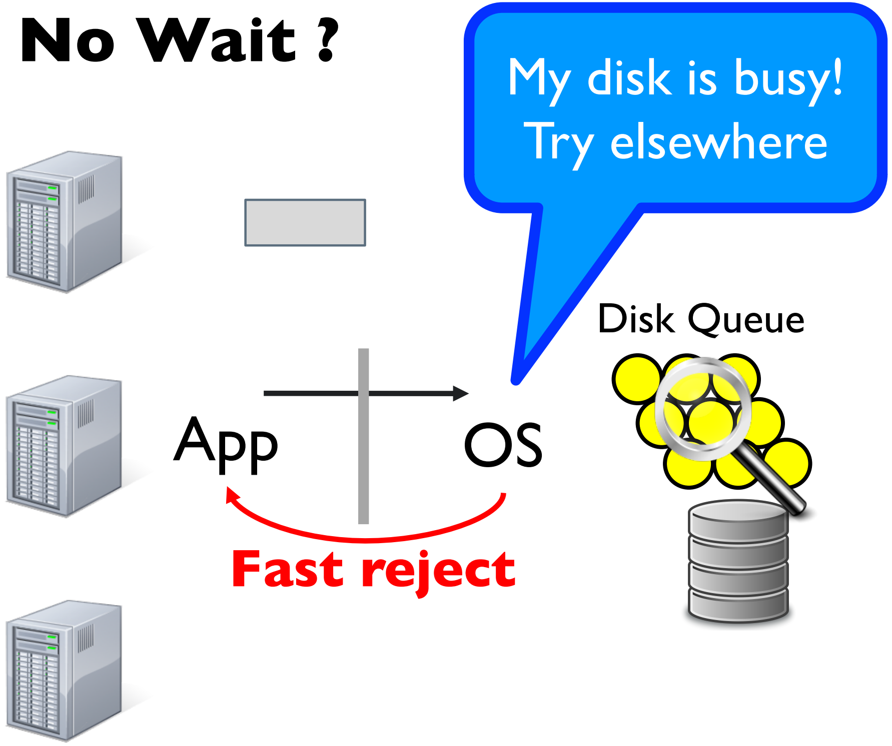
Wait

Completion

20ms

Backup

No Wait ?



My disk is busy!
Try elsewhere

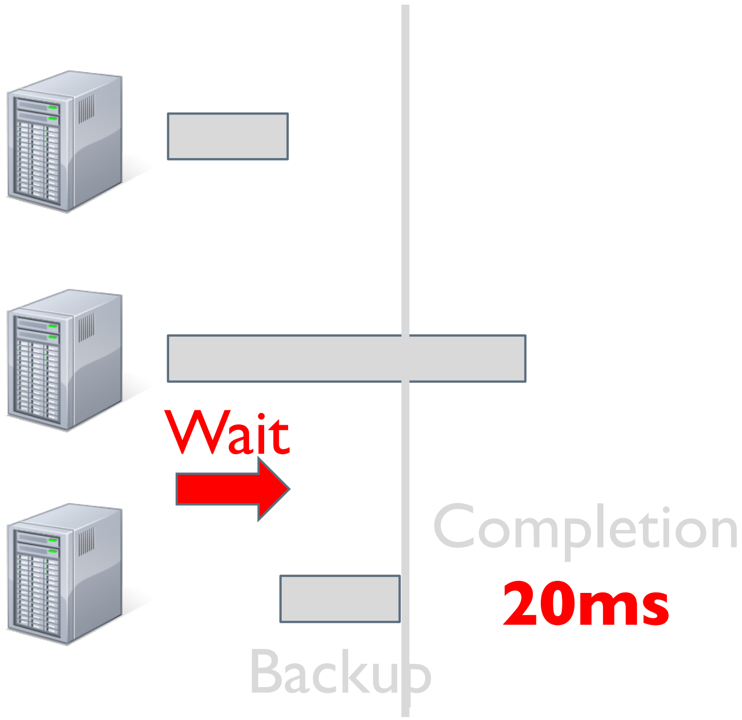
Disk Queue

App

OS

Fast reject

Must Wait!



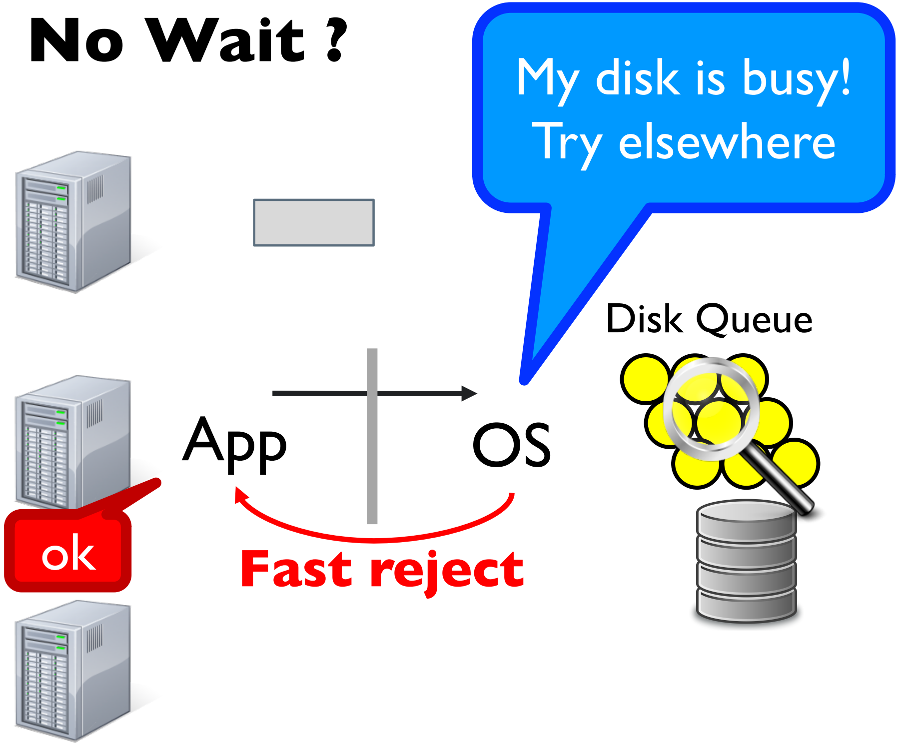
Wait

Completion

20ms

Backup

No Wait ?



My disk is busy!
Try elsewhere

Disk Queue

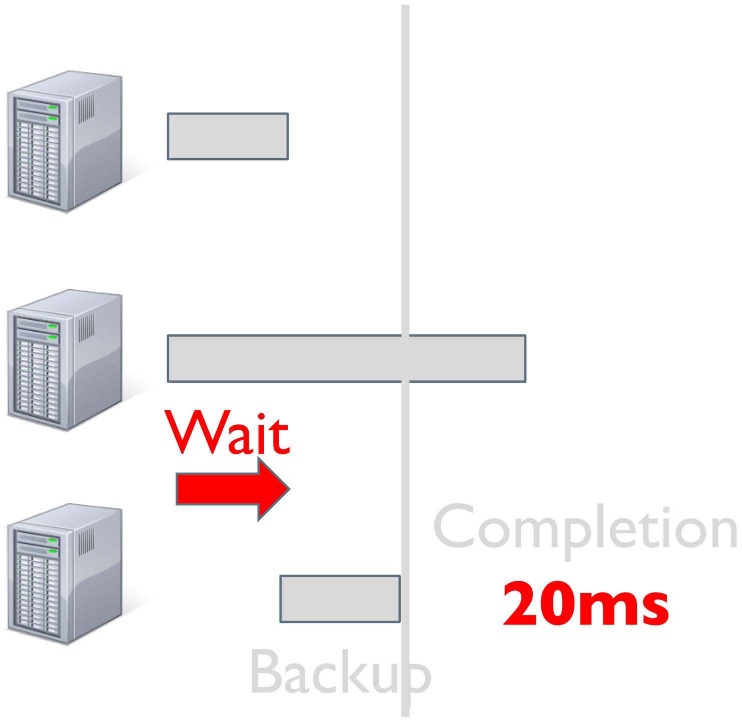
App

OS

ok

Fast reject

Must Wait!



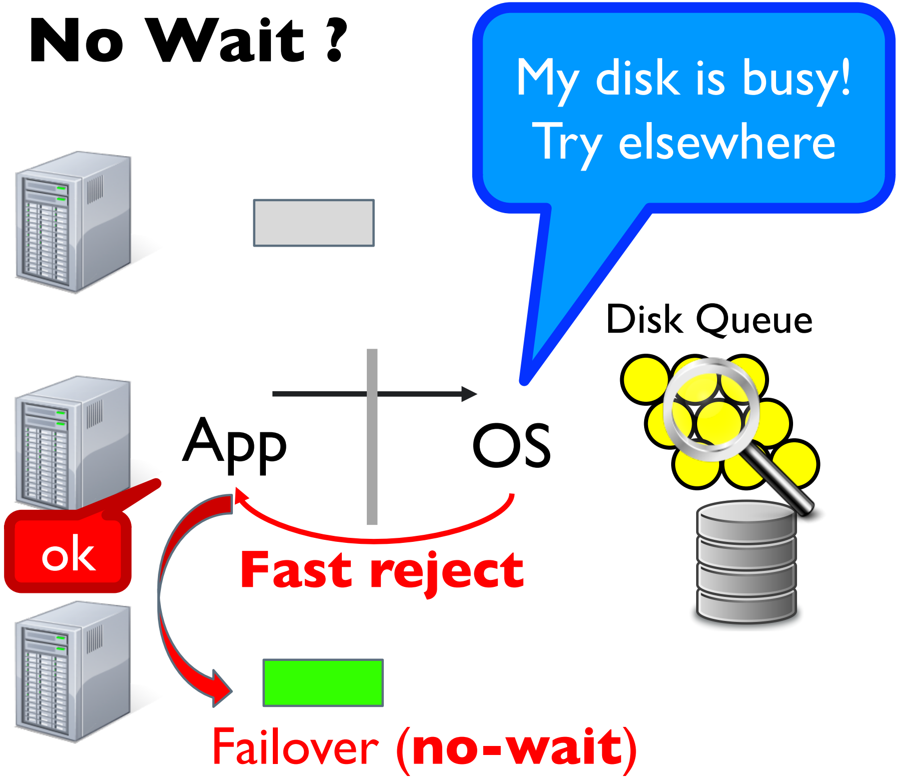
Wait

Completion

20ms

Backup

No Wait ?



My disk is busy!
Try elsewhere

Disk Queue

App

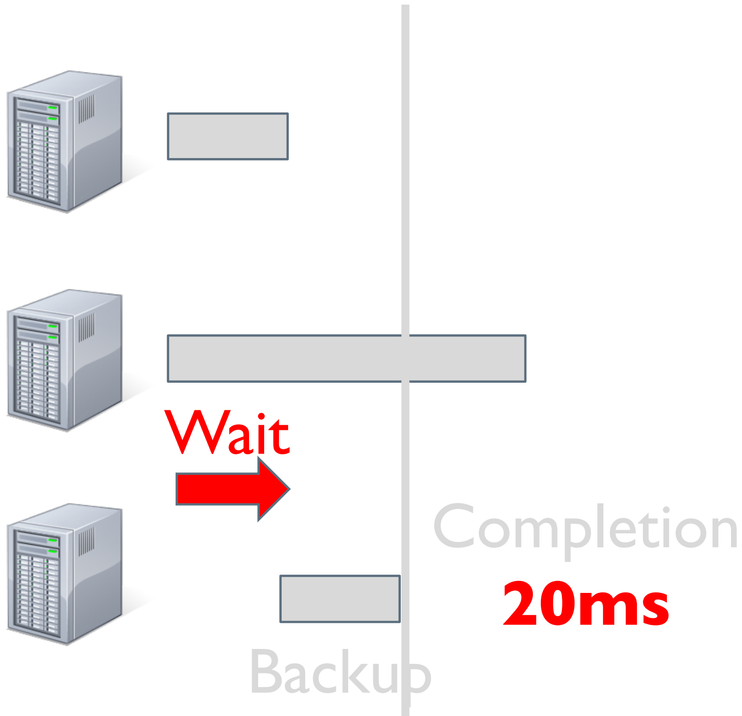
OS

ok

Fast reject

Failover (**no-wait**)

Must Wait!



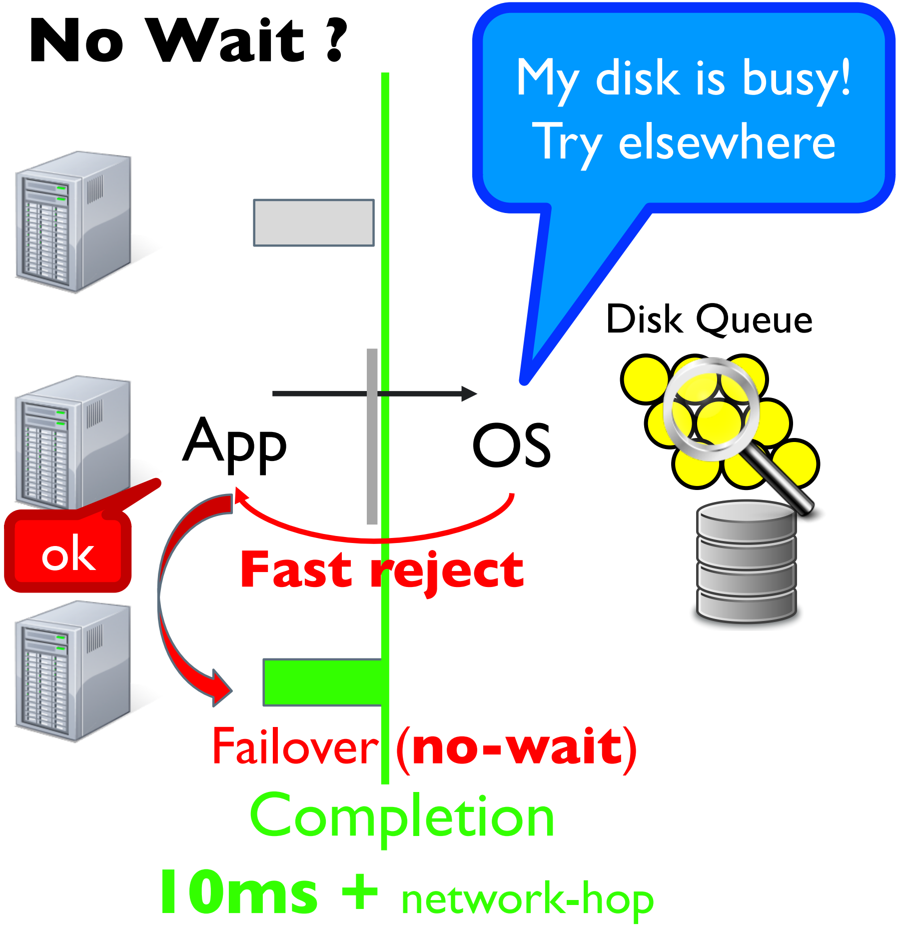
Wait

Completion

20ms

Backup

No Wait ?



My disk is busy!
Try elsewhere

Disk Queue

App

OS

ok

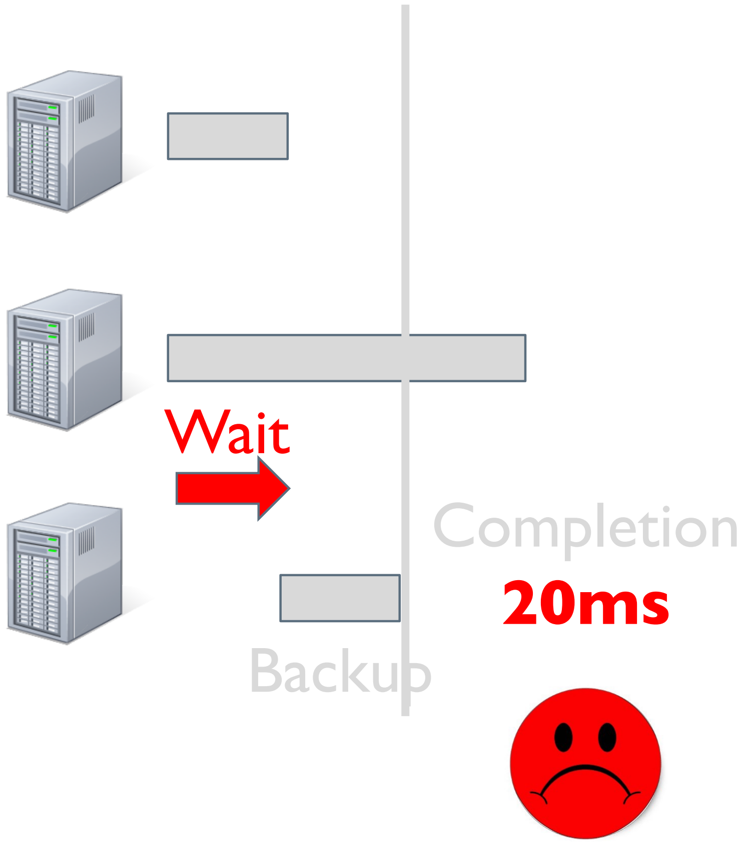
Fast reject

Failover (**no-wait**)

Completion

10ms + network-hop

Must Wait!



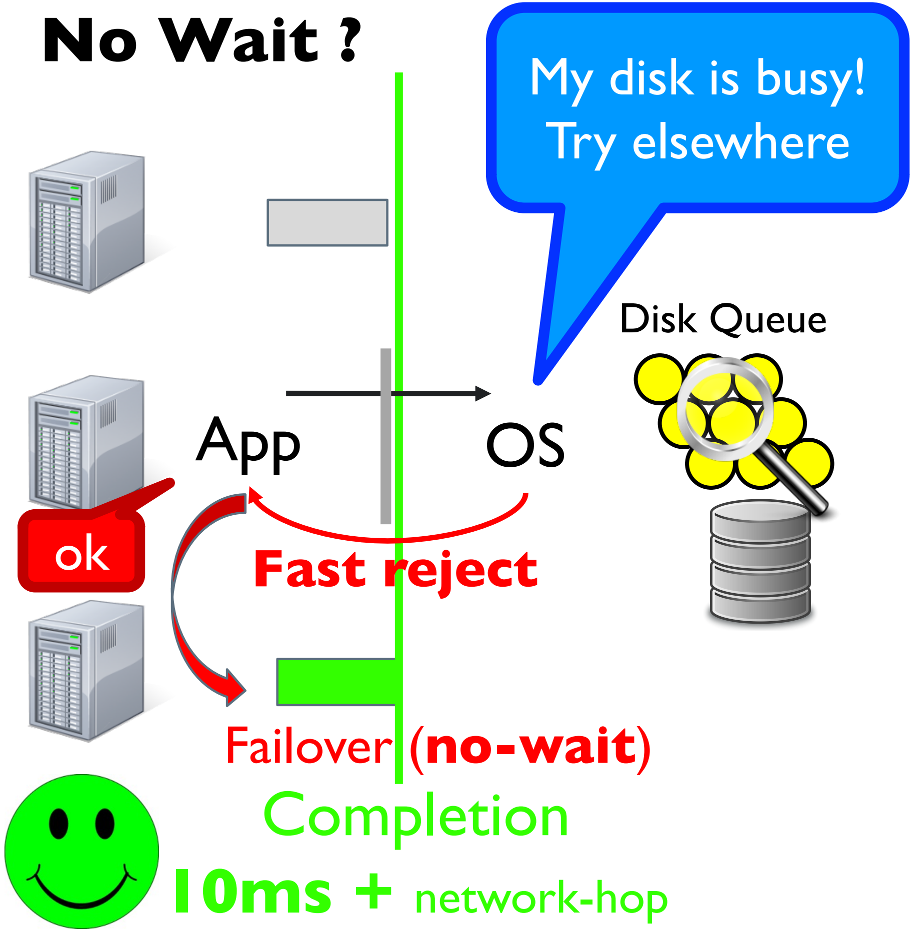
Wait

Completion

20ms

Backup

No Wait ?



My disk is busy!
Try elsewhere

Disk Queue

App

OS

ok

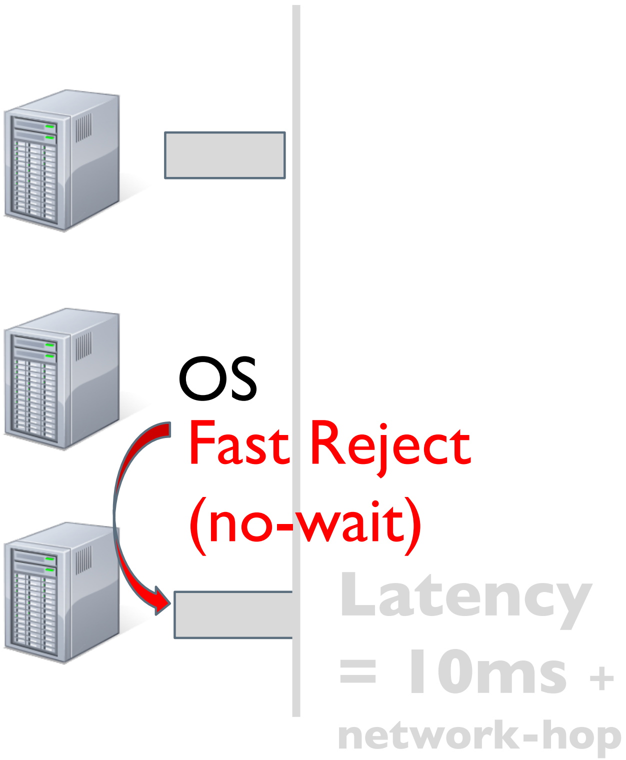
Fast reject

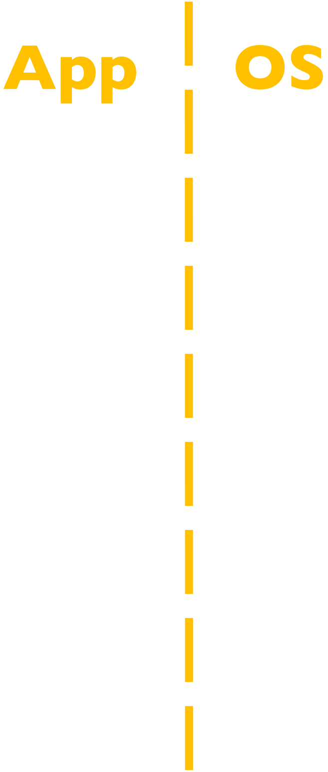
Failover (**no-wait**)

Completion

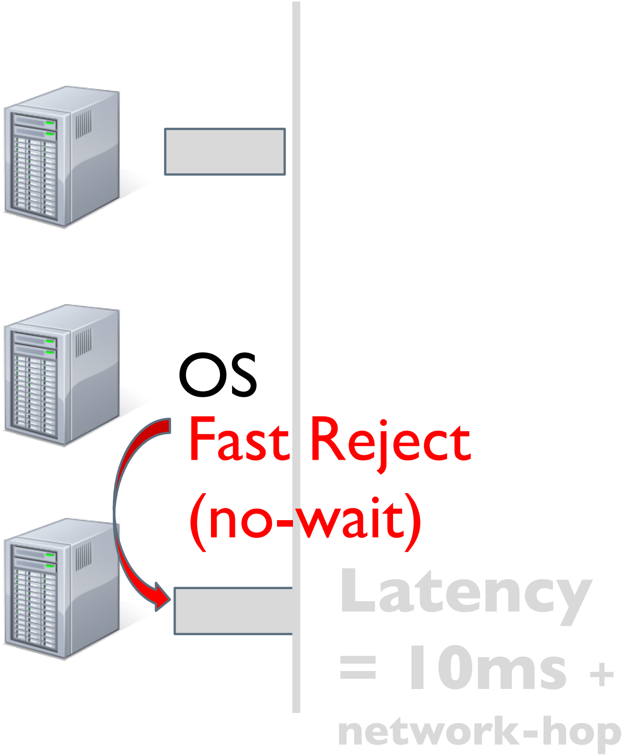
10ms + network-hop

Use-Case





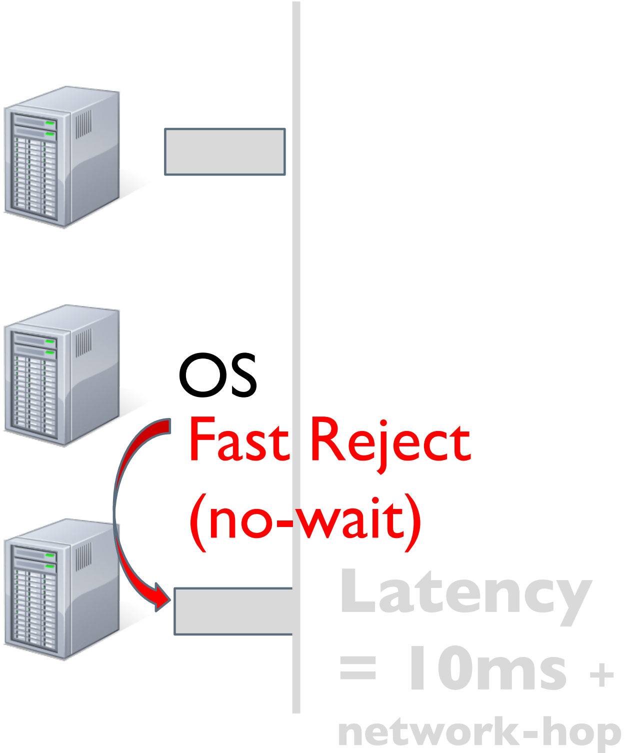
Use-Case



I want < 20ms latency

App | OS

Use-Case

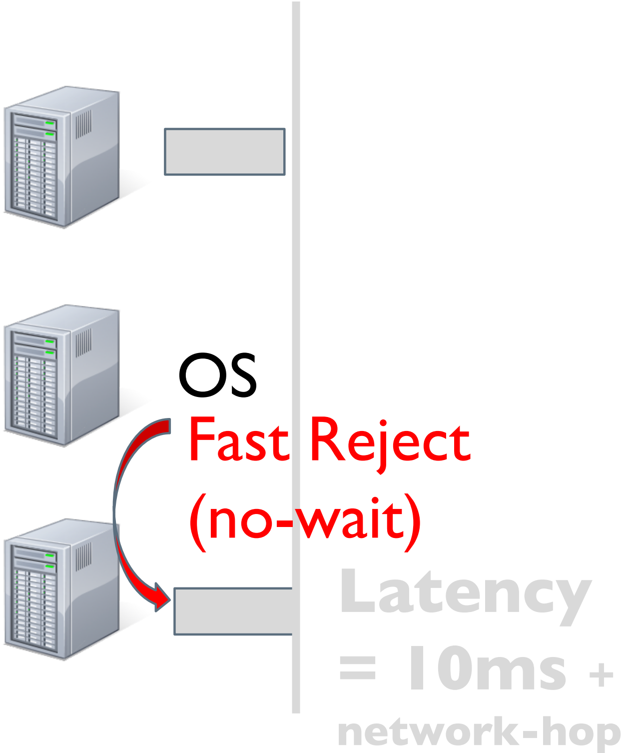


I want < **20ms** latency

App | **OS**

❶ SLO = 20ms

Use-Case



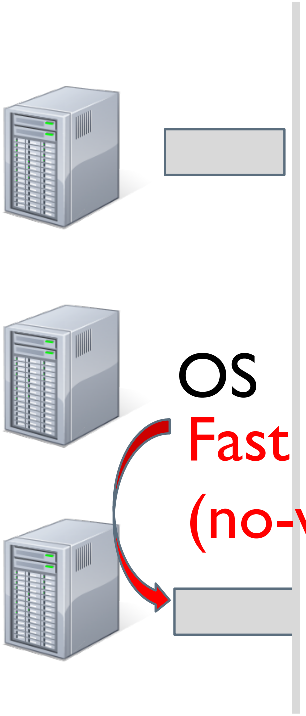
I want < **20ms** latency

App | **OS**

❶ SLO = 20ms

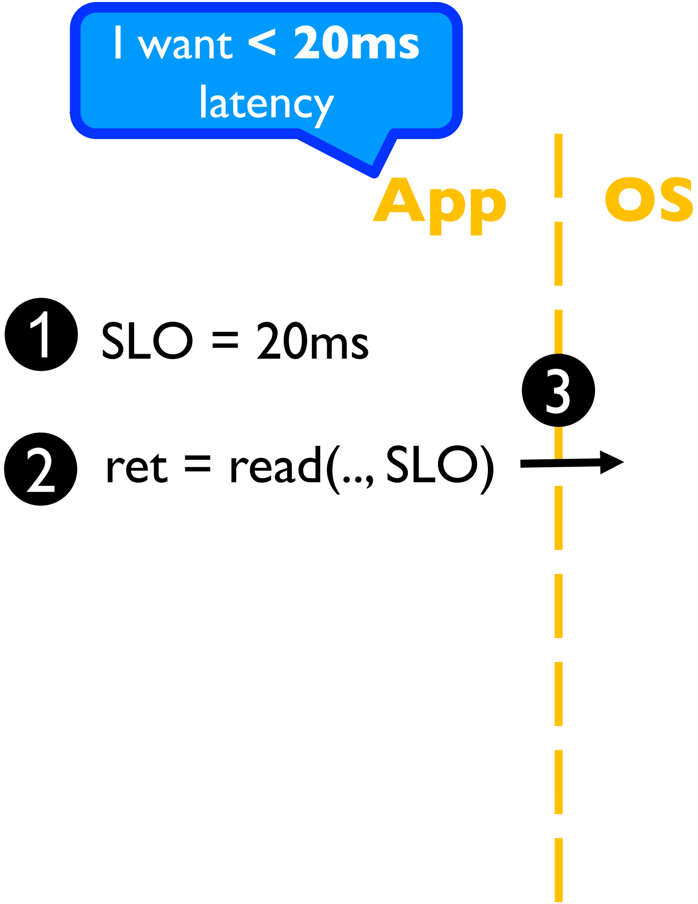
❷ ret = read(.., SLO)

Use-Case



Fast Reject (no-wait)

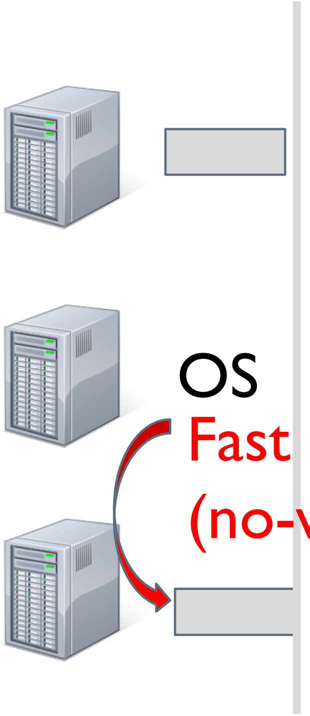
Latency = 10ms + network-hop



① SLO = 20ms

② ret = read(.., SLO)

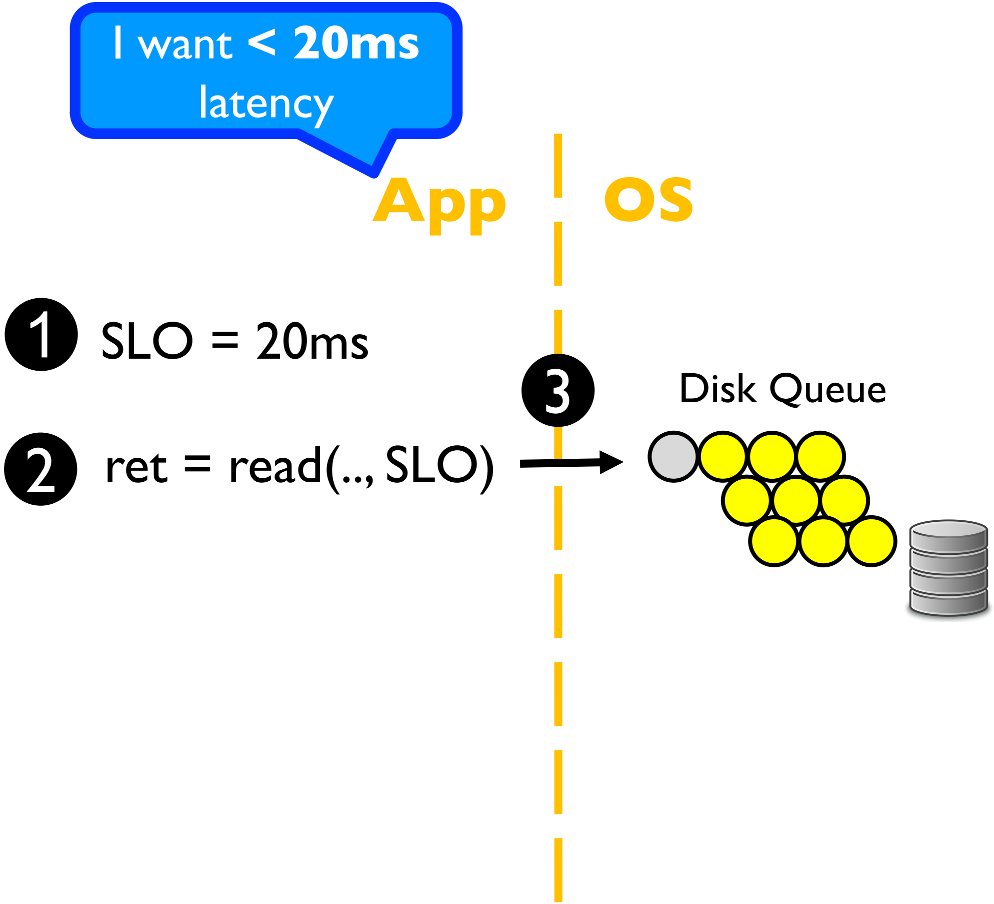
Use-Case



OS

Fast Reject (no-wait)

Latency = 10ms + network-hop

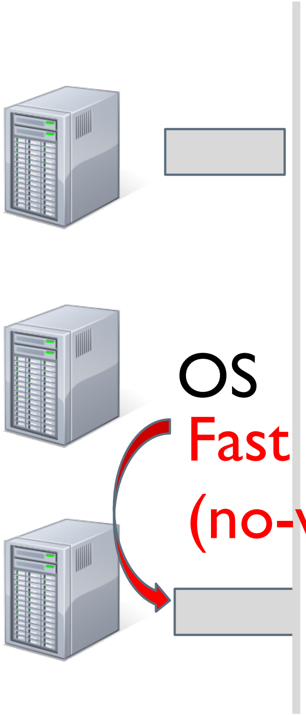


I want < 20ms latency

❶ SLO = 20ms

❷ ret = read(.., SLO)

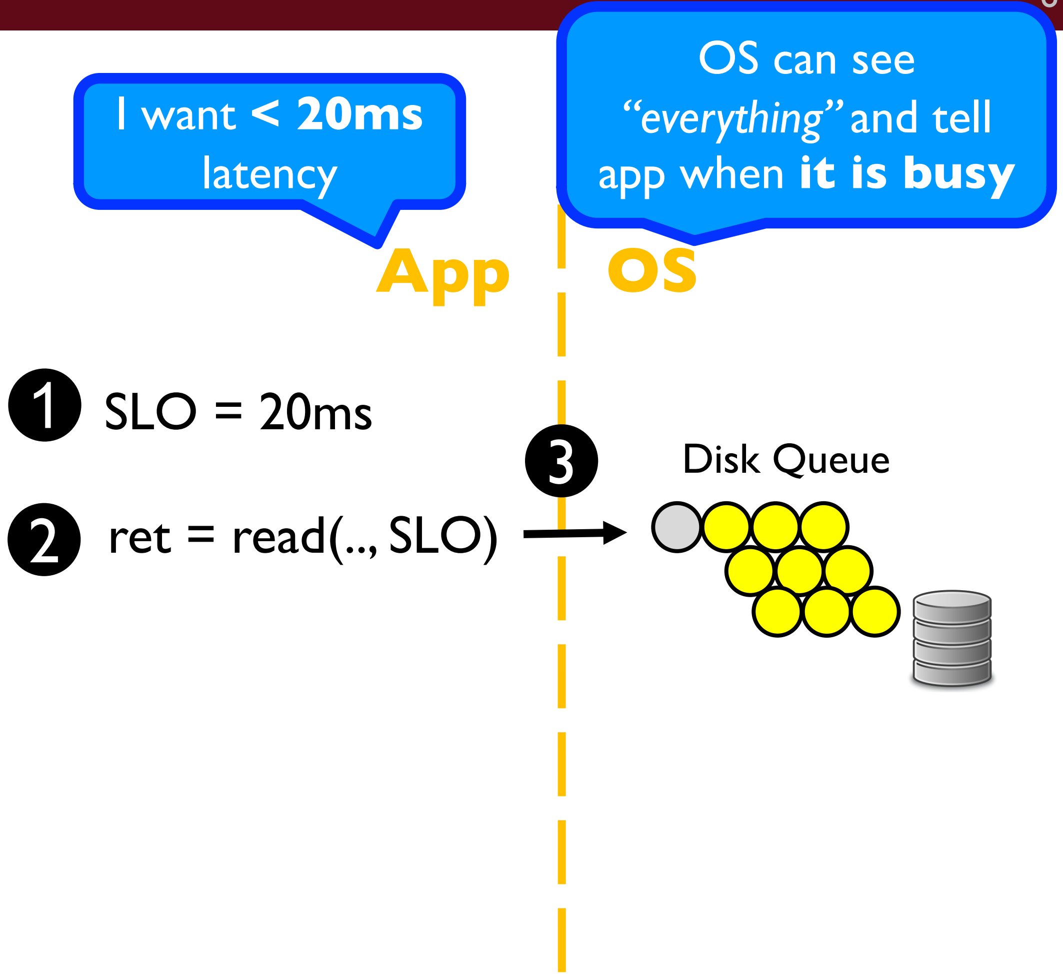
Use-Case



OS

Fast Reject
(no-wait)

Latency = 10ms + network-hop



I want **< 20ms** latency

OS can see *"everything"* and tell app when **it is busy**

App

OS

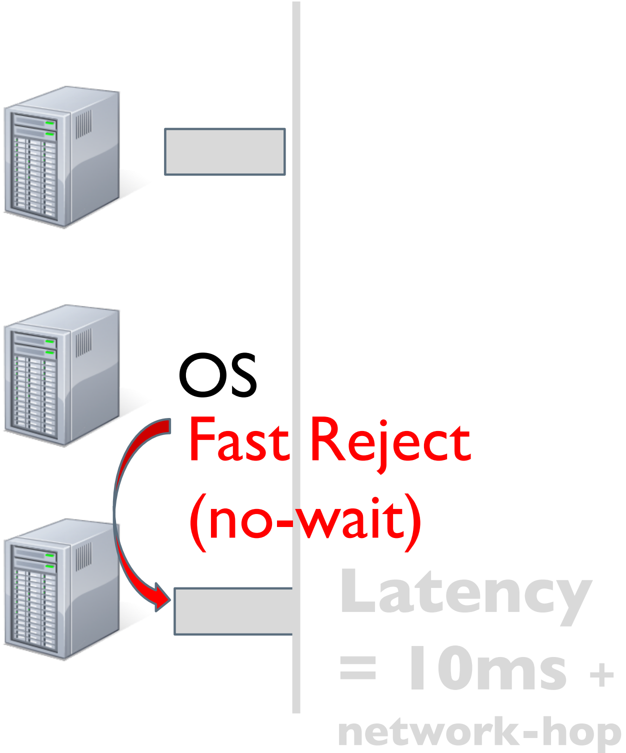
❶ SLO = 20ms

❷ ret = read(.., SLO)

❸

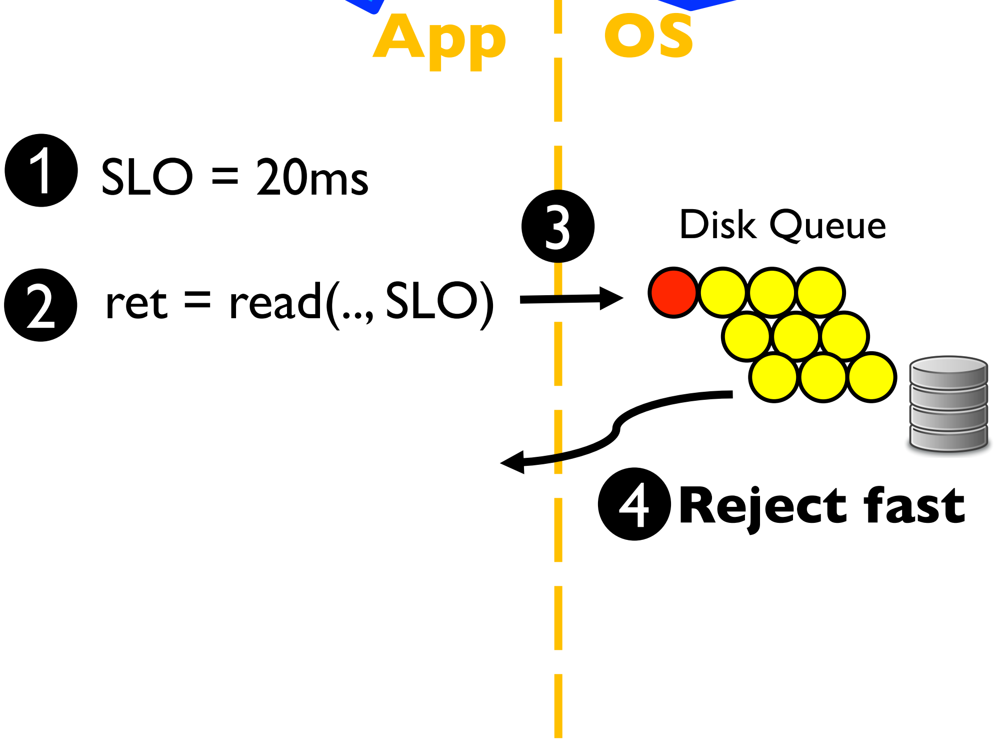
Disk Queue

Use-Case

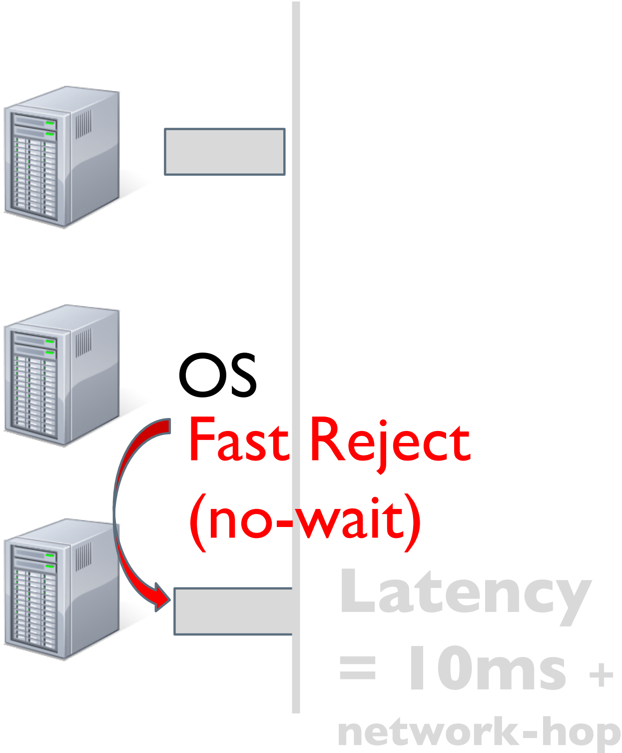


I want **< 20ms** latency

OS can see *"everything"* and tell app when **it is busy**

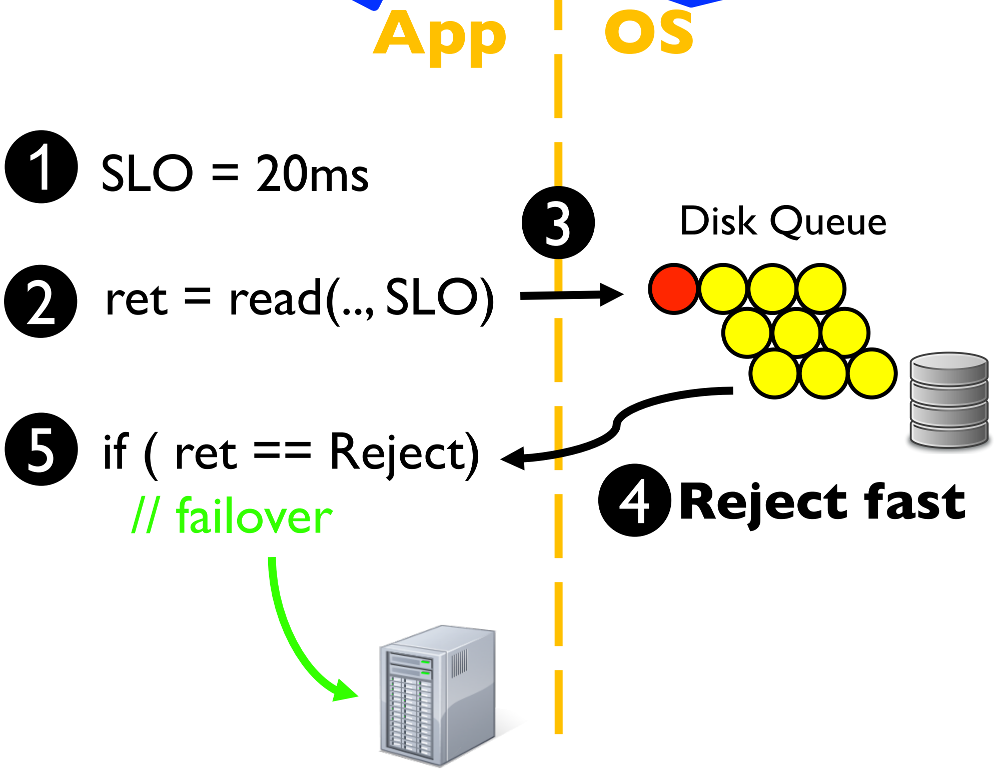


Use-Case

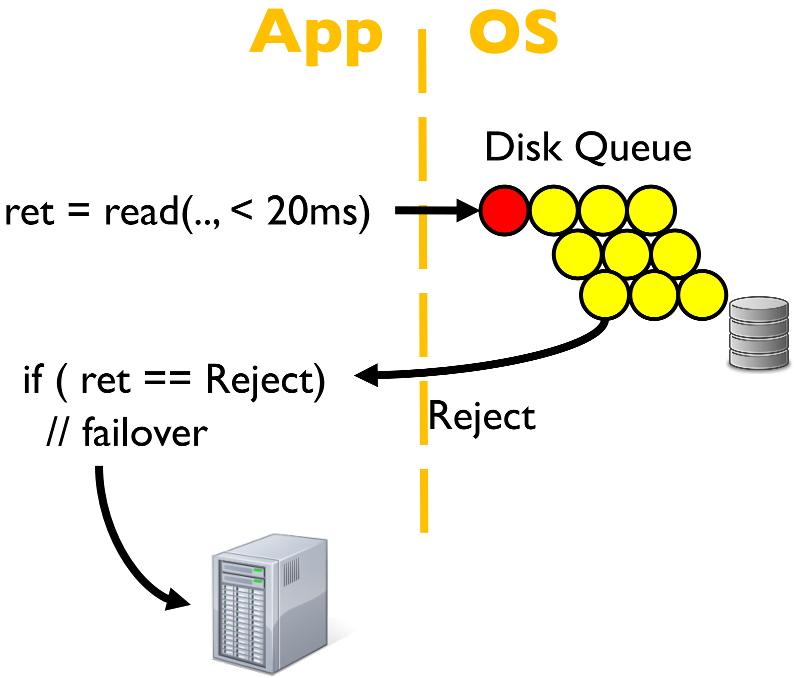


I want **< 20ms** latency

OS can see *“everything”* and tell app when **it is busy**

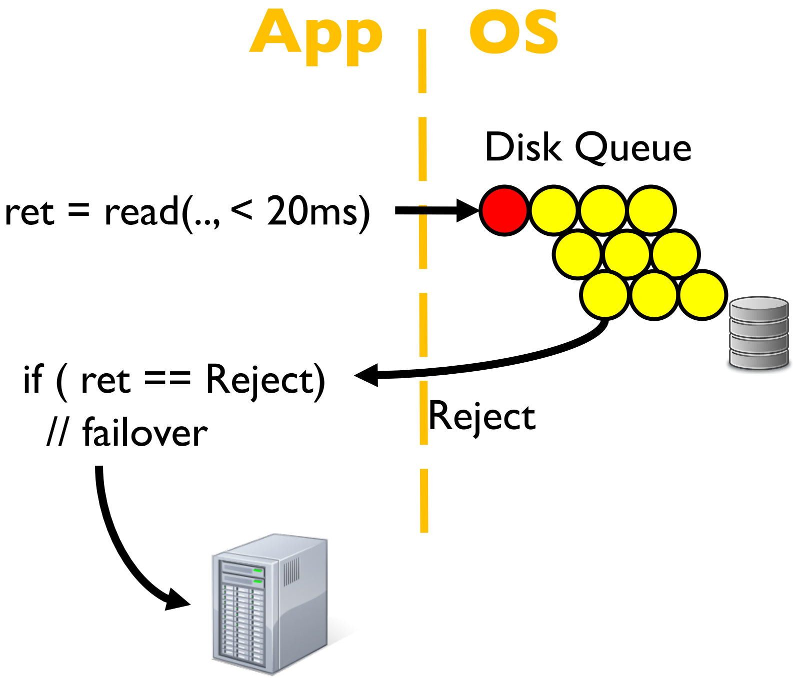


MittOS



- MittOS **Principles**

MittOS

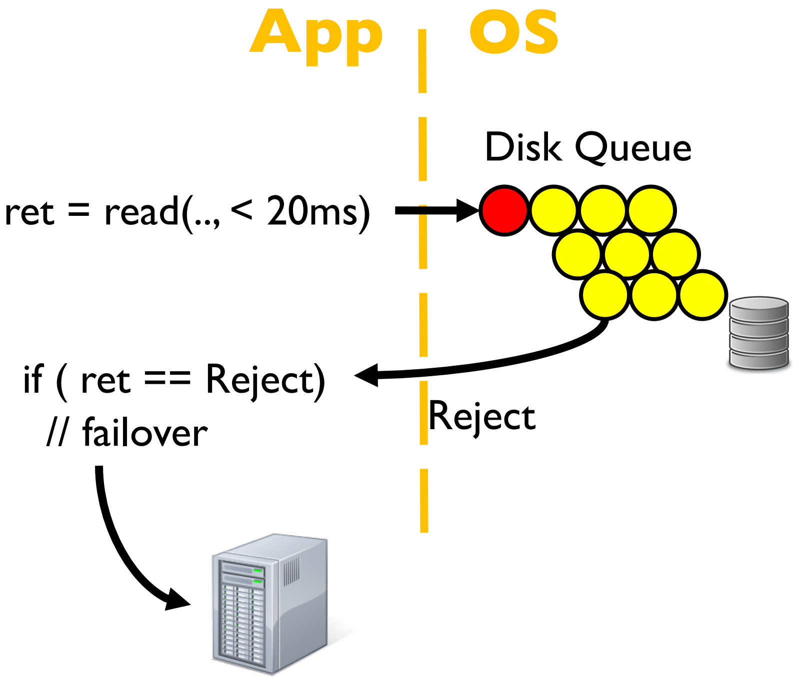


```
ret = read(.., < 20ms)

if ( ret == Reject)
  // failover
```

- MittOS **Principles**
  - SLO-aware interface
  - Reject fast

MittOS

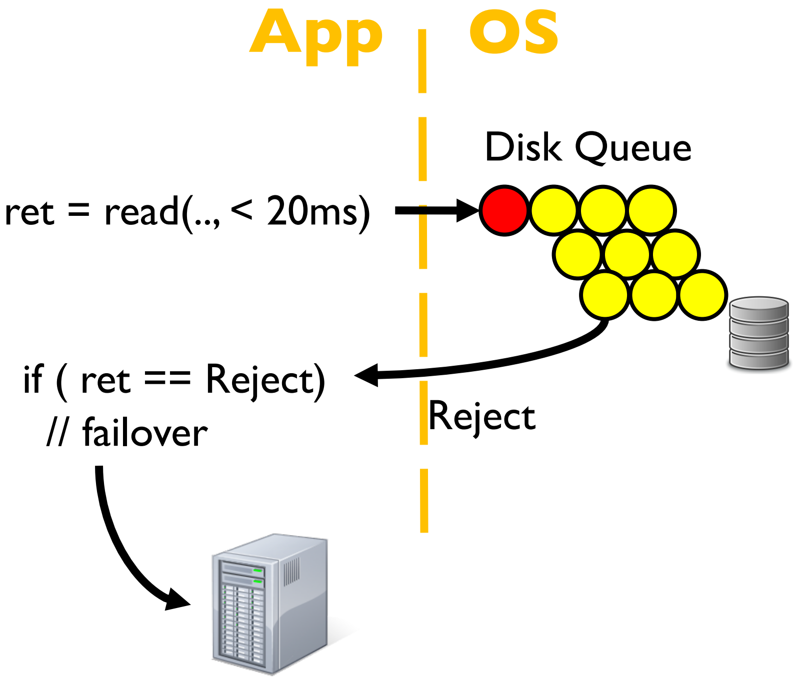


ret = read(.., < 20ms)

if (ret == Reject)
  // failover

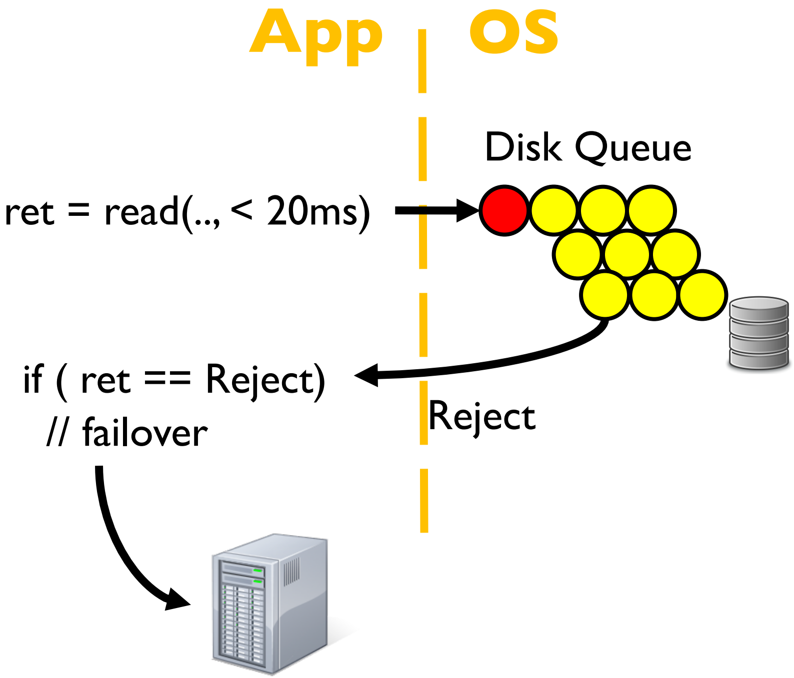
- MittOS **Principles**
  - SLO-aware interface
  - Reject fast
  - *Transparent* of *busyness*

MittOS



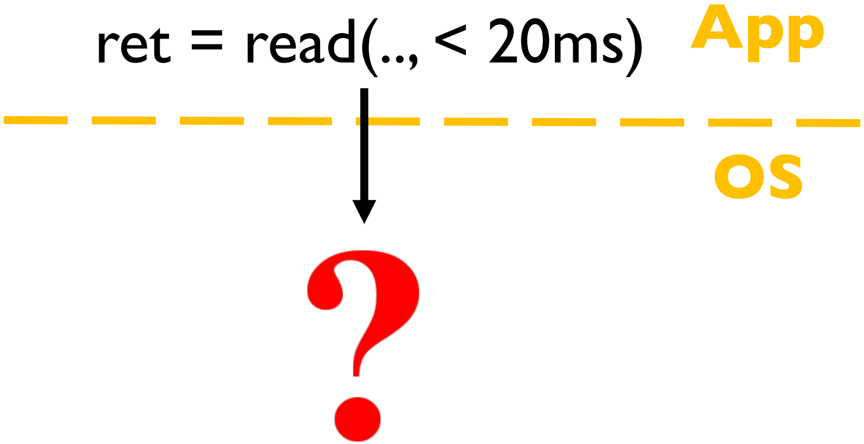
- MittOS **Principles**
  - SLO-aware interface
  - Reject fast
  - *Transparent* of *busyness*
    - **PC** era: is best effort (cannot reject IOs)

MittOS



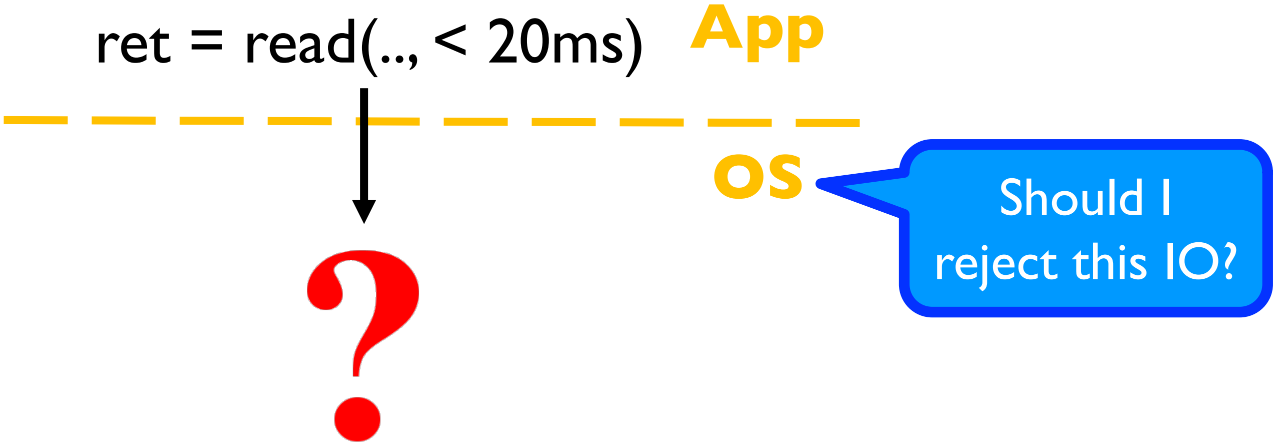
- MittOS **Principles**
  - SLO-aware interface
  - Reject fast
  - *Transparent* of *busyness*
    - **PC** era: is best effort (cannot reject IOs)
    - **DC** era: Less-busy replicas available

Challenge



ret = read(.., < 20ms)

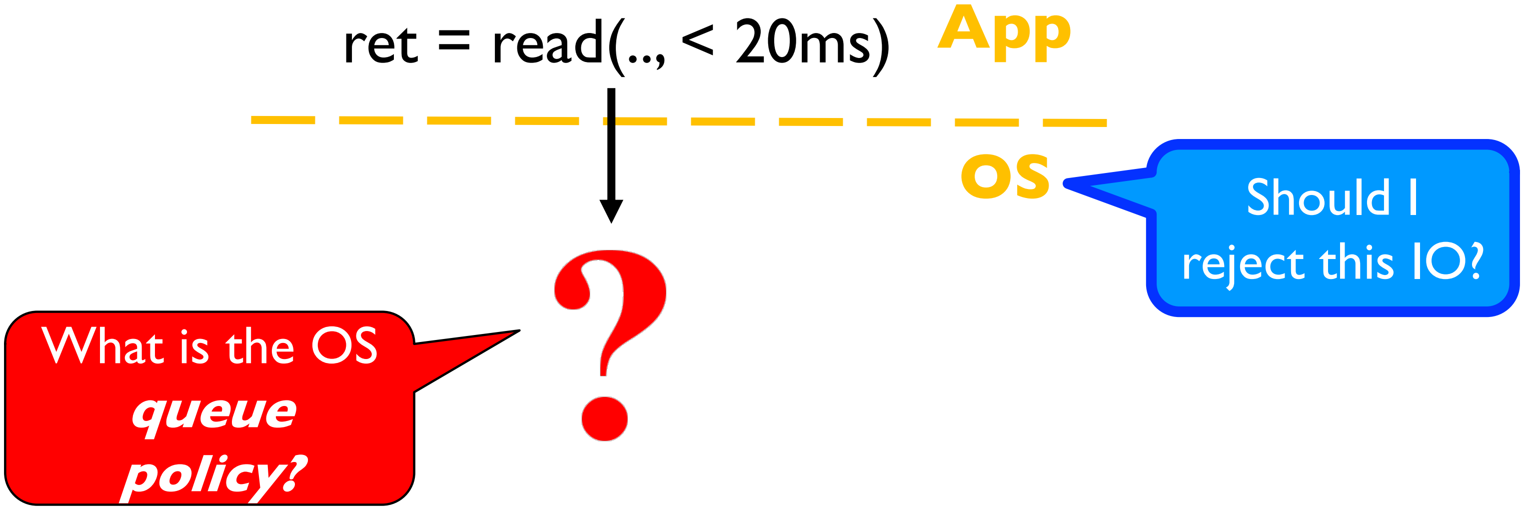
Challenge



ret = read(.., < 20ms)

Should I reject this IO?

Challenge



ret = read(.., < 20ms)

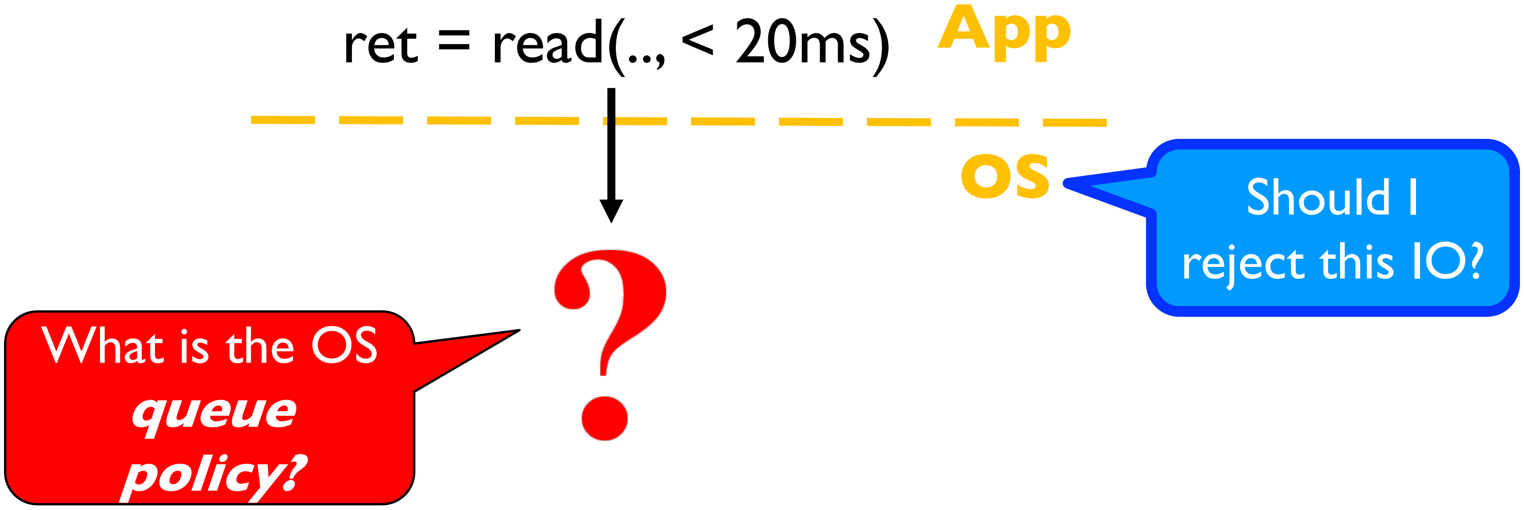
App

OS

Should I reject this IO?

What is the OS ***queue policy?***

Challenge



ret = read(.., < 20ms)

App

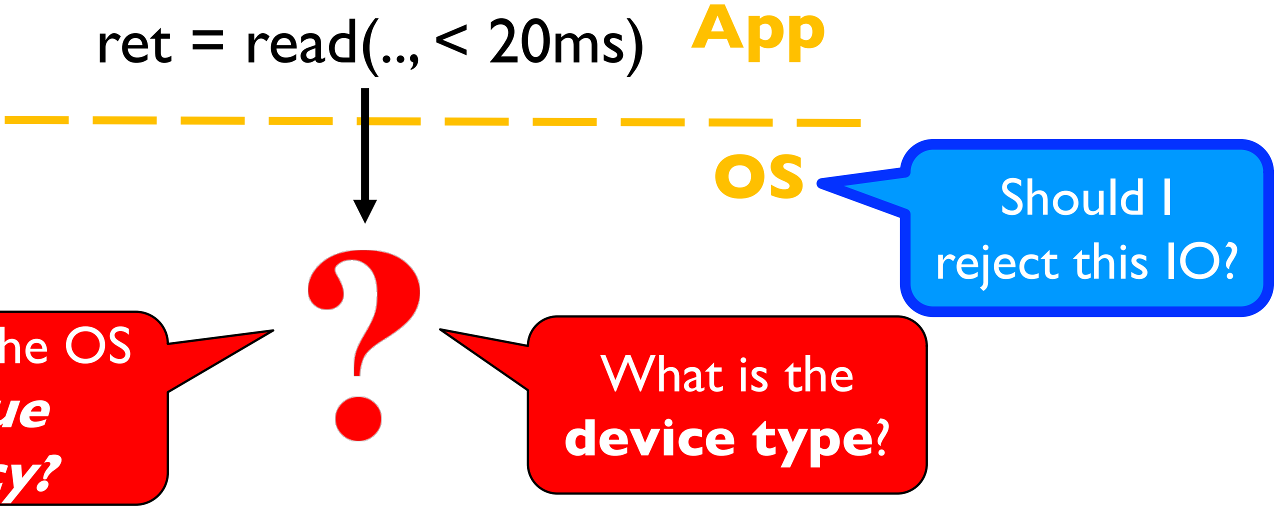
OS

Should I reject this IO?

What is the OS ***queue policy?***

FIFO, elevator, CFQ, etc.

Challenge



ret = read(.., < 20ms)

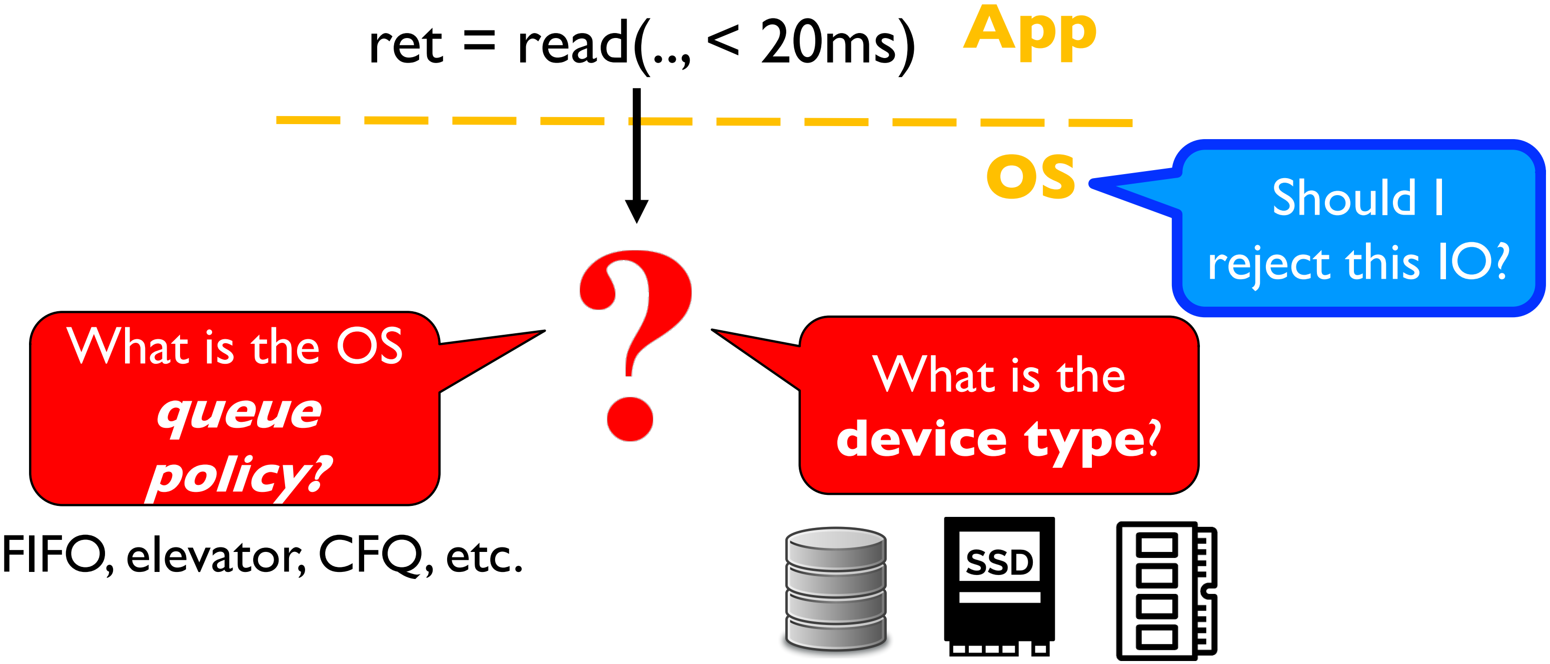
Should I reject this IO?

What is the OS ***queue policy?***

What is the **device type?**

FIFO, elevator, CFQ, etc.

Challenge



ret = read(.., < 20ms)

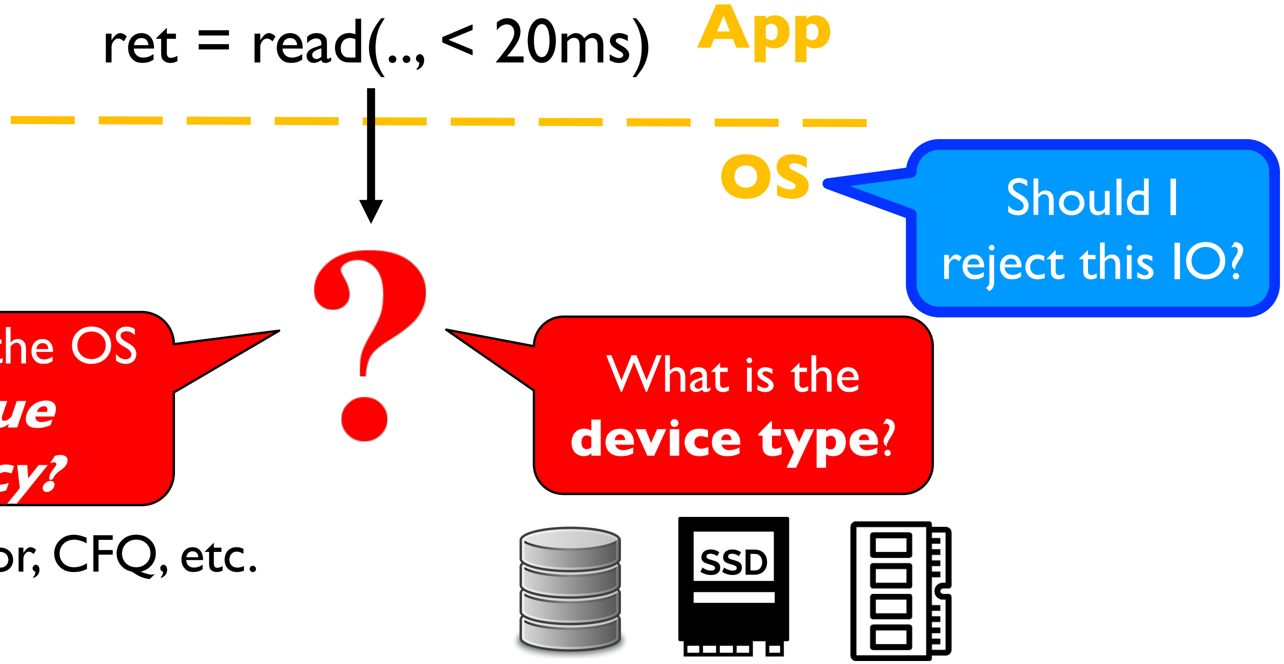
Should I reject this IO?

What is the OS ***queue policy?***

FIFO, elevator, CFQ, etc.

What is the **device type?**

Challenge



ret = read(.., < 20ms)

App

OS

Should I reject this IO?

?

What is the OS ***queue policy?***

What is the **device type?**

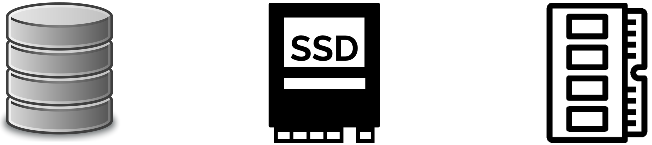
FIFO, elevator, CFQ, etc.

Prediction depends on queue policy and device type

Contribution

MittOS principle: Support fast rejecting SLO-aware interface

Contribution



MittOS principle: Support fast rejecting SLO-aware interface

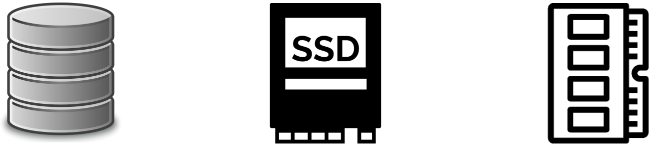
Contribution

MittOS Latency Prediction

Disk

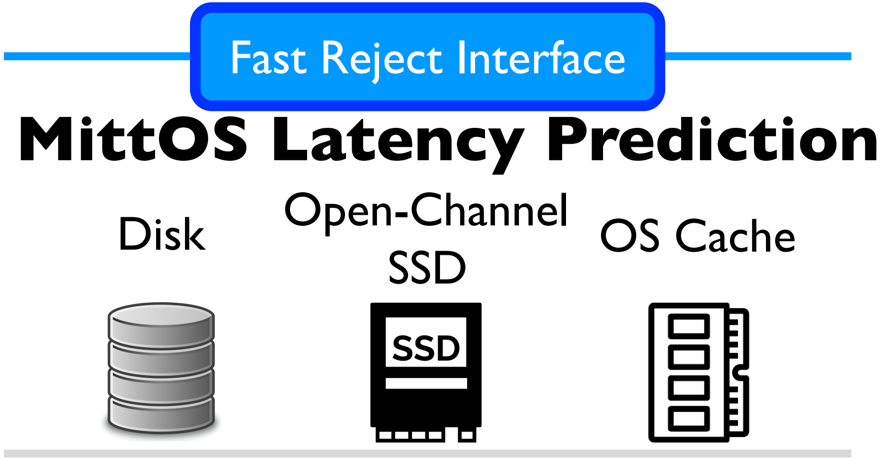
Open-Channel SSD

OS Cache



MittOS principle: Support fast rejecting SLO-aware interface

Contribution

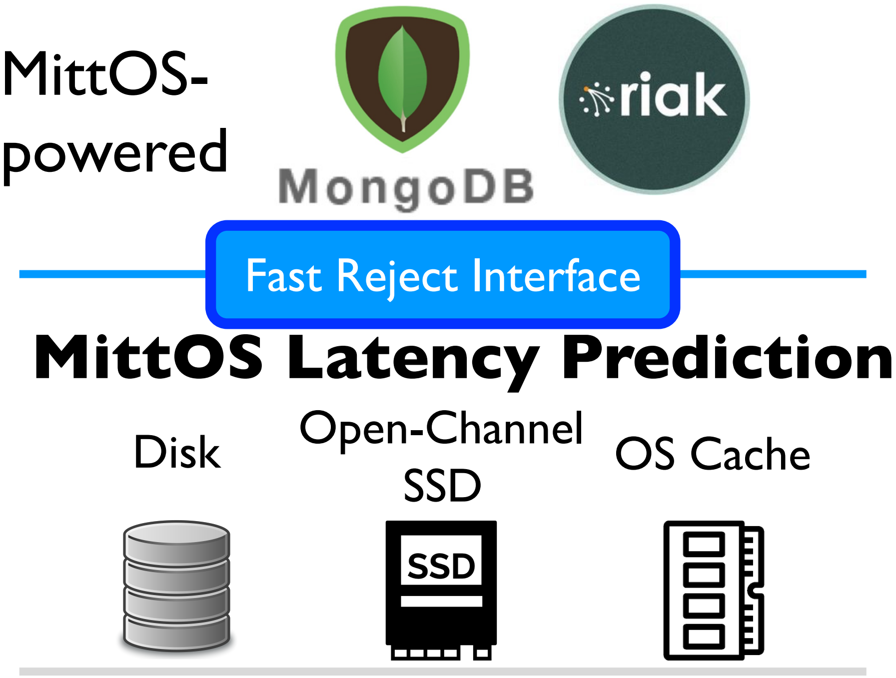


Fast Reject Interface

MittOS Latency Prediction

MittOS principle: Support fast rejecting SLO-aware interface

Contribution



MittOS-powered

Fast Reject Interface

MittOS Latency Prediction

Disk

Open-Channel SSD

OS Cache

MittOS principle: Support fast rejecting SLO-aware interface

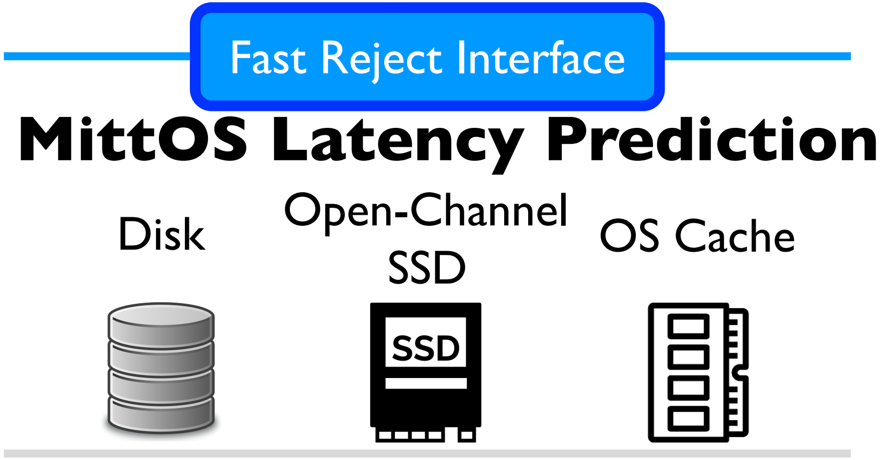
Contribution

+50 LOC

MittOS-powered

MongoDB

riak

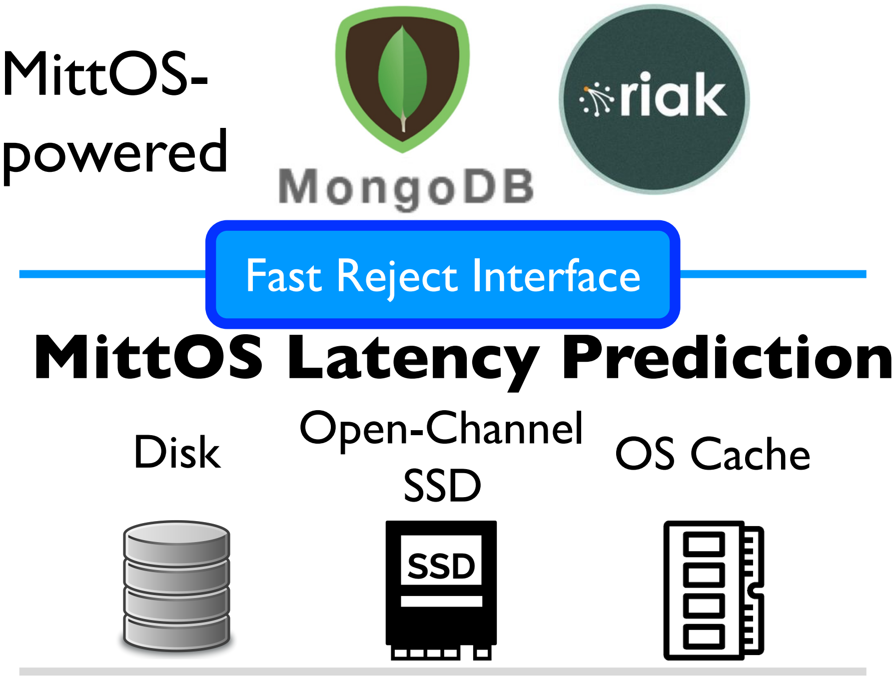


Fast Reject Interface

MittOS Latency Prediction

MittOS principle: Support fast rejecting SLO-aware interface

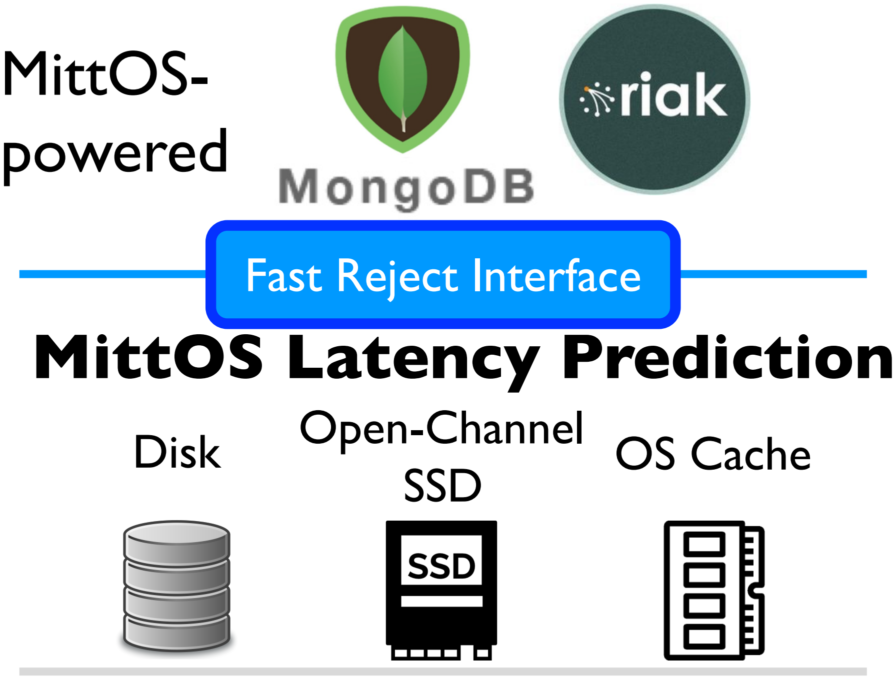
Contribution



MittOS principle: Support fast rejecting SLO-aware interface

vs. **state of the art**: hedged requests, cloning, application timeout, etc.

Contribution **+50 LOC**



MittOS-powered

Fast Reject Interface

MittOS Latency Prediction

Disk

Open-Channel SSD

OS Cache

MittOS principle: Support fast rejecting SLO-aware interface

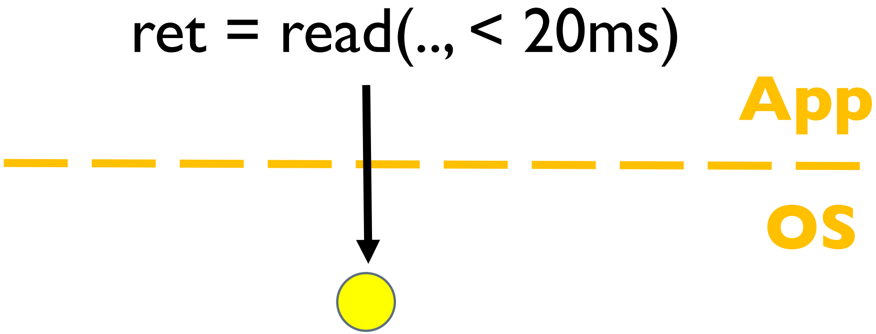
vs. **state of the art**: hedged requests, cloning, application timeout, etc.

Cut tail:
50% latency reduction above 75 percentile

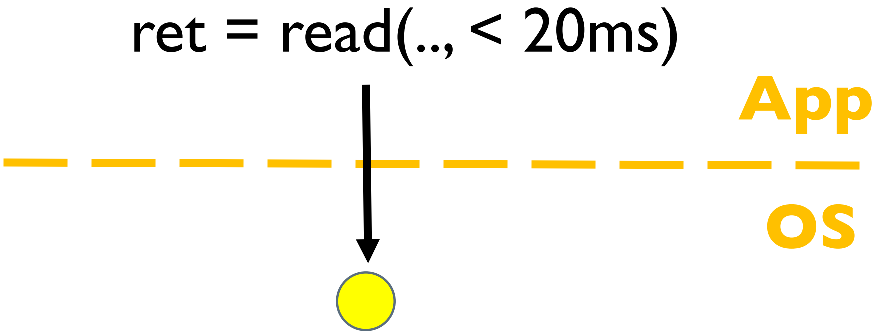
Outline

- ☐ Introduction
- ☐ **Design**
  - Challenges
  - Solutions
- ☐ Evaluation
- ☐ Conclusion

Prediction

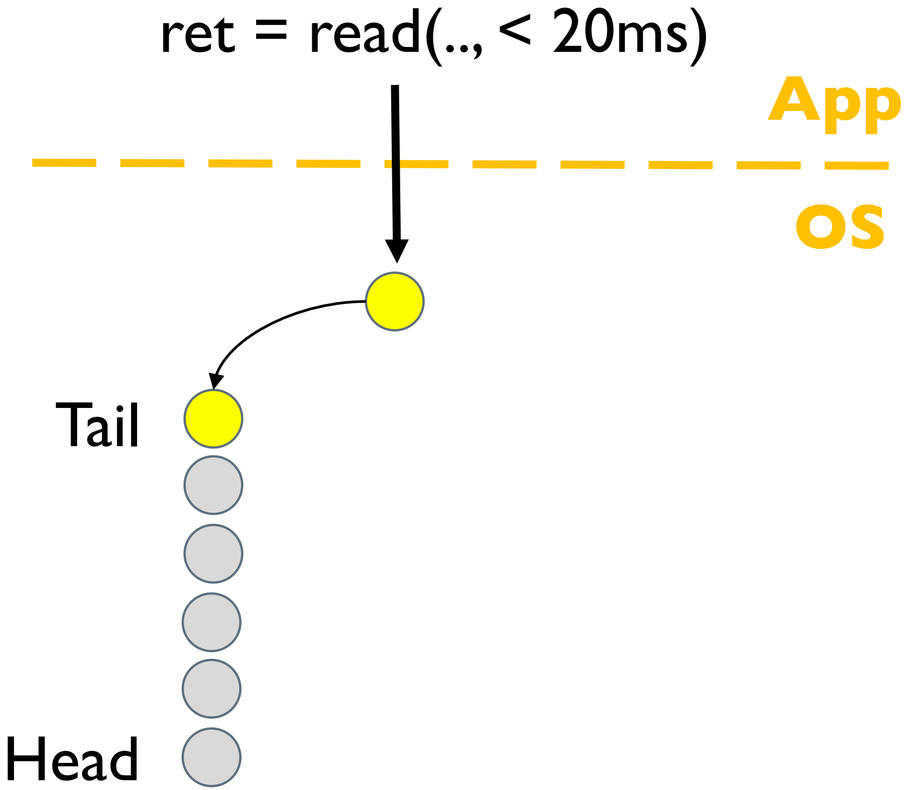

ret = read(.., < 20ms)

Prediction



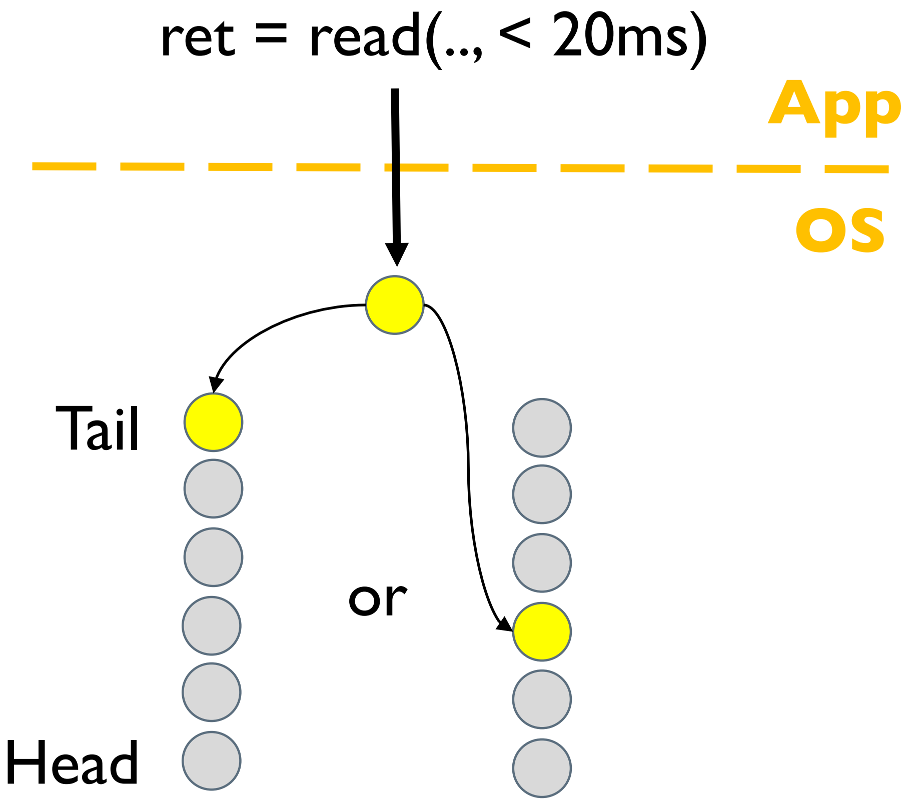
How to **predict** latency *before submitting* to the device?

Prediction



How to **predict** latency *before submitting* to the device?

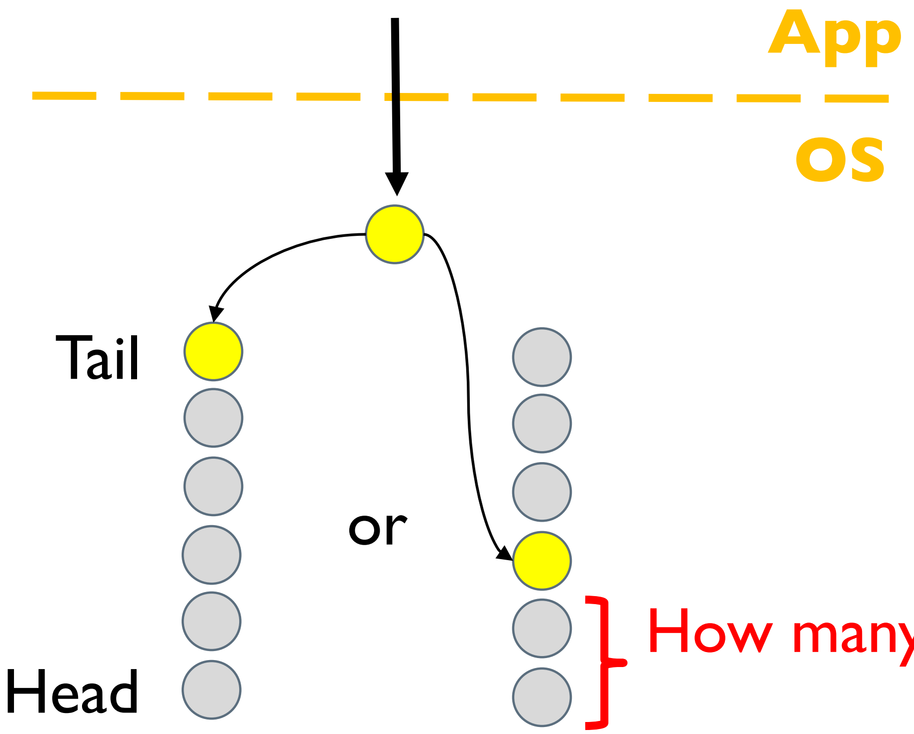
Prediction



How to **predict** latency *before submitting* to the device?

Prediction

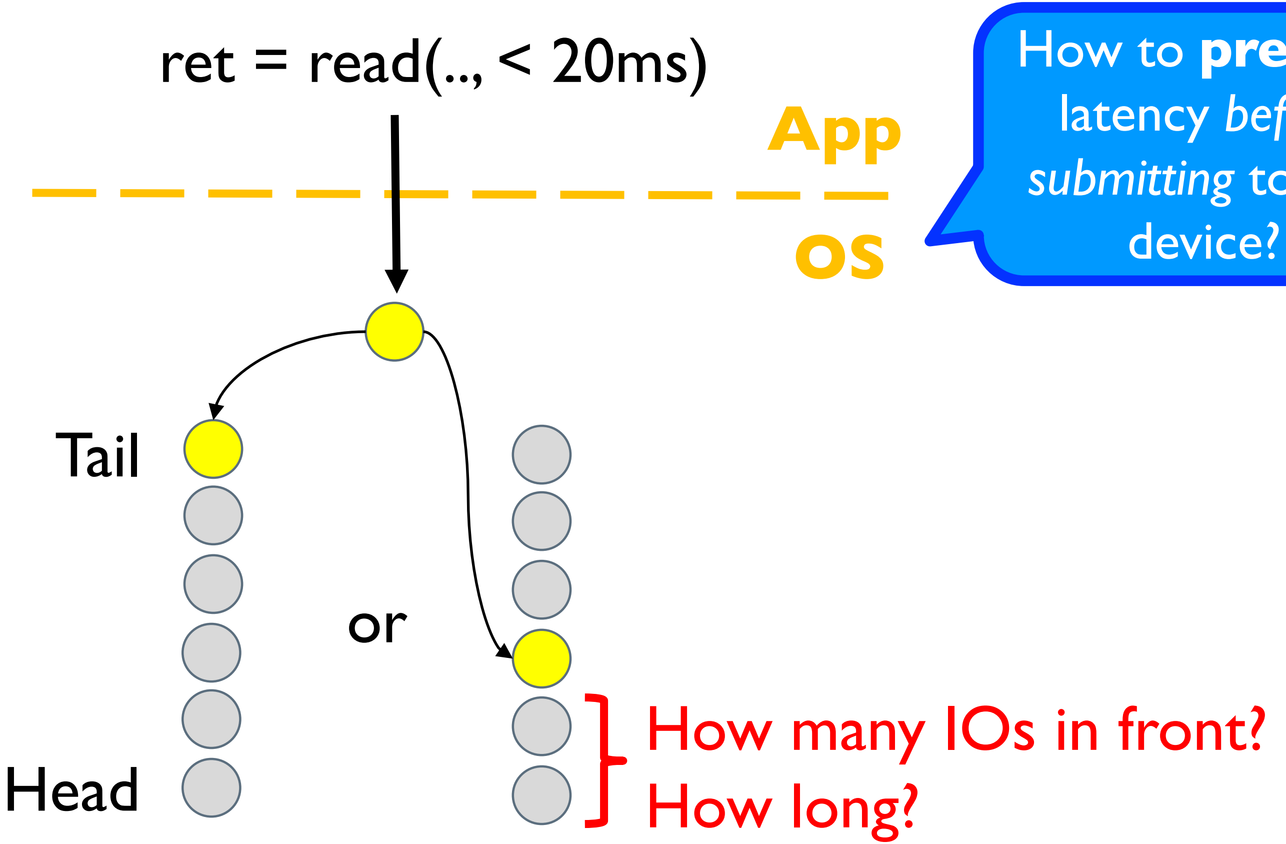
ret = read(.., < 20ms)



How to **predict** latency *before submitting* to the device?

How many IOs in front?

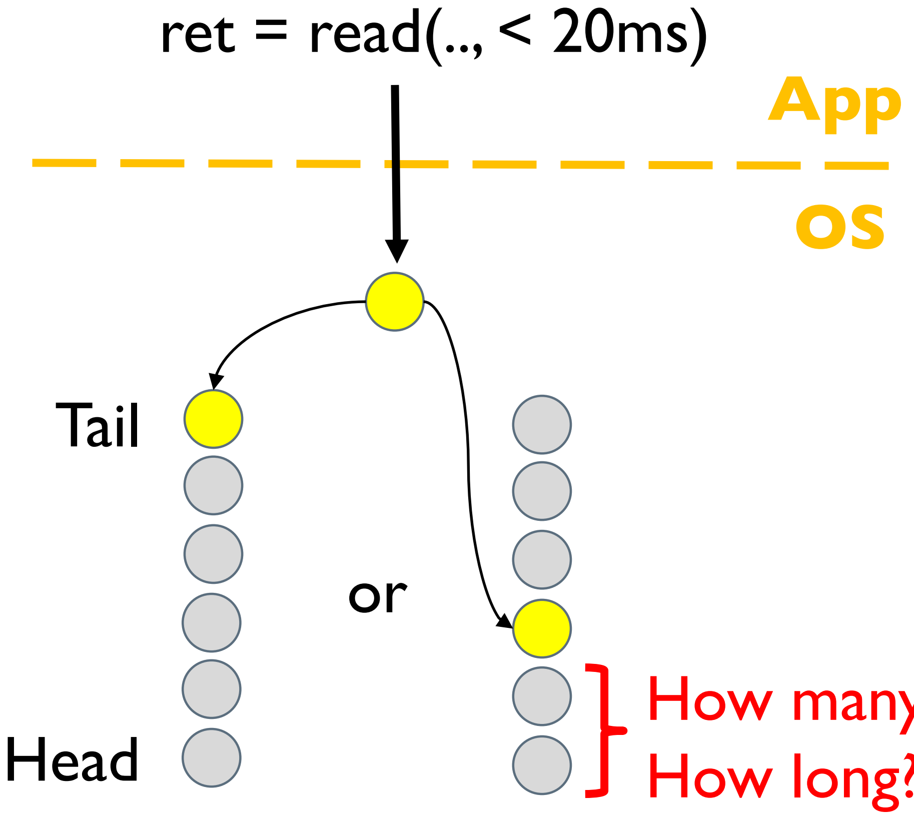
Prediction



How many IOs in front?
How long?

How to **predict** latency *before submitting* to the device?

Prediction



How to **predict** latency *before submitting* to the device?

Latency < SLO → **Accept**
Latency > SLO → **Reject**

How many IOs in front?
How long?

Challenge #1: Modeling Queue Policy

App

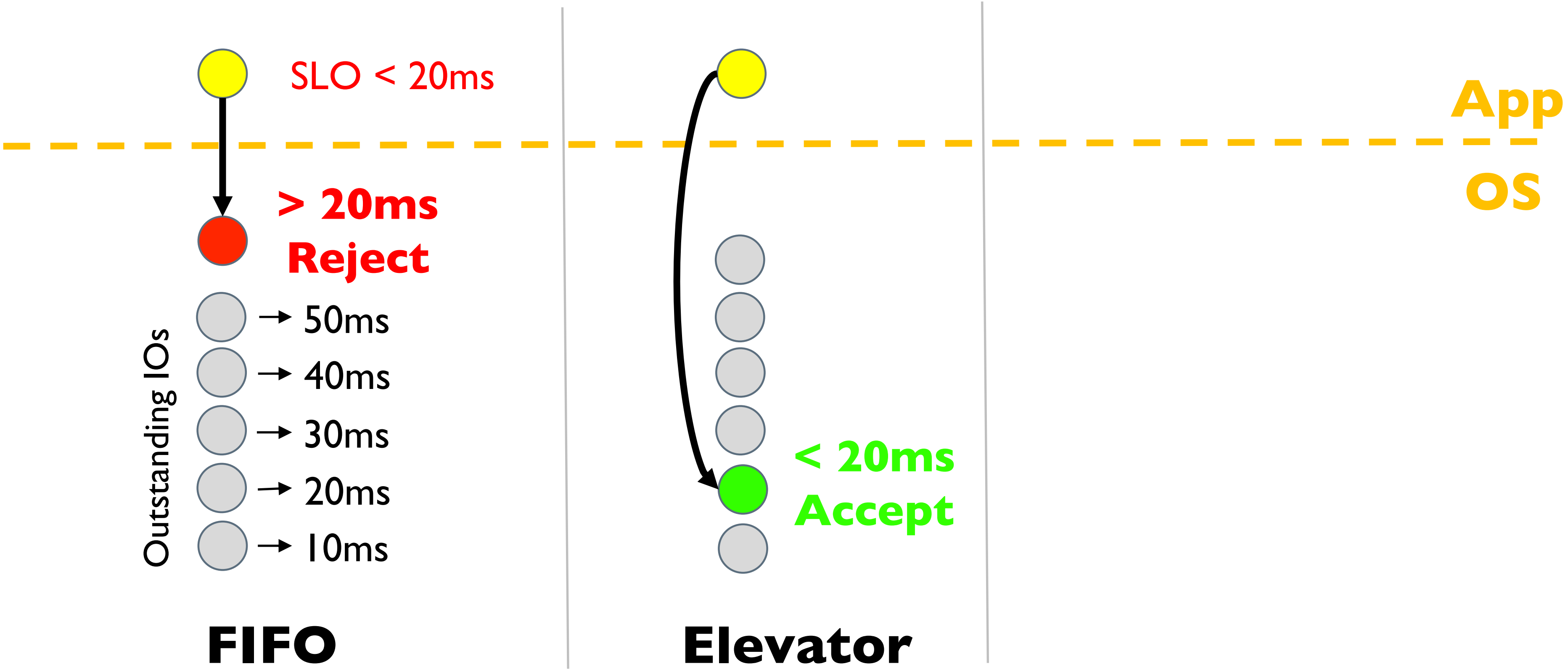
OS

Challenge #1: Modeling Queue Policy

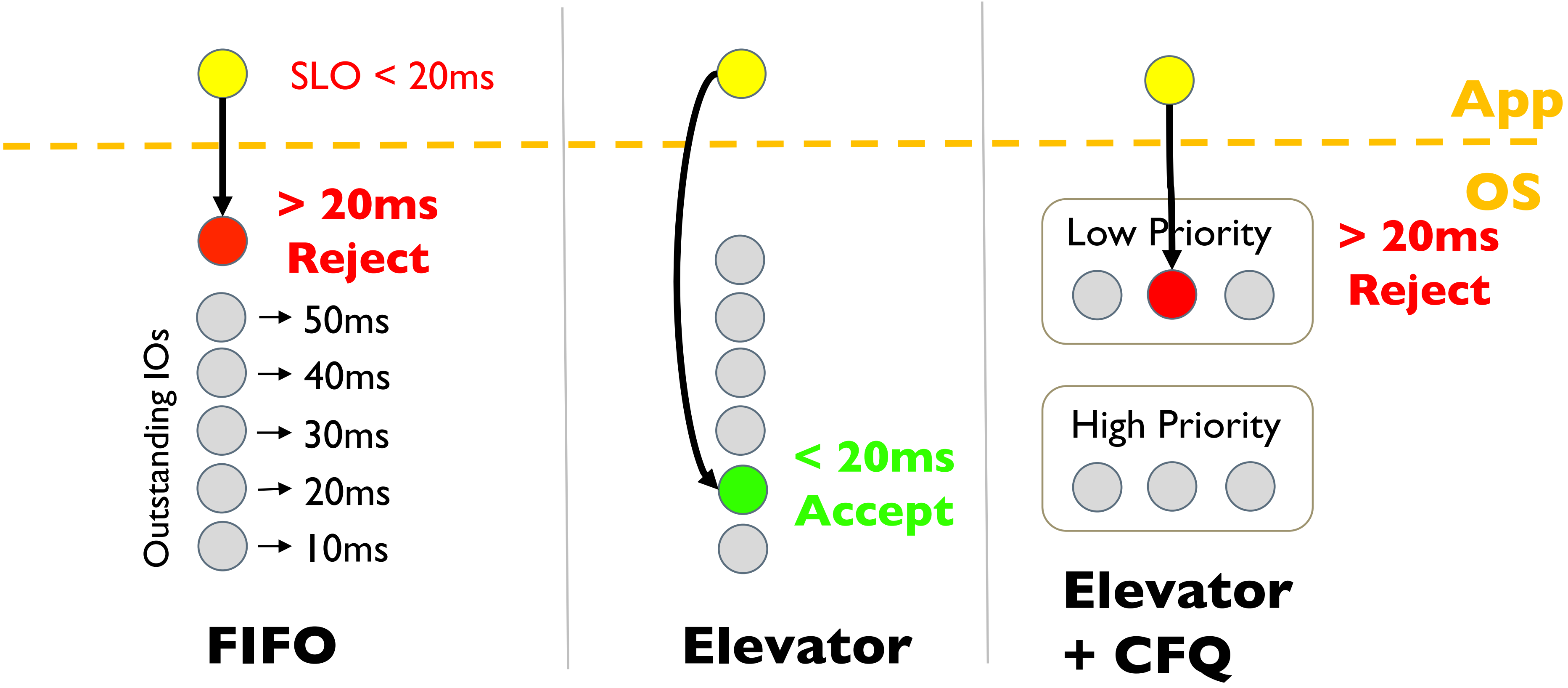
Challenge #1: Modeling Queue Policy

Challenge #1: Modeling Queue Policy

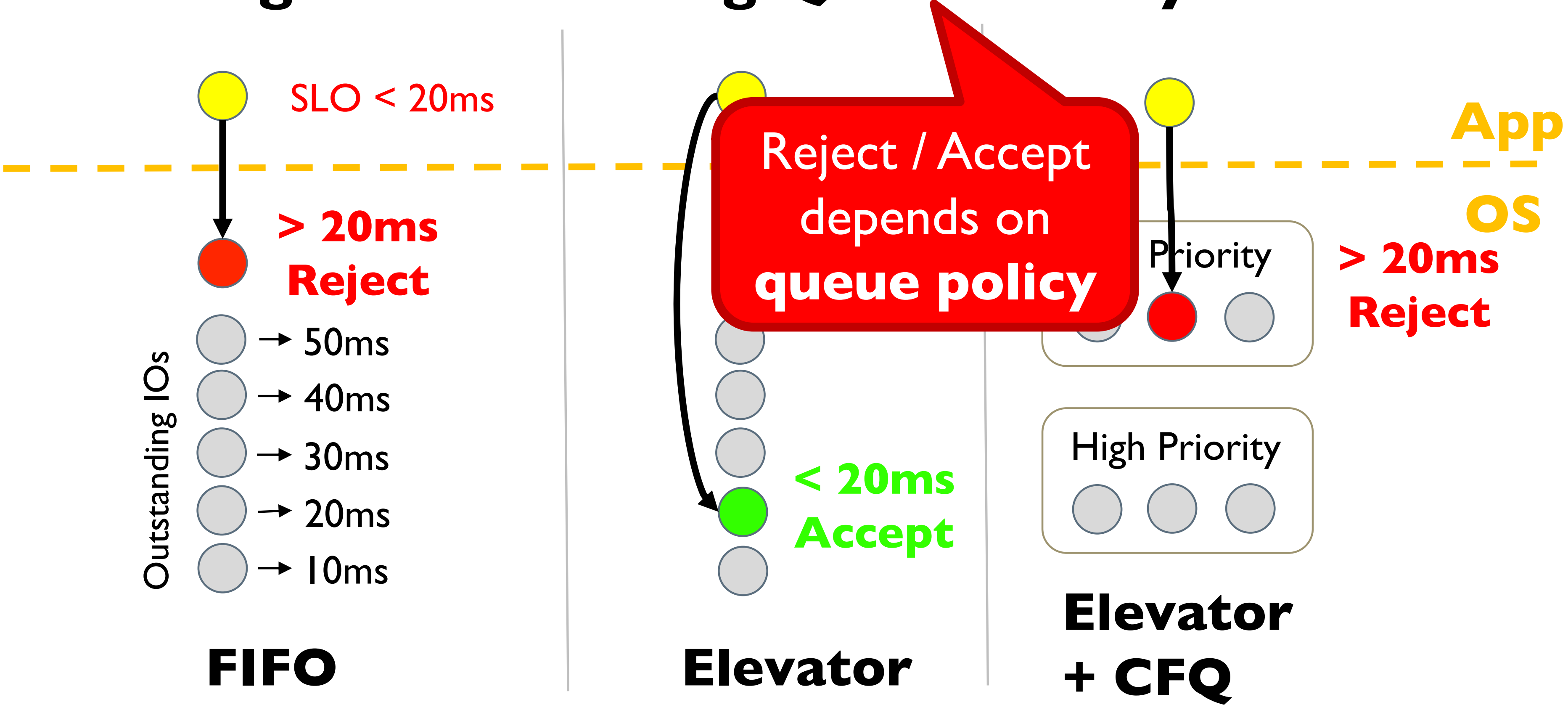
Challenge #1: Modeling Queue Policy



Challenge #1: Modeling Queue Policy



Challenge #1: Modeling Queue Policy



App

OS

SLO < 20ms

> 20ms
Reject

Outstanding IOs

→ 50ms
→ 40ms
→ 30ms
→ 20ms
→ 10ms

FIFO

Reject / Accept depends on **queue policy**

< 20ms
Accept

Elevator

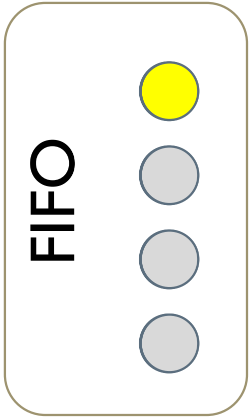
Priority

High Priority

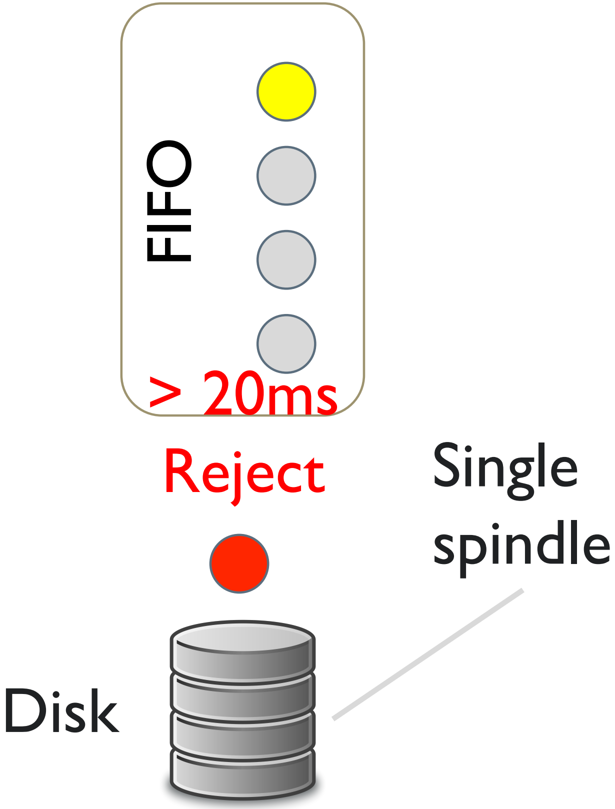
> 20ms
Reject

Elevator + CFQ

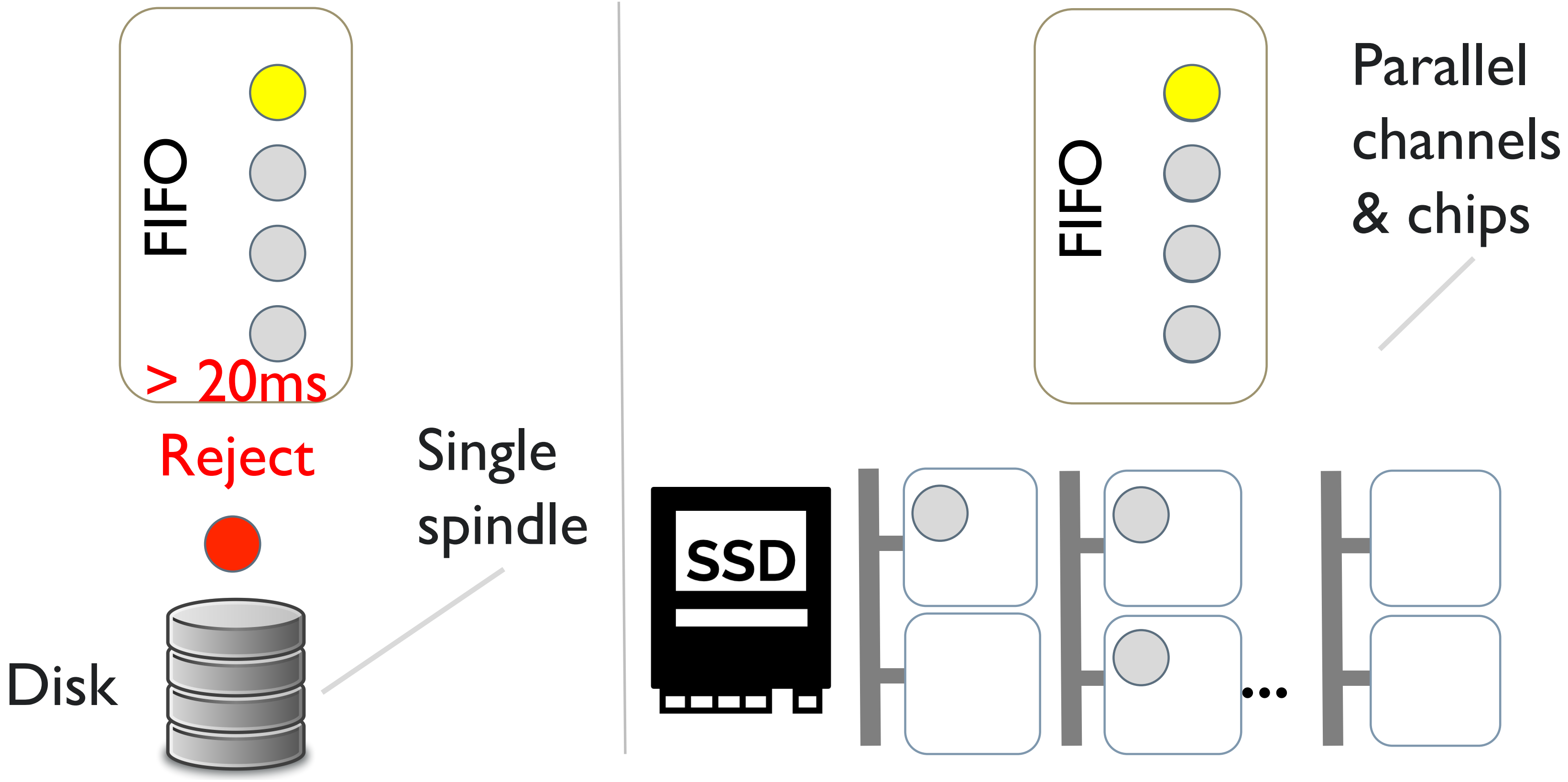
Challenge #2: Device Type



Challenge #2: Device Type



Challenge #2: Device Type



FIFO

> 20ms

Reject

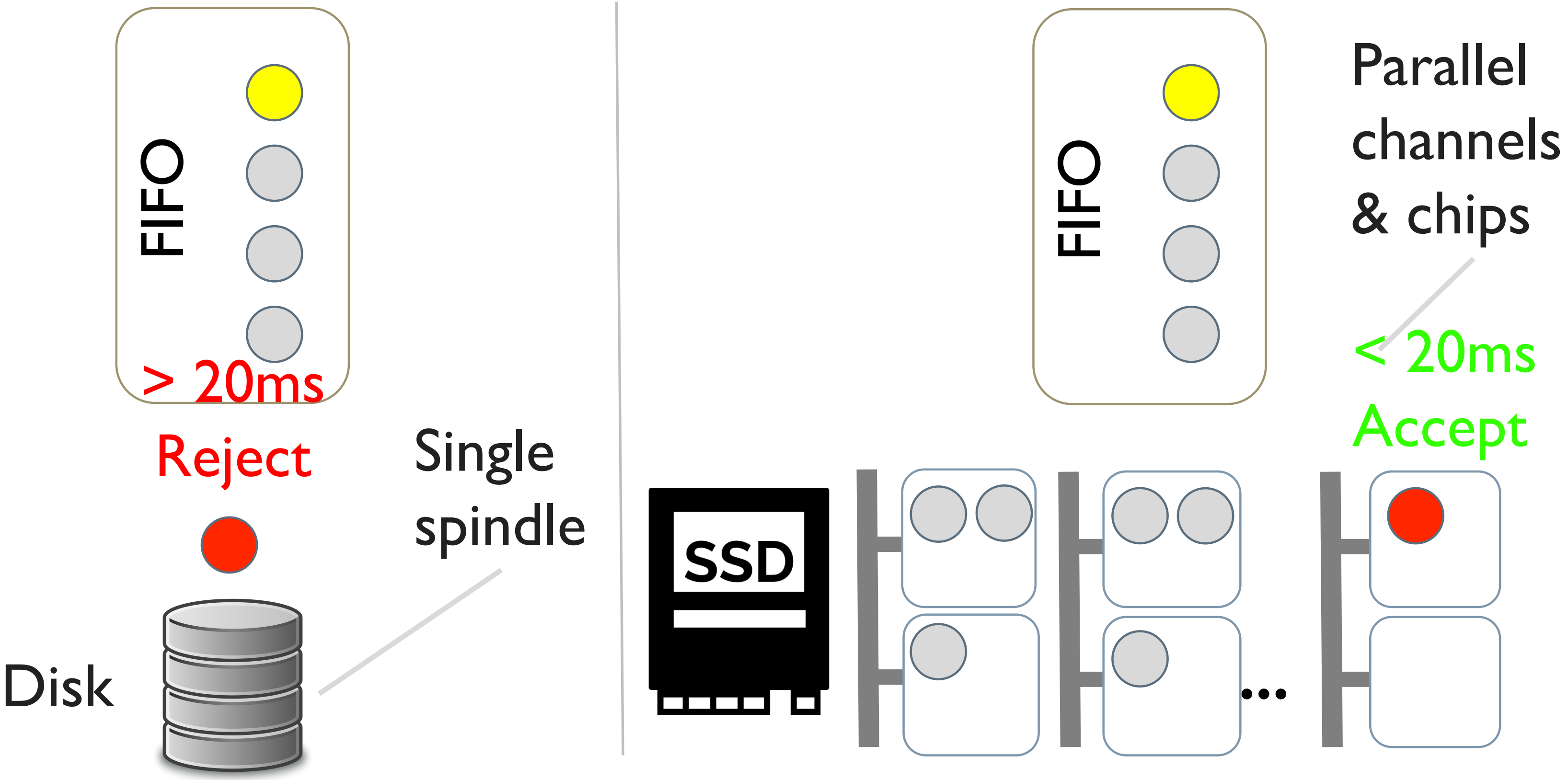
Single spindle

Disk

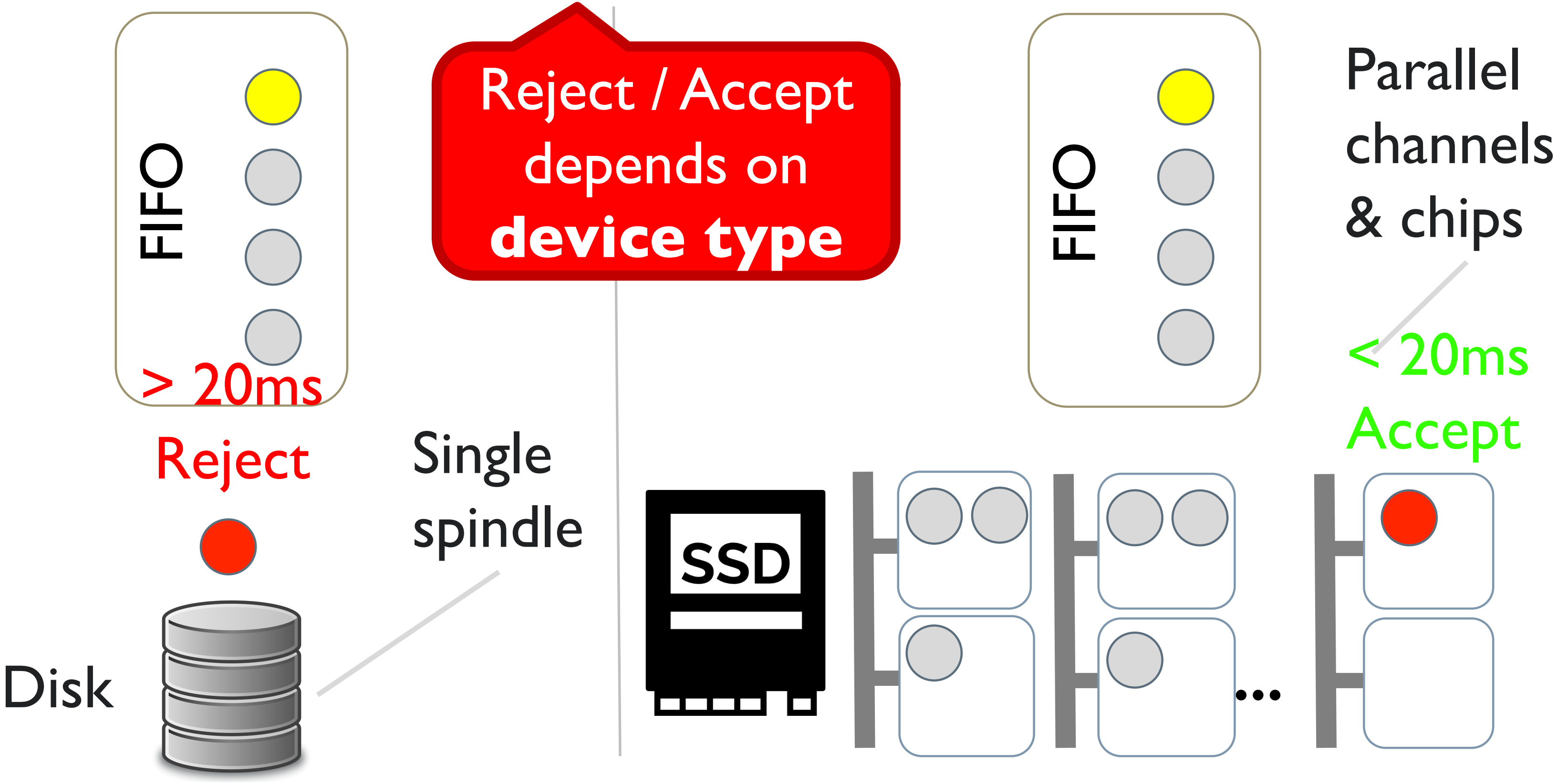
FIFO

Parallel channels & chips

Challenge #2: Device Type



Challenge #2: Device Type



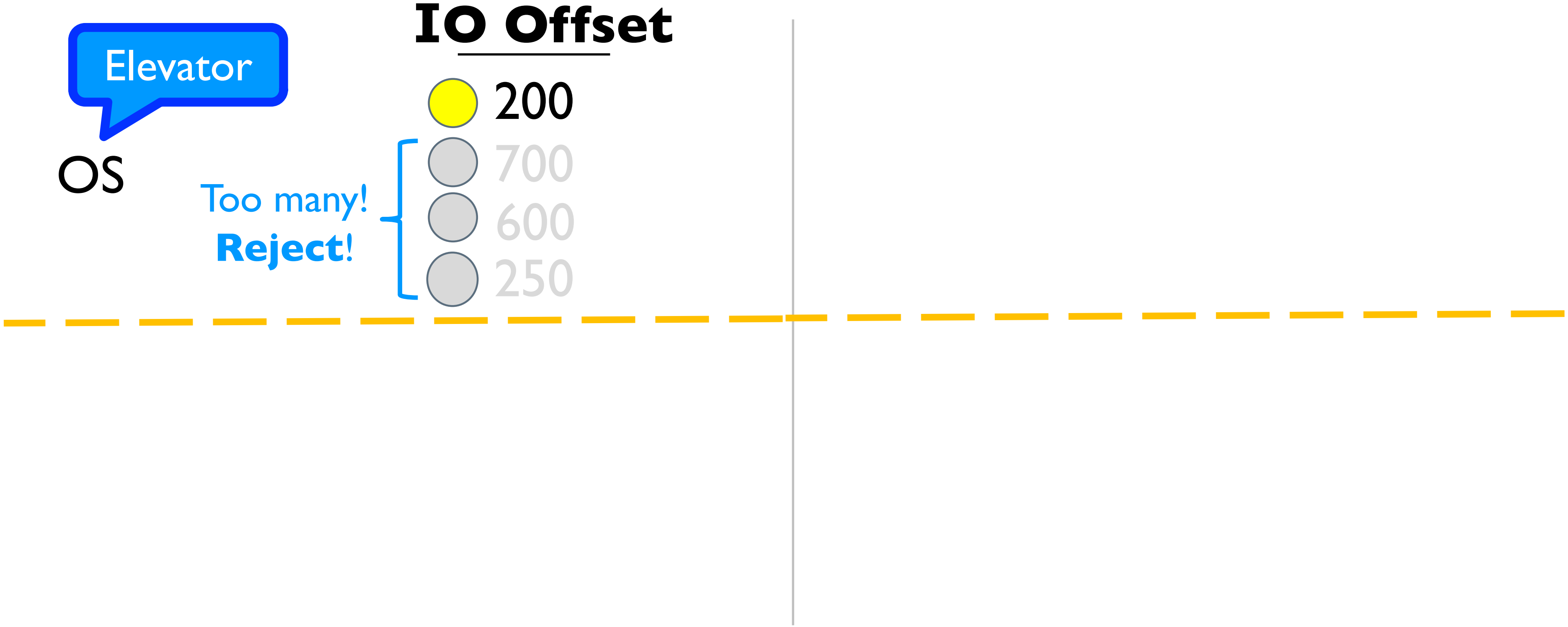
Reject / Accept depends on **device type**

Challenge #2: Device Type

Idiosyncrasies of devices are mostly unrevealed

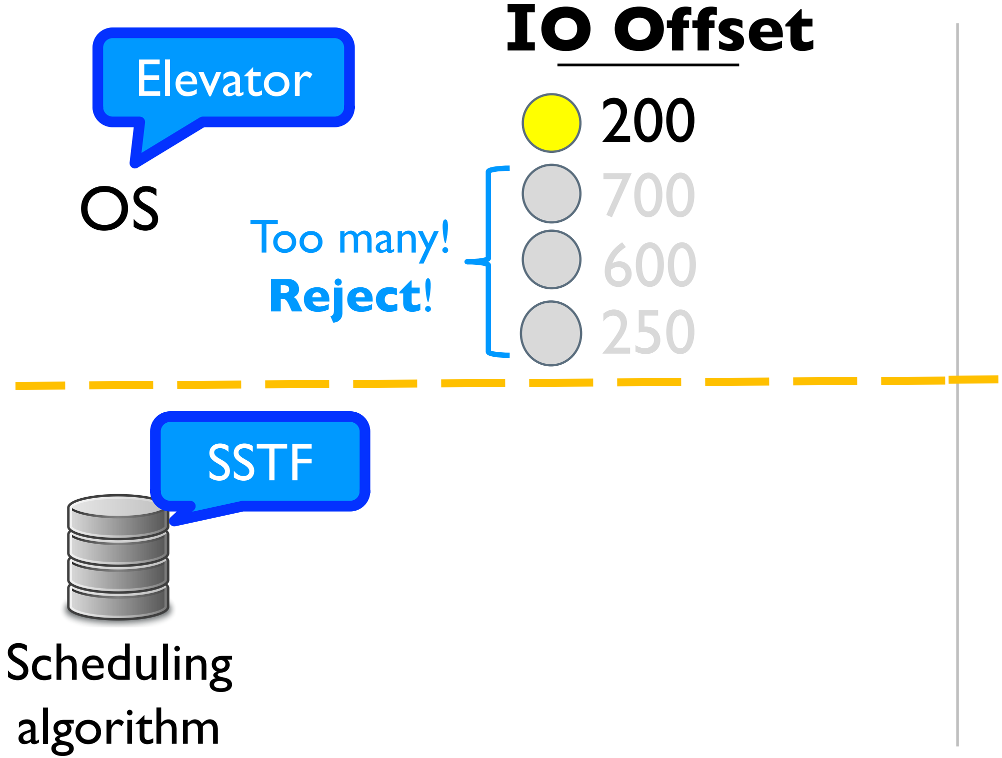
Challenge #2: Device Type

Idiosyncrasies of devices are mostly unrevealed



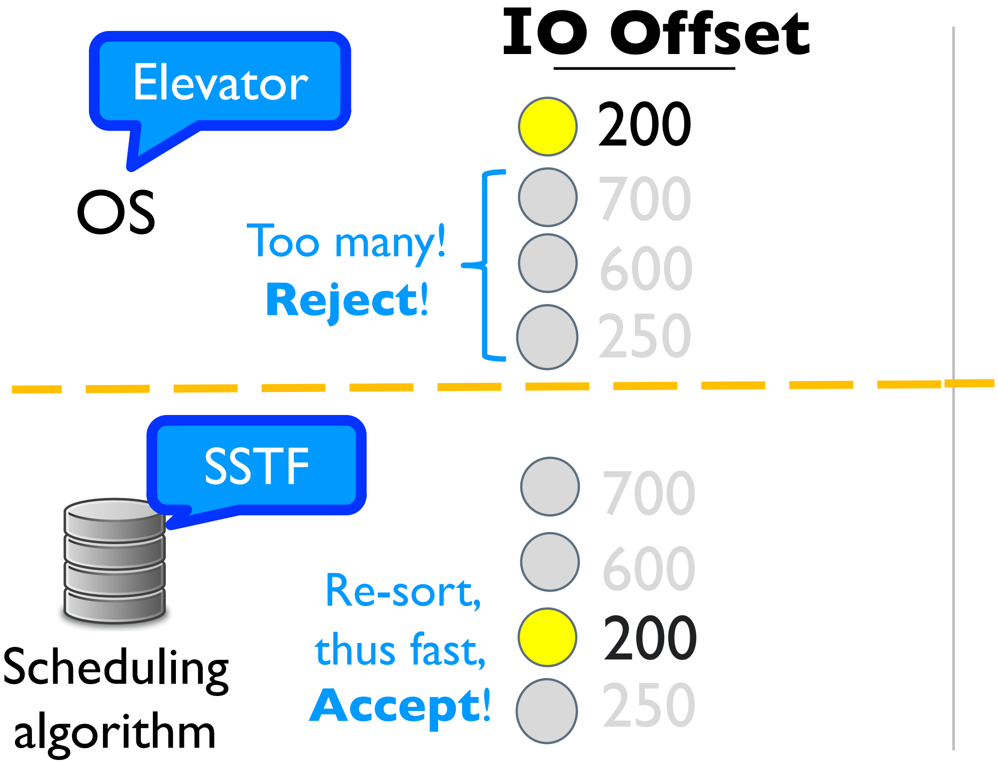
Challenge #2: Device Type

Idiosyncrasies of devices are mostly unrevealed



Challenge #2: Device Type

Idiosyncrasies of devices are mostly unrevealed



IO Offset

Elevator

OS

- 200
- 700
- 600
- 250

Too many! **Reject!**

SSTF

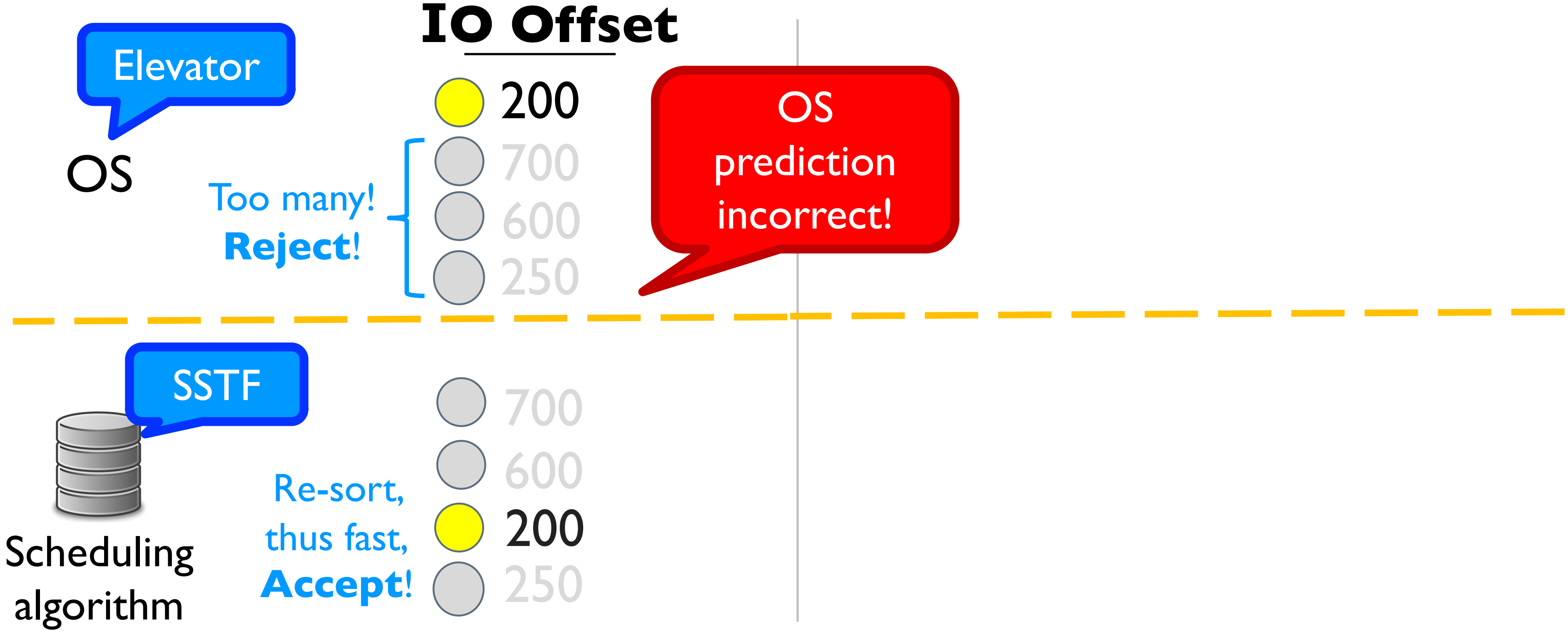
Scheduling algorithm

- 700
- 600
- 200
- 250

Re-sort, thus fast, **Accept!**

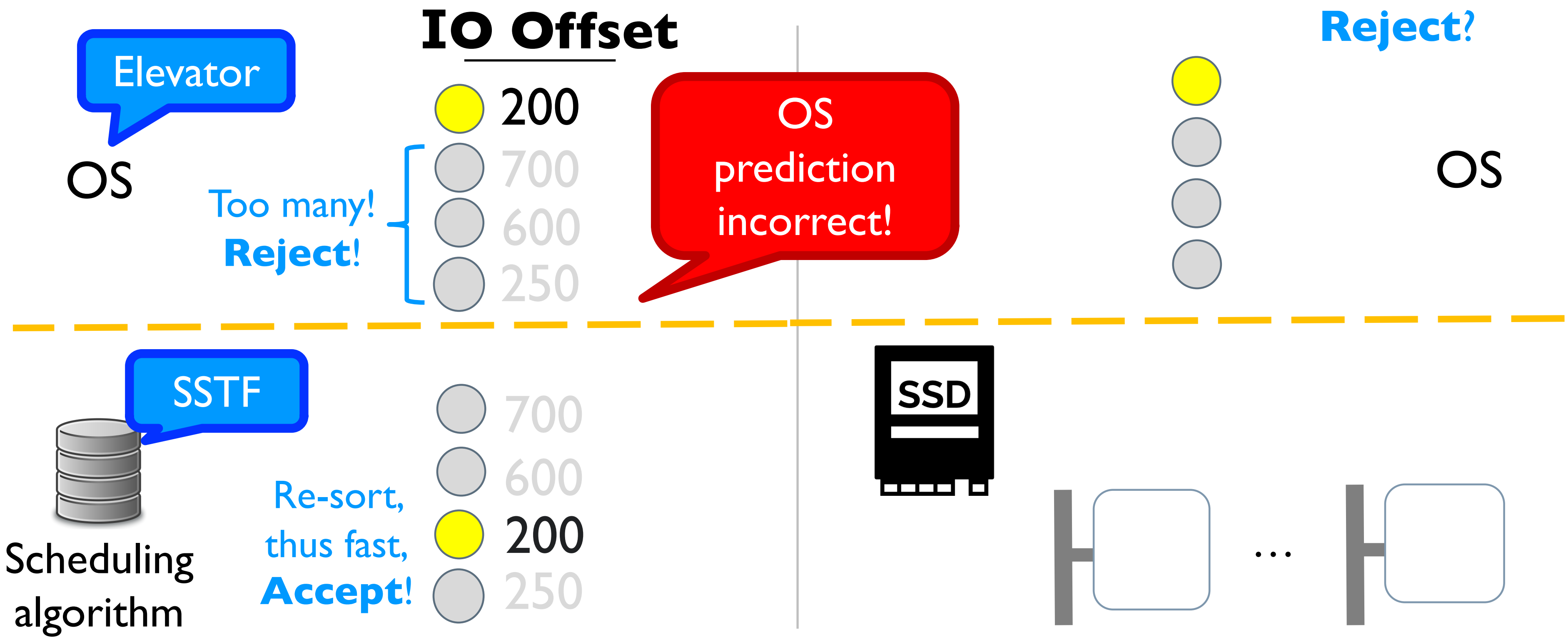
Challenge #2: Device Type

Idiosyncrasies of devices are mostly unrevealed



Challenge #2: Device Type

Idiosyncrasies of devices are mostly unrevealed



End of queue!
Reject?

IO Offset

Elevator

OS

Too many!
Reject!

200
700
600
250

OS prediction incorrect!

OS

SSTF

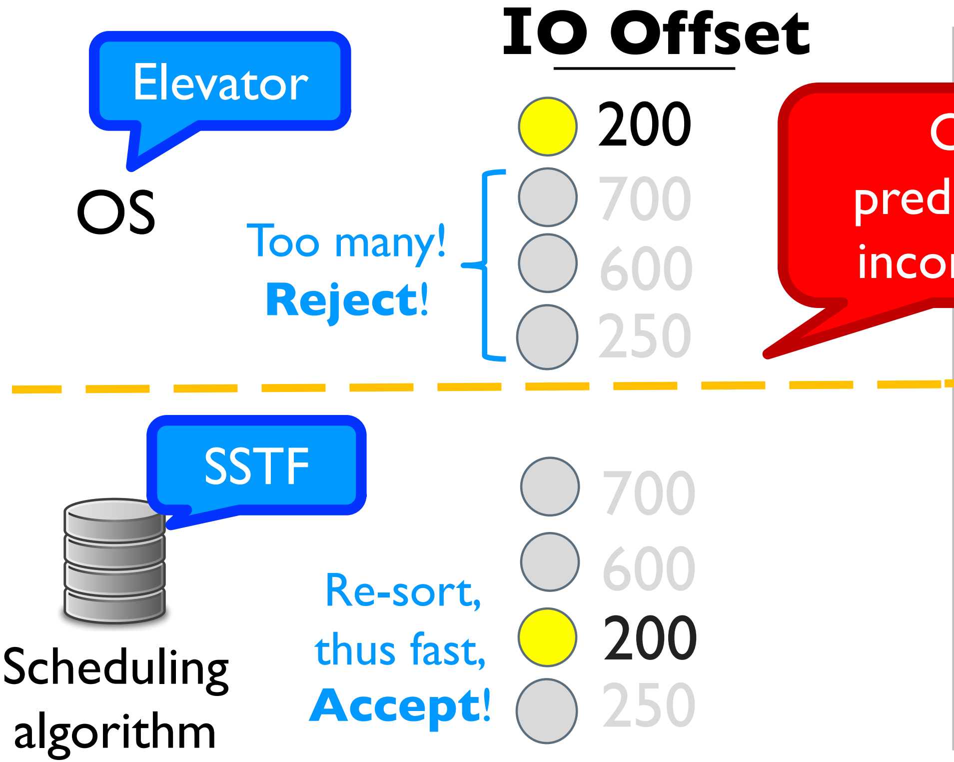
Scheduling algorithm

Re-sort,
thus fast,
Accept!

700
600
200
250

Challenge #2: Device Type

Idiosyncrasies of devices are mostly unrevealed

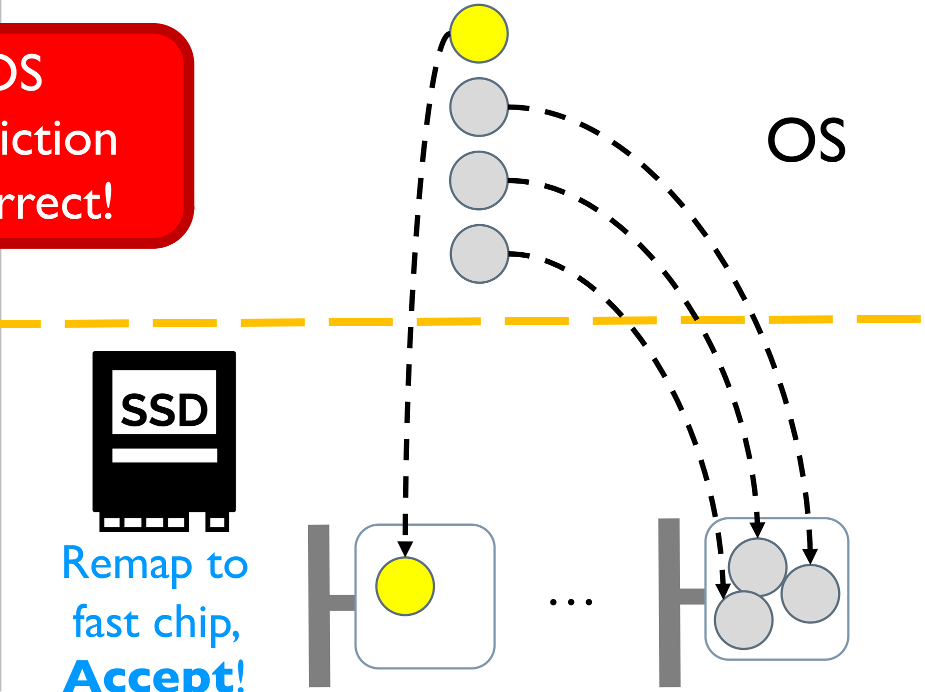


Too many! **Reject**!

OS prediction incorrect!

Re-sort, thus fast, **Accept**!

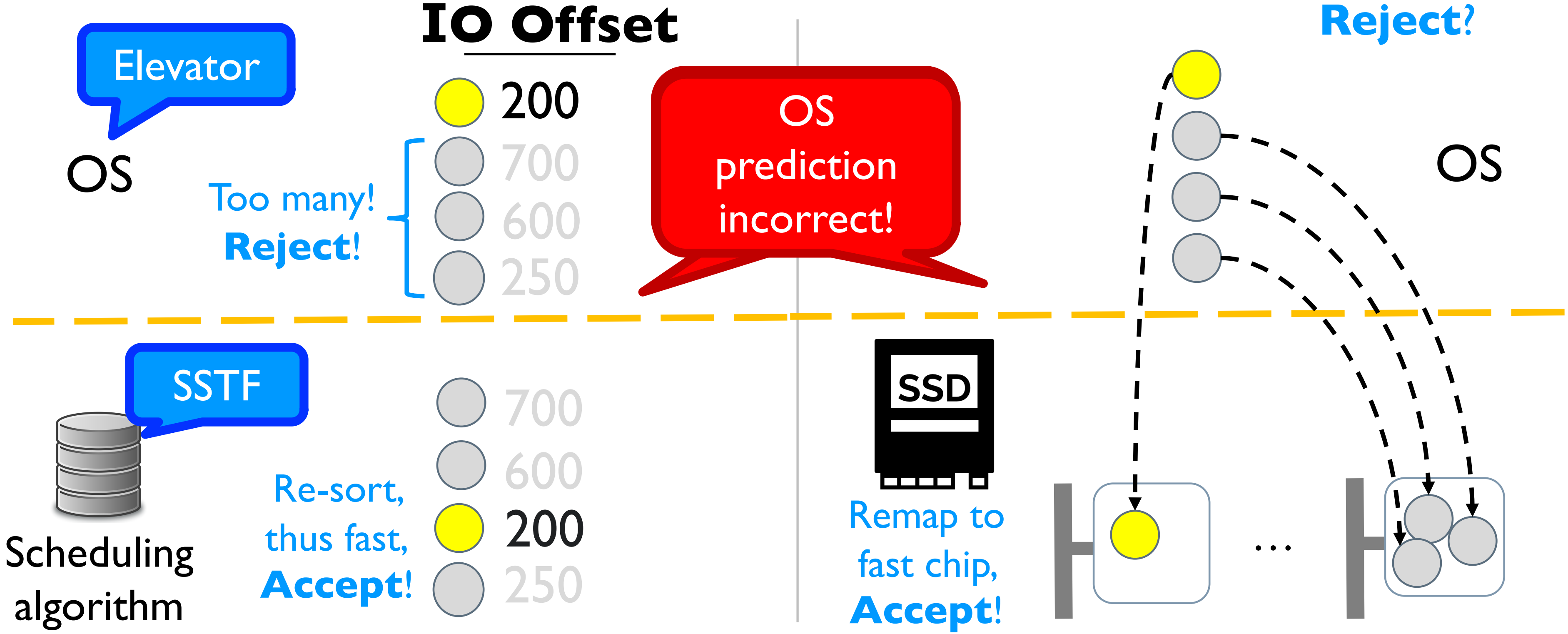
End of queue! **Reject**?



Remap to fast chip, **Accept**!

Challenge #2: Device Type

Idiosyncrasies of devices are mostly unrevealed



Outline

- ☐ Introduction
- ☐ **Design**
  - Challenges
  - Solutions
- ☐ Evaluation
- ☐ Conclusion

Reject/Latency Prediction

Reject? $= f(\qquad\qquad\qquad)$

Reject/Latency Prediction

Reject? = f(SLO,)

Reject/Latency Prediction

Reject? = f(SLO, queue policy,)

Reject/Latency Prediction

Reject? = f(SLO, queue policy, device type)

Reject/Latency Prediction

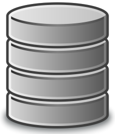
Reject? = *f* (SLO, queue policy, device type)

Get from source-code. e.g. CFQ, noop

Reject/Latency Prediction

Reject? = f(SLO, queue policy, device type)

Get from source-code. e.g. CFQ, noop



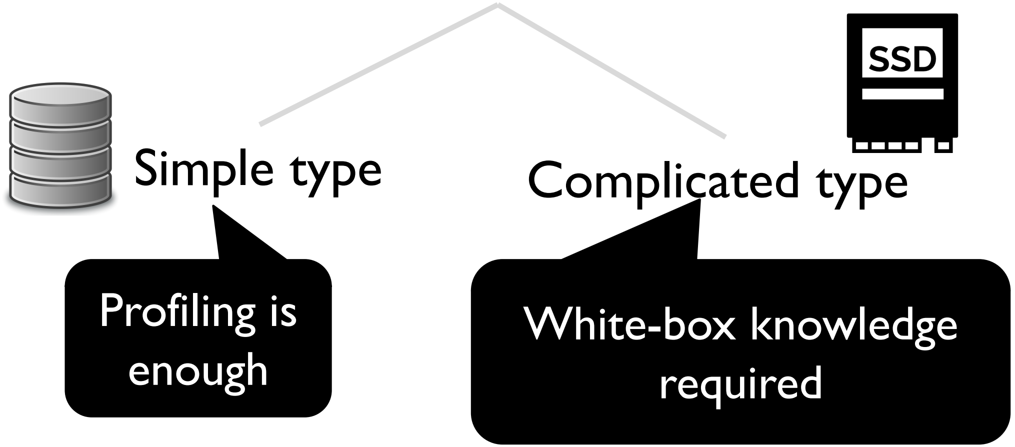
Simple type

Profiling is enough

Reject/Latency Prediction

Reject? = $f($ SLO, queue policy, device type $)$

Get from source-code. e.g. CFQ, noop



Simple type

Complicated type

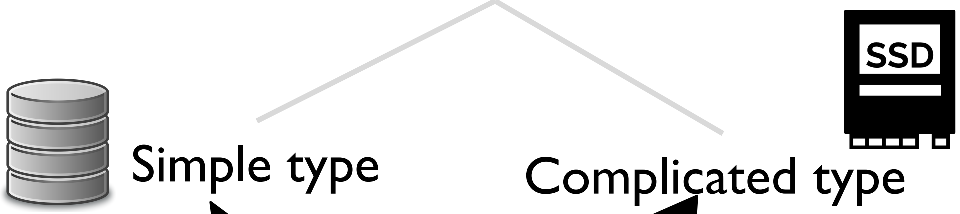
Profiling is enough

White-box knowledge required

Reject/Latency Prediction

Reject? = $f($ SLO, queue policy, device type $)$

Get from source-code. e.g. CFQ, noop



Simple type

Profiling is enough

MittCFQ

Complicated type

White-box knowledge required

MittSSD

MittCFQ

MittCFQ


Contains user group management

MittCFQ

Contains user group management
Contains 3 service trees

MittCFQ

Contains user group management
Contains 3 service trees
Contains 7 different IO priorities

CFQ

OS

MittCFQ

Contains user group management
Contains 3 service trees
Contains 7 different IO priorities
Contains ~4500 LOC

MittCFQ

Open-sourced

Contains user group management
Contains 3 service trees
Contains 7 different IO priorities
Contains ~4500 LOC

CFQ

OS

MittCFQ

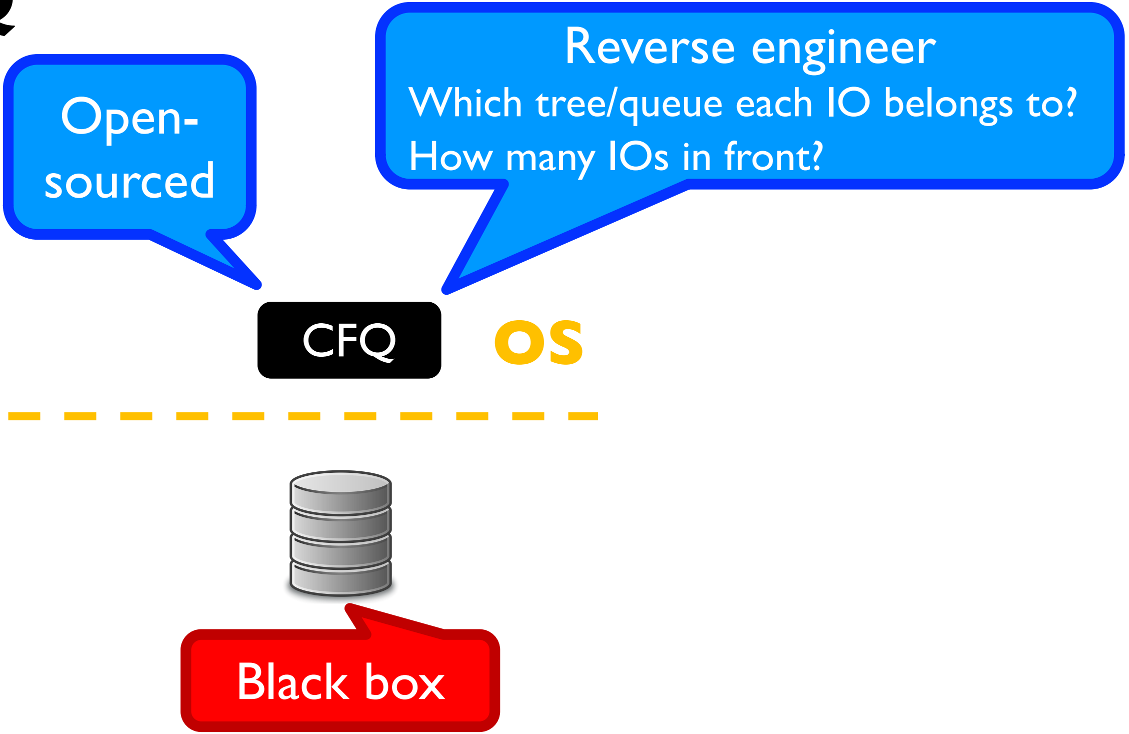
Open-sourced

Reverse engineer
Which tree/queue each IO belongs to?
How many IOs in front?

CFQ

OS

MittCFQ



Open-sourced

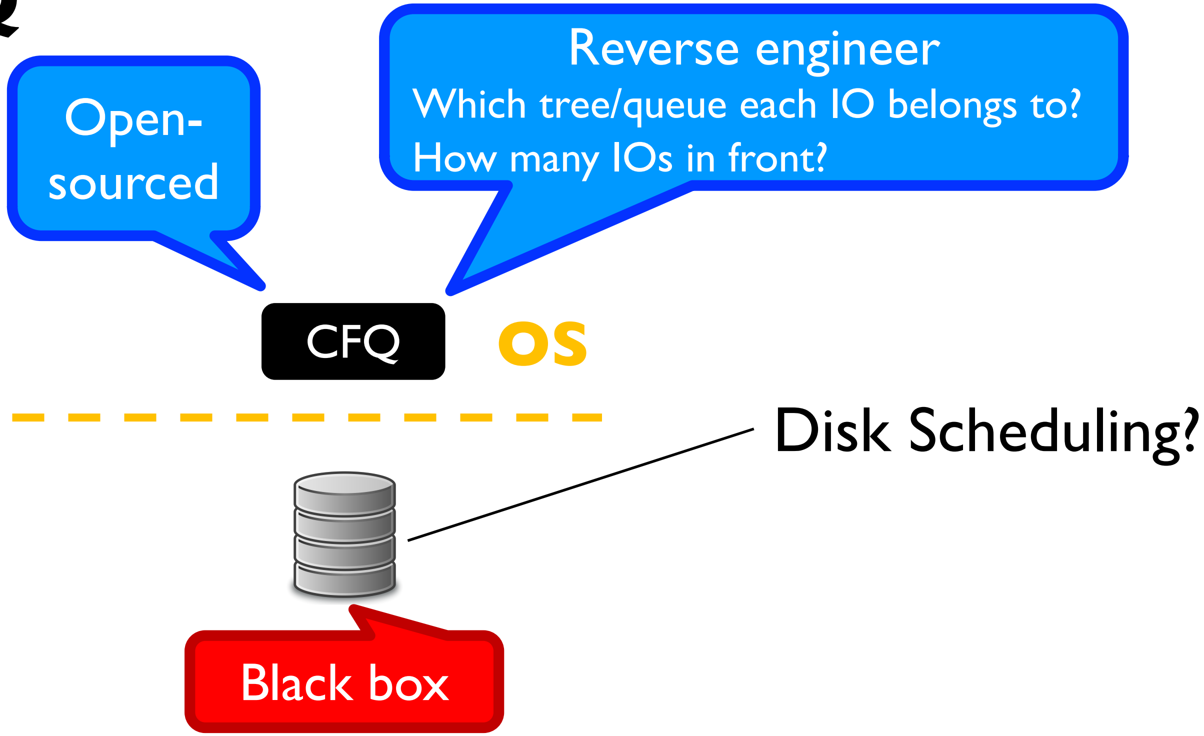
Reverse engineer
Which tree/queue each IO belongs to?
How many IOs in front?

CFQ

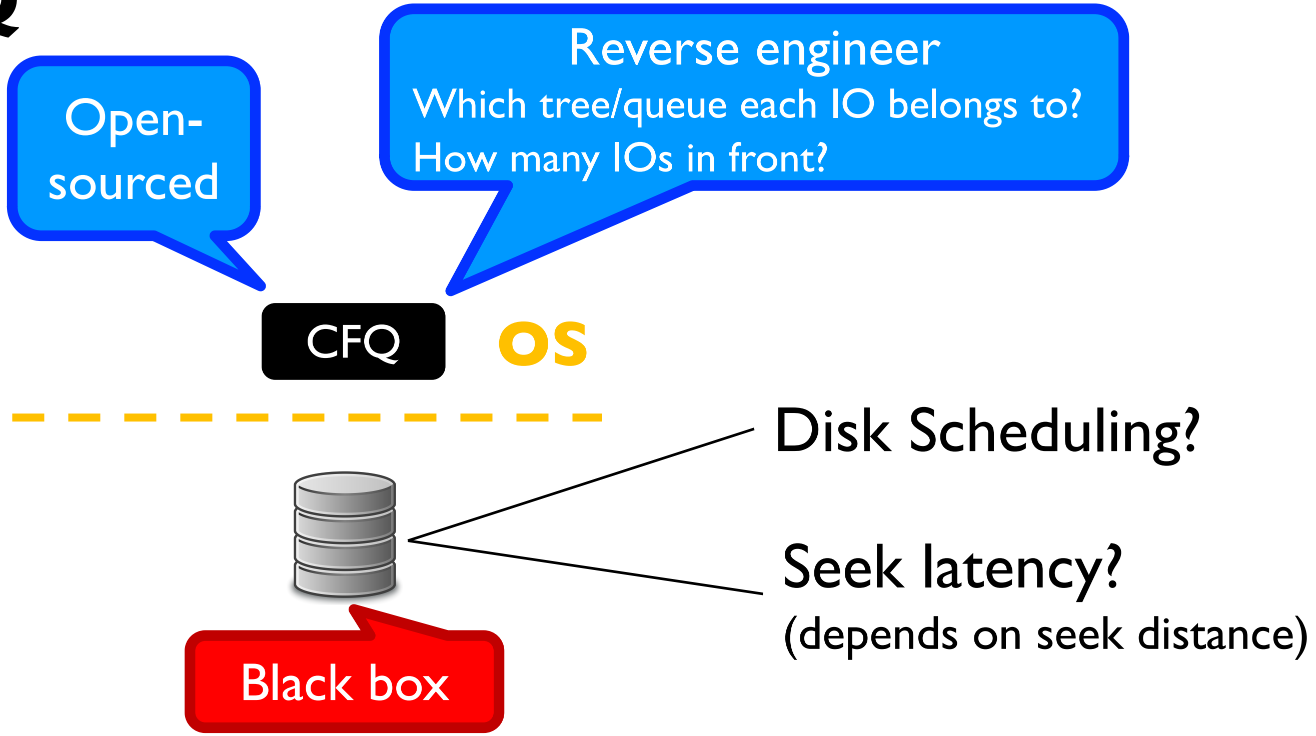
OS

Black box

MittCFQ


Open-sourced
Reverse engineer
Which tree/queue each IO belongs to?
How many IOs in front?
CFQ
OS
Disk Scheduling?
Black box

MittCFQ



Open-sourced

Reverse engineer
Which tree/queue each IO belongs to?
How many IOs in front?

CFQ

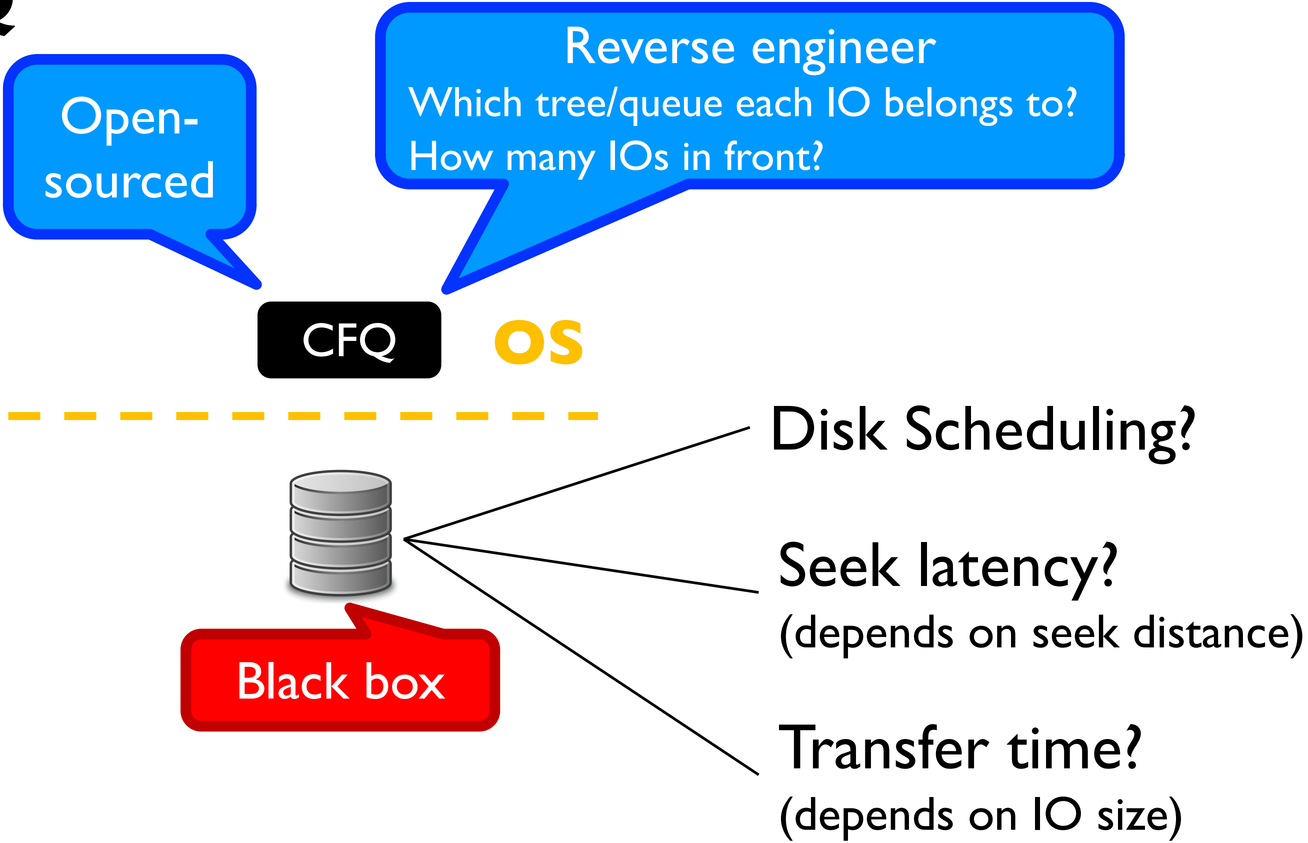
OS

Disk Scheduling?

Seek latency?
(depends on seek distance)

Black box

MittCFQ



Open-sourced

Reverse engineer

Which tree/queue each IO belongs to?

How many IOs in front?

CFQ

OS

Black box

Disk Scheduling?

Seek latency?

(depends on seek distance)

Transfer time?

(depends on IO size)

MittCFQ Profiling

```
For each interval in [ 100MB, 200MB, ..., 1GB ] do:
for (startOffset = 0; startOffset < maxOffset; startOffset += interval) {
    for (endOffset = 0; endOffset < maxOffset; endOffset += interval) {
        for (size = 0; size < maxSize; size += sizeInterval){
            start_ts = gettimeofday();
            seek (startOffset);
            read (endOffset, size);
            end_ts = gettimeofday();
            latency = start_ts - end_ts;
            print (endOffset - endOffset, size, latency);
        }
    }
}
```

MittCFQ Profiling

```
For each interval in [ 100MB, 200MB, ..., 1GB ] do:
for (startOffset = 0; startOffset < maxOffset; startOffset += interval) {
    for (endOffset = 0; endOffset < maxOffset; endOffset += interval) {
        for (size = 0; size < maxSize; size += sizeInterval){
            start_ts = gettimeofday();
            seek (startOffset)
            read (endOffset, size);
            end_ts = gettimeofday();
            latency = start_ts - end_ts;
            print (endOffset - endOffset, size, latency);
        }
    }
}
```

Random seek

MittCFQ Profiling

```
For each interval in [ 100MB, 200MB, ..., 1GB ] do:
for (startOffset = 0; startOffset < maxOffset; startOffset += interval) {
    for (endOffset = 0; endOffset < maxOffset; endOffset += interval) {
        for (size = 0; size < maxSize; size += sizeInterval){
            start_ts = gettimeofday();
            seek (startOffset)
            read (endOffset, size);
            end_ts = gettimeofday();
            latency = start_ts - end_ts;
            print (endOffset - endOffset, size, latency);
        }
    }
}
```

Random seek

Random read

MittCFQ Profiling

```
For each interval in [ 100MB, 200MB, ..., 1GB ] do:
for (startOffset = 0; startOffset < maxOffset; startOffset += interval) {
    for (endOffset = 0; endOffset < maxOffset; endOffset += interval) {
        for (size = 0; size < maxSize; size += sizeInterval){
            start_ts = gettimeofday();
            seek (startOffset)
            read (endOffset, size);
            end_ts = gettimeofday();
            latency = start_ts - end_ts;
            print (endOffset - endOffset, size, latency);
        }
    }
}
```

Random seek

Random read

Collect latency

MittCFQ Profiling

```
For each interval in [ 100MB, 200MB, ..., 1GB ] do:
for (startOffset = 0; startOffset < maxOffset; startOffset += interval) {
    for (endOffset = 0; endOffset < maxOffset; endOffset += interval) {
        for (size = 0; size < maxSize; size += sizeInterval){
            start_ts = gettimeofday();
            seek (startOffset);
            read (endOffset, size);
            end_ts = gettimeofday();
            latency = start_ts - end_ts;
            print (endOffset - endOffset, size, latency);
        }
    }
}
```

Random seek

Random read

Collect latency

2 disk models
11-hour profiling

MittCFQ Profiling

```
For each interval in [ 100MB, 200MB, ..., 1GB ] do:
for (startOffset = 0; startOffset < maxOffset; startOffset += interval) {
    for (endOffset = 0; endOffset < maxOffset; endOffset += interval) {
        for (size = 0; size < maxSize; size += sizeInterval){
            start_ts = gettimeofday();
            seek (startOffset);
            read (endOffset, size);
            end_ts = gettimeofday();
            latency = start_ts - end_ts;
            print (endOffset - endOffset, size, latency);
        }
    }
}
```

Random seek

Random read

Collect latency

scikit-learn

Linear Regression

2 disk models

11-hour profiling

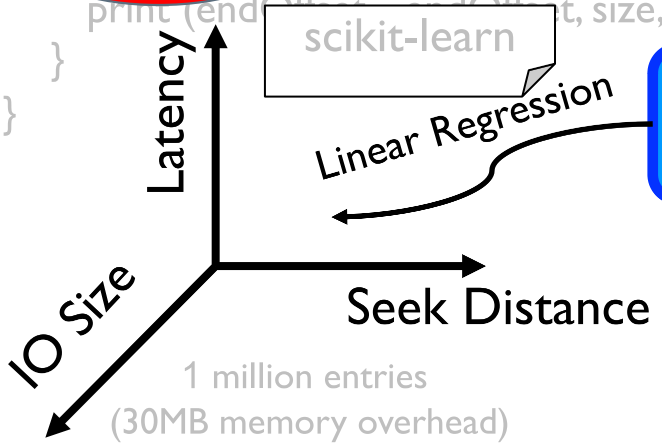
MittCFQ Profiling

```
For each interval in [ 100MB, 200MB, ..., 1GB ] do:
for (startOffset = 0; startOffset < maxOffset; startOffset += interval) {
    for (endOffset = 0; endOffset < maxOffset; endOffset += interval) {
        for (size = 0; size < maxSize; size += sizeInterval){
            start_ts = gettimeofday();
            seek (startOffset)
            read (endOffset, size);
            end_ts = gettimeofday();
            latency = start_ts - end_ts;
            print (endOffset – endOffset, size, latency);
        }
    }
}
```

Random seek

Random read

Collect latency



2 disk models
11-hour profiling

1 million entries
(30MB memory overhead)
For 1TB drive

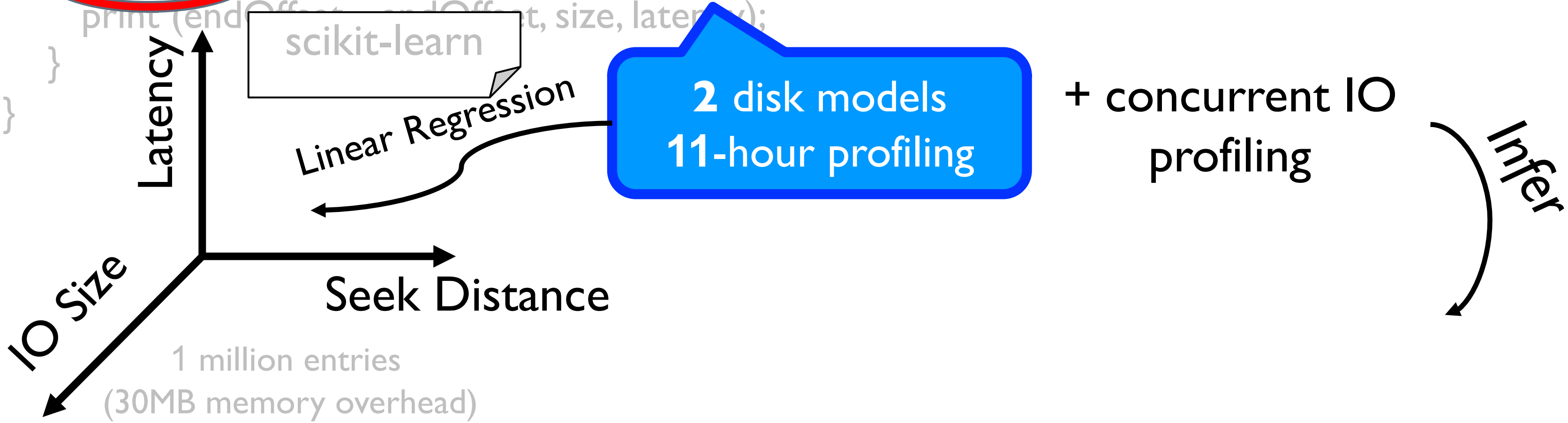
MittCFQ Profiling

```
For each interval in [ 100MB, 200MB, ..., 1GB ] do:
for (startOffset = 0; startOffset < maxOffset; startOffset += interval) {
    for (endOffset = 0; endOffset < maxOffset; endOffset += interval) {
        for (size = 0; size < maxSize; size += sizeInterval){
            start_ts = gettimeofday();
            seek (startOffset)
            read (endOffset, size);
            end_ts = gettimeofday();
            latency = start_ts - end_ts;
            print (end          t, size, later    );
        }
    }
}
```

Random seek

Random read

Collect latency



2 disk models
11-hour profiling

+ concurrent IO profiling

1 million entries
(30MB memory overhead)
For 1TB drive

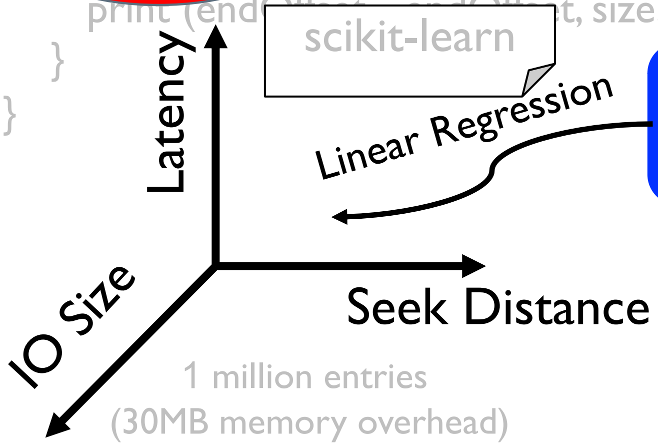
MittCFQ Profiling

```
For each interval in [ 100MB, 200MB, ..., 1GB ] do:
for (startOffset = 0; startOffset < maxOffset; startOffset += interval) {
    for (endOffset = 0; endOffset < maxOffset; endOffset += interval) {
        for (size = 0; size < maxSize; size += sizeInterval){
            start_ts = gettimeofday();
            seek (startOffset)
            read (endOffset, size);
            end_ts = gettimeofday();
            latency = start_ts - end_ts;
            print (endOffset, endOffset, size, latency);
        }
    }
}
```

Random seek

Random read

Collect latency



1 million entries
(30MB memory overhead)
For 1TB drive

2 disk models
11-hour profiling

+ concurrent IO profiling

Infer

SSTF scheduling

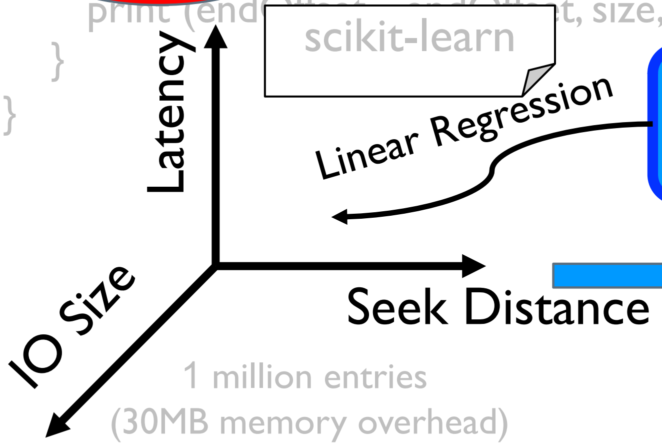
MittCFQ Profiling

```
For each interval in [ 100MB, 200MB, ..., 1GB ] do:
for (startOffset = 0; startOffset < maxOffset; startOffset += interval) {
    for (endOffset = 0; endOffset < maxOffset; endOffset += interval) {
        for (size = 0; size < maxSize; size += sizeInterval){
            start_ts = gettimeofday();
            seek (startOffset)
            read (endOffset, size);
            end_ts = gettimeofday();
            latency = start_ts - end_ts;
            print (end        t, size, laten   y);
        }
    }
}
```

Random seek

Random read

Collect latency



1 million entries
(30MB memory overhead)
For 1TB drive

2 disk models
11-hour profiling

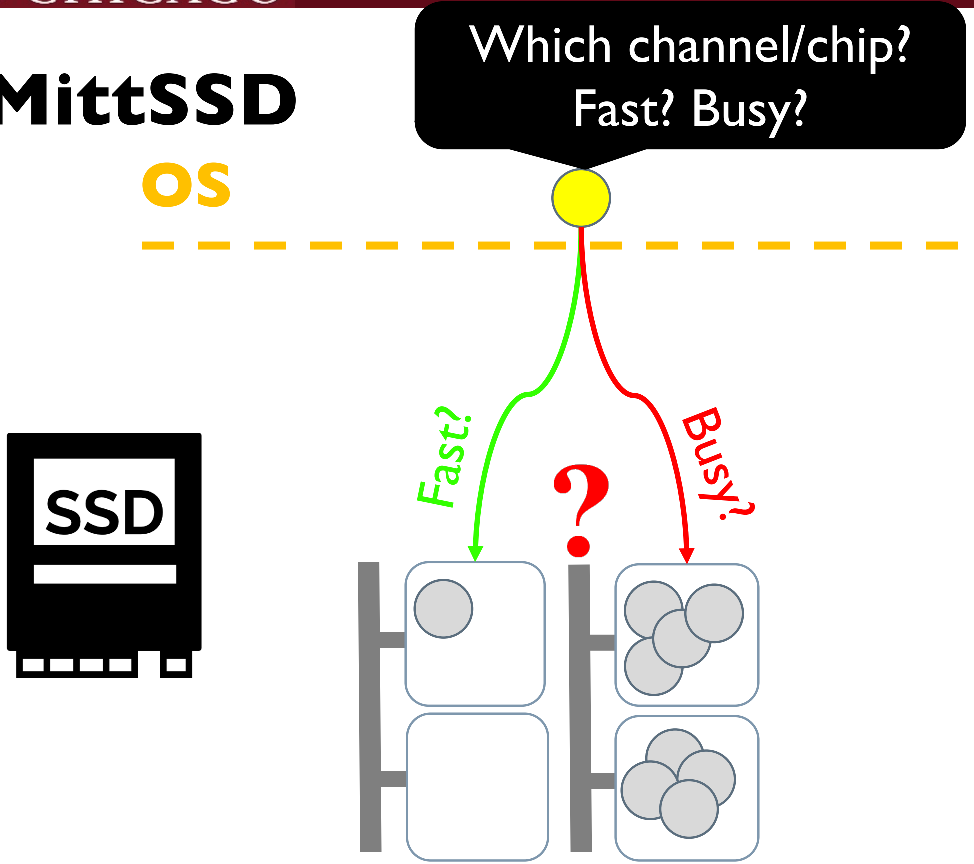
+ concurrent IO profiling

Infer

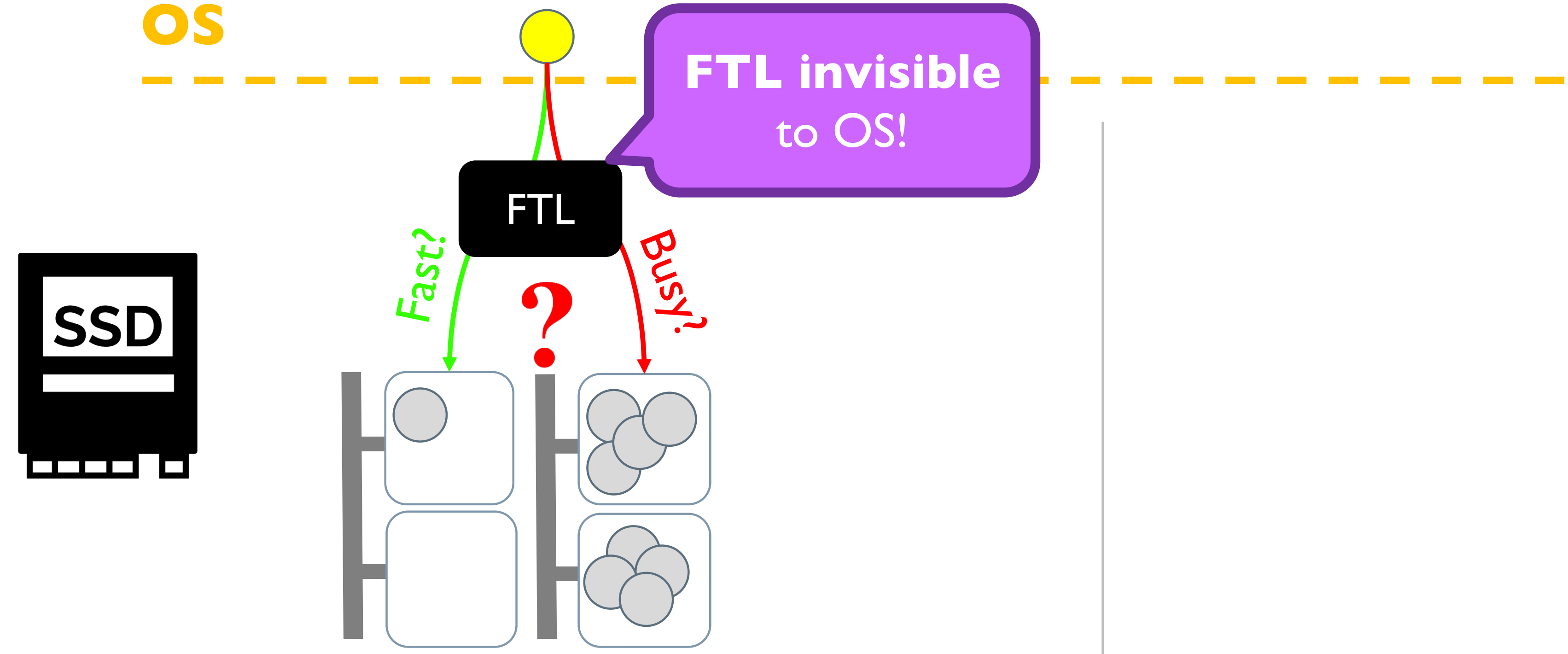
SSTF scheduling

Accurate prediction

MittSSD

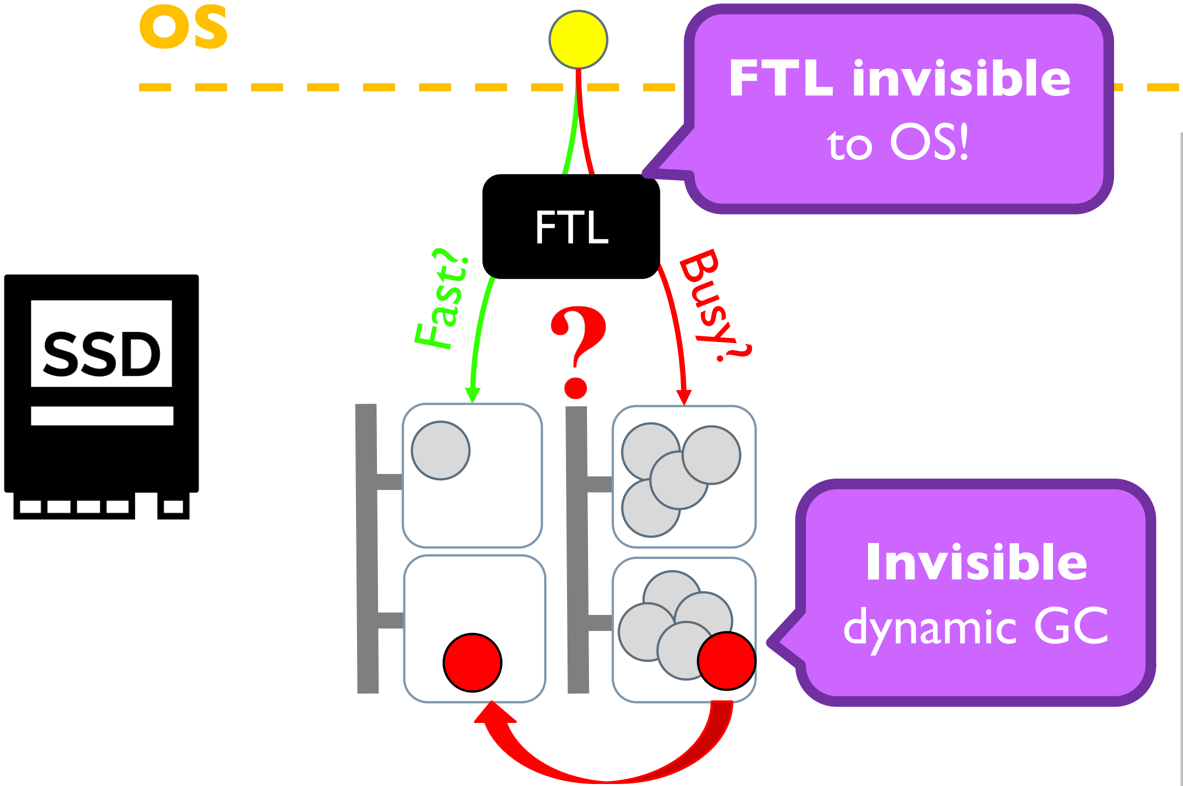


MittSSD



FTL invisible to OS!

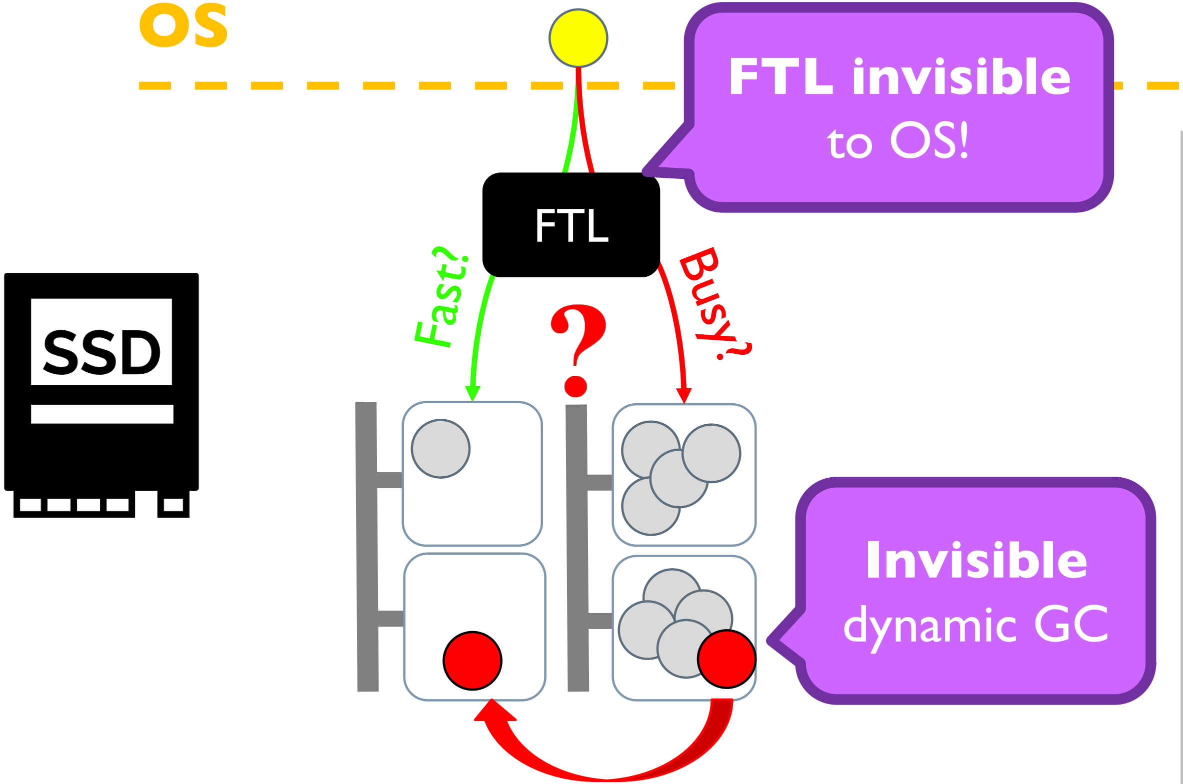
MittSSD



FTL invisible to OS!

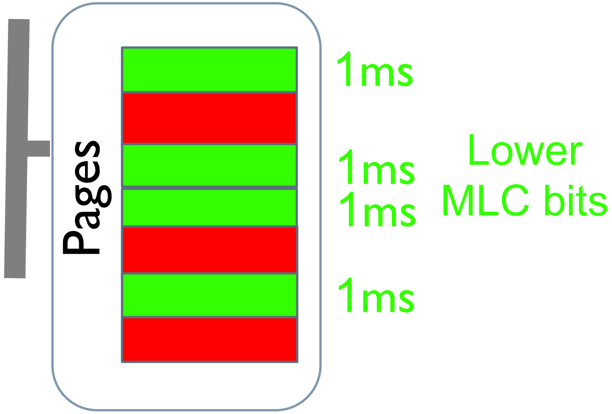
Invisible dynamic GC

MittSSD

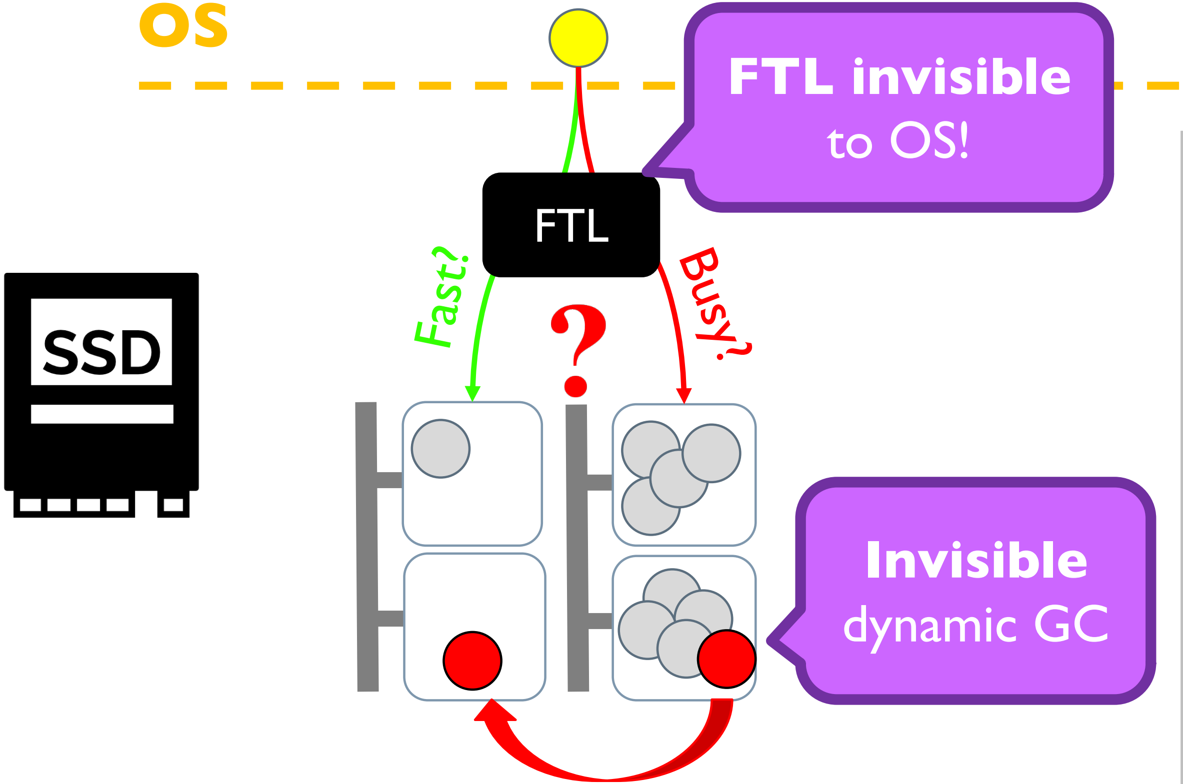


FTL invisible to OS!

Invisible dynamic GC


Write lat. variability

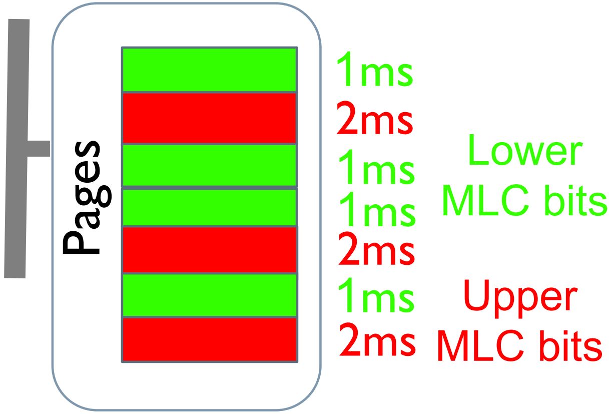
MittSSD



FTL invisible to OS!

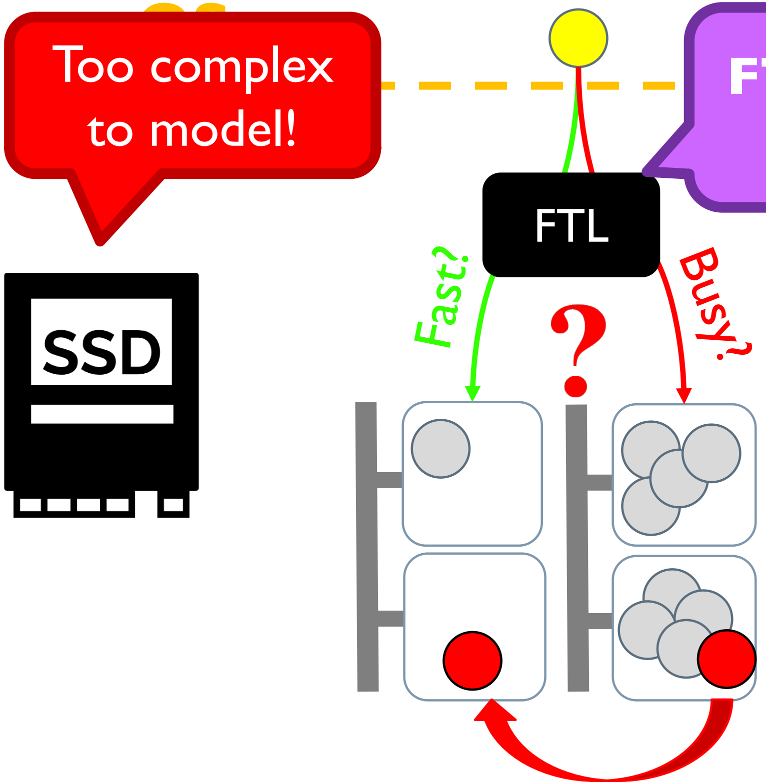
Invisible dynamic GC

Write lat. variability



MittSSD

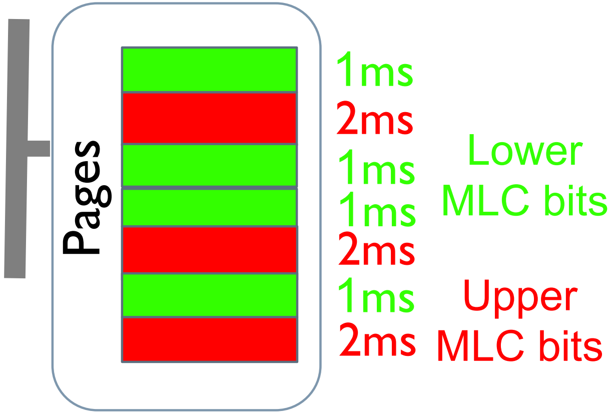
Too complex to model!



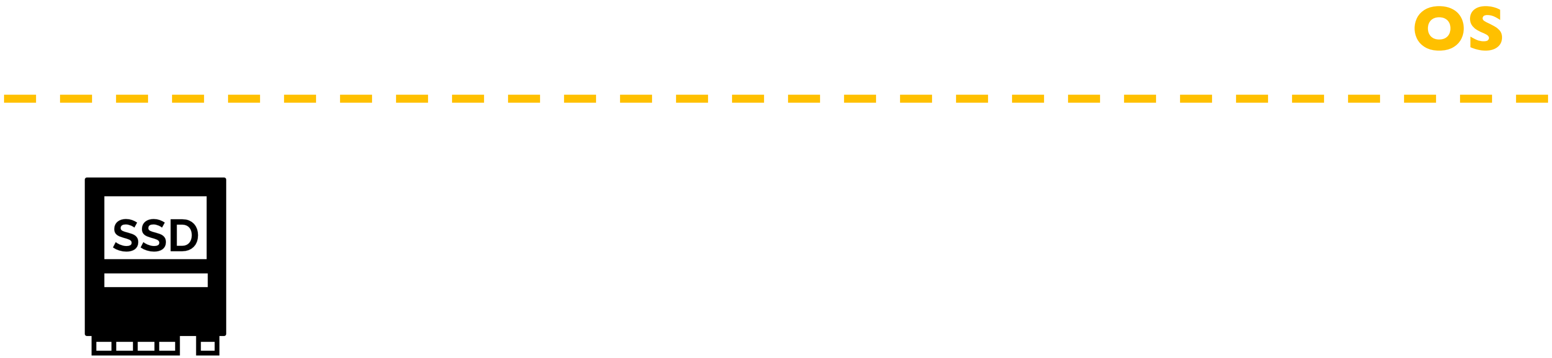
FTL invisible to OS!

Invisible dynamic GC

Write lat. variability

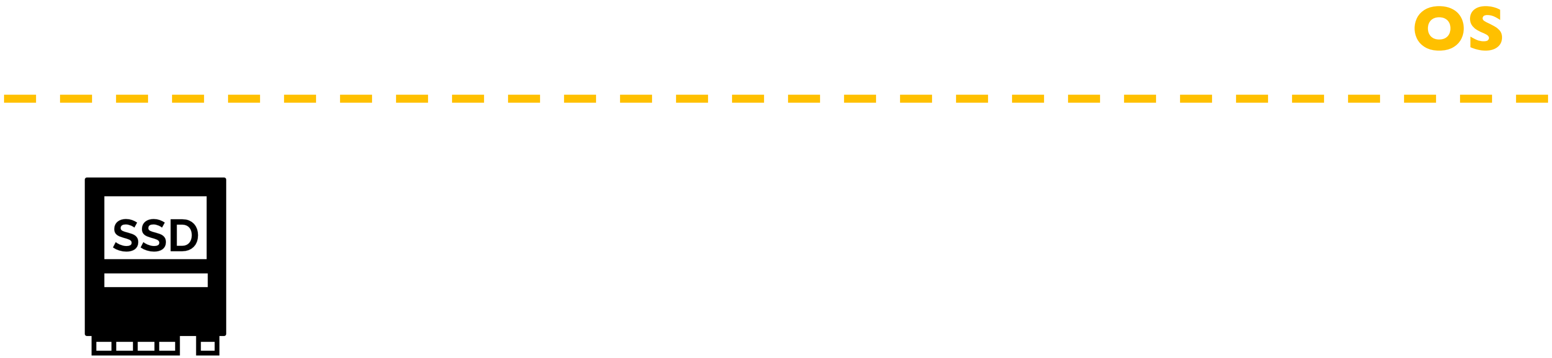


MittSSD



MittSSD

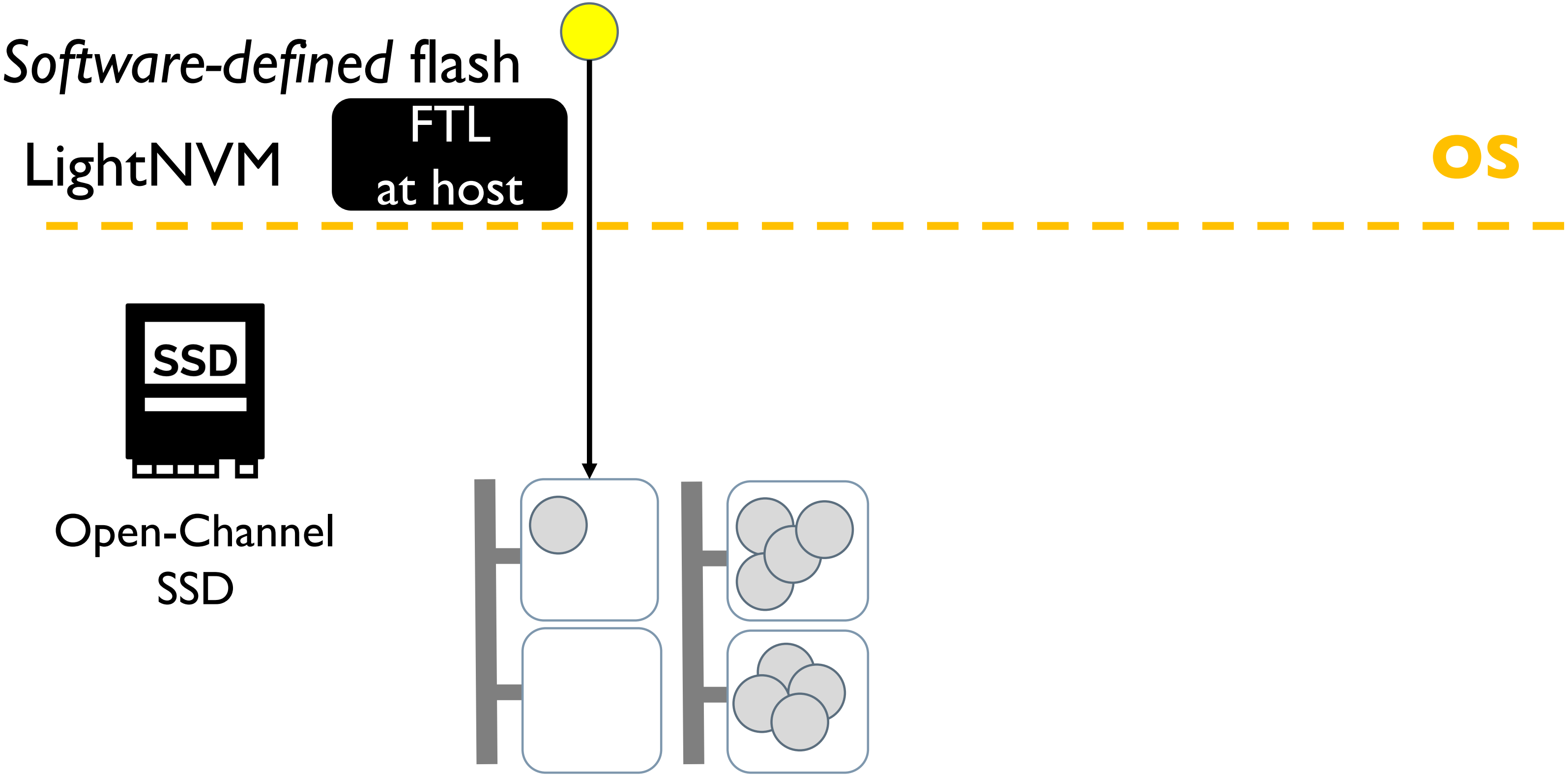
Software-defined flash



OS

MittSSD

Software-defined flash



MittSSD

Software-defined flash

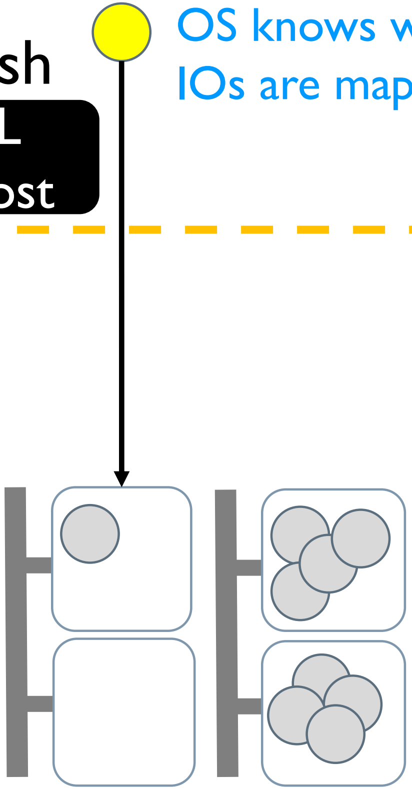
LightNVM

FTL at host

OS knows where IOs are mapped

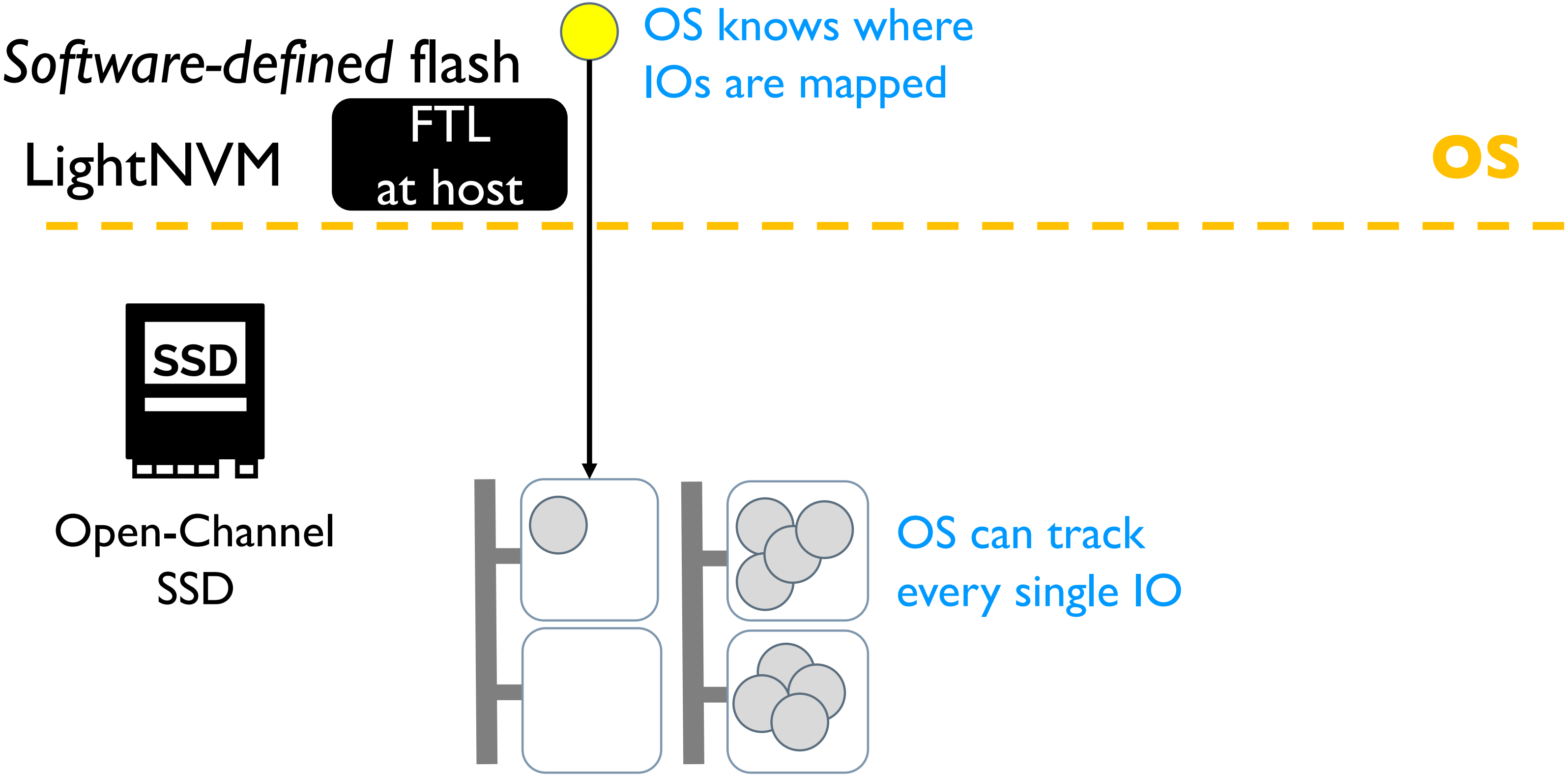
OS

Open-Channel SSD



MittSSD

Software-defined flash



OS knows where IOs are mapped

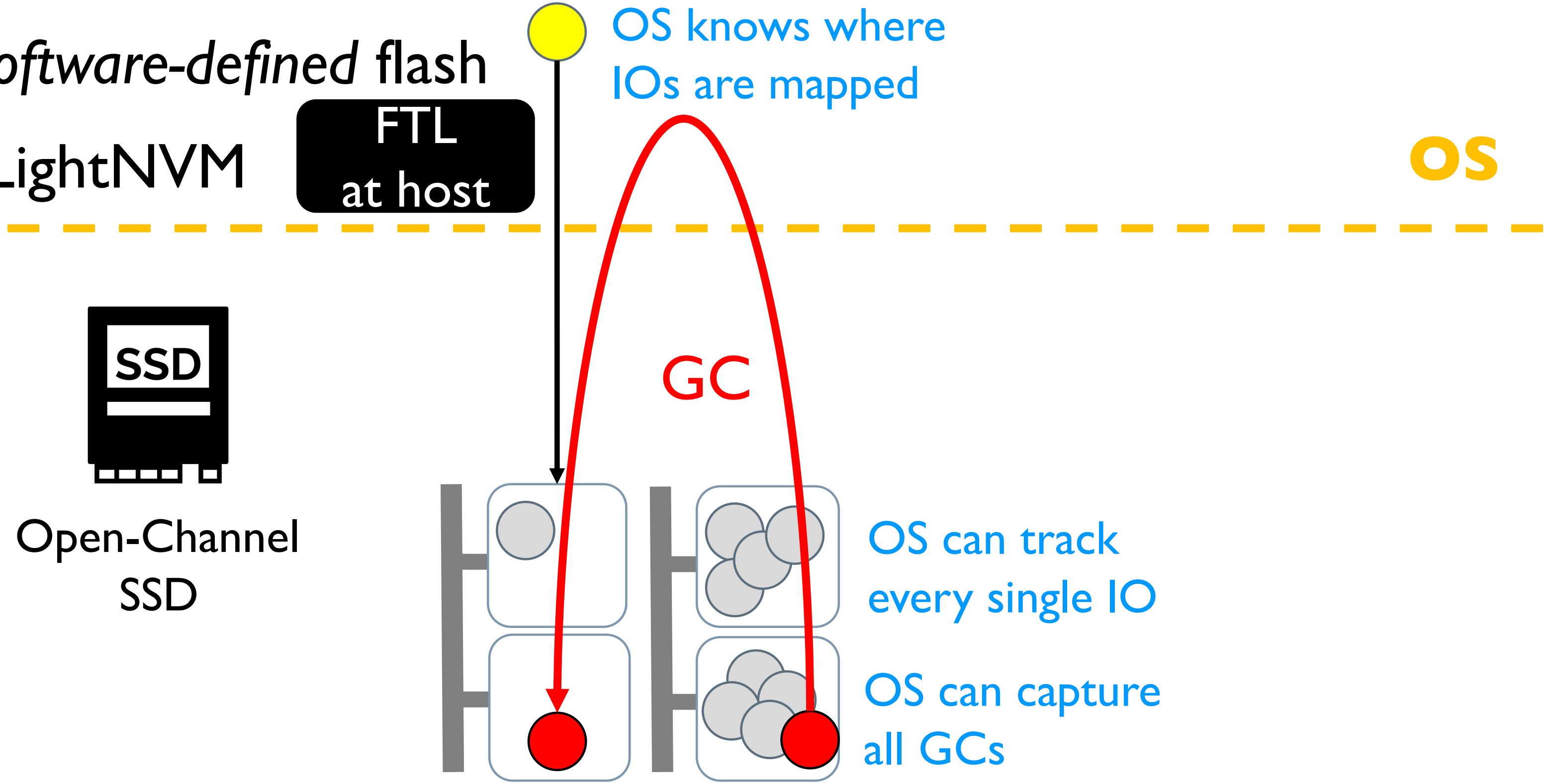
OS can track every single IO

MittSSD

Software-defined flash

LightNVM

FTL at host



OS knows where IOs are mapped

Open-Channel SSD

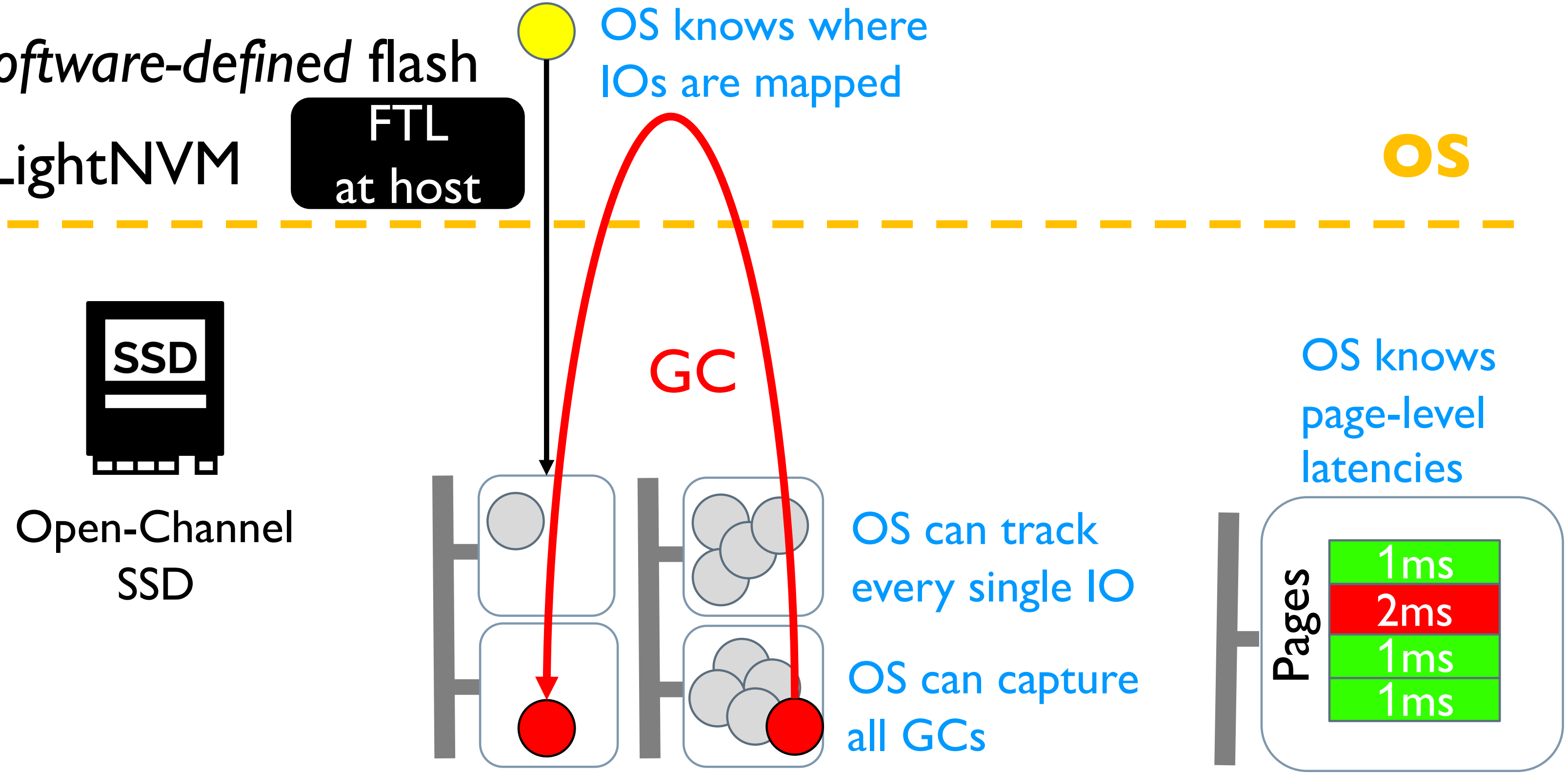
OS can track every single IO

OS can capture all GCs

MittSSD

Software-defined flash

LightNVM



OS knows where IOs are mapped

OS can track every single IO

OS can capture all GCs

OS knows page-level latencies

MittSSD

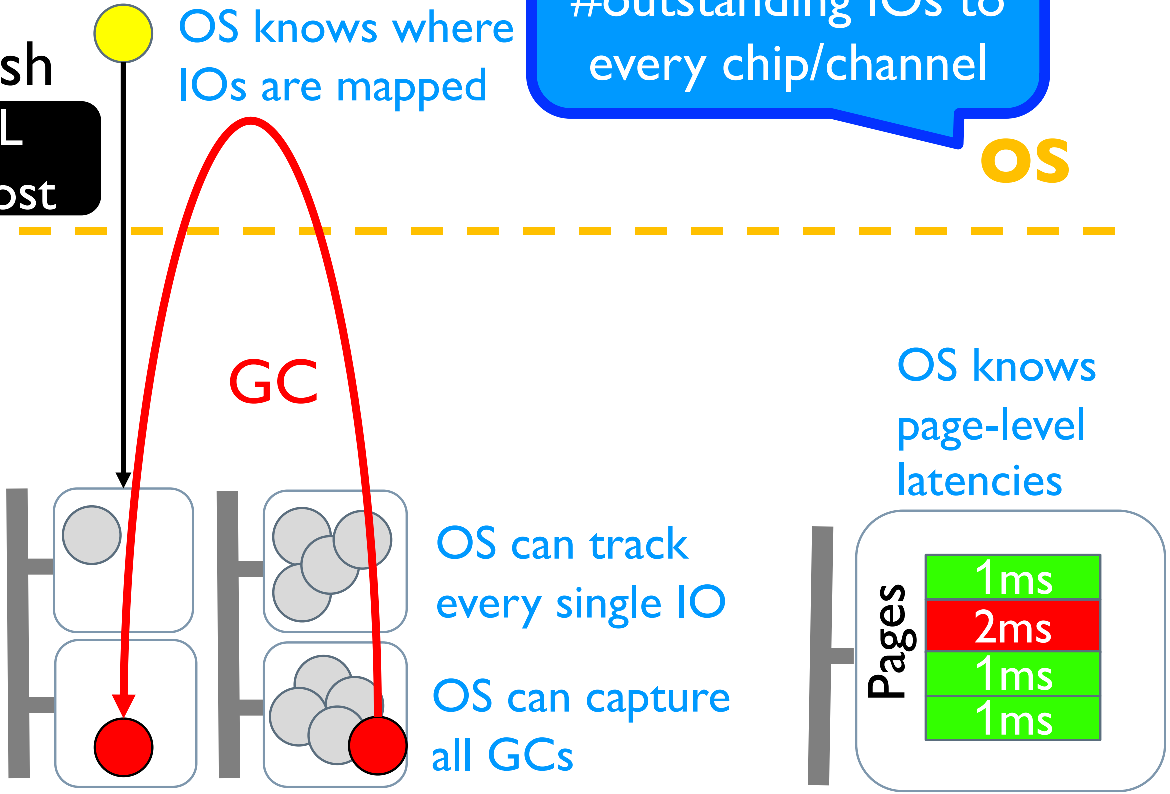
Software-defined flash

LightNVM

FTL
at host

OS

Open-Channel
SSD



OS knows where
IOs are mapped

OS can see
#outstanding IOs to
every chip/channel

OS can track
every single IO

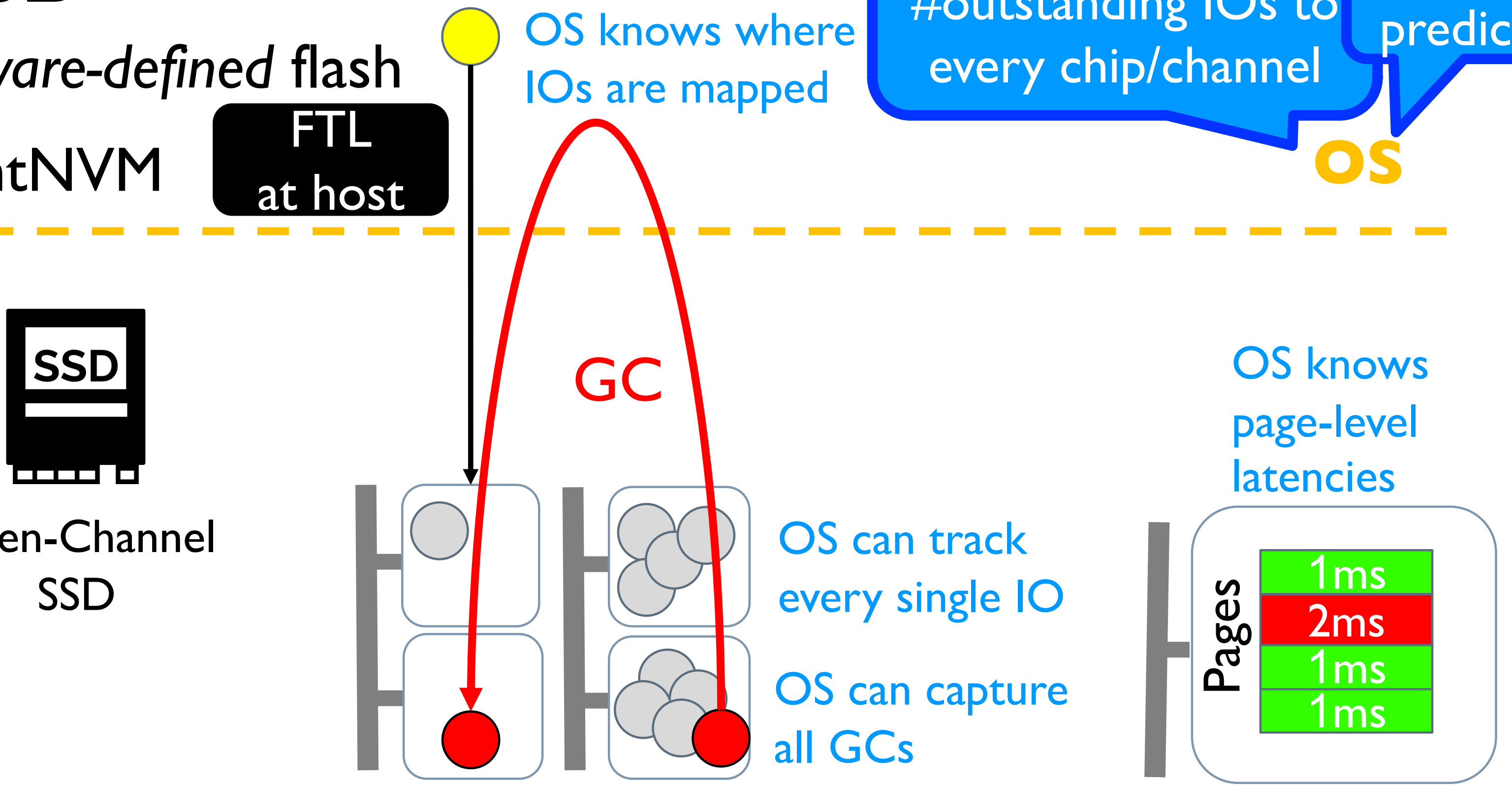
OS can capture
all GCs

OS knows
page-level
latencies

MittSSD

Software-defined flash

LightNVM



OS knows where IOs are mapped

OS can see #outstanding IOs to every chip/channel

Accurate prediction

Open-Channel SSD

OS can track every single IO

OS can capture all GCs

OS knows page-level latencies

Reject/Latency Prediction

Reject? = f(SLO, queue policy, device type)

Reverse engineering based on source code

Simple type

Complicated type

Profiling is enough

White-box knowledge required

Reject/Latency Prediction

Reject? = $f($ SLO, queue policy, device type $)$

Reverse engineering based on source code

device type:

- Simple type: Profiling is enough
- Complicated type: White-box knowledge required

SSD

MittCFQ ~**1800 LOC**

Reject/Latency Prediction

Reject? = $f($ SLO, queue policy, device type $)$

Reverse engineering based on source code

Simple type

Profiling is enough

MittCFQ ~**1800 LOC**

Complicated type

White-box knowledge required

MittSSD ~**1400 LOC**

LightNVM +
Open-Channel SSD

Other Solved Challenges

Other Solved Challenges

- Prediction overhead optimizations

Other Solved Challenges

- Prediction overhead optimizations
  - Avoids going through every IO in the queue

Other Solved Challenges

- Prediction overhead optimizations
  - Avoids going through every IO in the queue
  - Reduces overhead from O(n) to roughly O(1)

Other Solved Challenges

- Prediction overhead optimizations
  - Avoids going through every IO in the queue
  - Reduces overhead from O(n) to roughly O(1)
  - Shows < 5μs overhead for MittCFQ prediction
  - < 300ns for MittSSD prediction

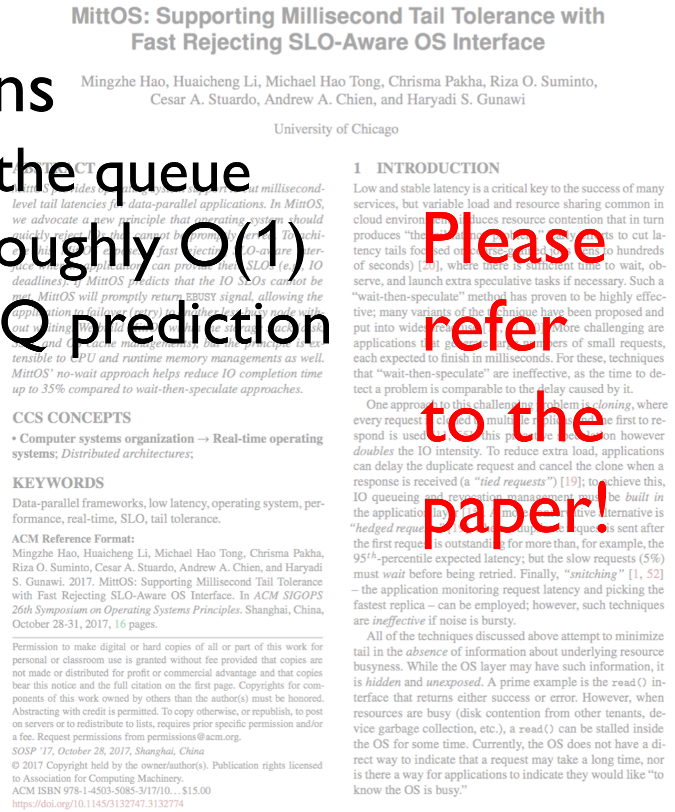
Other Solved Challenges

- Prediction overhead optimizations
  - Avoids going through every IO in the queue
  - Reduces overhead from O(n) to roughly O(1)
  - Shows < 5µs overhead for MittCFQ prediction
  - < 300ns for MittSSD prediction

- MittCache
  - Prediction for OS Cache

Other Solved Challenges

- Prediction overhead optimizations
  - Avoids going through every IO in the queue
  - Reduces overhead from O(n) to roughly O(1)
  - Shows < 5µs overhead for MittCFQ prediction
  - < 300ns for MittSSD prediction
- MittCache
  - Prediction for OS Cache



MittOS: Supporting Millisecond Tail Tolerance with Fast Rejecting SLO-Aware OS Interface

Mingzhe Hao, Huaicheng Li, Michael Hao Tong, Chrisma Pakha, Riza O. Suminto,
Cesar A. Stuardo, Andrew A. Chien, and Haryadi S. Gunawi

University of Chicago

ABSTRACT

MittOS provides operating system support to cut millisecond-level tail latencies for data-parallel applications. In MittOS, we advocate a new principle that operating system should quickly reject IOs that cannot be promptly served. To achieve this, MittOS exposes a fast rejecting SLO-aware interface where applications can provide their SLOs (e.g., IO deadlines). If MittOS predicts that the IO SLOs cannot be met, MittOS will promptly return EBUSY *signal, allowing the application to failover (retry) to another less-busy node without waiting. We build MittOS within the storage stack (disk, SSD, and OS cache managements), but the principle is extensible to CPU and runtime memory managements as well. MittOS' no-wait approach helps reduce IO completion time up to 35% compared to wait-then-speculate approaches.*

CCS CONCEPTS

• **Computer systems organization → Real-time operating systems**; *Distributed architectures*;

KEYWORDS

Data-parallel frameworks, low latency, operating system, performance, real-time, SLO, tail tolerance.

ACM Reference Format:
Mingzhe Hao, Huaicheng Li, Michael Hao Tong, Chrisma Pakha, Riza O. Suminto, Cesar A. Stuardo, Andrew A. Chien, and Haryadi S. Gunawi. 2017. MittOS: Supporting Millisecond Tail Tolerance with Fast Rejecting SLO-Aware OS Interface. In *ACM SIGOPS 26th Symposium on Operating Systems Principles*. Shanghai, China, October 28-31, 2017, 16 pages.

SOSP '17, October 28, 2017, Shanghai, China

ACM ISBN 978-1-4503-5085-3/17/10…$15.00
https://doi.org/10.1145/3132747.3132774

1 INTRODUCTION

Low and stable latency is a critical key to the success of many services, but variable load and resource sharing common in cloud environments induces resource contention that in turn produces "the tail latency problem." Early efforts to cut latency tails focused on coarse-grained jobs (tens to hundreds of seconds) [20], where there is sufficient time to wait, observe, and launch extra speculative tasks if necessary. Such a "wait-then-speculate" method has proven to be highly effective; many variants of the technique have been proposed and put into widespread use [[illegible]]. More challenging are applications that generate large numbers of small requests, each expected to finish in milliseconds. For these, techniques that "wait-then-speculate" are ineffective, as the time to detect a problem is comparable to the delay caused by it.

One approach to this challenging problem is *cloning*, where every request is cloned to multiple replicas and the first to respond is used [11, 55]; this proactive speculation however *doubles* the IO intensity. To reduce extra load, applications can delay the duplicate request and cancel the clone when a response is received (a *"tied requests"*) [19]; to achieve this, IO queueing and revocation management must be *built in* the application layer [15]. A more conservative alternative is *"hedged requests"* [19]; the duplicate request is sent after the first request is outstanding for more than, for example, the 95^{th}-percentile expected latency; but the slow requests (5%) must *wait* before being retried. Finally, *"snitching"* [1, 52] – the application monitoring request latency and picking the fastest replica – can be employed; however, such techniques are *ineffective* if noise is bursty.

All of the techniques discussed above attempt to minimize tail in the *absence* of information about underlying resource busyness. While the OS layer may have such information, it is *hidden* and *unexposed*. A prime example is the `read()` interface that returns either success or error. However, when resources are busy (disk contention from other tenants, device garbage collection, etc.), a `read()` can be stalled inside the OS for some time. Currently, the OS does not have a direct way to indicate that a request may take a long time, nor is there a way for applications to indicate they would like "to know the OS is busy."

Please refer to the paper!

Outline

- ☐ Introduction
- ☐ Design
- ☐ **Evaluation**
  - Tail reduction
  - Latency prediction accuracy
- ☐ Conclusion

MittCFQ-powered MongoDB

MittCFQ-powered MongoDB

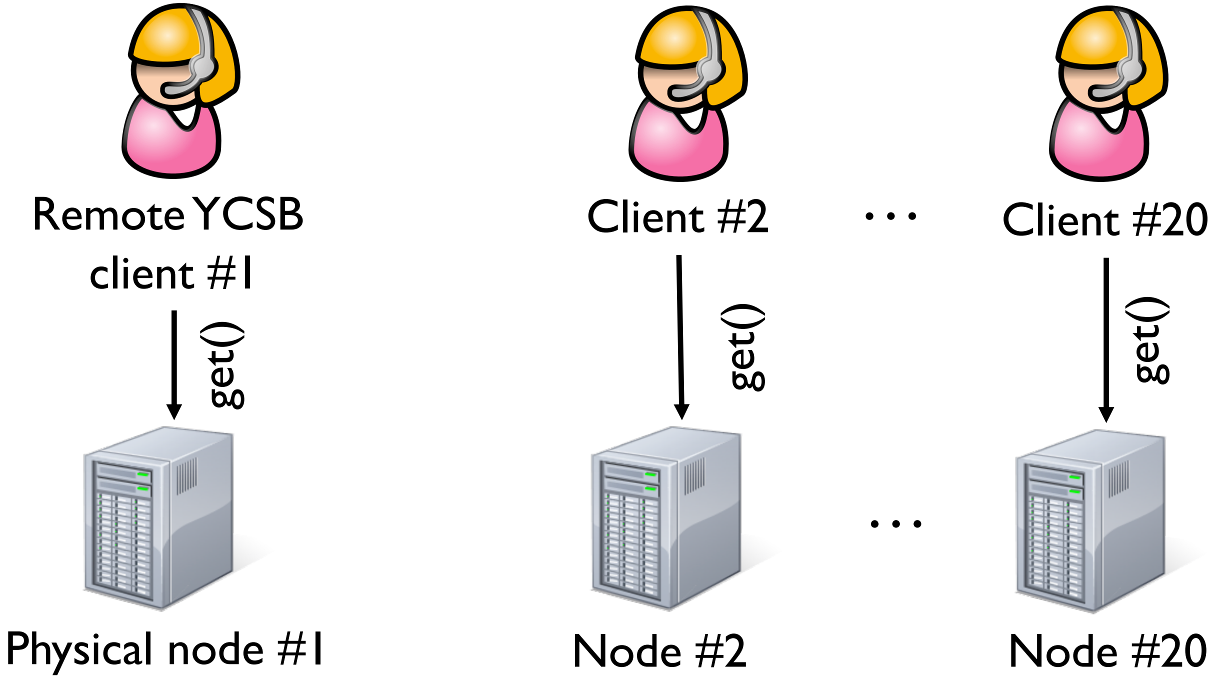
Physical node #1

Node #2

. . .

Node #20

MittCFQ-powered MongoDB



Remote YCSB client #1

Client #2 . . . Client #20

get()

get()

get()

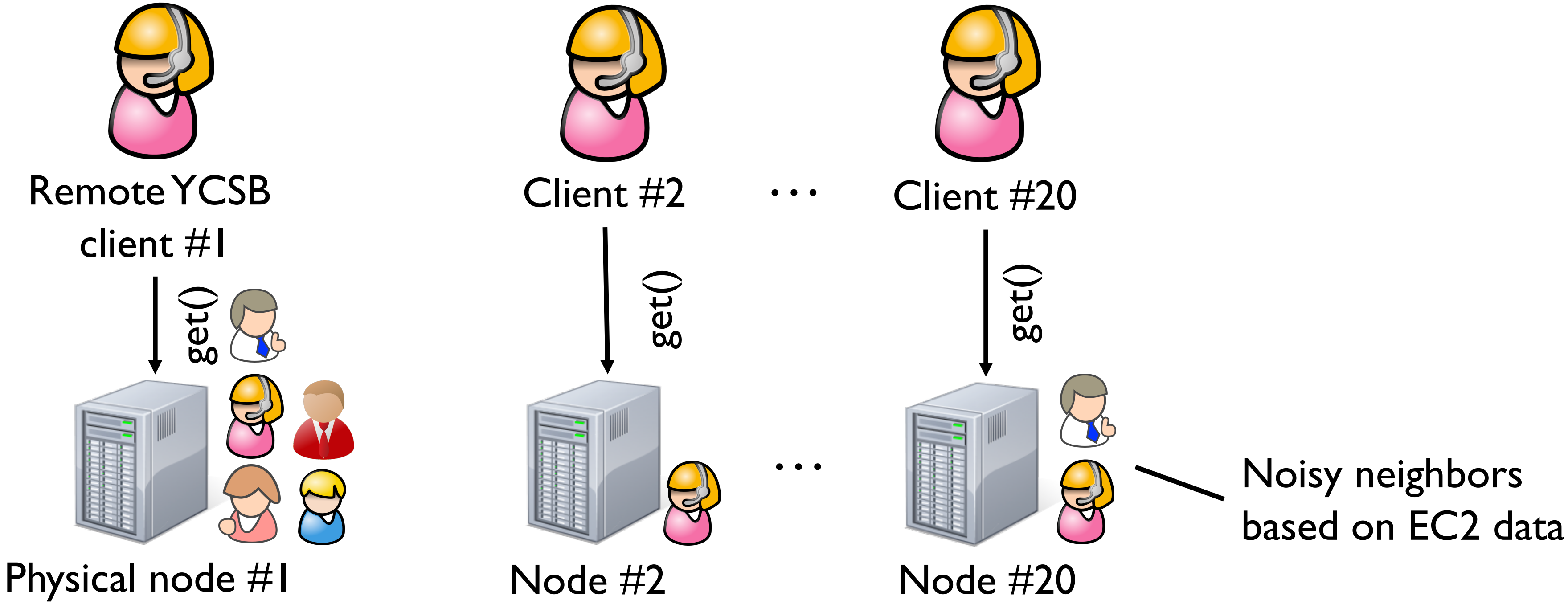
. . .

Physical node #1

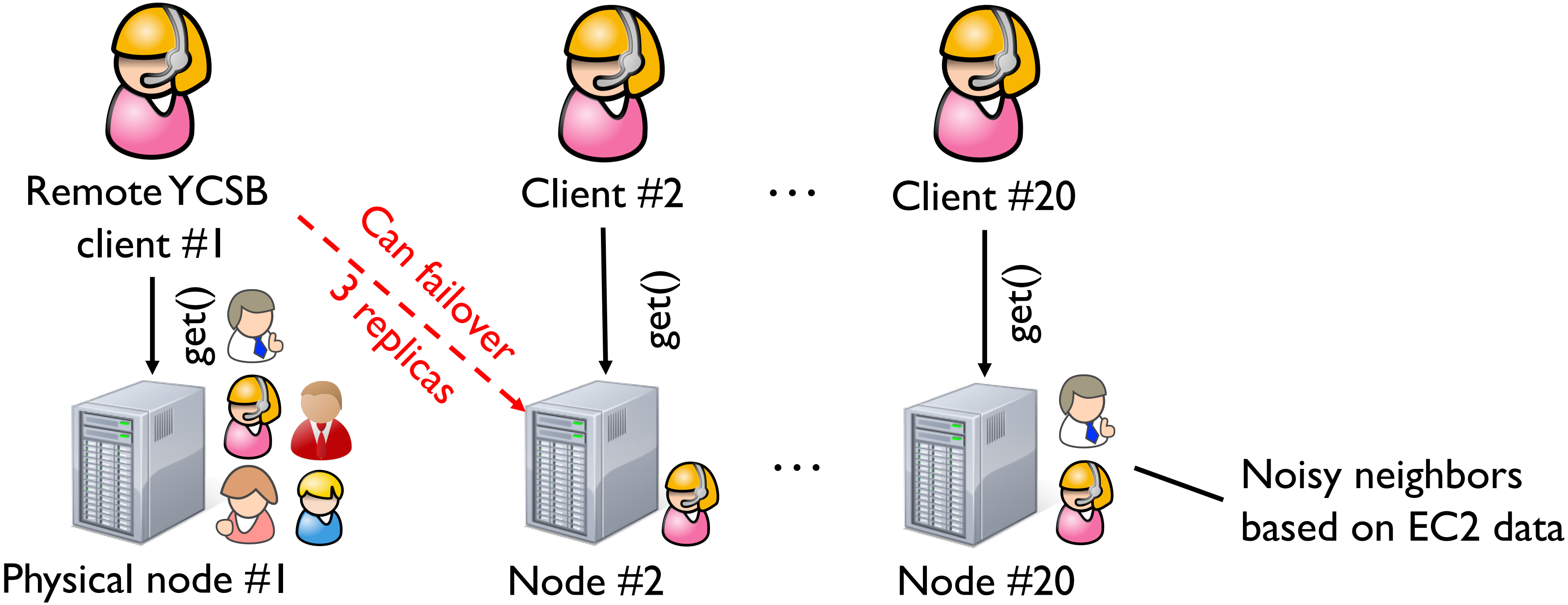
Node #2

Node #20

MittCFQ-powered MongoDB

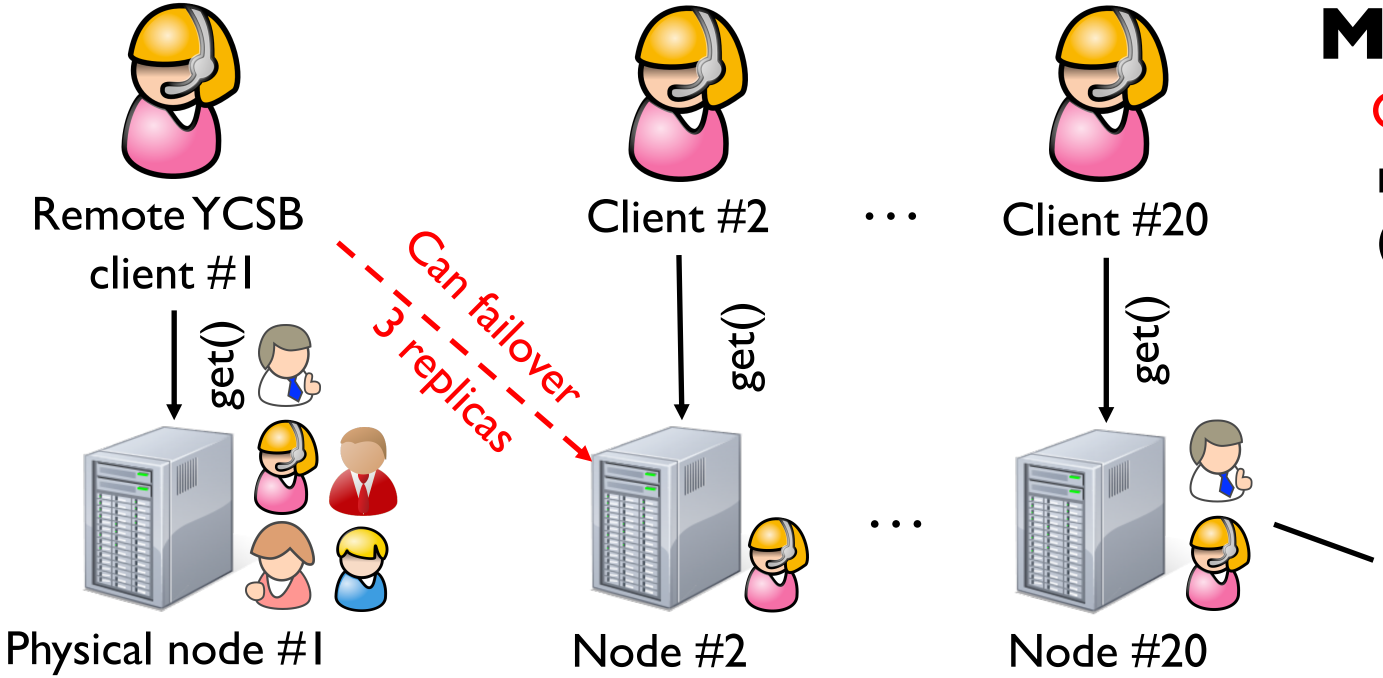


MittCFQ-powered MongoDB



Noisy neighbors based on EC2 data

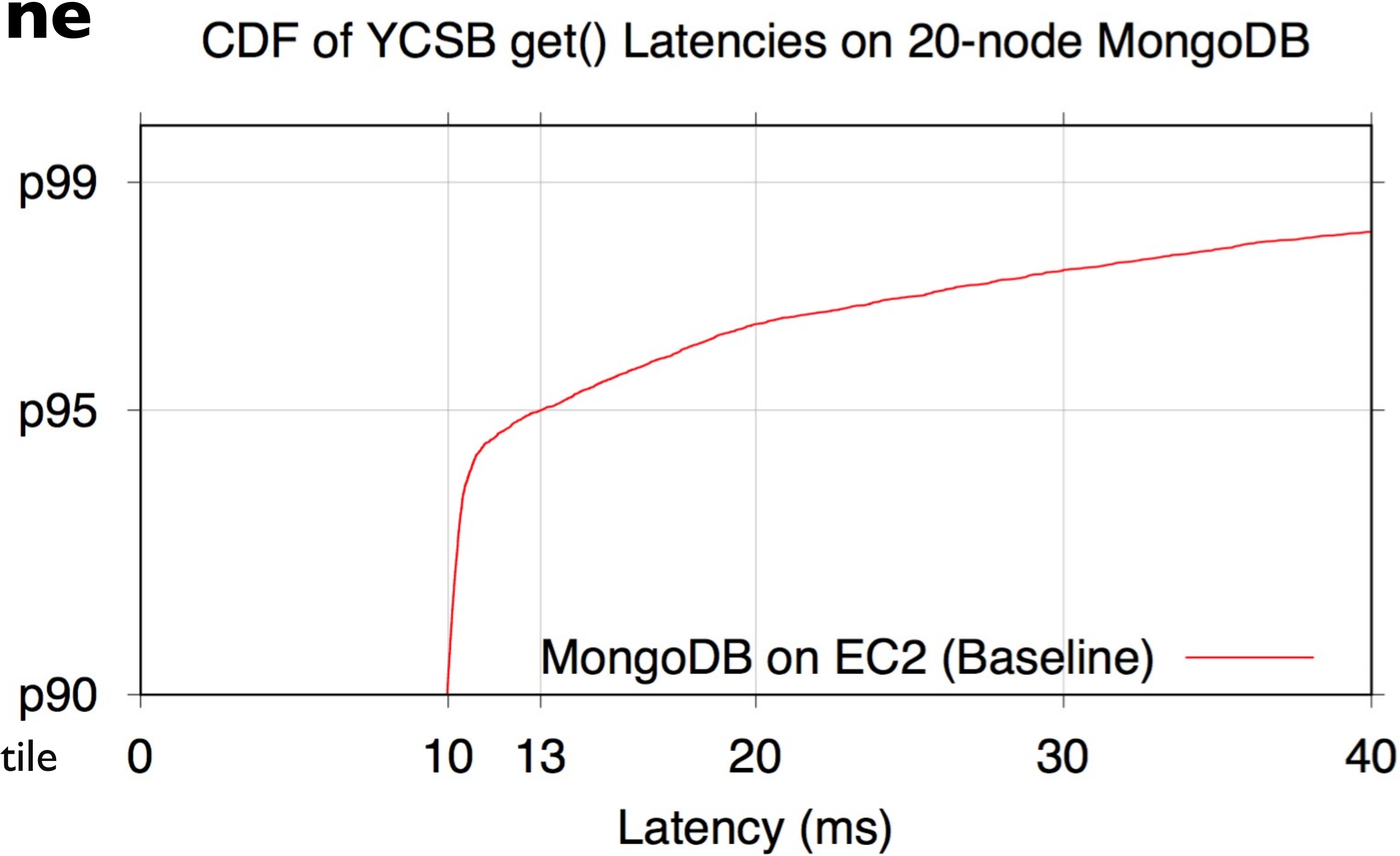
MittCFQ-powered MongoDB



Metric:

CDF of all **get()** requests latencies (total 6 million data points)

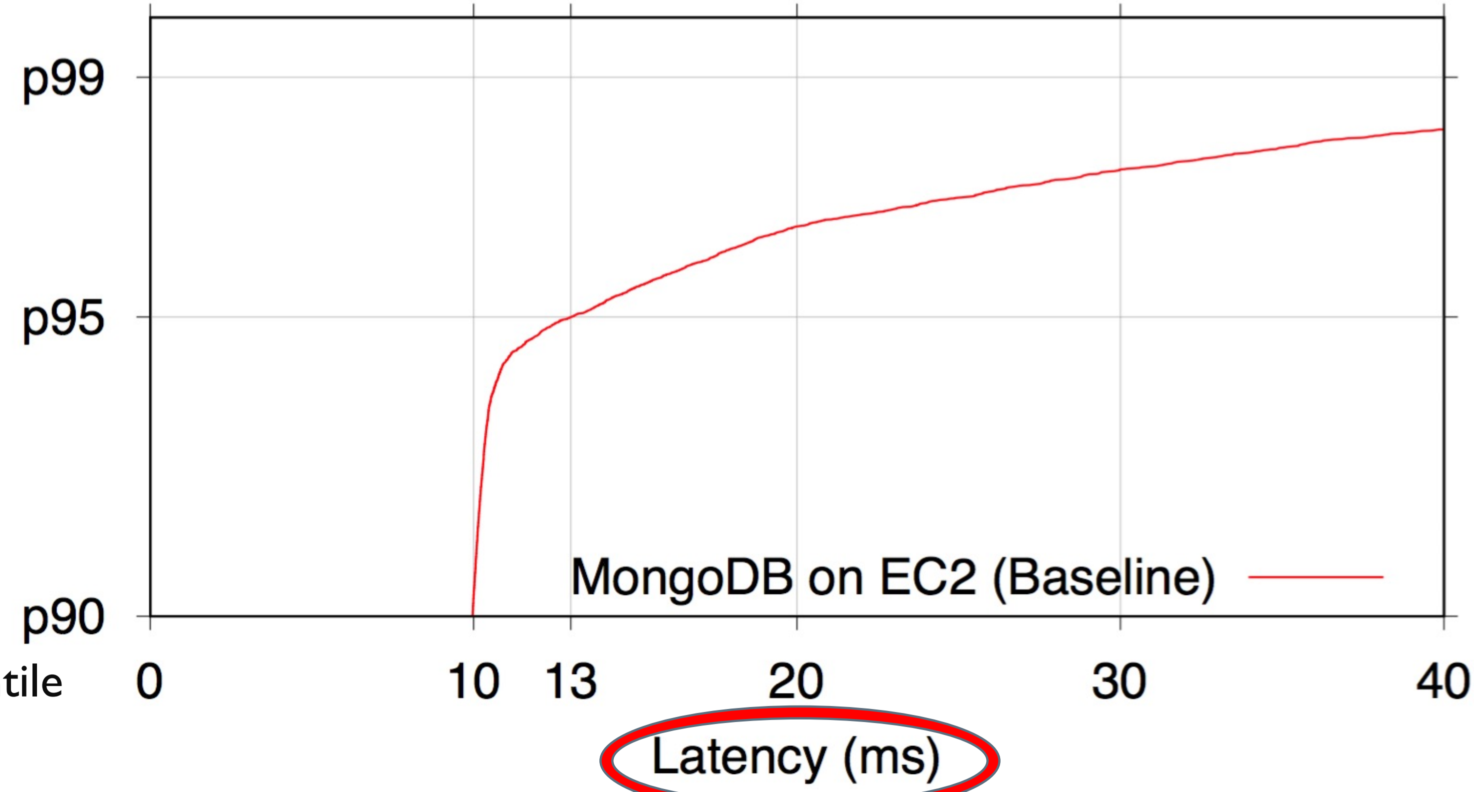
Baseline



90th percentile

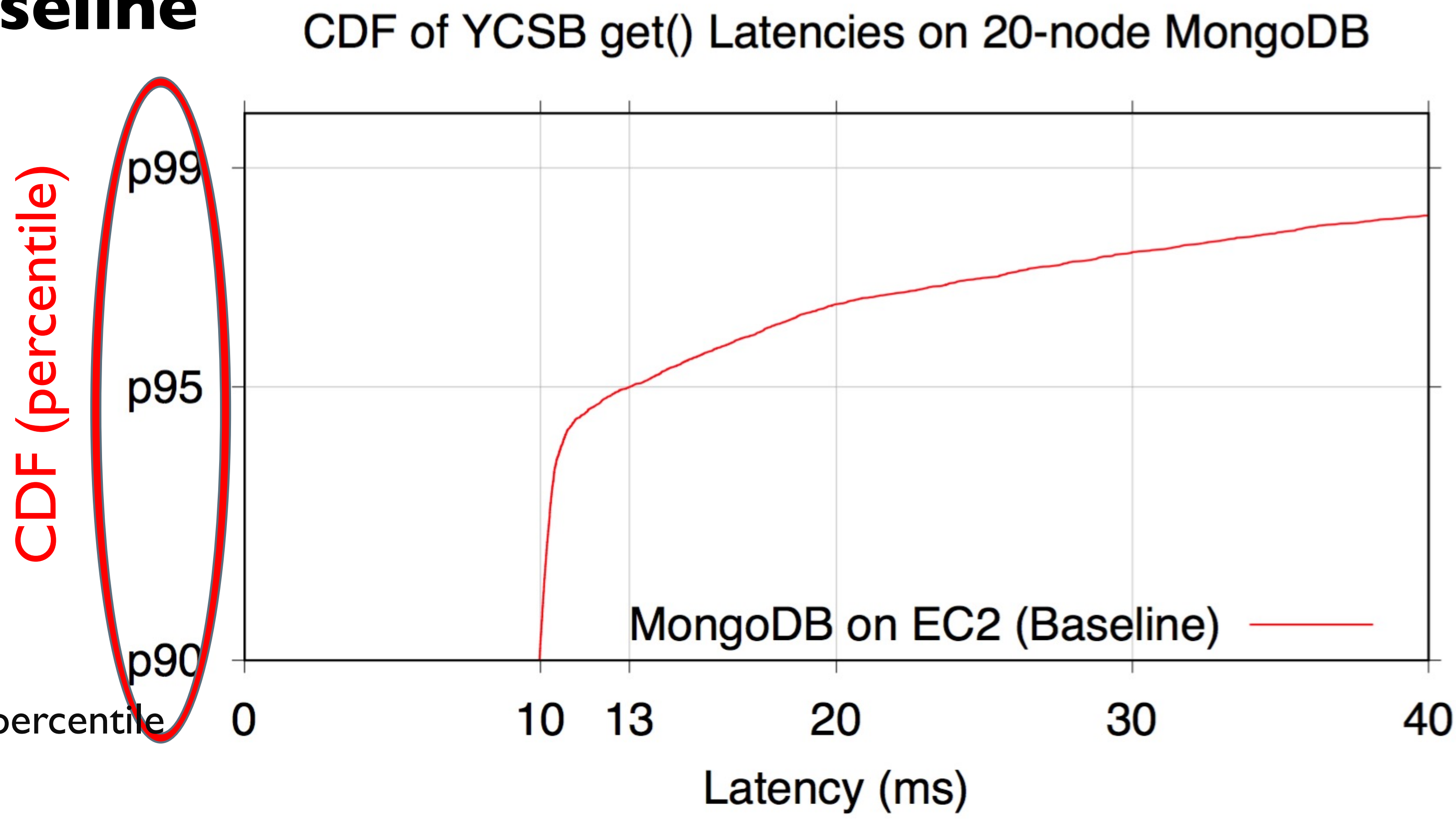
Baseline

CDF of YCSB get() Latencies on 20-node MongoDB

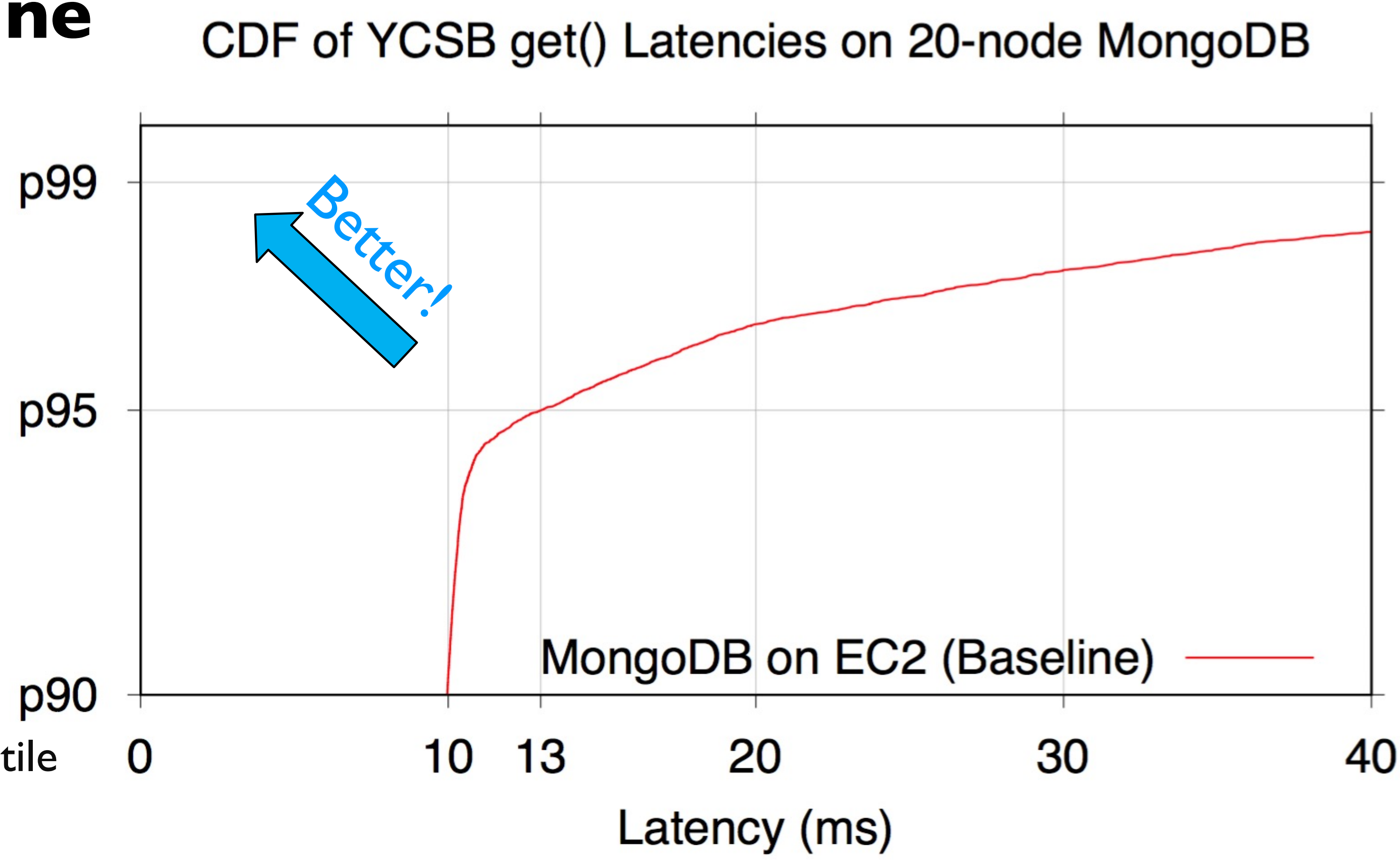


90th percentile

Baseline

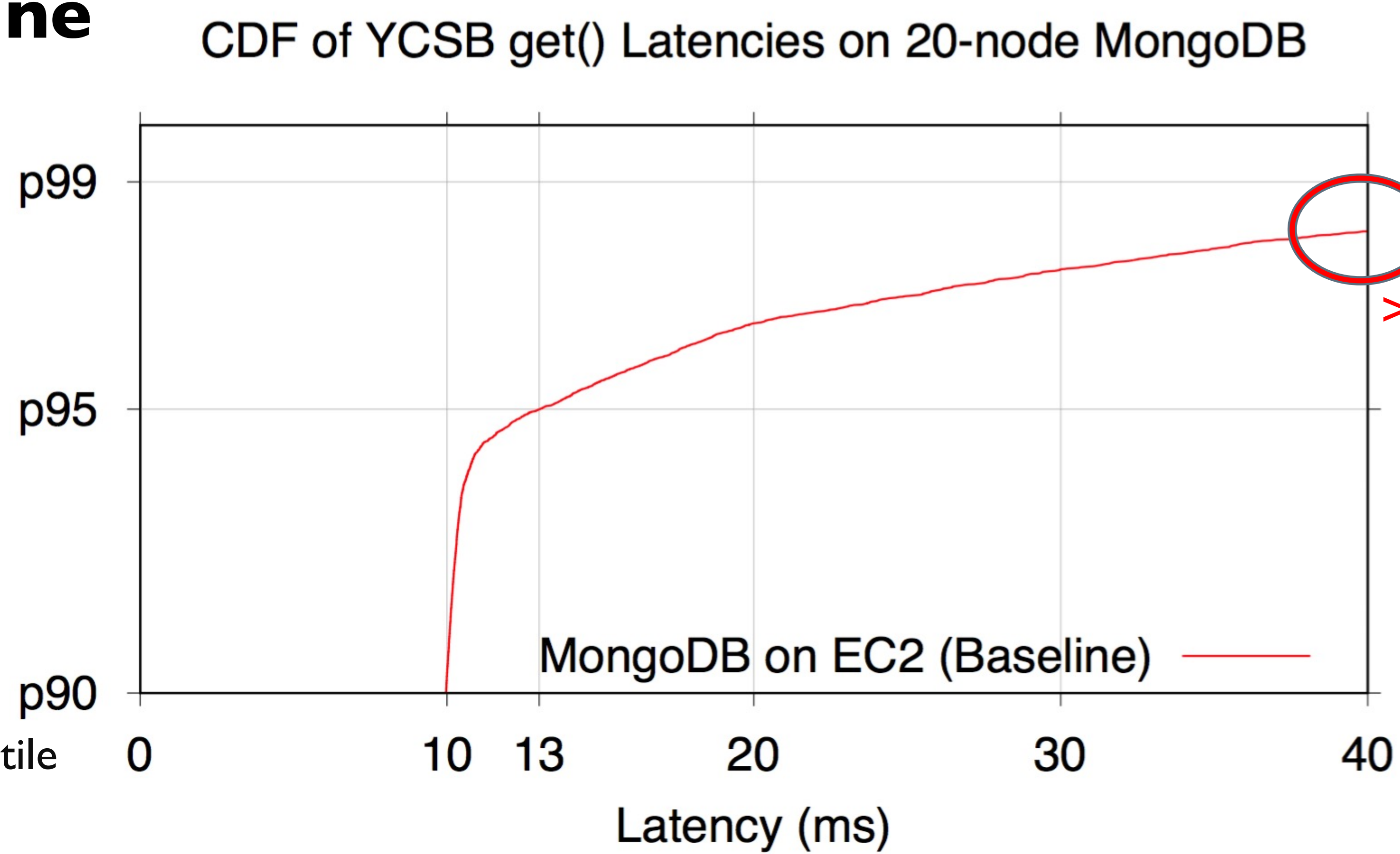

CDF of YCSB get() Latencies on 20-node MongoDB
CDF (percentile)
p99
p95
p90
90th percentile
0
10
13
20
30
40
Latency (ms)
MongoDB on EC2 (Baseline)

Baseline



90th percentile

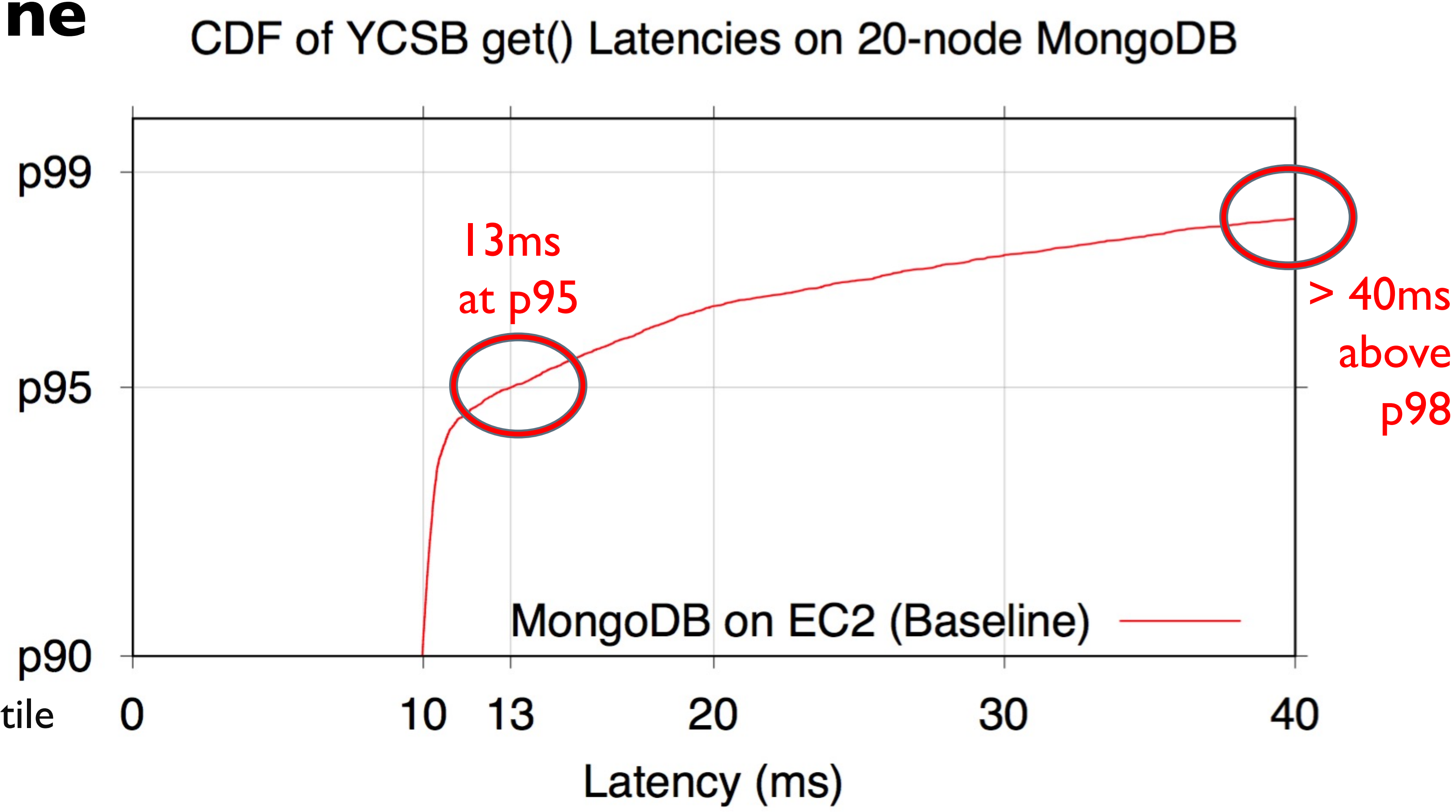
Baseline



90th percentile

> 40ms above p98

Baseline

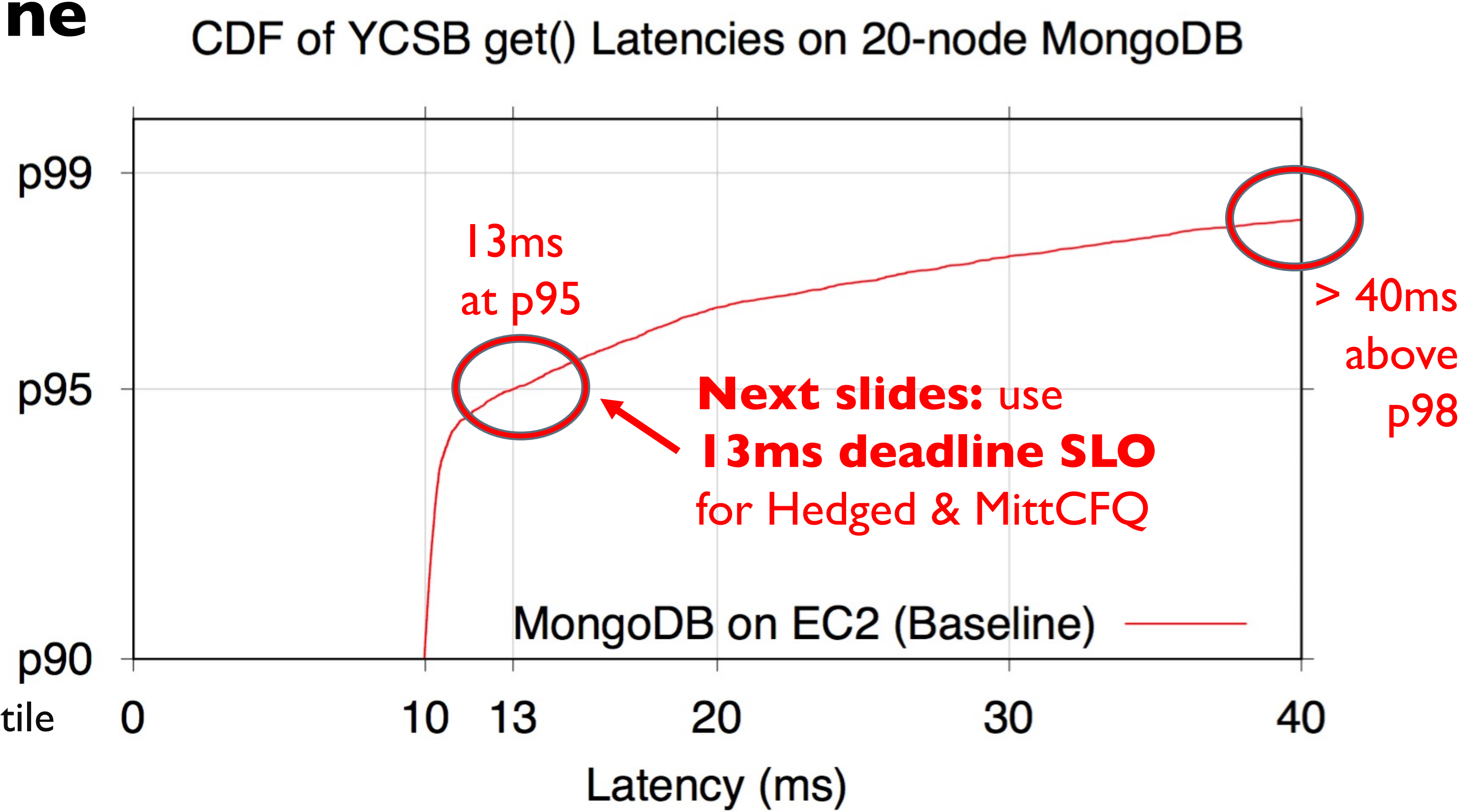


13ms at p95

> 40ms above p98

90th percentile

Baseline



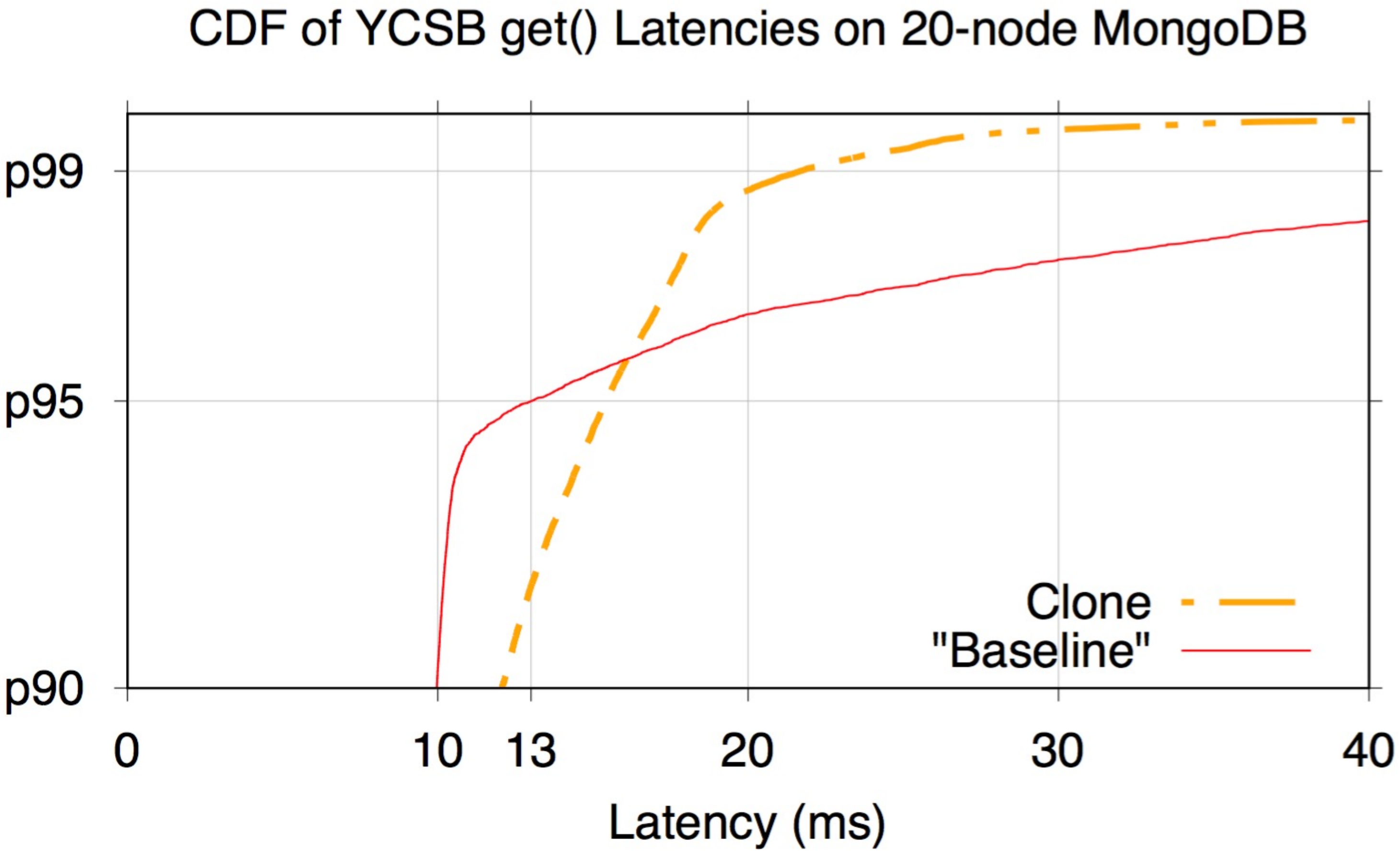
13ms at p95

> 40ms above p98

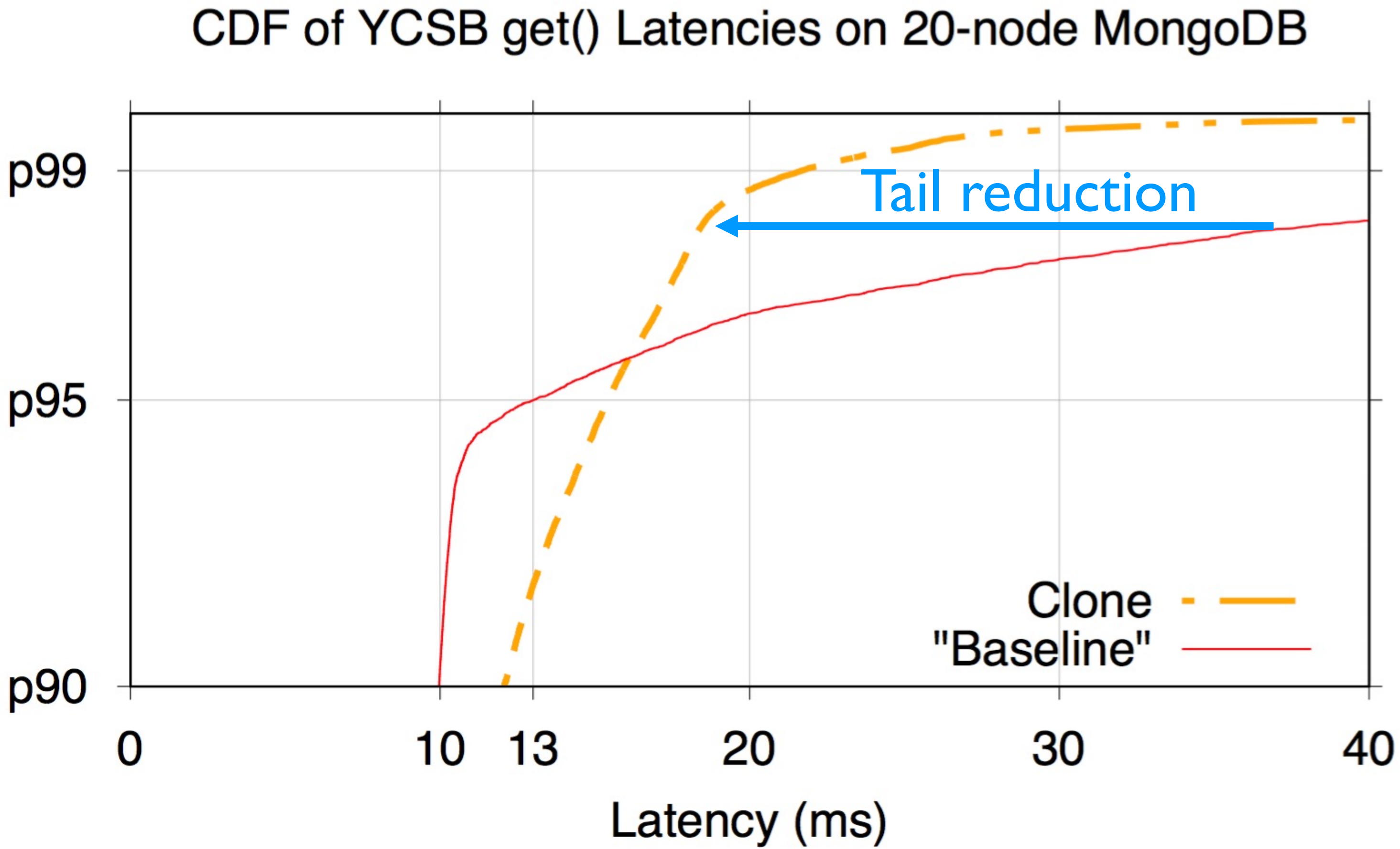
Next slides: use **13ms deadline SLO** for Hedged & MittCFQ

90th percentile

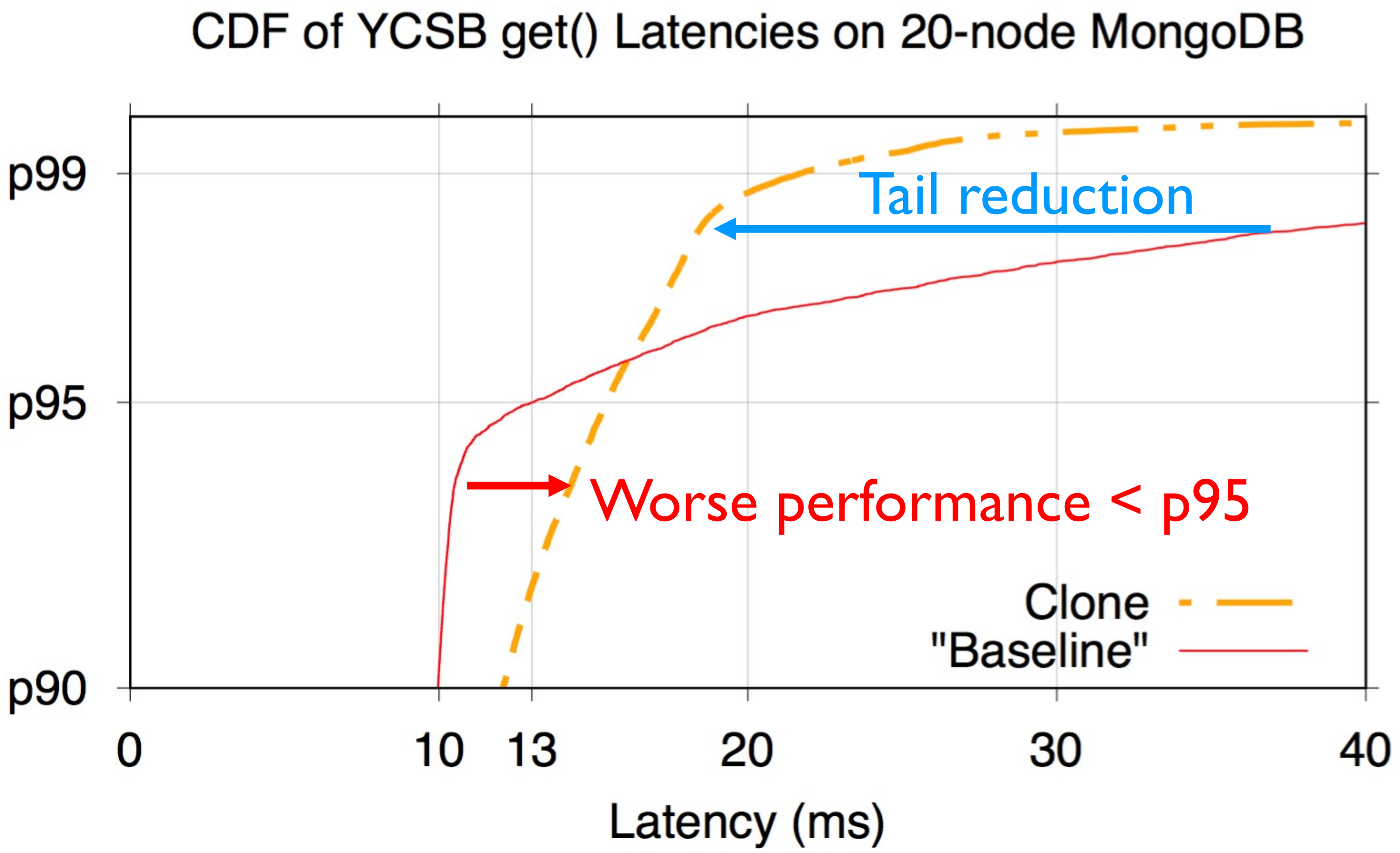
Clone


CDF of YCSB get() Latencies on 20-node MongoDB
p99
p95
p90
0
10
13
20
30
40
Latency (ms)
Clone
"Baseline"

Clone


CDF of YCSB get() Latencies on 20-node MongoDB
p99
p95
p90
0
10
13
20
30
40
Latency (ms)
Tail reduction
Clone
"Baseline"

Clone



Tail reduction

Worse performance < p95

Hedged Requests

DOI:10.1145/2408776.2408794

Software techniques that tolerate latency variability are vital to building responsive large-scale Web services.

BY JEFFREY DEAN AND LUIZ ANDRÉ BARROSO

The Tail at Scale

Communications of the ACM, vol. 56 (2013), pp. 74-80

Hedged Requests

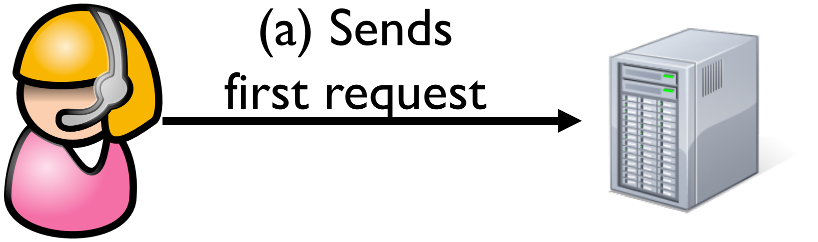
DOI:10.1145/2408776.2408794

Software techniques that tolerate latency variability are vital to building responsive large-scale Web services.

BY JEFFREY DEAN AND LUIZ ANDRÉ BARROSO

The Tail at Scale

Communications of the ACM,
vol. 56 (2013), pp. 74-80



(a) Sends first request

Hedged Requests

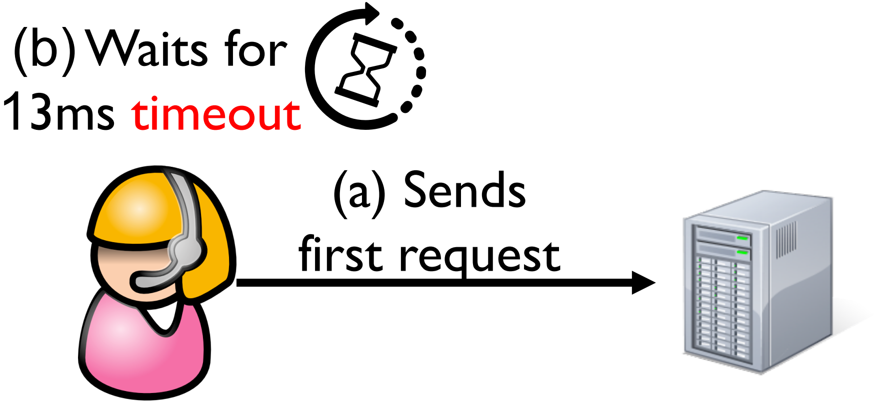
DOI:10.1145/2408776.2408794

Software techniques that tolerate latency variability are vital to building responsive large-scale Web services.

BY JEFFREY DEAN AND LUIZ ANDRÉ BARROSO

The Tail at Scale

Communications of the ACM, vol. 56 (2013), pp. 74-80



(b) Waits for 13ms timeout

(a) Sends first request

Hedged Requests

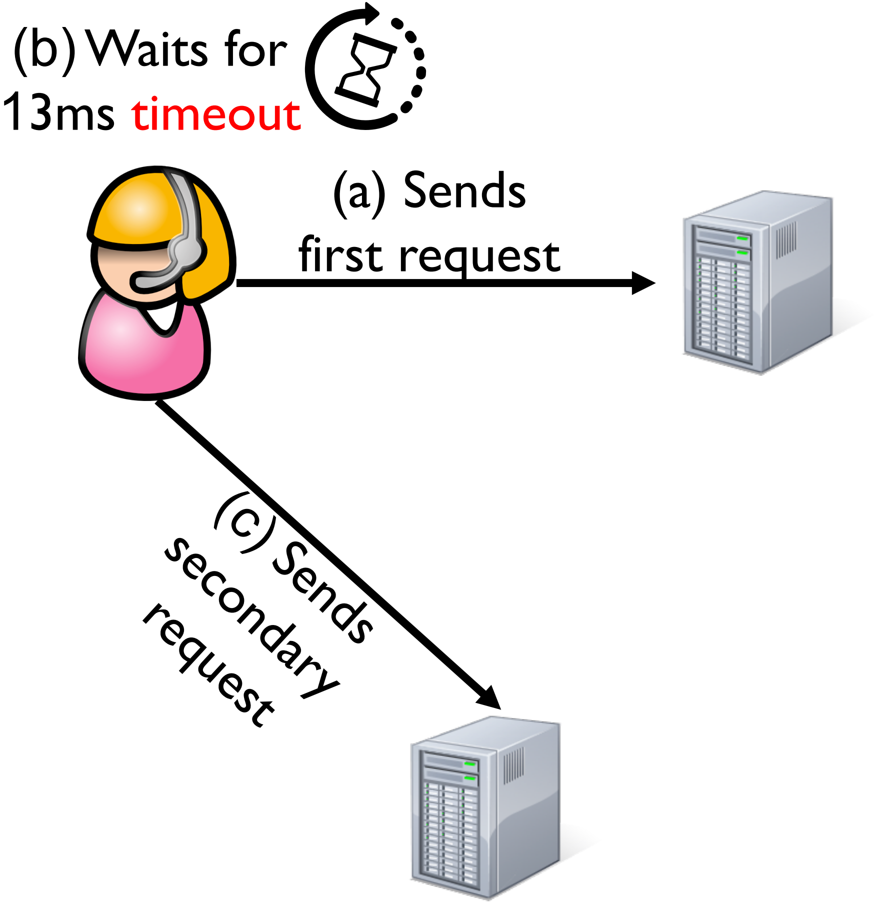
DOI:10.1145/2408776.2408794

Software techniques that tolerate latency variability are vital to building responsive large-scale Web services.

BY JEFFREY DEAN AND LUIZ ANDRÉ BARROSO

The Tail at Scale

Communications of the ACM, vol. 56 (2013), pp. 74-80



(b) Waits for 13ms timeout

(a) Sends first request

(c) Sends secondary request

Hedged Requests

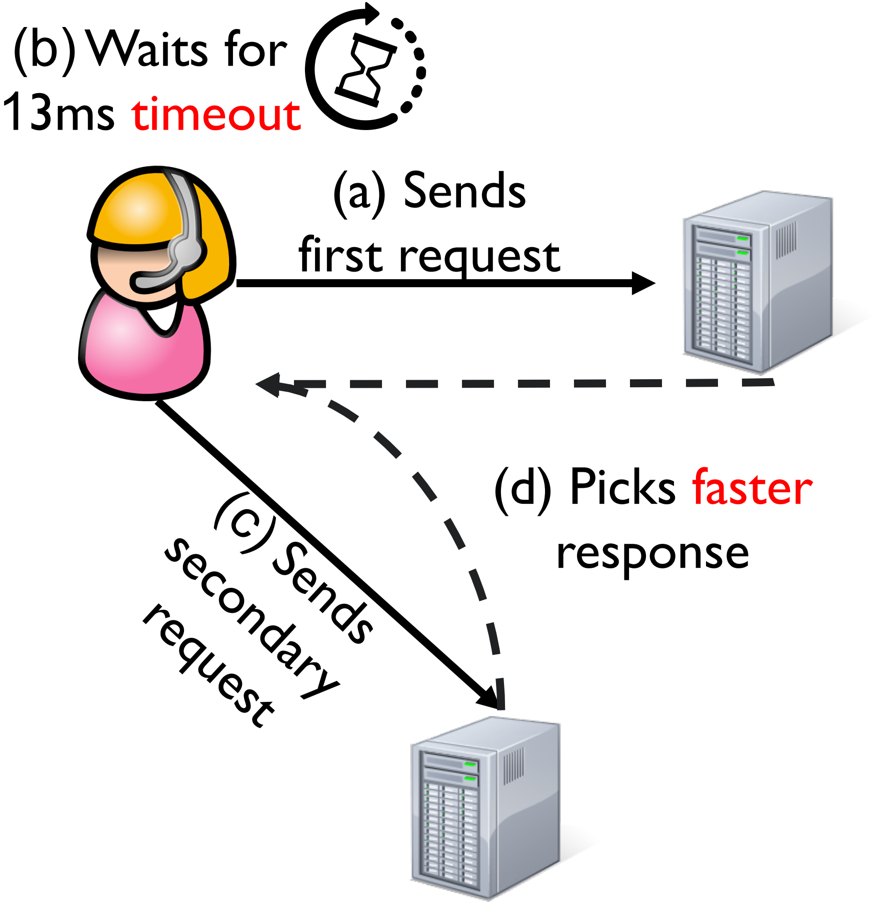
DOI:10.1145/2408776.2408794

Software techniques that tolerate latency variability are vital to building responsive large-scale Web services.

BY JEFFREY DEAN AND LUIZ ANDRÉ BARROSO

The Tail at Scale

Communications of the ACM, vol. 56 (2013), pp. 74-80

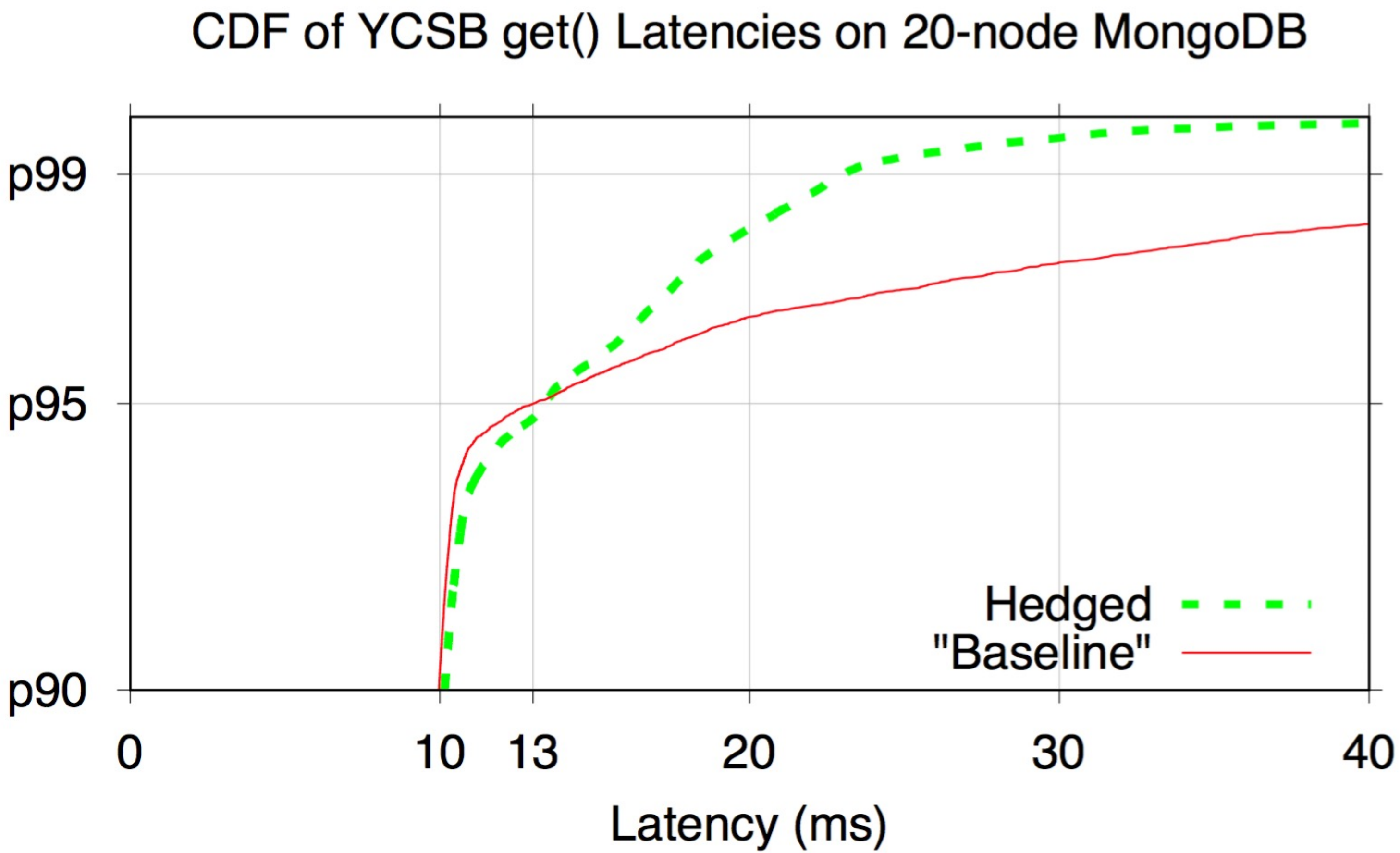


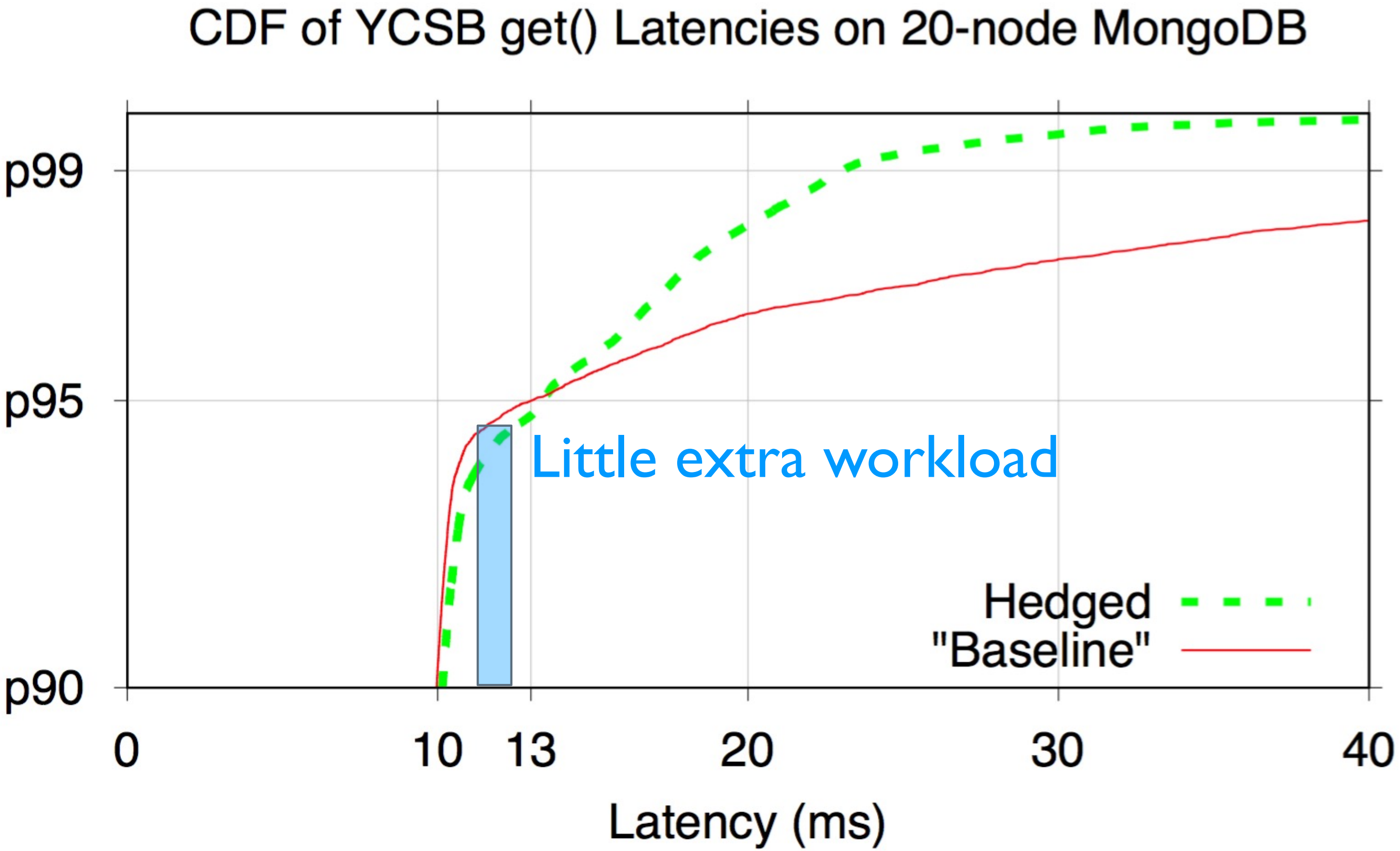
(b) Waits for 13ms timeout

(a) Sends first request

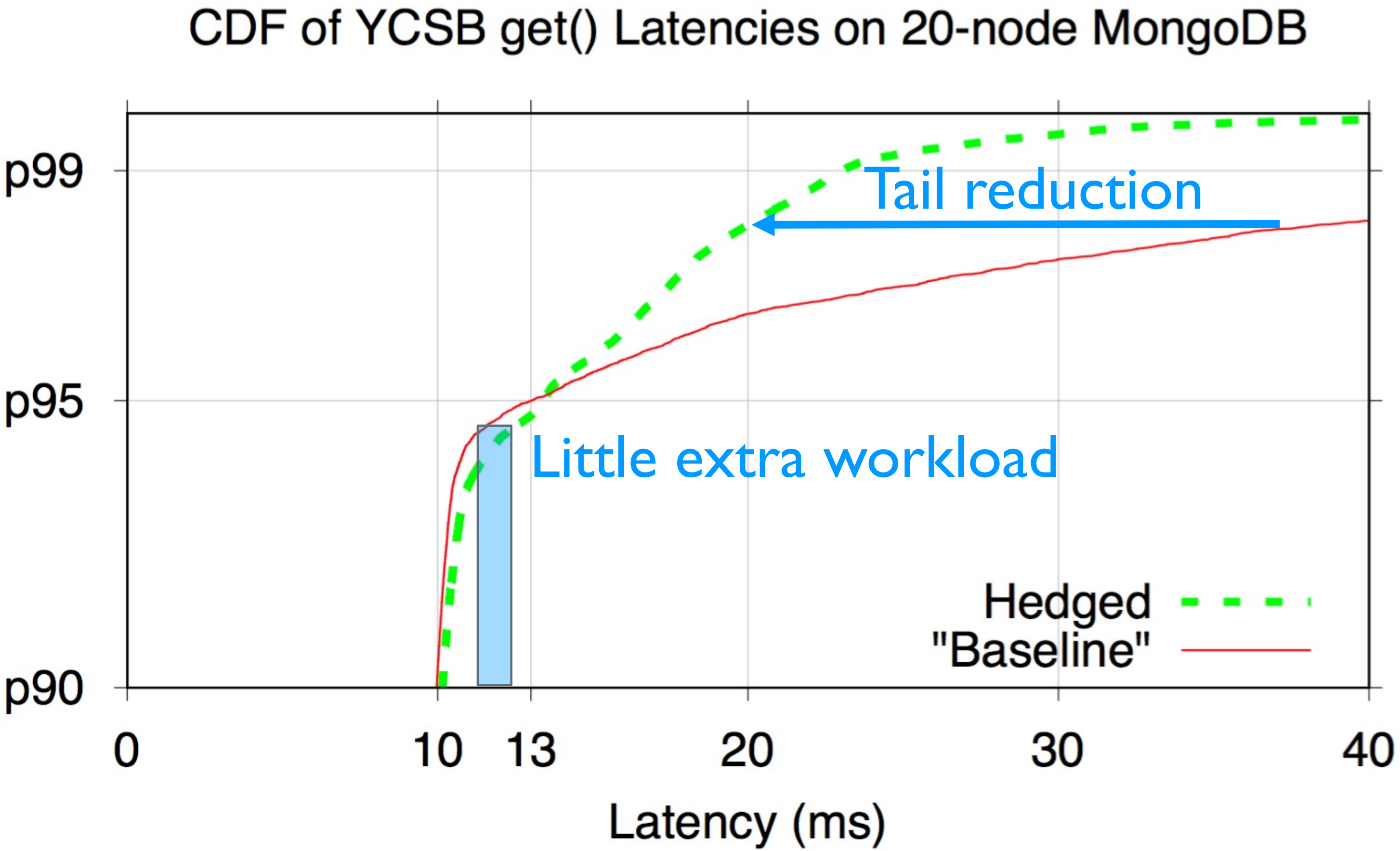
(c) Sends secondary request

(d) Picks faster response


CDF of YCSB get() Latencies on 20-node MongoDB
p99
p95
p90
0
10
13
20
30
40
Latency (ms)
Hedged
"Baseline"

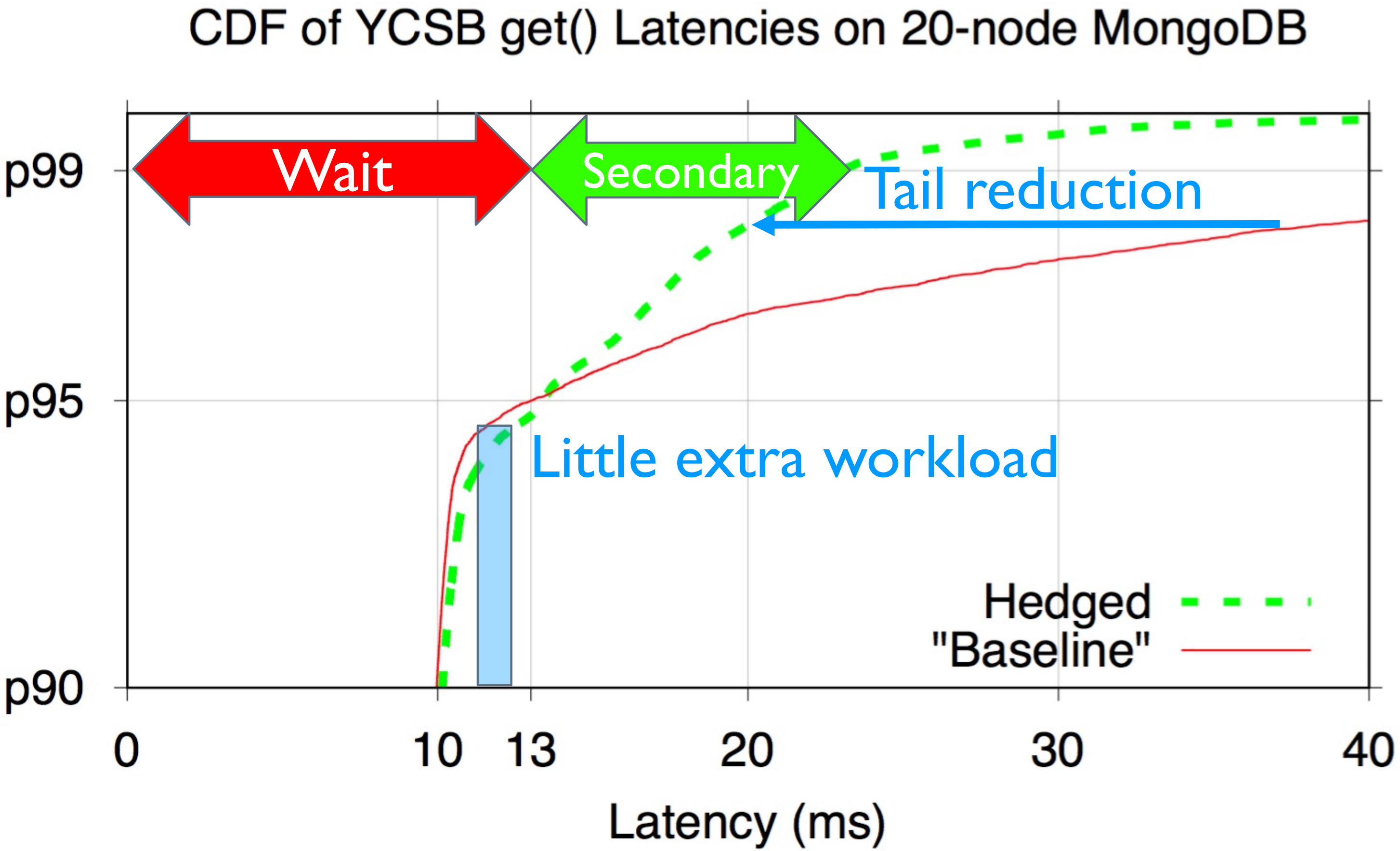


Little extra workload

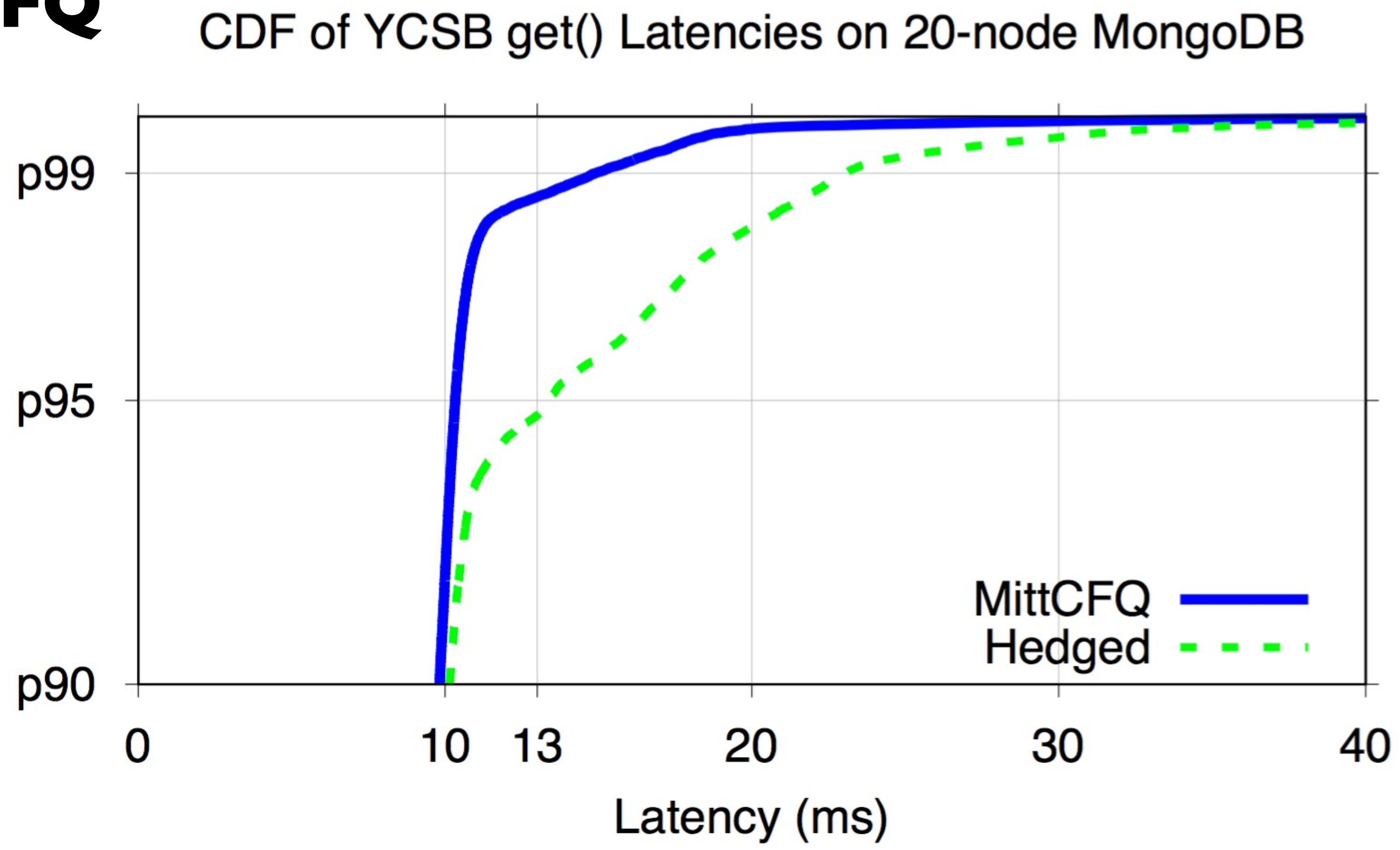


Tail reduction

Little extra workload

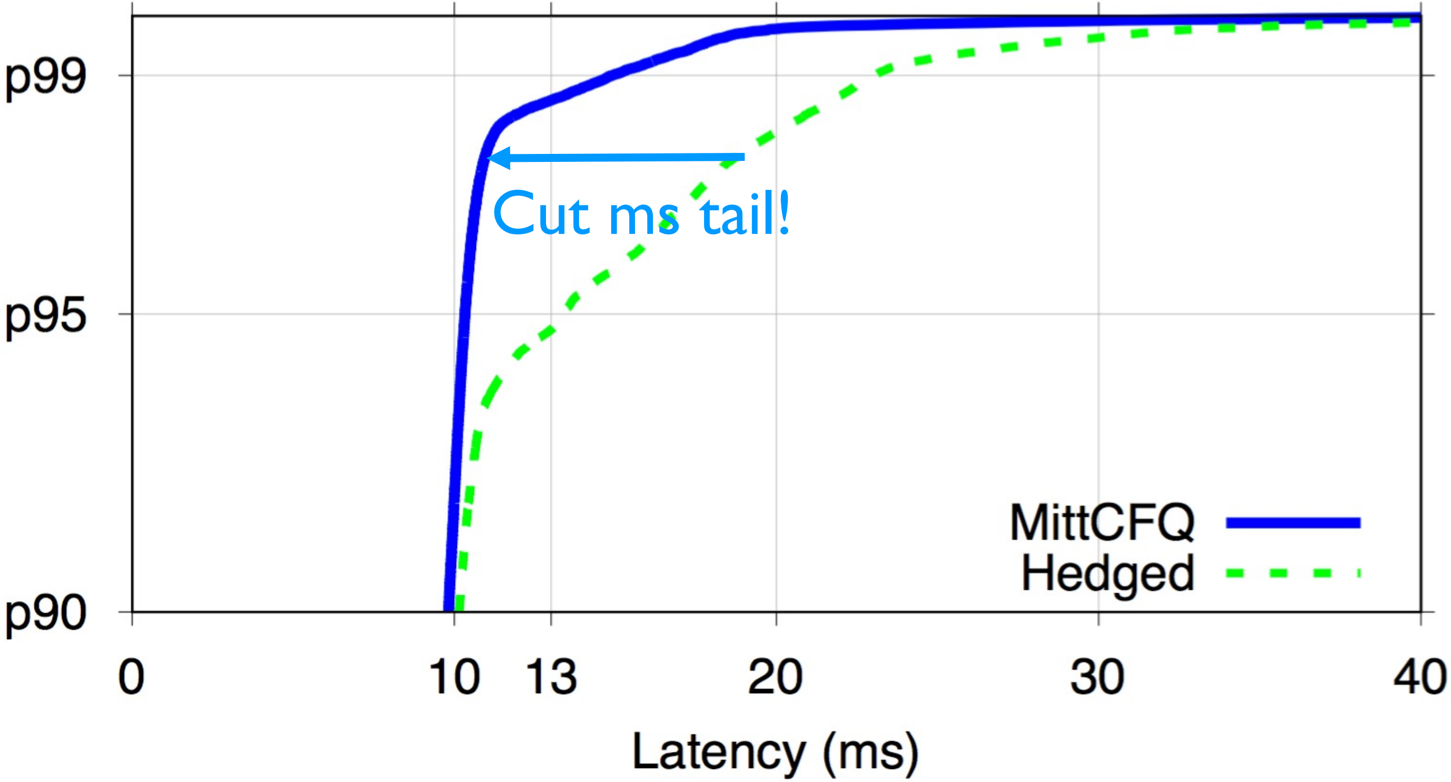

CDF of YCSB get() Latencies on 20-node MongoDB
Wait
Secondary
Tail reduction
Little extra workload
p99
p95
p90
0
10
13
20
30
40
Latency (ms)
Hedged
"Baseline"

MittCFQ

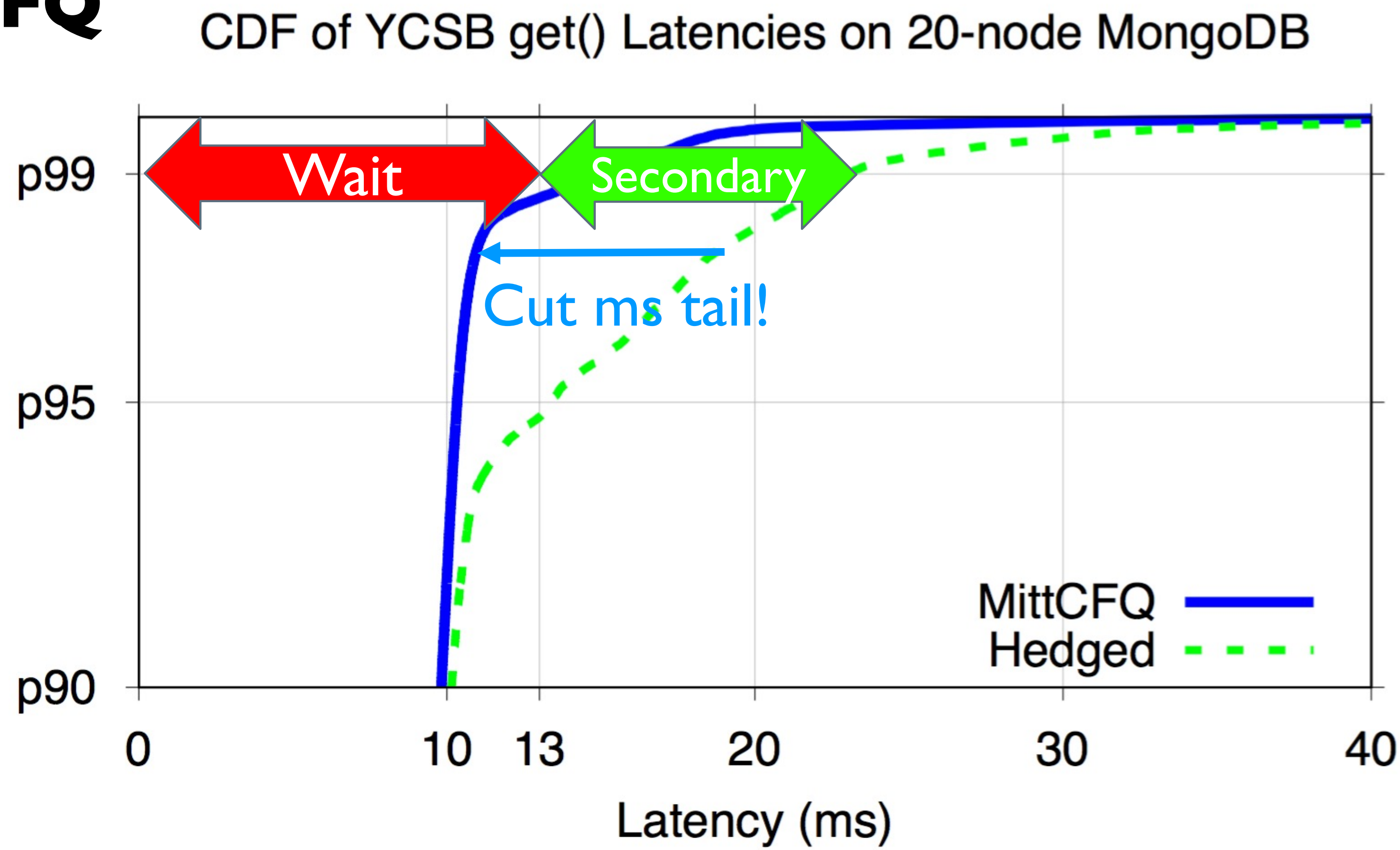

CDF of YCSB get() Latencies on 20-node MongoDB
p99
p95
p90
0
10
13
20
30
40
Latency (ms)
MittCFQ
Hedged

MittCFQ

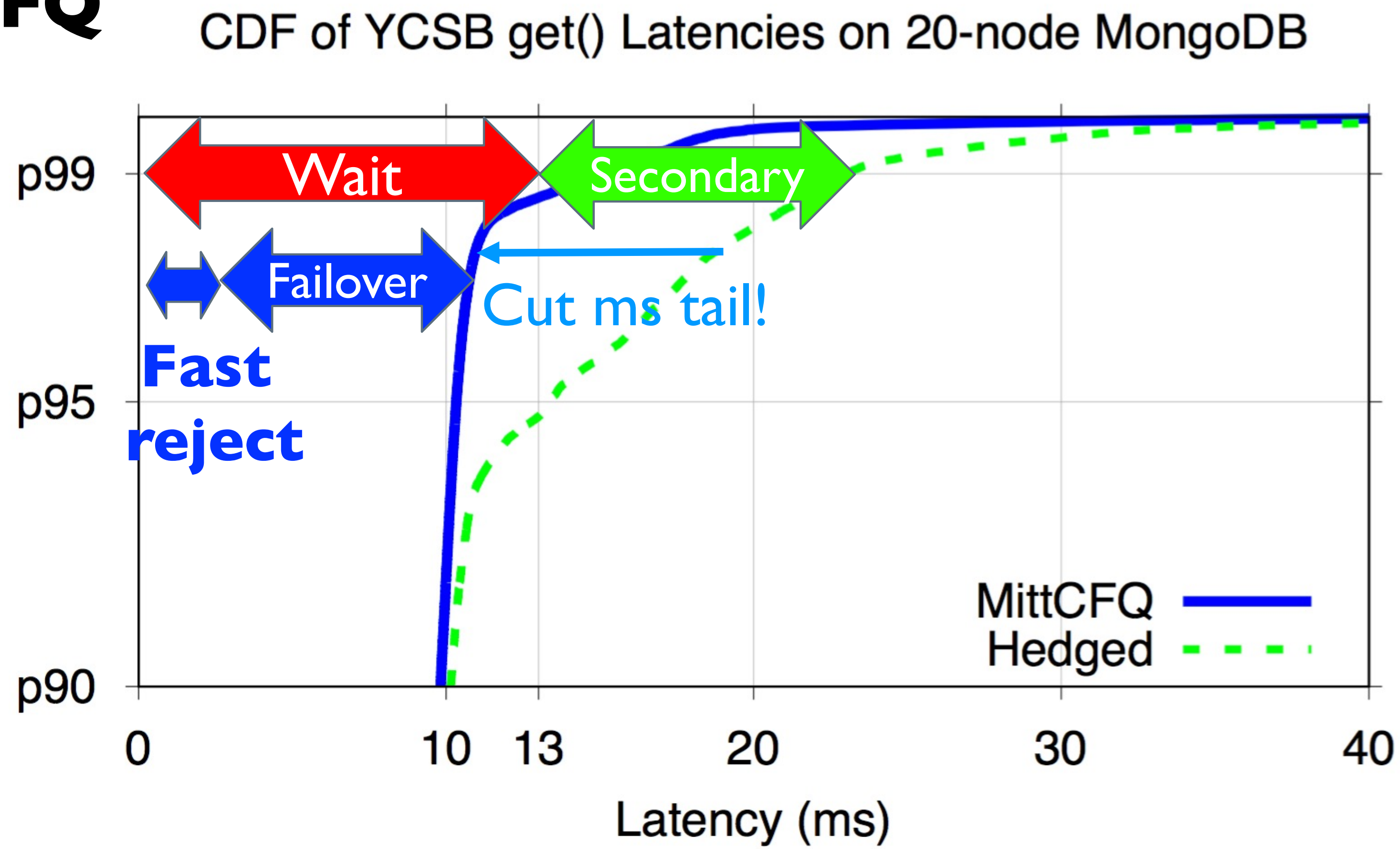
CDF of YCSB get() Latencies on 20-node MongoDB



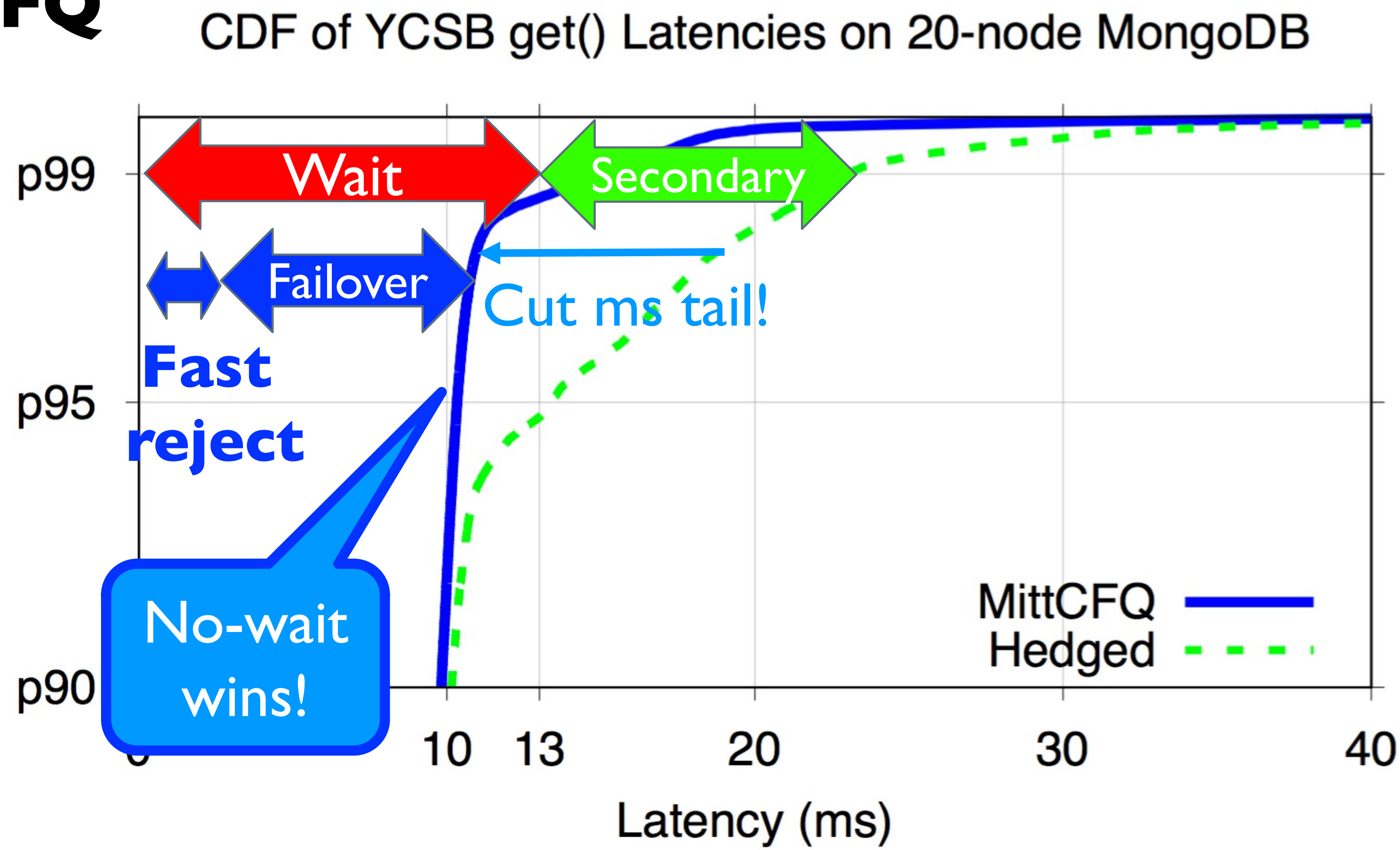
MittCFQ


CDF of YCSB get() Latencies on 20-node MongoDB
Wait
Secondary
Cut ms tail!
p99
p95
p90
0
10
13
20
30
40
Latency (ms)
MittCFQ
Hedged

MittCFQ

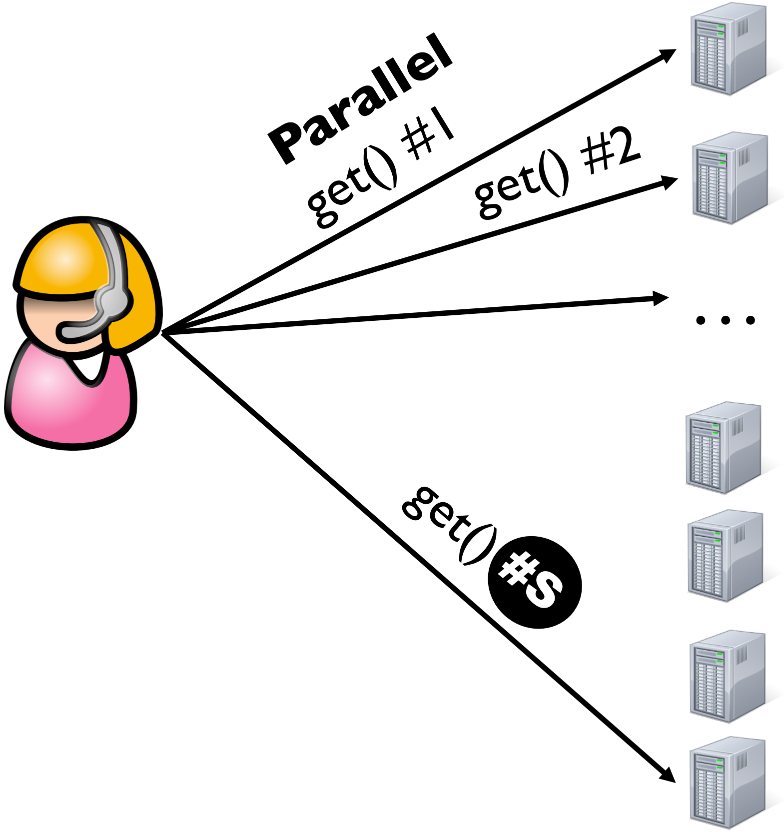

CDF of YCSB get() Latencies on 20-node MongoDB
p99
p95
p90
0
10
13
20
30
40
Latency (ms)
MittCFQ
Hedged
Wait
Secondary
Failover
Fast reject
Cut ms tail!

MittCFQ

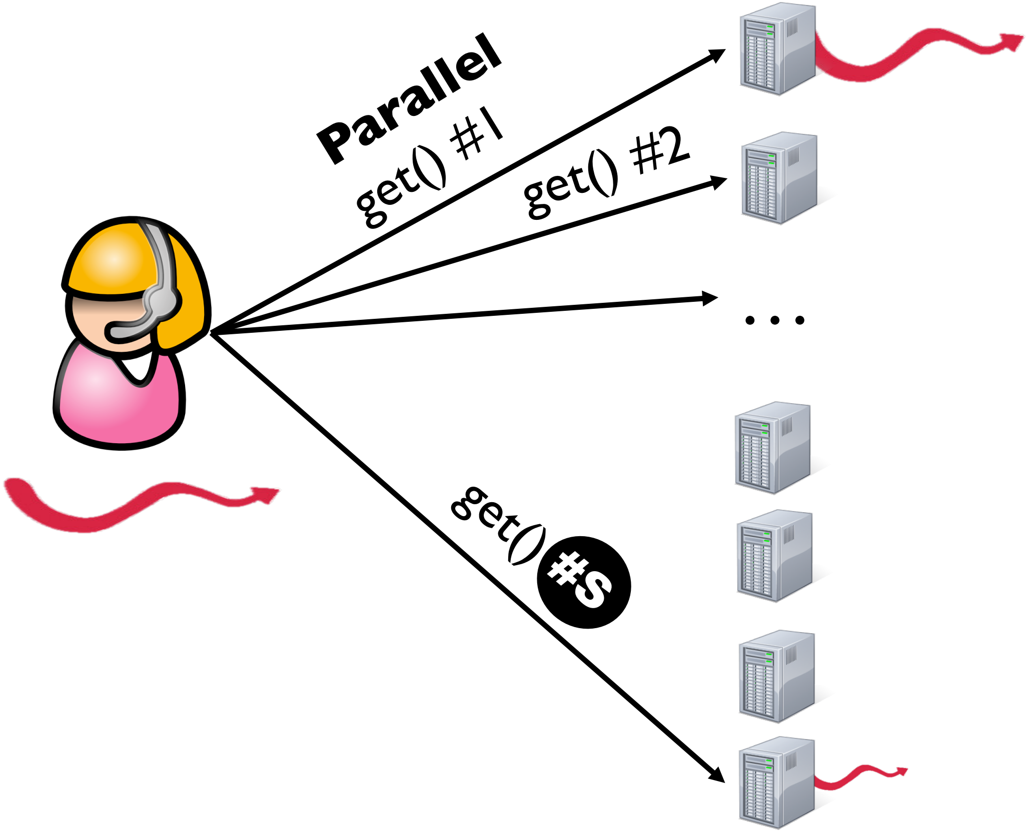

CDF of YCSB get() Latencies on 20-node MongoDB
p99
p95
p90
0
10
13
20
30
40
Latency (ms)
MittCFQ
Hedged
Wait
Secondary
Failover
Fast reject
Cut ms tail!
No-wait wins!

Tail amplified at Scale

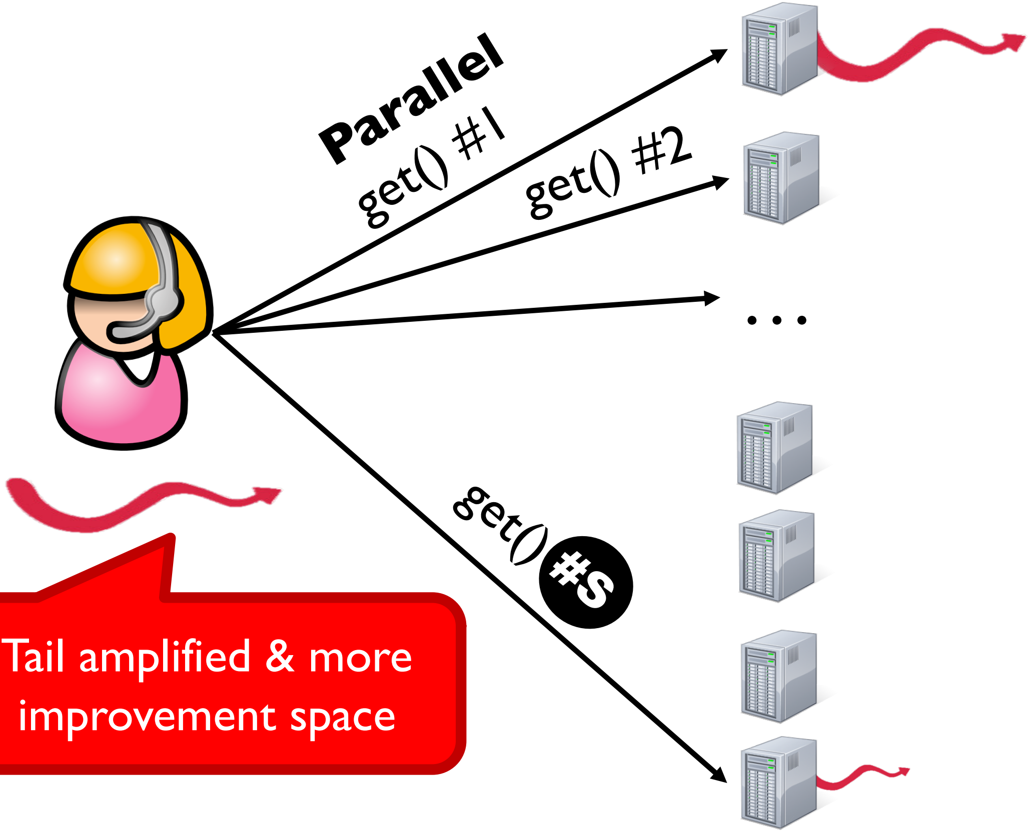
Tail amplified at Scale



Tail amplified at Scale

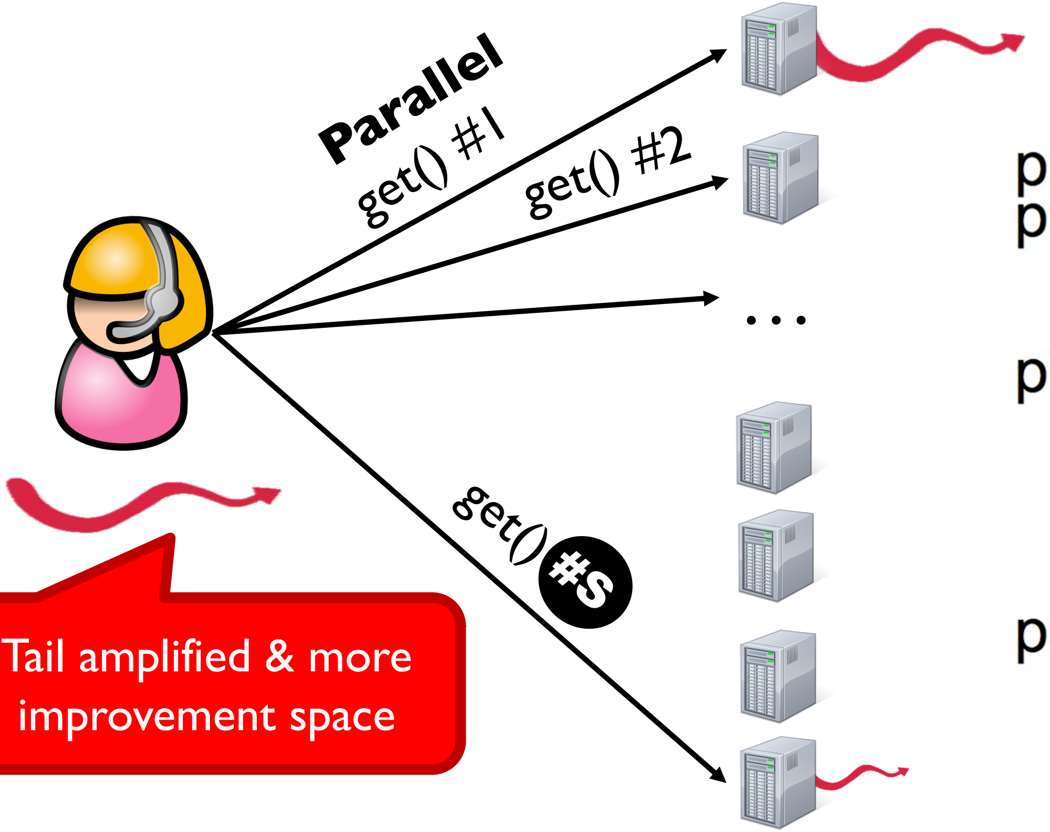


Tail amplified at Scale



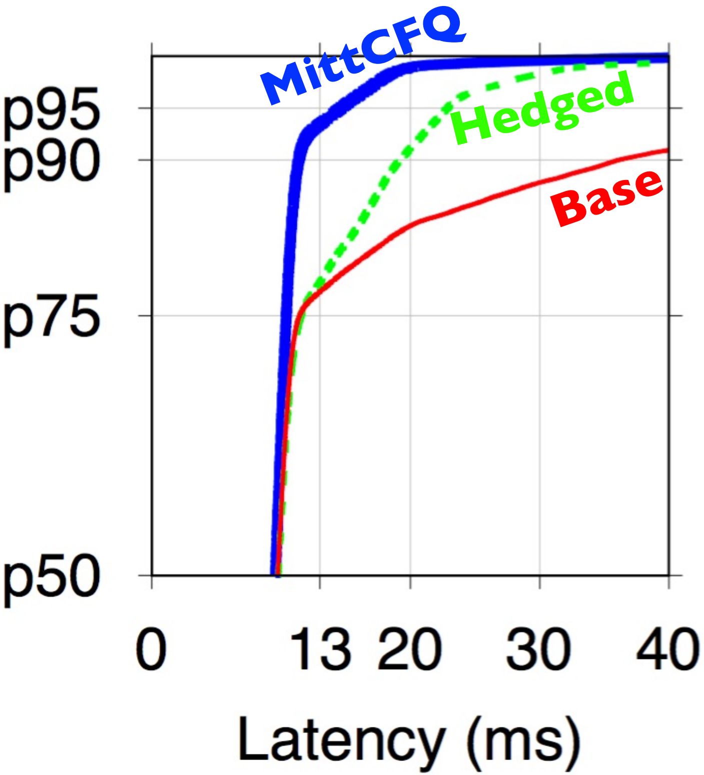
Tail amplified & more improvement space

Tail amplified at Scale

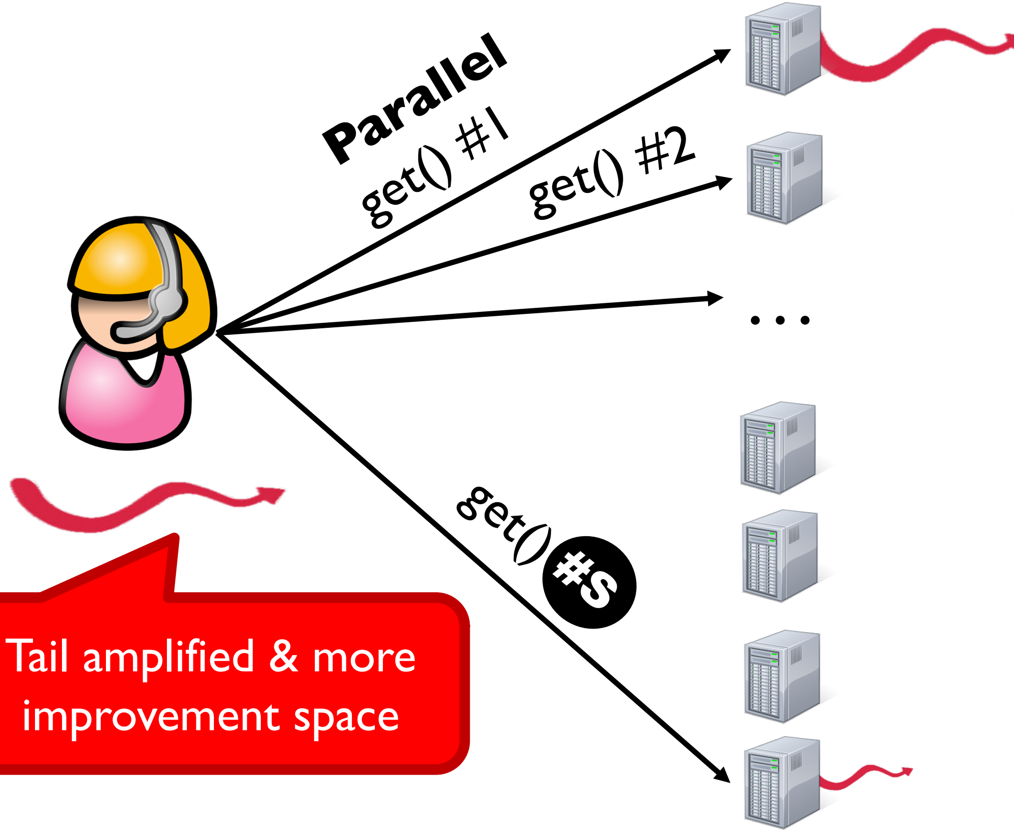


Tail amplified & more improvement space

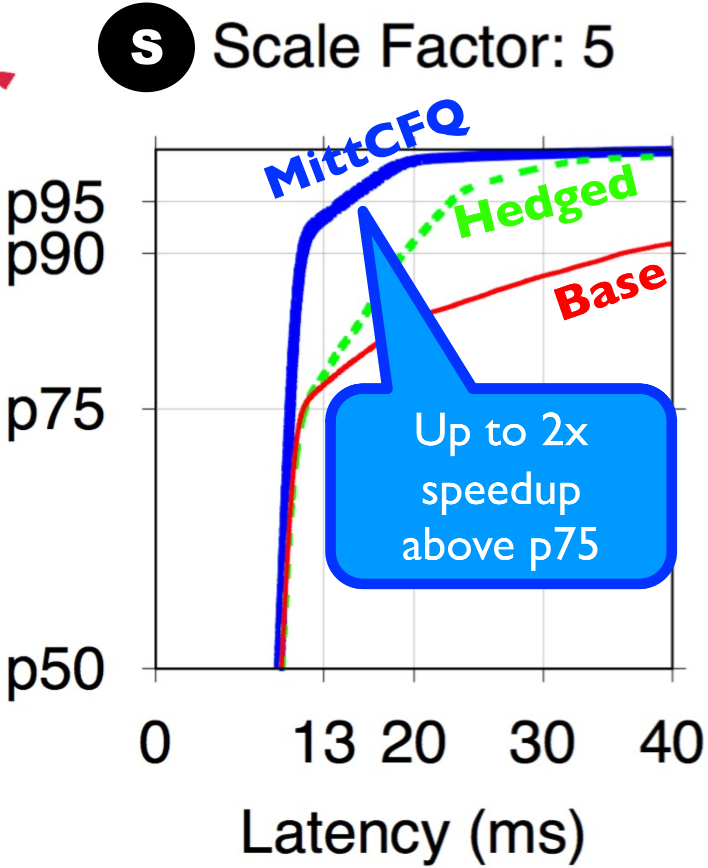
S Scale Factor: 5



Tail amplified at Scale



Tail amplified & more improvement space

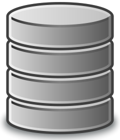


Up to 2x speedup above p75

Accuracy Evaluation

Accuracy Evaluation

MittCFQ



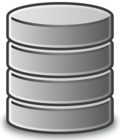
Disk

MittSSD

Open-Channel
SSD

Accuracy Evaluation

MittCFQ **MittSSD**



Disk

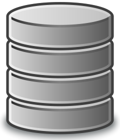
Open-Channel SSD

5 real-world block-level traces

DAPPS
DTRS
TPCC
EXCH
LMBE

Accuracy Evaluation

MittCFQ



Disk

MittSSD

Open-Channel SSD

Metrics:

- False positive: IO rejected, but deadline is met

5 real-world block-level traces

DAPPS

DTRS

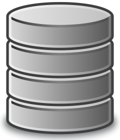
TPCC

EXCH

LMBE

Accuracy Evaluation

MittCFQ



Disk

MittSSD

Open-Channel SSD

5 real-world block-level traces

DAPPS

DTRS

TPCC

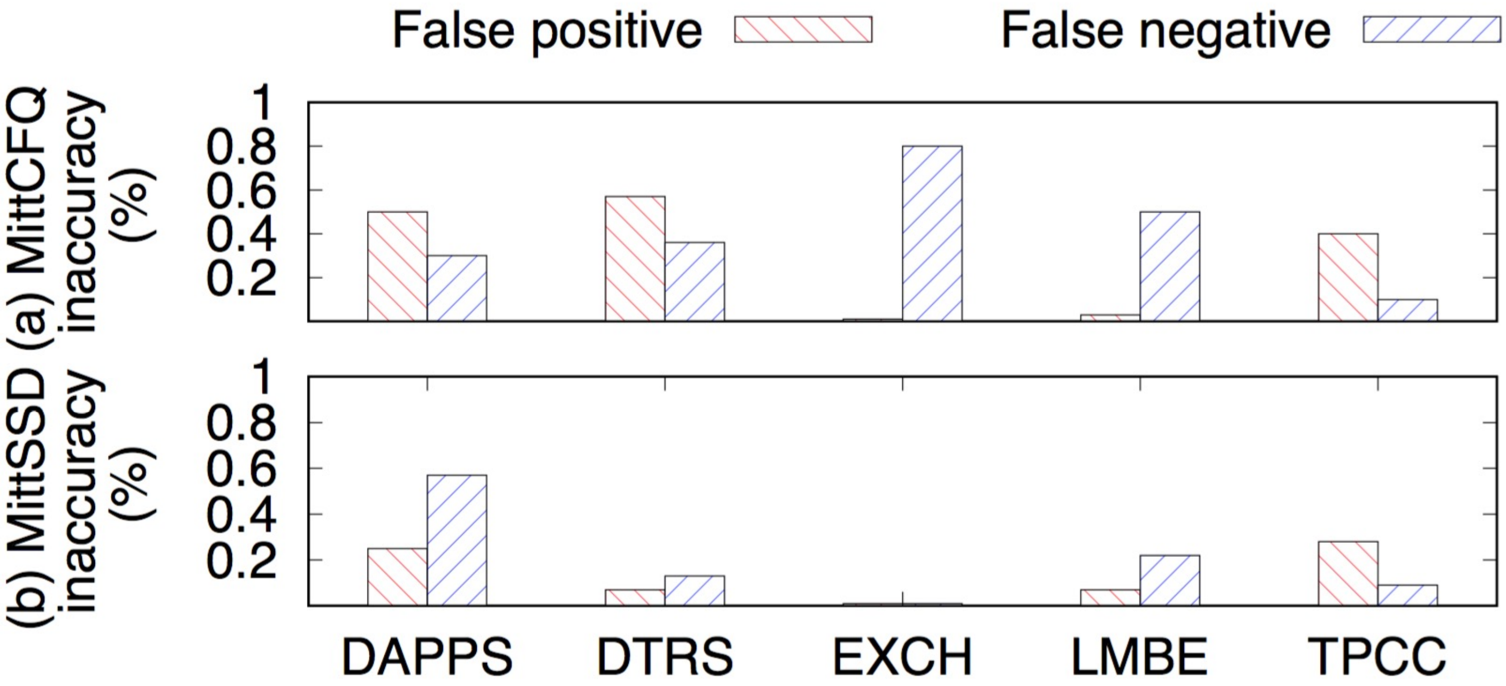
EXCH

LMBE

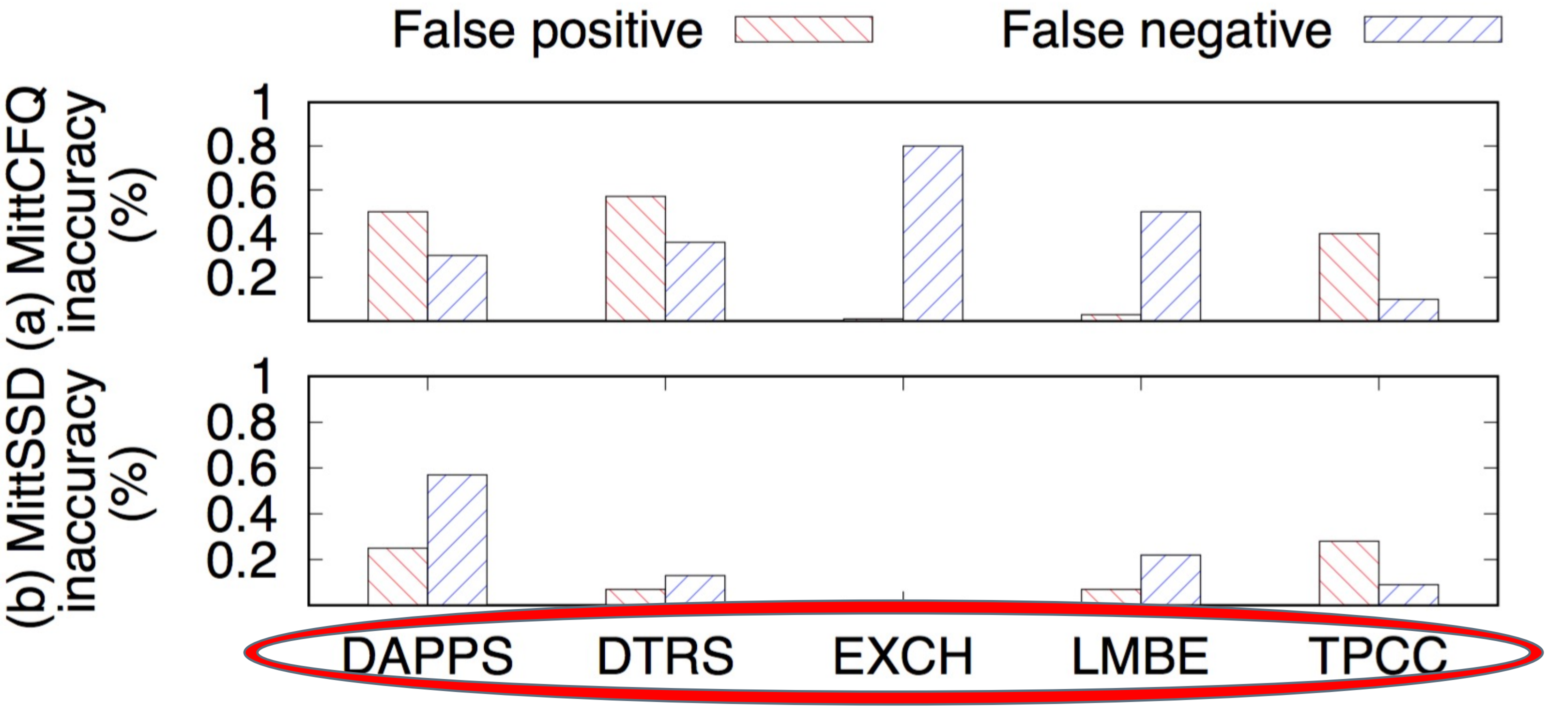
Metrics:

- False positive: IO rejected, but deadline is met
- False negative: Deadline violated, but IO is not rejected

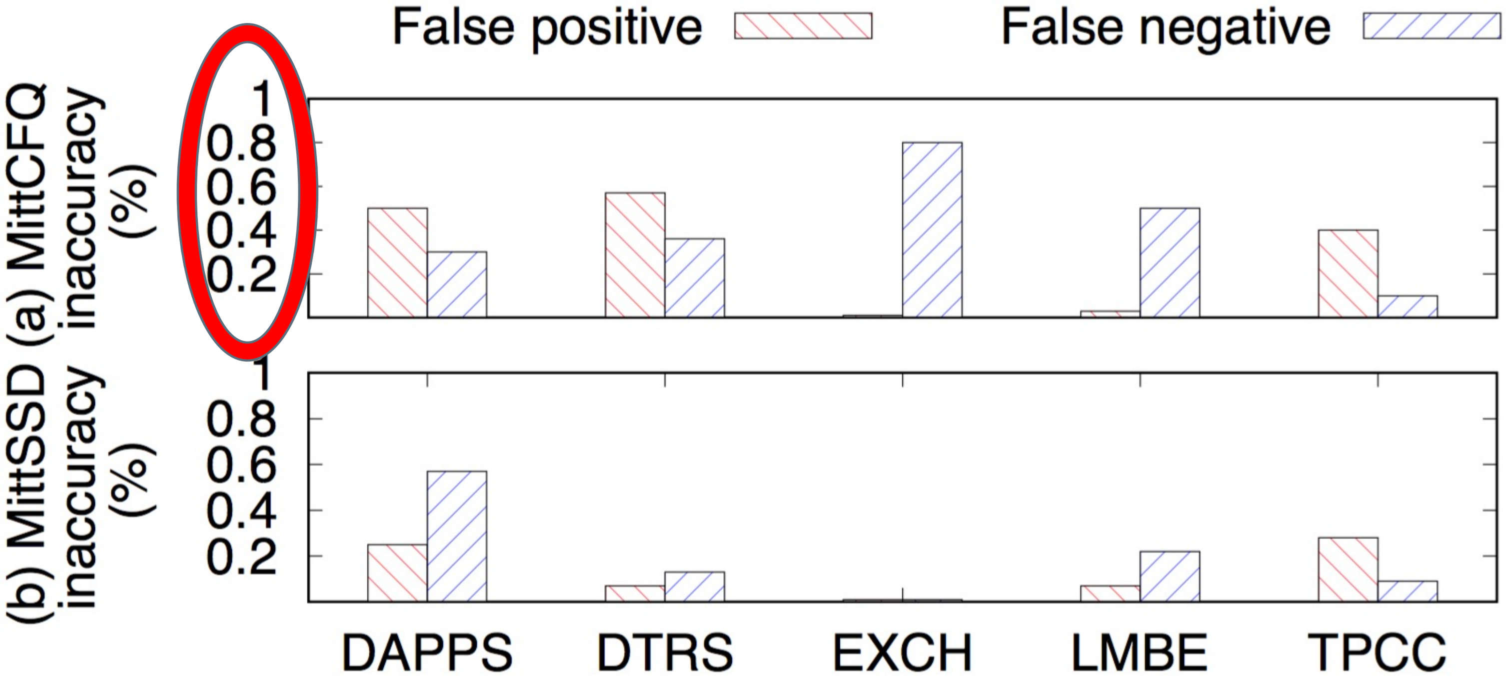
Accuracy Evaluation



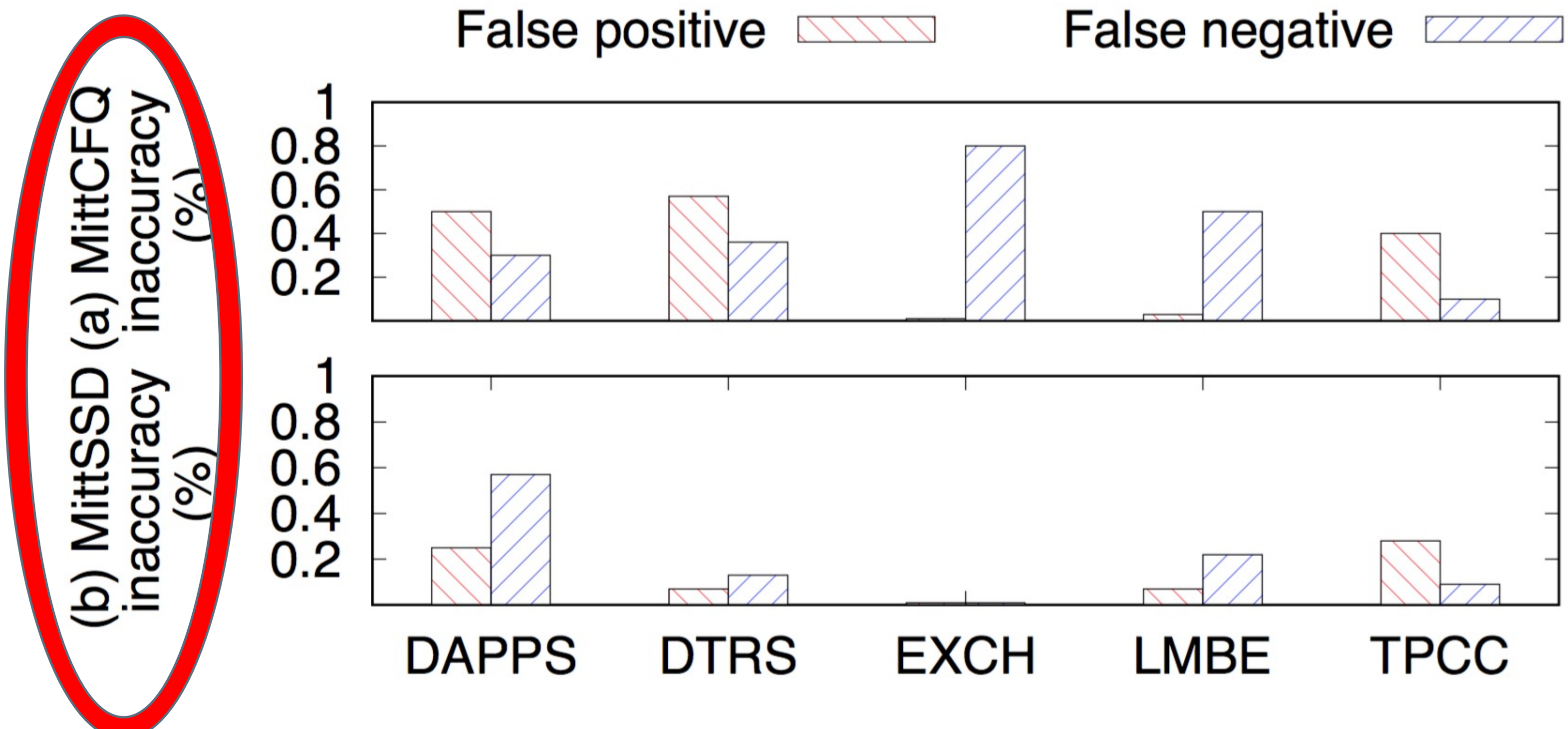
Accuracy Evaluation



Accuracy Evaluation

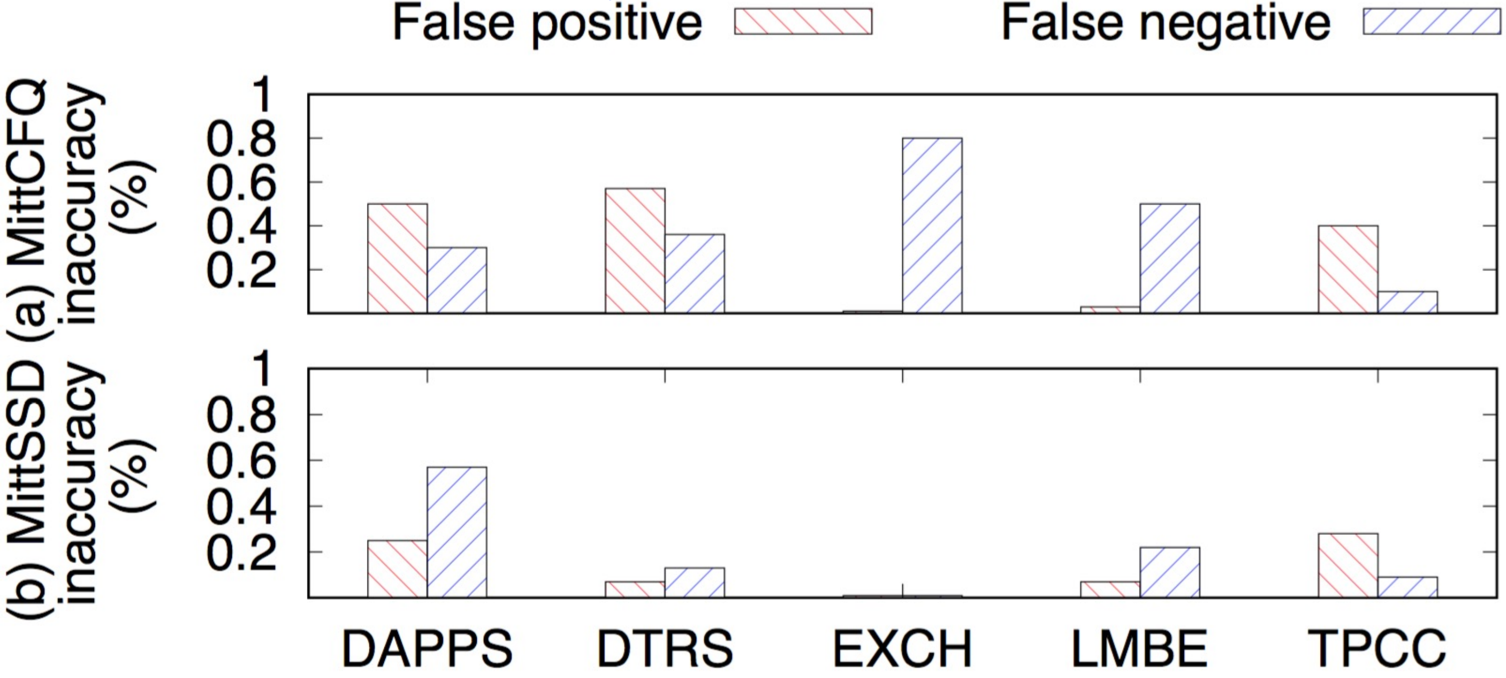


Accuracy Evaluation



Accuracy Evaluation

Only <1% inaccuracy!



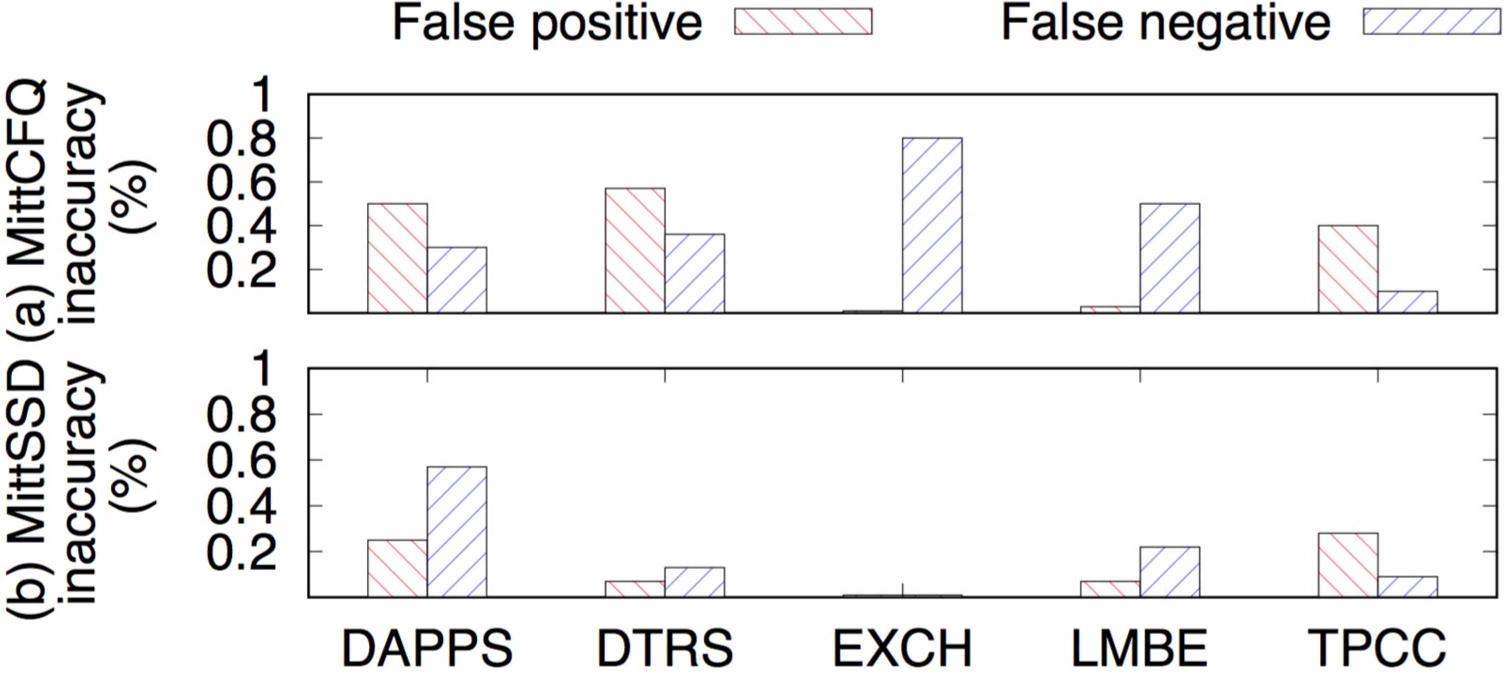
Accuracy Evaluation

Among *incorrect* cases:

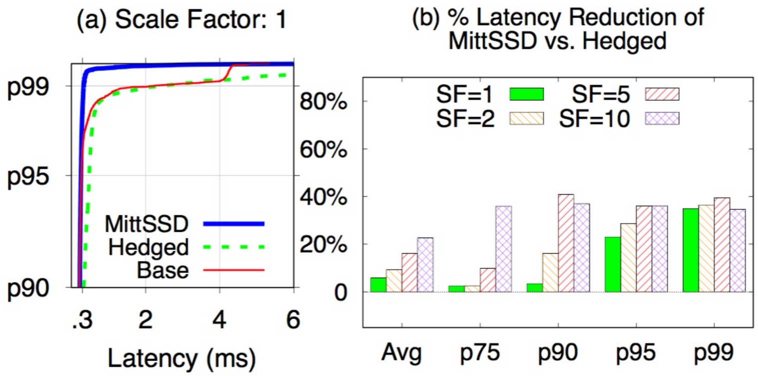
MittCFQ:
< 3ms diff

MittSSD:
< 1ms diff

Only <1% inaccuracy!



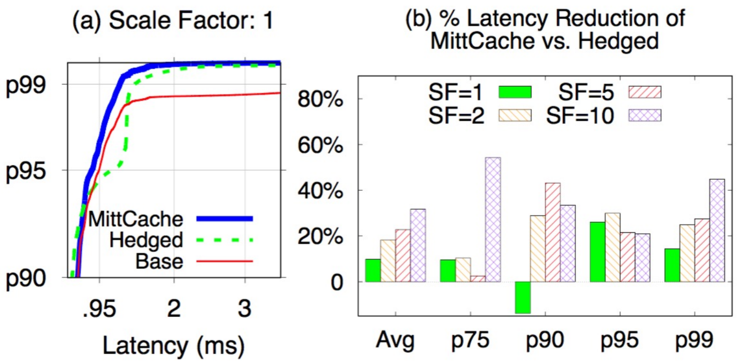
MittSSD



(a) Scale Factor: 1

(b) % Latency Reduction of MittSSD vs. Hedged

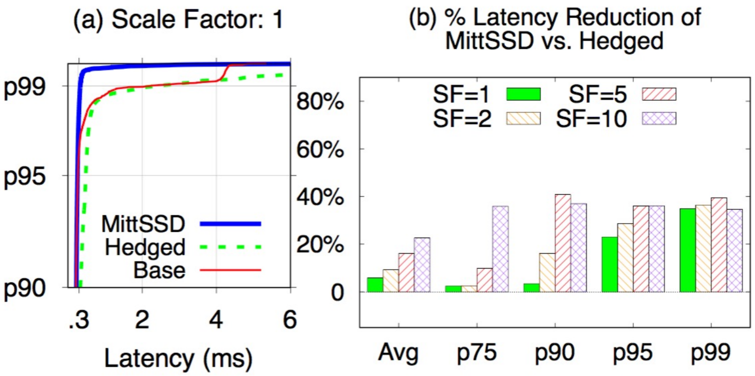
MittCache



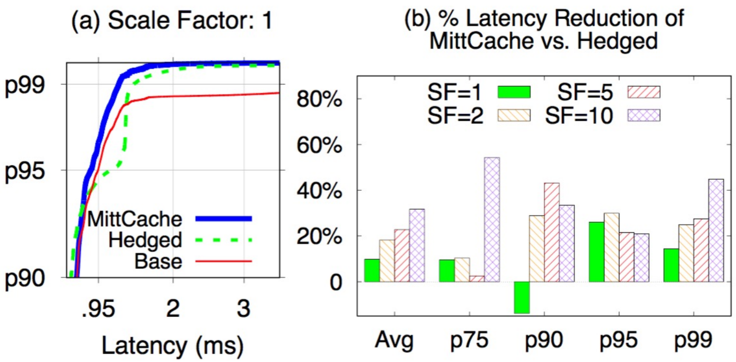
(a) Scale Factor: 1

(b) % Latency Reduction of MittCache vs. Hedged

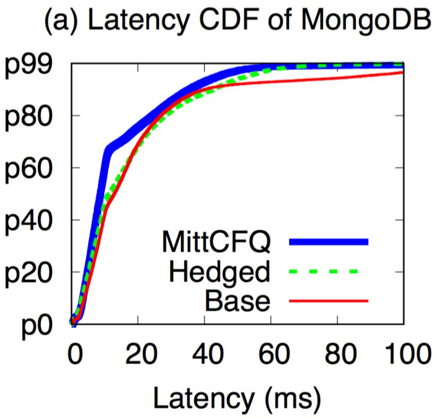
MittSSD



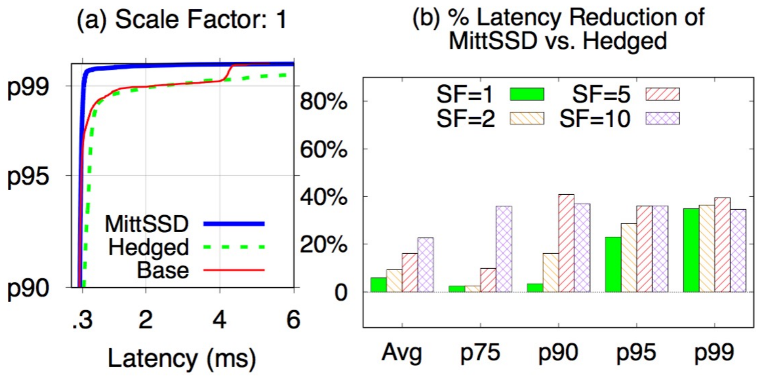
MittCache



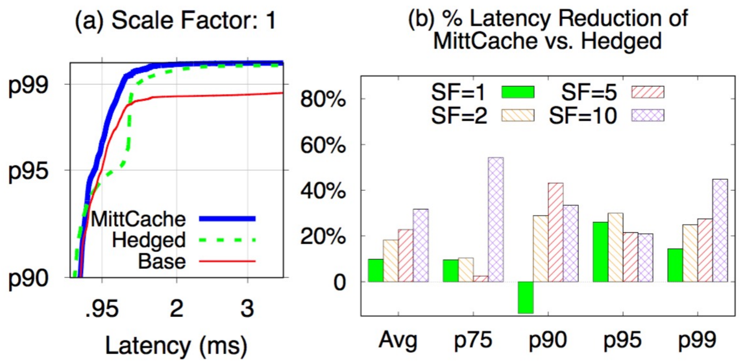
MongoDB + Filebench + Hadoop



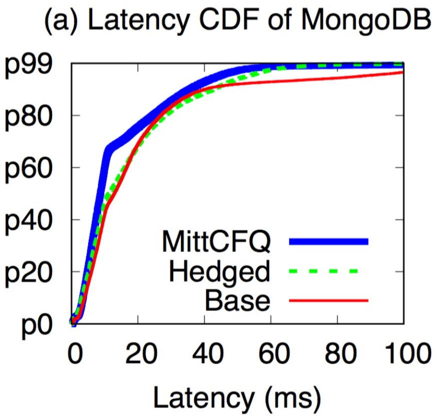
MittSSD


(a) Scale Factor: 1
(b) % Latency Reduction of MittSSD vs. Hedged

MittCache


(a) Scale Factor: 1
(b) % Latency Reduction of MittCache vs. Hedged

MongoDB + Filebench + Hadoop

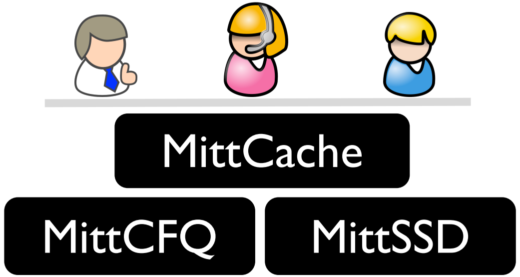

(a) Latency CDF of MongoDB

All in one

SLO 13ms

SLO 20ms

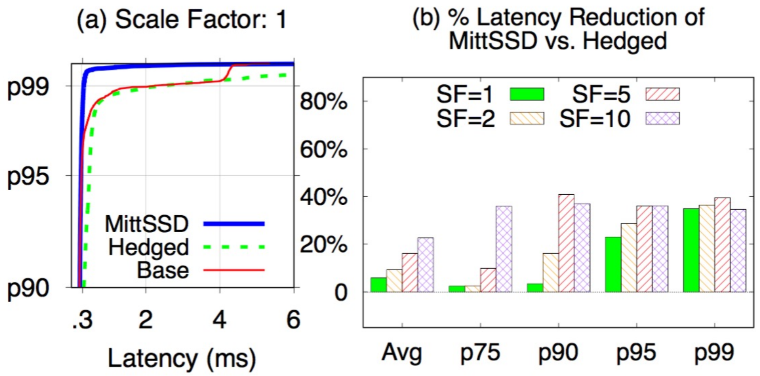
SLO 5ms


MittCache

MittCFQ

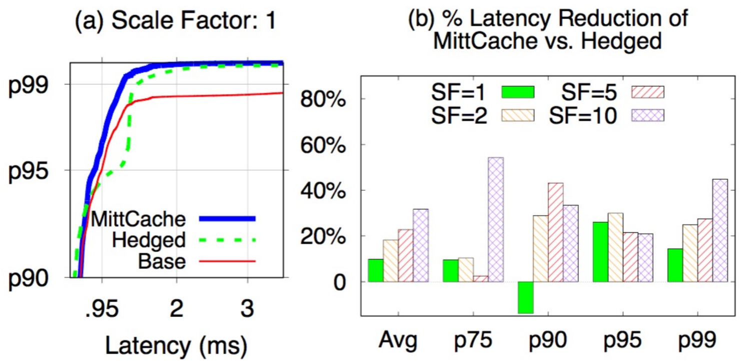
MittSSD

MittSSD


(a) Scale Factor: 1

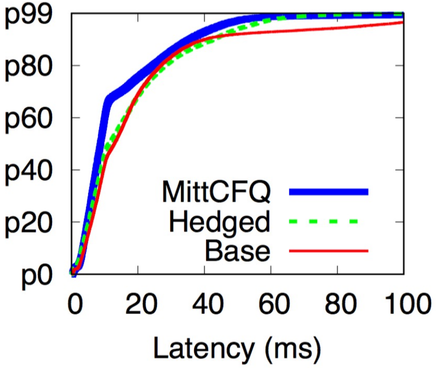
(b) % Latency Reduction of MittSSD vs. Hedged

MittCache

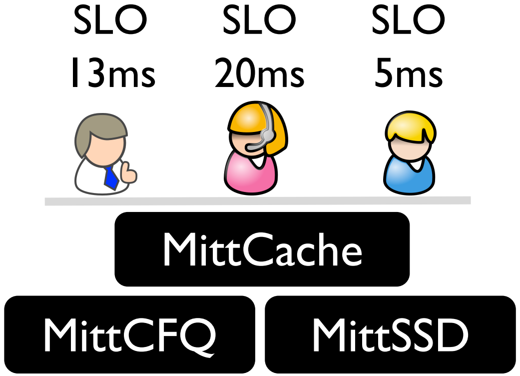

(a) Scale Factor: 1

(b) % Latency Reduction of MittCache vs. Hedged

MongoDB + Filebench + Hadoop


(a) Latency CDF of MongoDB

All in one


SLO 13ms

SLO 20ms

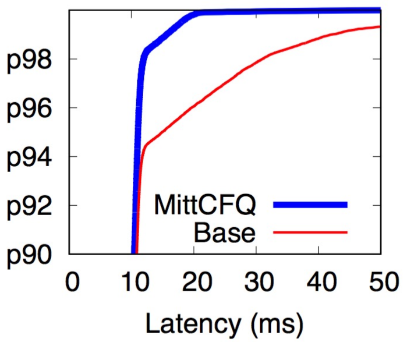
SLO 5ms

MittCache

MittCFQ

MittSSD

Riak


(a) Latency CDF of Riak

Conclusion

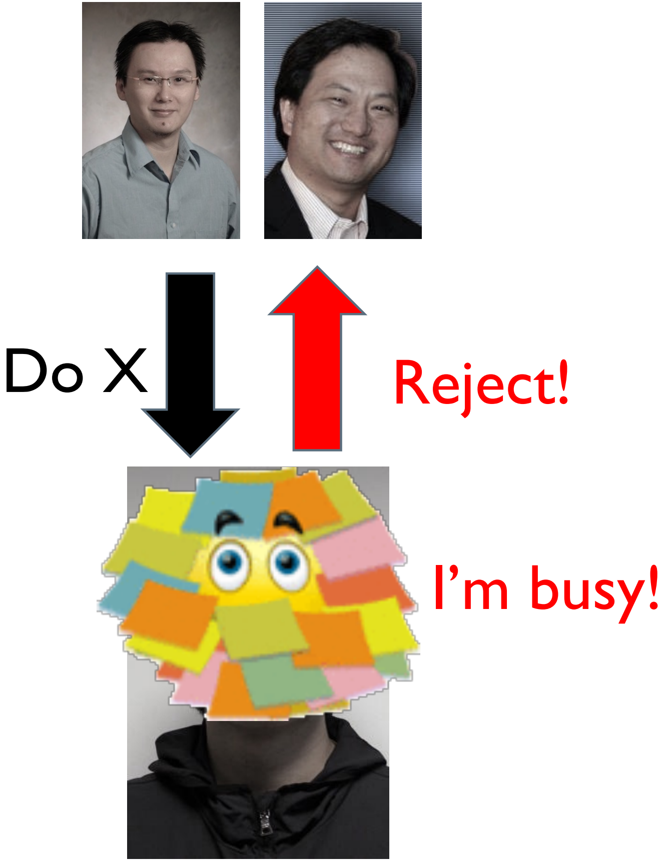
Do X

Conclusion

Do X

I'm busy!

Conclusion

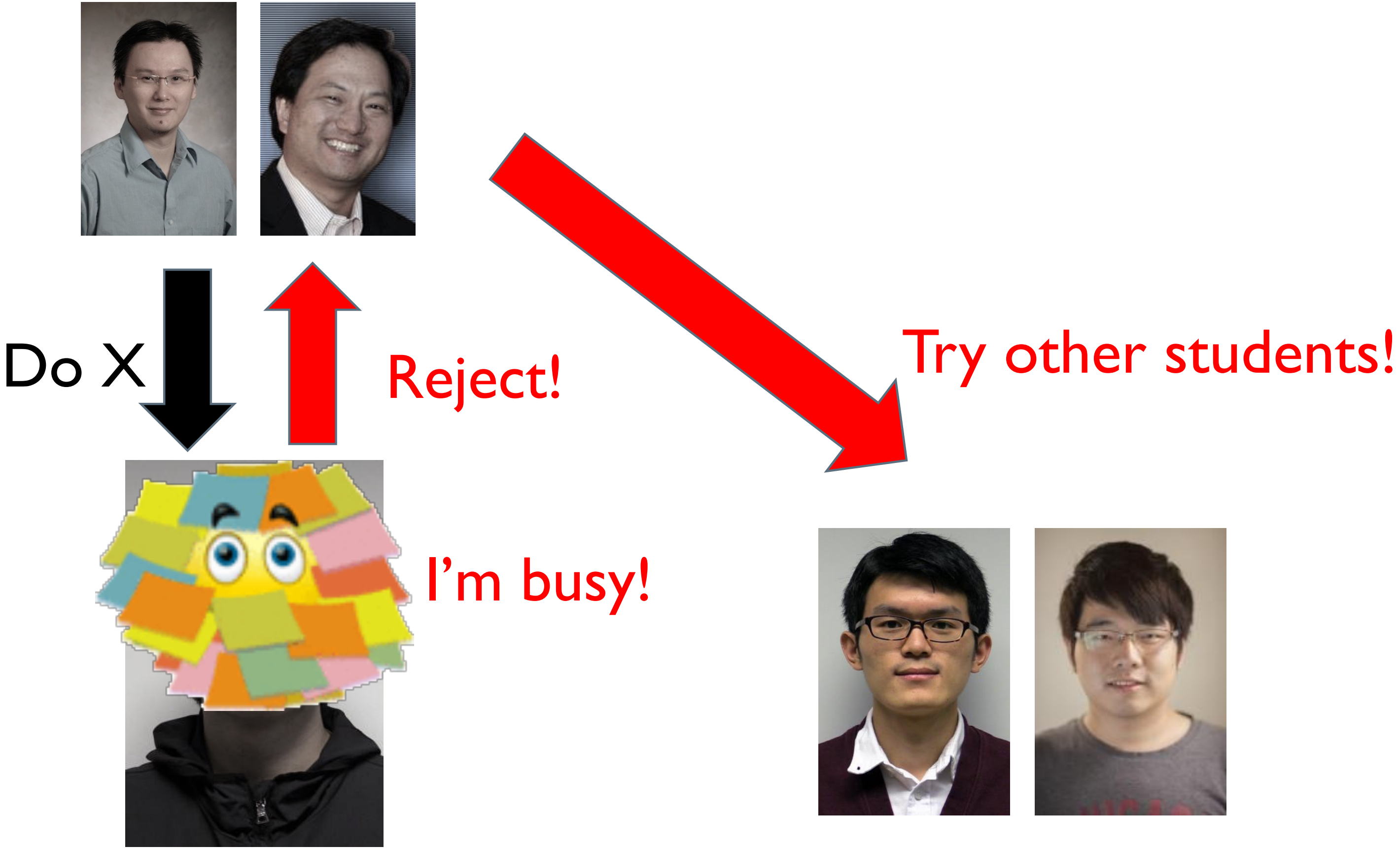


Do X

Reject!

I'm busy!

Conclusion



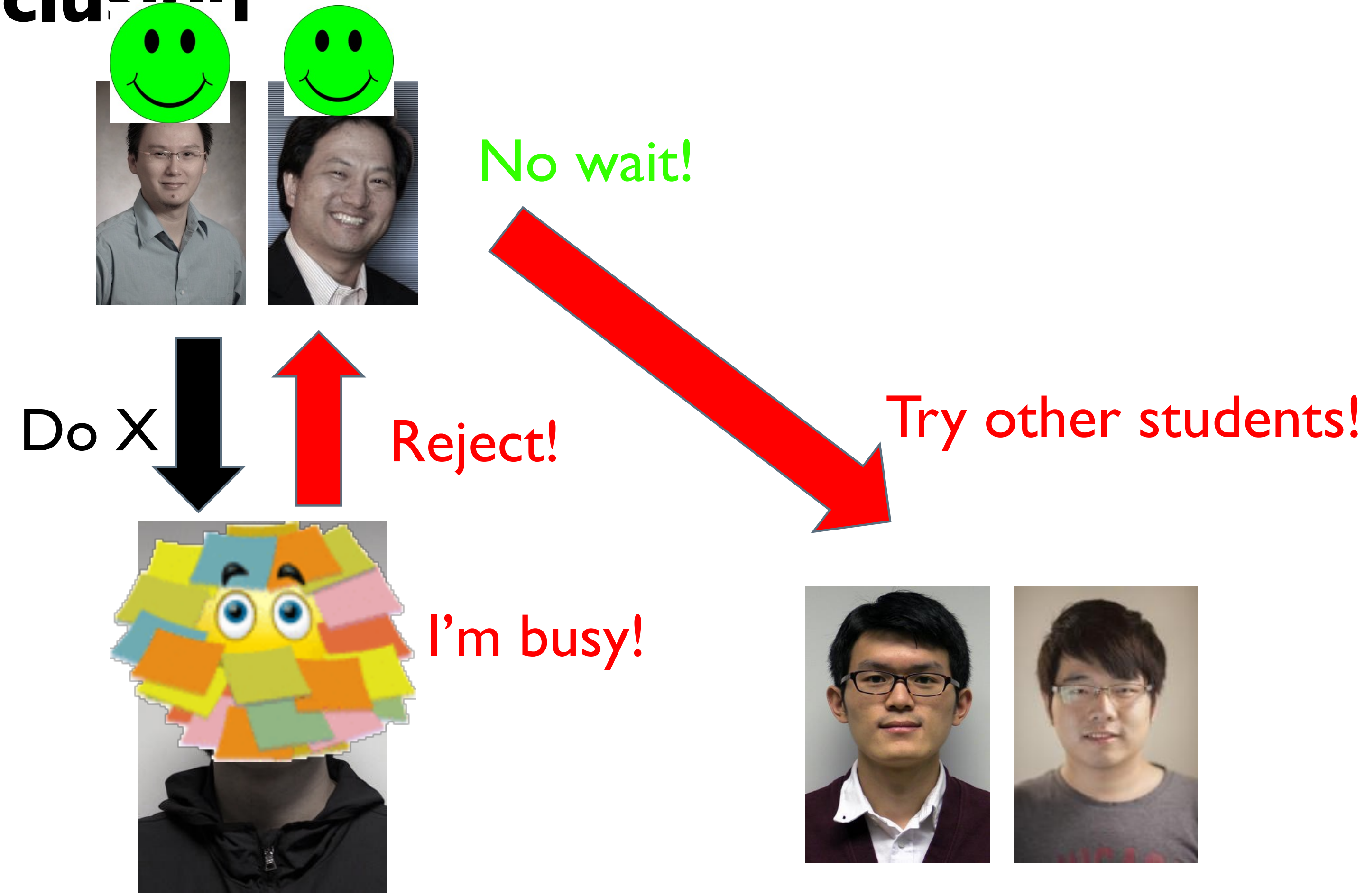
Do X

Reject!

Try other students!

I'm busy!

Conclusion



No wait!

Do X

Reject!

I'm busy!

Try other students!

Conclusion

Fast Reject (No-wait) Interface

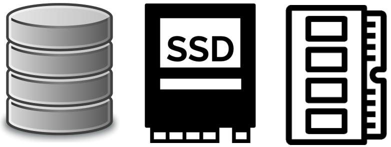
MittOS

Conclusion

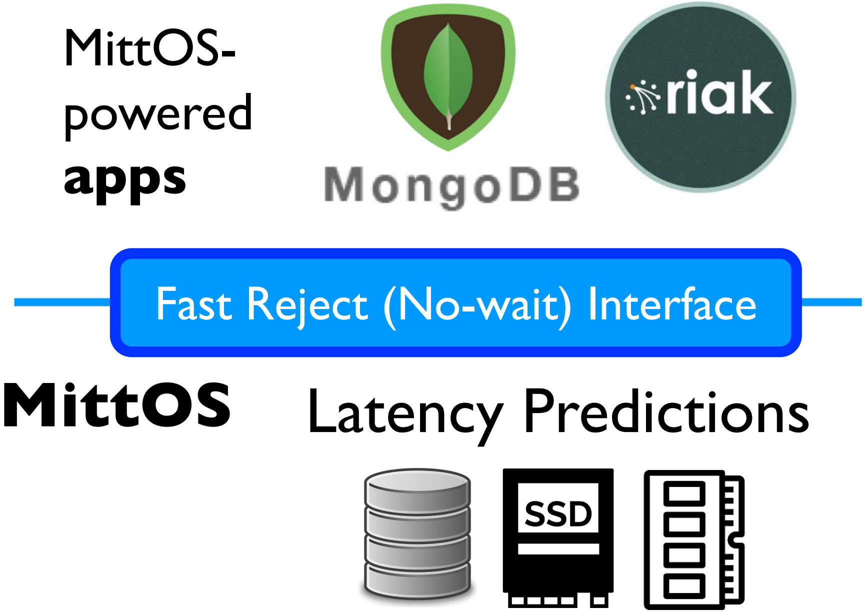
Fast Reject (No-wait) Interface

MittOS

Latency Predictions



Conclusion

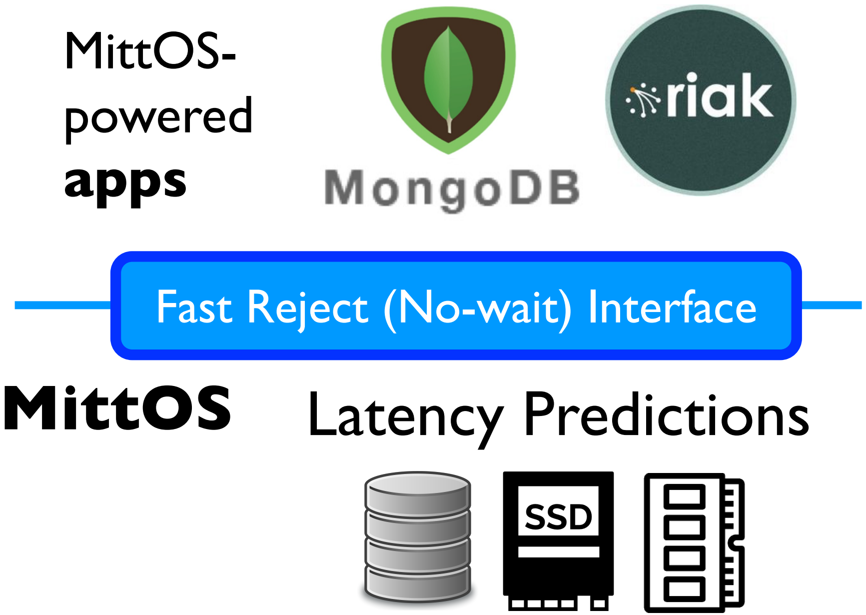


MittOS-
powered
apps

Fast Reject (No-wait) Interface

MittOS Latency Predictions

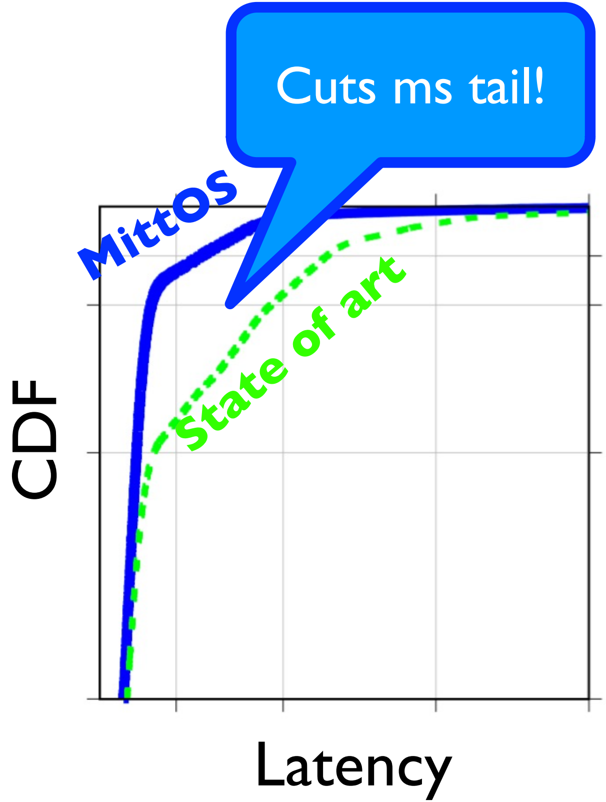
Conclusion



MittOS-powered **apps**

Fast Reject (No-wait) Interface

MittOS Latency Predictions



Cuts ms tail!

Conclusion

MittOS-powered **apps**

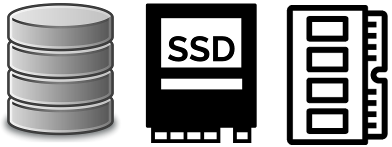
MongoDB

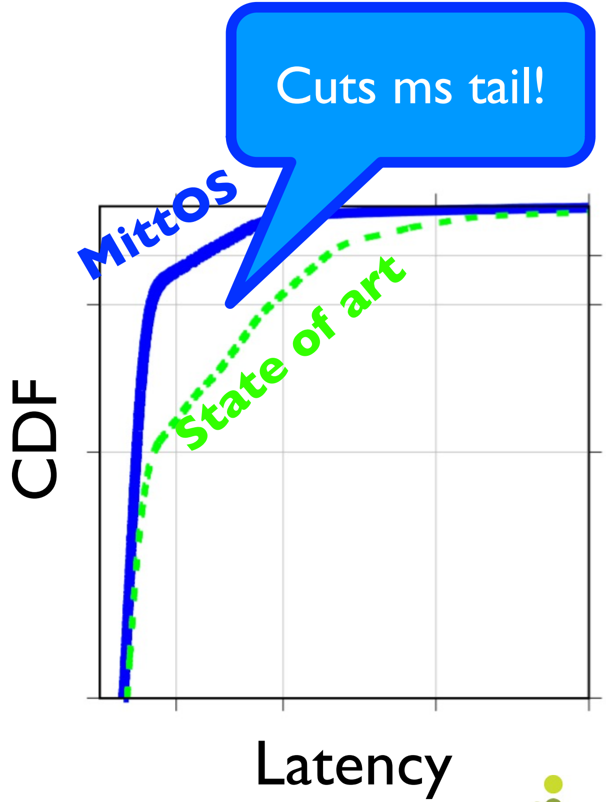
riak

Fast Reject (No-wait) Interface

MittOS

Latency Predictions





Cuts ms tail!

UCARE

http://ucare.cs.uchicago.edu

Thank you! Questions?

CERES Center for Unstoppable Computing

https://ceres.uchicago.edu