WISCONSIN
UNIVERSITY OF WISCONSIN–MADISON

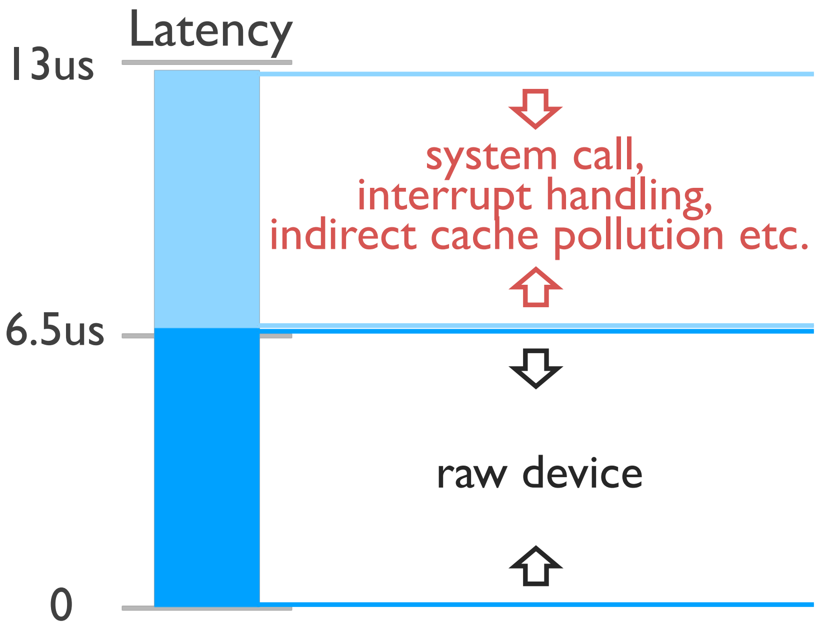
Scale and Performance in a Filesystem Semi-Microkernel

Jing Liu, Anthony Rebello, Yifan Dai, Chenhao Ye,
Sudarsun Kannan*, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau

University of Wisconsin – Madison
*Rutgers University**

HW is Fast – but SW Appears Slow

How to close the HW-SW performance gap in storage stack?

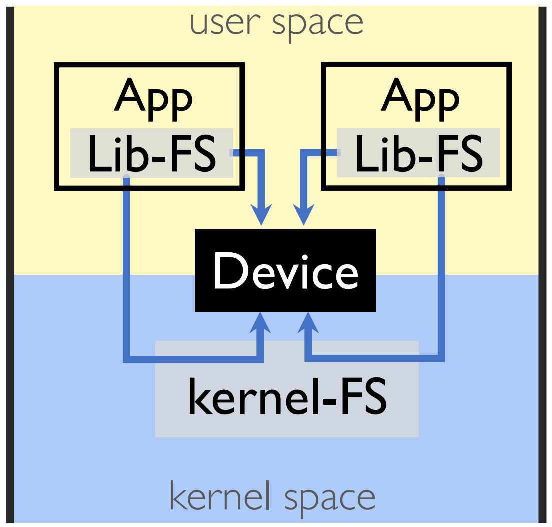


Barroso *et. al*, Attack of the Killer Microseconds, 2017

Existing Solutions

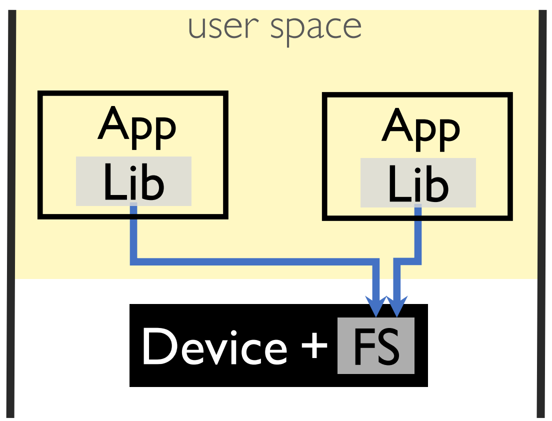
Libraries directly access the device

- E.g., Strata (SOSP-17), SplitFS (SOSP-19)
- Complicate the device access isolation and sharing



Move Filesystems to the device

- E.g., DevFS (FAST-18), CrossFS (OSDI-20)
- “Smarter-HW” assumption and unknown HW constraints

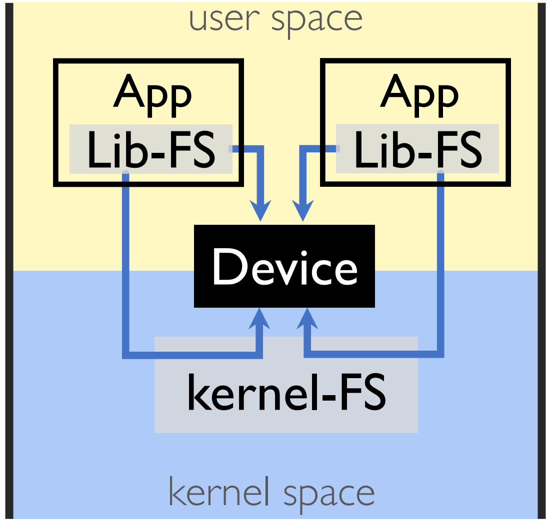


Existing Solutions

~~Libraries directly access the device~~

- ~~E.g., Strata (SOSP-17), SplitFS (SOSP-19)~~
- ~~Complicate the device access isolation and sharing~~

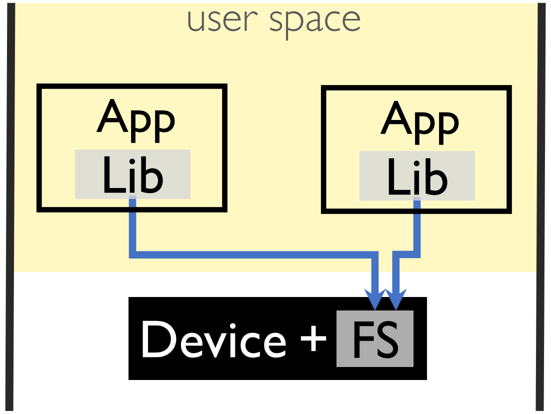
Centralized IO multiplexing; simpler isolation and sharing



~~Move Filesystems to the device~~

- ~~E.g., DevFS (FAST-18), CrossFS (OSDI-20)~~
- ~~"Smarter-HW" assumption and unknown HW constraints~~

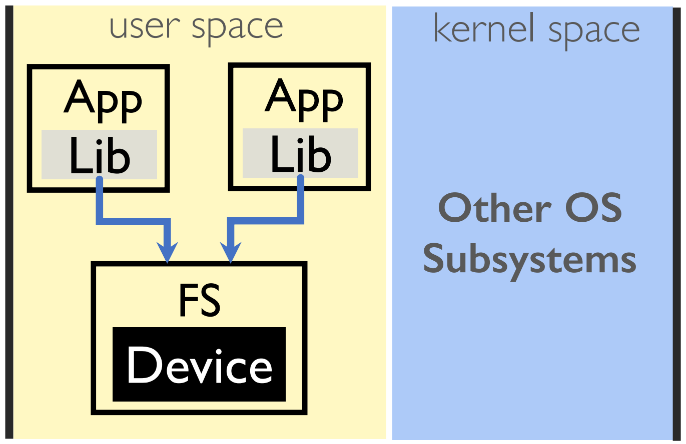
Realistic Assumption: Ultra-fast Devices and NVMe protocol



Our Approach: Filesystem Semi-Microkernel

What is a "Semi-Microkernel"?

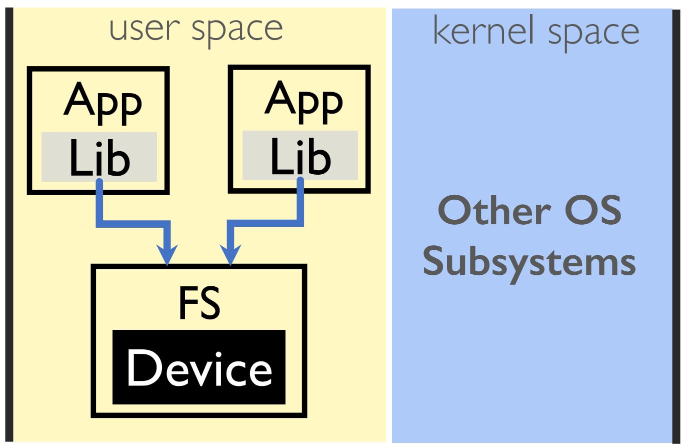
- An OS subsystem that runs as a user-level process
- Works in tandem with the monolithic kernel



Our Approach: Filesystem Semi-Microkernel

What is a "Semi-Microkernel"?

- An OS subsystem that runs as a user-level process
- Works in tandem with the monolithic kernel



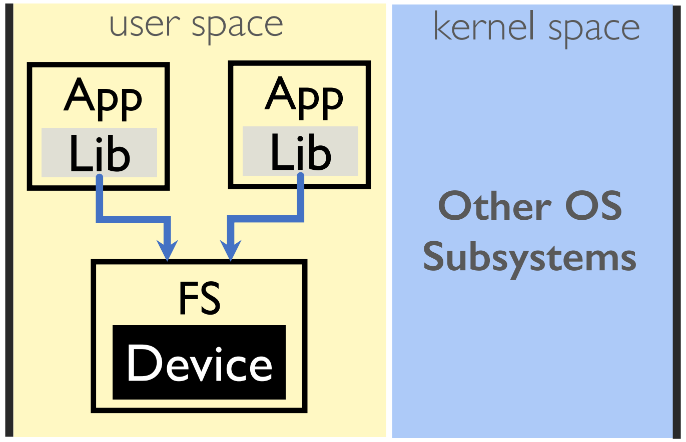
Prior networking semi-microkernels

- Snap (SOSP-19), TAS (Eurosys-19)

Possible for storage now

- User-level device drivers

Benefits of Filesystem Semi-Microkernels



Development and Deployment Velocity

- Developing tools and libraries for "application" code
- Rapidly adopt hardware and tailor for applications

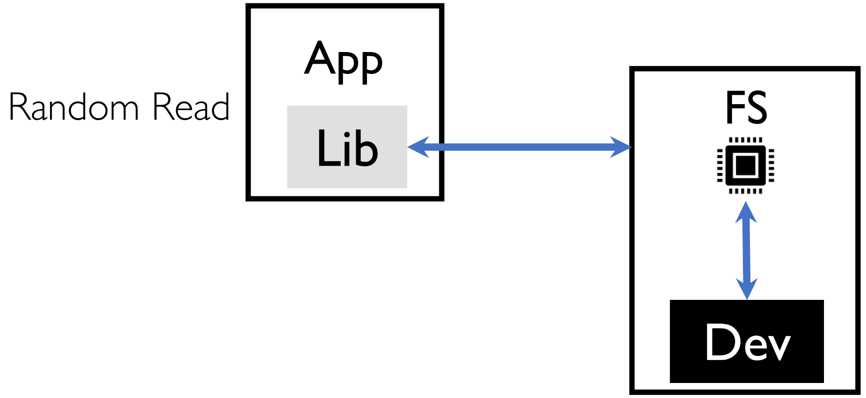
Performance

- Optimize for device access (avoid the kernel SW overhead)
- Scale filesystem independently from applications

Simplify the sharing and permission

- Untrusted applications cannot access the device

Challenges



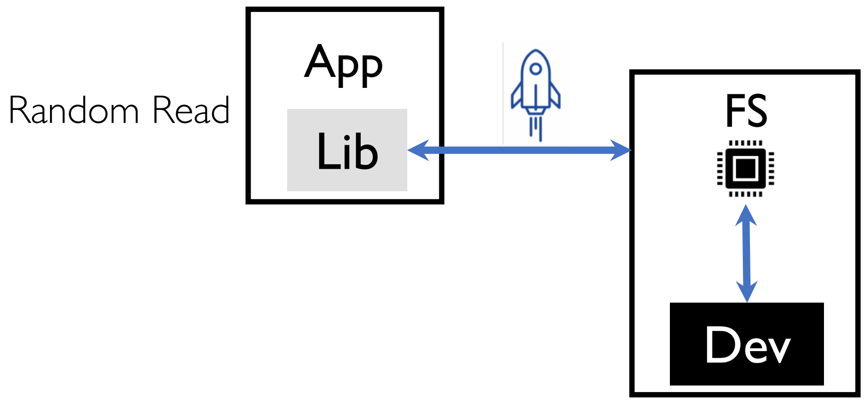
Base Performance

- Inter-process communication
- Device access

Challenges

Base Performance

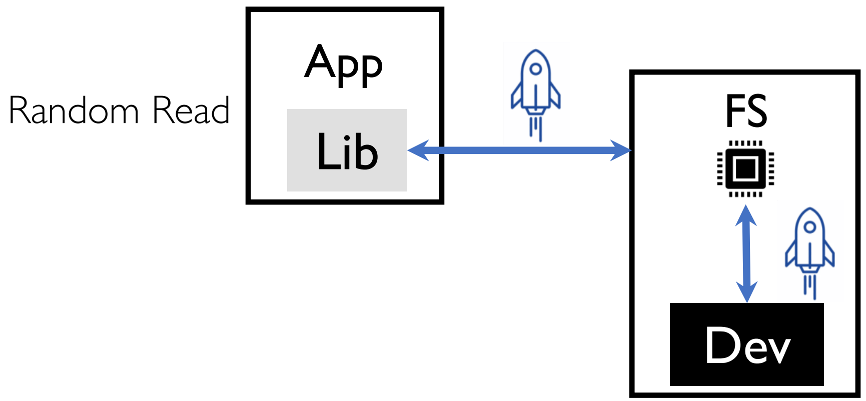
- Inter-process communication
- Device access



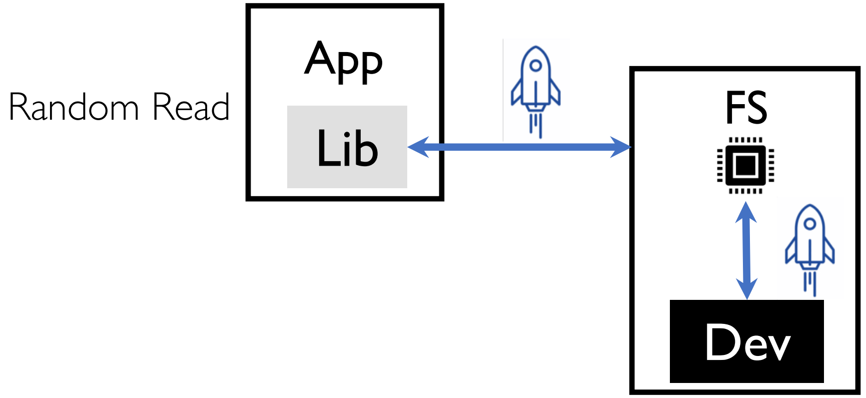
Challenges

Base Performance

- Inter-process communication
- Device access



Challenges



Base Performance

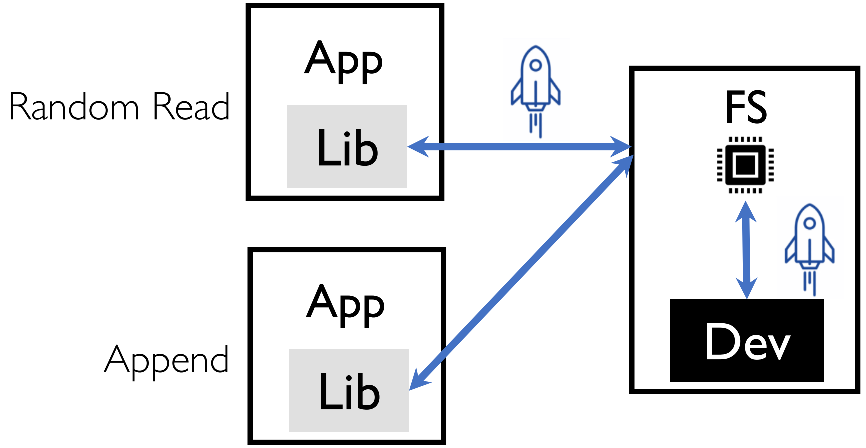
- Inter-process communication
- Device access

Scale up and down

- Dynamic and heterogeneous application demands
- Invest just-right amount of CPU
  - Fully utilize the devices
  - Keep up with the apps

simultaneously

Challenges



Base Performance

- Inter-process communication
- Device access

Scale up and down

- Dynamic and heterogeneous application demands
- Invest just-right amount of CPU
  - Fully utilize the devices
  - Keep up with the apps

  } simultaneously

Challenges

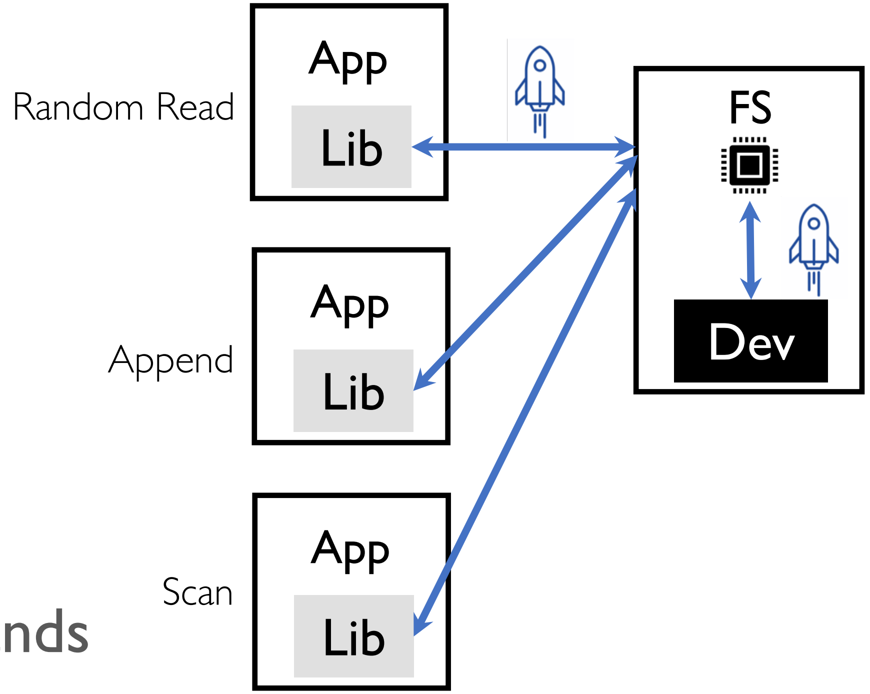
Base Performance

- Inter-process communication
- Device access

Scale up and down

- Dynamic and heterogeneous application demands
- Invest just-right amount of CPU
    - Fully utilize the devices
    - Keep up with the apps

simultaneously



Challenges

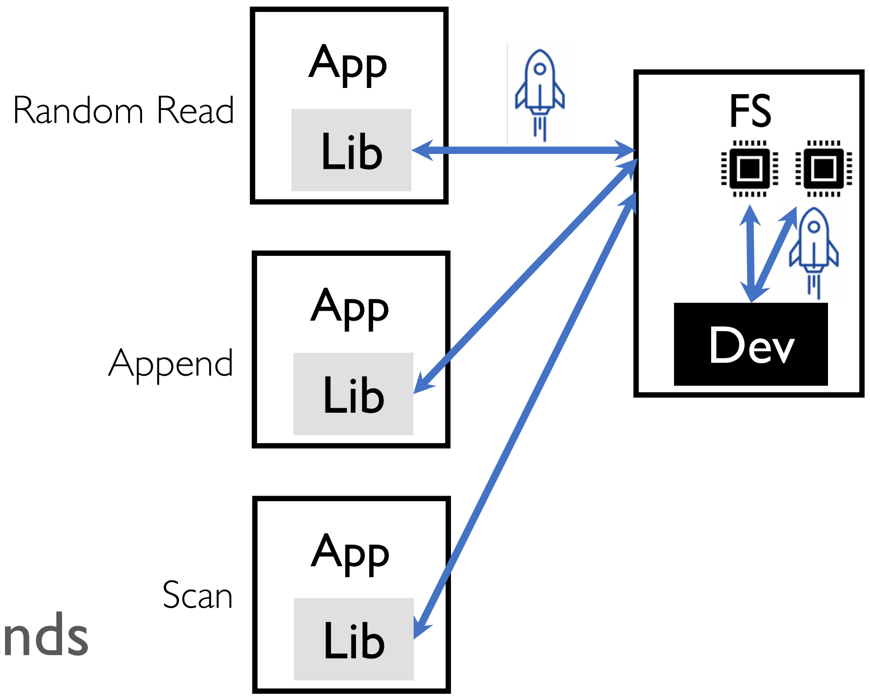
Base Performance

- Inter-process communication
- Device access

Scale up and down

- Dynamic and heterogeneous application demands
- Invest just-right amount of CPU
  - Fully utilize the devices
  - Keep up with the apps

simultaneously



Challenges

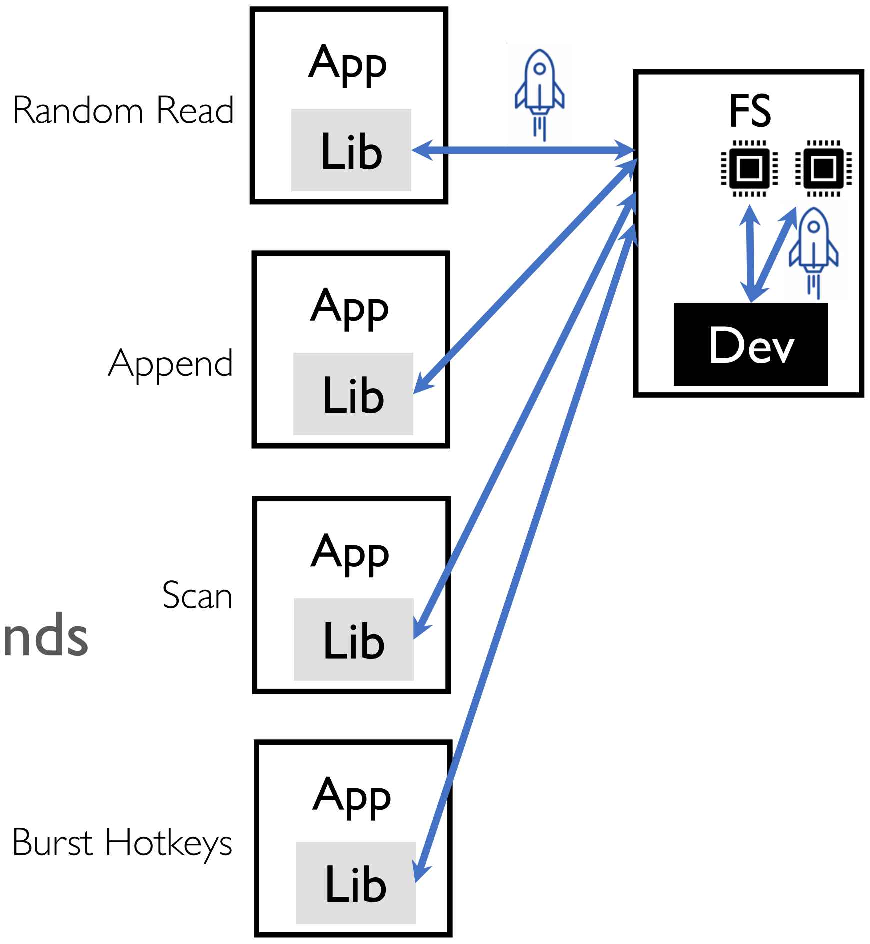
Base Performance

- Inter-process communication
- Device access

Scale up and down

- Dynamic and heterogeneous application demands
- Invest just-right amount of CPU
  - Fully utilize the devices
  - Keep up with the apps

  } simultaneously



Challenges

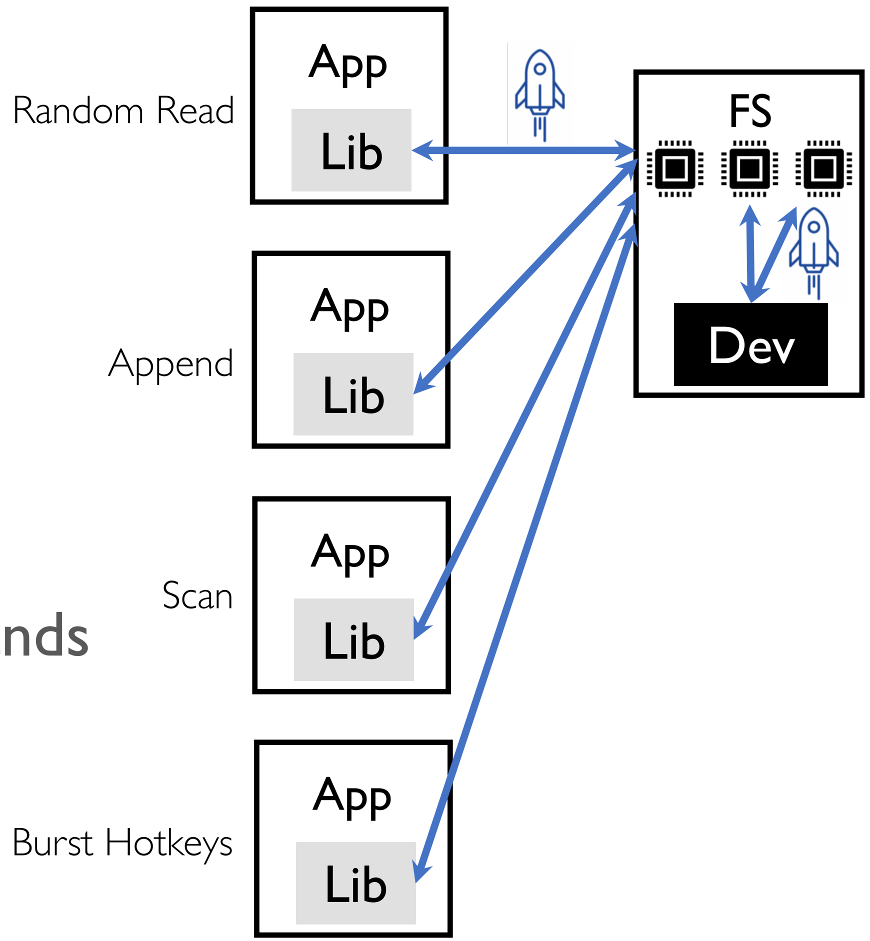
Base Performance

- Inter-process communication
- Device access

Scale up and down

- Dynamic and heterogeneous application demands
- Invest just-right amount of CPU
  - Fully utilize the devices
  - Keep up with the apps

simultaneously



Challenges

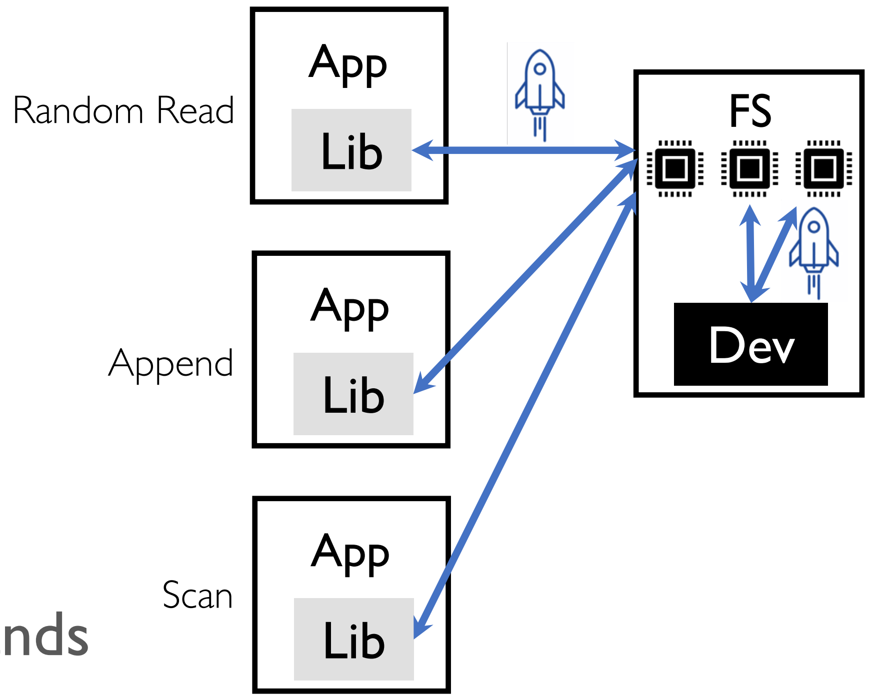
Base Performance

- Inter-process communication
- Device access

Scale up and down

- Dynamic and heterogeneous application demands
- Invest just-right amount of CPU
  - Fully utilize the devices
  - Keep up with the apps

  } simultaneously



Challenges

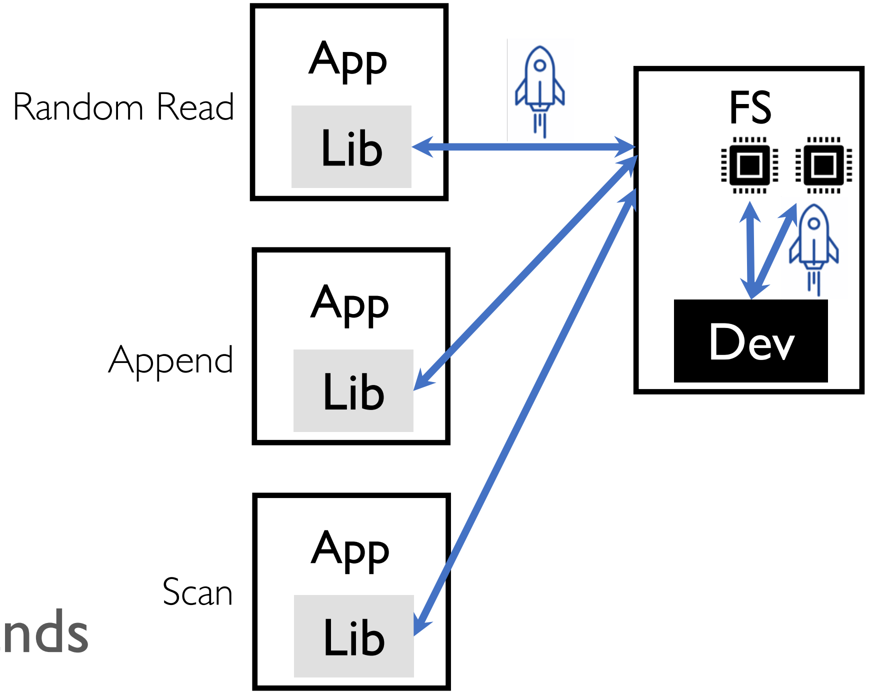
Base Performance

- Inter-process communication
- Device access

Scale up and down

- Dynamic and heterogeneous application demands
- Invest just-right amount of CPU
  - Fully utilize the devices
  - Keep up with the apps

simultaneously



uFS: A Filesystem Semi-microkernel

Build for performance and scalability from scratch

- Fully functional with crash consistency guaranteed by journaling
- Ensure lock-free access for main data structures
- Dynamically partition inodes to filesystem threads
- Adapt # of uFS cores according to filesystem demands
- Implemented by C++ (~35K LoC)

uFS offers good base performance and excellent scalability

- 1.2X-4.6X throughput compared to ext4 when running 10 LevelDB instances

Outline

Introduction

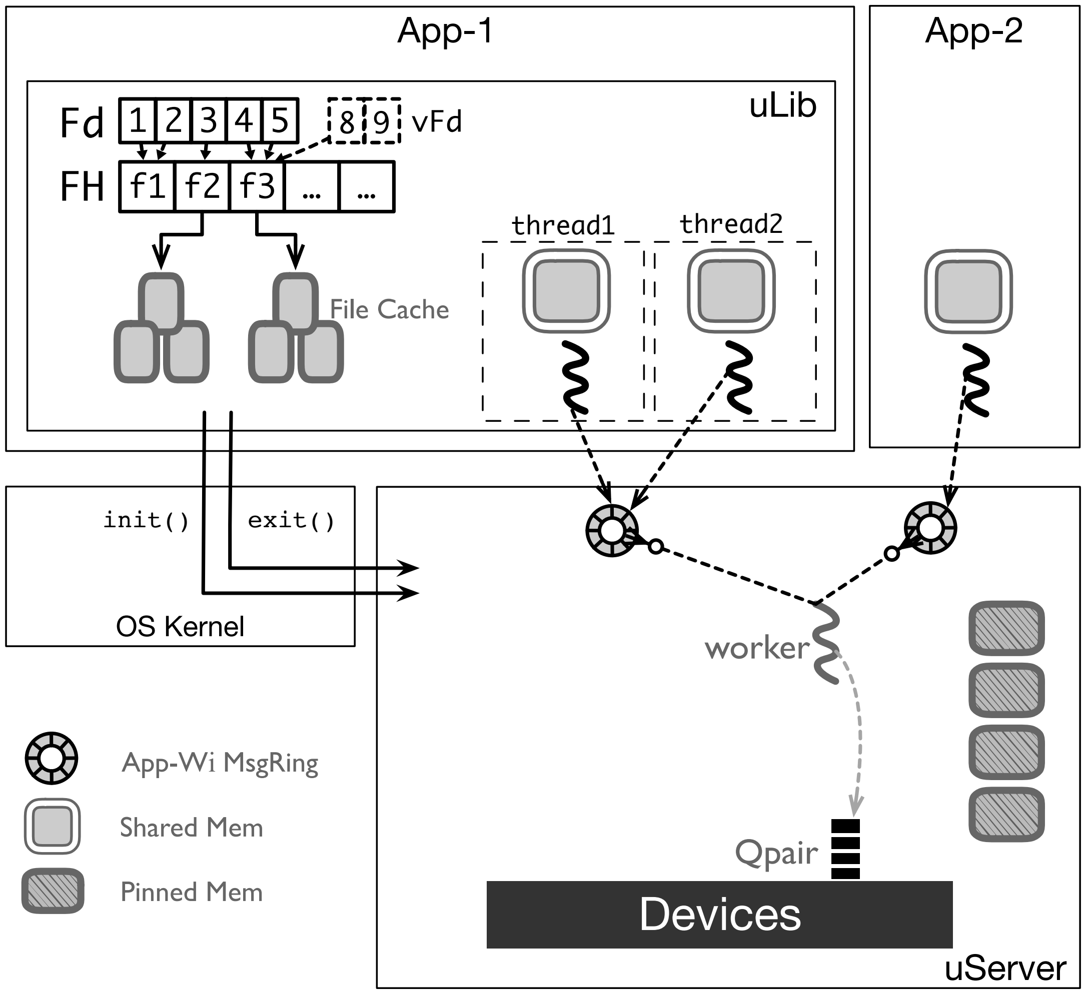
uFS Architecture

Design

Evaluation

Conclusion

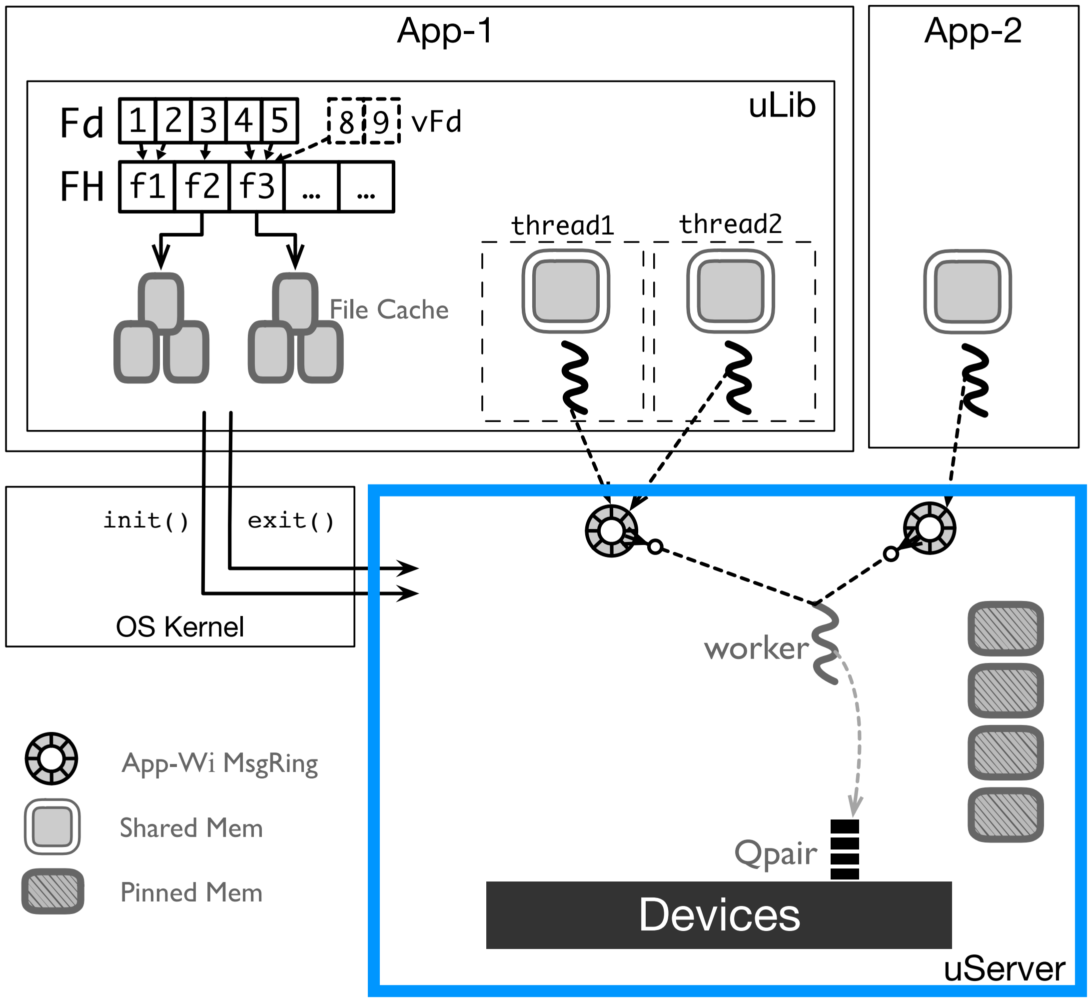
uFS Architecture



uFS Architecture

uServer

- Directly accesses the device via NVMe commands
- Non-blocking: polling the device
- Manage pinned memory as block buffer cache



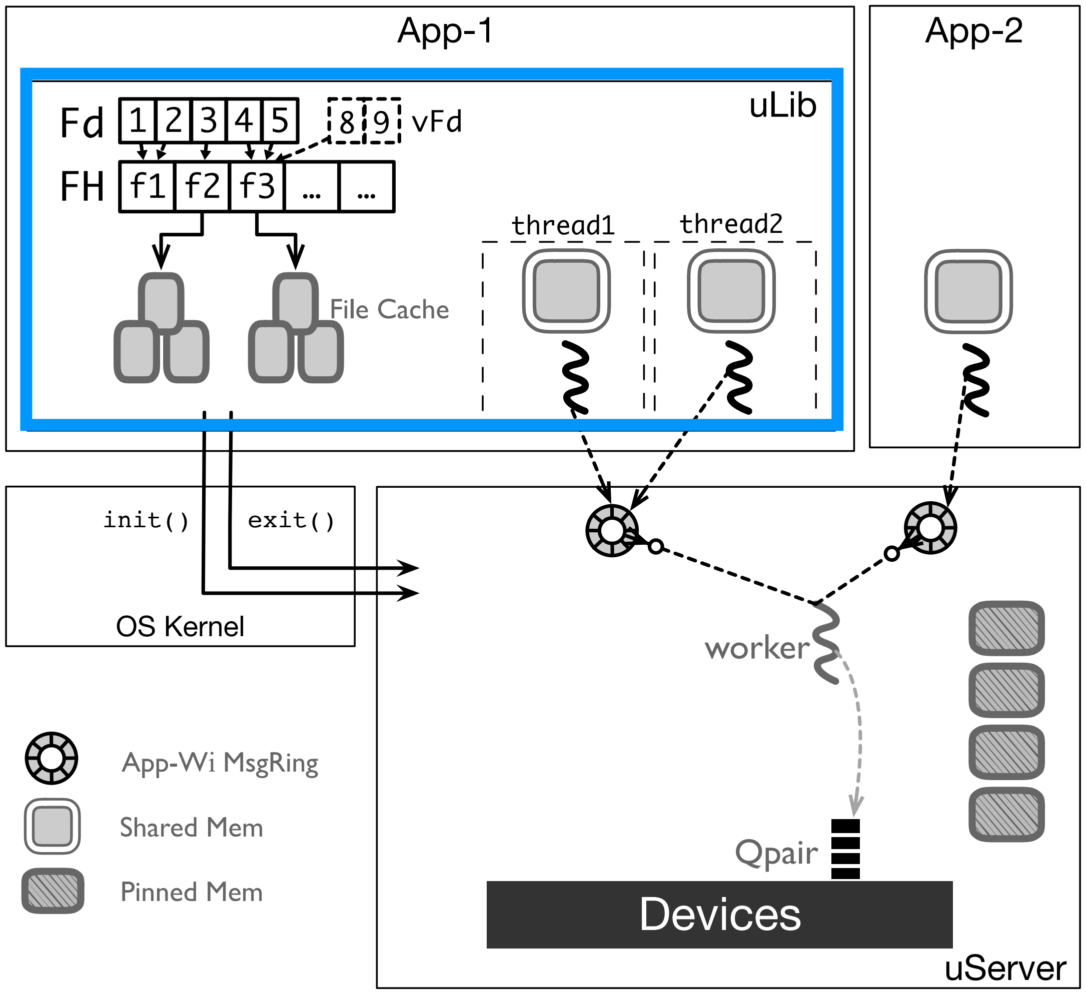
uFS Architecture

uServer

- Directly accesses the device via NVMe commands
- Non-blocking: polling the device
- Manage pinned memory as block buffer cache

uLib

- POSIX-API
- App-integrated file cache (lease-based)
- Open-lease management (vFd)



uFS Architecture

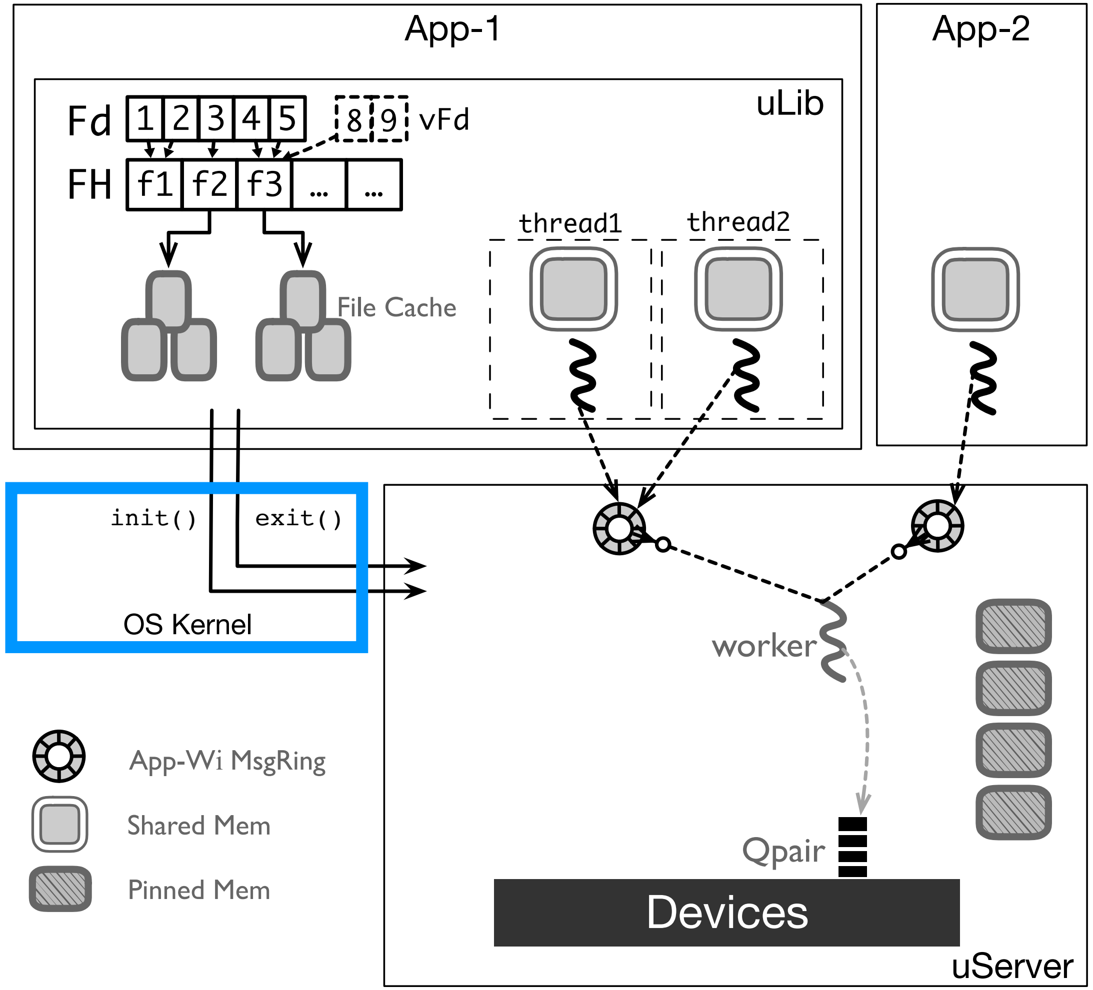
uServer

- Directly accesses the device via NVMe commands
- Non-blocking: polling the device
- Manage pinned memory as block buffer cache

uLib

- POSIX-API
- App-integrated file cache (lease-based)
- Open-lease management (vFd)

The OS kernel only involves for initial authentication (`fs_init`)



uFS Architecture

uServer

- Directly accesses the device via NVMe commands
- Non-blocking: polling the device
- Manage pinned memory as block buffer cache

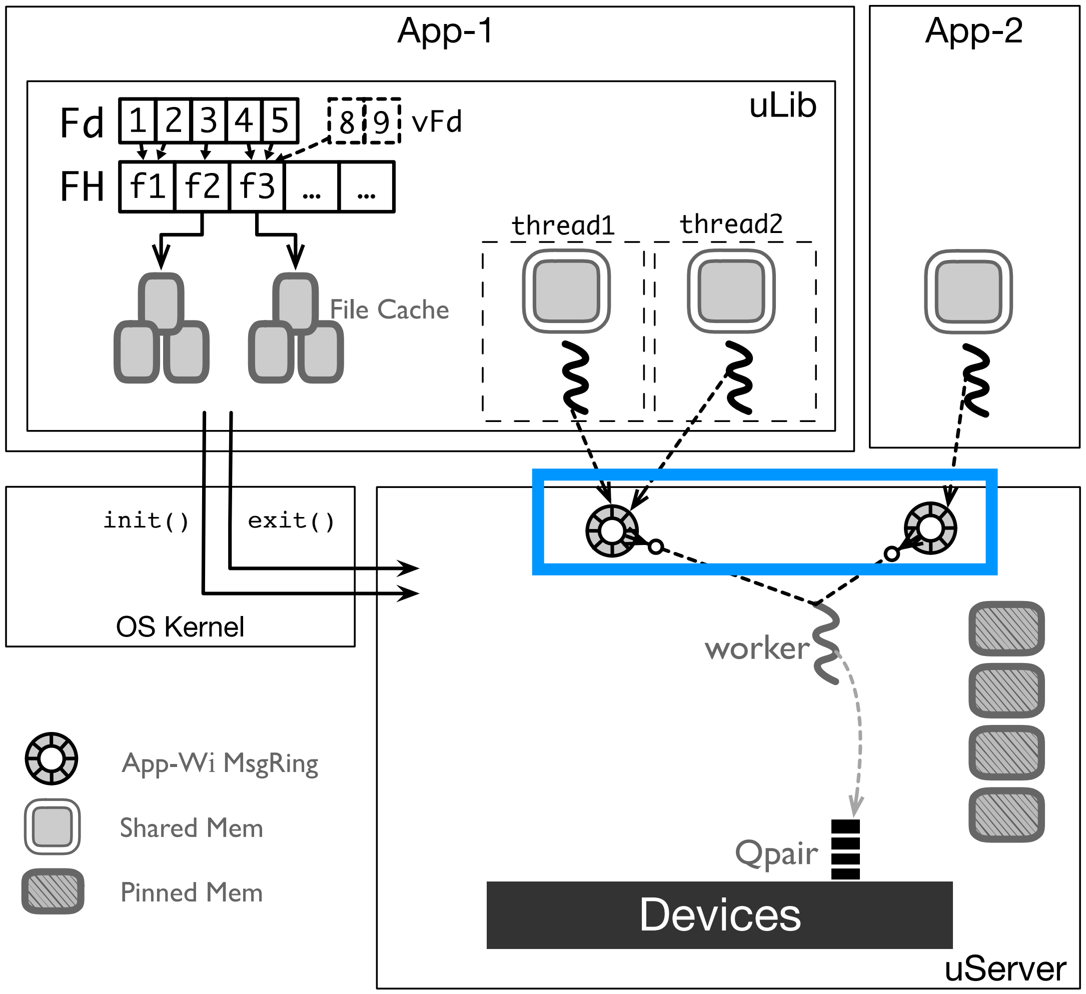
uLib

- POSIX-API
- App-integrated file cache (lease-based)
- Open-lease management (vFd)

The OS kernel only involves for initial authentication (`fs_init`)

uLib ⟷ uServer

- Control: shared-mem IPC (cache-line-size message)



uFS Architecture

uServer

- Directly accesses the device via NVMe commands
- Non-blocking: polling the device
- Manage pinned memory as block buffer cache

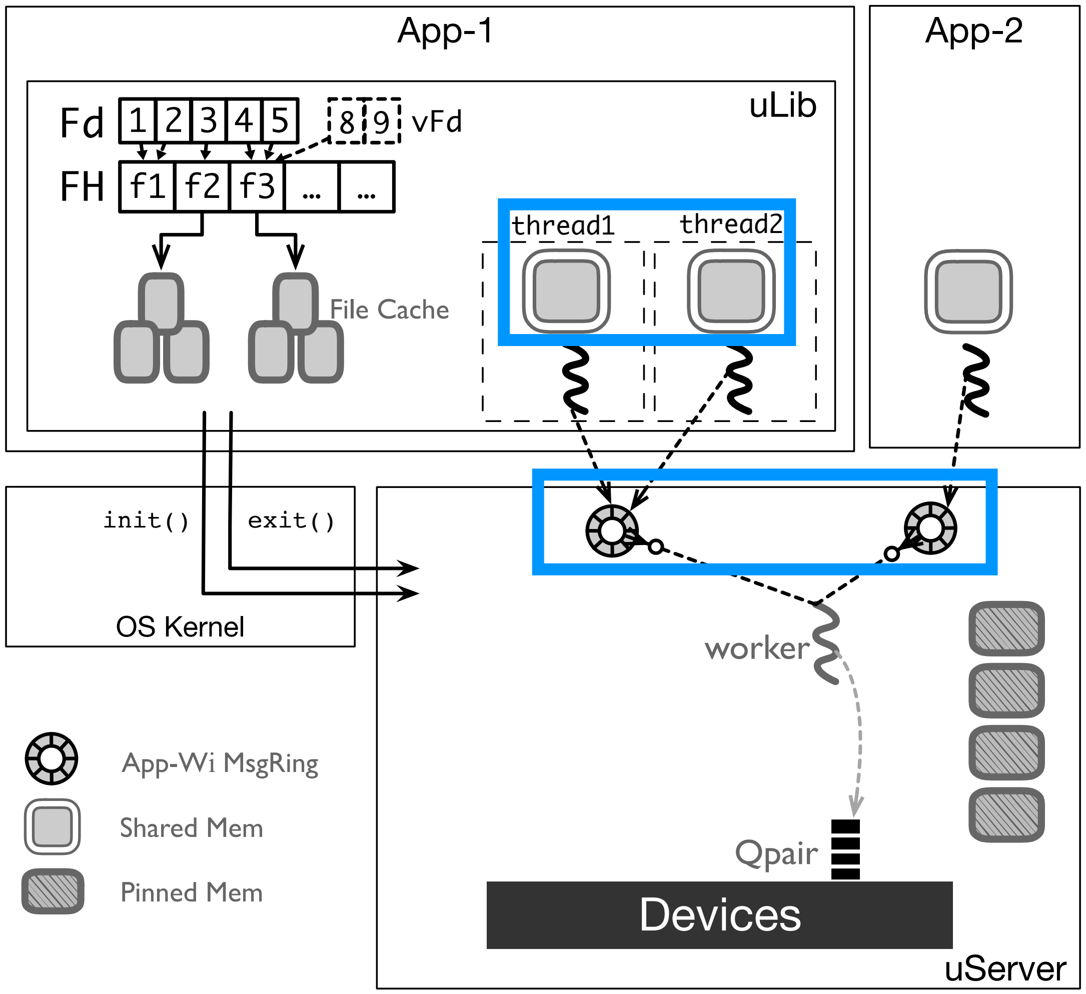
uLib

- POSIX-API
- App-integrated file cache (lease-based)
- Open-lease management (vFd)

The OS kernel only involves for initial authentication (`fs_init`)

uLib ⬌ uServer

- Control: shared-mem IPC (cache-line-size message)
- Data: customized malloc in uLib
  - uLib shares pages with uServer



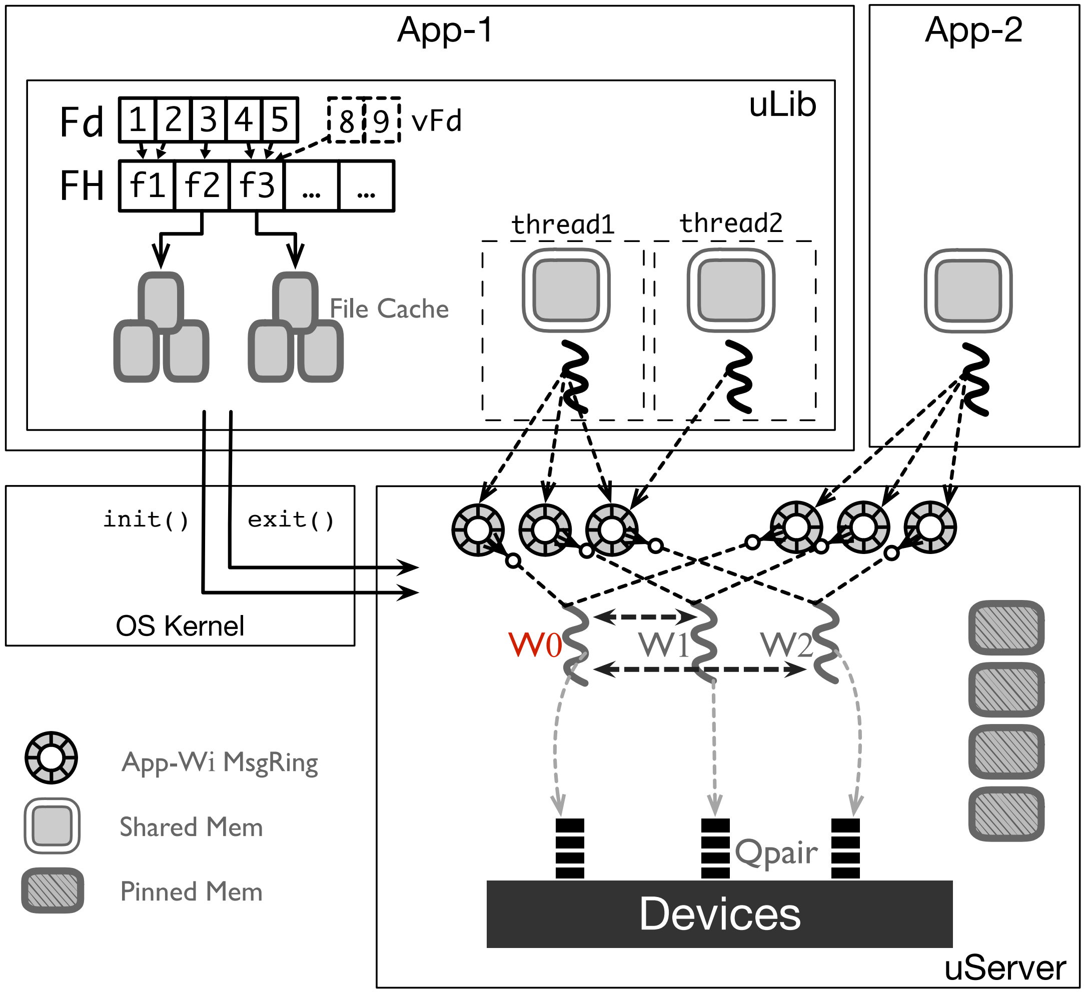
uFS Architecture

uServer: single worker is not enough

- More computing power to saturate device
- In-mem op capacity limited by one core

uServer – multiple workers

- Scalable by design: avoid sharing



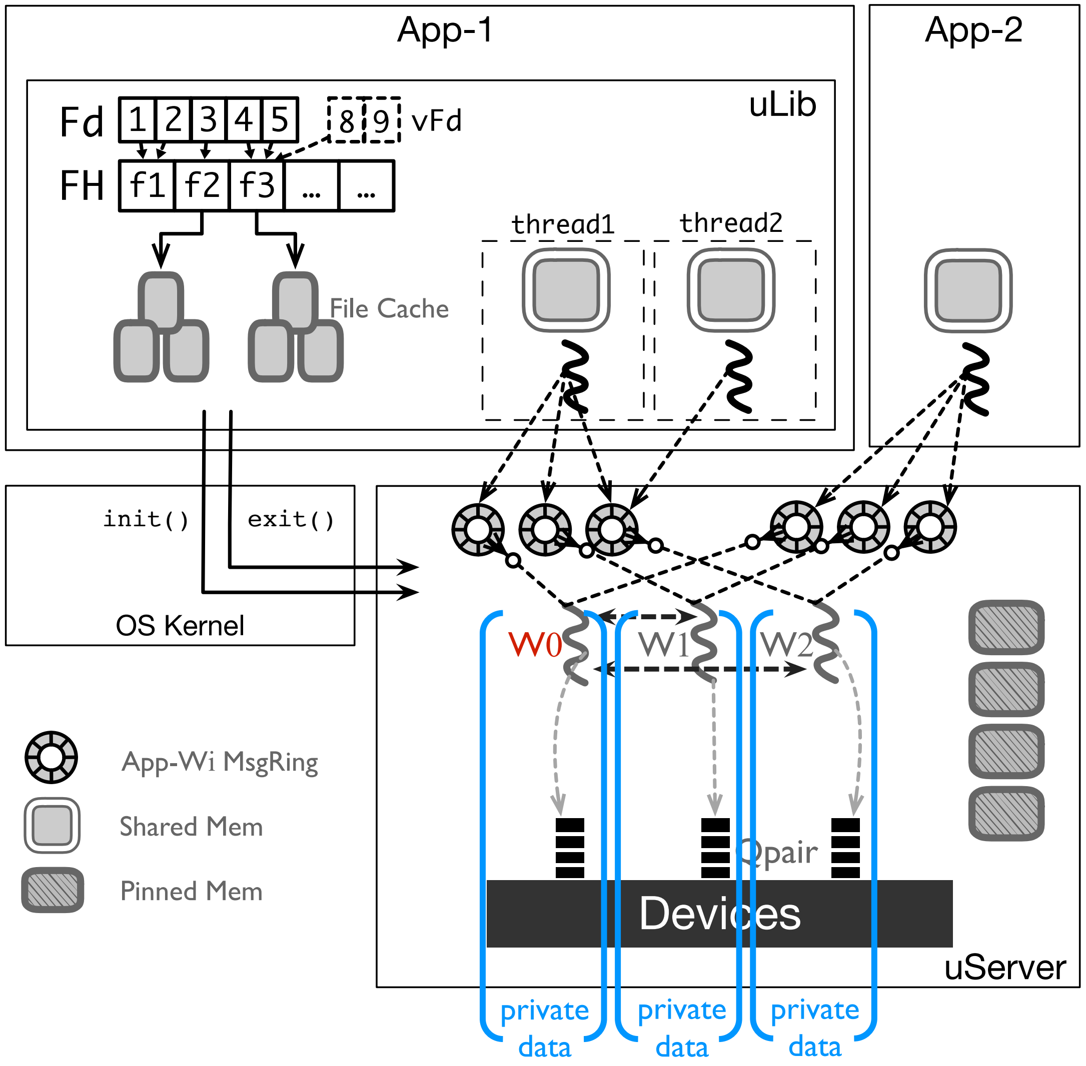
uFS Architecture

uServer: single worker is not enough

- More computing power to saturate device
- In-mem op capacity limited by one core

uServer – multiple workers

- Scalable by design: avoid sharing
- Each worker has several private data structures
  - [in-mem] block buffer cache
  - [in-mem] data bitmaps
  - HW qpair to submit device requests



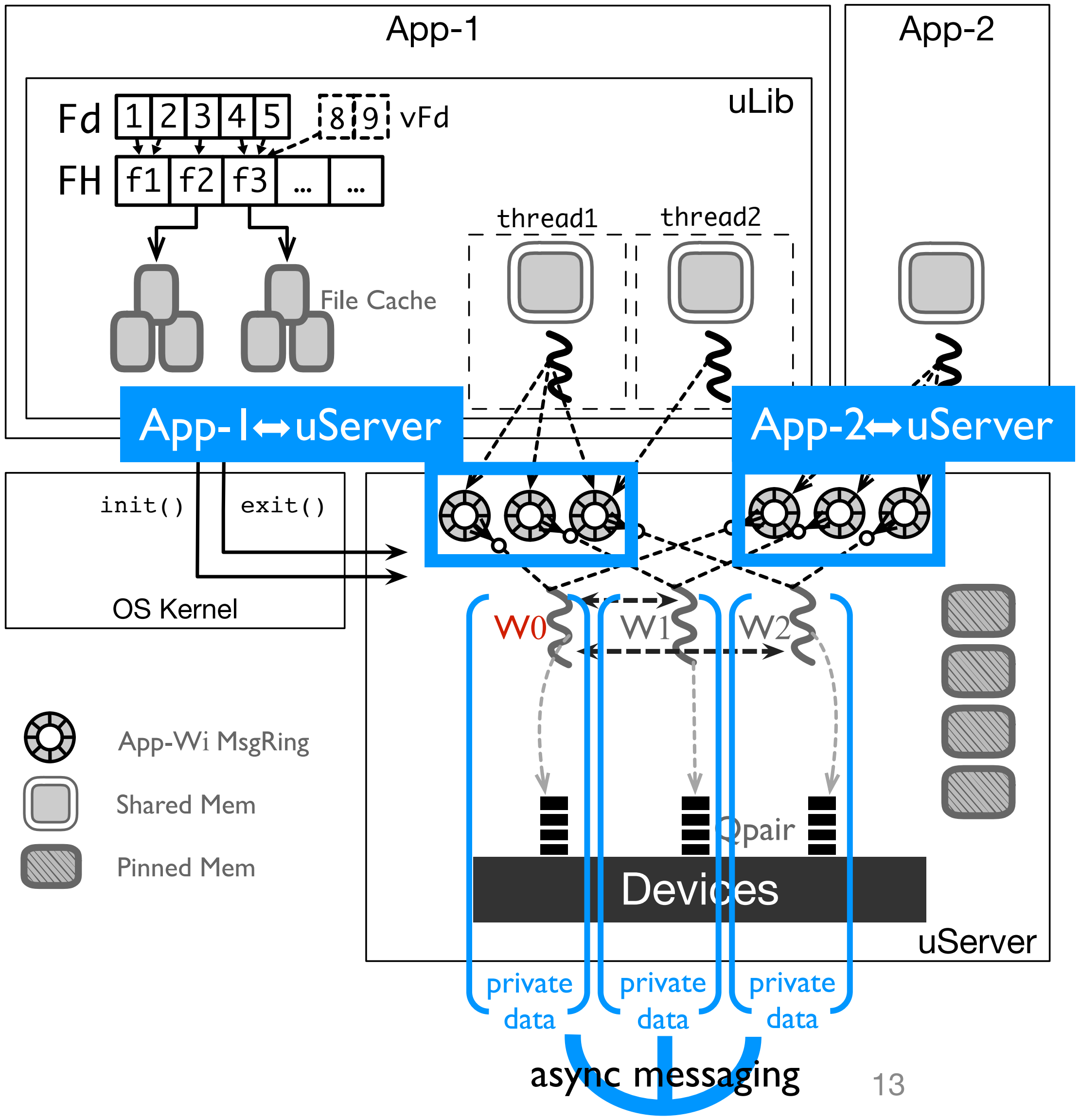
uFS Architecture

uServer: single worker is not enough

- More computing power to saturate device
- In-mem op capacity limited by one core

uServer – multiple workers

- Scalable by design: avoid sharing
- Each worker has several private data structures
  - [in-mem] block buffer cache
  - [in-mem] data bitmaps
  - HW qpair to submit device requests
- Each App-W_{i} has separate message ring
  - Threads in one app will share the ring



Design Overview

Design Overview

Data parallelism for scalability

- Shared-nothing architecture
- Divide filesystem states and data into threads

Design Overview

Data parallelism for scalability

Runtime Inode Ownership

- Shared-nothing architecture
- Divide filesystem states and data into threads

Design Overview

Data parallelism for scalability

Runtime Inode Ownership

- Shared-nothing architecture
- Divide filesystem states and data into threads

The dynamic nature of filesystem workloads

- Data partitioning must be dynamic
- Decides number of cores uFS needs

Design Overview

Data parallelism for scalability

- Shared-nothing architecture
- Divide filesystem states and data into threads

Runtime Inode Ownership

The dynamic nature of filesystem workloads

- Data partitioning must be dynamic
- Decides number of cores uFS needs

Dynamic Load Management

- Load balancing
- Core allocation

Design Overview

Data parallelism for scalability

- Shared-nothing architecture
- Divide filesystem states and data into threads

Runtime Inode Ownership

The dynamic nature of filesystem workloads

- Data partitioning must be dynamic
- Decides number of cores uFS needs

Dynamic Load Management

- Load balancing
- Core allocation

Designs for essential filesystem features

- Performance and scalability in a holistic solution
  - Dentry cache, permission checking, etc.
  - Scalable journaling for crash consistency

Design Overview

Data parallelism for scalability

- Shared-nothing architecture
- Divide filesystem states and data into threads

Runtime Inode Ownership

The dynamic nature of filesystem workloads

- Data partitioning must be dynamic
- Decides number of cores uFS needs

Dynamic Load Management

- Load balancing
- Core allocation

Designs for essential filesystem features

- Performance and scalability in a holistic solution
  - Dentry cache, permission checking, etc.
  - Scalable journaling for crash consistency

Non-blocking Shared Structures

Runtime Inode Ownership

Each group of inodes is exclusively accessed by one worker

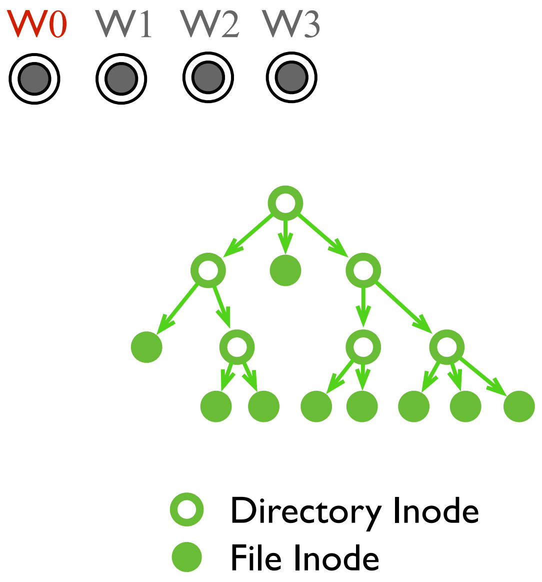
- No need for synchronization

Decouple the namespace and the ownership

- Inodes in one directory can be owned by two workers

Asymmetric Workers

- A primary worker (W0)
  - Owns all the directory inodes: handle all the directory ops
  - Default owner of all the file inodes
  - Coordinates the inode reassignment protocol through message passing
- Secondary workers: file ops



Runtime Inode Ownership

Each group of inodes is exclusively accessed by one worker

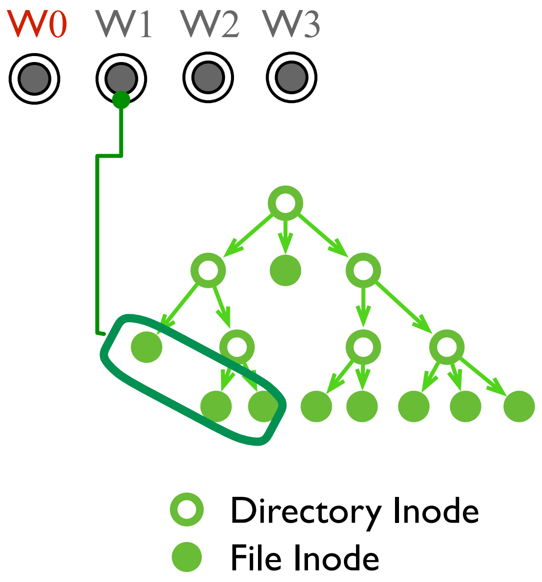
- No need for synchronization

Decouple the namespace and the ownership

- Inodes in one directory can be owned by two workers

Asymmetric Workers

- A primary worker (W0)
  - Owns all the directory inodes: handle all the directory ops
  - Default owner of all the file inodes
  - Coordinates the inode reassignment protocol through message passing
- Secondary workers: file ops



Runtime Inode Ownership

Each group of inodes is exclusively accessed by one worker

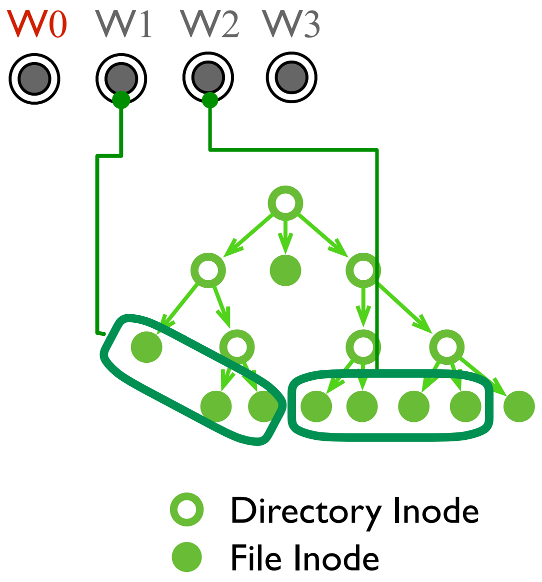
- No need for synchronization

Decouple the namespace and the ownership

- Inodes in one directory can be owned by two workers

Asymmetric Workers

- A primary worker (W0)
  - Owns all the directory inodes: handle all the directory ops
  - Default owner of all the file inodes
  - Coordinates the inode reassignment protocol through message passing
- Secondary workers: file ops



Runtime Inode Ownership

Each group of inodes is exclusively accessed by one worker

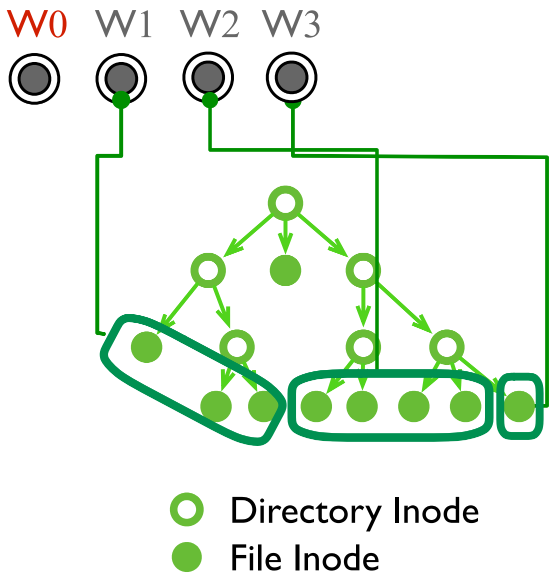
- No need for synchronization

Decouple the namespace and the ownership

- Inodes in one directory can be owned by two workers

Asymmetric Workers

- A primary worker (W0)
  - Owns all the directory inodes: handle all the directory ops
  - Default owner of all the file inodes
  - Coordinates the inode reassignment protocol through message passing
- Secondary workers: file ops



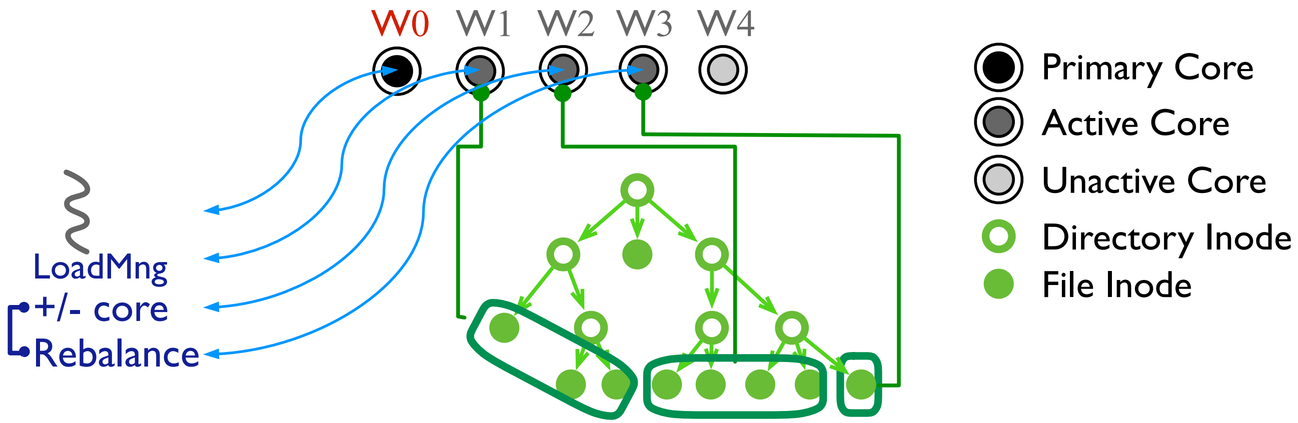
Dynamic Load Management

Separate load managing thread (LoadMng)

- Periodically gathers load stats from each worker (a monitoring window)
- Decides per-worker [load goal] ➡ Informs each worker the desired goal to achieve
- Decides number of cores ➡ (De)activates cores

Worker invokes inode reassignment

- Tracks per-inode stats
- Given [load goal], decides which groups of inodes to be re-assigned



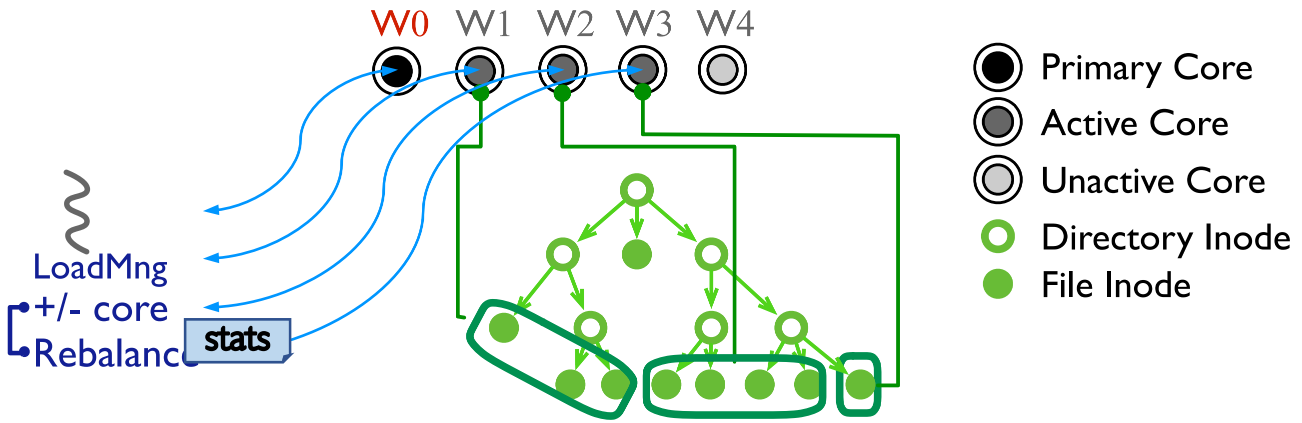
Dynamic Load Management

Separate load managing thread (LoadMng)

- Periodically gathers load stats from each worker (a monitoring window)
- Decides per-worker [load goal] ➡ Informs each worker the desired goal to achieve
- Decides number of cores ➡ (De)activates cores

Worker invokes inode reassignment

- Tracks per-inode stats
- Given [load goal], decides which groups of inodes to be re-assigned



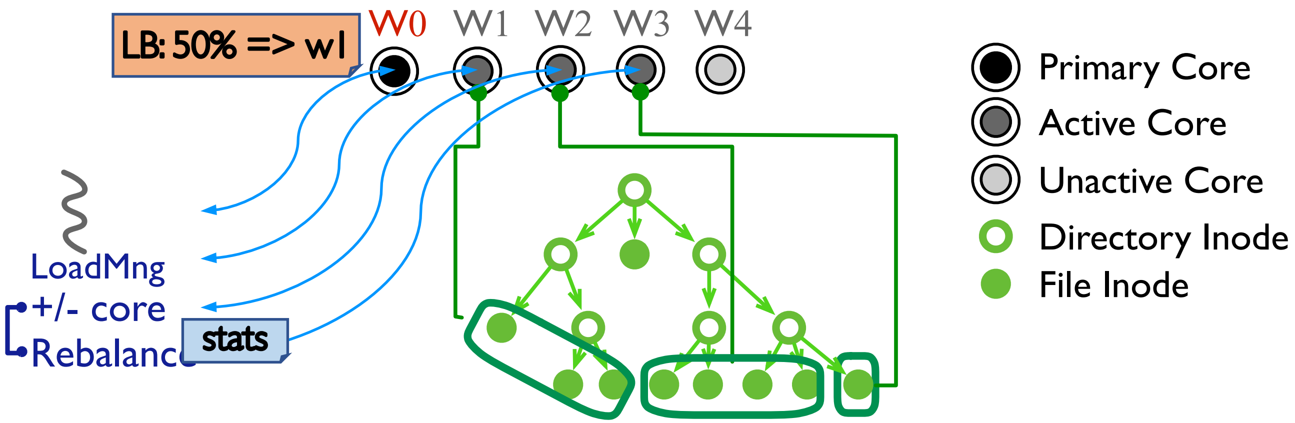
Dynamic Load Management

Separate load managing thread (LoadMng)

- Periodically gathers load stats from each worker (a monitoring window)
- Decides per-worker [load goal] ➡ Informs each worker the desired goal to achieve
- Decides number of cores ➡ (De)activates cores

Worker invokes inode reassignment

- Tracks per-inode stats
- Given [load goal], decides which groups of inodes to be re-assigned



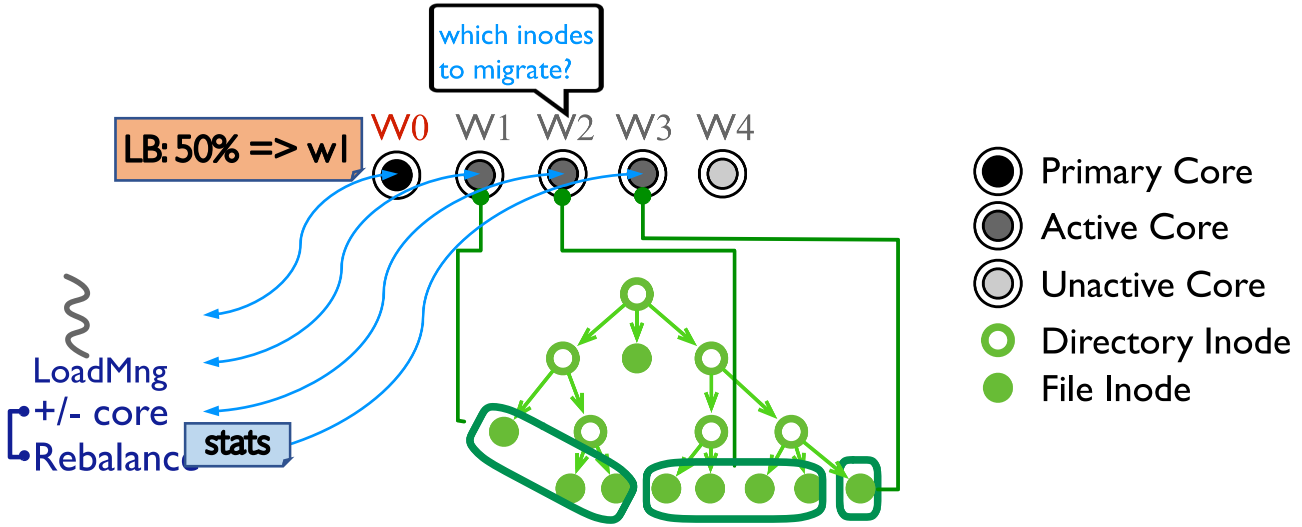
Dynamic Load Management

Separate load managing thread (LoadMng)

- Periodically gathers load stats from each worker (a monitoring window)
- Decides per-worker [load goal] ➡ Informs each worker the desired goal to achieve
- Decides number of cores ➡ (De)activates cores

Worker invokes inode reassignment

- Tracks per-inode stats
- Given [load goal], decides which groups of inodes to be re-assigned



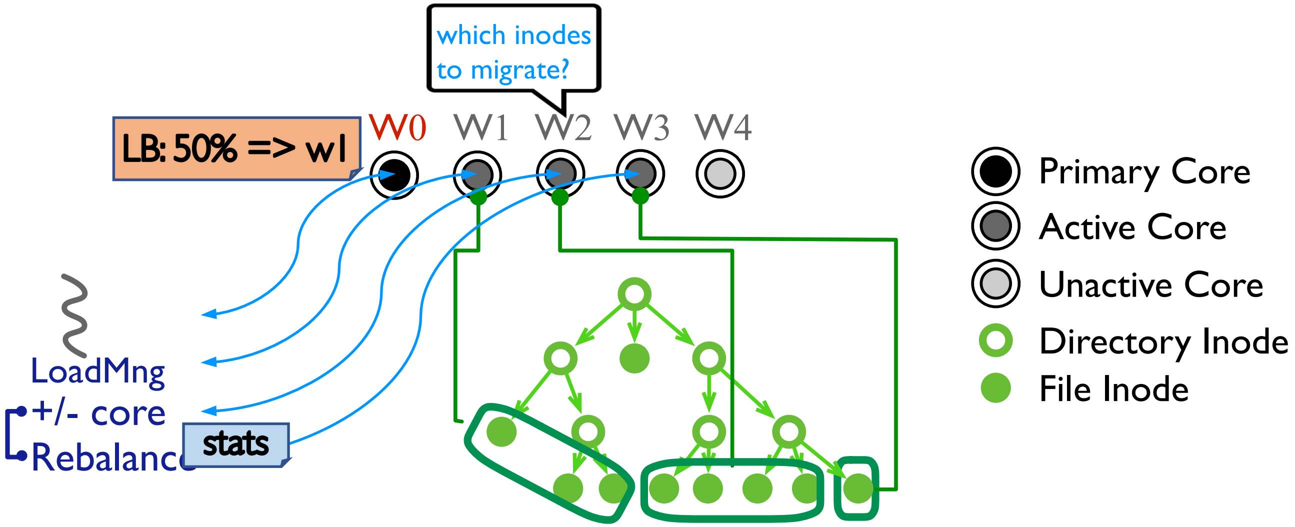
Dynamic Load Management: Algorithms

Load balancing

- Towards minimizing congestion on each core

Core allocation

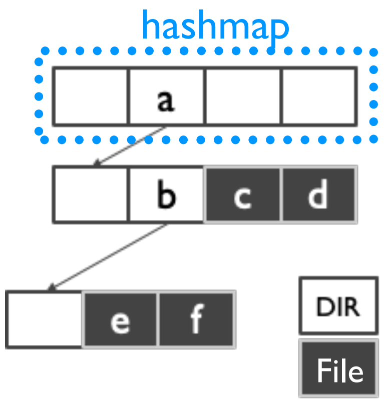
- Meets a per-core CPU utilization goal
- Answer the "what if" questions by algorithmically emulating the load balancing results
  - Load balancing as a black-box
  - What if [add one core | no change | remove one core]



Employ Non-blocking Shared Structures Judiciously

Dentry Cache and Permission Checking

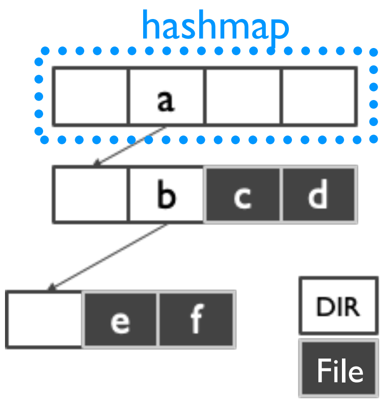
- Recursive HashMap
- Only the primary worker can update and all can read
- Leverage industrial-quality lock-free data structures



Employ Non-blocking Shared Structures Judiciously

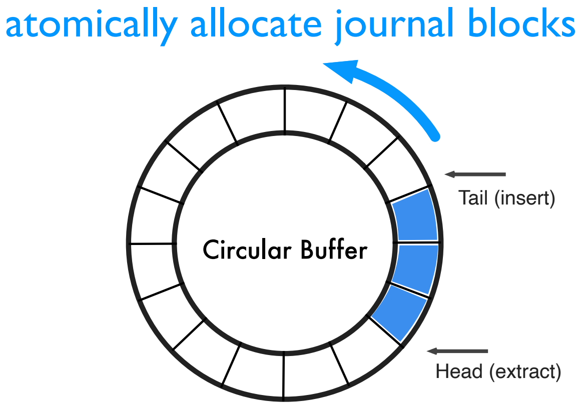
Dentry Cache and Permission Checking

- Recursive HashMap
- Only the primary worker can update and all can read
- Leverage industrial-quality lock-free data structures



Global Logic Journal that allows maximal parallelism

- Each worker can initialize journal transactions independently for owned inodes
- Negligible overhead added
  - Recording logic modification is lightweight
  - Minimal critical section when reserving journal blocks



Evaluation

uFS offers good single-threaded base performance

uFS performs well as a multi-threaded microkernel

uFS dynamically scales to match demand

- Load Balancing Experiments
- Core Allocation Experiments

uFS performs and scales well with real applications

- LevelDB and YCSB workloads

Platform

- Intel Optane 905P SSD; Intel(R) Xeon(R) Gold 5218R CPU
- Linux 5.4, SPDK 18.04

Evaluation

uFS offers good single-threaded base performance

uFS performs well as a multi-threaded microkernel

uFS dynamically scales to match demand

- Load Balancing Experiments
- Core Allocation Experiments

uFS performs and scales well with real applications

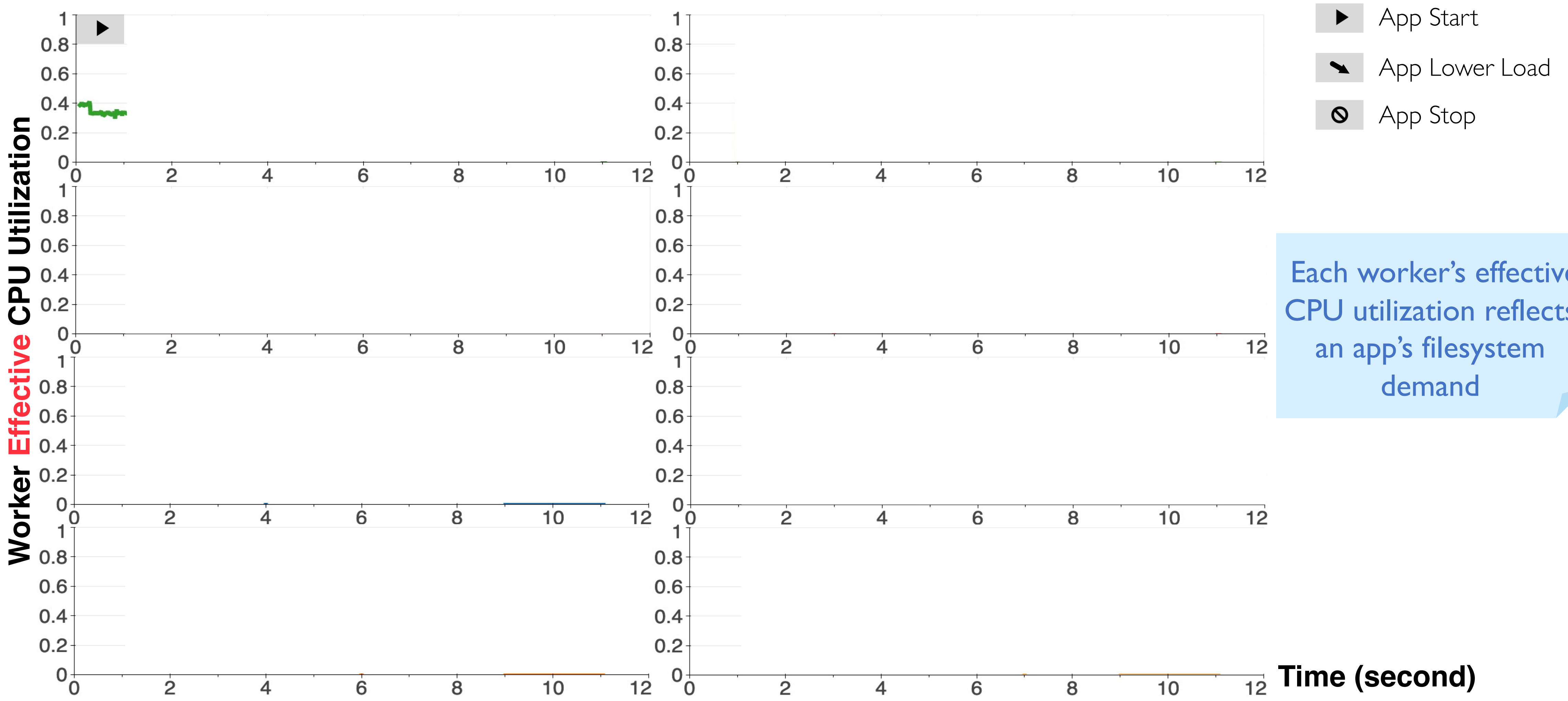
- LevelDB and YCSB workloads

More detailed results in our paper

Platform

- Intel Optane 905P SSD; Intel(R) Xeon(R) Gold 5218R CPU
- Linux 5.4, SPDK 18.04

Determining Number of uServer Cores

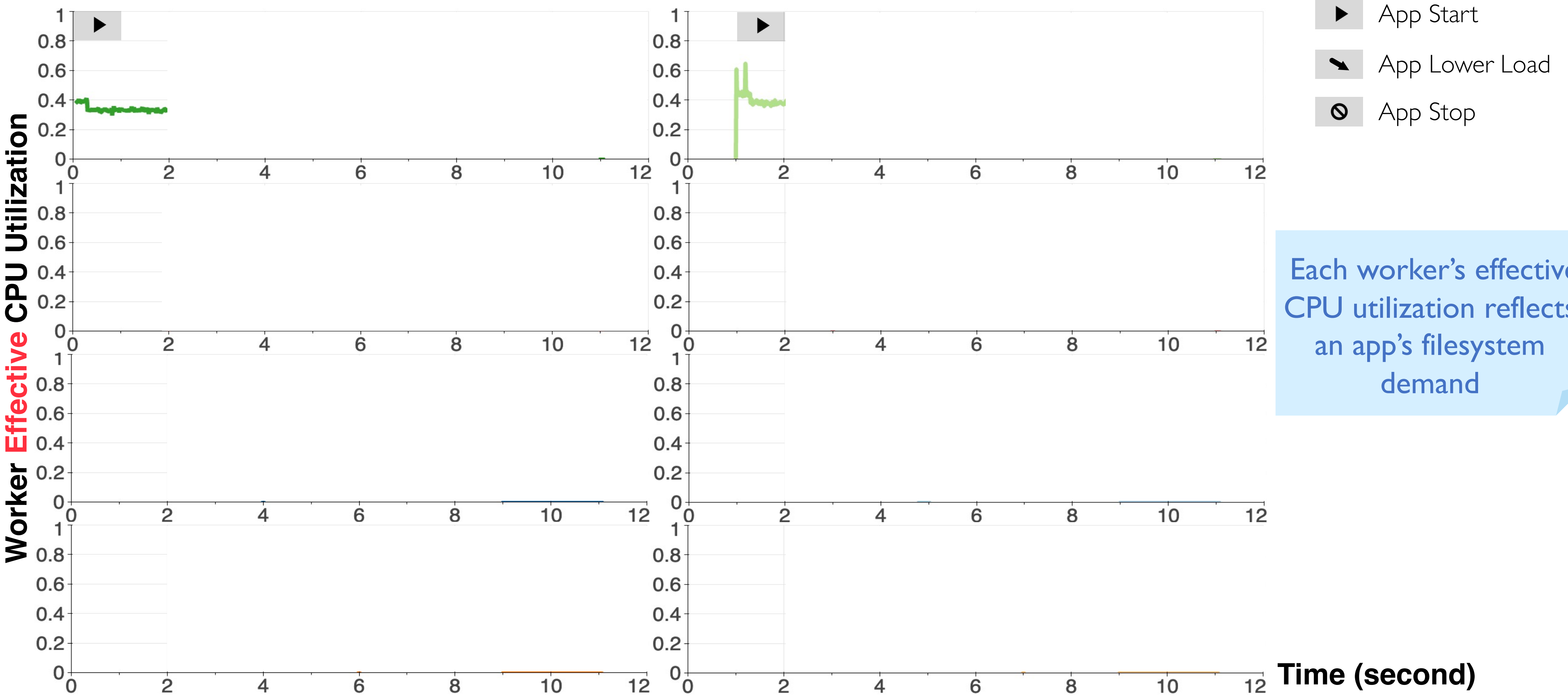


Each worker's effective CPU utilization reflects an app's filesystem demand

Problem: Apps' filesystem demands change over time, one worker per app wastes CPU

Complication: To OS scheduler, 8 cores appear to be 100% utilized due to polling

Determining Number of uServer Cores

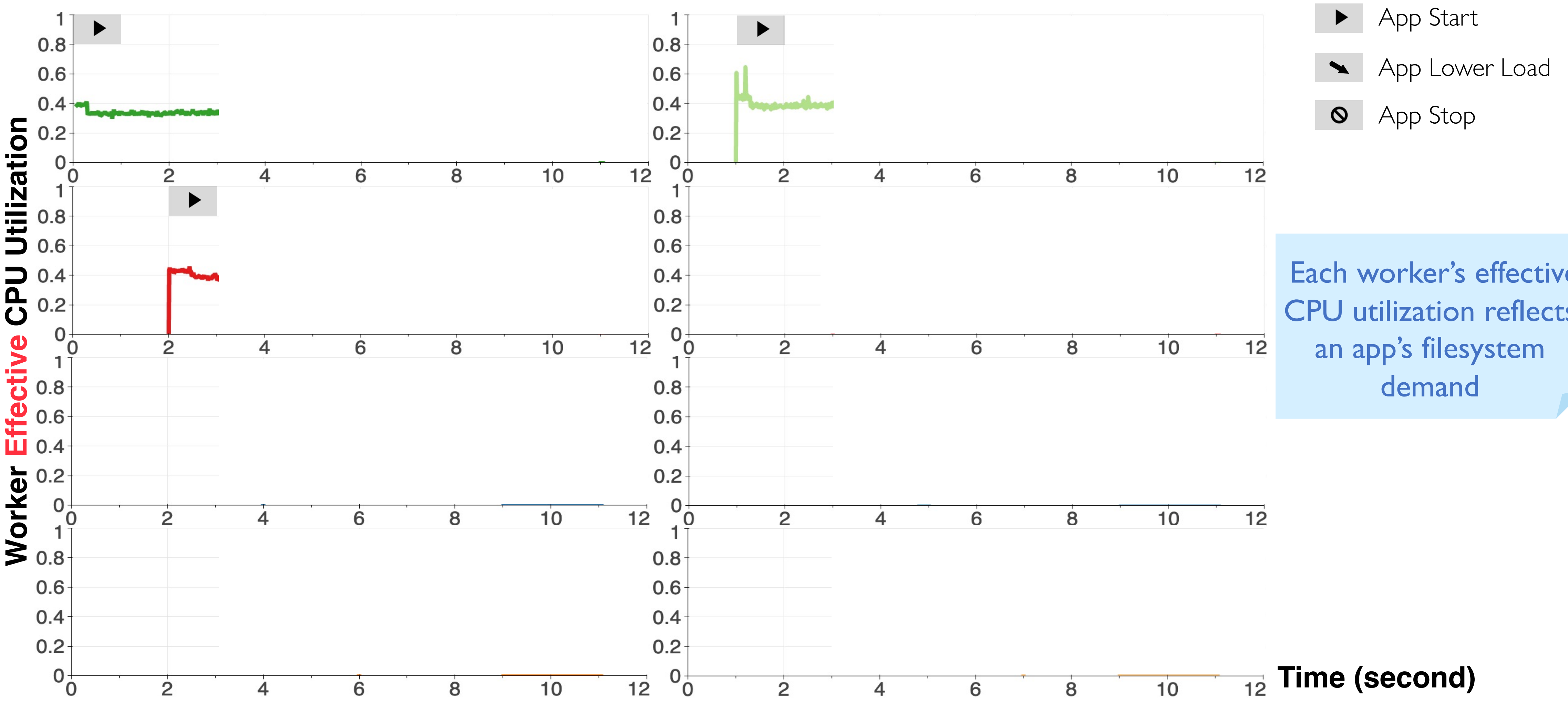


Each worker's effective CPU utilization reflects an app's filesystem demand

Problem: Apps' filesystem demands change over time, one worker per app wastes CPU

Complication: To OS scheduler, 8 cores appear to be 100% utilized due to polling

Determining Number of uServer Cores

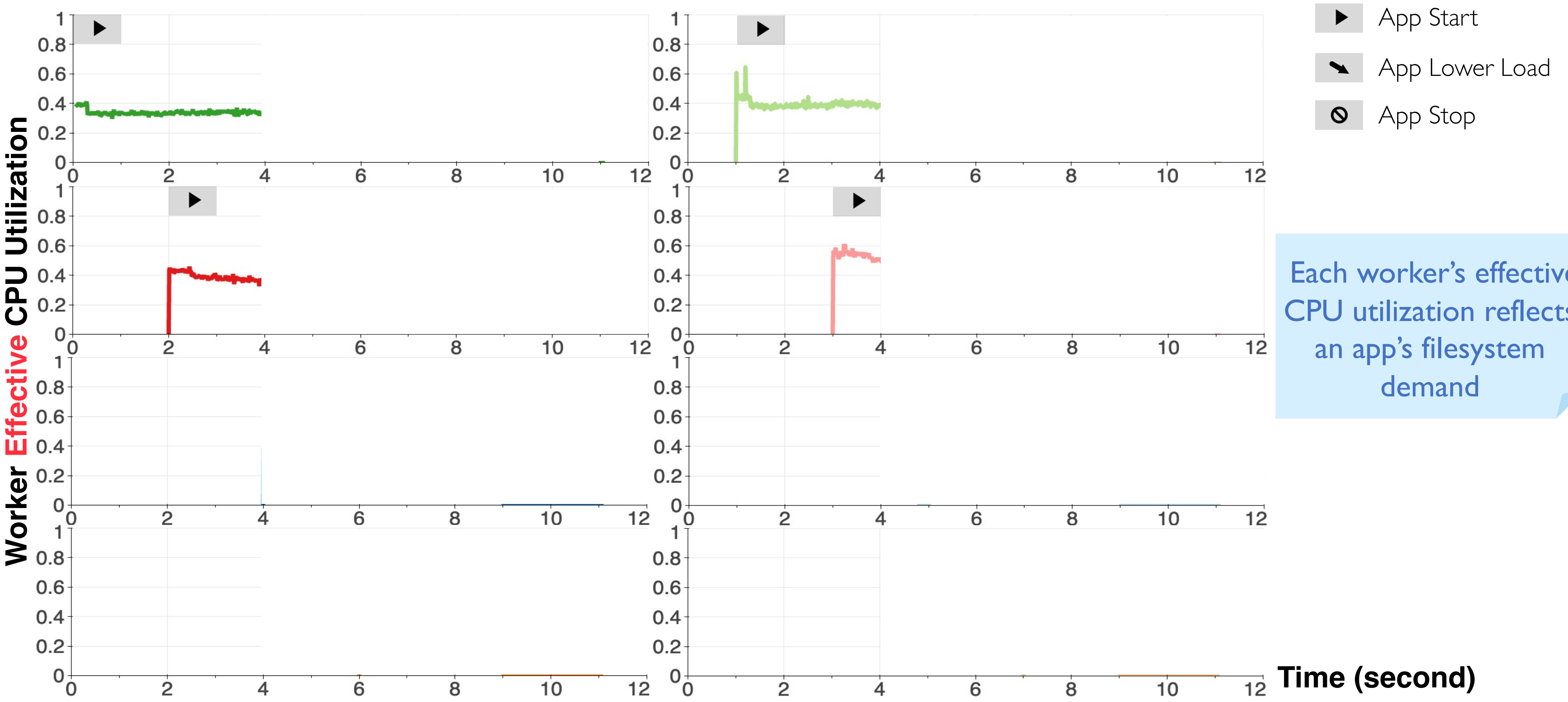


Each worker's effective CPU utilization reflects an app's filesystem demand

Problem: Apps' filesystem demands change over time, one worker per app wastes CPU

Complication: To OS scheduler, 8 cores appear to be 100% utilized due to polling

Determining Number of uServer Cores

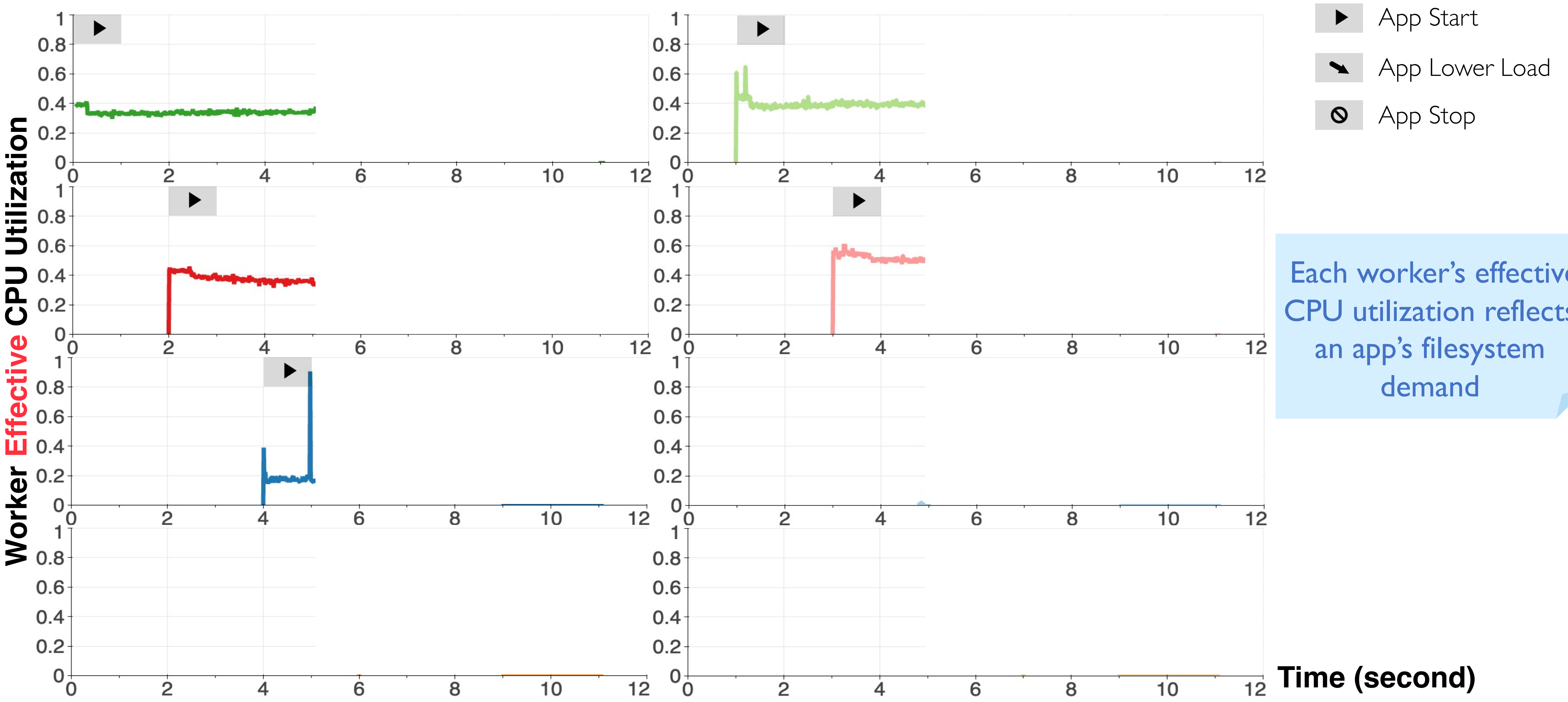


Each worker's effective CPU utilization reflects an app's filesystem demand

Problem: Apps' filesystem demands change over time, one worker per app wastes CPU

Complication: To OS scheduler, 8 cores appear to be 100% utilized due to polling

Determining Number of uServer Cores

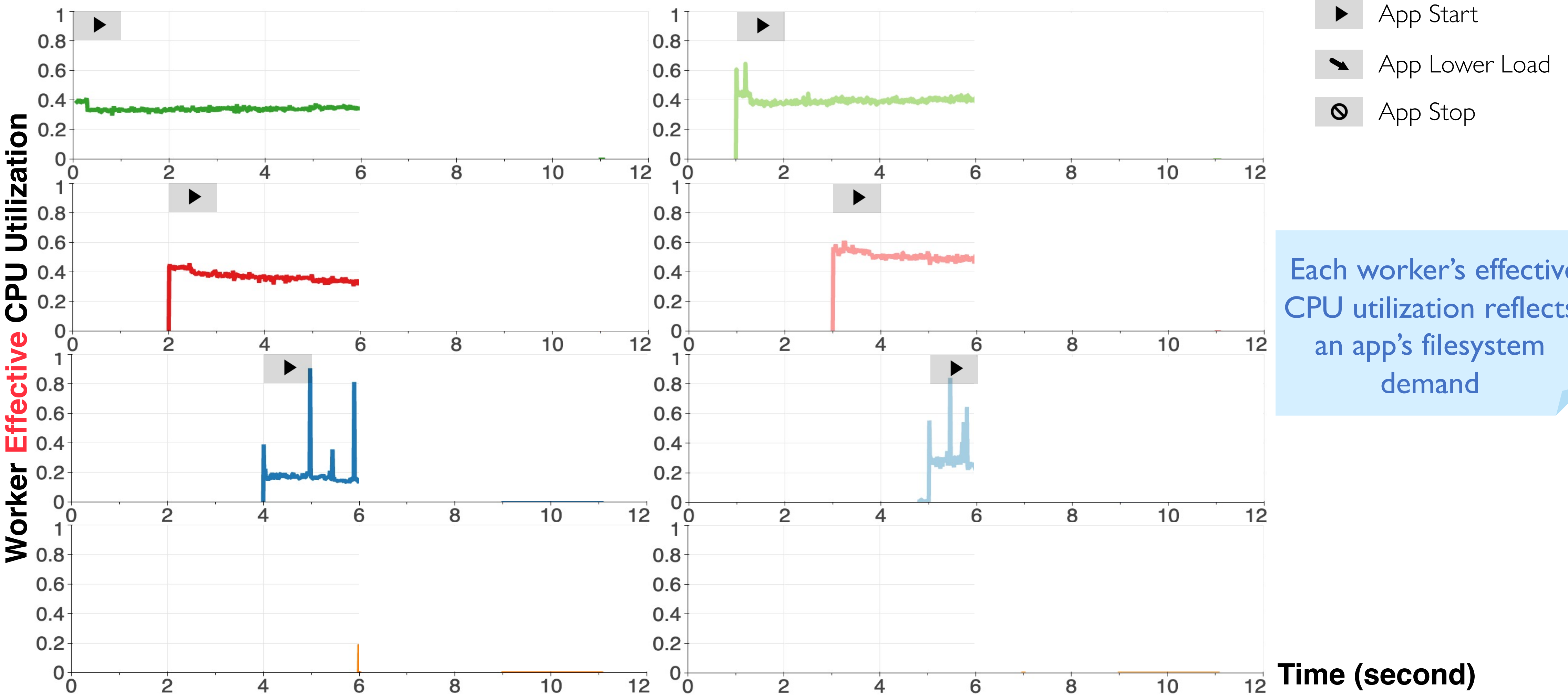


Each worker's effective CPU utilization reflects an app's filesystem demand

Problem: Apps' filesystem demands change over time, one worker per app wastes CPU

Complication: To OS scheduler, 8 cores appear to be 100% utilized due to polling

Determining Number of uServer Cores

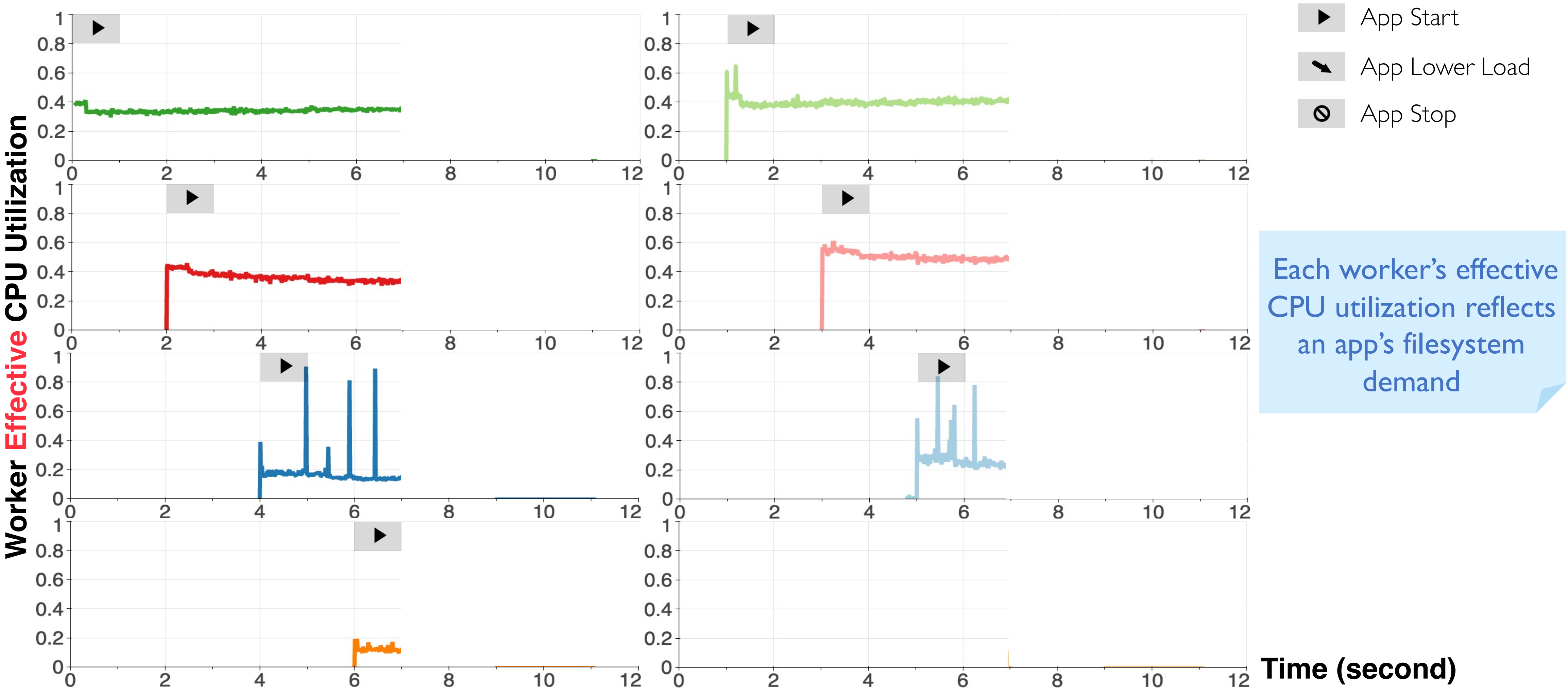


Each worker's effective CPU utilization reflects an app's filesystem demand

Problem: Apps' filesystem demands change over time, one worker per app wastes CPU

Complication: To OS scheduler, 8 cores appear to be 100% utilized due to polling

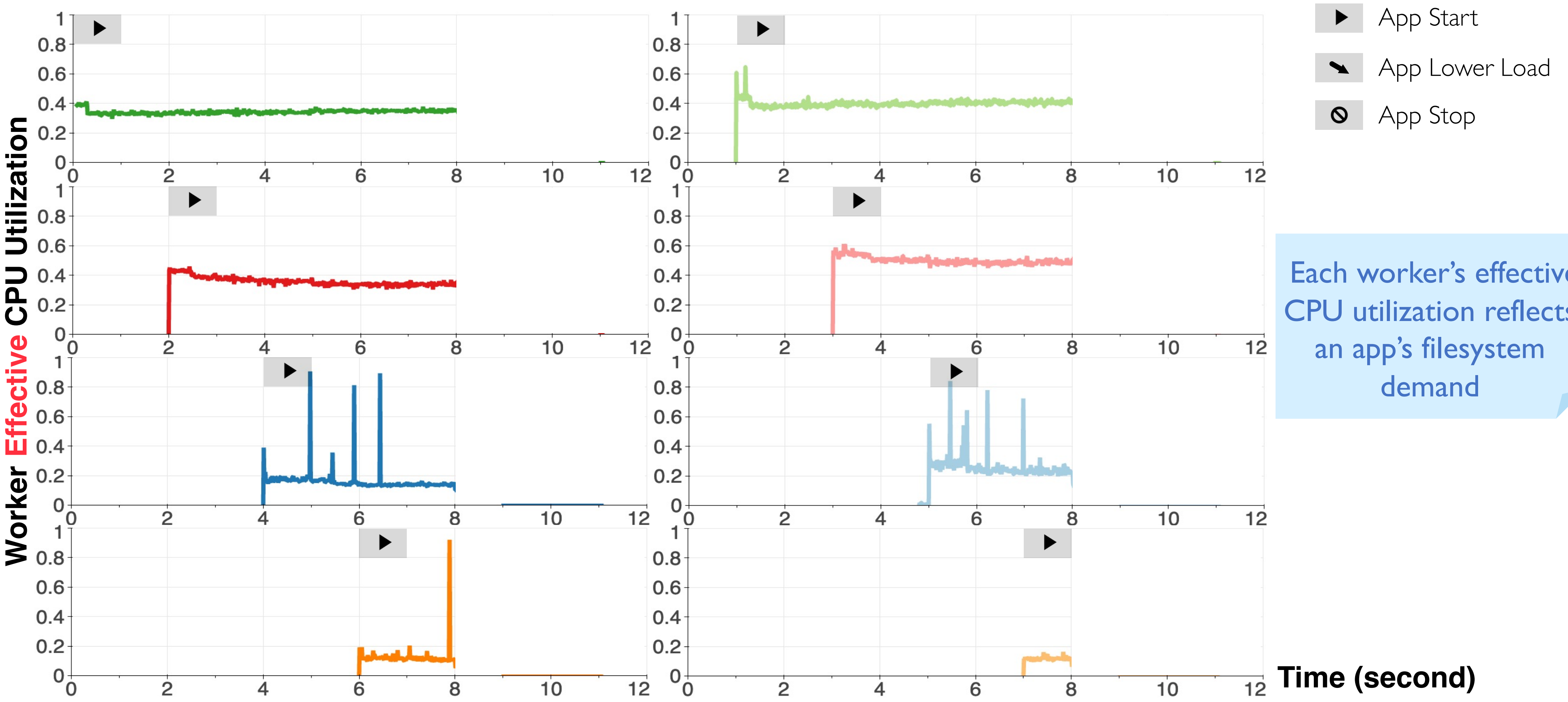
Determining Number of uServer Cores



Problem: Apps' filesystem demands change over time, one worker per app wastes CPU

Complication: To OS scheduler, 8 cores appear to be 100% utilized due to polling

Determining Number of uServer Cores

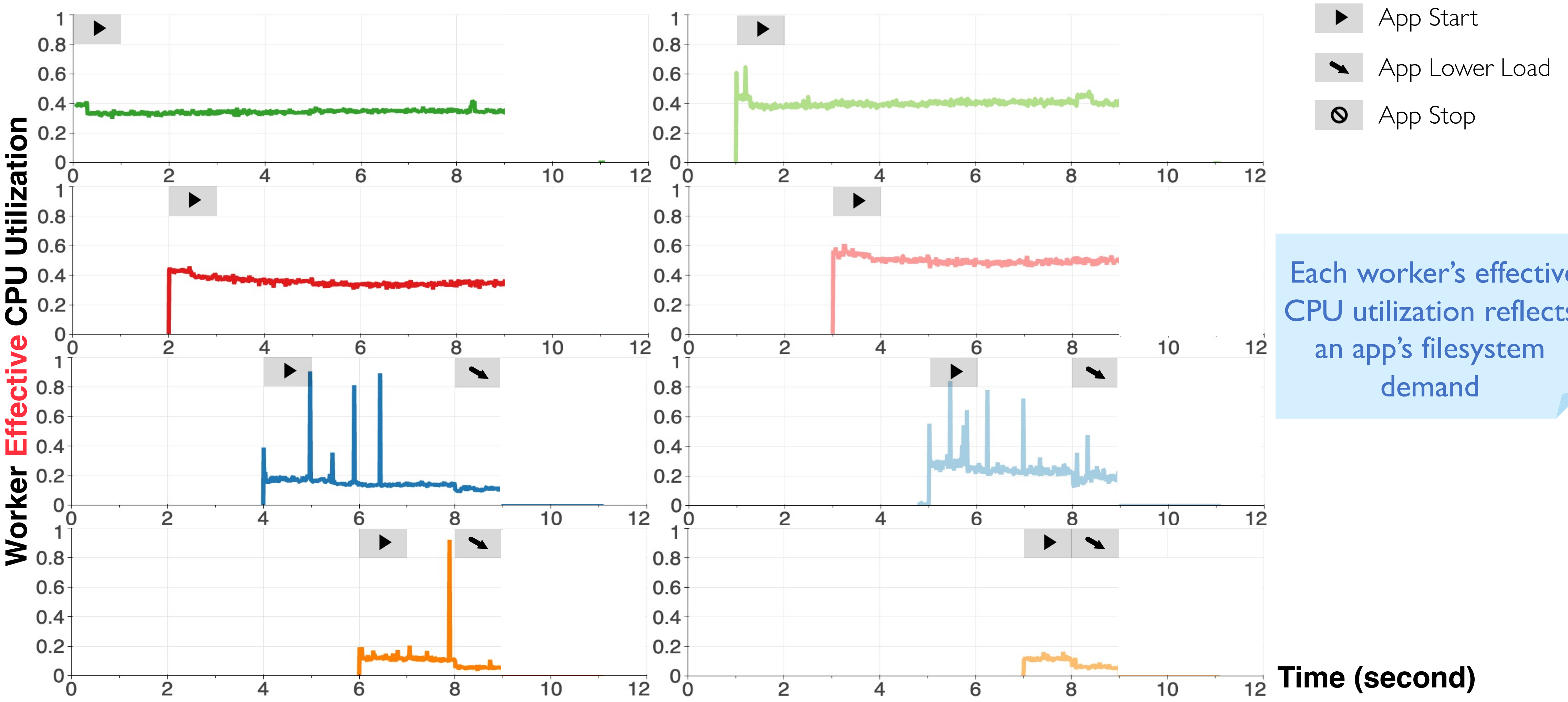


Each worker's effective CPU utilization reflects an app's filesystem demand

Problem: Apps' filesystem demands change over time, one worker per app wastes CPU

Complication: To OS scheduler, 8 cores appear to be 100% utilized due to polling

Determining Number of uServer Cores

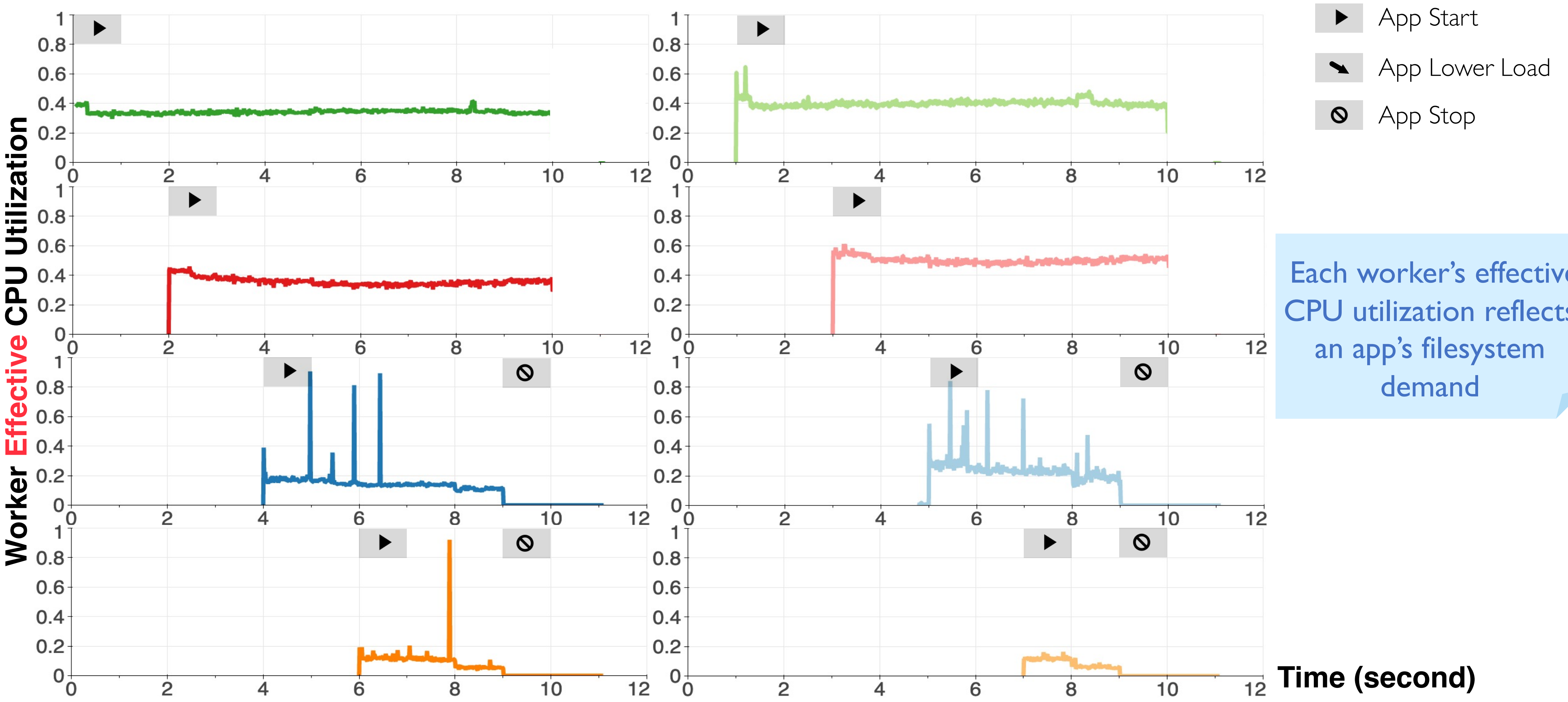


Each worker's effective CPU utilization reflects an app's filesystem demand

Problem: Apps' filesystem demands change over time, one worker per app wastes CPU

Complication: To OS scheduler, 8 cores appear to be 100% utilized due to polling

Determining Number of uServer Cores

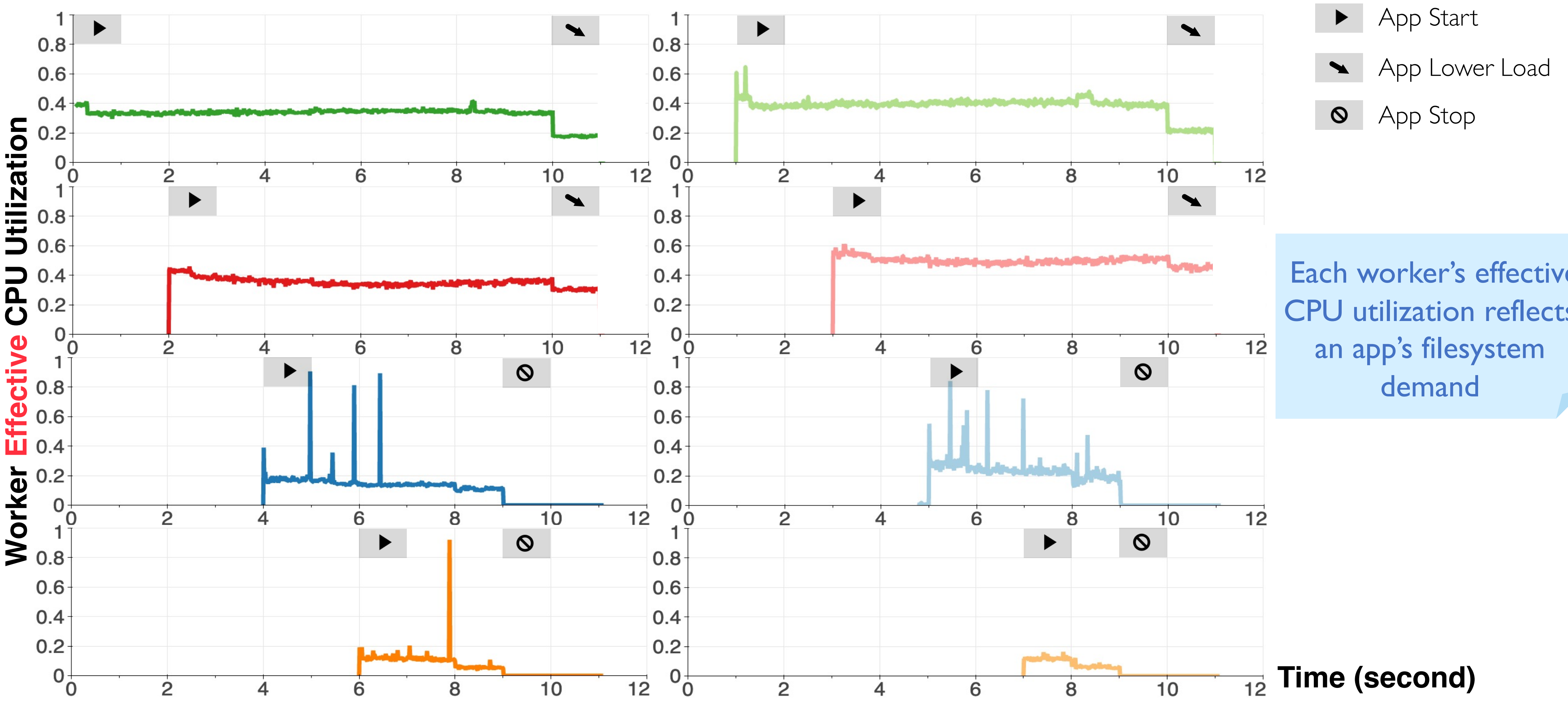


Each worker's effective CPU utilization reflects an app's filesystem demand

Problem: Apps' filesystem demands change over time, one worker per app wastes CPU

Complication: To OS scheduler, 8 cores appear to be 100% utilized due to polling

Determining Number of uServer Cores

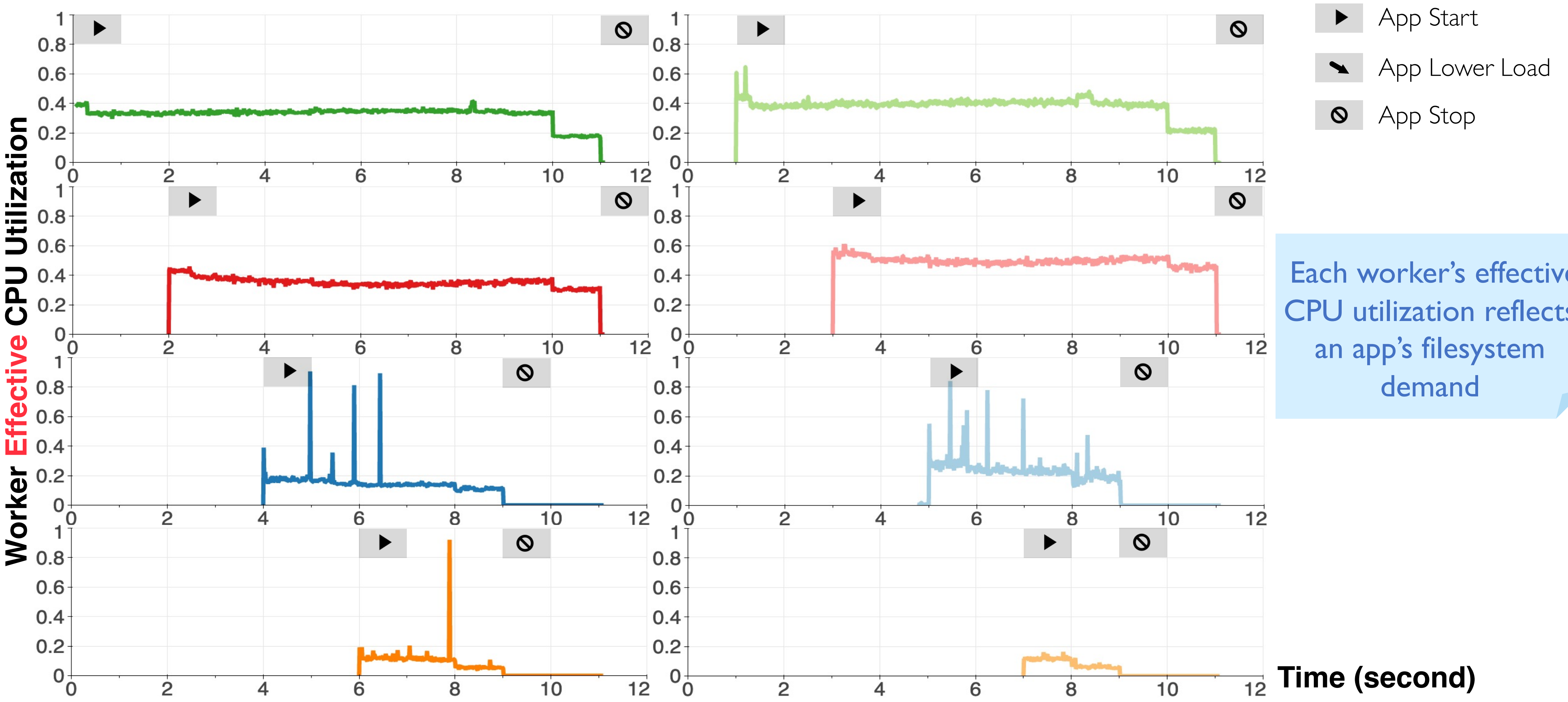


Each worker's effective CPU utilization reflects an app's filesystem demand

Problem: Apps' filesystem demands change over time, one worker per app wastes CPU

Complication: To OS scheduler, 8 cores appear to be 100% utilized due to polling

Determining Number of uServer Cores

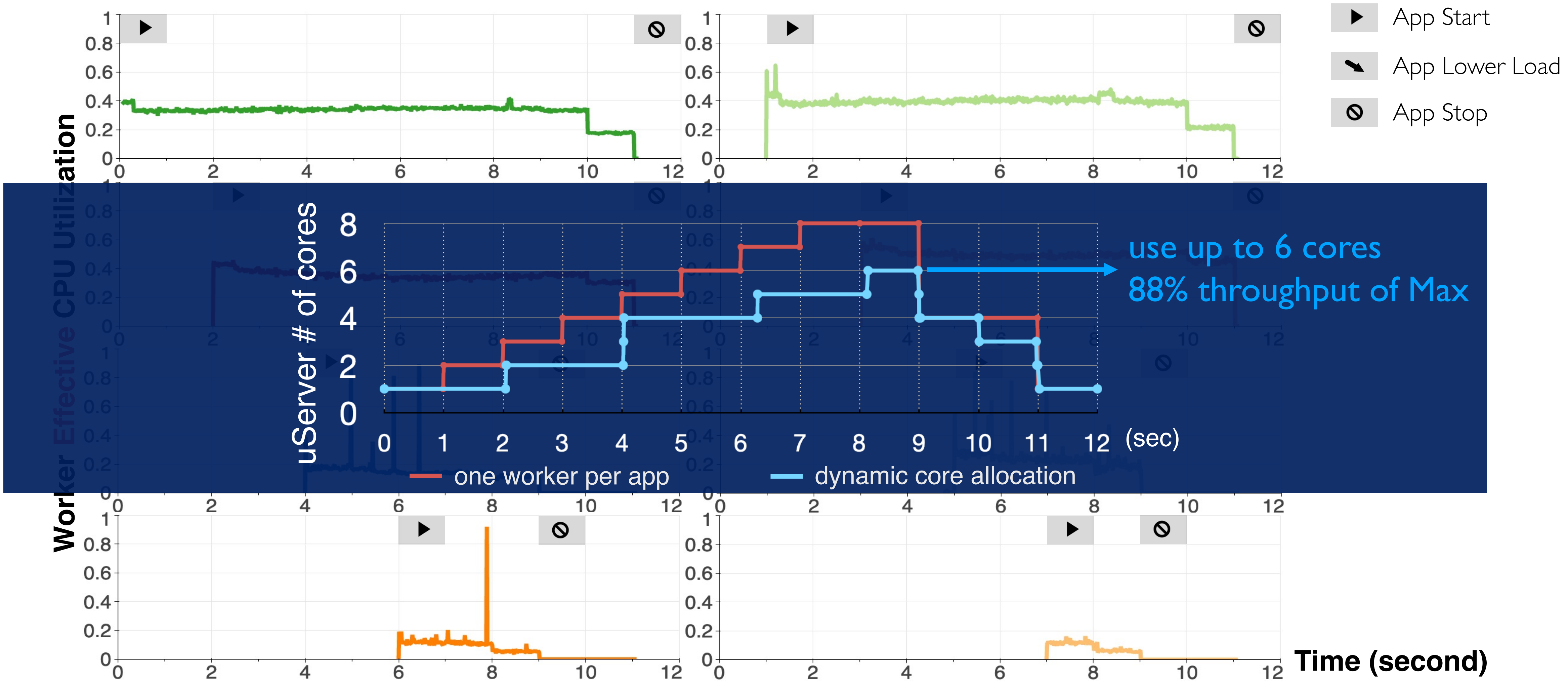


Each worker's effective CPU utilization reflects an app's filesystem demand

Problem: Apps' filesystem demands change over time, one worker per app wastes CPU

Complication: To OS scheduler, 8 cores appear to be 100% utilized due to polling

Determining Number of uServer Cores



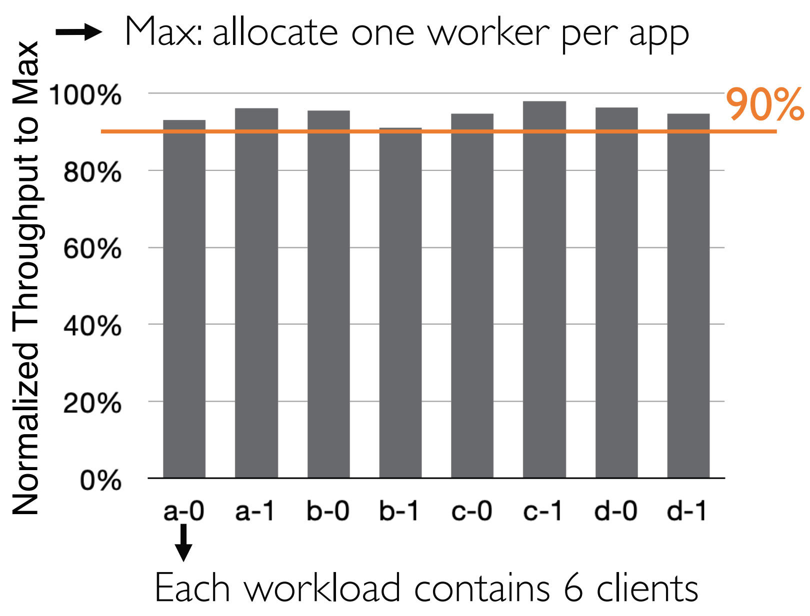
Problem: Apps' filesystem demands change over time, one worker per app wastes CPU

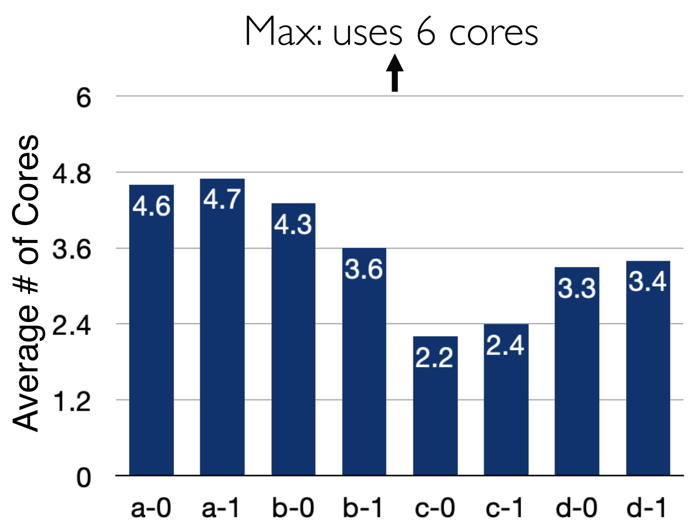
Complication: To OS scheduler, 8 cores appear to be 100% utilized due to polling

Core Allocation Experiments

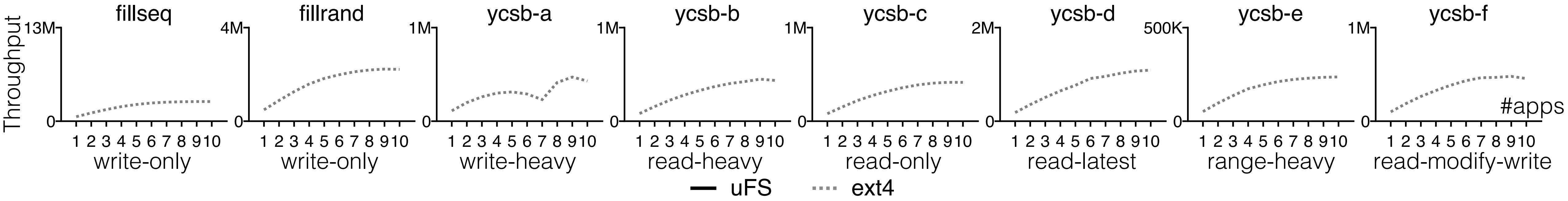
8 workloads: each changes one factor by N steps along the time

- Factor example: think-time, data screw degree, request size
- uFS delivers between 91% to 98% throughput of Max
- uFS controls number of cores as needed





LevelDB: uFS Performs and Scales Well with Real Apps

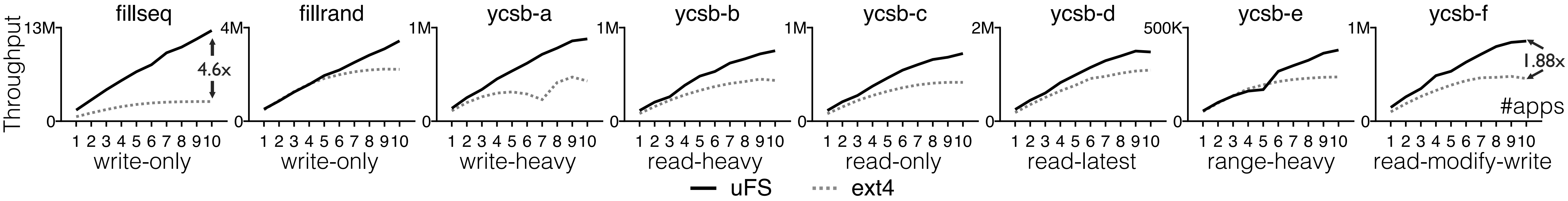


uFS can scale much better than ext4

uFS will allocate different number of cores for various workloads

Giving more cores (>10) to ext4 does not help much for performance

LevelDB: uFS Performs and Scales Well with Real Apps

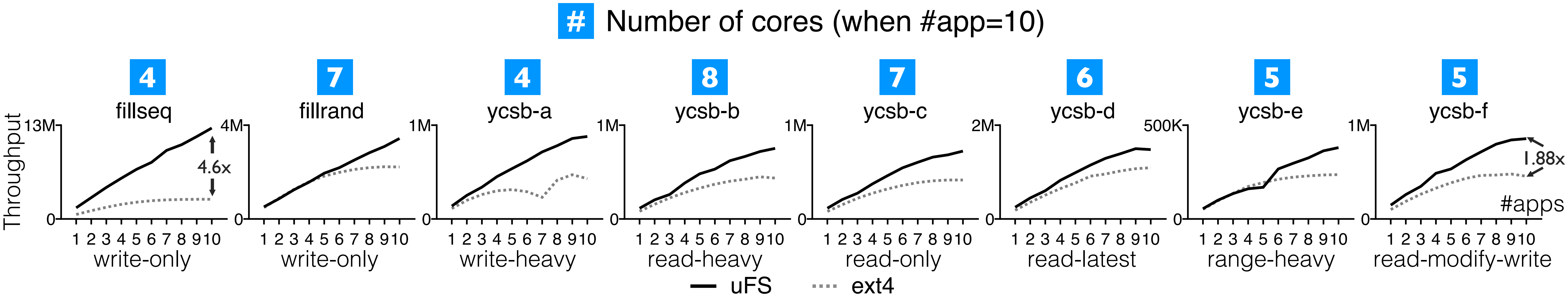


uFS can scale much better than ext4

uFS will allocate different number of cores for various workloads

Giving more cores (>10) to ext4 does not help much for performance

LevelDB: uFS Performs and Scales Well with Real Apps



uFS can scale much better than ext4

uFS will allocate different number of cores for various workloads

Giving more cores (>10) to ext4 does not help much for performance

Conclusion

uFS: a filesystem semi-microkernel

- Designs for modern storage *device performance delivery* and *scalability*
  - Outperforms ext4 under LevelDB workloads by 1.22x to 4.6x
- Scales independently from the applications and dynamically matches demand

Filesystem Semi-Microkernel Approach

- Performs and scales well under various workloads
- Has all the benefits of user-level development

Available at: https://research.cs.wisc.edu/adsl/Software/uFS/

Conclusion

uFS: a filesystem semi-microkernel

- Designs for modern storage *device performance delivery* and *scalability*
  - Outperforms ext4 under LevelDB workloads by 1.22x to 4.6x
- Scales independently from the applications and dynamically matches demand

Filesystem Semi-Microkernel Approach

- Performs and scales well under various workloads
- Has all the benefits of user-level development

Available at: https://research.cs.wisc.edu/adsl/Software/uFS/

Thank you!