High Performance Data Center Operating Systems

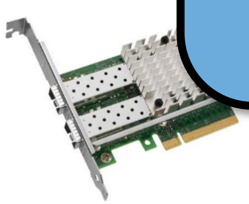
Tom Anderson

Antoine Kaufmann, Youngjin Kwon, Naveen Kr. Sharma, Arvind Krishnamurthy, Simon Peter, Mothy Roscoe, and Emmett Witchel

An OS for the Data Center

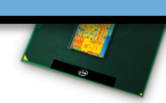
- Server I/O performance matters
  - Key-value stores, web & file servers, databases, mail servers, machine learning
- **Can w** **ware?**
- Examp

Today's I/O devices are fast and getting faster



Intel X520
10G NIC
2 us / 1KB packet

Intel RS3 RAID
1GB flash-backed cache
25 us / 1KB write



Sandy Bridge CPU
6 cores, 2.2 GHz

40G NIC: **500 ns / 1KB packet**
NVDIMM: **500 ns / 1KB write**

Can’t we just use Linux?

Linux I/O Performance

% OF 1KB REQUEST TIME SPENT

Redis

GET **HW 18%** | **Kernel 62%** | **App 20%** **9 us**

us

Kernel

Kernel mediation is too heavyweight

I/O Processing

Copying

Protection

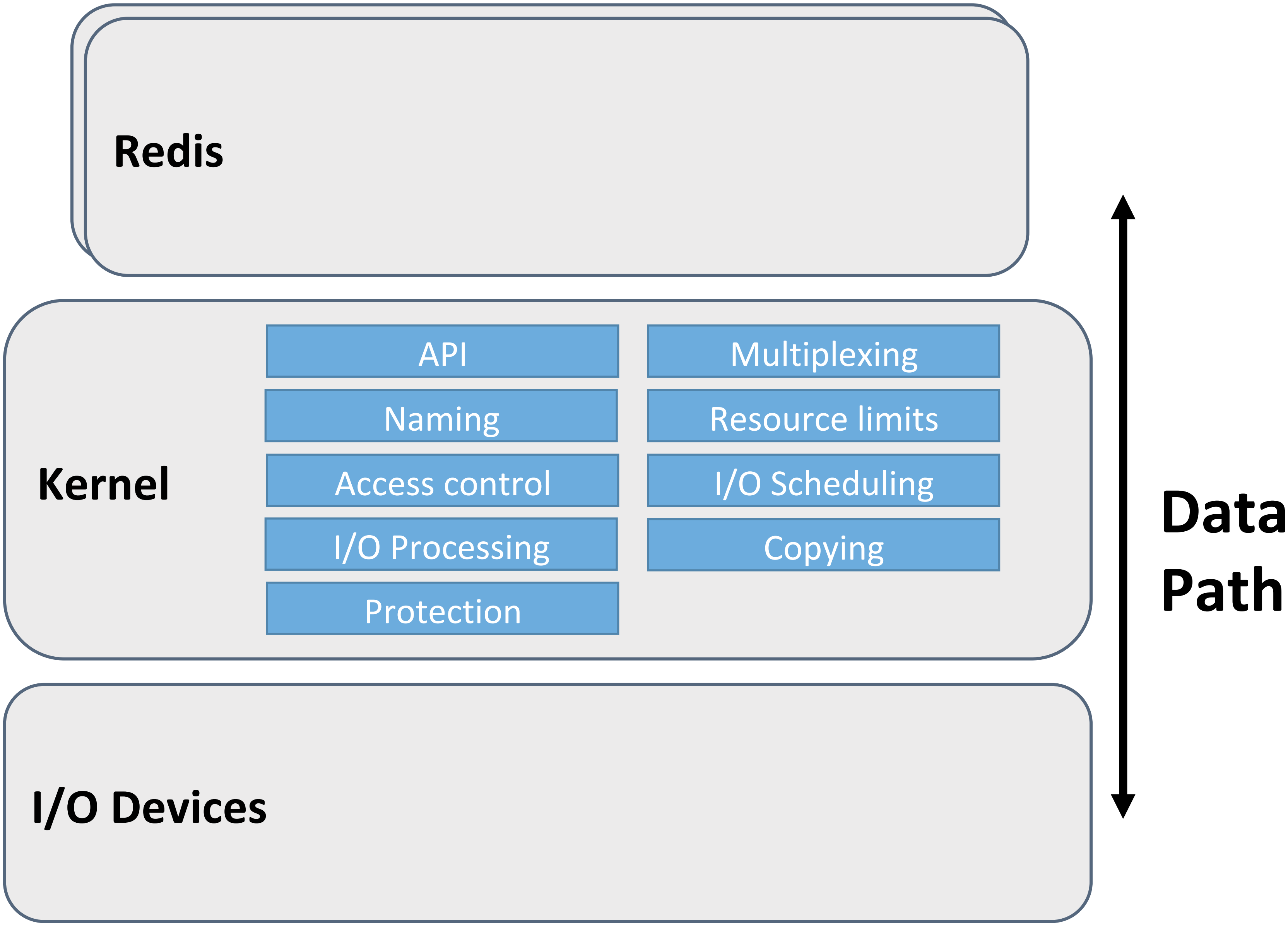
Data Path

10G NIC
2 us / 1KB packet

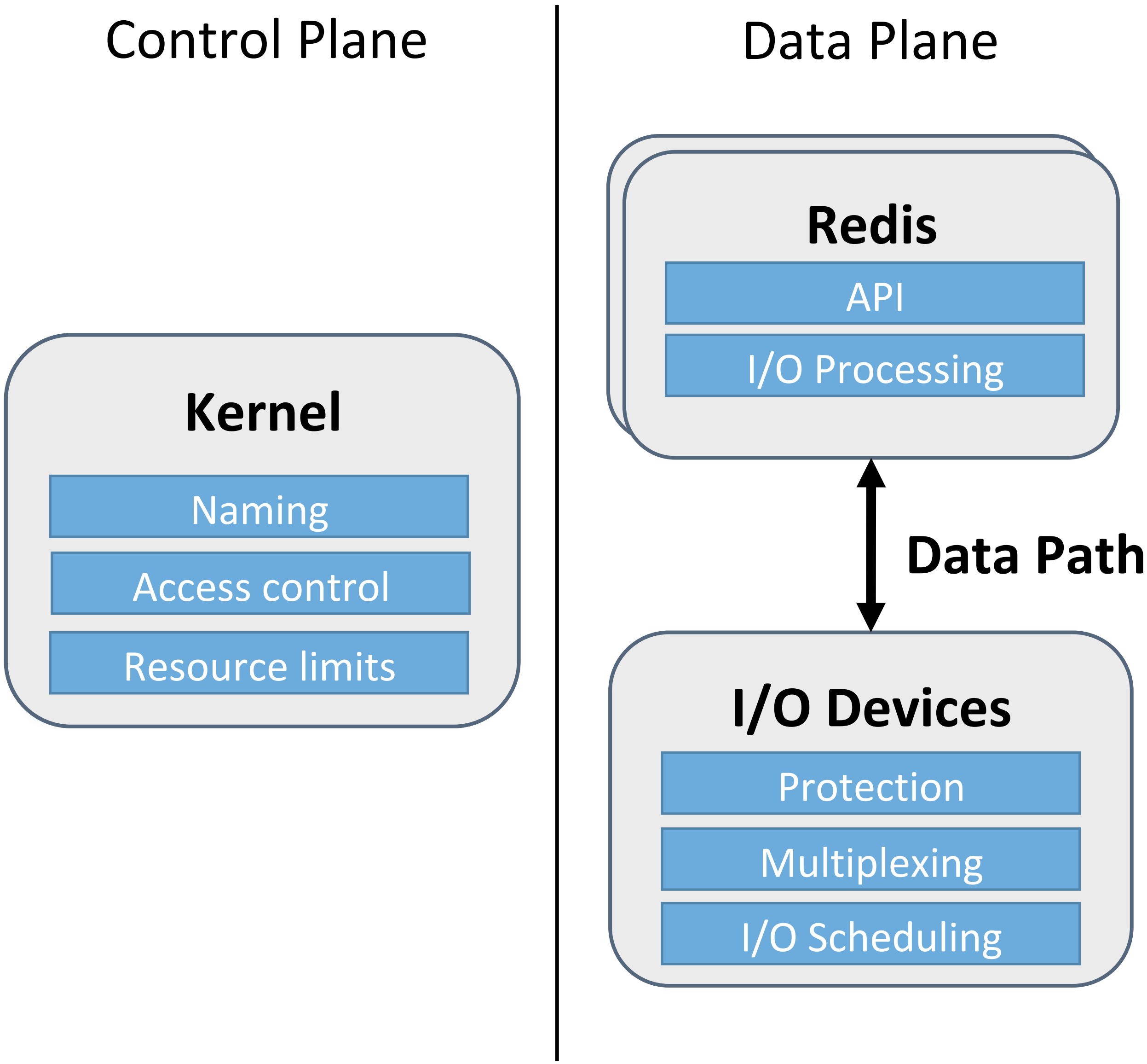
RAID Storage
25 us / 1KB write

Arrakis: Separate the OS control and data plane

How to skip the kernel?



Arrakis I/O Architecture



Design Goals

- Streamline network and storage I/O
  - Eliminate OS mediation in the common case
  - Application-specific customization vs. kernel one size fits all
- Keep OS functionality
  - Process (container) isolation and protection
  - Resource arbitration, enforceable resource limits
  - Global naming, sharing semantics
- POSIX compatibility at the application level
  - Additional performance gains from rewriting the API

This Talk

Arrakis (OSDI 14)

- OS architecture that separates the control and data plane, for both networking and storage

Strata (SOSP 17)

- File system design for low latency persistence (NVM) and multi-tier storage (NVM, SSD, HDD)

TCP as a Service/FlexNIC/Floem (ASPLOS 15, OSDI 18)

- OS, NIC, and app library support for fast, agile, secure protocol processing

Storage diversification

Byte-addressable: cache-line granularity IO
Direct access with load/store instructions

	Latency	Throughput	$/GB
DRAM	80 ns	200 GB/s	10.8
NVDIMM	200 ns	20 GB/s	2
SSD	10 us	2.4 GB/s	0.26
HDD	10 ms	0.25 GB/s	0.02

Better performance
Higher capacity

Large erasure block: hardware GC overhead
Random writes cause 5-6x slowdown by GC

Let's Build a Fast Server

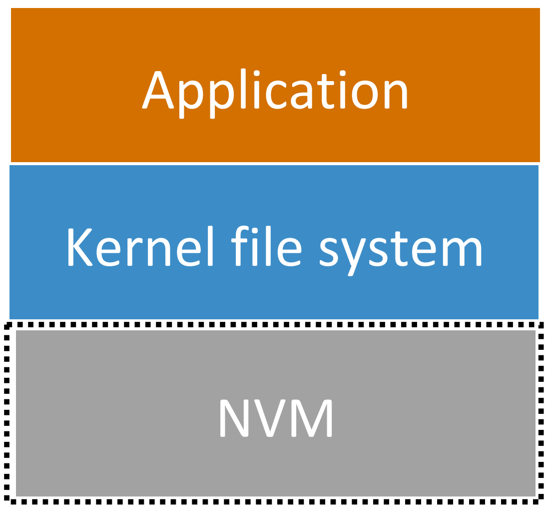
Key value store, database, file server, mail server, …

Requirements:

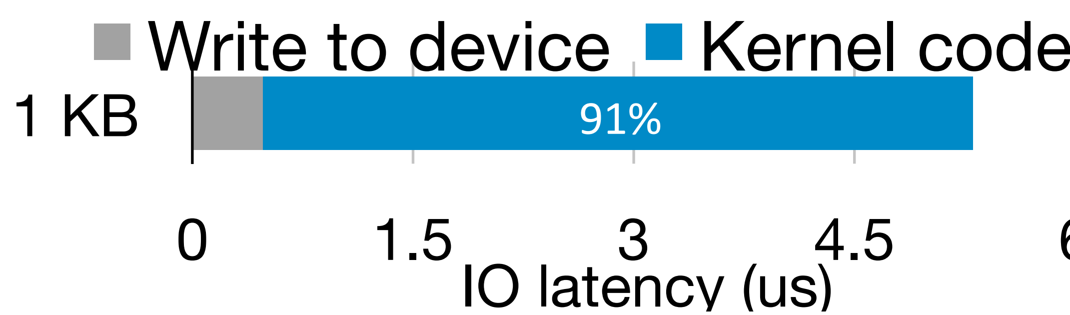
- Small updates dominate
- Dataset scales up to many terabytes
- Updates must be crash consistent

A fast server on today’s file system

- **Small updates (1 Kbytes) dominate**
- Dataset scales up to 100TB
- Updates must be crash consistent



Small, random IO is slow!

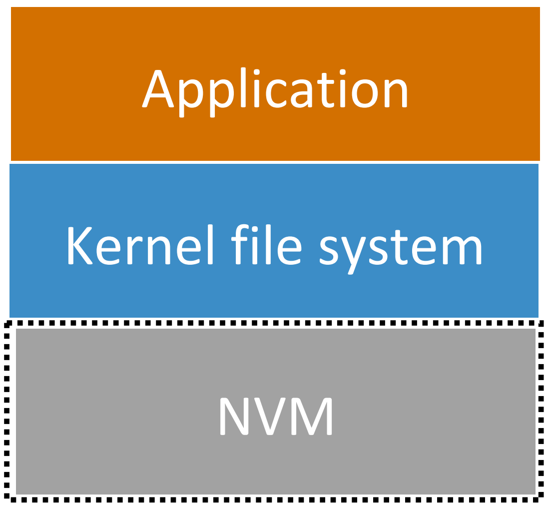


Kernel file system:
NOVA [FAST 16, SOSP 17]

Even with an optimized kernel file system, NVM is too fast, kernel is the bottleneck

A fast server on today's file system

- Small updates (1 Kbytes) dominate
- **Dataset scales up to 100TB**
- Updates must be crash consistent

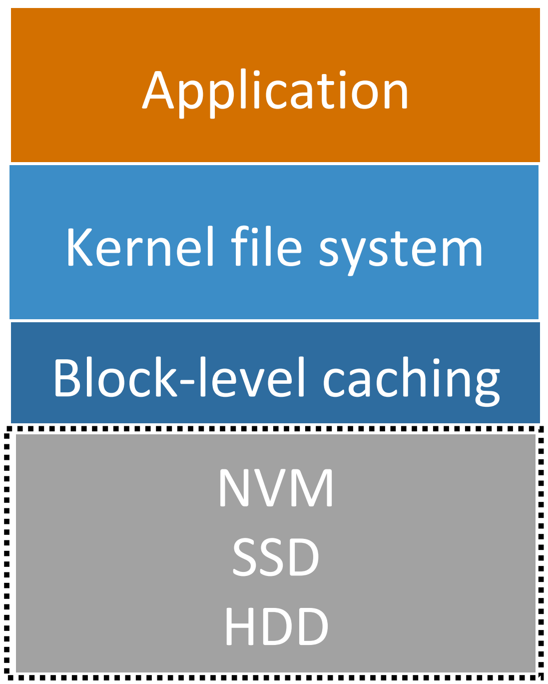


Using only NVM is too expensive!
$200K for 100TB

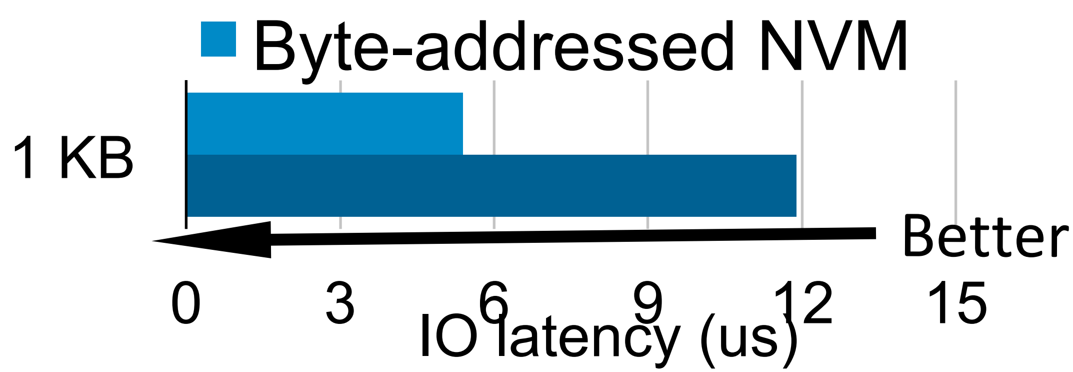
To save cost, need a way to use multiple device types: NVM, SSD, HDD

A fast server on today's file system

- Small updates (1 Kbytes) dominate
- **Dataset scales up to 10TB**
- Updates must be crash consistent



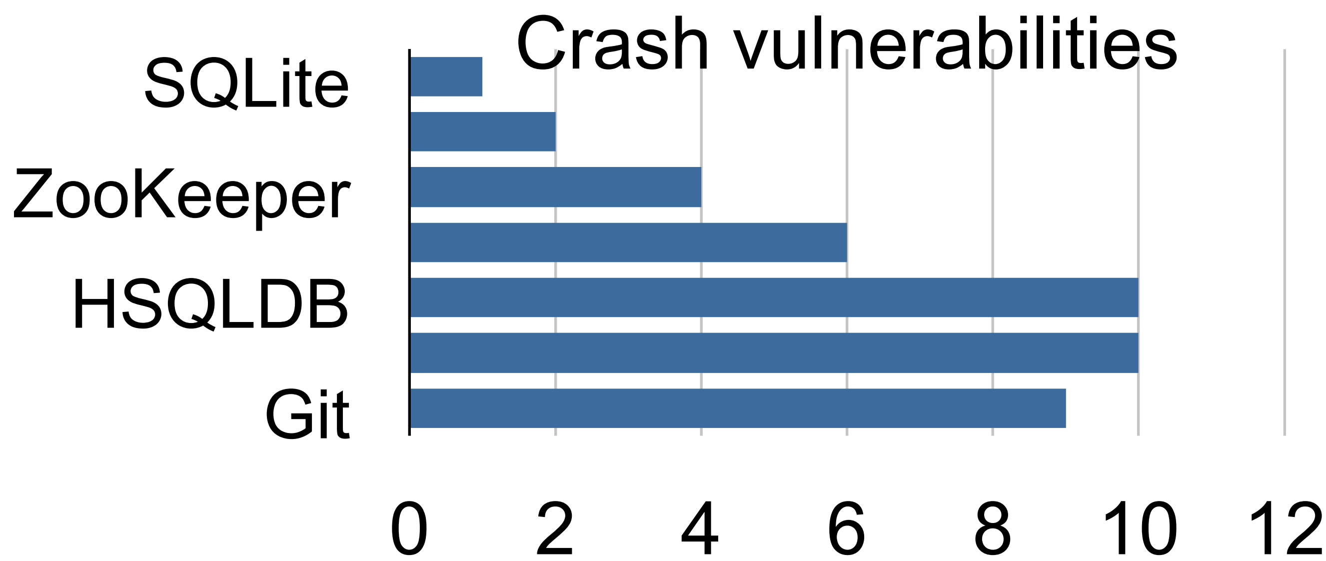
Block-level caching manages data in blocks, but NVM is byte-addressable!



For low-cost capacity with high performance, must leverage multiple device types

A fast server on today's file system

- Small updates (1 Kbytes) dominate
- Dataset scales up to 10TB
- **Updates must be crash consistent**



Pillai et al., OSDI 2014

Applications struggle for crash consistency

Today's file systems:
Limited by old design assumptions

- Kernel mediates every operation
  - **NVM is too fast, kernel is the bottleneck**
- Tied to a single type of device
  - **For low-cost capacity with high performance,**
  - **must leverage multiple device types (NVM, SSD, HDD)**
- Aggressive caching in DRAM, only write to device when you must (fsync)
  - **Applications struggle for crash consistency**

Strata: A Cross Media File System

Performance: Especially small, random IO

- Fast user-level device access

Capacity: leverage NVM, SSD & HDD for low cost

- Transparent data migration across different media
- Efficiently handle device IO properties

Simplicity: intuitive crash consistency model

- In-order, synchronous IO
- No fsync() required

Strata: main design principle

Performance, simplicity:

LibFS Log operations to NVM at user-level

Intuitive crash consistency

Kernel bypass, but private

Capacity:

KernelFS Digest and migrate data in kernel

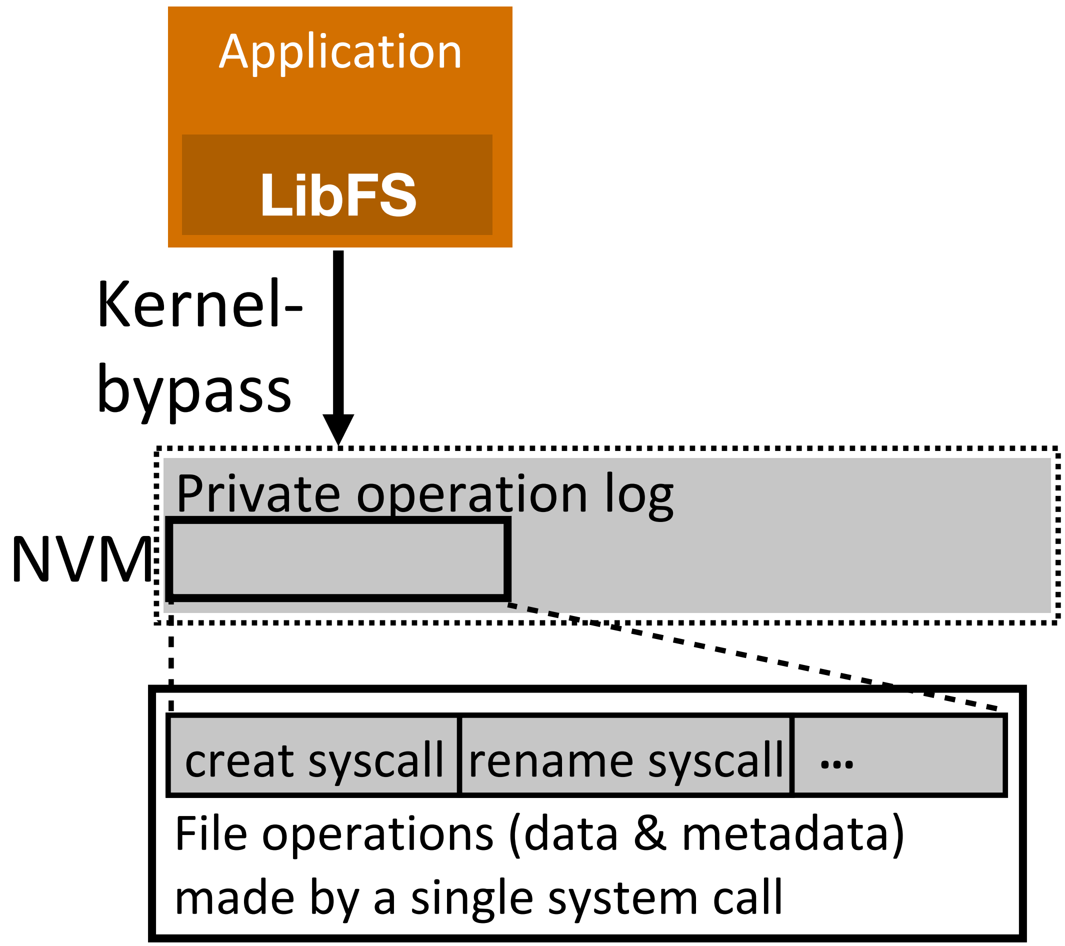
Apply log operations to shared data

Coordinate multi-process accesses

Strata

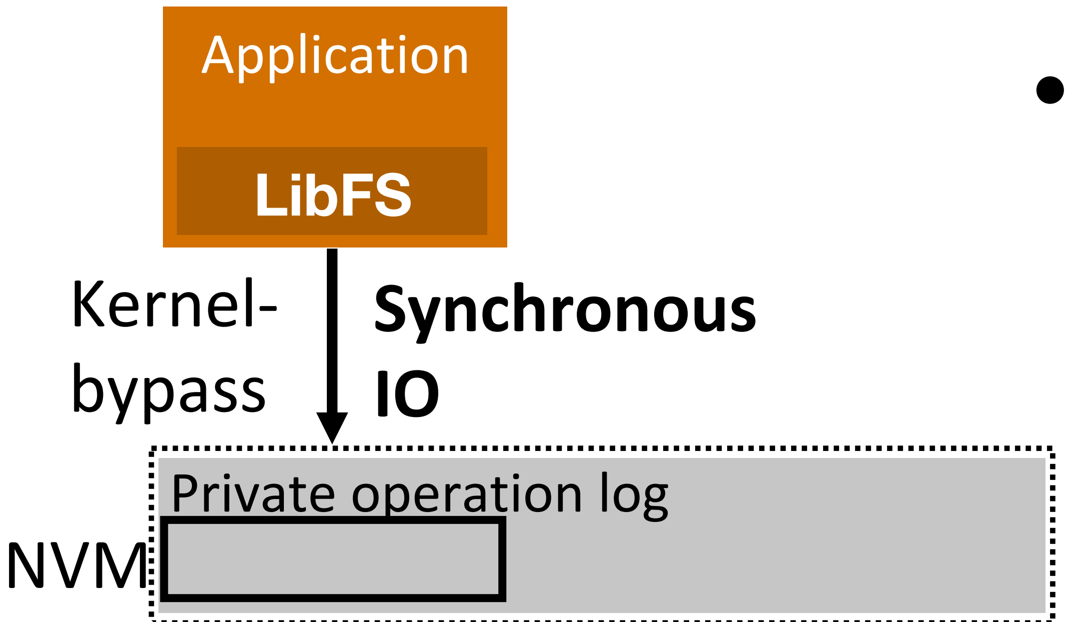
- LibFS: log operations to NVM at user-level
  - Fast user-level access
  - In-order, synchronous IO
- KernelFS: Digest and migrate data in kernel
  - Asynchronous digest
  - Transparent data migration
  - Shared file access

Log operations to NVM at user-level



- Fast writes
  - Directly access fast NVM
  - Sequentially append data
  - Cache-line granularity
    - Blind writes
- Crash consistency
  - On crash, kernel replays log

Intuitive crash consistency

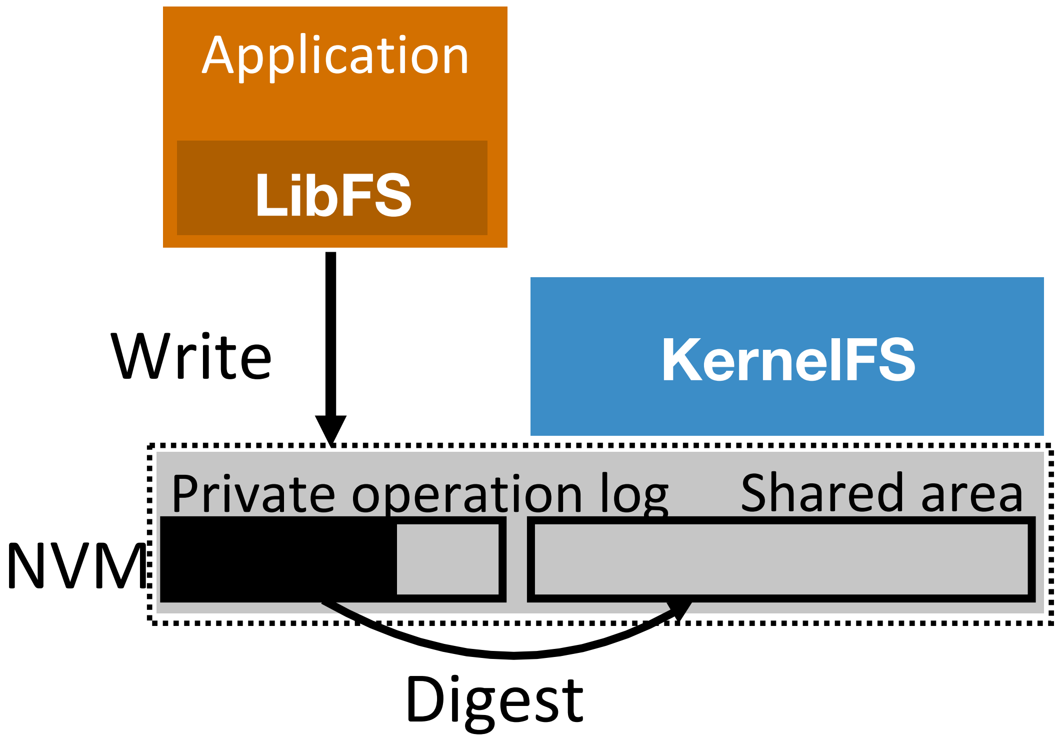


- When each system call returns:
  - Data/metadata is durable
  - In-order update
  - Atomic write
    - Limited size (log size)

fsync() is no-op

Fast synchronous IO: NVM and kernel-bypass

Digest data in kernel



- **Visibility:**
  make private log visible to other applications
- **Data layout:**
  turn write-optimized to read-optimized format
- Large, batched IO
  - Coalesce log

Digest optimization: Log coalescing

SQLite, Mail server: crash consistent update using write ahead logging

Application

LibFS

Digest eliminates unneeded work

Remove
temporary durable writes

KernelFS

Private operation log

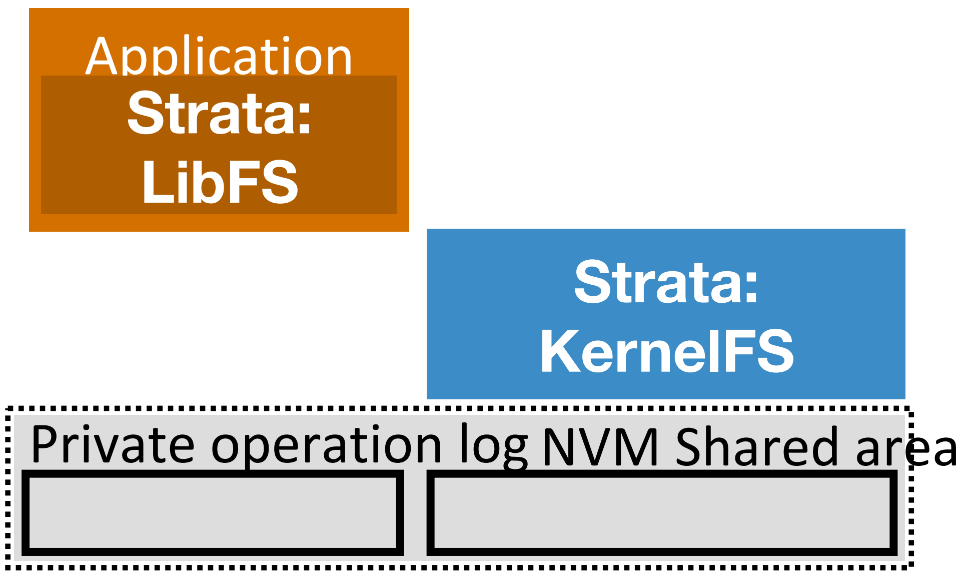
```
Create journal file
Write data to journal file
Write data to database file
Delete journal file
```

NVM Shared area

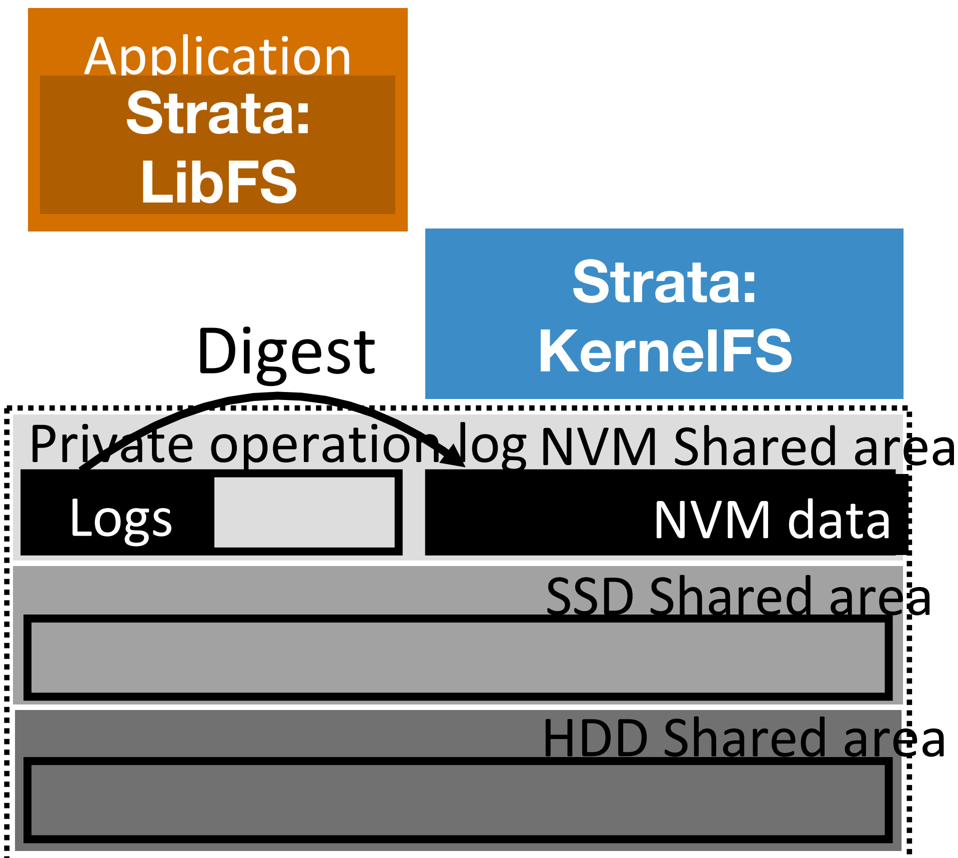
```
Write data to database file
```

Throughput optimization:
Log coalescing saves IO while digesting

Digest and migrate data in kernel



Digest and migrate data in kernel



Low-cost capacity

KernelFS migrates data to lower layer

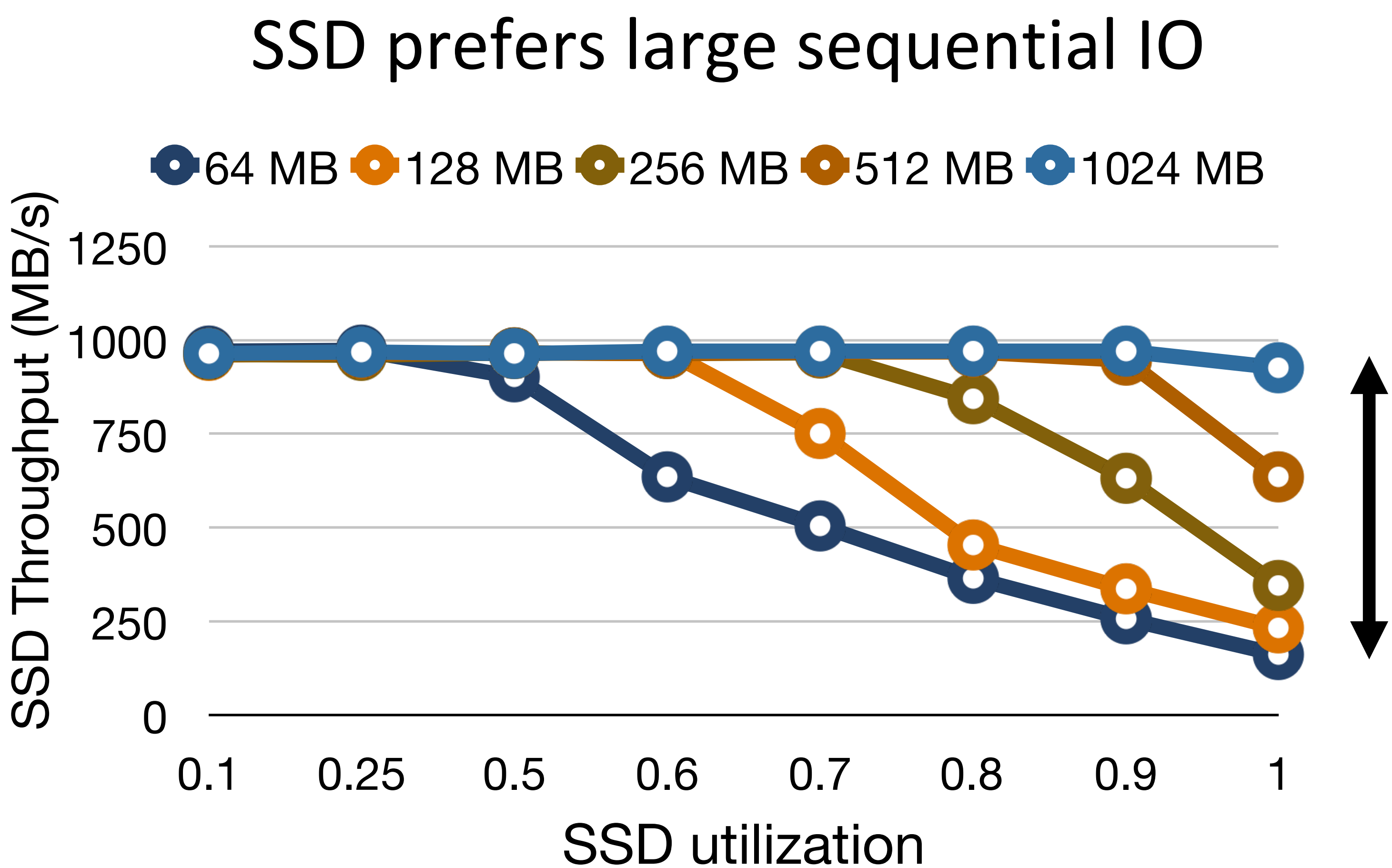
Handle device IO properties

Write 1 GB sequentially from NVM to SSD

Avoid SSD garbage collection overhead

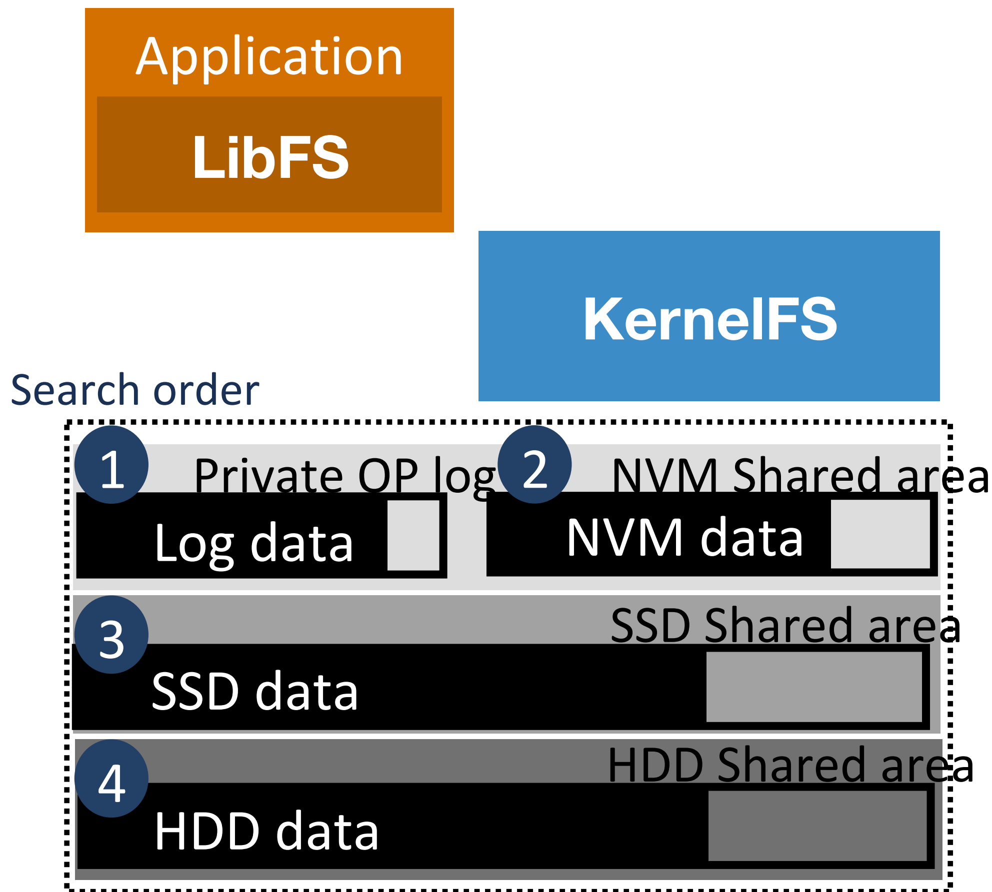
Resembles log-structured merge (LSM) tree

Device management overhead



Use NVM layer as **persistent** write buffer

Read: hierarchical search



Shared file access

Leases grant access rights to applications [SOSP’89]

- Required for files and directories
- Function like lock, but revocable
- Exclusive writer, shared readers

On revocation, LibFS digests leased data

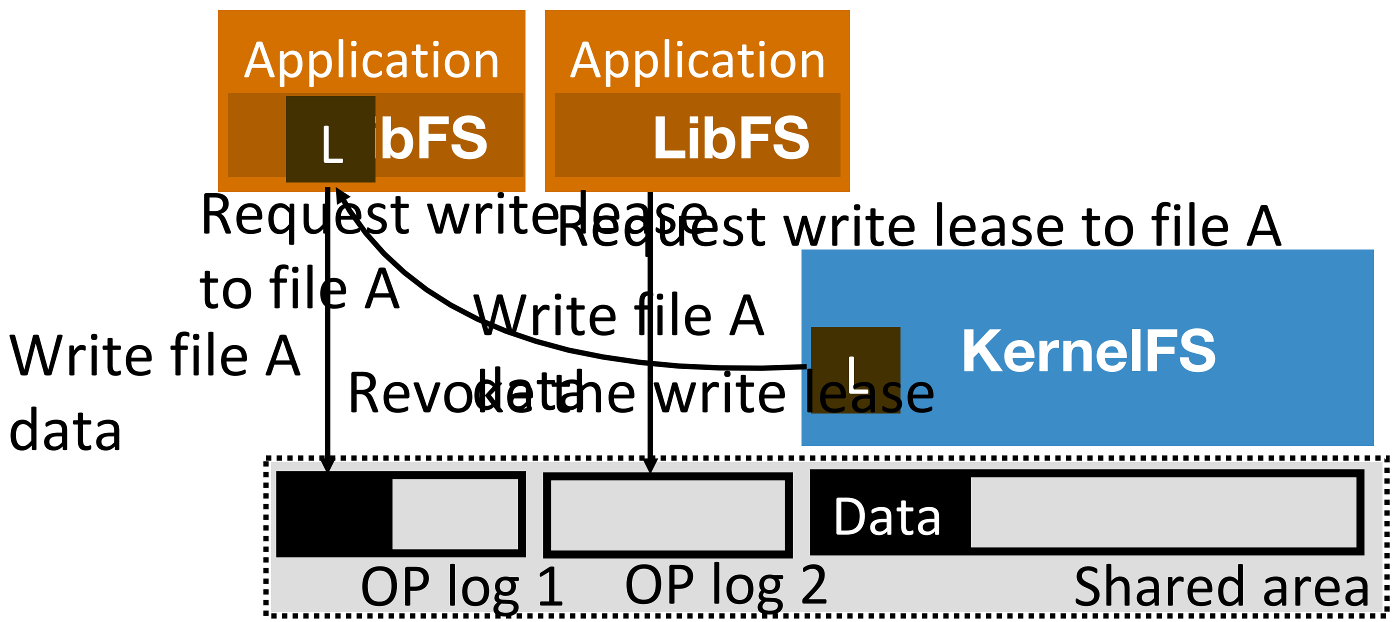
Leases serialize concurrent updates

Shared file access

Leases grant access rights to applications

- Applied to a directory or a file
- Exclusive writer, shared readers

Example: concurrent writes to the same file A



Experimental setup

- 2x Intel Xeon E5-2640 CPU, 64 GB DRAM
  - 400 GB NVMe SSD, 1 TB HDD
  - Ubuntu 16.04 LTS, Linux kernel 4.8.12
- Emulated NVM
  - Use 40 GB of DRAM
  - Performance model [Y. Zhang et al. MSST 2015]
    - Throttle latency & throughput in software
- Compare Strata vs.
  - PMFS, Nova, ext4-DAX: NVM kernel file systems
  - Nova: atomic update, in-order synch I/O
  - PMFS, ext4-DAX: no atomic write

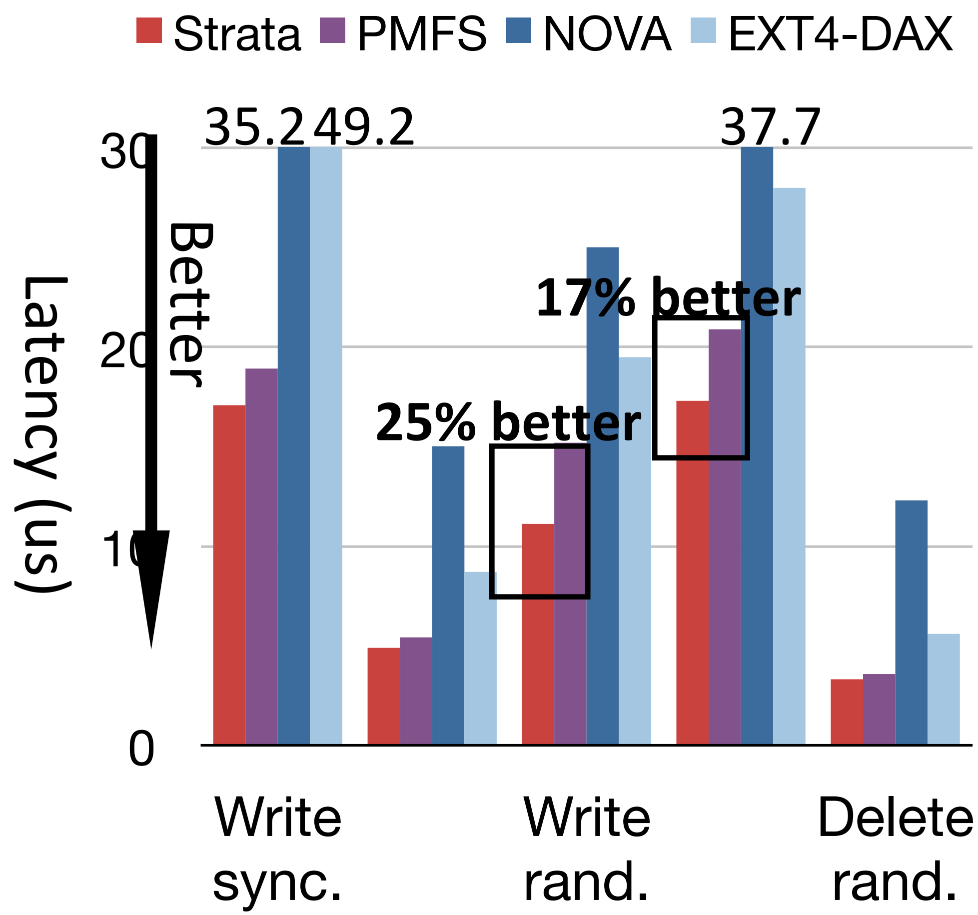
Latency: LevelDB

LevelDB (NVM)

- Key size: 16 B
- Value size: 1 KB
- 300,000 objects
- Level compaction causes asynchronous digests

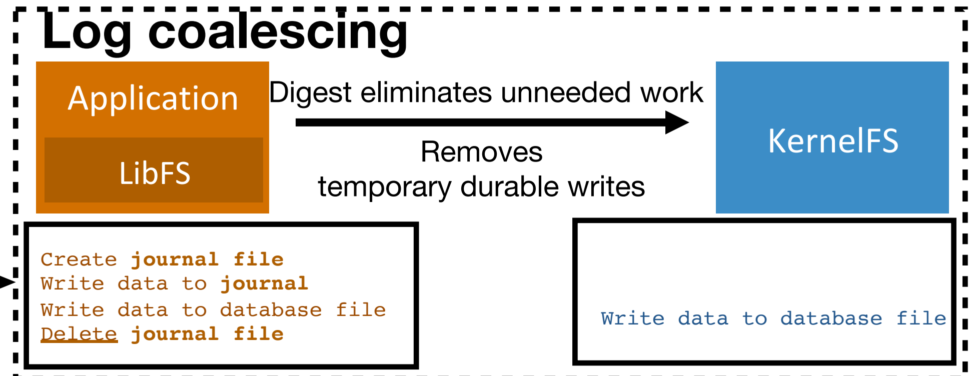
Fast user-level logging

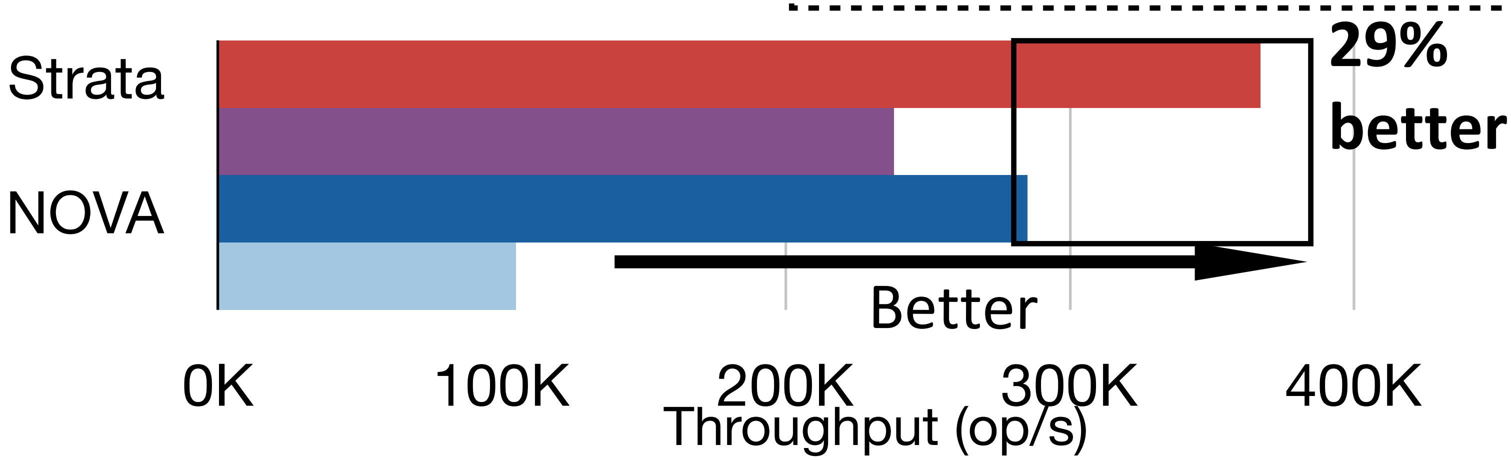
- Random write
  - 25% better than PMFS
- Overwrite
  - 17% better than PMFS



Throughput: Varmail

Mail server workload from Filebench
- Using only NVM
- 10000 files
- Read/Write ratio is 1:1
- **Write-ahead logging**





Log coalescing eliminates 86% of log records, saving 14 GB of IO

This Talk

Arrakis (OSDI 14)

- OS architecture that separates the control and data plane, for both networking and storage

Strata (SOSP 17)

- File system design for low latency persistence (NVM) and multi-tier storage (NVM, SSD, HDD)

TCP as a Service/FlexNIC/Floem (ASPLOS 15, OSDI 18)

- OS, NIC, and app library support for fast, agile, secure protocol processing

Let’s Build a Fast Server

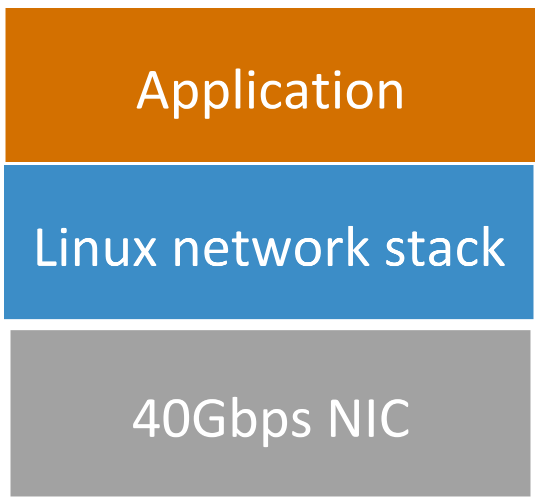
Key value store, database, mail server, ML, …

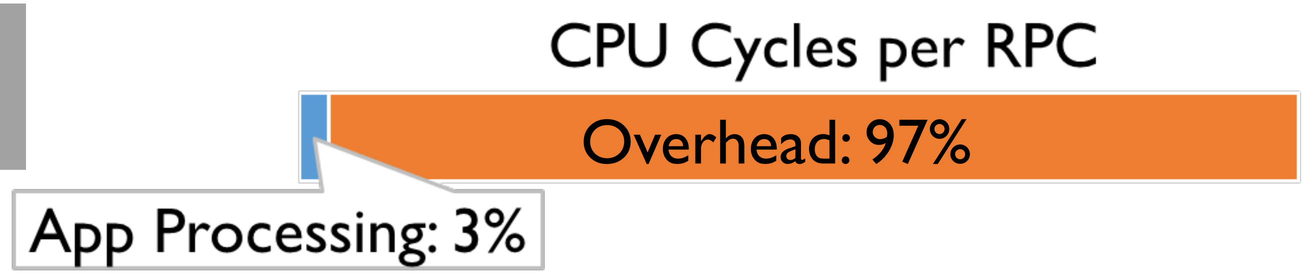
Requirements:

- Mostly small RPCs over TCP
- 40 Gbps network links (100+ Gbps soon)
- Enforceable resource sharing (multi-tenant)
- Agile protocol development: kernel and app
- Tail latency, cost efficient hardware

Let’s Build a Fast Server

- **Small RPCs dominate**
- Enforceable resource sharing
- Agile protocol development
- Cost-efficient hardware

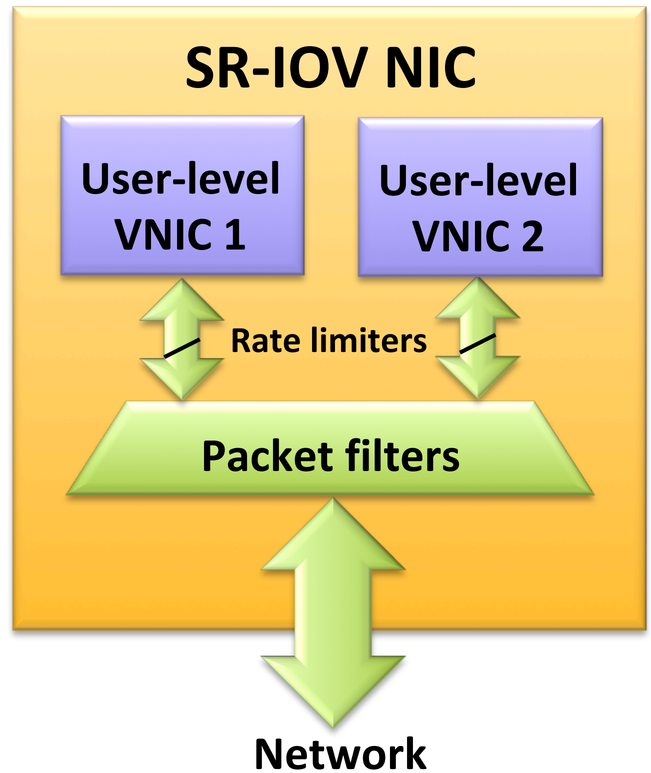




Kernel mediation too slow

Hardware I/O Virtualization

- Direct access to device at user-level
- Multiplexing
  - **SR-IOV**: Virtual PCI devices w/ own registers, queues, INTs
- Protection
  - **IOMMU**: DMA to/from app virtual memory
  - **Packet filters**: ex: legal source IP header
- mTCP: 2-3x faster than Linux
- **Who enforces congestion control?**



Remote DMA (RDMA)

Programming model: read/write to (limited) region of remote server memory

- Model dates to the 80's (Alfred Spector)
- HPC community revived for communication within a rack
- Extended to data center over Ethernet (RoCE)
- Commercially available 100G NICs

No CPU involvement on the remote node

- Fast if app can use programming model

Limitations:

- **What if you need remote application computation (RPC)?**
- **Lossless model is performance-fragile**

Smart NICs (Cavium, …)

NIC with array of low-end CPU cores (Cavium, …)

If compute on the NIC, maybe don't need to go CPU?

- Applications in high speed trading

We've been here before: "wheel of reinvention"

- **Hardware relatively expensive**
- **Apps often slower on NIC vs. CPU (cf. Floem)**

Step 1

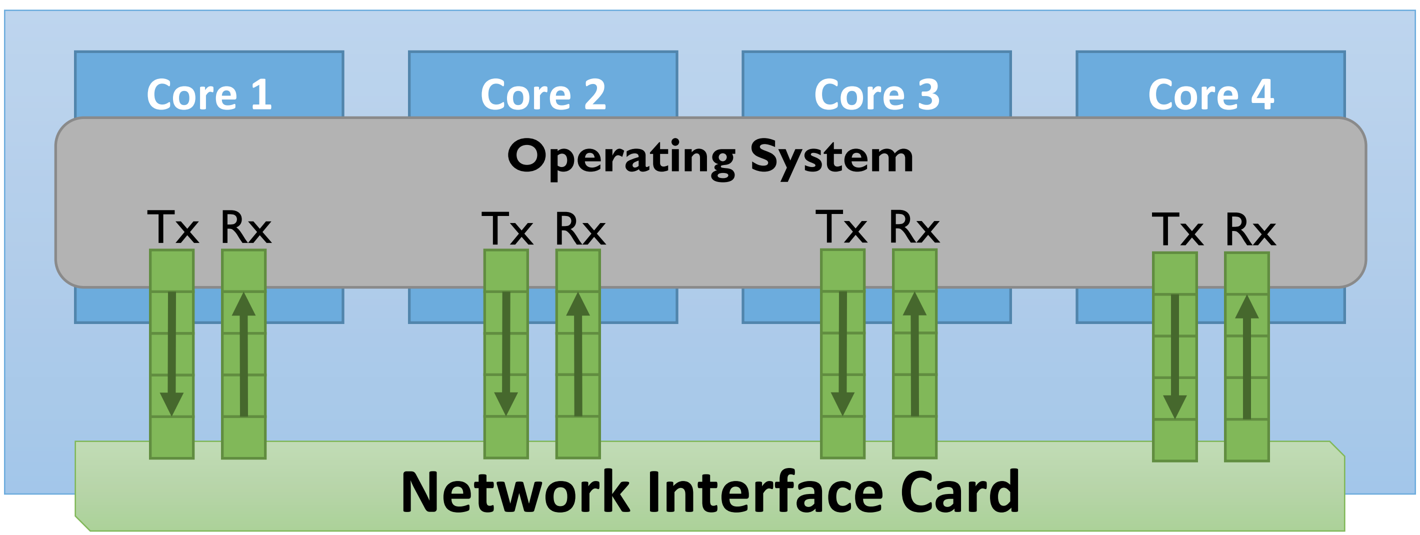
Build a faster kernel TCP in software

No change in isolation, resource allocation, API

Q: Why is RPC over Linux TCP is so slow?

OS - Hardware Interface

- Highly optimized code path
- Buffer descriptor queues
  - No interrupt in common case
  - Maximize concurrency



OS Transmit Packet Processing

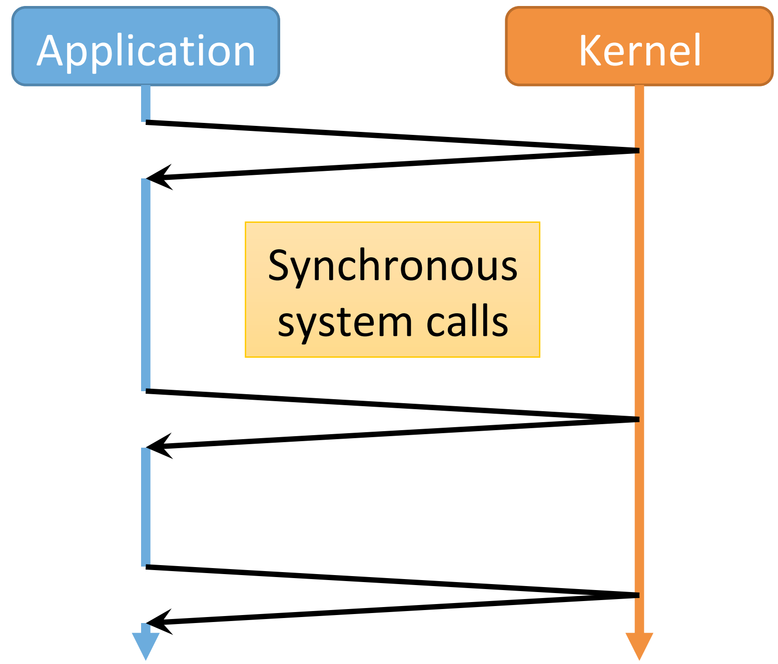
- TCP layer: move from socket buffer to IP queue
  - Lock socket
  - Congestion/flow contol limit
  - Fill in TCP header, calculate checksum
  - Copy data
  - Arm re-transmission timeout
- IP layer:
  - firewall, routing, ARP, traffic shaping
- Driver: move from IP queue to NIC queue
- Allocate and free packet buffers

Sidebar: Tail Latency

On Linux with a 40Gbps link, 400 outbound TPC flows sending RPCs, no congestion, what is the minimum rate across all flows?

Kernel and Socket Overhead

```
events = poll(….)
For e in events:
    If e.socket != listen_sock:
        receive(e.socket, ….)
        …
        send(e.socket, ….)
```



Multiple synchronous kernel transitions:

- Parameter checks and copies
- Cache pollution, pipeline stalls

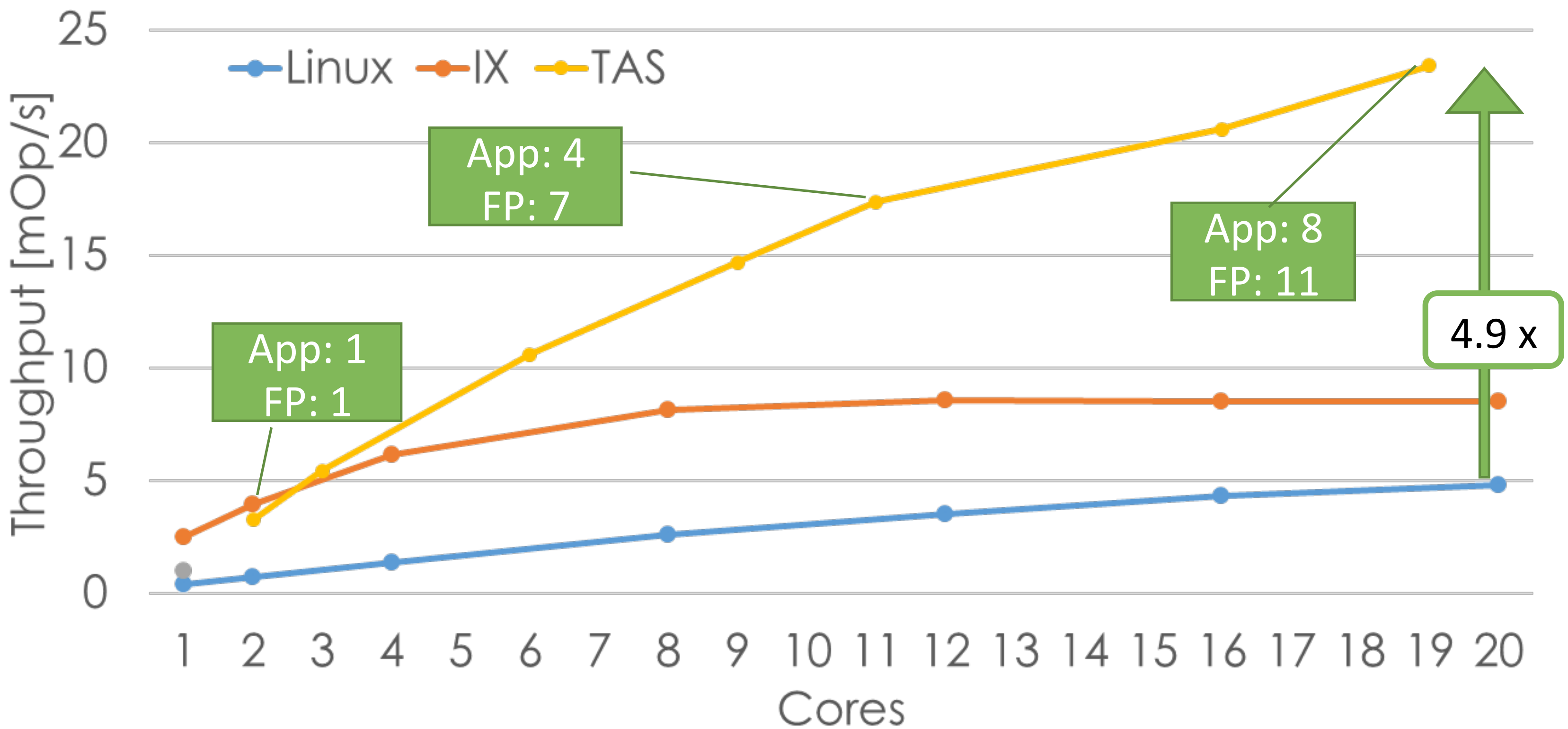
TCP Acceleration as a Service (TaS)

- TCP as a user-level OS service
  - SRIO-V to dedicated cores
  - Scale number of cores up/down to match demand
  - Optimized data plane for common case operations
- Application uses its own dedicated cores
  - Avoid polluting application level cache
  - Fewer cores => better performance scaling
- To the application, per-socket tx/rx queues with doorbells
  - Analogous to hardware device tx/rx queues

Streamline common-case data path

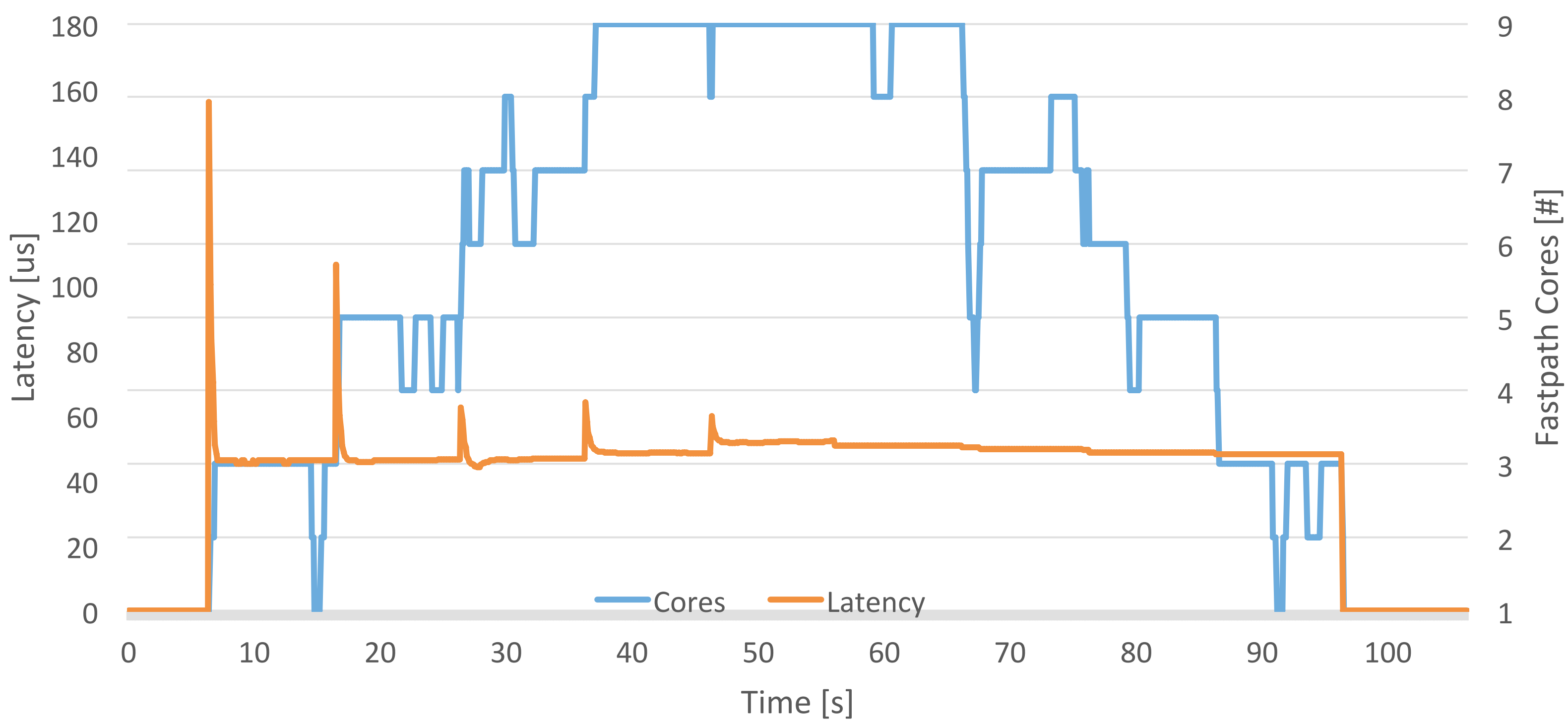
- Remove unneeded computation from data path
  - Congestion control, timeouts per RTT (not per packet)
- Minimize per-flow TCP state
  - prefetch 2 cache lines on packet arrival
- Linearized code
  - Better branch prediction
  - Super-scalar execution
- Enforce IP level access control on control plane at connection setup

Small RPC Microbenchmark



- IX: fast kernel TCP with syscall batching, non-socket API
- Linux/TaS latency: 7.3x; IX/TaS latency: 2.9x

TAS is workload proportional



- Setup: 4 clients starting every 10 seconds, then stopping incrementally

Step 2

TCP as a Service can saturate a 40Gbps link with small RPCs, but what about 100Gbps or 400Gbps links?

Network link speeds scaling up faster than cores

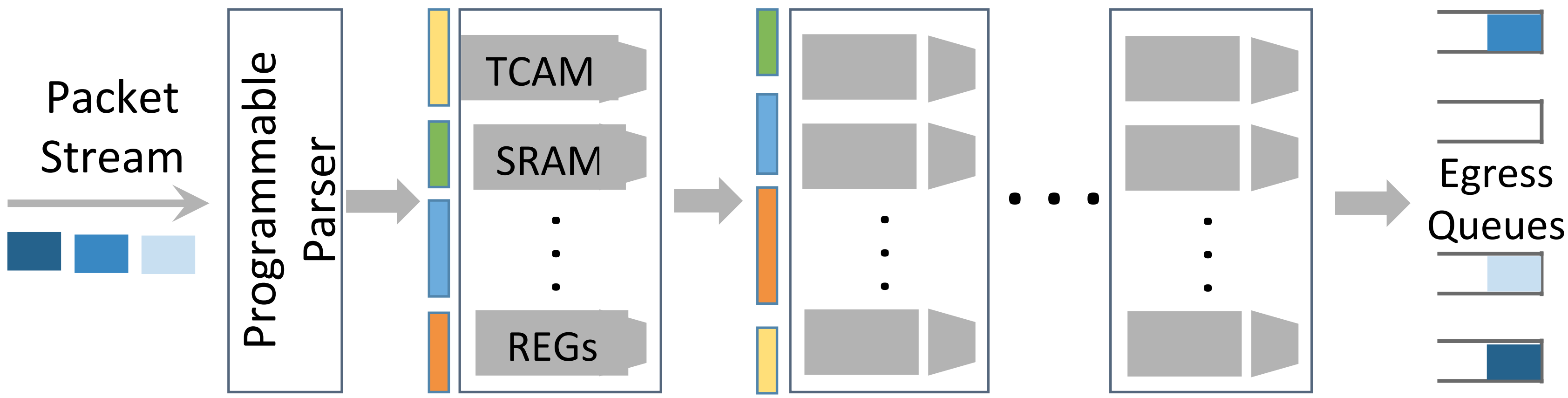
What NIC hardware do we need fast data center communication?

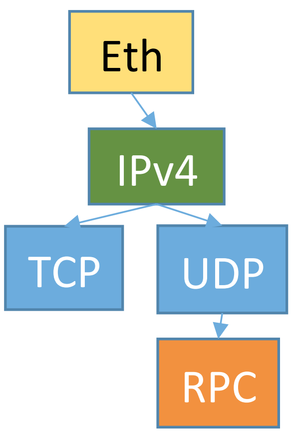
TCP as a Service data plane can be efficiently built in hardware

FlexNIC Design Principles

- RPCs are the common case
  - Kernel bypass to application logic
- Enforceable per-flow resource sharing
  - Data plane in hardware, policy in kernel
- Agile protocol development
  - Protocol agnostic (ex: Timely and DCTCP and RDMA)
  - Offload both kernel and app packet handling
- Cost-efficient
  - Minimal instruction set for packet processing

FlexNIC Hardware Model





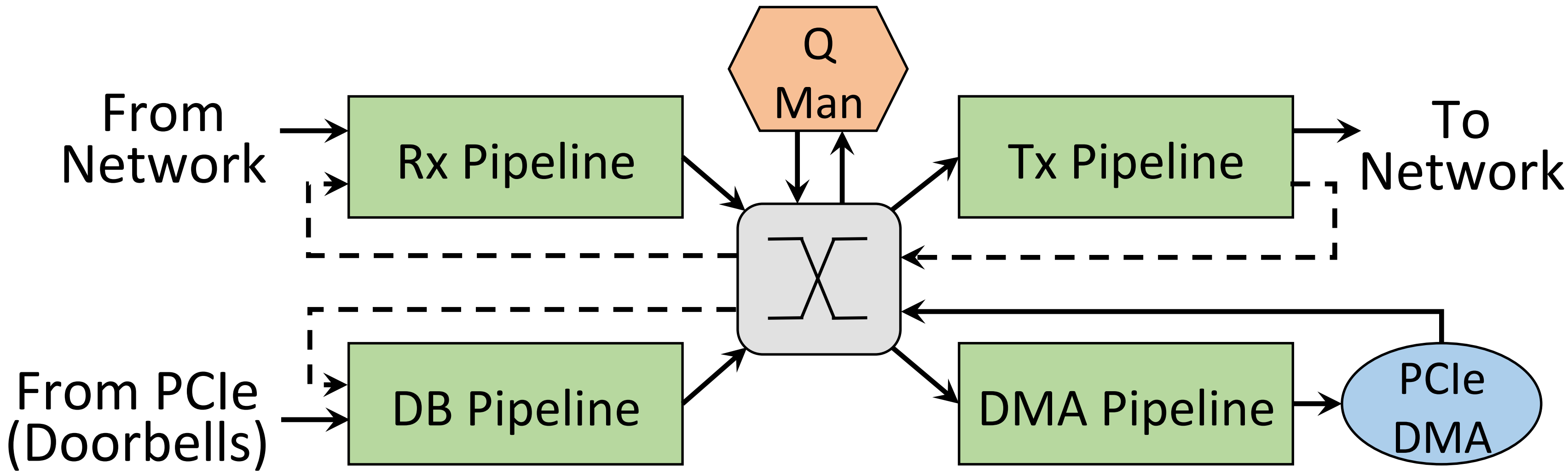
- TCAM for arbitrary wildcard matches
- SRAM for exact/LPM lookups
- Stateful memory for counters
- ALUs for modifying headers and registers

Match (TCAM, SRAM, stateful memory) — Action (ALUs)

1. p = lookup(eth.dst_mac)
2. pkt.egress_port = p
3. counter [ipv4.src_ip] ++

Barefoot Networks switch RMT ~ 6 Tbps (with parallel streams)

FlexNIC Hardware Model



- Transform packets for efficient processing in SW
- DMA directly into and out of application data structures
- Send acknowledgements on NIC
- Queue manager implements rate limits
- Improve locality by steering to cores based on app criteria

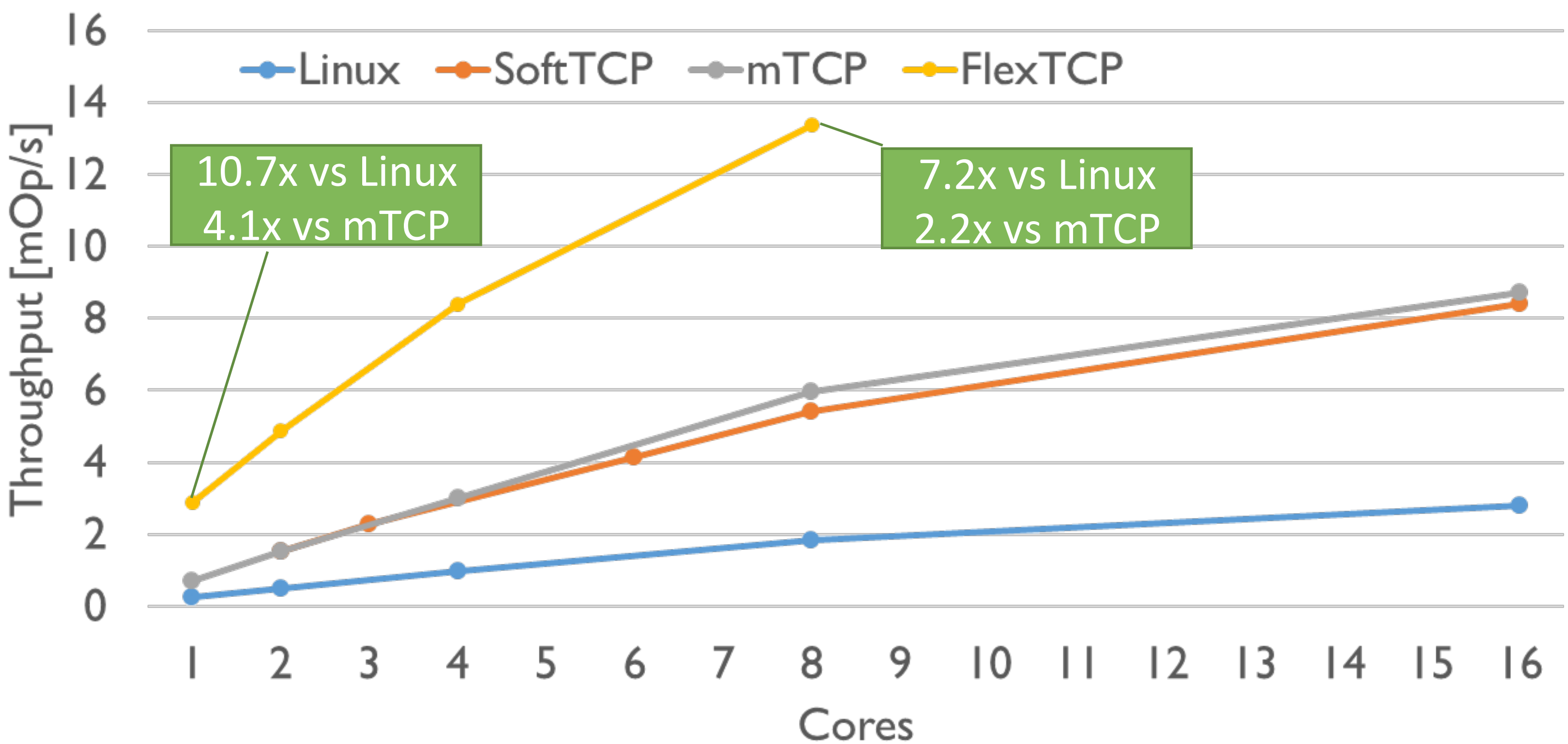
FlexTCP: H/W Accelerated TCP

- Fast path is simple enough for FlexNIC model
- Applications directly access NIC for RX/TX
  - Similar interface to TCP as a service: in-memory queues
- Software slow-path manages NIC state

- Streamlines NIC processing
  - ACKs consumed/generated in NIC, reduces PCIe traffic
  - No descriptor queues w/ dependent DMA reads

- **Evaluation**: software FlexNIC emulator

FlexTCP Performance



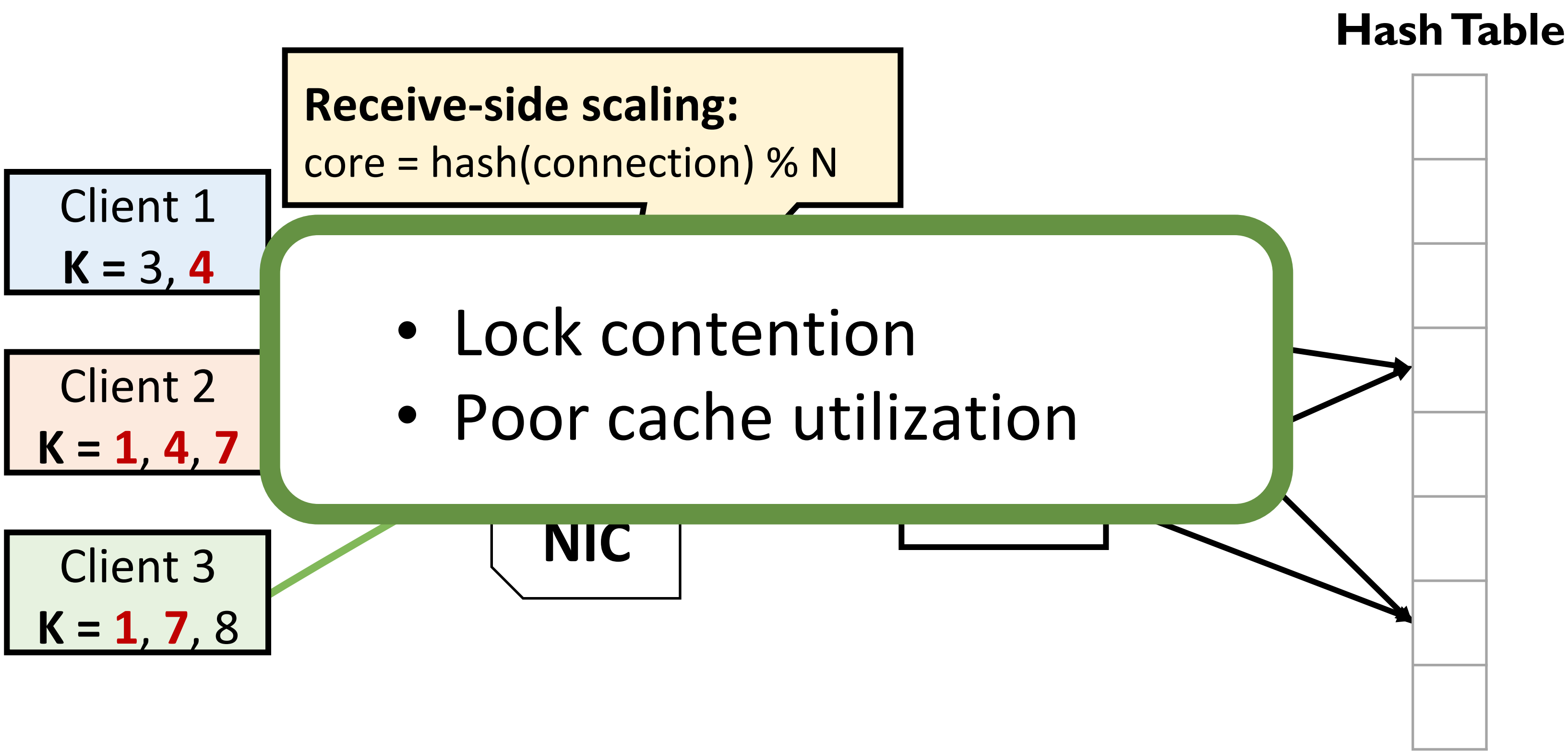
- Latency: 7.8x better vs Linux

Streamlining App RPC Processing

- How do we further reduce CPU overhead?
- Integration of app logic with network code
  - Remove socket abstraction, streamline app code
- But fundamental app bottlenecks remain
  - Data structure synchronization, cache utilization
  - Copies between app data structures and RPC buffers

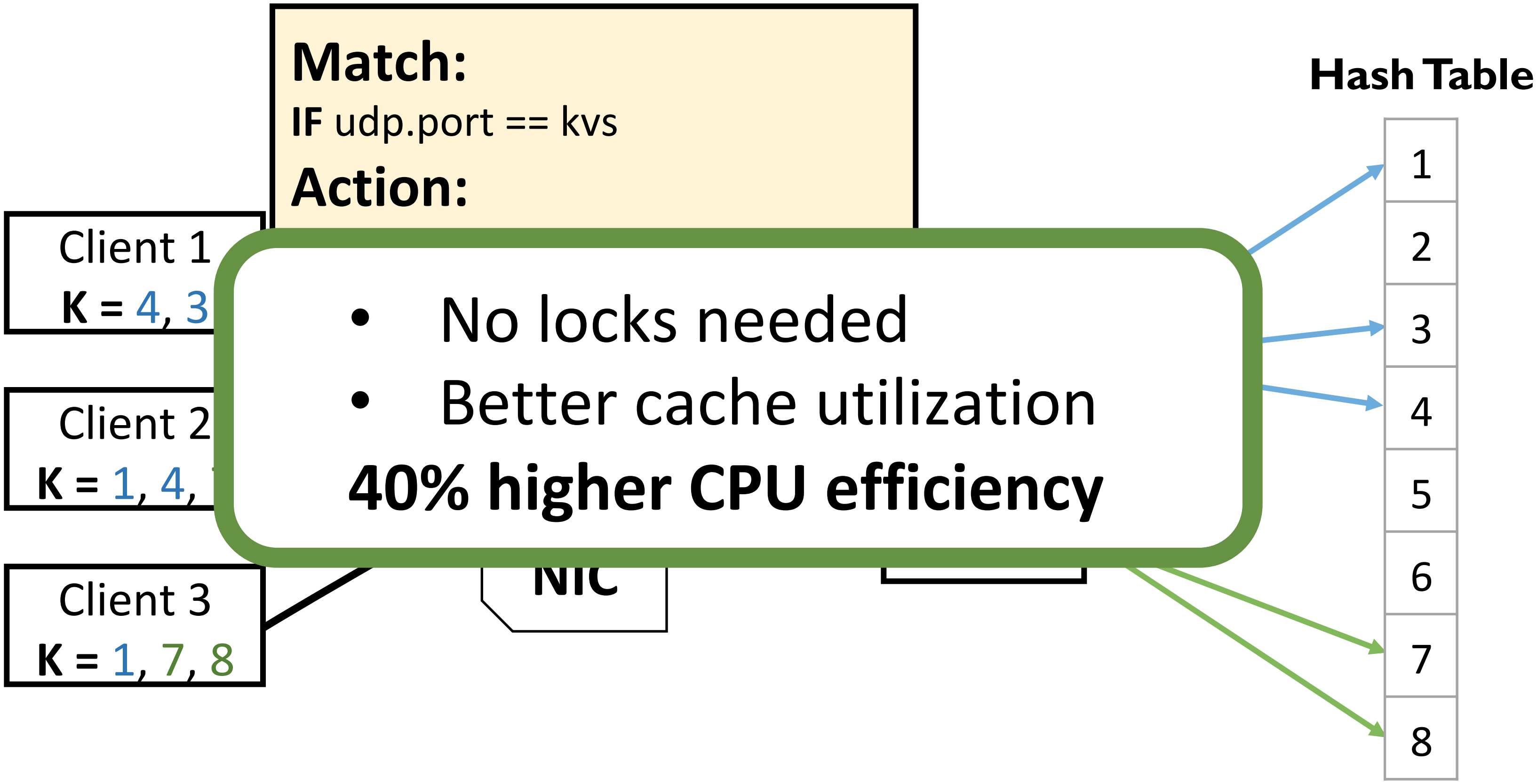
- **Idea:** leverage FlexNIC to streamline app
  - FlexNIC is protocol agnostic, can parse on app protocol

Example: Key-Value Store



- Lock contention
- Poor cache utilization

Optimizing Reads: Key-based Steering



- No locks needed
- Better cache utilization

40% higher CPU efficiency

Summary

Arrakis (OSDI 14)

- OS architecture that separates the control and data plane, for both networking and storage

Strata (SOSP 17)

- File system design for low latency persistence (NVM) and multi-tier storage (NVM, SSD, HDD)

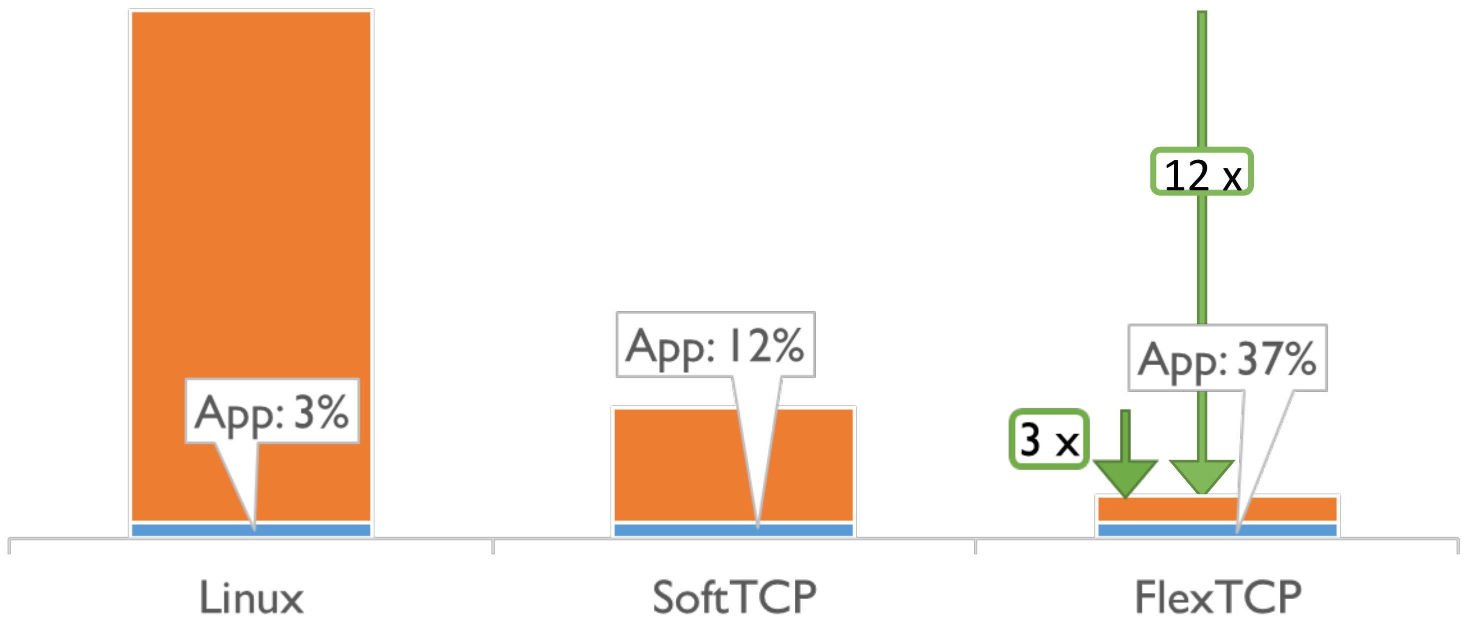
TCP as a Service/FlexNIC/Floem (ASPLOS 15, OSDI 18)

- OS, NIC, and app library support for fast, agile, secure protocol processing

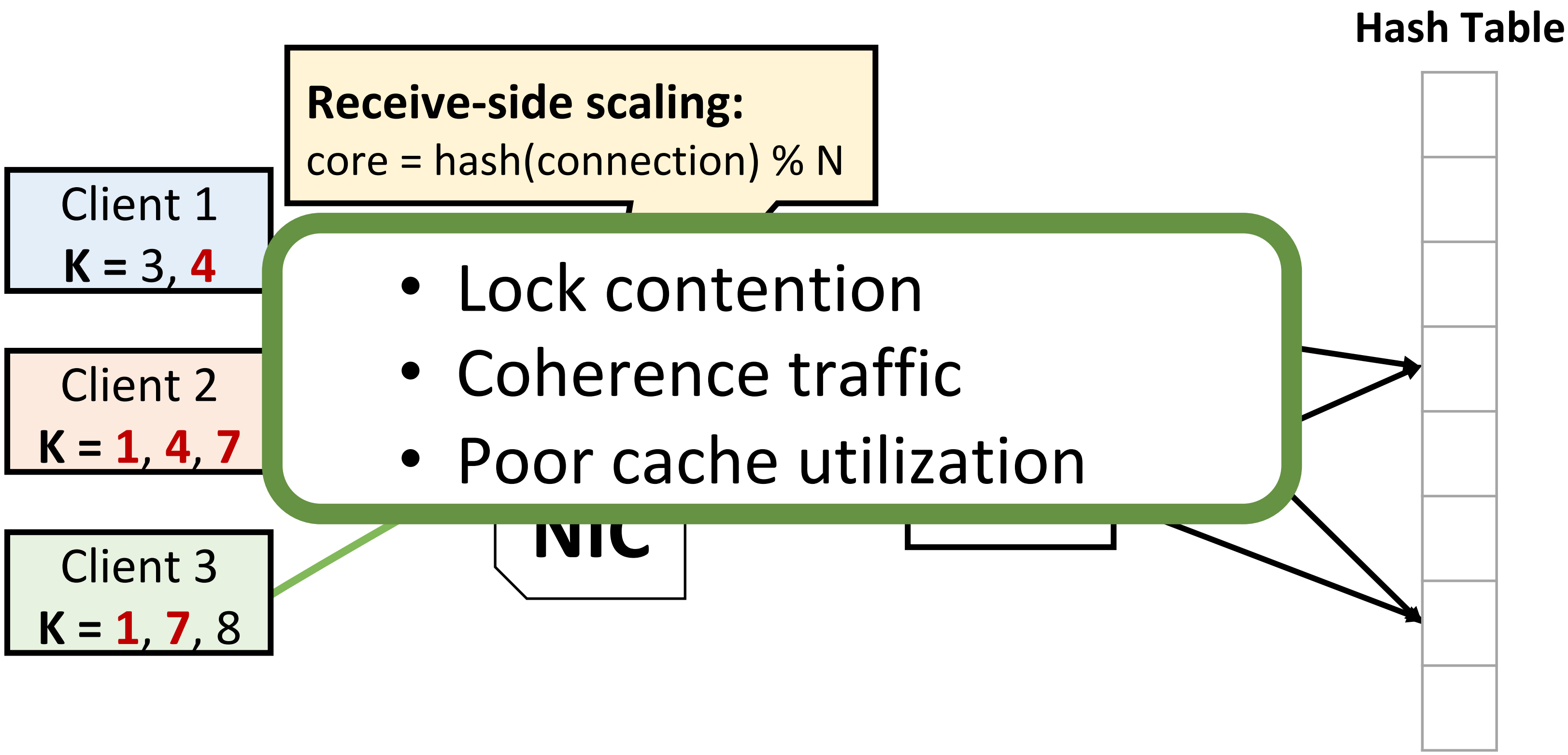
Biography

- College: physics -> psychology -> philosophy
  - Took three CS classes as a senior
- After college: developed an OS for a z80
  - After project shipped, project got cancelled
  - Applied to grad school: seven out of eight turned me down
- Grad school
  - Learned a lot
  - Dissertation had zero commercial impact for decades
- As faculty
  - I learn a lot from the people I work with
  - Try to pick topics that matter, and where I get to learn

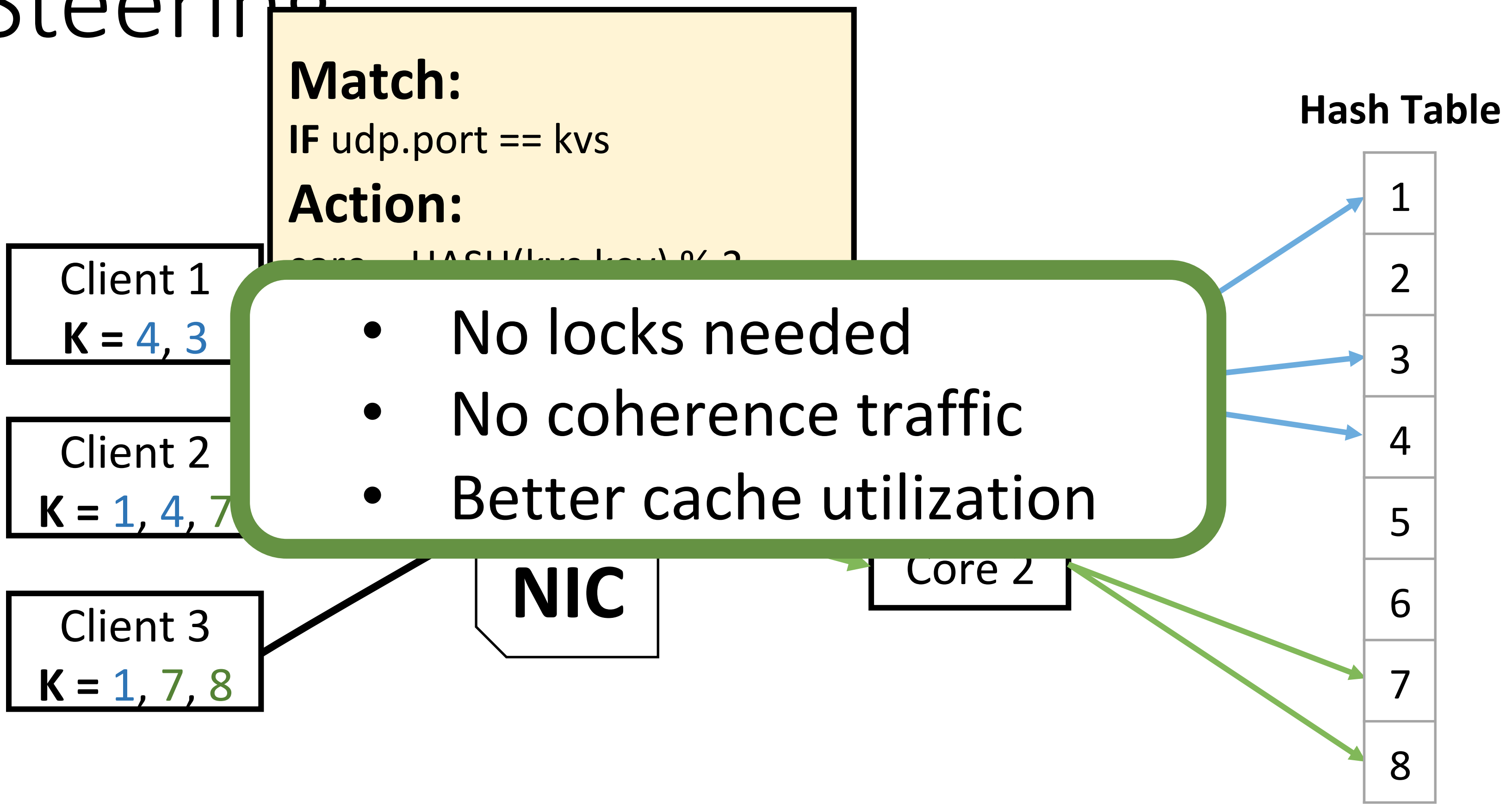
What about the CPU overhead?



Example: Key-Value Store



Optimizing Reads: Key-based Steering



- No locks needed
- No coherence traffic
- Better cache utilization

Where is hardware going?

The Example of Bitcoin

Bitcoin is a protocol for maintaining a byzantine fault tolerant public ledger

- Cryptographically signed ledger of of a sequence of transfers of digital currency, but could be anything

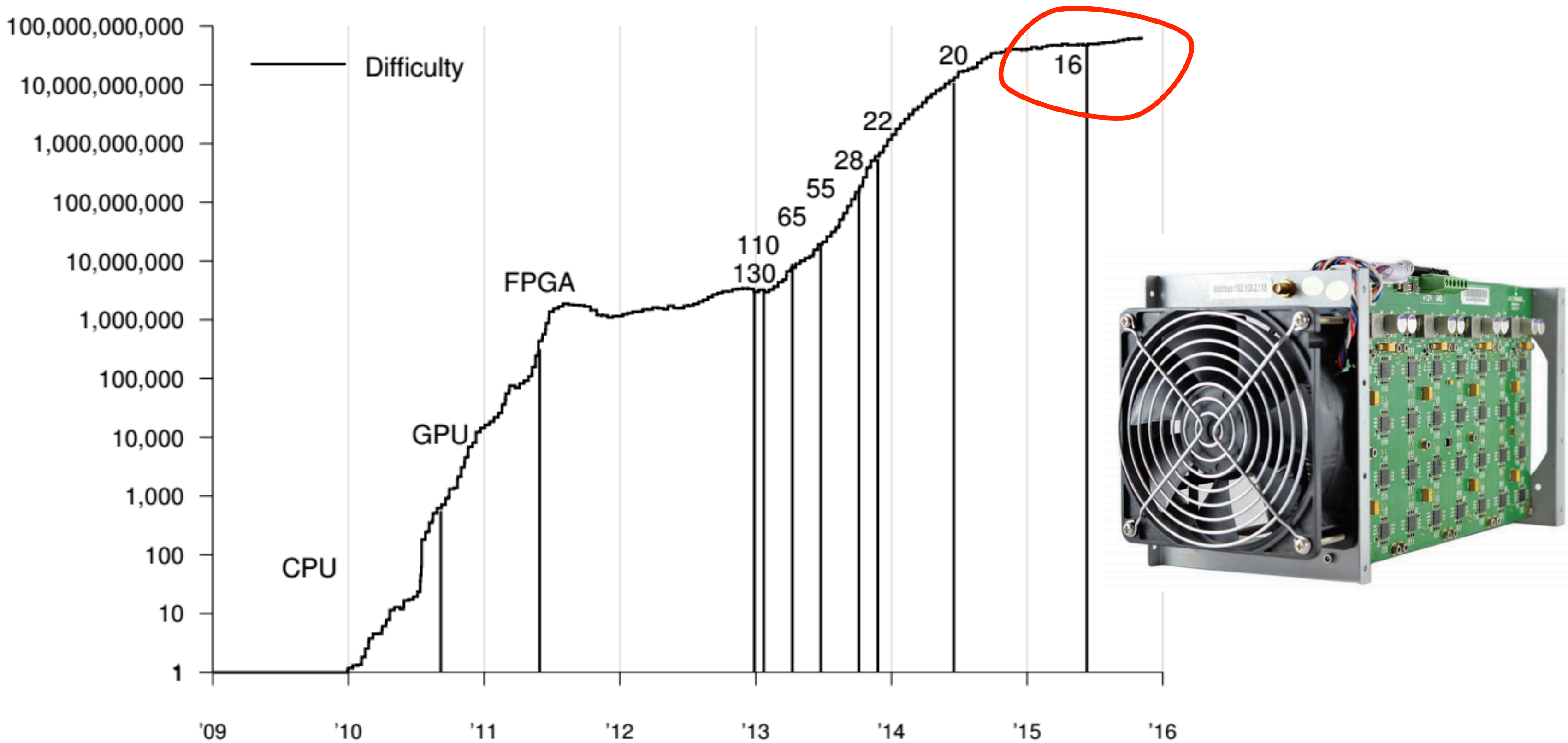
Provides an incentive to participate in the fault tolerant protocol

- Proof of work: Solve a cryptographic puzzle, get a reward
- Hard to monopolize

Extremely low bandwidth: a few transactions/second

- Energy cost roughly the same as Ireland

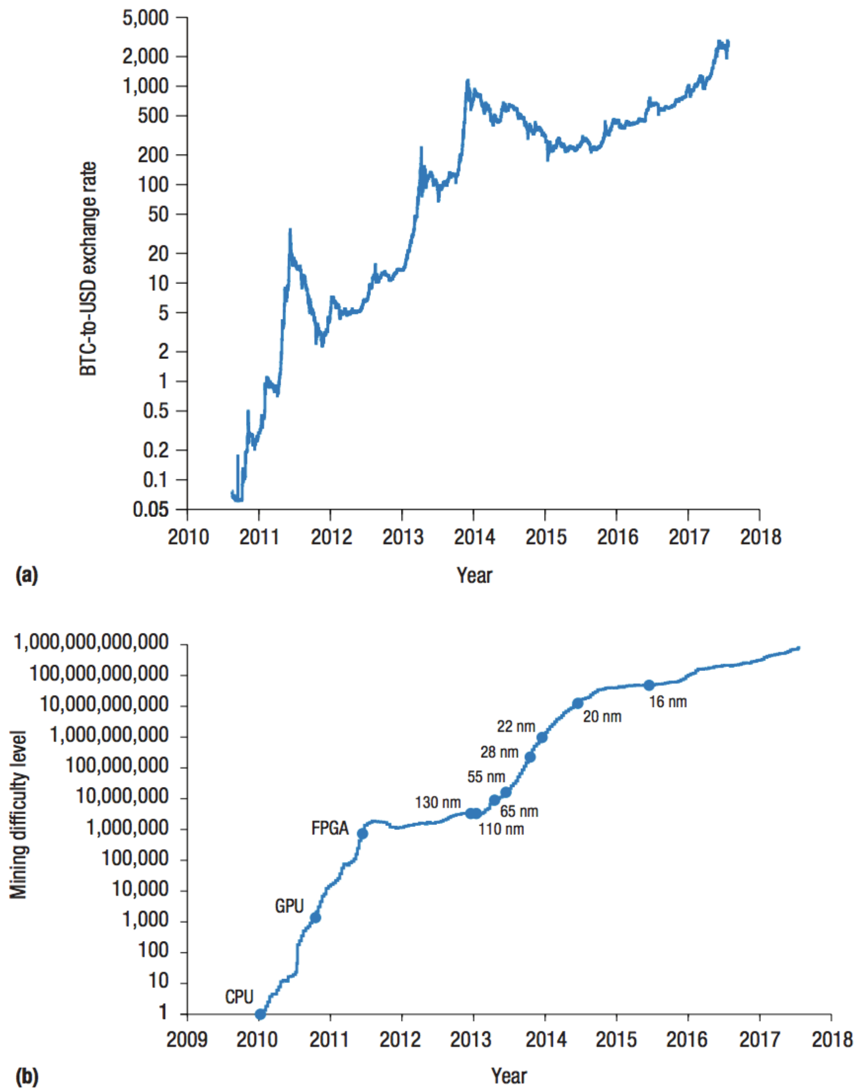
Bitcoin Hardware Progression



CPU -> GPU -> FPGA -> Low tech VLSI -> High tech VLSI

Market price: 1 petahash of SHA-256 = $0.01

Price and Difficulty



FPGAs, GPU, >55nm out of business