# Systems Research Overview

Francesco Bronzino

ArchiSys L3

ENS DE LYON

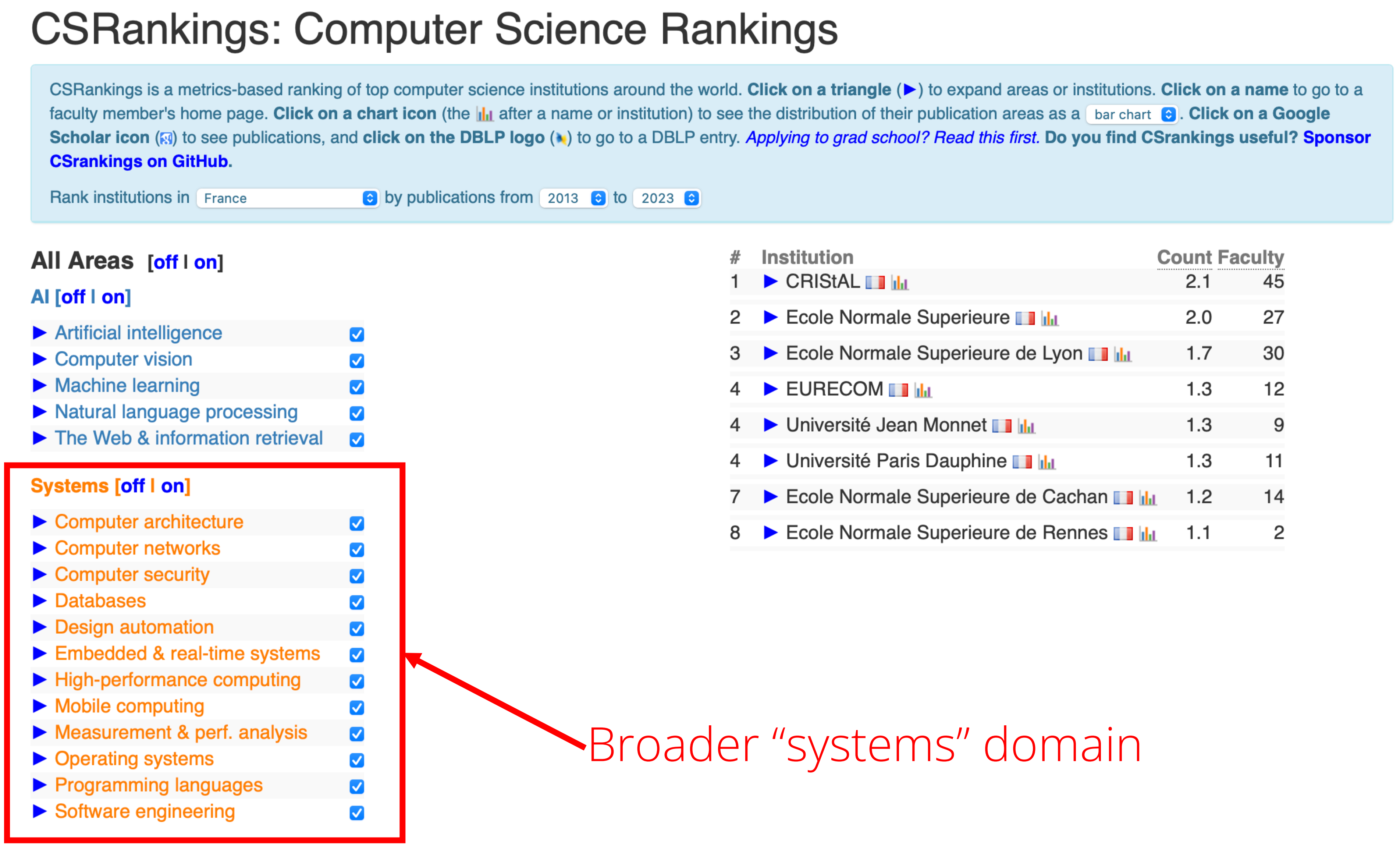
## Program of the day

- Systems research
  - Where to find it
  - What are the hot topics
  - Overview of recent work

## Where to find it

- First rule of the game: most Computer Science lives and die its by conferences
- Pure systems conferences:
  - Usenix OSDI, ACM SOSP TOP
  - Usenix ATC, Eurosys, Usenix FAST
- At the intersection of systems and …
  - Networking: Usenix NSDI, ACM SIGCOMM, ACM CoNEXT
  - Performance evaluation: ACM Sigmetrics
  - Security: Usenix Security, ACM CCS
  - Machine Learning: MLSys

## When in doubt



CSRankings: Computer Science Rankings

CSRankings is a metrics-based ranking of top computer science institutions around the world. **Click on a triangle** (▶) to expand areas or institutions. **Click on a name** to go to a faculty member's home page. **Click on a chart icon** (the 📊 after a name or institution) to see the distribution of their publication areas as a bar chart. **Click on a Google Scholar icon** to see publications, and **click on the DBLP logo** to go to a DBLP entry. *Applying to grad school? Read this first.* **Do you find CSrankings useful? Sponsor CSrankings on GitHub.**

Rank institutions in France by publications from 2013 to 2023

**All Areas [off | on]**

**AI [off | on]**

| | |
|---|---|
| ▶ Artificial intelligence | ✓ |
| ▶ Computer vision | ✓ |
| ▶ Machine learning | ✓ |
| ▶ Natural language processing | ✓ |
| ▶ The Web & information retrieval | ✓ |

**Systems [off | on]**

| | |
|---|---|
| ▶ Computer architecture | ✓ |
| ▶ Computer networks | ✓ |
| ▶ Computer security | ✓ |
| ▶ Databases | ✓ |
| ▶ Design automation | ✓ |
| ▶ Embedded & real-time systems | ✓ |
| ▶ High-performance computing | ✓ |
| ▶ Mobile computing | ✓ |
| ▶ Measurement & perf. analysis | ✓ |
| ▶ Operating systems | ✓ |
| ▶ Programming languages | ✓ |
| ▶ Software engineering | ✓ |

| # | Institution | Count | Faculty |
|---|---|---|---|
| 1 | ▶ CRIStAL | 2.1 | 45 |
| 2 | ▶ Ecole Normale Superieure | 2.0 | 27 |
| 3 | ▶ Ecole Normale Superieure de Lyon | 1.7 | 30 |
| 4 | ▶ EURECOM | 1.3 | 12 |
| 4 | ▶ Université Jean Monnet | 1.3 | 9 |
| 4 | ▶ Université Paris Dauphine | 1.3 | 11 |
| 7 | ▶ Ecole Normale Superieure de Cachan | 1.2 | 14 |
| 8 | ▶ Ecole Normale Superieure de Rennes | 1.1 | 2 |

Broader "systems" domain

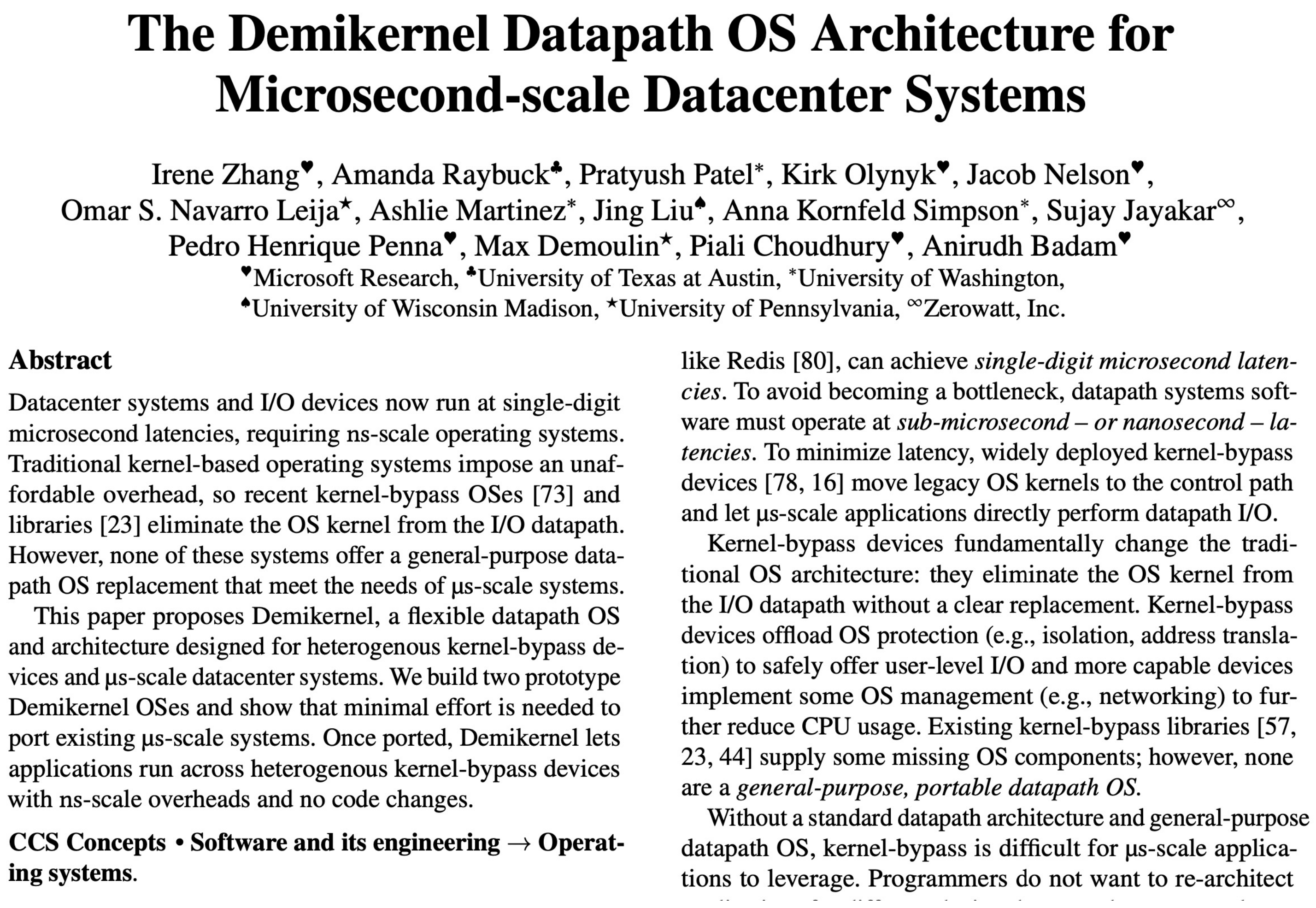
# USENIX Symposium on Operating Systems Design and Implementation 2022 Sessions

- Distributed Storage and Far Memory
- Bugs
- Persistent Memory
- Machine Learning 1
- Potpourri
- Storage
- Formal Verification
- Machine Learning 2
- Isolation and OS Services
- Security and Private Messaging
- Managed Languages
- Recommenders and Pattern Mining

## Some considerations on systems research

- Driven by real world problems
- Modeling / theory helps, but requires understing how systems work
- Most often requires a real implementation
  - Simulation at the very mininum
- Ultimately, it can be painful…
- But fun!
- Not a lot of systems research in France ☹
  - 2 articles in 2022
  - Not better if we include networked systems into the picture

# Specialized kernels



## The Demikernel Datapath OS Architecture for Microsecond-scale Datacenter Systems

Irene Zhang[♥], Amanda Raybuck[♣], Pratyush Patel[*], Kirk Olynyk[♥], Jacob Nelson[♥], Omar S. Navarro Leija[★], Ashlie Martinez[*], Jing Liu[♠], Anna Kornfeld Simpson[*], Sujay Jayakar[∞], Pedro Henrique Penna[♥], Max Demoulin[★], Piali Choudhury[♥], Anirudh Badam[♥]
[♥]Microsoft Research, [♣]University of Texas at Austin, [*]University of Washington, [♠]University of Wisconsin Madison, [★]University of Pennsylvania, [∞]Zerowatt, Inc.

### Abstract

Datacenter systems and I/O devices now run at single-digit microsecond latencies, requiring ns-scale operating systems. Traditional kernel-based operating systems impose an unaffordable overhead, so recent kernel-bypass OSes [73] and libraries [23] eliminate the OS kernel from the I/O datapath. However, none of these systems offer a general-purpose datapath OS replacement that meet the needs of µs-scale systems.

This paper proposes Demikernel, a flexible datapath OS and architecture designed for heterogenous kernel-bypass devices and µs-scale datacenter systems. We build two prototype Demikernel OSes and show that minimal effort is needed to port existing µs-scale systems. Once ported, Demikernel lets applications run across heterogenous kernel-bypass devices with ns-scale overheads and no code changes.

**CCS Concepts • Software and its engineering → Operating systems**.

like Redis [80], can achieve *single-digit microsecond latencies*. To avoid becoming a bottleneck, datapath systems software must operate at *sub-microsecond – or nanosecond – latencies*. To minimize latency, widely deployed kernel-bypass devices [78, 16] move legacy OS kernels to the control path and let µs-scale applications directly perform datapath I/O.

Kernel-bypass devices fundamentally change the traditional OS architecture: they eliminate the OS kernel from the I/O datapath without a clear replacement. Kernel-bypass devices offload OS protection (e.g., isolation, address translation) to safely offer user-level I/O and more capable devices implement some OS management (e.g., networking) to further reduce CPU usage. Existing kernel-bypass libraries [57, 23, 44] supply some missing OS components; however, none are a *general-purpose, portable datapath OS*.

Without a standard datapath architecture and general-purpose datapath OS, kernel-bypass is difficult for µs-scale applications to leverage. Programmers do not want to re-architect

# Modern I/O requires bypassing the kernel

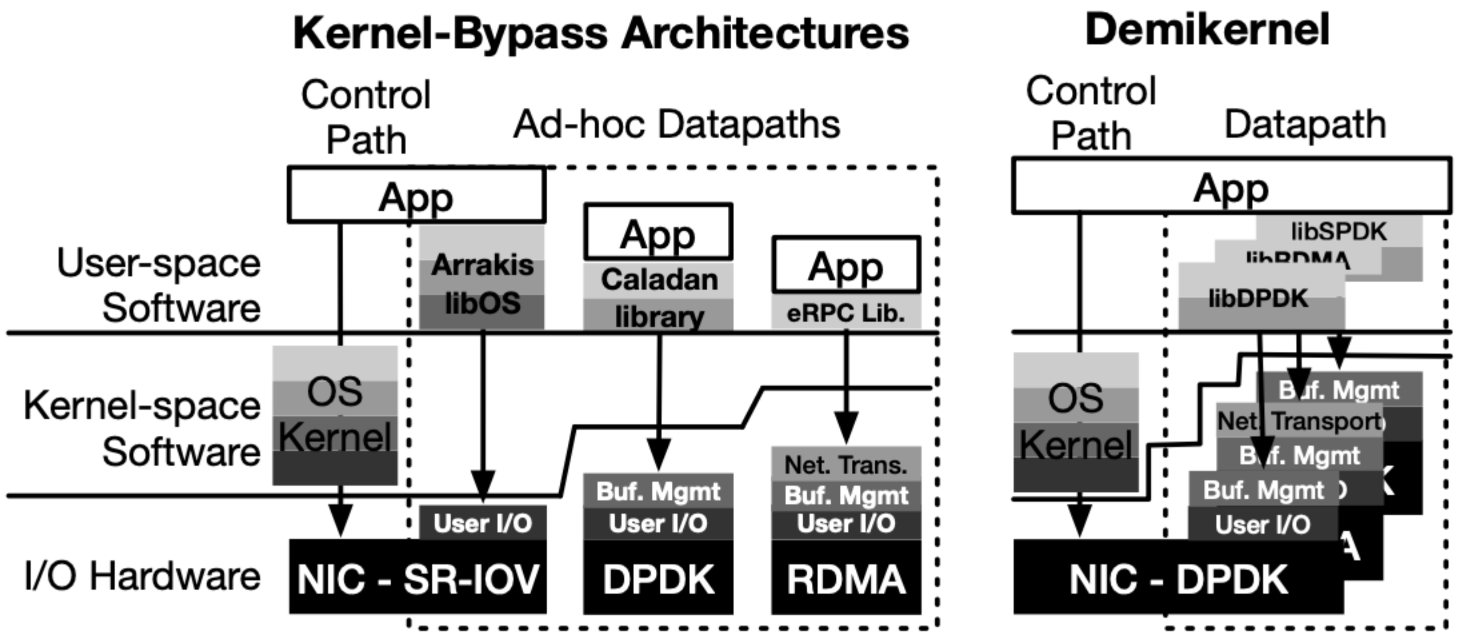


**Figure 1.** *Example kernel-bypass architectures.* Unlike the Demikernel architecture (right), Arrakis [73], Caladan [23] and eRPC [8]'s architectures do not flexibly support heterogenous devices.

## Requirements

1. Support Heterogenous OS Offloads
2. Coordinate Zero-Copy Memory Access
3. Multiplex and Schedule the CPU at μs-scale

# No added programming complexity

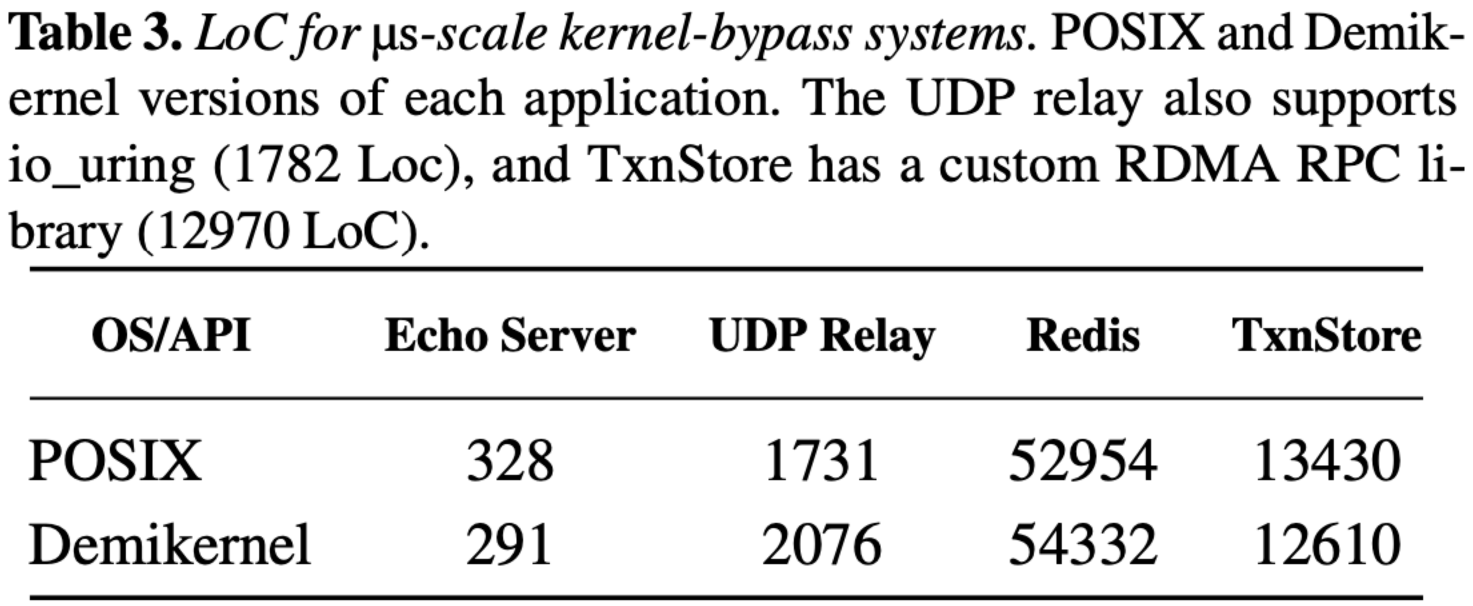


**Table 3.** *LoC for µs-scale kernel-bypass systems*. POSIX and Demikernel versions of each application. The UDP relay also supports io_uring (1782 Loc), and TxnStore has a custom RDMA RPC library (12970 LoC).

| OS/API | Echo Server | UDP Relay | Redis | TxnStore |
|---|---|---|---|---|
| POSIX | 328 | 1731 | 52954 | 13430 |
| Demikernel | 291 | 2076 | 54332 | 12610 |

# Performance

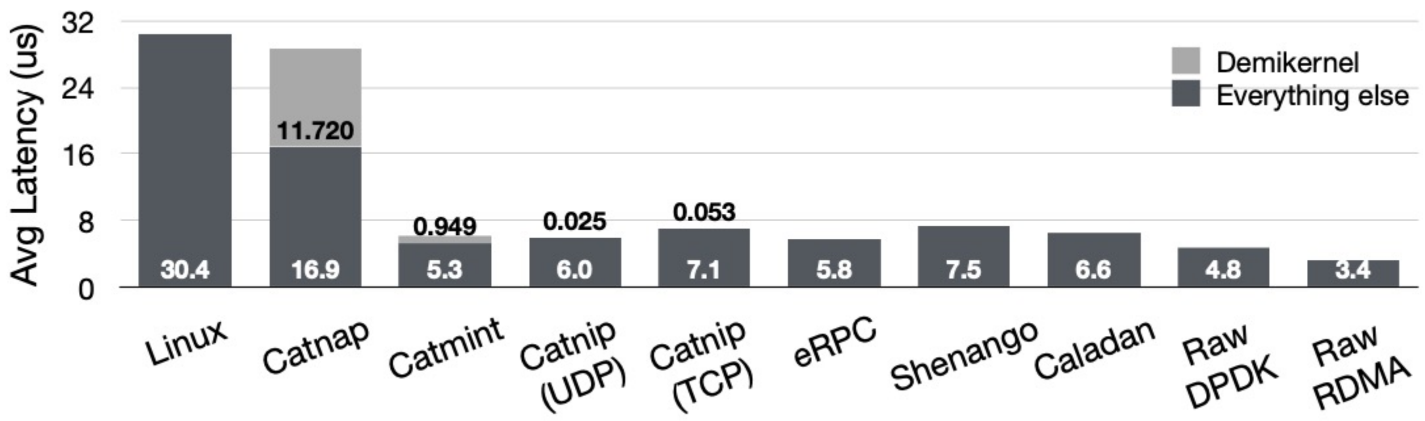


**Figure 5.** *Echo latencies on Linux (64*B*).* The upper number reports total time spent in Demikernel for 4 I/O operations: client and server send and receive; the lower ones show network and other latency; their sum is the total RTT on Demikernel. Demikernel achieves ns-scale overheads per I/O and has latencies close to those of eRPC, Shenango and Caladan, while supporting a greater range of devices and network protocols. We perform 1 million echos over 5 runs, the variance between runs was below 1%.

# Performance 2

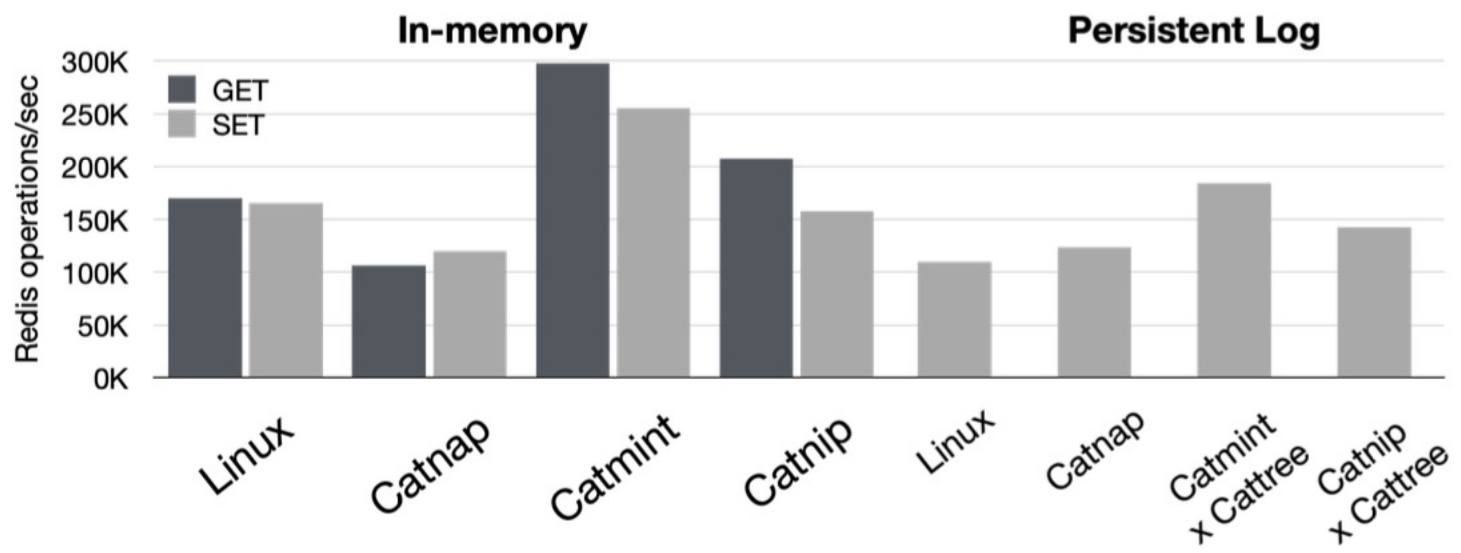
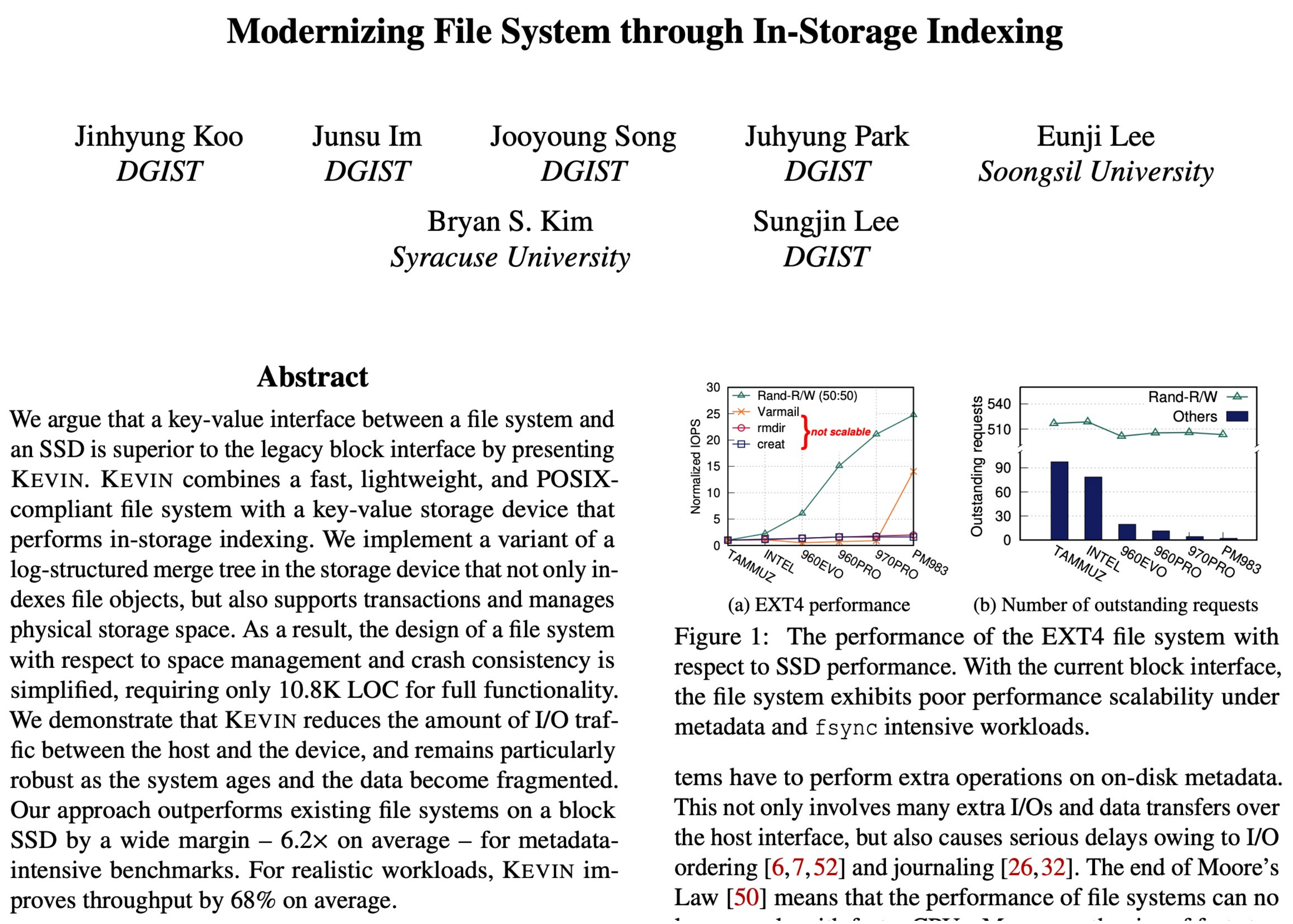


**Figure 11.** *Redis benchmark throughput in-memory and on-disk.* We use 64B values and 1 million keys. We perform separate runs for each operation with 500,000 accesses repeated 5 times. Demikernel improves Redis performance and lets it maintain that performance with synchronous writes to disk.

# Storage design



## Modernizing File System through In-Storage Indexing

Jinhyung Koo, DGIST
Junsu Im, DGIST
Jooyoung Song, DGIST
Juhyung Park, DGIST
Eunji Lee, Soongsil University
Bryan S. Kim, Syracuse University
Sungjin Lee, DGIST

**Abstract**

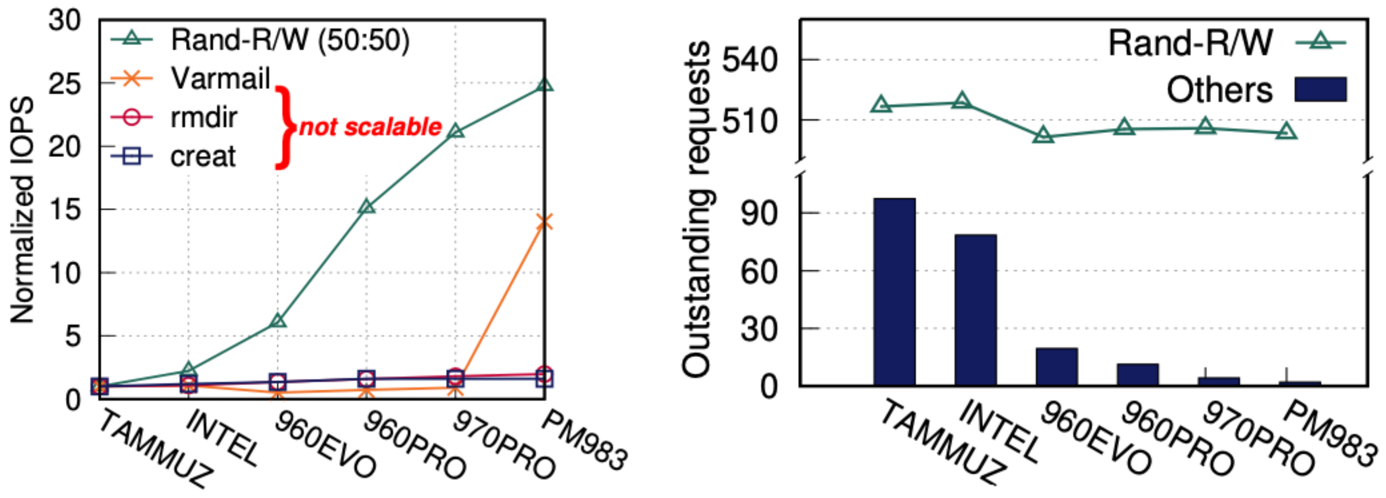
We argue that a key-value interface between a file system and an SSD is superior to the legacy block interface by presenting KEVIN. KEVIN combines a fast, lightweight, and POSIX-compliant file system with a key-value storage device that performs in-storage indexing. We implement a variant of a log-structured merge tree in the storage device that not only indexes file objects, but also supports transactions and manages physical storage space. As a result, the design of a file system with respect to space management and crash consistency is simplified, requiring only 10.8K LOC for full functionality. We demonstrate that KEVIN reduces the amount of I/O traffic between the host and the device, and remains particularly robust as the system ages and the data become fragmented. Our approach outperforms existing file systems on a block SSD by a wide margin – 6.2× on average – for metadata-intensive benchmarks. For realistic workloads, KEVIN improves throughput by 68% on average.

(a) EXT4 performance (b) Number of outstanding requests

Figure 1: The performance of the EXT4 file system with respect to SSD performance. With the current block interface, the file system exhibits poor performance scalability under metadata and fsync intensive workloads.

tems have to perform extra operations on on-disk metadata. This not only involves many extra I/Os and data transfers over the host interface, but also causes serious delays owing to I/O ordering [6, 7, 52] and journaling [26, 32]. The end of Moore's Law [50] means that the performance of file systems can no

# The metadata overhead



(a) EXT4 performance (b) Number of outstanding requests

Figure 1: The performance of the EXT4 file system with respect to SSD performance. With the current block interface, the file system exhibits poor performance scalability under metadata and `fsync` intensive workloads.

# Key value storage

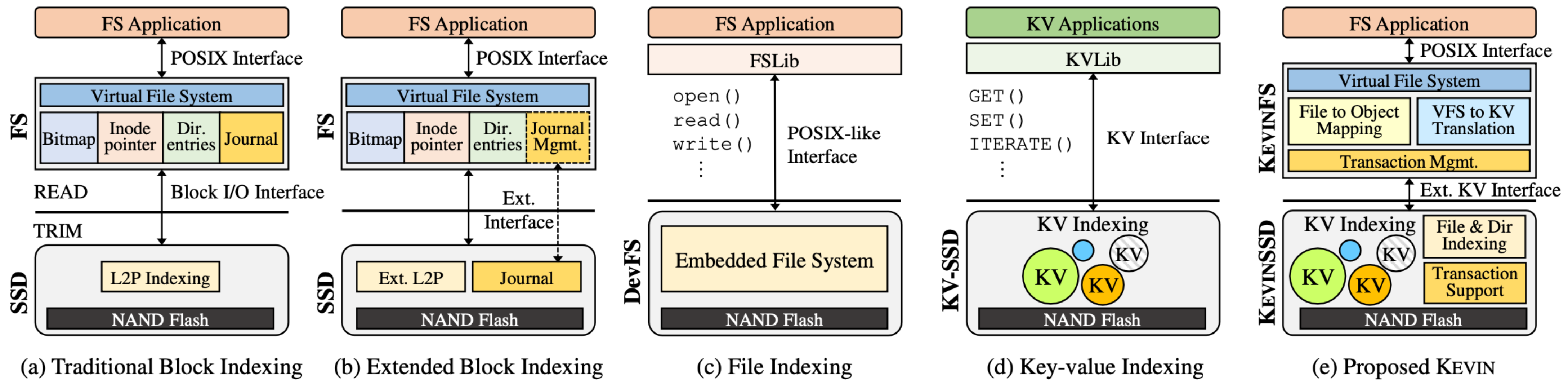
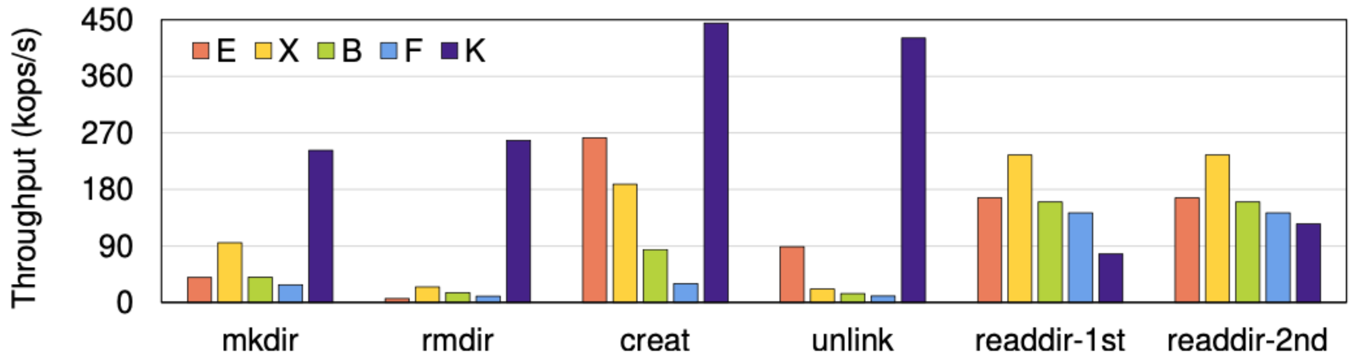


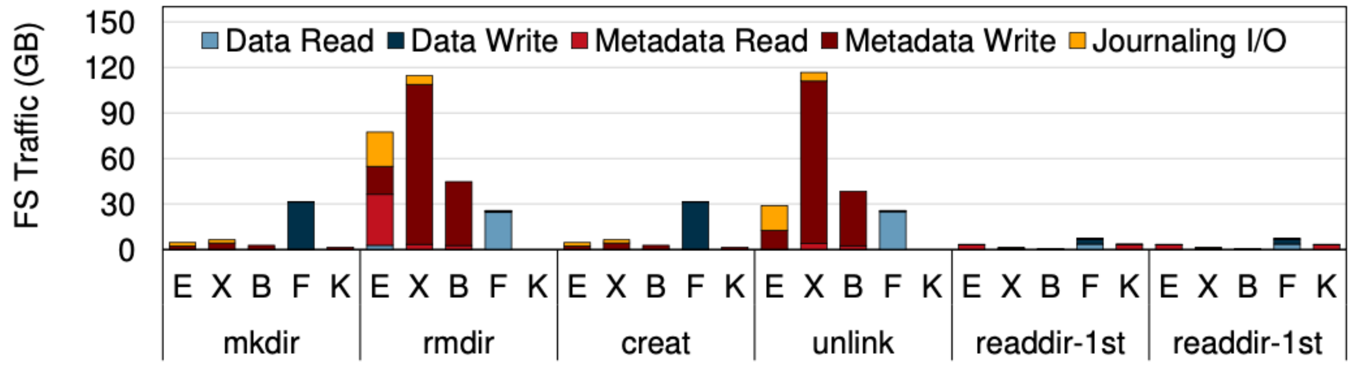
Figure 2: Categories of in-storage indexing technologies

# Performance

EXT4, XFS, BTRFS, F2FS, and KEVIN are abbreviated as 'E', 'X', 'B', 'F'. and 'K', respectively.



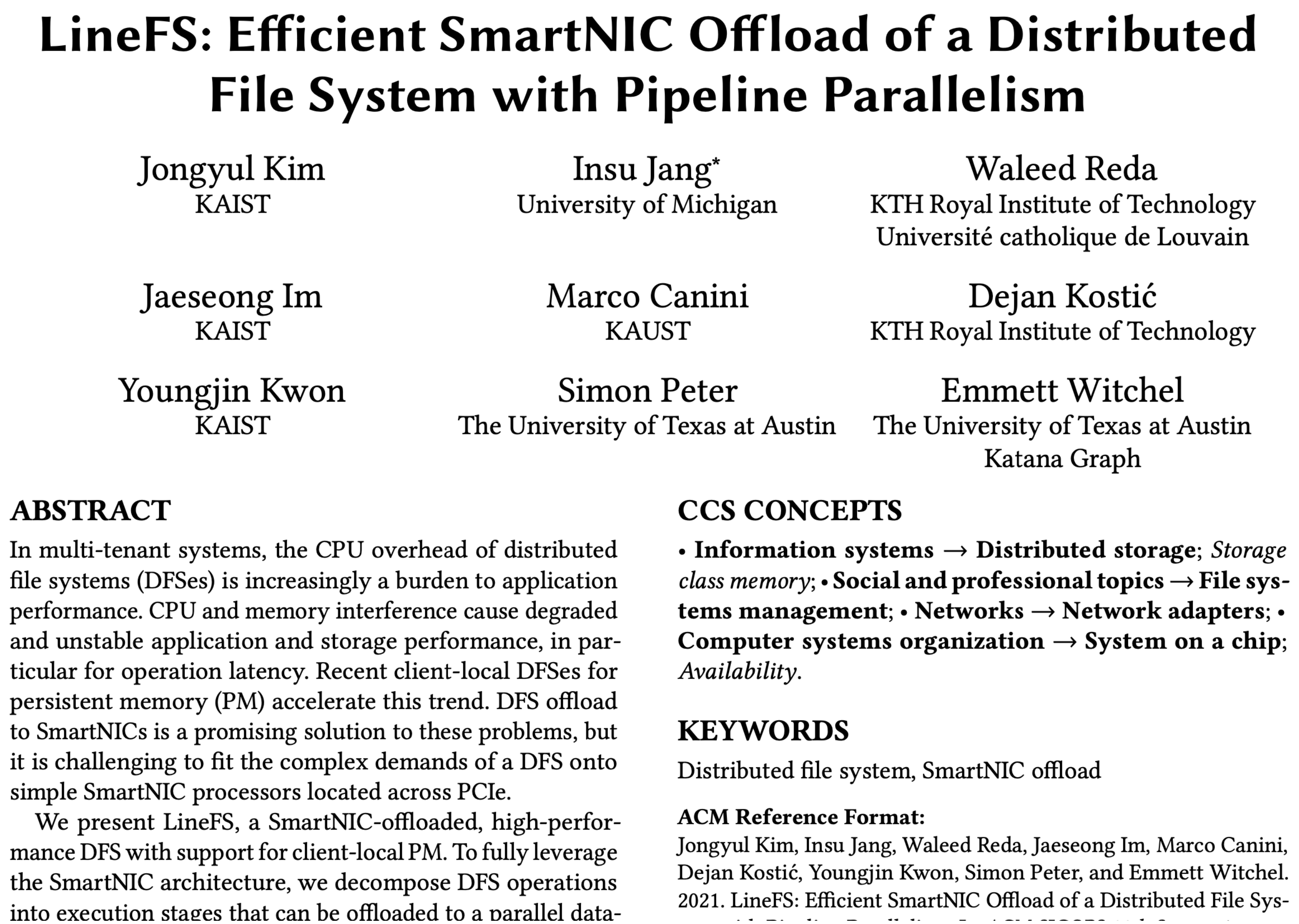
(a) Throughput



(b) File system traffic

Figure 9: Metadata intensive workloads

# Specialized hardware



## LineFS: Efficient SmartNIC Offload of a Distributed File System with Pipeline Parallelism

Jongyul Kim
KAIST

Insu Jang*
University of Michigan

Waleed Reda
KTH Royal Institute of Technology
Université catholique de Louvain

Jaeseong Im
KAIST

Marco Canini
KAUST

Dejan Kostić
KTH Royal Institute of Technology

Youngjin Kwon
KAIST

Simon Peter
The University of Texas at Austin

Emmett Witchel
The University of Texas at Austin
Katana Graph

**ABSTRACT**

In multi-tenant systems, the CPU overhead of distributed file systems (DFSes) is increasingly a burden to application performance. CPU and memory interference cause degraded and unstable application and storage performance, in particular for operation latency. Recent client-local DFSes for persistent memory (PM) accelerate this trend. DFS offload to SmartNICs is a promising solution to these problems, but it is challenging to fit the complex demands of a DFS onto simple SmartNIC processors located across PCIe.

We present LineFS, a SmartNIC-offloaded, high-performance DFS with support for client-local PM. To fully leverage the SmartNIC architecture, we decompose DFS operations into execution stages that can be offloaded to a parallel data-

**CCS CONCEPTS**

• **Information systems** → **Distributed storage**; *Storage class memory*; • **Social and professional topics** → **File systems management**; • **Networks** → **Network adapters**; • **Computer systems organization** → **System on a chip**; *Availability*.

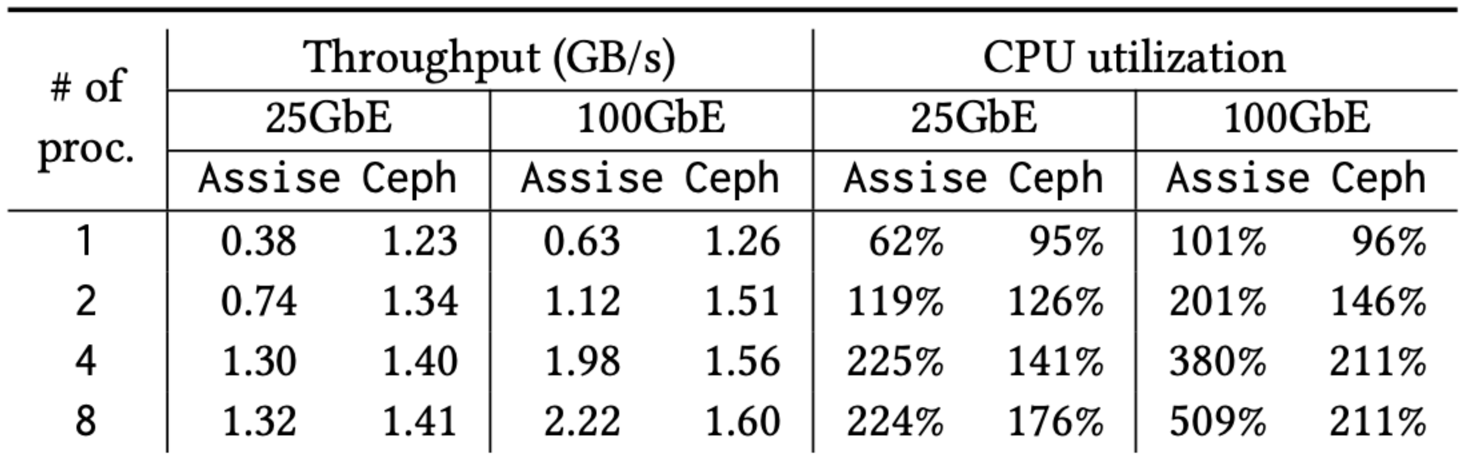
**KEYWORDS**

Distributed file system, SmartNIC offload

**ACM Reference Format:**

Jongyul Kim, Insu Jang, Waleed Reda, Jaeseong Im, Marco Canini, Dejan Kostić, Youngjin Kwon, Simon Peter, and Emmett Witchel. 2021. LineFS: Efficient SmartNIC Offload of a Distributed File Sys-

# DFS uses precious CPU resources



| # of proc. | Throughput (GB/s) | | | | CPU utilization | | | |
|---|---|---|---|---|---|---|---|---|
| | 25GbE | | 100GbE | | 25GbE | | 100GbE | |
| | Assise | Ceph | Assise | Ceph | Assise | Ceph | Assise | Ceph |
| 1 | 0.38 | 1.23 | 0.63 | 1.26 | 62% | 95% | 101% | 96% |
| 2 | 0.74 | 1.34 | 1.12 | 1.51 | 119% | 126% | 201% | 146% |
| 4 | 1.30 | 1.40 | 1.98 | 1.56 | 225% | 141% | 380% | 211% |
| 8 | 1.32 | 1.41 | 2.22 | 1.60 | 224% | 176% | 509% | 211% |

**Table 1: CPU utilization of Assise and Ceph for different numbers of benchmark processes and network speeds. 100% = 1 core.**

# Exploit tasks parallelisms

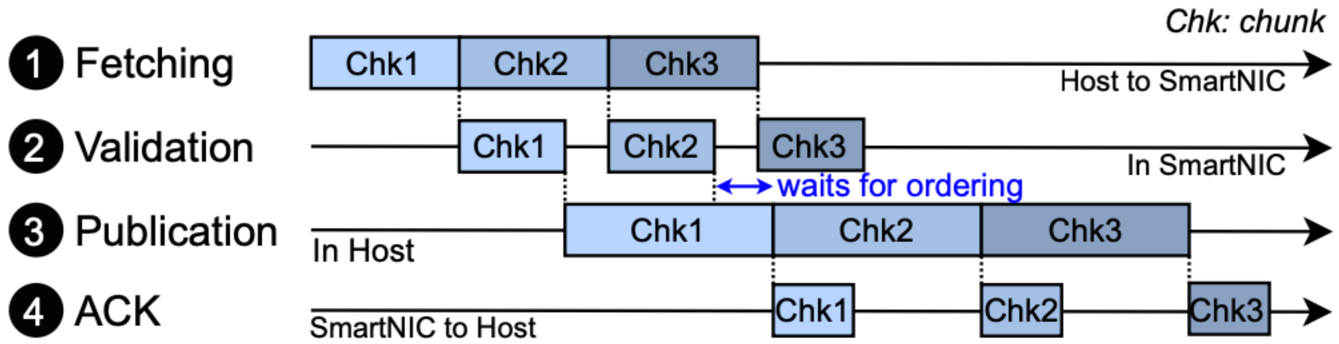


**Figure 2: Publishing with pipeline parallelism.**

## Performance

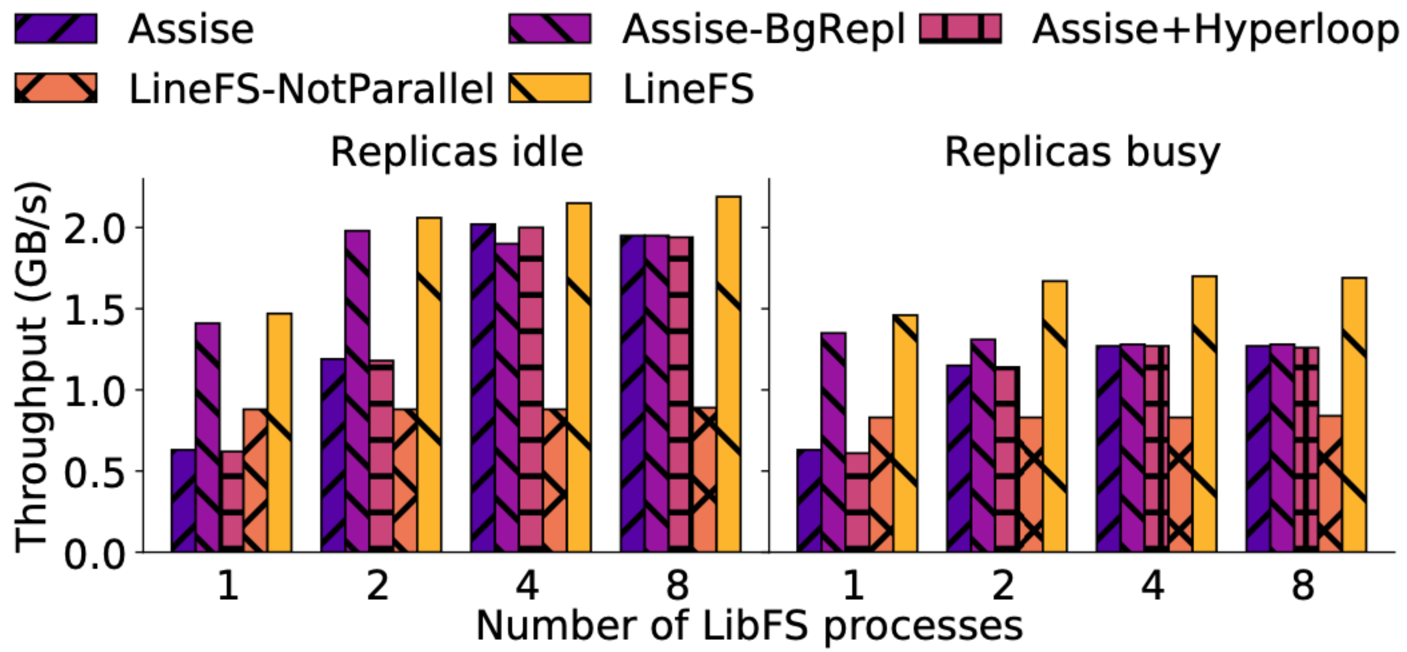


**Figure 4: Write throughput scalability when replicas are idle and busy.**

# Performance 2

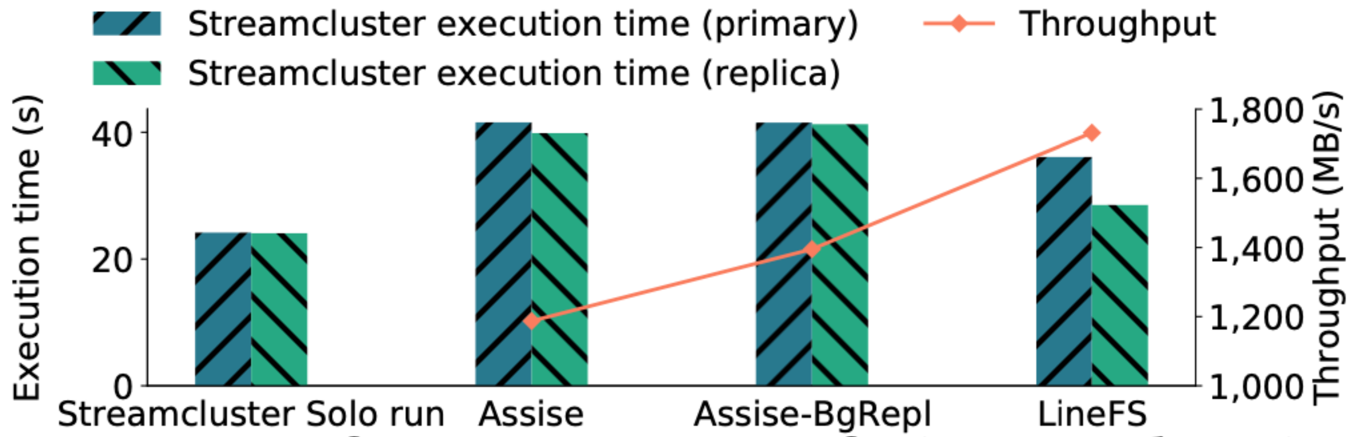
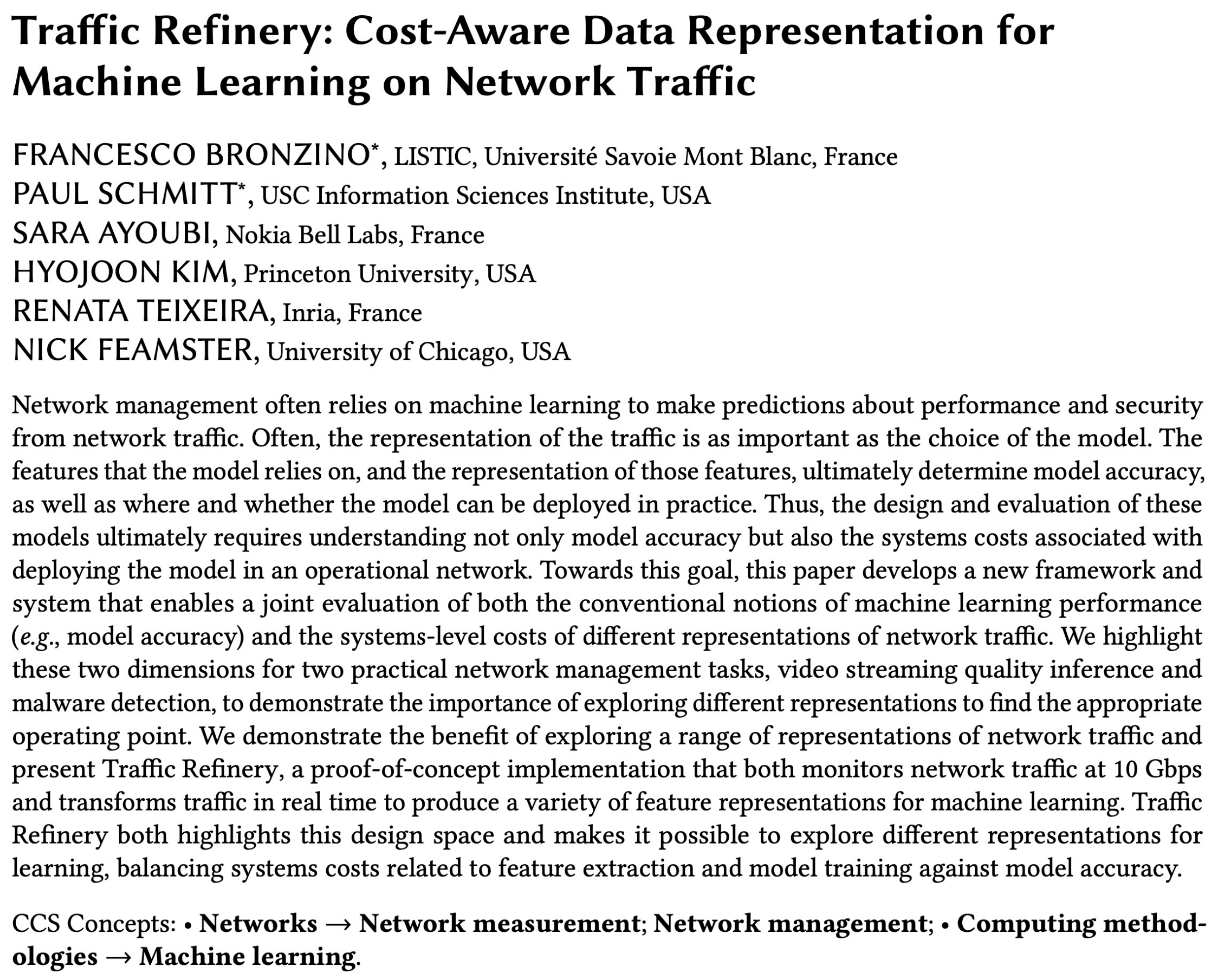


**Figure 6: Performance impact of `LineFS` and `Assise` co-execution on `streamcluster` execution time (left Y-axis) and DFS throughput (right Y-axis).**

# Network systems / Machine Learning



## Traffic Refinery: Cost-Aware Data Representation for Machine Learning on Network Traffic

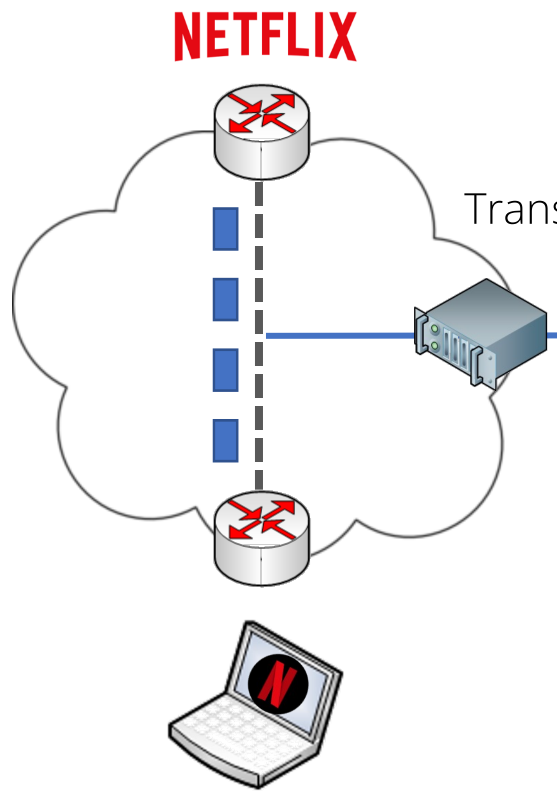
FRANCESCO BRONZINO*, LISTIC, Université Savoie Mont Blanc, France
PAUL SCHMITT*, USC Information Sciences Institute, USA
SARA AYOUBI, Nokia Bell Labs, France
HYOJOON KIM, Princeton University, USA
RENATA TEIXEIRA, Inria, France
NICK FEAMSTER, University of Chicago, USA

Network management often relies on machine learning to make predictions about performance and security from network traffic. Often, the representation of the traffic is as important as the choice of the model. The features that the model relies on, and the representation of those features, ultimately determine model accuracy, as well as where and whether the model can be deployed in practice. Thus, the design and evaluation of these models ultimately requires understanding not only model accuracy but also the systems costs associated with deploying the model in an operational network. Towards this goal, this paper develops a new framework and system that enables a joint evaluation of both the conventional notions of machine learning performance (*e.g.*, model accuracy) and the systems-level costs of different representations of network traffic. We highlight these two dimensions for two practical network management tasks, video streaming quality inference and malware detection, to demonstrate the importance of exploring different representations to find the appropriate operating point. We demonstrate the benefit of exploring a range of representations of network traffic and present Traffic Refinery, a proof-of-concept implementation that both monitors network traffic at 10 Gbps and transforms traffic in real time to produce a variety of feature representations for machine learning. Traffic Refinery both highlights this design space and makes it possible to explore different representations for learning, balancing systems costs related to feature extraction and model training against model accuracy.

CCS Concepts: • **Networks** → **Network measurement**; **Network management**; • **Computing methodologies** → **Machine learning**.

# A video quality inference use case

*Data Collection / Cleaning*



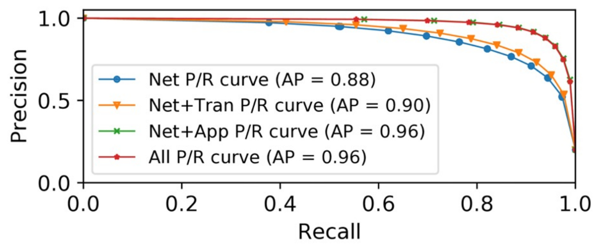
Transformations

*Feature Engineering*

| |
|---|
| **Network layer (*Net*)** |
| Throughput |
| Pkt counts |
| … |
| **Transport layer (*Tran*)** |
| Flags |
| Retransmissions |
| … |
| **Application layer (*App*)** |
| Segment sizes |
| Segment inter-arrivals |
| … |

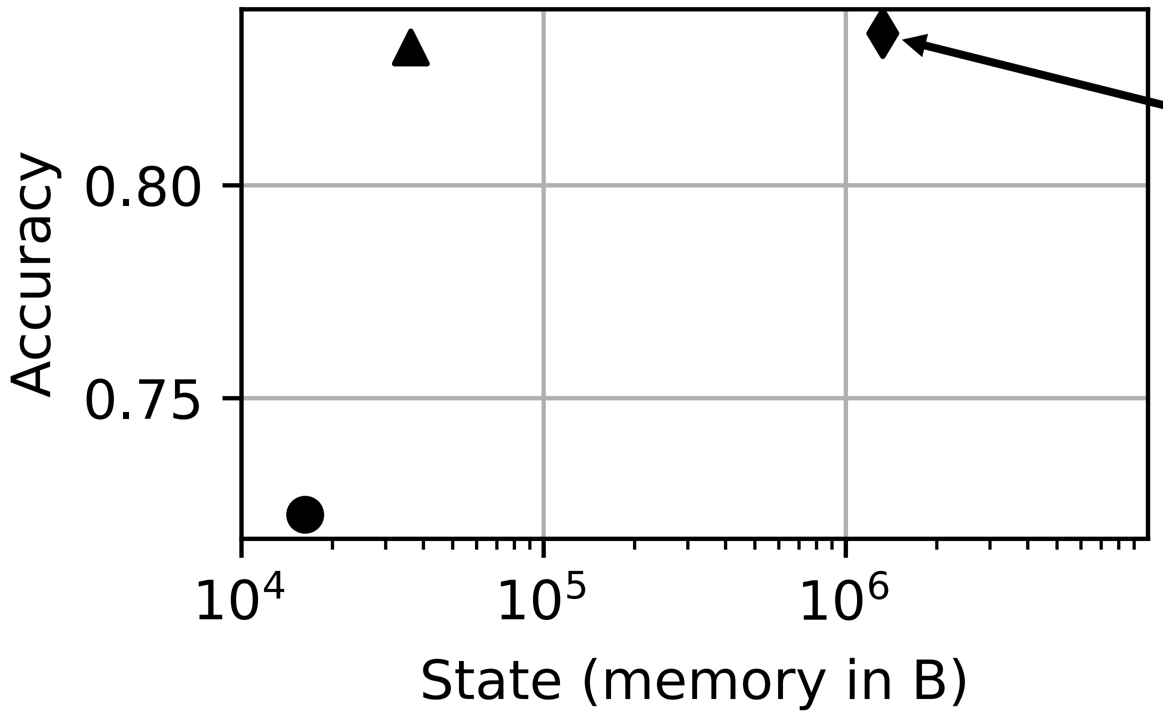
*Feature Training / Testing*

Resolution



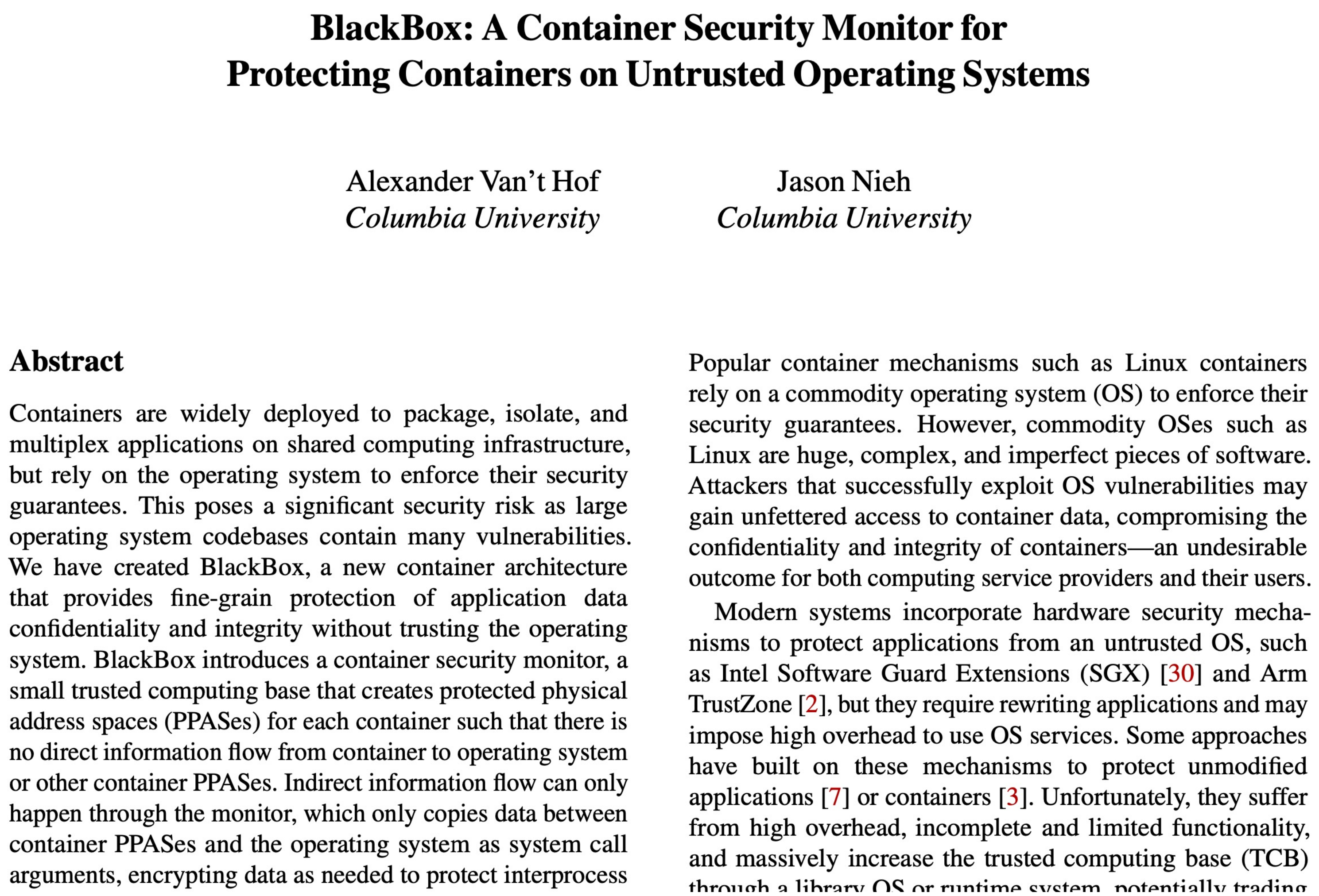
*From "Inferring Streaming Video Quality from Encrypted Traffic: Practical Models and Deployment Experience", F. Bronzino et al., in ACM Sigmetrics 2020*

# Representations impact more than just accuracy



Best accuracy, but what about cost?

# Security



## BlackBox: A Container Security Monitor for Protecting Containers on Untrusted Operating Systems

Alexander Van't Hof
*Columbia University*

Jason Nieh
*Columbia University*

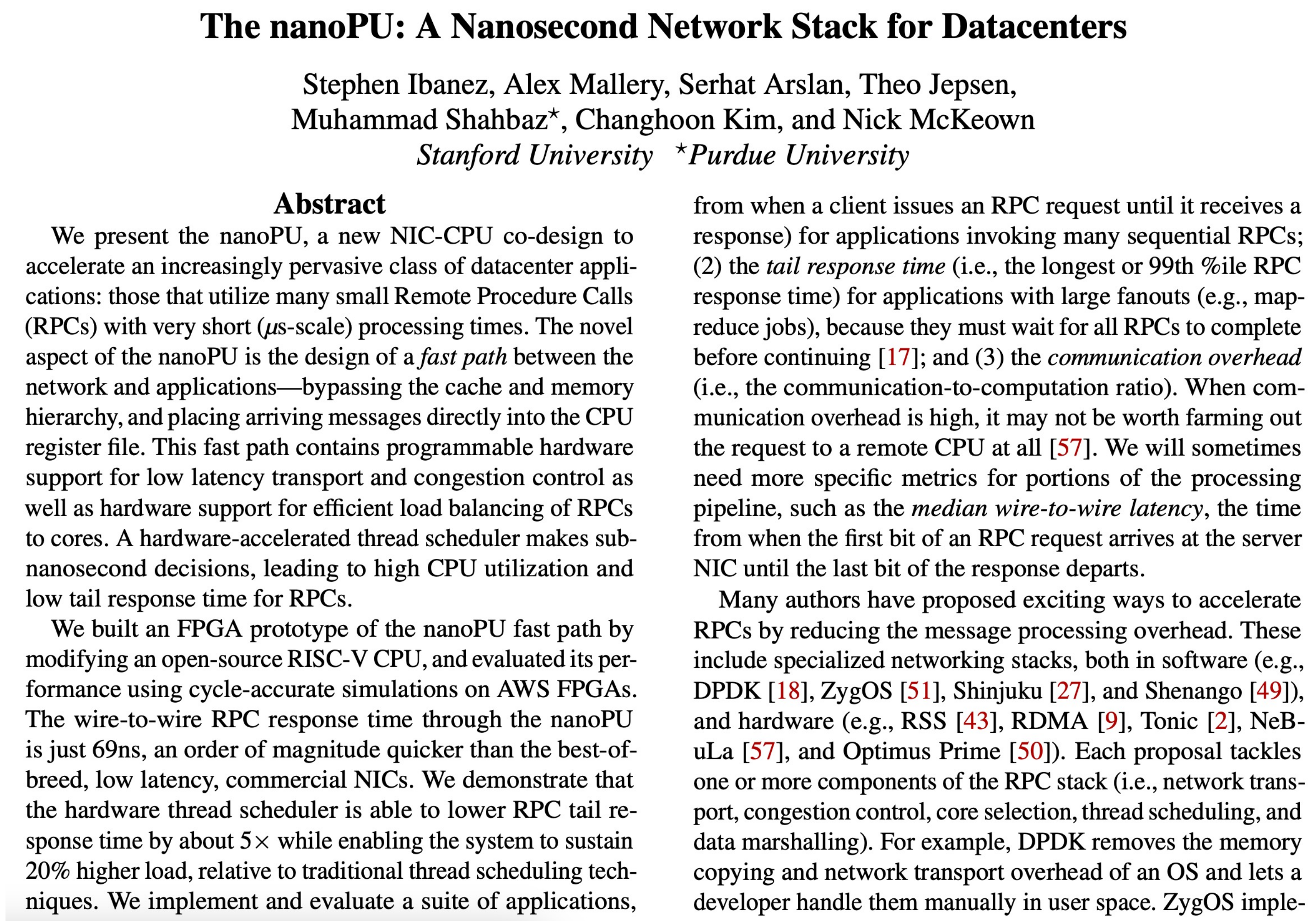
### Abstract

Containers are widely deployed to package, isolate, and multiplex applications on shared computing infrastructure, but rely on the operating system to enforce their security guarantees. This poses a significant security risk as large operating system codebases contain many vulnerabilities. We have created BlackBox, a new container architecture that provides fine-grain protection of application data confidentiality and integrity without trusting the operating system. BlackBox introduces a container security monitor, a small trusted computing base that creates protected physical address spaces (PPASes) for each container such that there is no direct information flow from container to operating system or other container PPASes. Indirect information flow can only happen through the monitor, which only copies data between container PPASes and the operating system as system call arguments, encrypting data as needed to protect interprocess

Popular container mechanisms such as Linux containers rely on a commodity operating system (OS) to enforce their security guarantees. However, commodity OSes such as Linux are huge, complex, and imperfect pieces of software. Attackers that successfully exploit OS vulnerabilities may gain unfettered access to container data, compromising the confidentiality and integrity of containers—an undesirable outcome for both computing service providers and their users.

Modern systems incorporate hardware security mechanisms to protect applications from an untrusted OS, such as Intel Software Guard Extensions (SGX) [30] and Arm TrustZone [2], but they require rewriting applications and may impose high overhead to use OS services. Some approaches have built on these mechanisms to protect unmodified applications [7] or containers [3]. Unfortunately, they suffer from high overhead, incomplete and limited functionality, and massively increase the trusted computing base (TCB) through a library OS or runtime system, potentially trading

# Network stack



## The nanoPU: A Nanosecond Network Stack for Datacenters

Stephen Ibanez, Alex Mallery, Serhat Arslan, Theo Jepsen,
Muhammad Shahbaz[⋆], Changhoon Kim, and Nick McKeown
*Stanford University* [⋆]*Purdue University*

### Abstract

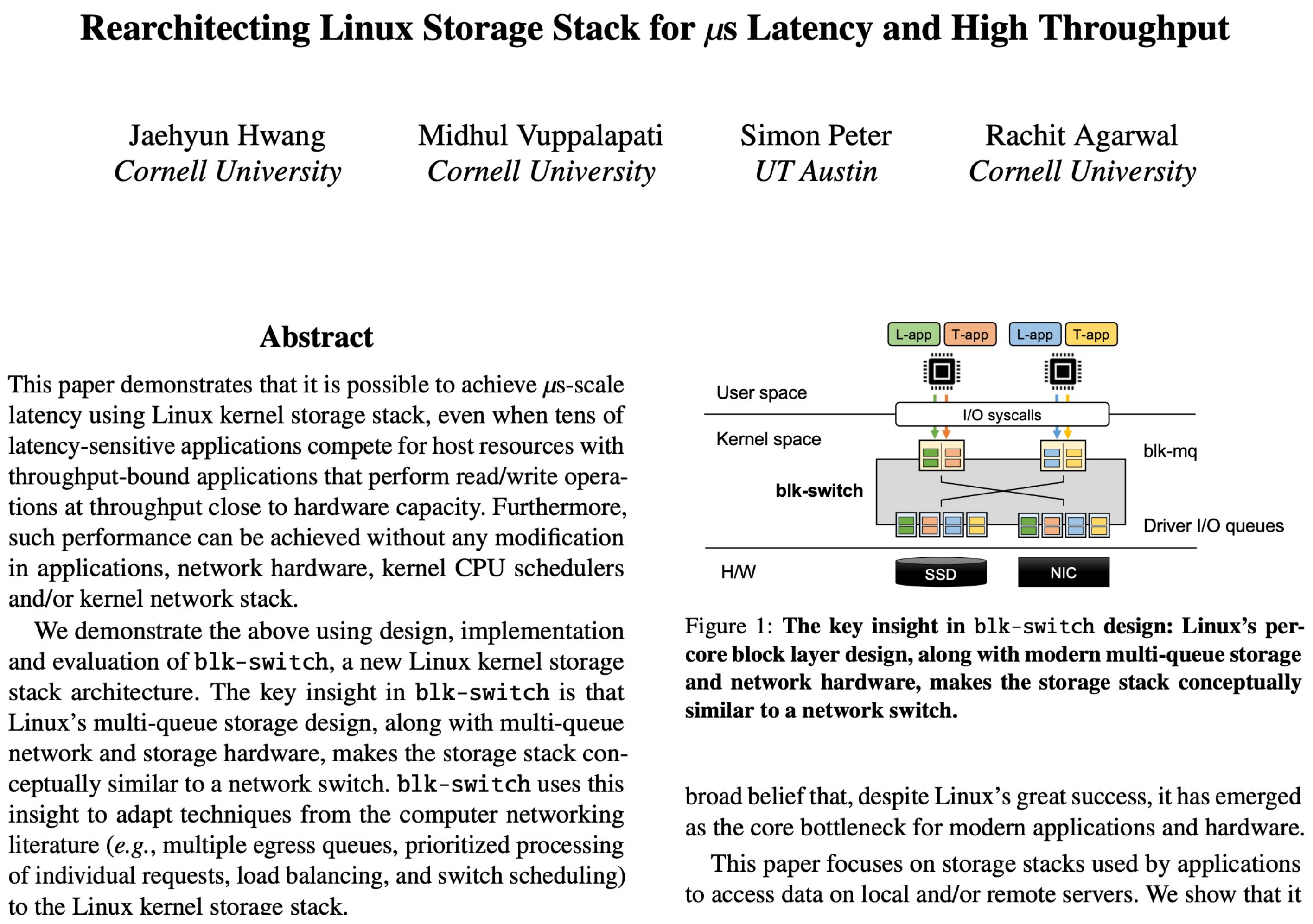
We present the nanoPU, a new NIC-CPU co-design to accelerate an increasingly pervasive class of datacenter applications: those that utilize many small Remote Procedure Calls (RPCs) with very short ($\mu$s-scale) processing times. The novel aspect of the nanoPU is the design of a *fast path* between the network and applications—bypassing the cache and memory hierarchy, and placing arriving messages directly into the CPU register file. This fast path contains programmable hardware support for low latency transport and congestion control as well as hardware support for efficient load balancing of RPCs to cores. A hardware-accelerated thread scheduler makes sub-nanosecond decisions, leading to high CPU utilization and low tail response time for RPCs.

We built an FPGA prototype of the nanoPU fast path by modifying an open-source RISC-V CPU, and evaluated its performance using cycle-accurate simulations on AWS FPGAs. The wire-to-wire RPC response time through the nanoPU is just 69ns, an order of magnitude quicker than the best-of-breed, low latency, commercial NICs. We demonstrate that the hardware thread scheduler is able to lower RPC tail response time by about 5× while enabling the system to sustain 20% higher load, relative to traditional thread scheduling techniques. We implement and evaluate a suite of applications,

from when a client issues an RPC request until it receives a response) for applications invoking many sequential RPCs; (2) the *tail response time* (i.e., the longest or 99th %ile RPC response time) for applications with large fanouts (e.g., map-reduce jobs), because they must wait for all RPCs to complete before continuing [17]; and (3) the *communication overhead* (i.e., the communication-to-computation ratio). When communication overhead is high, it may not be worth farming out the request to a remote CPU at all [57]. We will sometimes need more specific metrics for portions of the processing pipeline, such as the *median wire-to-wire latency*, the time from when the first bit of an RPC request arrives at the server NIC until the last bit of the response departs.

Many authors have proposed exciting ways to accelerate RPCs by reducing the message processing overhead. These include specialized networking stacks, both in software (e.g., DPDK [18], ZygOS [51], Shinjuku [27], and Shenango [49]), and hardware (e.g., RSS [43], RDMA [9], Tonic [2], NeBuLa [57], and Optimus Prime [50]). Each proposal tackles one or more components of the RPC stack (i.e., network transport, congestion control, core selection, thread scheduling, and data marshalling). For example, DPDK removes the memory copying and network transport overhead of an OS and lets a developer handle them manually in user space. ZygOS imple-

# Storage design 2



## Rearchitecting Linux Storage Stack for $\mu$s Latency and High Throughput

Jaehyun Hwang
*Cornell University*

Midhul Vuppalapati
*Cornell University*

Simon Peter
*UT Austin*

Rachit Agarwal
*Cornell University*

### Abstract

This paper demonstrates that it is possible to achieve $\mu$s-scale latency using Linux kernel storage stack, even when tens of latency-sensitive applications compete for host resources with throughput-bound applications that perform read/write operations at throughput close to hardware capacity. Furthermore, such performance can be achieved without any modification in applications, network hardware, kernel CPU schedulers and/or kernel network stack.

We demonstrate the above using design, implementation and evaluation of `blk-switch`, a new Linux kernel storage stack architecture. The key insight in `blk-switch` is that Linux's multi-queue storage design, along with multi-queue network and storage hardware, makes the storage stack conceptually similar to a network switch. `blk-switch` uses this insight to adapt techniques from the computer networking literature (*e.g.*, multiple egress queues, prioritized processing of individual requests, load balancing, and switch scheduling) to the Linux kernel storage stack.

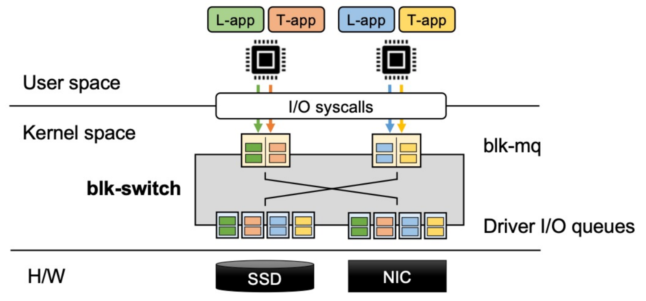
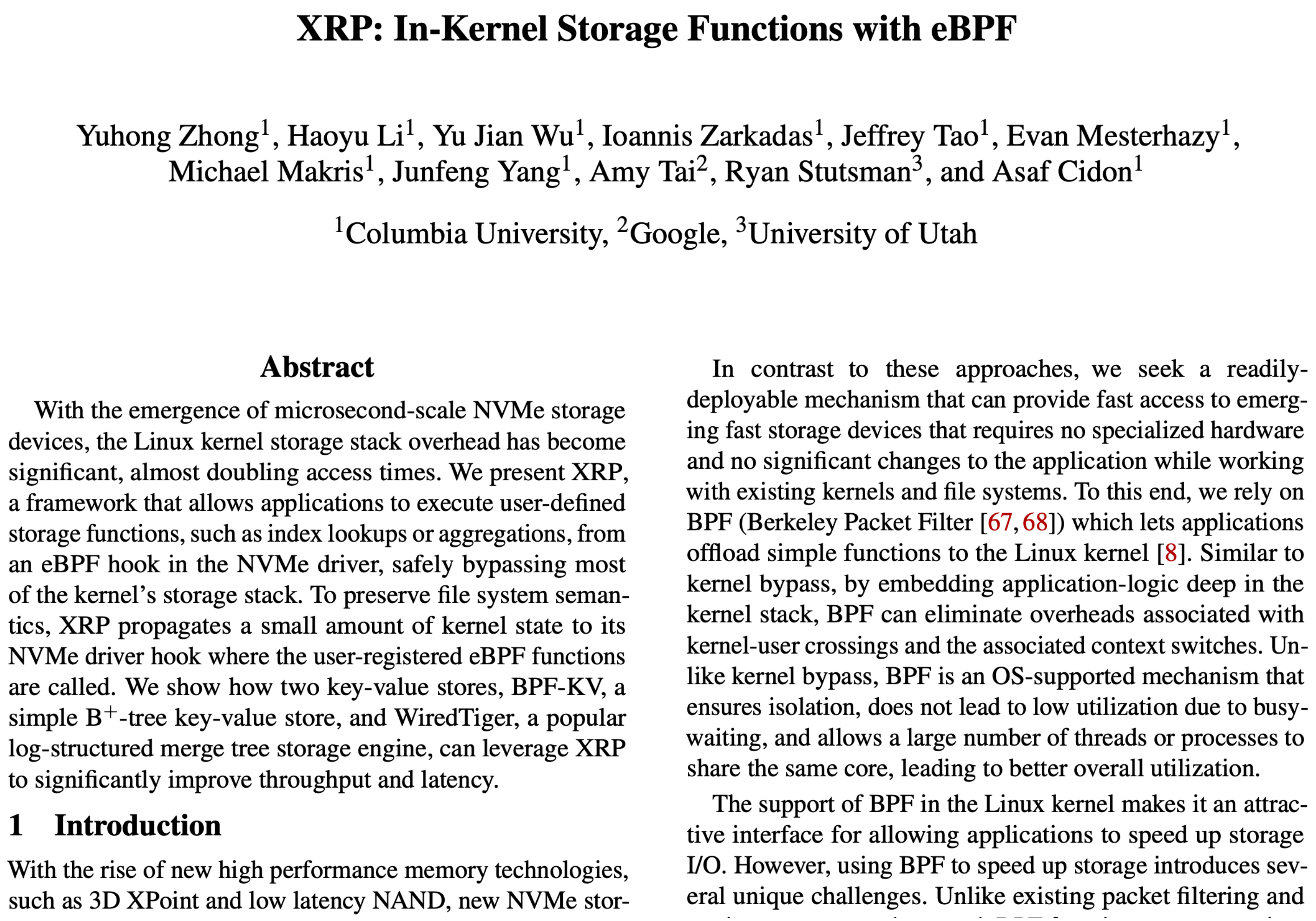


Figure 1: **The key insight in `blk-switch` design: Linux's per-core block layer design, along with modern multi-queue storage and network hardware, makes the storage stack conceptually similar to a network switch.**

broad belief that, despite Linux's great success, it has emerged as the core bottleneck for modern applications and hardware.

This paper focuses on storage stacks used by applications to access data on local and/or remote servers. We show that it

# Storage design 3



## XRP: In-Kernel Storage Functions with eBPF

Yuhong Zhong[1], Haoyu Li[1], Yu Jian Wu[1], Ioannis Zarkadas[1], Jeffrey Tao[1], Evan Mesterhazy[1], Michael Makris[1], Junfeng Yang[1], Amy Tai[2], Ryan Stutsman[3], and Asaf Cidon[1]

[1]Columbia University, [2]Google, [3]University of Utah

### Abstract

With the emergence of microsecond-scale NVMe storage devices, the Linux kernel storage stack overhead has become significant, almost doubling access times. We present XRP, a framework that allows applications to execute user-defined storage functions, such as index lookups or aggregations, from an eBPF hook in the NVMe driver, safely bypassing most of the kernel's storage stack. To preserve file system semantics, XRP propagates a small amount of kernel state to its NVMe driver hook where the user-registered eBPF functions are called. We show how two key-value stores, BPF-KV, a simple B$^+$-tree key-value store, and WiredTiger, a popular log-structured merge tree storage engine, can leverage XRP to significantly improve throughput and latency.

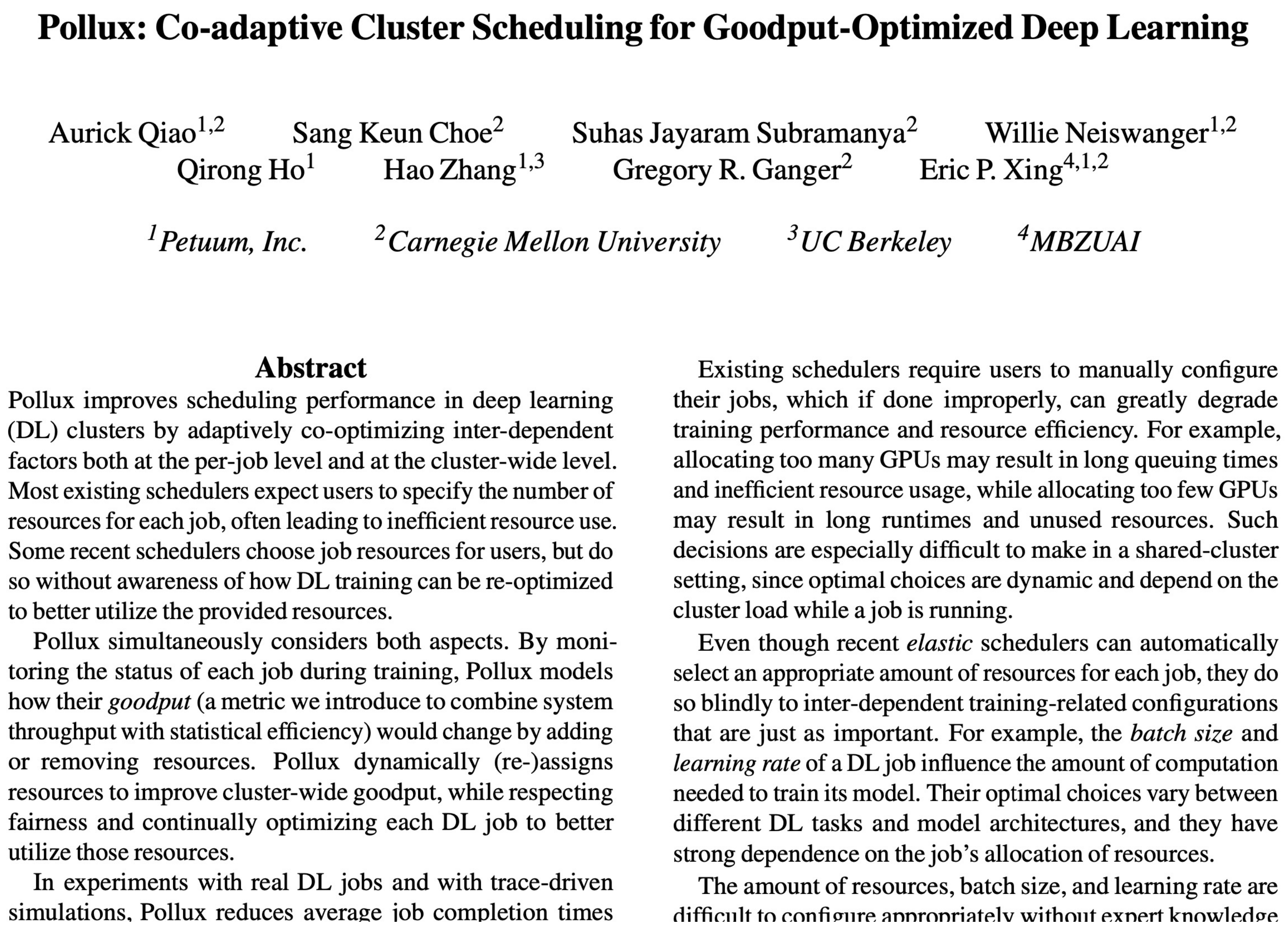
### 1 Introduction

With the rise of new high performance memory technologies, such as 3D XPoint and low latency NAND, new NVMe stor-

In contrast to these approaches, we seek a readily-deployable mechanism that can provide fast access to emerging fast storage devices that requires no specialized hardware and no significant changes to the application while working with existing kernels and file systems. To this end, we rely on BPF (Berkeley Packet Filter [67, 68]) which lets applications offload simple functions to the Linux kernel [8]. Similar to kernel bypass, by embedding application-logic deep in the kernel stack, BPF can eliminate overheads associated with kernel-user crossings and the associated context switches. Unlike kernel bypass, BPF is an OS-supported mechanism that ensures isolation, does not lead to low utilization due to busy-waiting, and allows a large number of threads or processes to share the same core, leading to better overall utilization.

The support of BPF in the Linux kernel makes it an attractive interface for allowing applications to speed up storage I/O. However, using BPF to speed up storage introduces several unique challenges. Unlike existing packet filtering and

# Machine learning training



Pollux: Co-adaptive Cluster Scheduling for Goodput-Optimized Deep Learning

Aurick Qiao[1,2] Sang Keun Choe[2] Suhas Jayaram Subramanya[2] Willie Neiswanger[1,2]
Qirong Ho[1] Hao Zhang[1,3] Gregory R. Ganger[2] Eric P. Xing[4,1,2]

[1] *Petuum, Inc.* [2] *Carnegie Mellon University* [3] *UC Berkeley* [4] *MBZUAI*

**Abstract**

Pollux improves scheduling performance in deep learning (DL) clusters by adaptively co-optimizing inter-dependent factors both at the per-job level and at the cluster-wide level. Most existing schedulers expect users to specify the number of resources for each job, often leading to inefficient resource use. Some recent schedulers choose job resources for users, but do so without awareness of how DL training can be re-optimized to better utilize the provided resources.

Pollux simultaneously considers both aspects. By monitoring the status of each job during training, Pollux models how their *goodput* (a metric we introduce to combine system throughput with statistical efficiency) would change by adding or removing resources. Pollux dynamically (re-)assigns resources to improve cluster-wide goodput, while respecting fairness and continually optimizing each DL job to better utilize those resources.

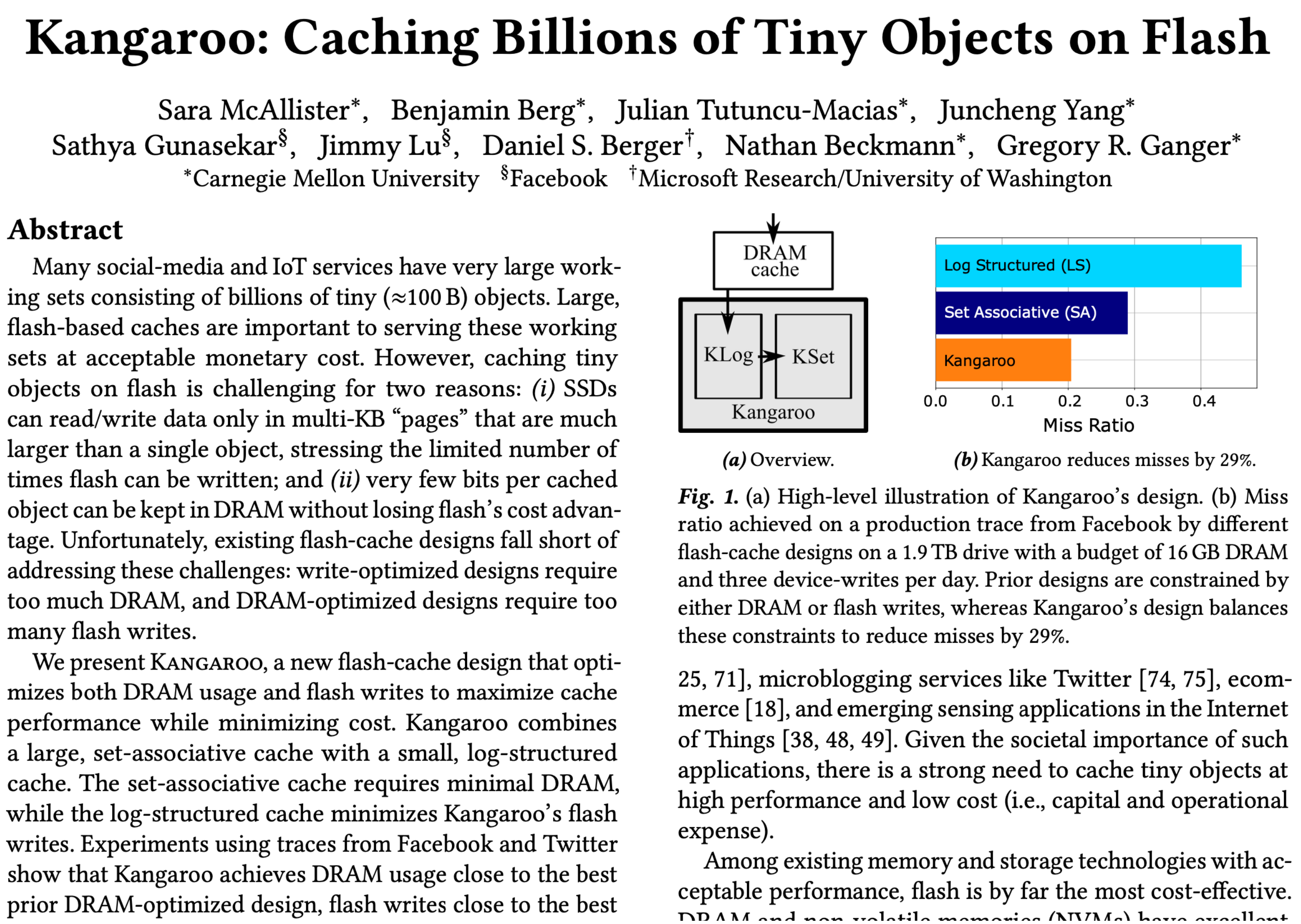
In experiments with real DL jobs and with trace-driven simulations, Pollux reduces average job completion times

Existing schedulers require users to manually configure their jobs, which if done improperly, can greatly degrade training performance and resource efficiency. For example, allocating too many GPUs may result in long queuing times and inefficient resource usage, while allocating too few GPUs may result in long runtimes and unused resources. Such decisions are especially difficult to make in a shared-cluster setting, since optimal choices are dynamic and depend on the cluster load while a job is running.

Even though recent *elastic* schedulers can automatically select an appropriate amount of resources for each job, they do so blindly to inter-dependent training-related configurations that are just as important. For example, the *batch size* and *learning rate* of a DL job influence the amount of computation needed to train its model. Their optimal choices vary between different DL tasks and model architectures, and they have strong dependence on the job's allocation of resources.

The amount of resources, batch size, and learning rate are difficult to configure appropriately without expert knowledge

# RAM / Flash interactions



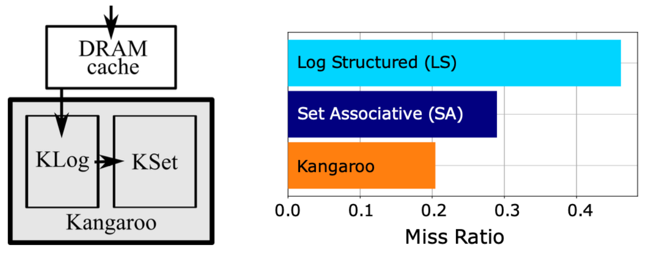
## Kangaroo: Caching Billions of Tiny Objects on Flash

Sara McAllister*, Benjamin Berg*, Julian Tutuncu-Macias*, Juncheng Yang*
Sathya Gunasekar§, Jimmy Lu§, Daniel S. Berger†, Nathan Beckmann*, Gregory R. Ganger*
*Carnegie Mellon University §Facebook †Microsoft Research/University of Washington

**Abstract**

Many social-media and IoT services have very large working sets consisting of billions of tiny (≈100 B) objects. Large, flash-based caches are important to serving these working sets at acceptable monetary cost. However, caching tiny objects on flash is challenging for two reasons: *(i)* SSDs can read/write data only in multi-KB "pages" that are much larger than a single object, stressing the limited number of times flash can be written; and *(ii)* very few bits per cached object can be kept in DRAM without losing flash's cost advantage. Unfortunately, existing flash-cache designs fall short of addressing these challenges: write-optimized designs require too much DRAM, and DRAM-optimized designs require too many flash writes.

We present KANGAROO, a new flash-cache design that optimizes both DRAM usage and flash writes to maximize cache performance while minimizing cost. Kangaroo combines a large, set-associative cache with a small, log-structured cache. The set-associative cache requires minimal DRAM, while the log-structured cache minimizes Kangaroo's flash writes. Experiments using traces from Facebook and Twitter show that Kangaroo achieves DRAM usage close to the best prior DRAM-optimized design, flash writes close to the best



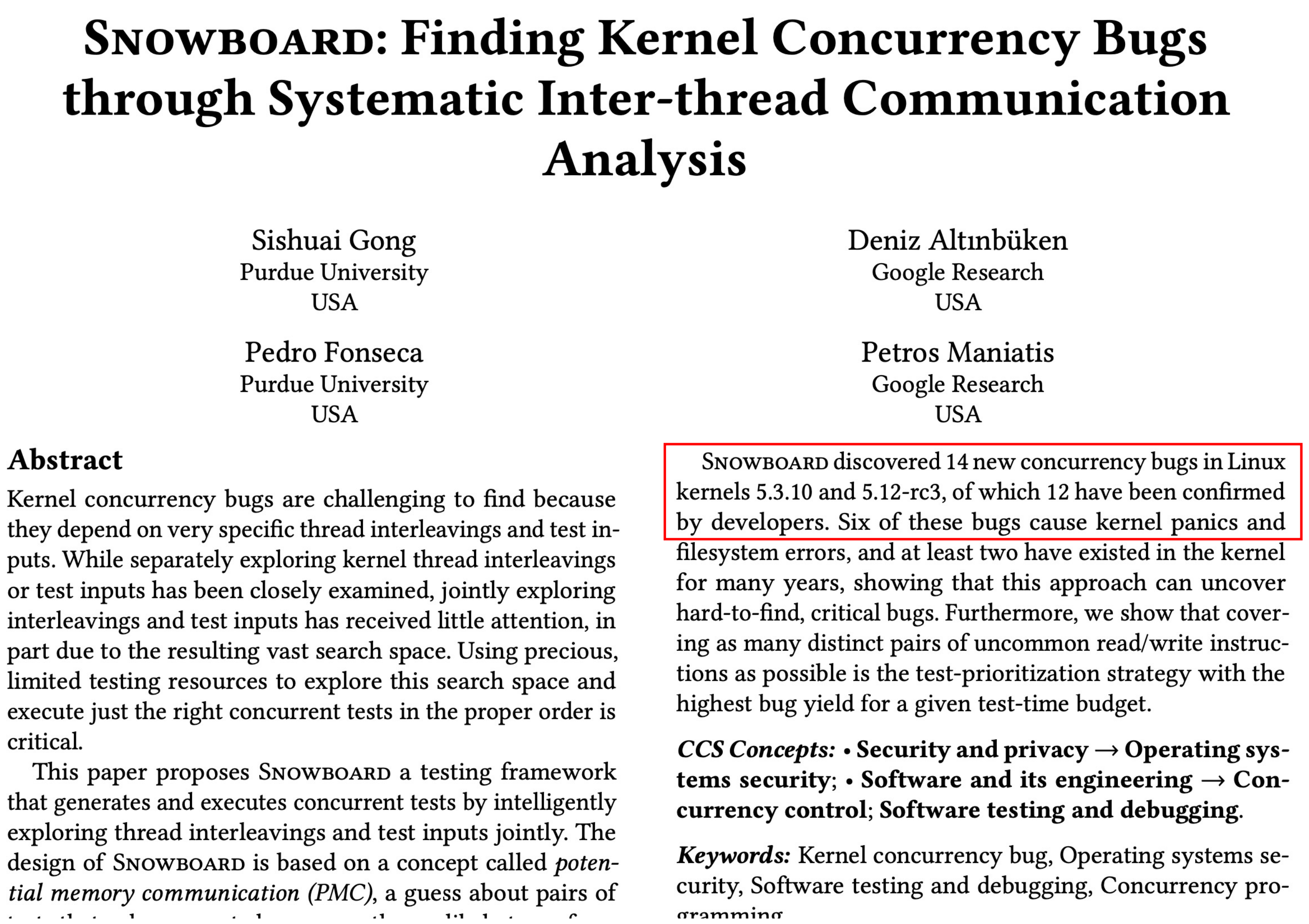
***(a)*** Overview. ***(b)*** Kangaroo reduces misses by 29%.

***Fig. 1.*** (a) High-level illustration of Kangaroo's design. (b) Miss ratio achieved on a production trace from Facebook by different flash-cache designs on a 1.9 TB drive with a budget of 16 GB DRAM and three device-writes per day. Prior designs are constrained by either DRAM or flash writes, whereas Kangaroo's design balances these constraints to reduce misses by 29%.

25, 71], microblogging services like Twitter [74, 75], ecommerce [18], and emerging sensing applications in the Internet of Things [38, 48, 49]. Given the societal importance of such applications, there is a strong need to cache tiny objects at high performance and low cost (i.e., capital and operational expense).

Among existing memory and storage technologies with acceptable performance, flash is by far the most cost-effective.

# Concurrency bugs



## SNOWBOARD: Finding Kernel Concurrency Bugs through Systematic Inter-thread Communication Analysis

Sishuai Gong
Purdue University
USA

Deniz Altınbüken
Google Research
USA

Pedro Fonseca
Purdue University
USA

Petros Maniatis
Google Research
USA

**Abstract**

Kernel concurrency bugs are challenging to find because they depend on very specific thread interleavings and test inputs. While separately exploring kernel thread interleavings or test inputs has been closely examined, jointly exploring interleavings and test inputs has received little attention, in part due to the resulting vast search space. Using precious, limited testing resources to explore this search space and execute just the right concurrent tests in the proper order is critical.

This paper proposes SNOWBOARD a testing framework that generates and executes concurrent tests by intelligently exploring thread interleavings and test inputs jointly. The design of SNOWBOARD is based on a concept called *potential memory communication (PMC)*, a guess about pairs of

SNOWBOARD discovered 14 new concurrency bugs in Linux kernels 5.3.10 and 5.12-rc3, of which 12 have been confirmed by developers. Six of these bugs cause kernel panics and filesystem errors, and at least two have existed in the kernel for many years, showing that this approach can uncover hard-to-find, critical bugs. Furthermore, we show that covering as many distinct pairs of uncommon read/write instructions as possible is the test-prioritization strategy with the highest bug yield for a given test-time budget.

***CCS Concepts:*** • **Security and privacy** → **Operating systems security**; • **Software and its engineering** → **Concurrency control**; **Software testing and debugging**.

***Keywords:*** Kernel concurrency bug, Operating systems security, Software testing and debugging, Concurrency programming