



15th USENIX Symposium on Operating Systems
Design and Implementation

Preview of Operating Systems and Hardware Session



Michio Honda

University of Edinburgh



Sudarsun Kannan

Rutgers University

Memory Scaling Challenges

- Scaling application memory capacity without increasing management cost is becoming critical
- Scaling memory protection and isolation for large address space equally critical

1. Memory tuning is tedious



Requires extensive application knowledge and requires constant tuning

2. Memory security is challenging



Hardware memory security non-scalable

Beyond malloc efficiency to fleet efficiency: a hugepage-aware memory allocator

A.H. Hunter
*Jane Street Capital**

Chris Kennelly
Google

Paul Turner
Google

Darryl Gove
Google

Tipp Moseley
Google

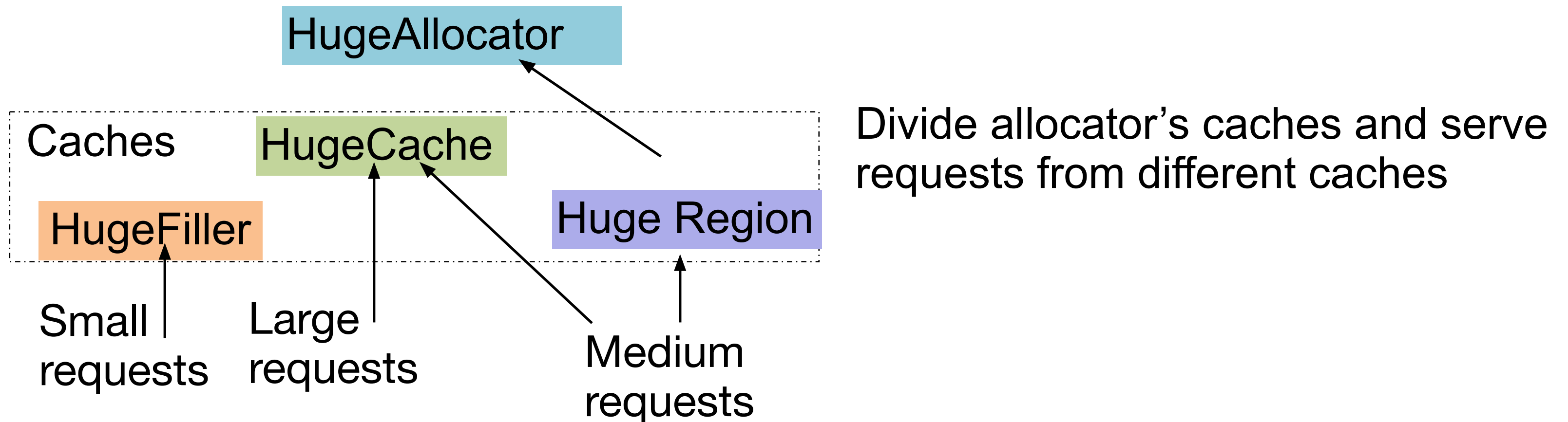
Parthasarathy Ranganathan
Google

Memory Allocator Challenges

- Long history of memory allocators designed for specific application needs
 - Concurrency and low fragmentation (e.g., Hoard, jemalloc, TCMalloc)
 - Minimize L1 misses (e.g., Dice), increase locality (e.g., mimalloc)
- However, allocators are not optimized for HugePages
 - HugePages becoming increasingly ubiquitous in large scale applications
 - Could substantially reduce TLB misses by increasing RAM coverage
- Using existing allocators with HugePages could increase fragmentation and are inefficient for warehouse scale systems running several applications

TEMEIRAIRE

- Hugepage-aware user-level allocator using TCMALLOC
- Aims at densely packing huge pages grouped into few, saturated bins
- Balances memory usage and page allocation costs through adaptive huge page release



- Average 6% reduction TLB misses and 26% reduction in memory usage across a fleet of applications

Scalable Memory Protection in the PENGLAI Enclave

*Erhu Feng^{1†‡}, Xu Lu^{1†‡}, Dong Du^{†‡}, Bicheng Yang^{†‡}, Xueqiang Jiang^{†‡}, Yubin Xia^{†§‡},
Binyu Zang^{†§‡}, Haibo Chen^{†§‡}*

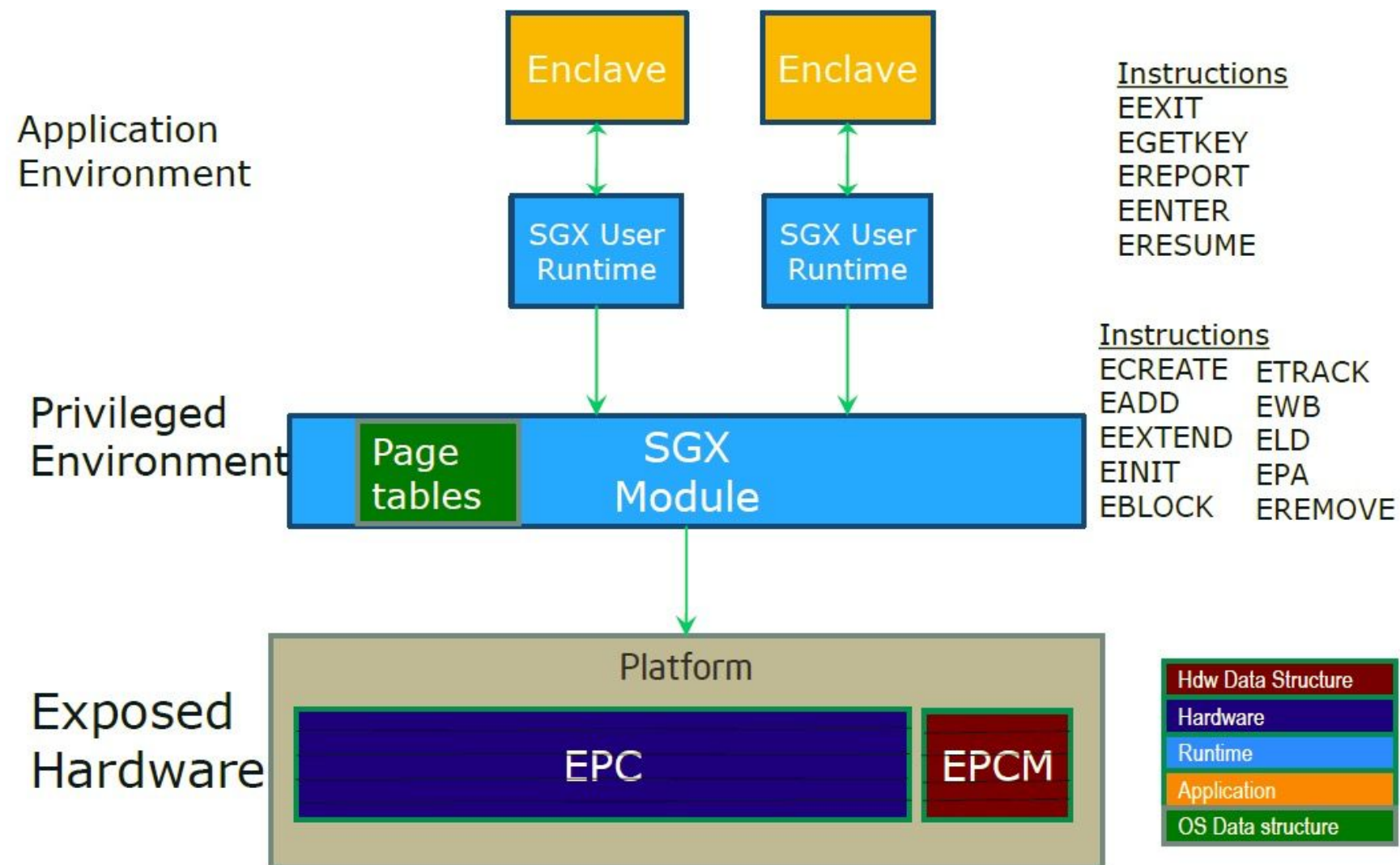
[†]Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University

[§]Shanghai AI Laboratory

[‡]Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China

Hardware Enclaves 101

→ Hardware abstractions and support for trusted execution on untrusted platforms



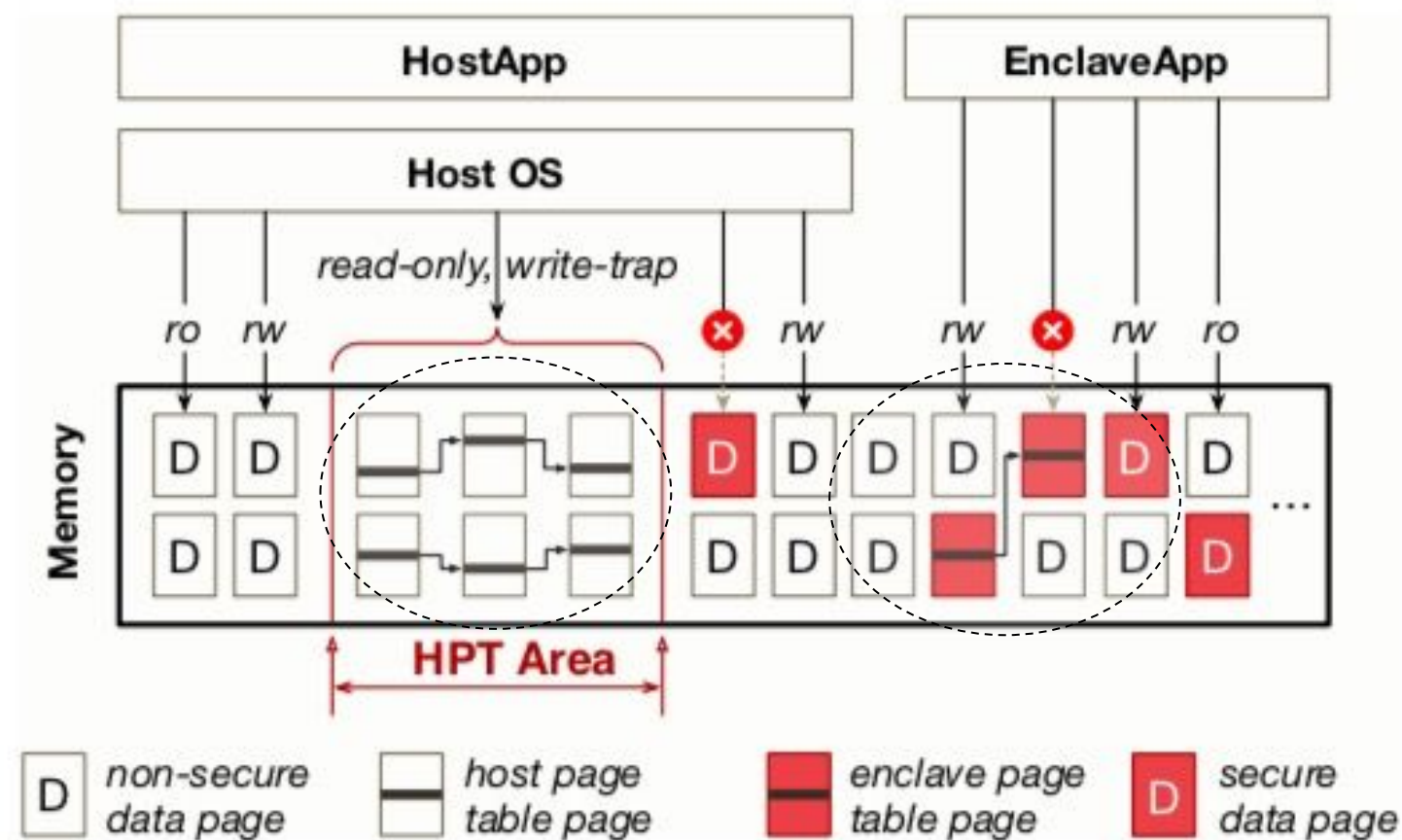
→ Hardware enclaves: secure boot, on-chip program isolation, protected external memory, execution integrity, and other capabilities

Hardware Enclaves Challenges

- Non-scalable memory partition/isolation
 - Current hardware supports only 256MB enclaves
 - Some restrict the number of enclaves
 - Require static partitioning
- Non-scalable memory integrity protection
 - Huge memory overhead to store memory integrity information (e.g., hash)
 - Hardware (e.g., Intel SGX) only supports ~256MB, demands swapping
- Non-scalable secure memory initialization
 - High-cost secure memory initialization increases enclave setup cost
 - Impractical for serverless applications

PENGLAI Enclave

- Scalable secure memory protection mechanisms for enclaves
- Approach to Scaling: novel *Guarded Page Table* structure
- Guarded Page Table Intuition: map secure and unsecure pages to separate non-secure host page table and secure enclave page table



- Scaling Integrity Protection: Mountable Merkle Tree (MMT), a SubTree structure to reduce both on-die and in-memory storage overhead

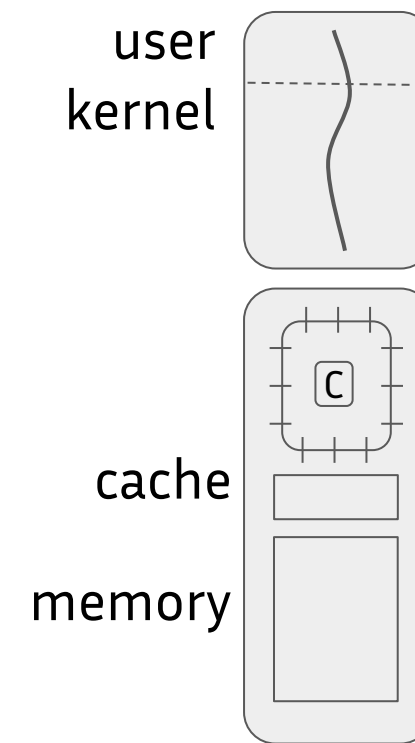


15th USENIX Symposium on Operating Systems
Design and Implementation

(NrOS and nanoPU)

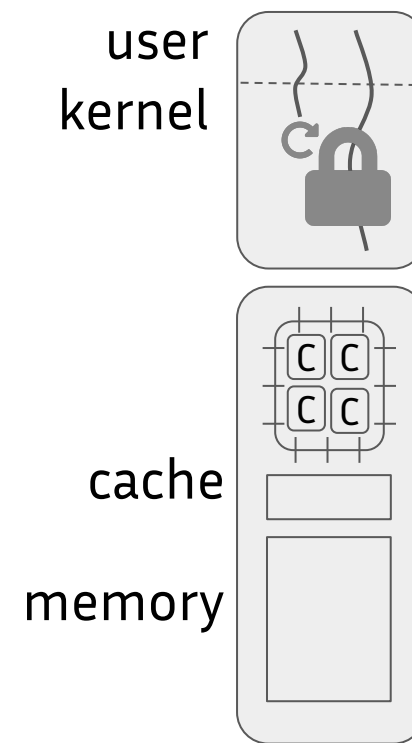
OS evolution over HW generations

- Single core
 - Just protect critical sections from interrupts
 - I/O was also slow



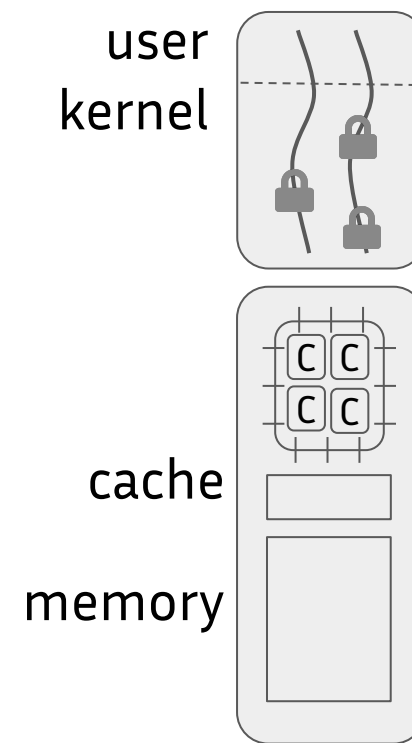
OS evolution over HW generations

- Single core
 - Just protect critical sections from interrupts
 - I/O was also slow
- Multiple CPU cores
 - Giant lock
 - Fine-grained locks
 - Reader-writer locks



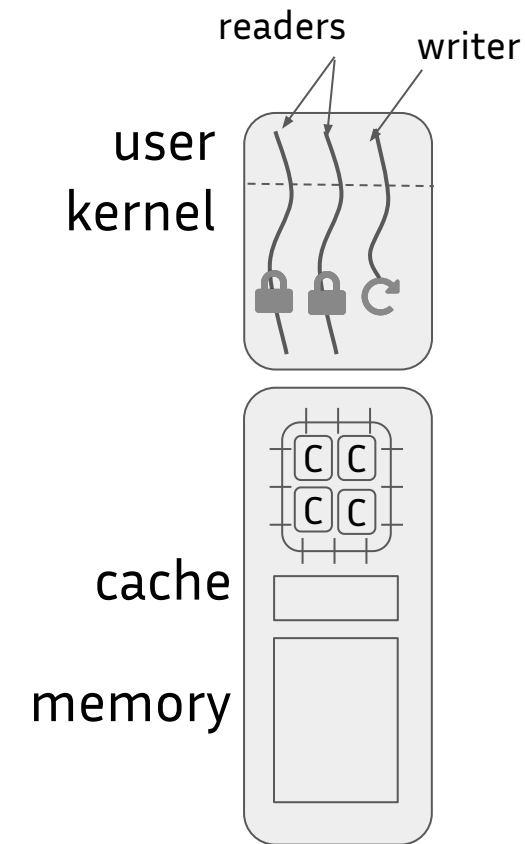
OS evolution over HW generations

- Single core
 - Just protect critical sections from interrupts
 - I/O was also slow
- Multiple CPU cores
 - Giant lock
 - Fine-grained locks
 - Reader-writer locks



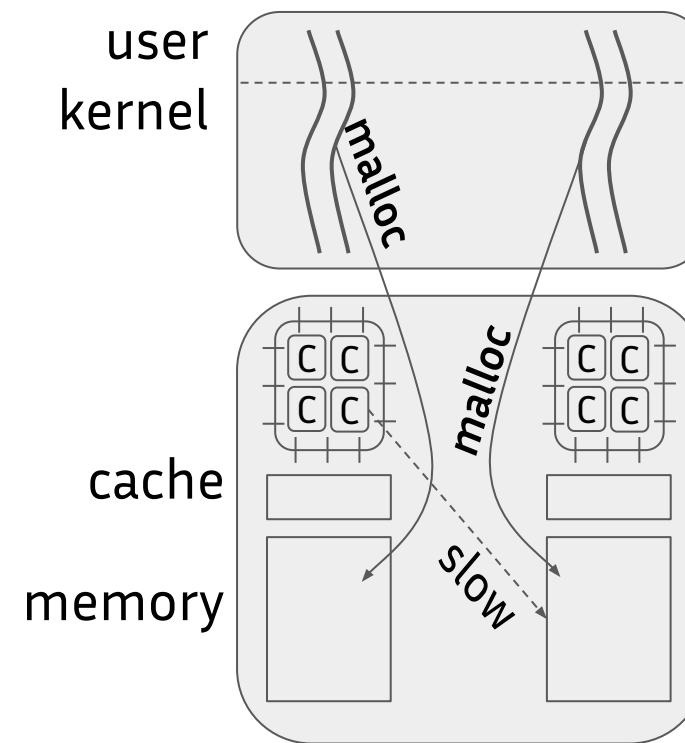
OS evolution over HW generations

- Single core
 - Just protect critical sections from interrupts
 - I/O was also slow
- Multiple CPU cores
 - Giant lock
 - Fine-grained locks
 - Reader-writer locks



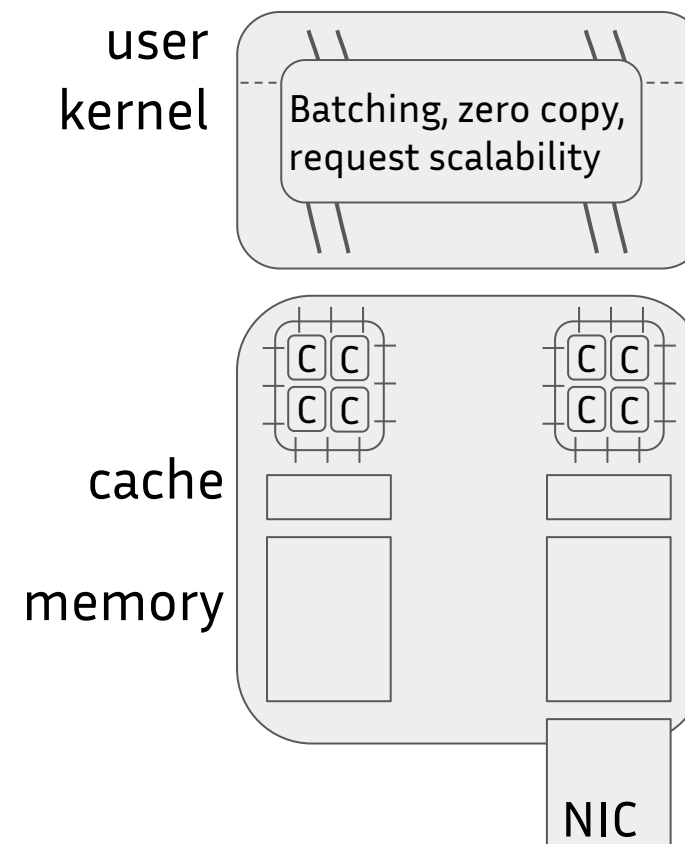
OS evolution over HW generations

- Single core
 - Just protect critical sections from interrupts
 - I/O was also slow
- Multiple CPU cores
 - Giant lock
 - Fine-grained locks
 - Reader-writer locks
- Multiple CPU packages (sockets)
 - NUMA-aware memory allocation and scheduling



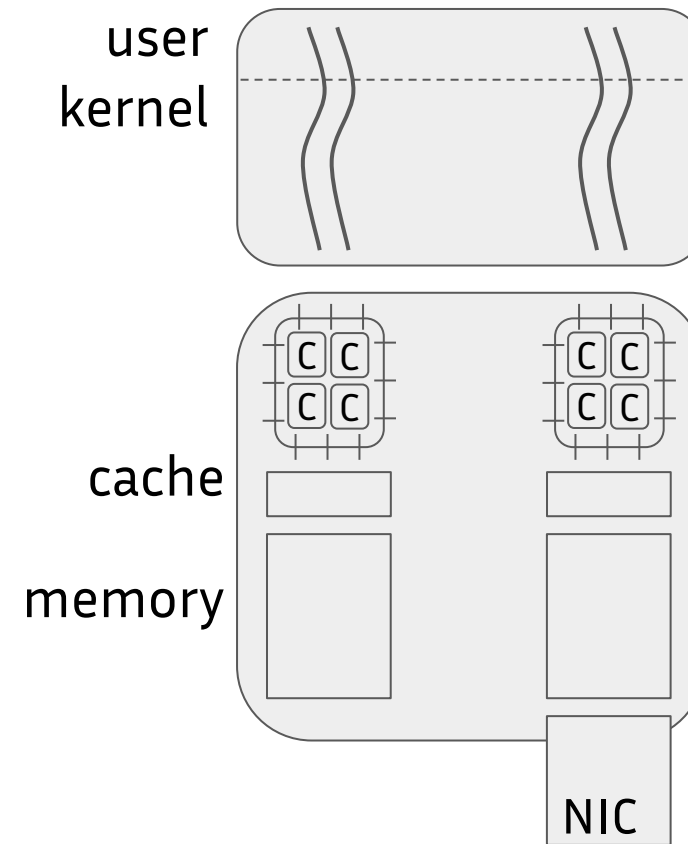
OS evolution over HW generations

- Single core
 - Just protect critical sections from interrupts
 - I/O was also slow
- Multiple CPU cores
 - Giant lock
 - Fine-grained locks
 - Reader-writer locks
- Multiple CPU packages (sockets)
 - NUMA-aware memory allocation and scheduling
- Fast I/O
 - Interrupt mitigation and load-balancing
 - New APIs
(kqueue/epoll/netmap/io_uring)



OS evolution over HW generations

- Single core
 - Just protect critical sections from interrupts
 - I/O was also slow
- Multiple CPU cores
 - Giant lock
 - Fine-grained locks
 - Reader-writer locks
- Multiple CPU packages (sockets)
 - NUMA-aware memory allocation and scheduling
- Fast I/O
 - Interrupt mitigation and load-balancing
 - New APIs (kqueue/epoll/netmap/io_uring)



```
bh_lock_sock_nested(sk);
tcp_segs_in(tcp_sk(sk), skb);
ret = 0;
if (!sock_owned_by_user(sk)) {
    skb_to_free = sk->sk_rx_skb_cache;
    sk->sk_rx_skb_cache = NULL;
    ret = tcp_v4_do_rcv(sk, skb);
} else {
    if (tcp_add_backlog(sk, skb))
        goto discard_and_release;
    skb_to_free = NULL;
}
bh_unlock_sock(sk);
if (skb_to_free)
    __kfree_skb(skb_to_free);

out_and_return:
if (refcounted)
    sock_put(sk);
```

Check another socket lock (sleepable one)

Lock socket (non-sleepable)

socket is also ref-counted

All of these make kernel code complex and error-prone, but such a kernel is still not scalable!

NrOS

| Design | Synchronization | Kernel programming | Scalability |
|-------------|-----------------------|--------------------|-------------|
| Monolithic | Shared states | Hard | Low |
| Multikernel | Message passing | Easy | Low |
| NrOS | Operation logs | Easy | High |

- Use of shared last-level CPU cache
- Operation logs shared by per-NUMA-node replicas
 - Synchronization batching
- NetBSD LibOS
 - POSIX app support

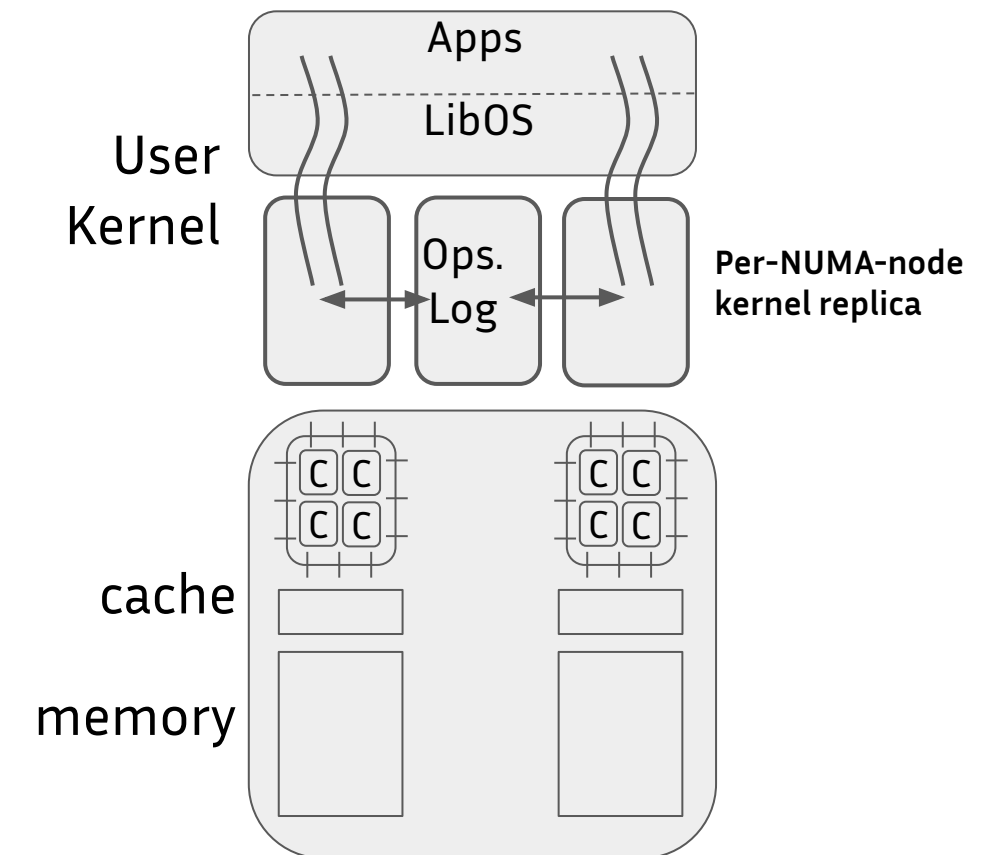


NrOS: Effective Replication and Sharing in an Operating System

Ankit Bhardwaj¹, Chinmay Kulkarni¹, Reto Achermann², Irina Calciu³,
Sanidhya Kashyap⁴, Ryan Stutsman¹, Amy Tai³, and Gerd Zellweger³

¹University of Utah, ²University of British Columbia, ³VMware Research, ⁴EPFL

Abstract
Writing a correct operating system kernel is notoriously difficult. For monolithic kernels, this slows development. For multikernel systems, this slows deployment. We present NrOS, a new operating system kernel that uses custom-tailored concurrent data structures with fine-grained locking or techniques like read-copy-update (RCU) to achieve good performance. For monolithic kernels, this slows development.




nanoPU

- Co-designing NIC and CPU
 - NIC places receiving data directly in a CPU register file
- Ultrafast small RPCs (nanoRequests)
 - High-rate small requests are hard to handle, because most overheads are per-packet or per-request, NOT per bytes
 - nanoPU reduces both average and tail latency

Design highlights

- Avoid the two latency sources:
 - Host stack
 - Bypass the stack and memory hierarchy
 - Queues in networks
 - Transport protocol in HW

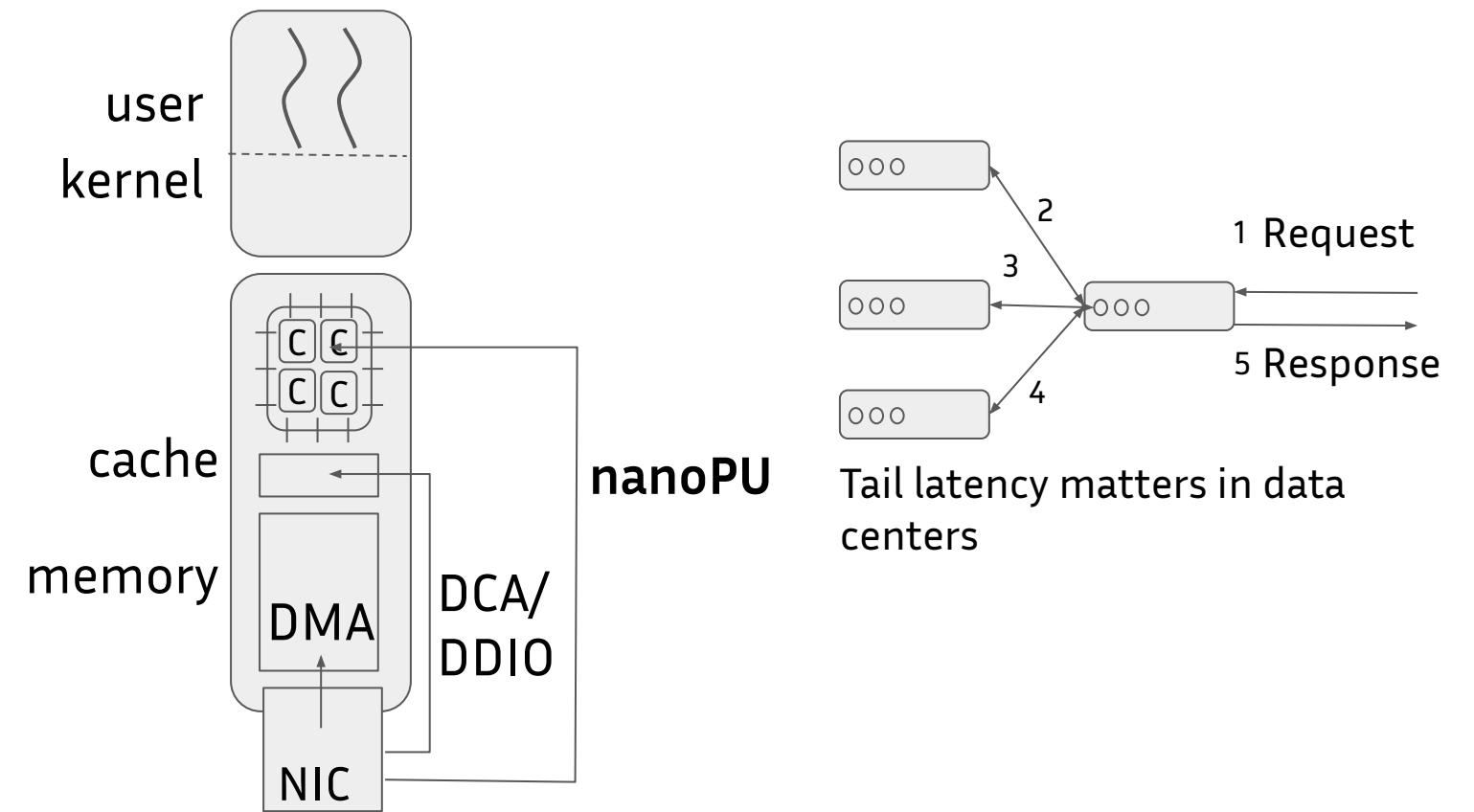


The nanoPU: A Nanosecond Network Stack for Datacenters

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 low fanouts. The nanoPU is designed to reduce (1) the latency 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).





15th USENIX Symposium on Operating Systems
Design and Implementation

Operating Systems and Hardware Session

Thursday, July 15

7:00 am–8:15 am (PDT)