15-712:
Advanced Operating Systems & Distributed Systems

Multicore Kernels

Prof. Phillip Gibbons

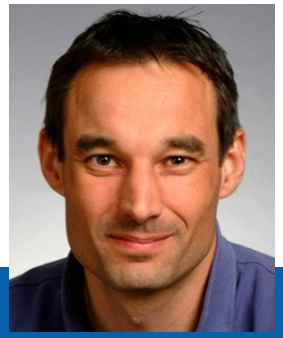
Spring 2023, Lecture 18

"The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors"

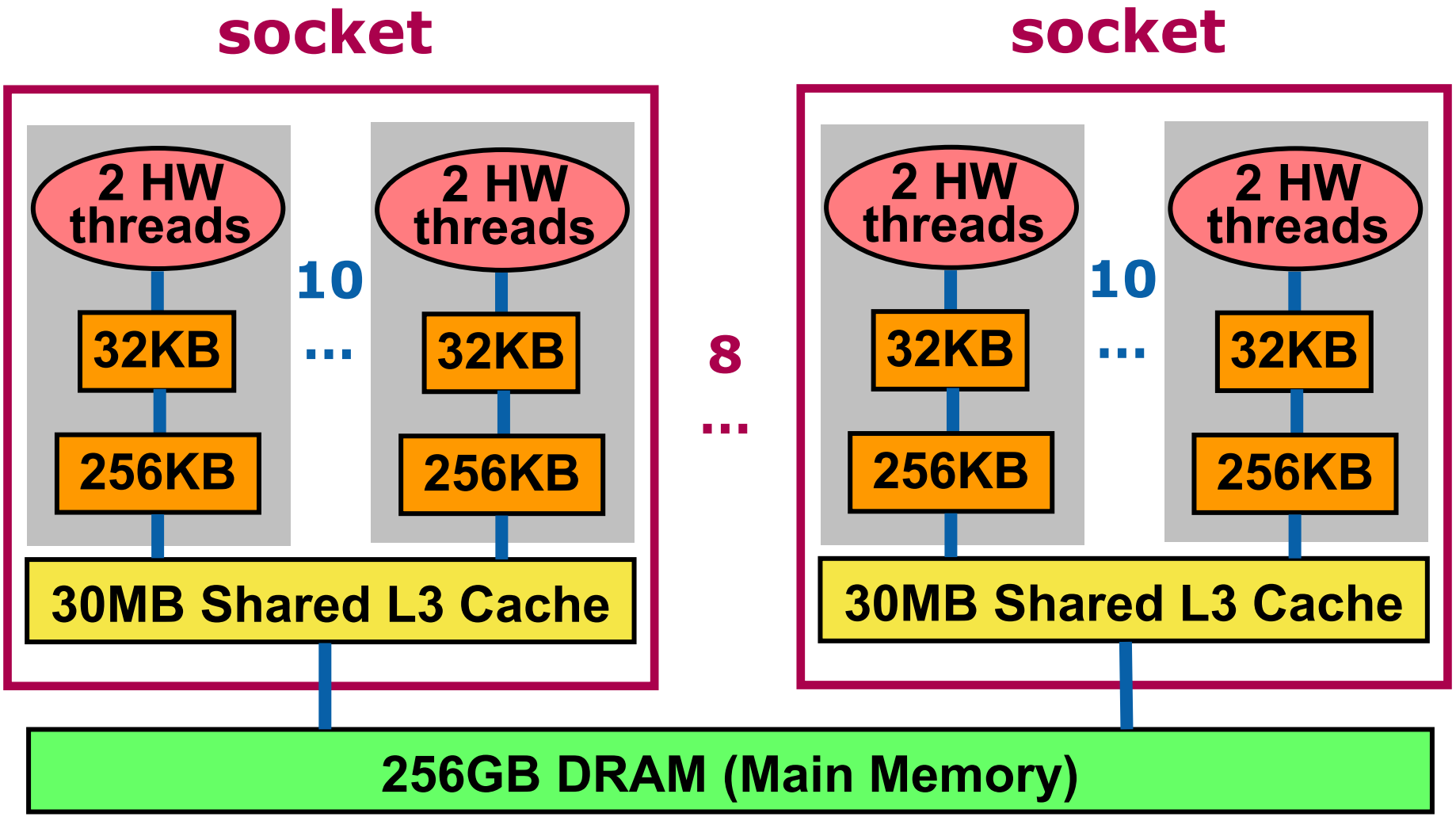
Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, Eddie Kohler 2013

- **Austin Clements** (MIT, Google) – His PhD thesis
- **Frans Kaashoek** (MIT) – NAE, NAAS
- **Nickolai Zeldovich** (MIT) – Mark Weiser Award
- **Robert Morris** (MIT) – Morris worm
- **Eddie Kohler** (Harvard) – developed HotCrp

SOSP'13 Best Paper


Multicore: 80-core Intel Xeon E7-8870 (2013)



Not shown: 32KB L1 Instruction cache per core

Aside: An Analysis of Linux Scalability to Many Cores [same group, OSDI'10]

- **Analyzed 7 system apps running on Linux on 48-cores**
  - Exim, memcached, Apache, PostgreSQL, gmake, Psearchy, MapReduce
  - Used an in-memory file system to explore non-disk limitations
- **Identified & sought to remove all scalability bottlenecks**
- **Kernel changes are all localized**
  - typically involve avoiding locks & atomic instructions by organizing data structures to avoid unnecessary sharing

Serializing Actions

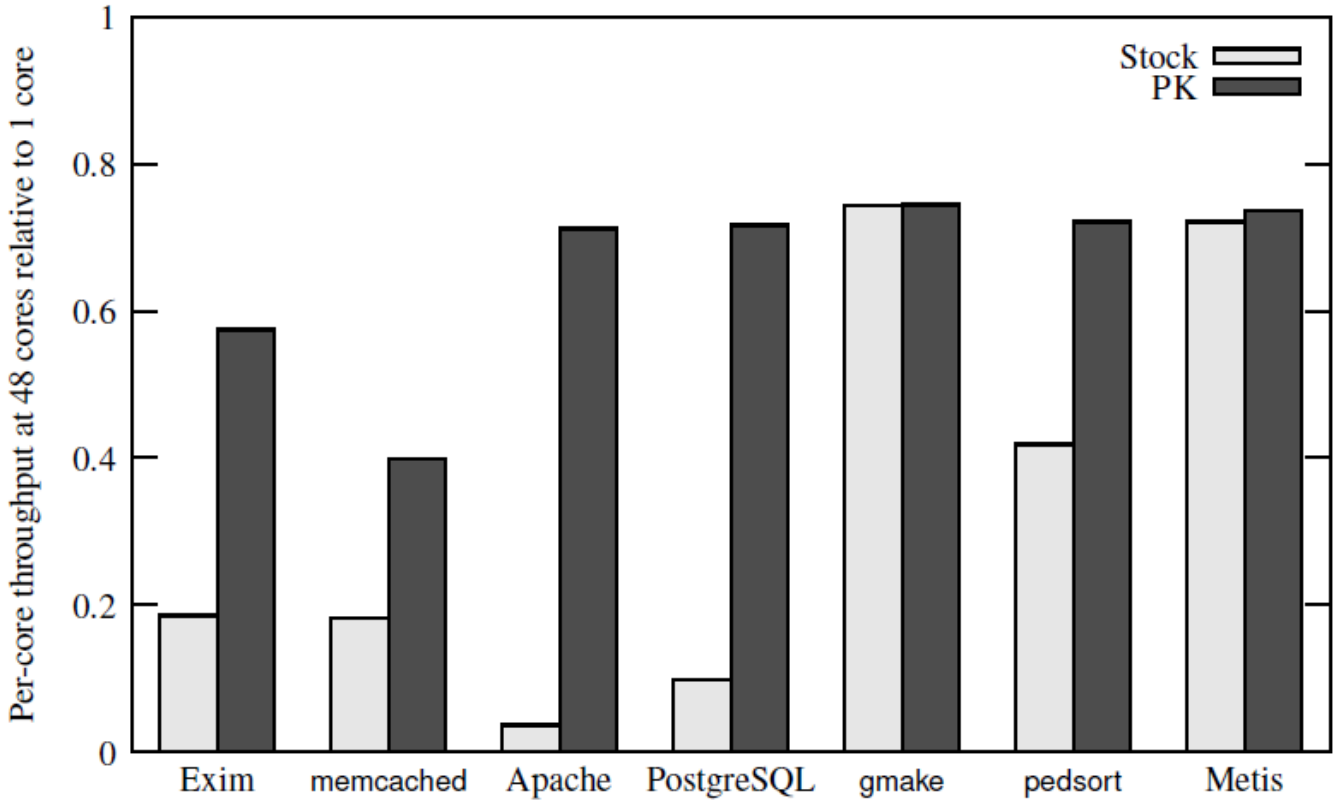
Scalability bottlenecks can arise when tasks:

- **Lock a shared data structure**
- **Write a shared memory location**
- **Compete for on-chip cache space**
- **Compete for on-chip interconnect or DRAM interface**
- **Are already mostly idle (not enough parallel work)**

Scalability Problems & Fixes

Parallel `accept`			Apache
Concurrent `accept` system calls contend on shared `socket` fields.	⇒	User per-core backlog queues for listening sockets.	
`dentry` reference counting			Apache, Exim
File name resolution contends on directory entry reference counts.	⇒	Use sloppy counters to reference count directory entry objects.	
Mount point (`vfsmount`) reference counting			Apache, Exim
Walking file name paths contends on mount point reference counts.	⇒	Use sloppy counters for mount point objects.	
IP packet destination (`dst_entry`) reference counting			memcached, Apache
IP packet transmission contends on routing table entries.	⇒	Use sloppy counters for IP routing table entries.	
Protocol memory usage tracking			memcached, Apache
Cores contend on counters for tracking protocol memory consumption.	⇒	Use sloppy counters for protocol usage counting.	
Acquiring directory entry (`dentry`) spin locks			Apache, Exim
Walking file name paths contends on per-directory entry spin locks.	⇒	Use a lock-free protocol in `dlookup` for checking filename matches.	
Mount point table spin lock			Apache, Exim
Resolving path names to mount points contends on a global spin lock.	⇒	Use per-core mount table caches.	
Adding files to the open list			Apache, Exim
Cores contend on a per-super block list that tracks open files.	⇒	Use per-core open file lists for each super block that has open files.	
Allocating DMA buffers			memcached, Apache
DMA memory allocations contend on the memory node 0 spin lock.	⇒	Allocate Ethernet device DMA buffers from the local memory node.	
False sharing in `net_device` and `device`			memcached, Apache, PostgreSQL
False sharing causes contention for read-only structure fields.	⇒	Place read-only fields on their own cache lines.	
False sharing in `page`			Exim
False sharing causes contention for read-mostly structure fields.	⇒	Place read-only fields on their own cache lines.	
`inode` lists			memcached, Apache
Cores contend on global locks protecting lists used to track `inodes`.	⇒	Avoid acquiring the locks when not necessary.	
Dcache lists			memcached, Apache
Cores contend on global locks protecting lists used to track `dentrys`.	⇒	Avoid acquiring the locks when not necessary.	
Per-`inode` mutex			PostgreSQL
Cores contend on a per-`inode` mutex in `lseek`.	⇒	Use atomic reads to eliminate the need to acquire the mutex.	
Super-page fine grained locking			Metis
Super-page soft page faults contend on a per-process mutex.	⇒	Protect each super-page memory mapping with its own mutex.	
Zeroing super-pages			Metis
Zeroing super-pages flushes the contents of on-chip caches.	⇒	Use non-caching instructions to zero the contents of super-pages.	

Throughput: Before & After



PK = Patched Kernel

Application	Bottleneck
Exim	App: Contention on spool directories
memcached	HW: Transmit queues on NIC
Apache	HW: Receive queues on NIC
PostgreSQL	App: Application-level spin lock
gmake	App: Serial stages and stragglers
pedsort	HW: Cache capacity
Metis	HW: DRAM throughput

Claim: Remaining bottlenecks are not due to Linux

[Adapted from slides in SOSP'13 talk]

The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors

Austin T. Clements
M. Frans Kaashoek
Nickolai Zeldovich
Robert Morris
Eddie Kohler †

MIT CSAIL and † Harvard

Current approach to scalable software development

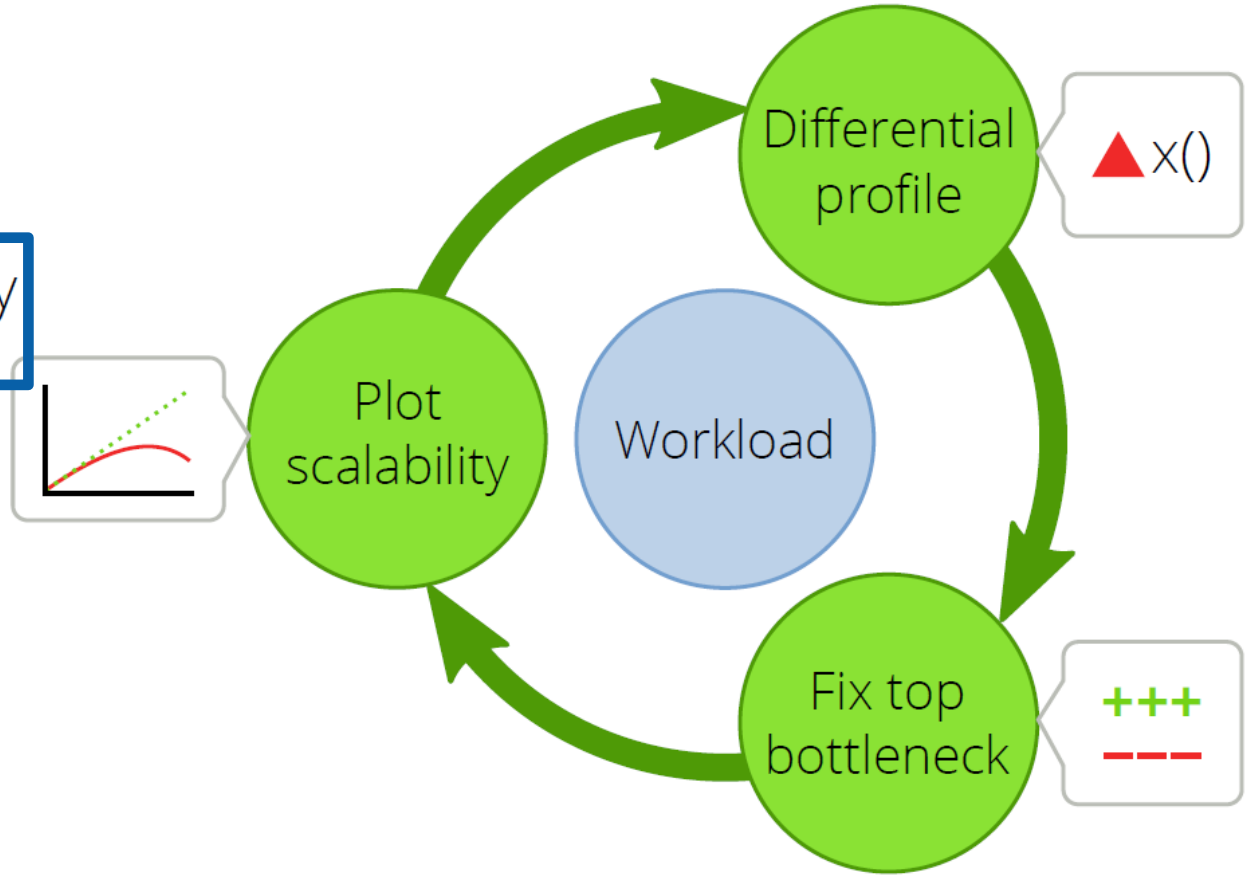
2009 Corey
OSDI '08

2010 Linux scalability
OSDI '10

2011

2012 Bonsai VM
ASPLOS '12

2013 RadixVM
EuroSys '13



Disadvantages?

Current approach to scalable software development

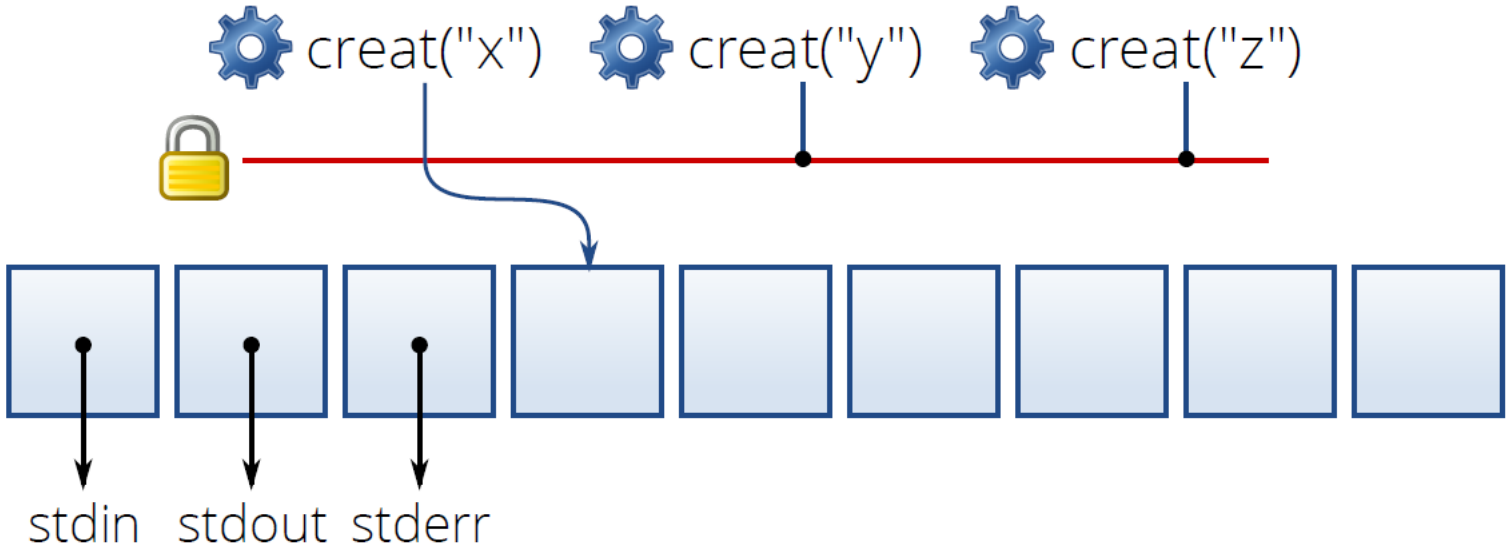
Successful in practice because it focuses developer effort

Disadvantages

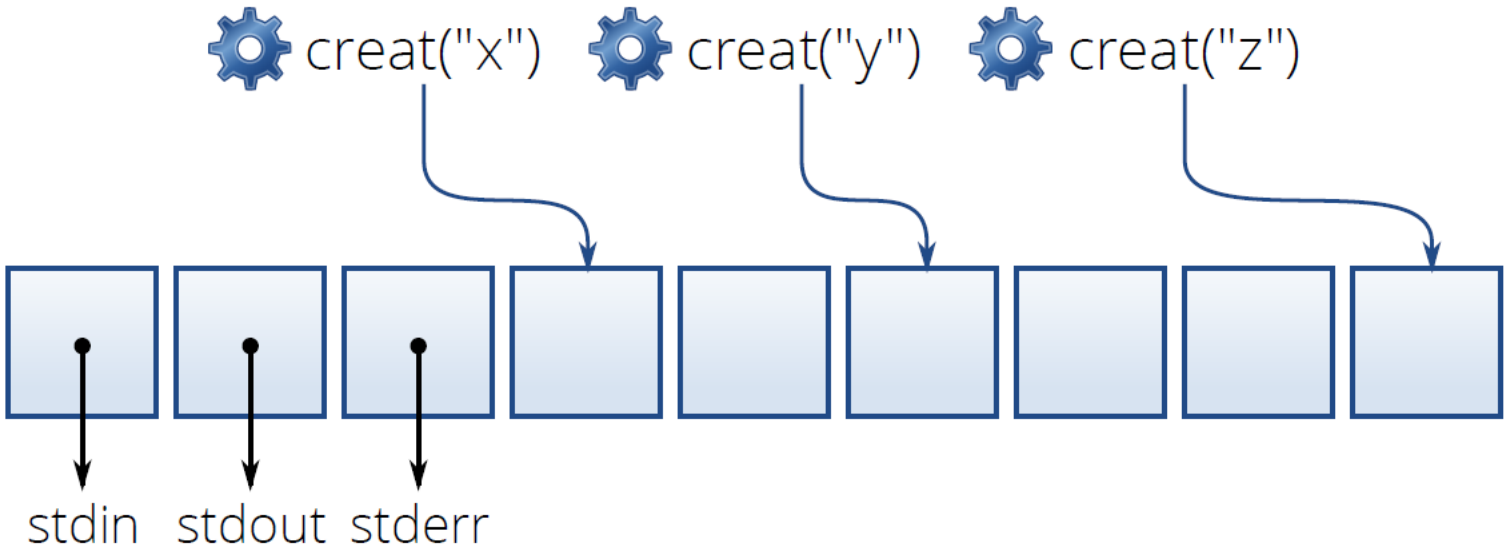
- New workloads expose new bottlenecks
- More cores expose new bottlenecks
- The real bottlenecks may be in the interface design

- Developers may give up too soon, w/o realizing a scalable solution is possible
- Problems discovered (too) late in the development process

Interface scalability example



Interface scalability example



Approach: Interface-driven scalability

The scalable commutativity rule

Whenever interface operations commute, they can be implemented in a way that scales.

Approach: Interface-driven scalability

The scalable commutativity rule

Whenever interface operations commute, they can be implemented in a way that scales.

	Commutes	Scalable implementation exists
creat with lowest FD	?	
	creat → 3 creat → 4	

Approach: Interface-driven scalability

The scalable commutativity rule

Whenever interface operations commute, they can be implemented in a way that scales.

	Commutes		Scalable implementation exists
creat with lowest FD	✗		
creat with any FD	✓	rule →	✓

The intuition behind the rule

Whenever interface operations commute, they can be implemented in a way that scales.

Operations commute
⇒ results independent of order
⇒ communication is unnecessary
⇒ without communication, no conflicts

Advantages of interface-driven scalability

The rule enables reasoning about scalability throughout the software design process

Design	Guides design of scalable interfaces
Implement	Sets a clear implementation target
Test	Systematic, workload-independent scalability testing

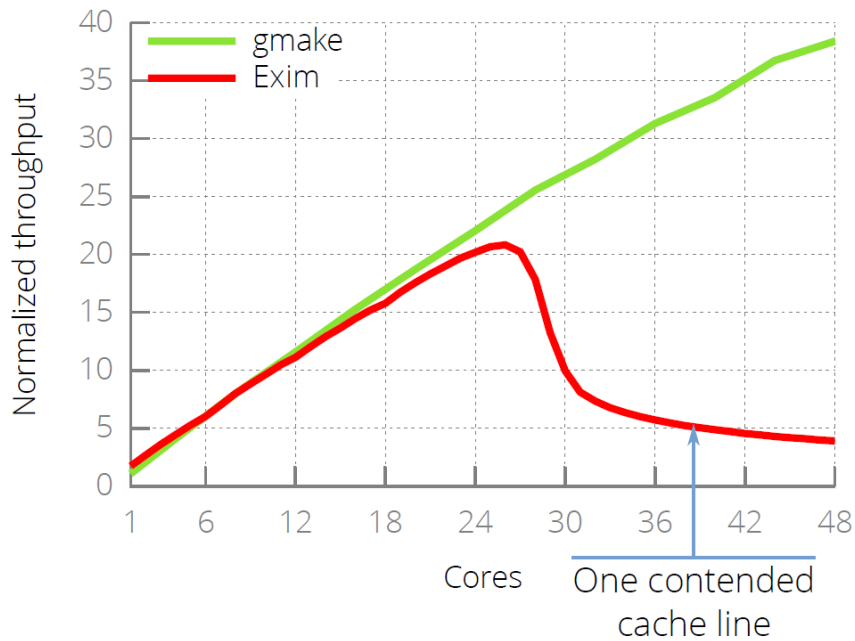
What scales on today's multicores?

Core *Y* \ Core *X*	W	R	-
W	✗	✗	✓
R	✗	✓	✓
-	✓	✓	-

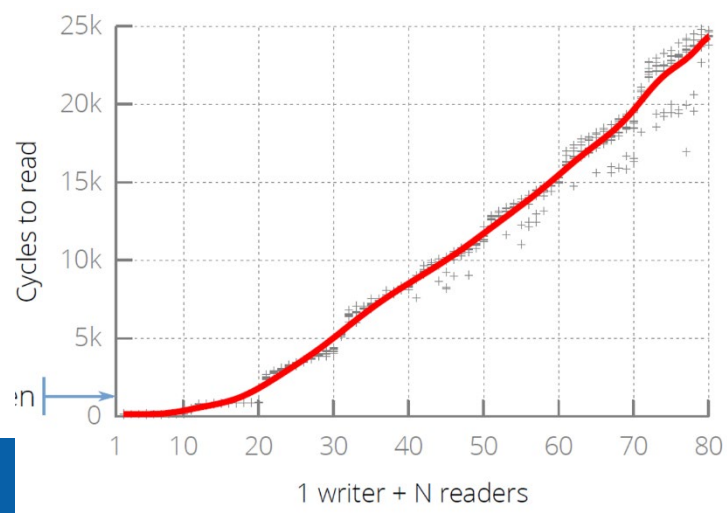
We say two or more operations are *scalable* if they are *conflict-free*.

Conflict-free:
No core writes a cache line that was read/written by another core

A scalability bottleneck



A single contended cache line can wreck scalability



Contributions

The scalable commutativity rule

- Formalization of the rule and proof of its correctness
- State-dependent, interface-based commutativity

Commuter: An automated scalability testing tool

sv6: A scalable POSIX-like kernel

Commutativity ≡ conflict-free at interface-level for specific operations, arguments, state:

Y SI-commutes in X || Y ≡ for any reordering Y' of Y and any Z, X || Y || Z is valid iff X || Y' || Z is valid

---

+ Monotonicity: Y SIM-commutes in X || Y ≡ P SI-commutes in X || P, for any prefix P of any reordering Y' of Y

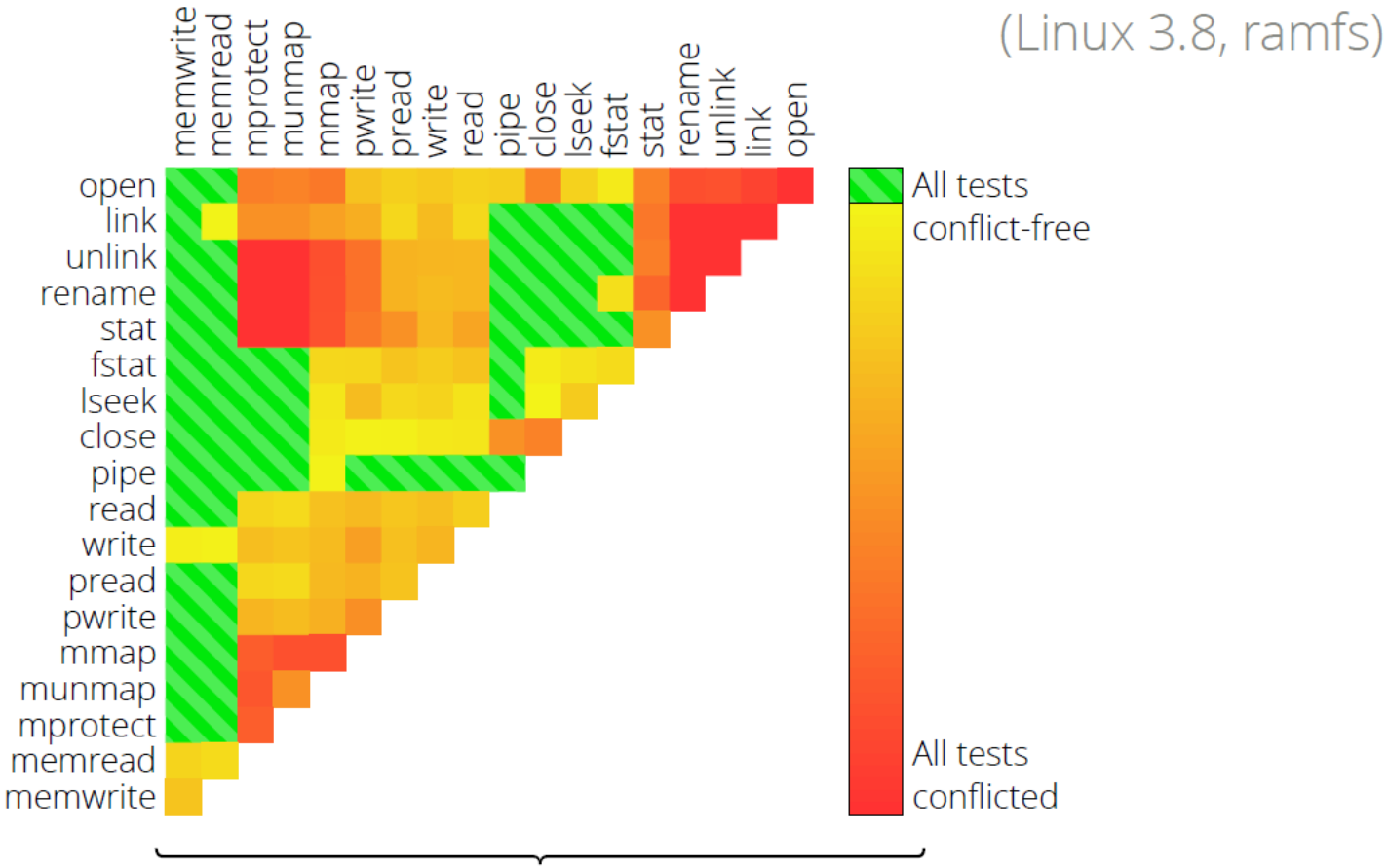
Example of using the rule

	Commutes	Scalable implementation exists
P1: creat P1: creat	✗	
P1: creat("/tmp/x") P2: creat("/etc/y")	✓	✓ (Linux)
P1: creat("/x") P2: creat("/y")	✓	✓
P1: creat("x", O_EXCL) P2: creat("x", O_EXCL)		
Same CWD	✗	
Different CWD	✓	✓

Discussion: Summary Question #1

- **State the 3 most important things the paper says.** These could be some combination of their motivations, observations, interesting parts of the design, or clever parts of their implementation.

Commuter finds non-scalable cases in Linux

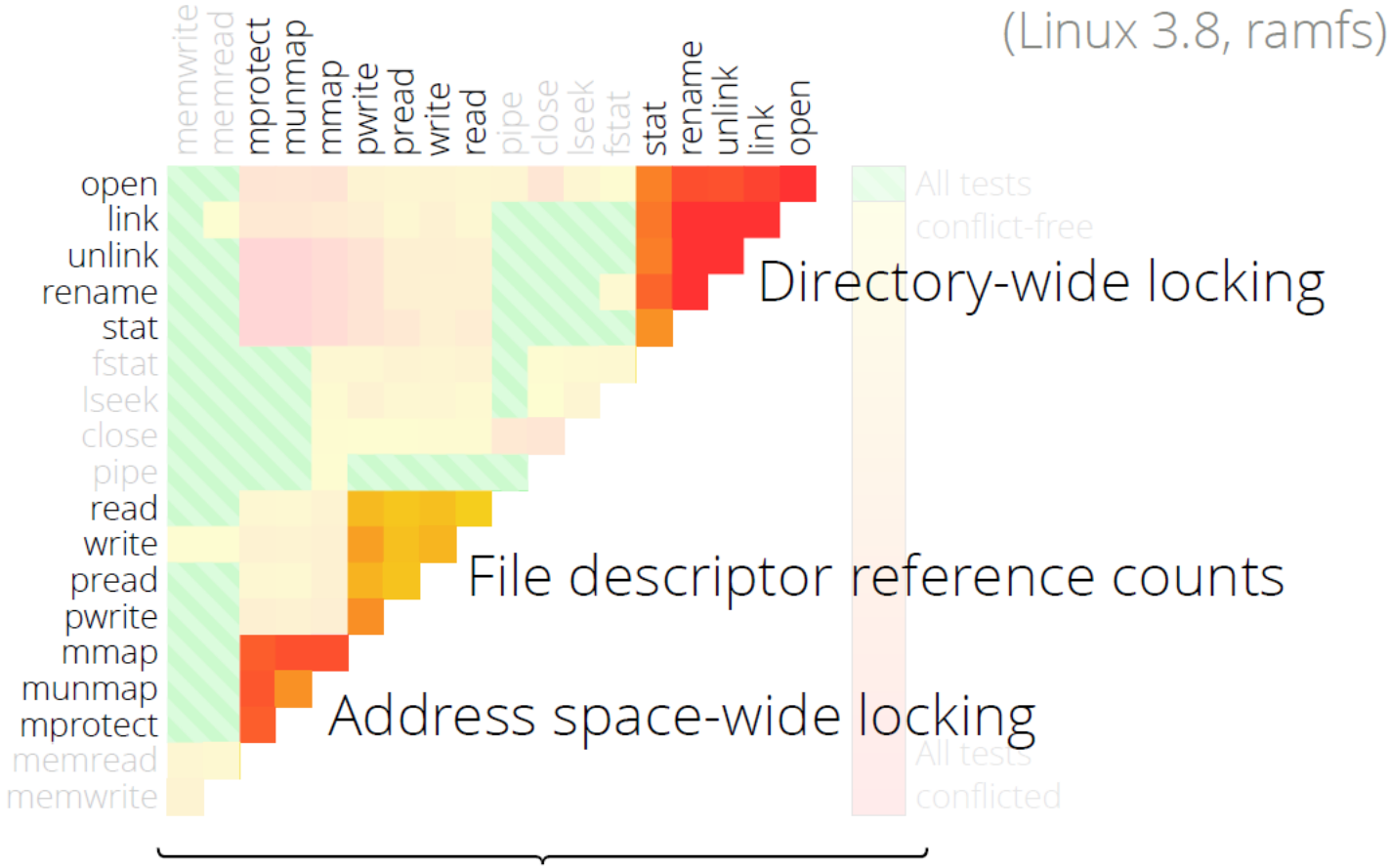


13,664 total test cases

68% are conflict-free

Many are "corner cases," many are not.

Commuter finds non-scalable cases in Linux



13,664 total test cases

68% are conflict-free

Many are "corner cases," many are not.

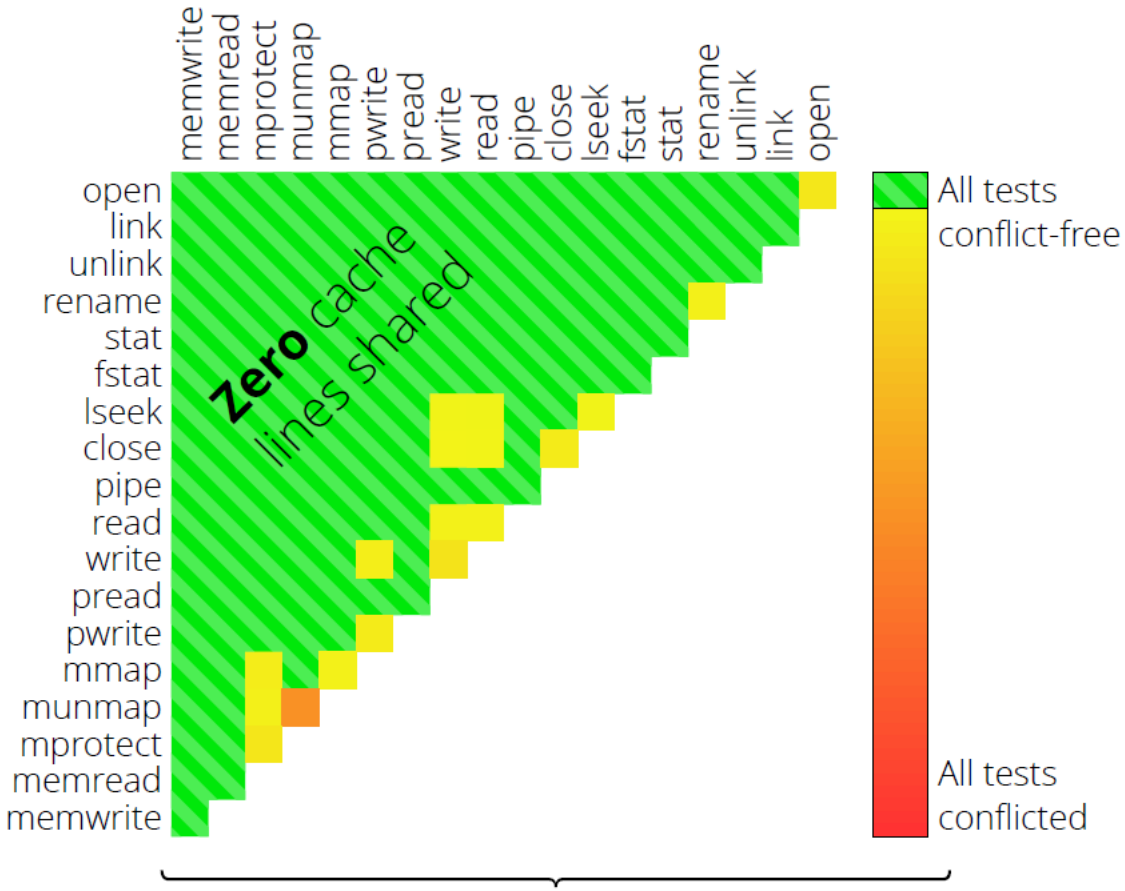
sv6: A scalable OS

POSIX-like operating system

File system and virtual memory system follow commutativity rule

Implementation using standard parallel programming techniques, but guided by Commuter

Commutative operations can be made to scale

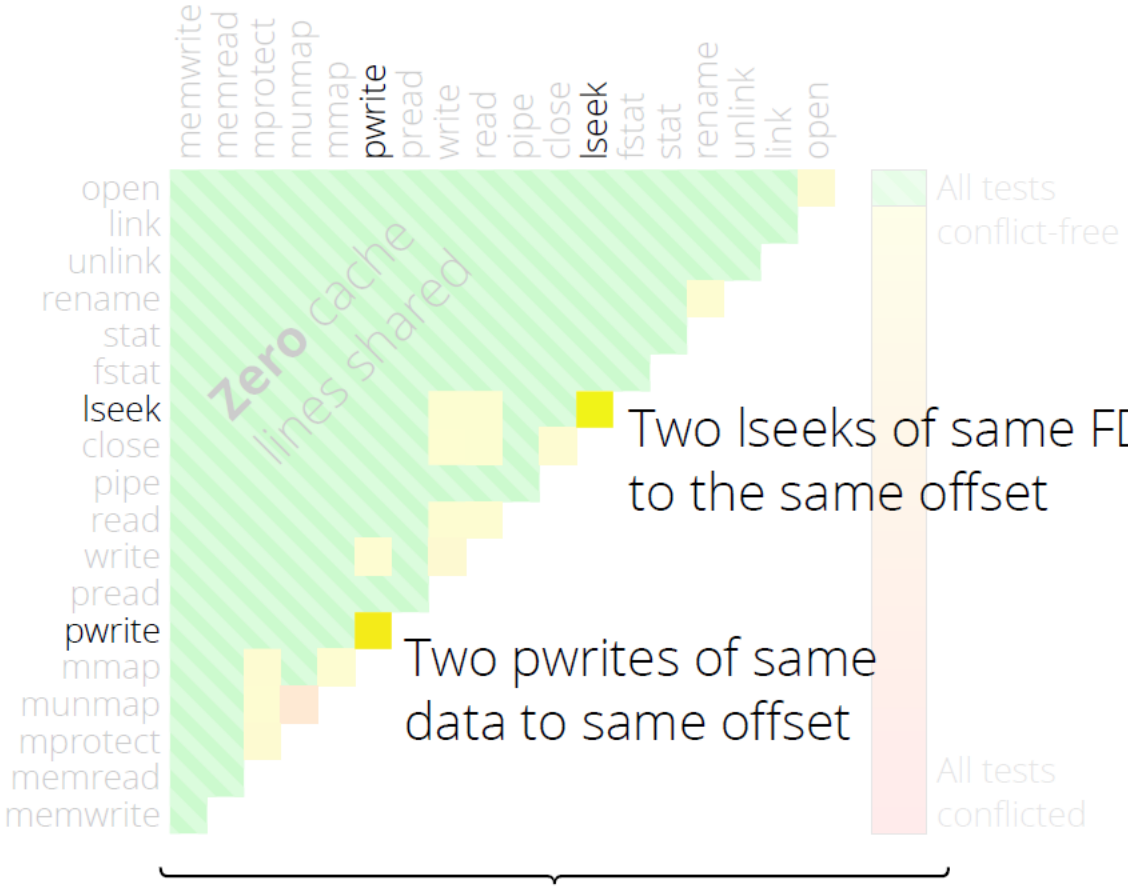


13,664 total test cases

99% are conflict-free

Remaining 1% are mostly "idempotent updates"

Commutative operations can be made to scale

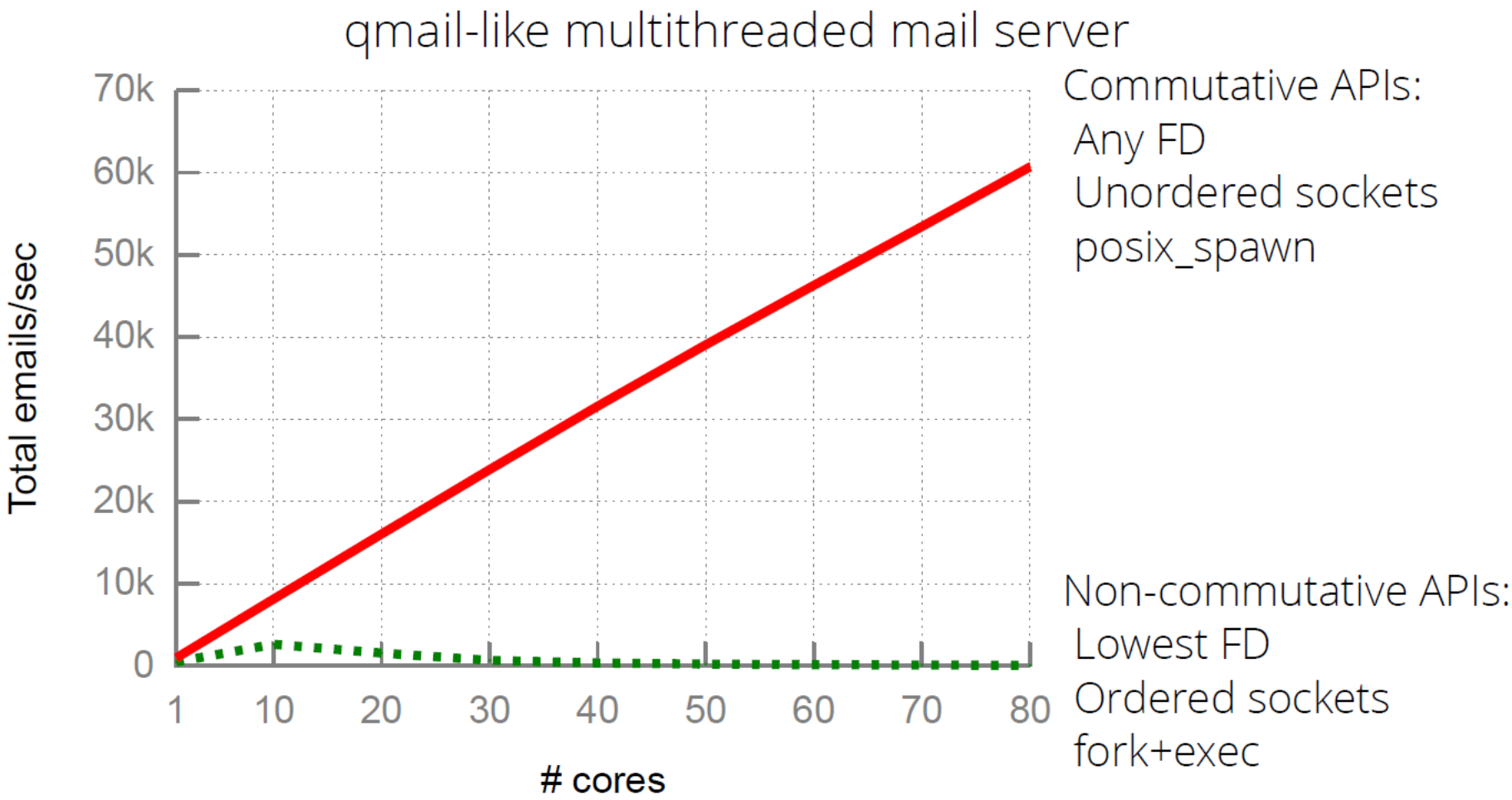


13,664 total test cases

99% are conflict-free

Remaining 1% are mostly "idempotent updates"

Commutative operations matter to app scalabiliy



Discussion: Summary Question #2

- **Describe the paper's single most glaring deficiency.** Every paper has some fault. Perhaps an experiment was poorly designed or the main idea had a narrow scope or applicability.

Input: Symbolic model

```
SymInode    = tstruct(data  = tlist(SymByte),
                      nlink = SymInt)
SymIMap     = tdict(SymInt, SymInode)
SymFilename = tuninterpreted('Filename')
SymDir      = tdict(SymFilename, SymInt)

class POSIX:
  def __init__(self):
    self.fname_to_inum = SymDir.any()
    self.inodes = SymIMap.any()

  @symargs(src=SymFilename, dst=SymFilename)
  def rename(self, src, dst):
    if src not in self.fname_to_inum:
      return (-1, errno.ENOENT)
    if src == dst:
      return 0
    if dst in self.fname_to_inum:
      self.inodes[self.fname_to_inum[dst]].nlink -= 1
    self.fname_to_inum[dst] = self.fname_to_inum[src]
    del self.fname_to_inum[src]
    return 0
```

Commutativity conditions

```
@symargs(src=SymFilename, dst=SymFilename)
def rename(self, src, dst):
  if src not in self.fname_to_inum:
    return (-1, errno.ENOENT)
  if src == dst:
    return 0
  if dst in self.fname_to_inum:
    self.inodes[self.fname_to_inum[dst]].nlink -= 1
  self.fname_to_inum[dst] = self.fname_to_inum[src]
  del self.fname_to_inum[src]
  return 0
```

rename(a, b) and rename(c, d) commute if:

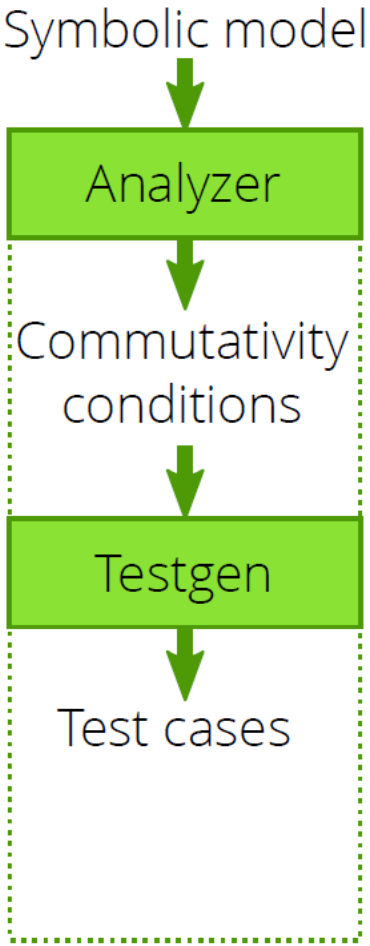
- Both source files exist and all names are different
- Neither source file exists
- a xor c exists, and it is not the other rename's destination
- Both calls are self-renames
- One call is a self-rename of an existing file and a != c
- a & c are hard links to the same inode, a != c, and b == d

Test cases

rename(a, b) and rename(c, d) commute if:

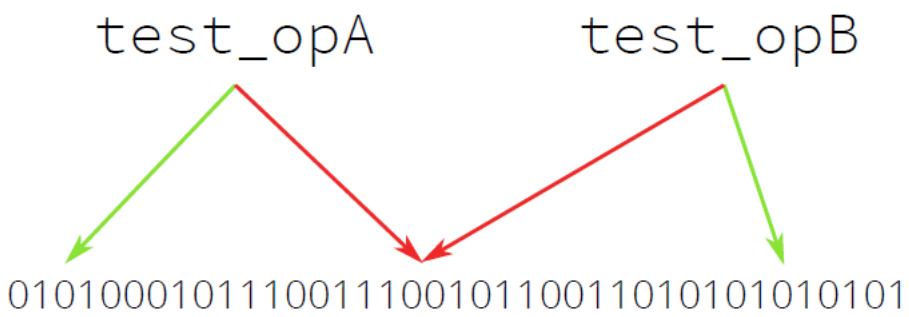
- Both source files exist and all names are different
- Neither source file exists
- a xor c exists, and it is not the other rename's destination
- Both calls are self-renames
- One call is a self-rename of an existing file and a != c
- a & c are hard links to the same inode, a != c, and b == d

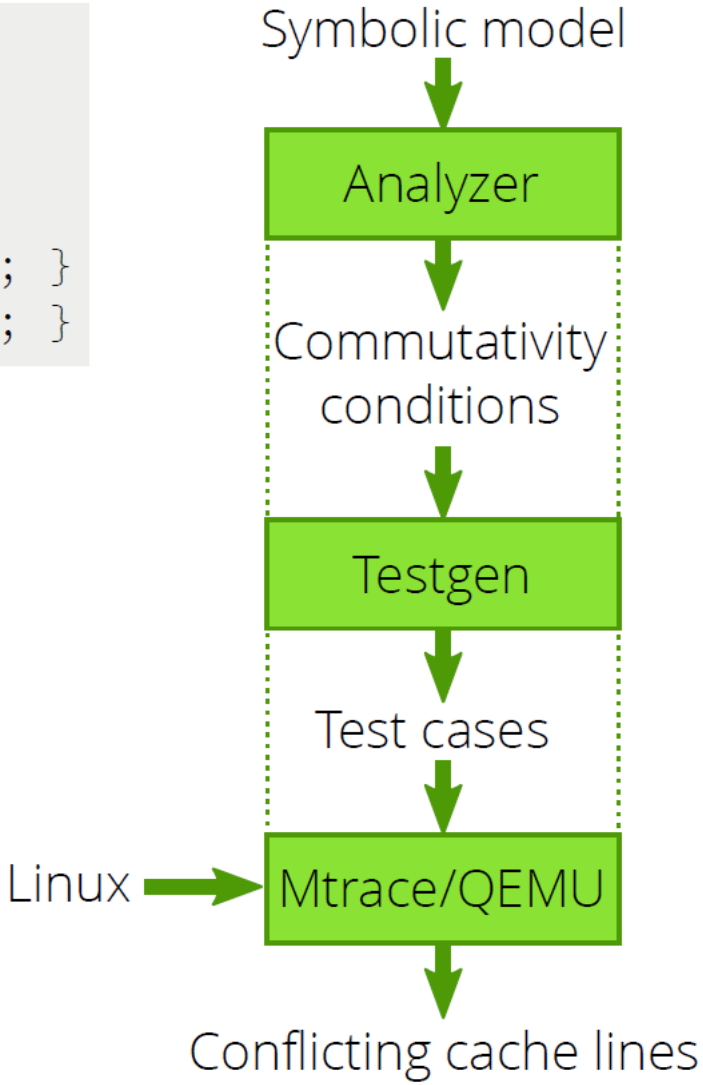
```
void setup() {
    close(creat("f0", 0666));
    close(creat("f2", 0666));
}
void test_opA() { rename("f0", "f1"); }
void test_opB() { rename("f2", "f3"); }
```



Output: Conflicting cache lines

```
void setup() {
    close(creat("f0", 0666));
    close(creat("f2", 0666));
}
void test_opA() { rename("f0", "f1"); }
void test_opB() { rename("f2", "f3"); }
```





Interface Changes for POSIX

1. **Decompose compound operations**
2. **Embrace specification non-determinism**
3. **Permit weak ordering**
4. **Release resources asynchronously**

- Lowest FD versus any FD **(2)**
- stat versus xstat **(1)**
- Unordered sockets **(3)**
- Delayed munmap **(4)**
- fork+exec versus posix_spawn **(1)**

Important Limitations

- **Scalable Commutativity Rule only implies that there exists an implementation with conflict-free accesses**
  - Implementation constructed in proof is not practical

- **Can scalability suffer even for conflict-free accesses?**
  1. Lock a shared data structure
  2. Write a shared memory location
  3. Compete for on-chip cache space
  4. Compete for on-chip interconnect or DRAM interface
  5. Are already mostly idle

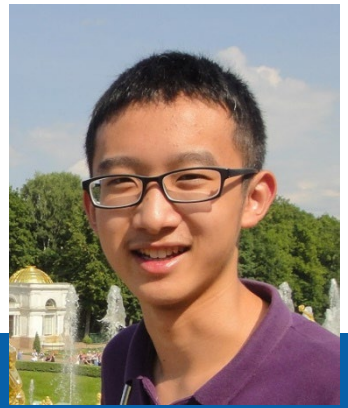
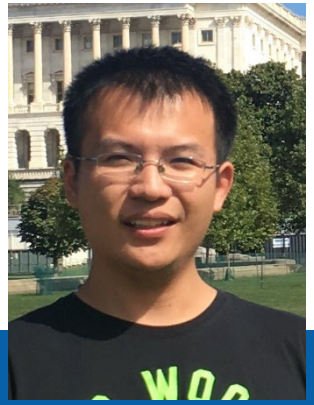
Yes. Only addresses first two sources of lack of scalability

"LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation"

Yizhou Shan, Yutong Huang, Yilun Chen, Yiying Zhang 2018

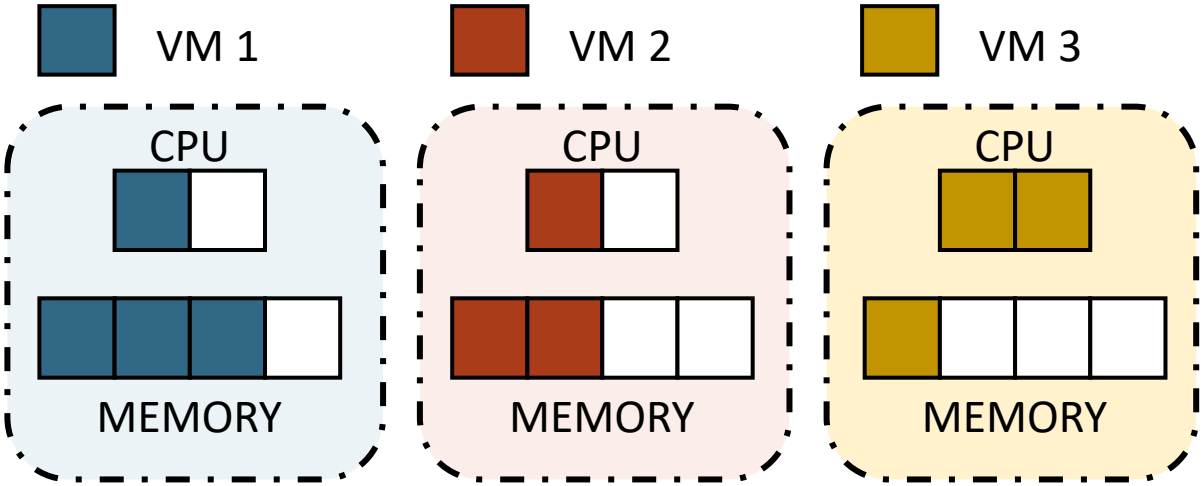
- **Yizhou Shan (Purdue, UCSD)** – PhD student
- **Yutong Huang (Purdue)** – MS student
- **Yilun Chen (Purdue)** – PhD student
- **Yiying Zhang (Purdue, UCSD)** – NSF CAREER award

OSDI'18 Best Paper

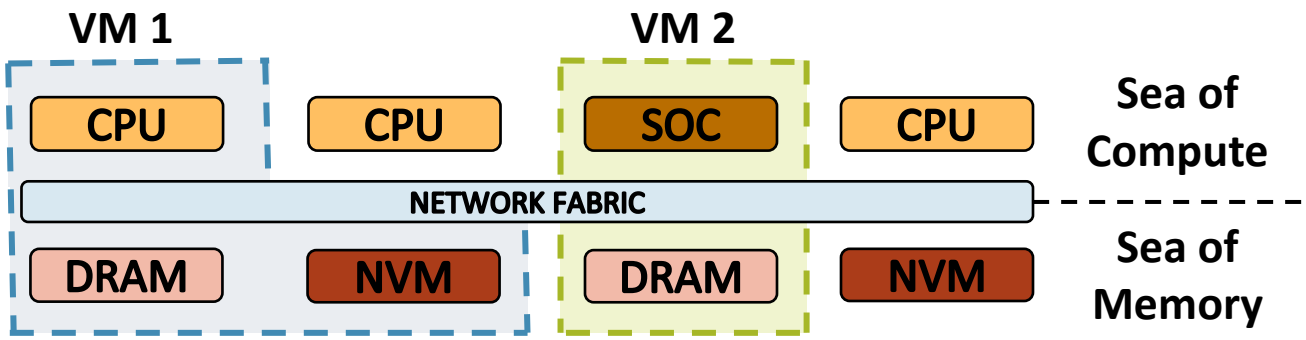

The Case for Disaggregated Resources

- **Break monolithic servers into disaggregated, network-attached hardware components (processor, DRAM, disk). Why?**
  - One size fits all => Underutilized components
  - Poor hardware elasticity
  - Coarse failure domain
  - Bad support for heterogeneity

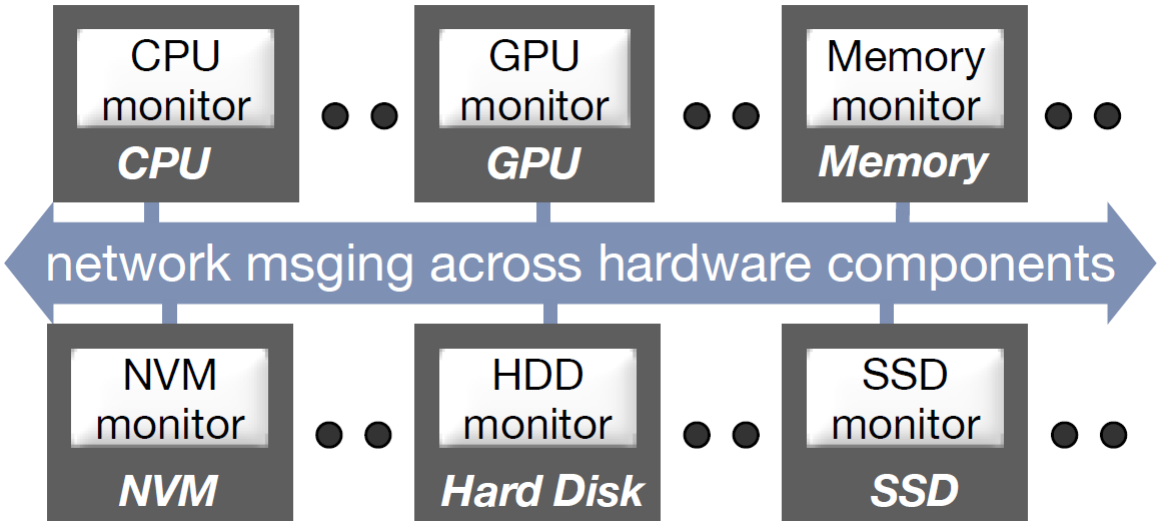


Disaggregated Resources

- Each component has its own controller, is failure-isolated

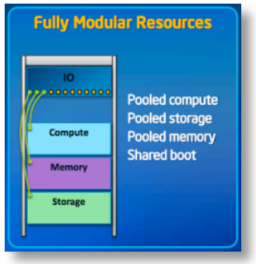


- Splitkernel Architecture: loosely-coupled monitors

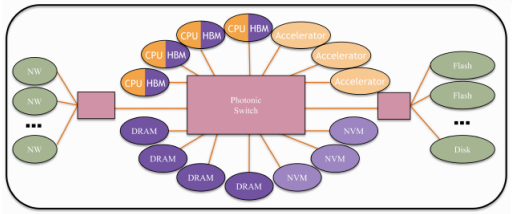


Why Possible Now

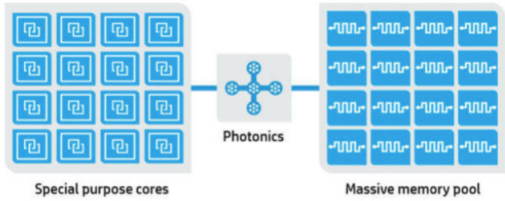
- **Network is faster**
  - InfiniBand (*200Gbps, 600ns*)
  - Optical Fabric (*400Gbps, 100ns*)
- **More processing power at device**
  - SmartNIC, SmartSSD, PIM
- **Network interface closer to device**
  - Omni-Path, Innova-2



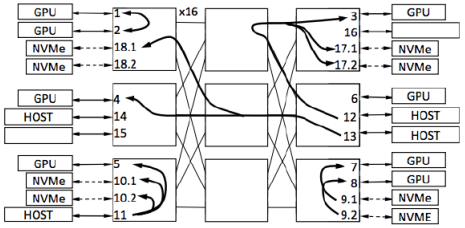
Intel Rack-Scale System



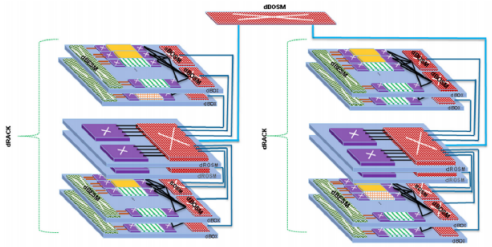
Berkeley Firebox



HP The Machine



IBM Composable System

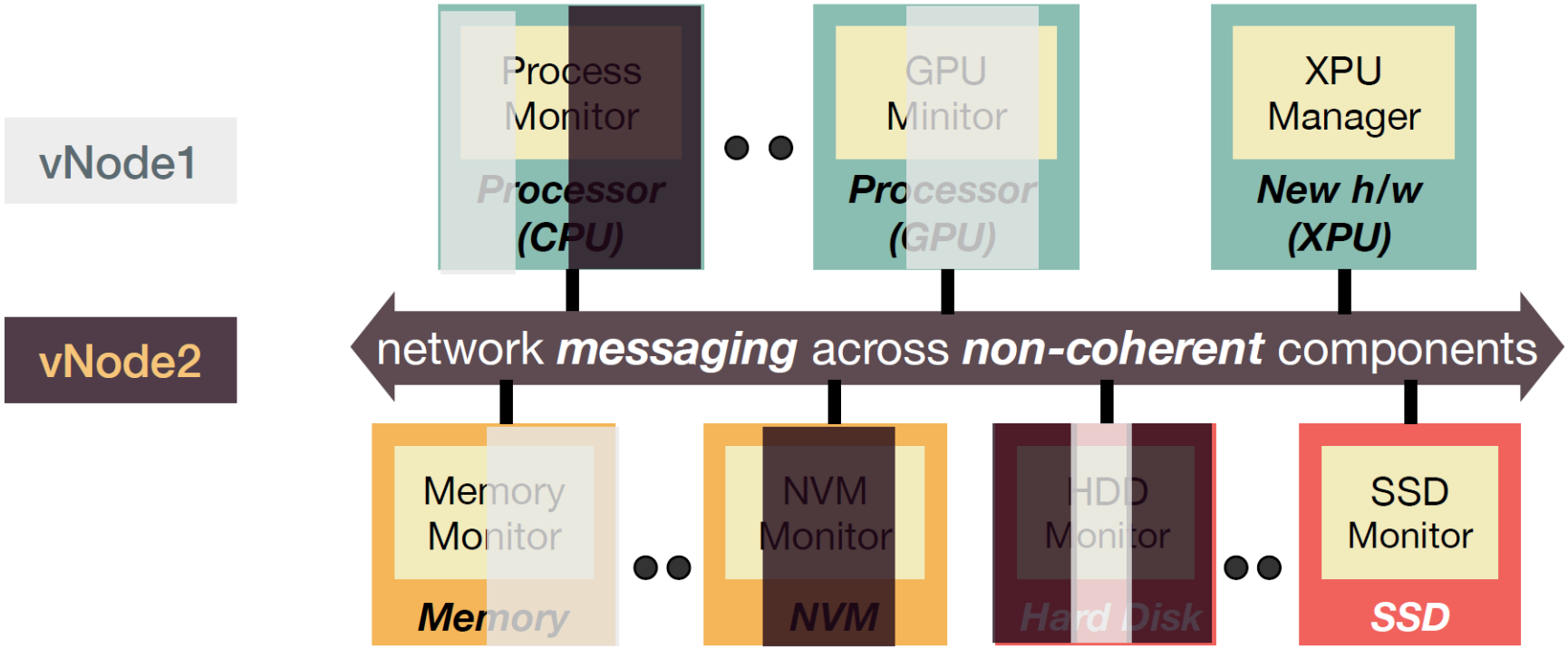


dReDBox

10

Slide from OSDI'18 talk

Abstraction - vNode

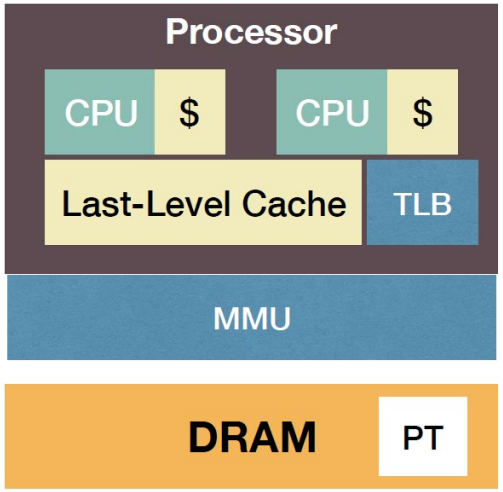


One vNode can run multiple hardware components

One hardware component can run multiple vNodes

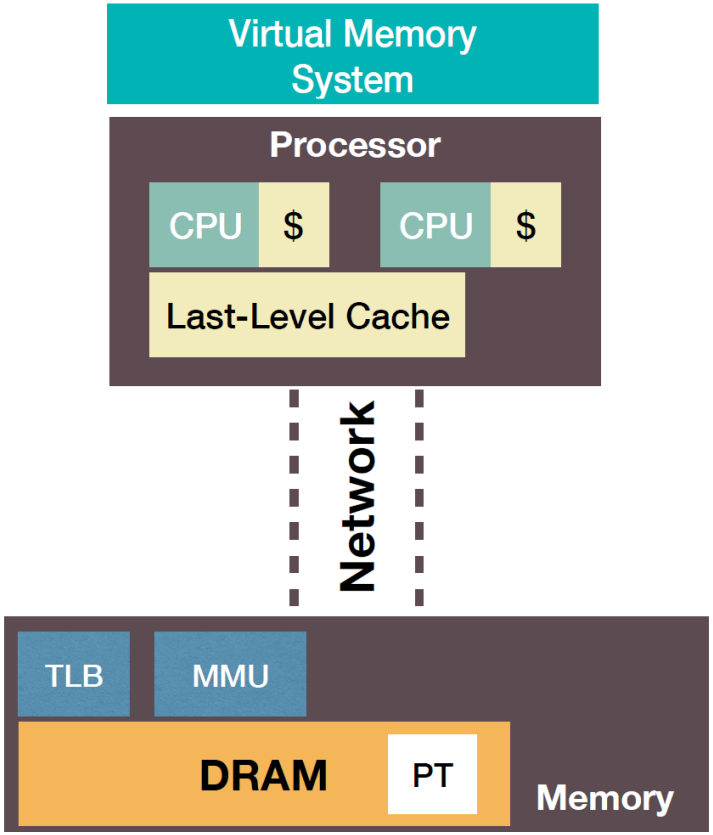
Slide from OSDI'18 talk

Separate Processor & Memory



Slide from OSDI'18 talk

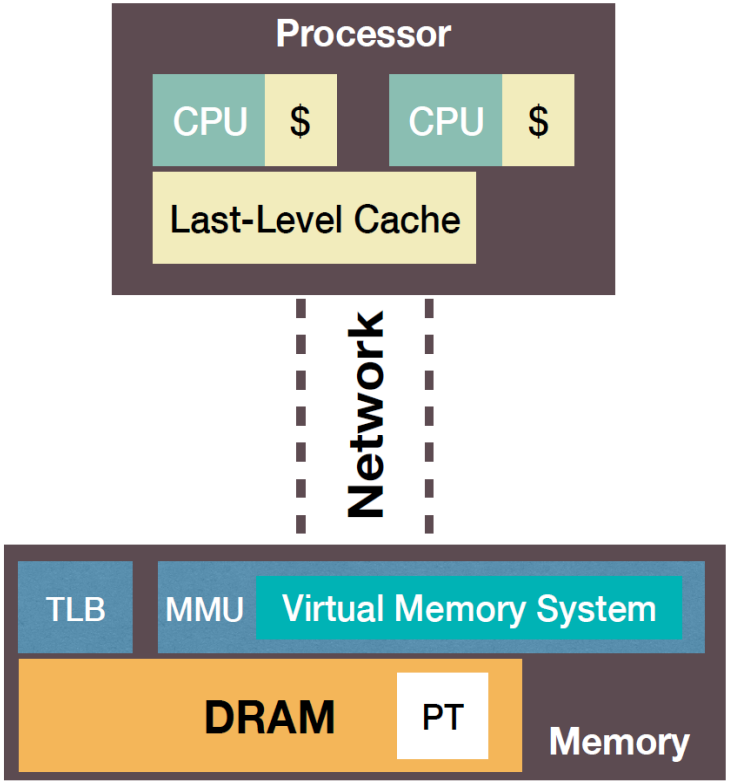
Separate Processor & Memory



Separate and move *hardware units* to memory component

Slide from OSDI'18 talk

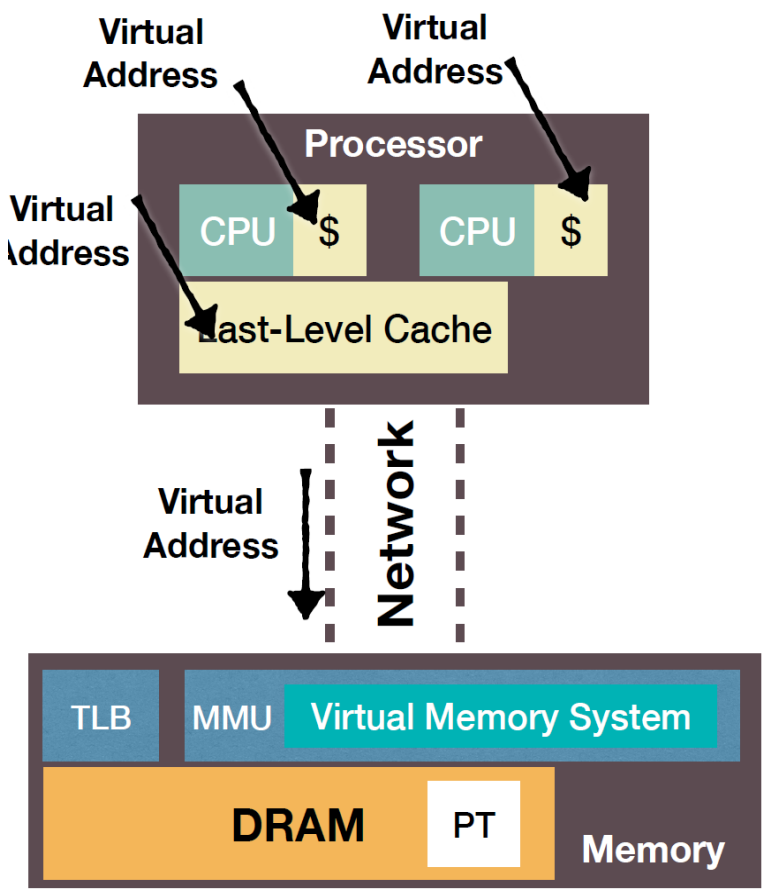
Separate Processor & Memory



Separate and move *virtual memory system* to memory component

Slide from OSDI'18 talk

Separate Processor & Memory



Processor components only see virtual memory addresses

All levels of cache are ***virtual cache***

(Add address space IDs to cache lines to disambiguate VAs)

Memory components manage virtual and physical memory

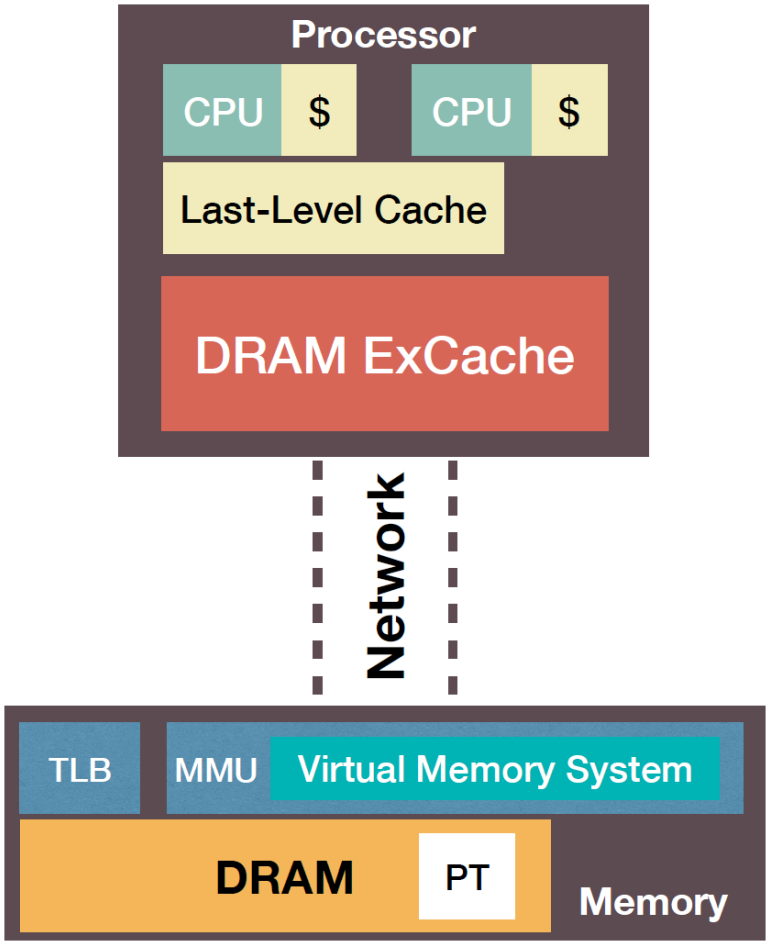
Slide from OSDI'18 talk

Challenge: Remote Memory Access

- **Network is still slower than local memory bus**
  - Bandwidth: 2x - 4x slower, improving fast
  - Latency: ~12x slower, and improving slowly

Solution?

Add Extended Cache at Processor



- Add small DRAM/HBM at processor (4 GB)
- Use it as Extended Cache, or ***ExCache***
  - Software and hardware co-managed
  - Inclusive
  - Virtual cache

Slide from OSDI'18 talk

LegoOS Design

1. Clean separation of OS and hardware functionalities
2. Build monitor with hardware constraints
3. RDMA-based message passing for both kernel and applications
4. Two-level distributed resource management (coarse and fine)
5. Memory failure tolerance through replication

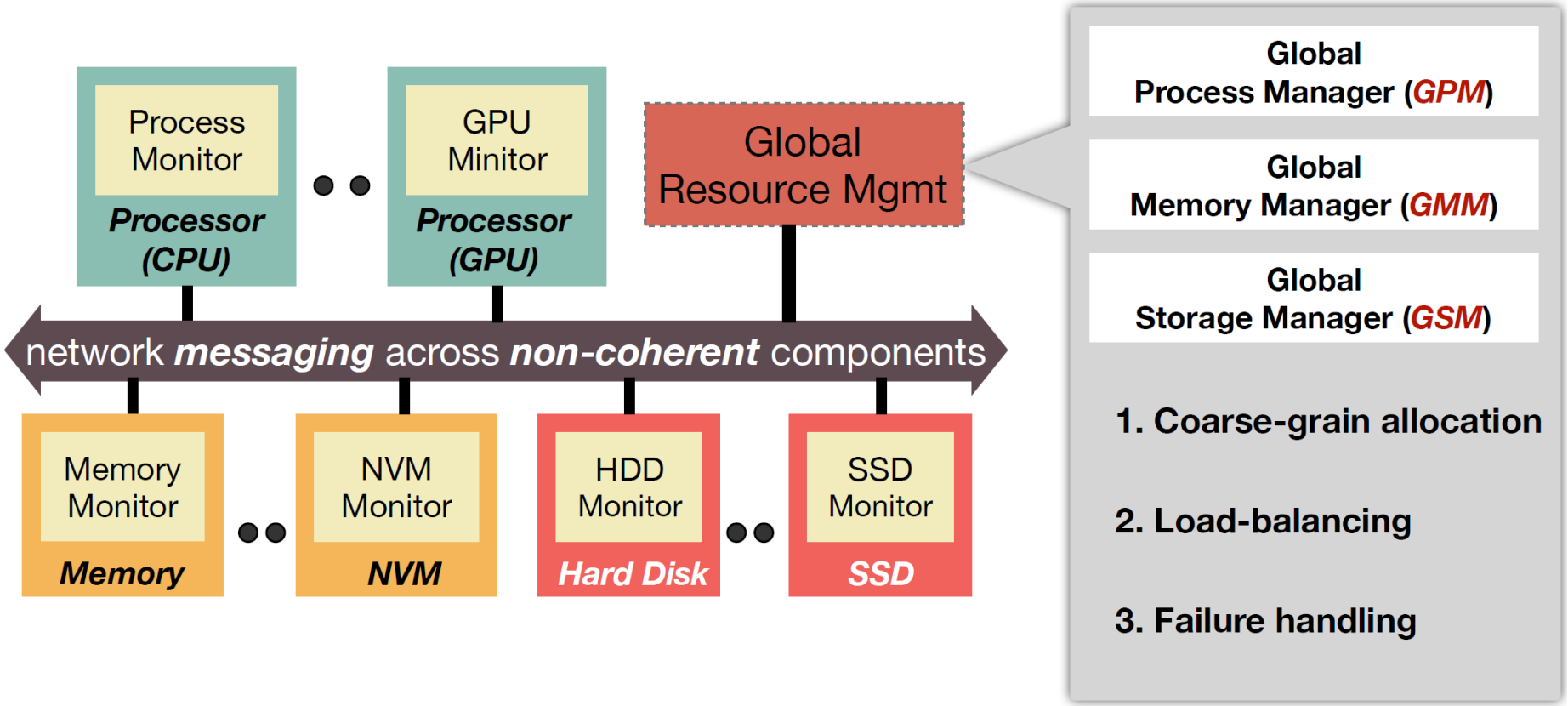
Wednesday's Class

Kernels & Virtual Machines (V)

"The Multikernel: A New OS Architecture for Scalable Multicore Systems"

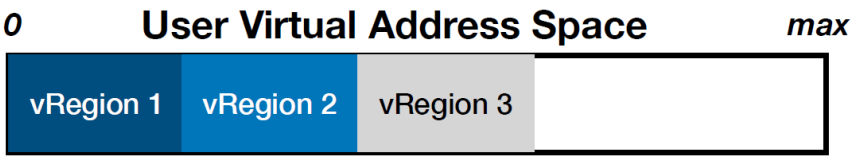
Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, Akhilesh Singhaniad 2009

Distributed Resource Management



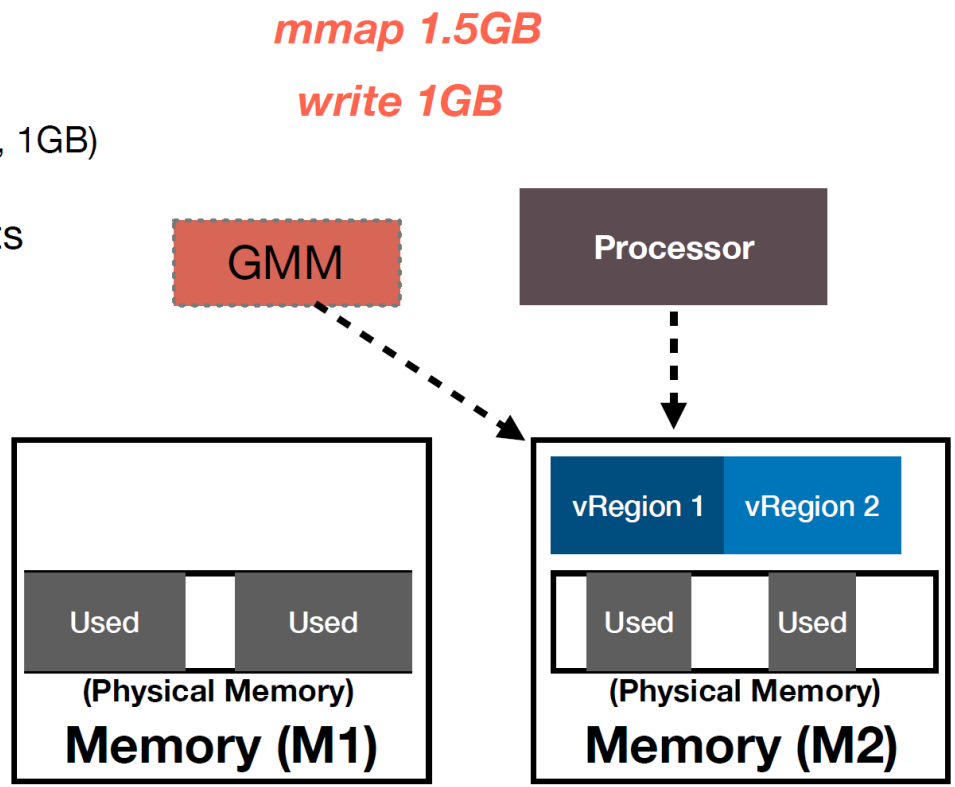
Slide from OSDI'18 talk

Distributed Memory Management



fix-sized, **coarse-grain** virtual region (***vRegion***) (e.g., 1GB)

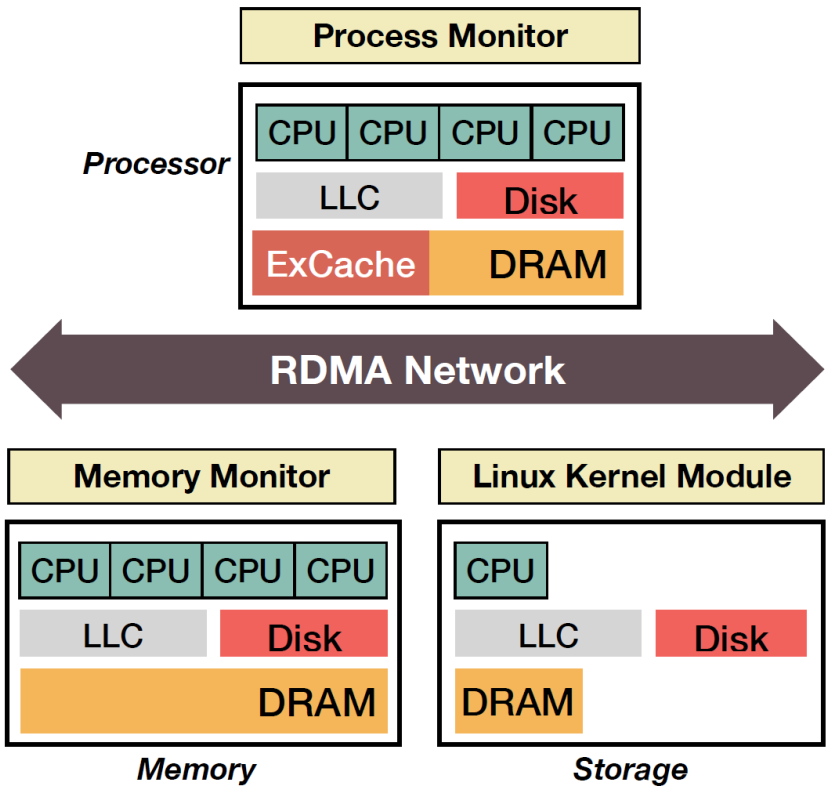
- GMM assigns vRegions to mem components
  - On virtual mem alloc syscalls (e.g., `mmap`)
  - Make decisions based on global loads
- Owner of a vRegion
  - Fine-grained virtual memory allocation
  - **On-demand** physical memory allocation
  - Handle memory accesses



Allocate non-file-backed memory in ExCache; delay physical memory allocation until write back time

Slide from OSDI'18 talk

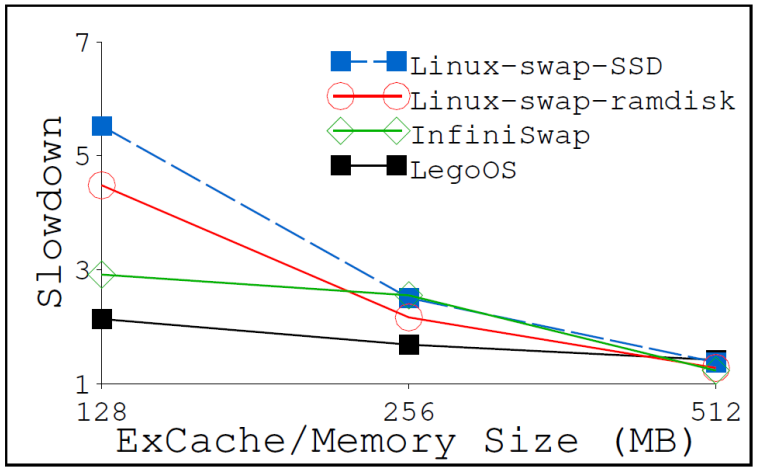
Implementation & Emulation



- **Status**
  - 206K SLOC, runs on x86-64, **113** common Linux syscalls
- **Processor**
  - Reserve DRAM as ExCache (4KB page as cache line)
  - h/w only on hit path, s/w managed miss path
- **Memory**
  - Limit number of cores, kernel-space only
- **Storage/Global Resource Monitors**
  - Implemented as kernel modules on Linux
- **Network**
  - RDMA RPC stack based on LITE [*SOSP'17*]

Slide from OSDI'18 talk

Performance Evaluation



LegoOS Config: 1P, 1M, 1S

- Unmodified TensorFlow, running CIFAR-10
  - Working set: 0.9G
  - 4 threads
- Systems in comparison
  - Baseline: Linux with unlimited memory

Only 1.3x to 1.7x slowdown when disaggregating devices with LegoOS

To gain better resource packing, elasticity, and fault tolerance!

Slide from OSDI'18 talk