Kernel IO Optimization

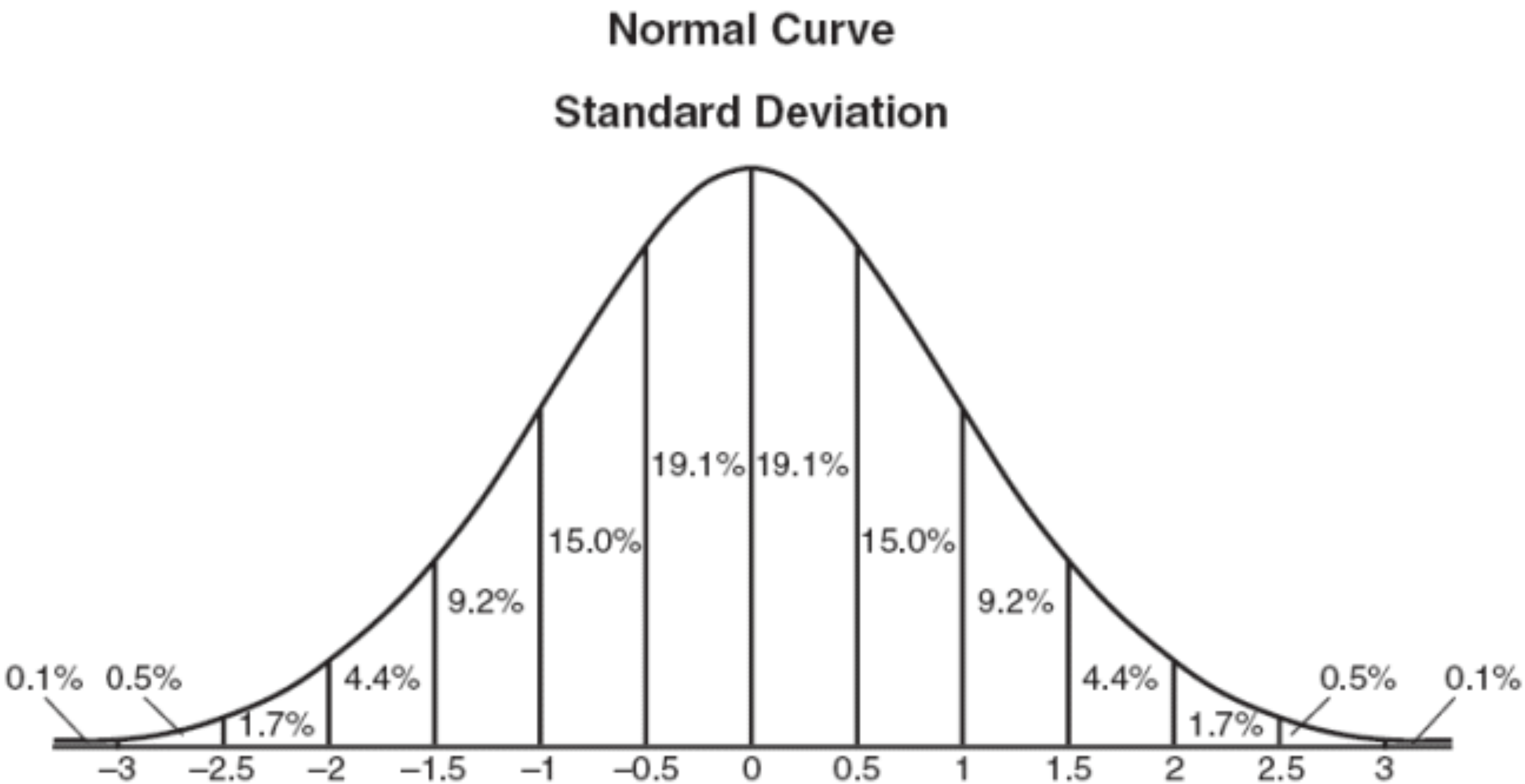
Scott Constable

Overview

1. Background
2. Problem Statement
3. More Background
4. Arrakis OS
5. Comparison with IX OS

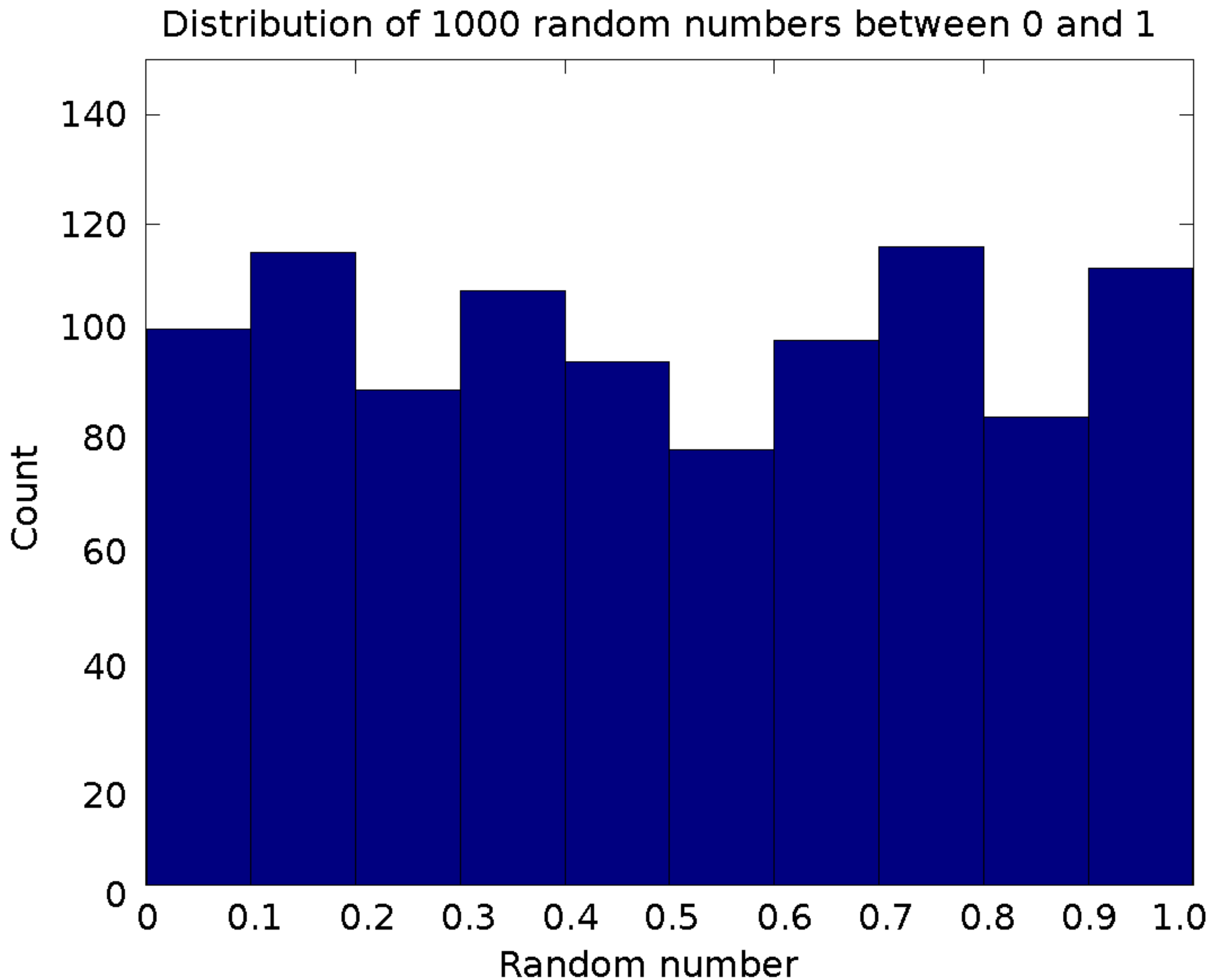
Background

Background



Normal Curve
Standard Deviation
19.1%
19.1%
15.0%
15.0%
9.2%
9.2%
0.1%
0.5%
4.4%
1.7%
4.4%
1.7%
0.5%
0.1%
–3
–2.5
–2
–1.5
–1
–0.5
0
0.5
1
1.5
2
2.5
3

Background



Distribution of 1000 random numbers between 0 and 1
Count
Random number
0
20
40
60
80
100
120
140
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0

Background

Background

1. Traditional
2. Zesty
3. Chunky

Background


Prego
Three Cheese
Prego
Fresh
Mushroom
$2.49
GF
www.MyPublixCouponing101.com
Prego
Flavored with
Meat
Prego
HEART
Smart
Traditional
Prego
Traditional
$2.49

Background

"You're not making the wrong pasta sauce…"

"You're making the wrong pasta sauce**<u>s</u>**!"

Background


pepsi
PEPSI
7up
Tropicana
Gatorade
Propel
MUG
ROOT BEER
STING
Lipton
MAX

Background

?

Background

1994: The Microkernel Dilemma

“Who killed the microkernel???”

Background

1995: The Microkernel Revelation

“Who **saved** the microkernel???”

Problem

2014: “Tuned” Linux kernels dominate cloud server architectures, yet they do not come close to achieving the theoretical optimum performance for a common bottleneck, namely I/O.

Problem

Traditional OS assumptions:

- many small applications share few processing cores
- applications exhibit a wide variety of behavior
- can’t rely on hardware to arbitrate I/O isolation among processes

Problem

Server OS assumptions:

- few large applications share many processing cores
- applications exhibit simple and predictable behavior
- modern virtualization hardware is increasingly flexible

Problem

Given this mismatch, why would we still use a commodity OS on servers?

If we instead build the right OSes, we can take full advantage of a server's hardware, and the knowledge of what software that server is running.

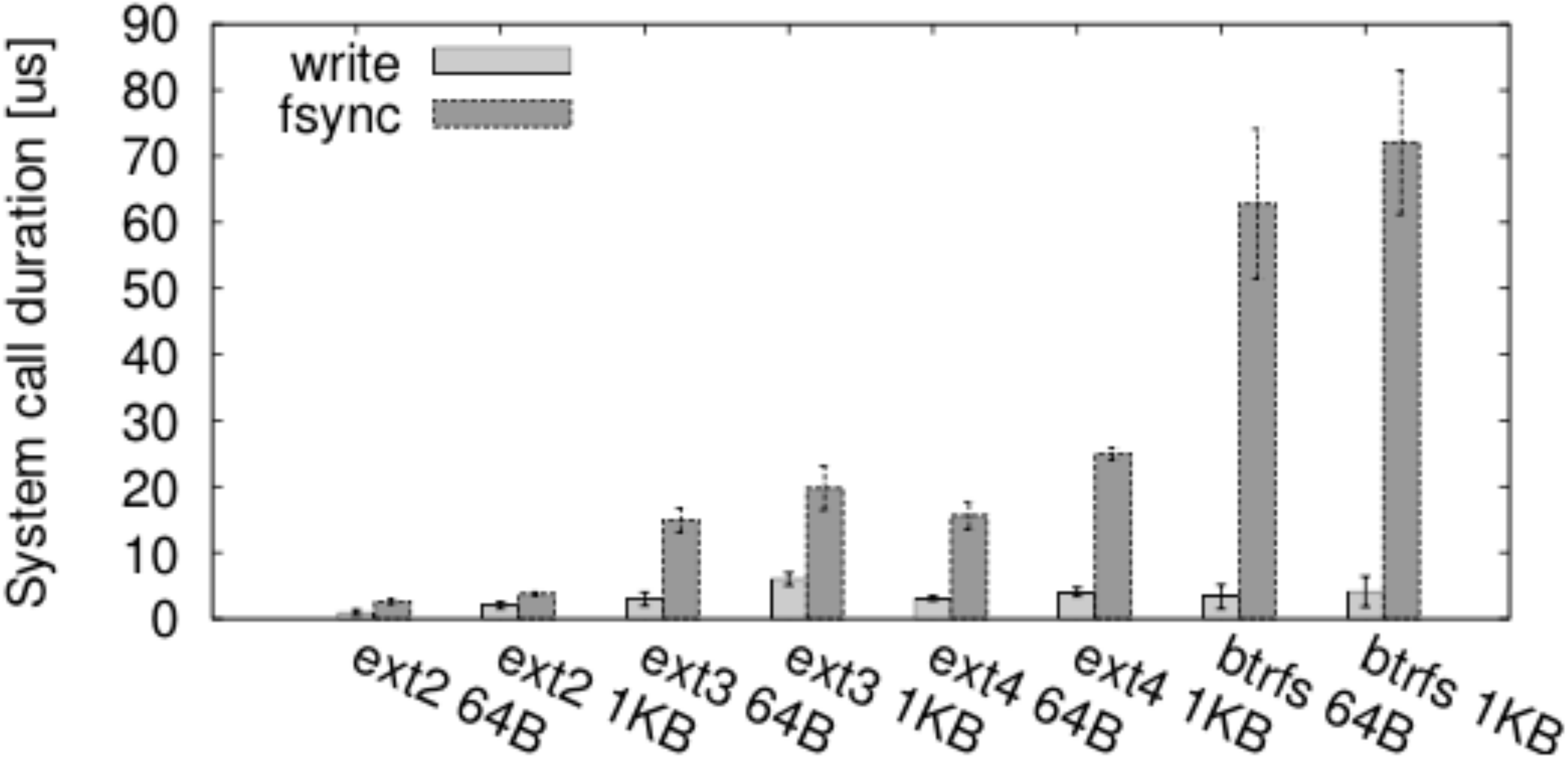
Problem

		Linux			
		Receiver running		CPU idle	
Network stack	in	1.26	(37.6%)	1.24	(20.0%)
	out	1.05	(31.3%)	1.42	(22.9%)
Scheduler		0.17	(5.0%)	2.40	(38.8%)
Copy	in	0.24	(7.1%)	0.25	(4.0%)
	out	0.44	(13.2%)	0.55	(8.9%)
Kernel crossing	return	0.10	(2.9%)	0.20	(3.3%)
	syscall	0.10	(2.9%)	0.13	(2.1%)
Total		3.36	($\sigma = 0.66$)	6.19	($\sigma = 0.82$)

Problem

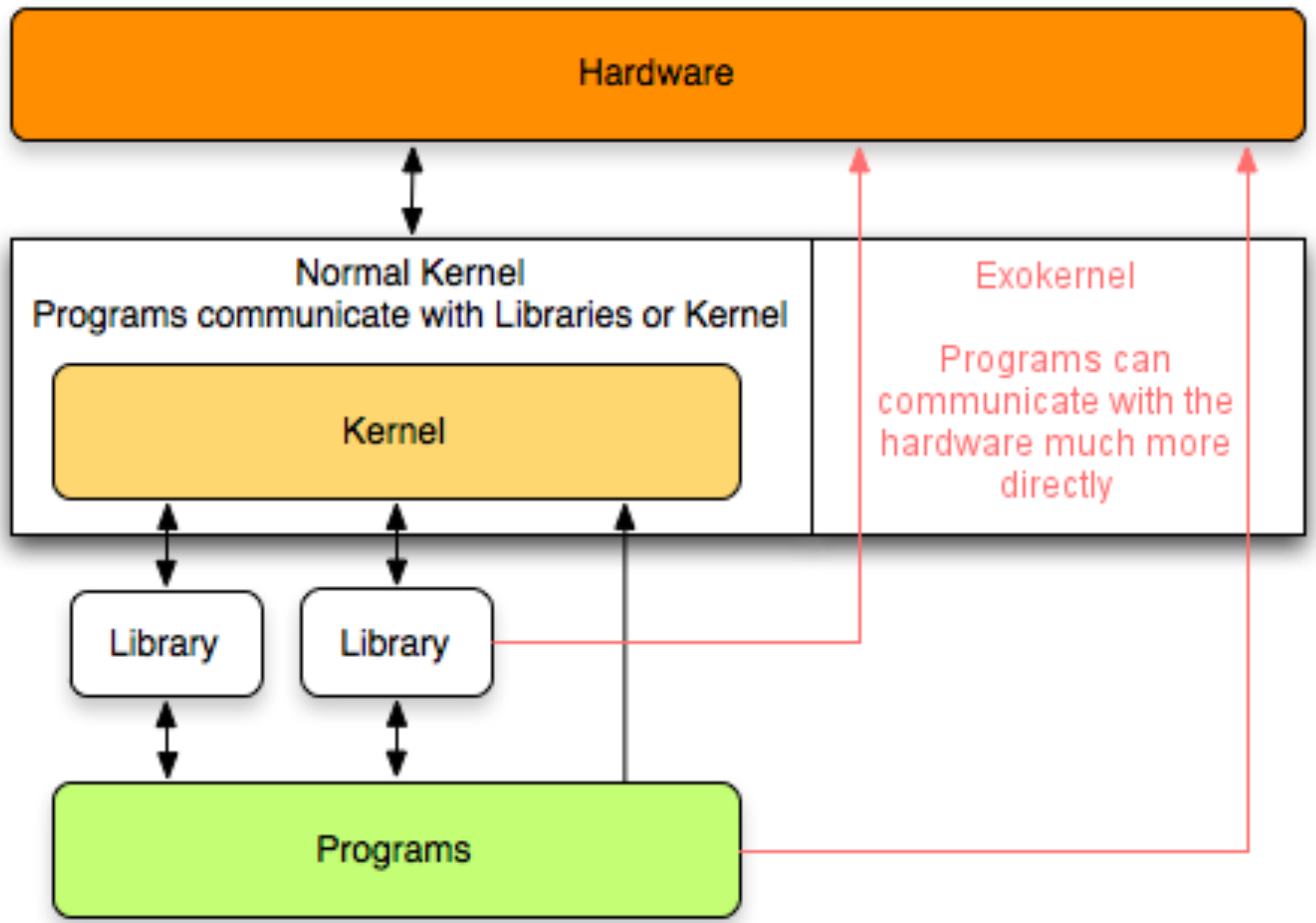
	Read		Write	
	Linux		Linux	
epoll()	2.42	(27.91%)	2.64	(1.62%)
recv()	0.98	(11.30%)	1.55	(0.95%)
send()	3.17	(36.56%)	5.06	(3.10%)
Parse input	0.85	(9.80%)	2.34	(1.43%)
Lookup/set key	0.10	(1.15%)	1.03	(0.63%)
Prepare response	0.60	(6.92%)	0.59	(0.36%)
Log marshaling	-		3.64	(2.23%)
Write log	-		6.33	(3.88%)
Persistence	-		137.84	(84.49%)
Other	0.55	(6.34%)	2.12	(1.30%)
Total	8.67	($\sigma = 2.55$)	163.14	($\sigma = 13.68$)

Problem



System call duration [us]
90
80
70
60
50
40
30
20
10
0
write
fsync
ext2 64B
ext2 1KB
ext3 64B
ext3 1KB
ext4 64B
ext4 1KB
btrfs 64B
btrfs 1KB

Background: ExoKernel



Background: ExoKernel

Advantages:

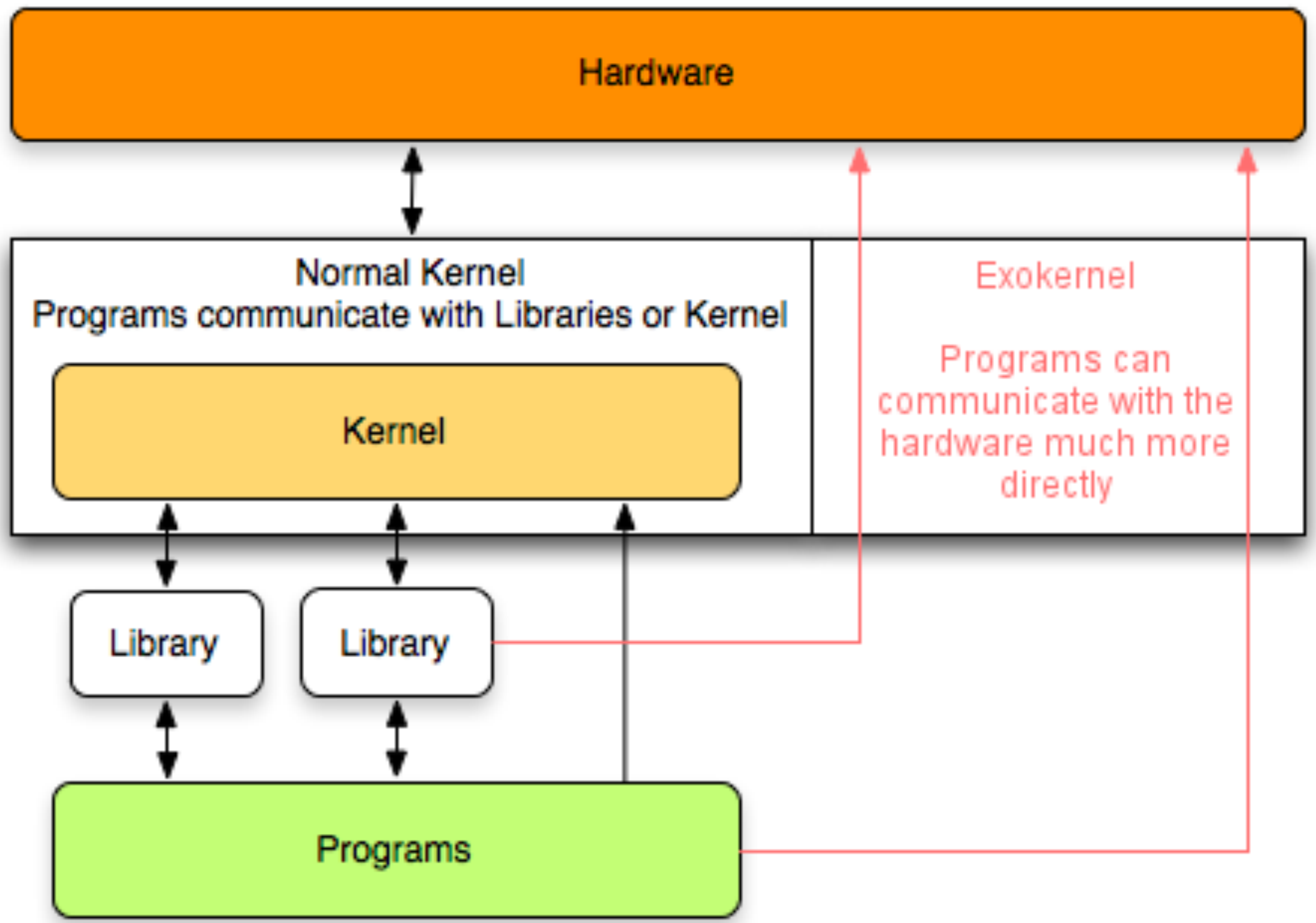
- programmer can implement custom abstractions
- programmer can omit unnecessary abstractions
- substantially less kernel overhead
- performance++

Background: ExoKernel

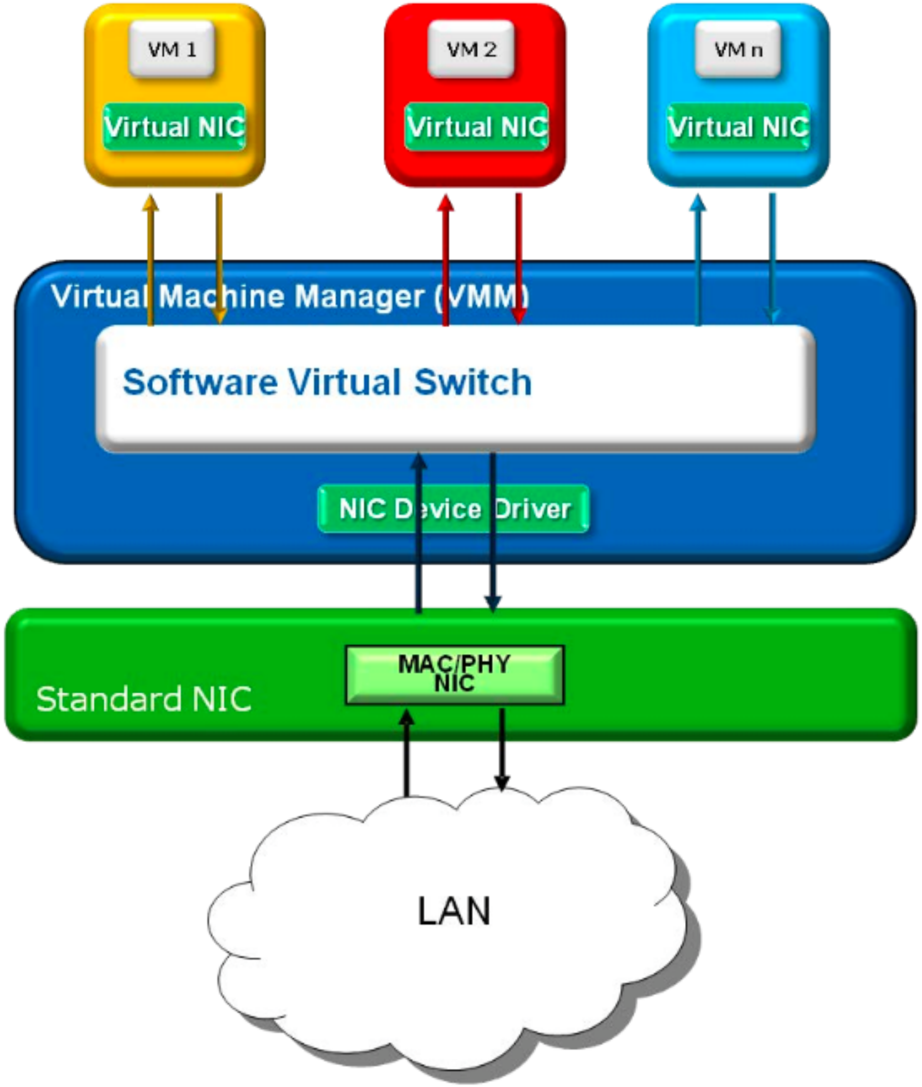
Disadvantages:

- True isolation is impossible
- OS arbitration is difficult

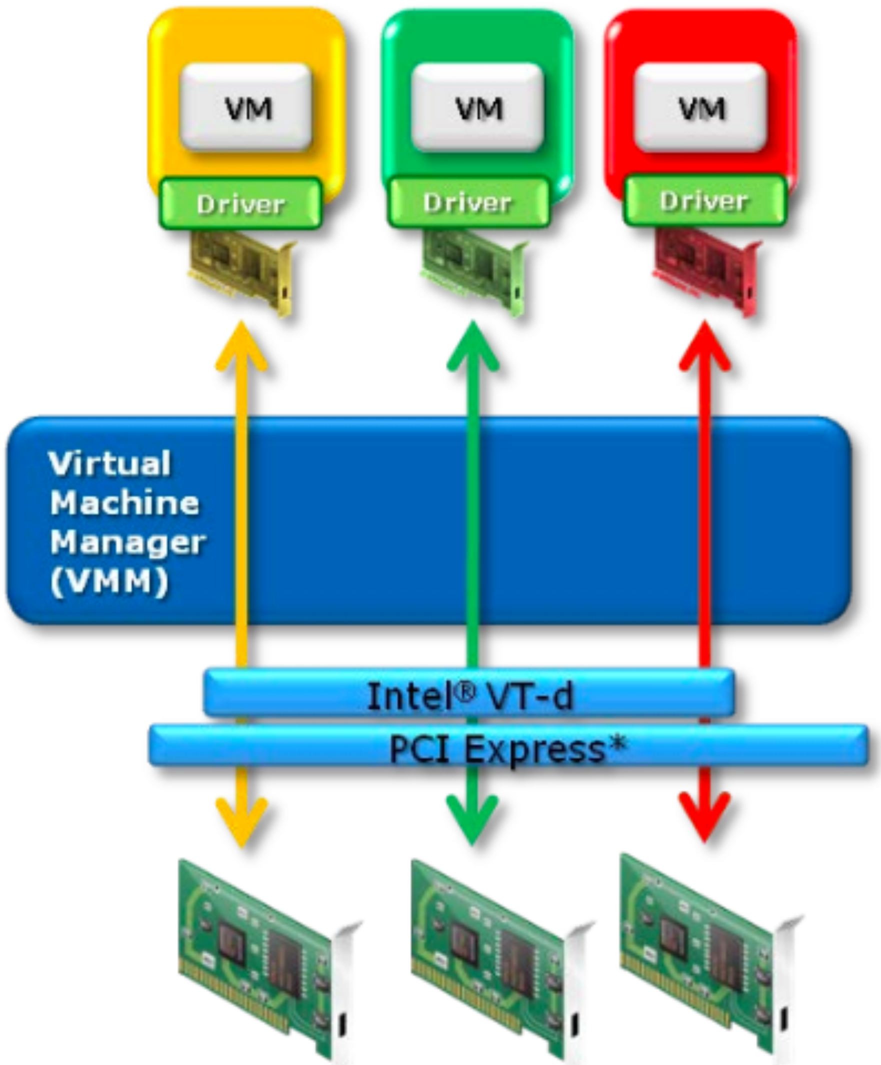
Background: ExoKernel



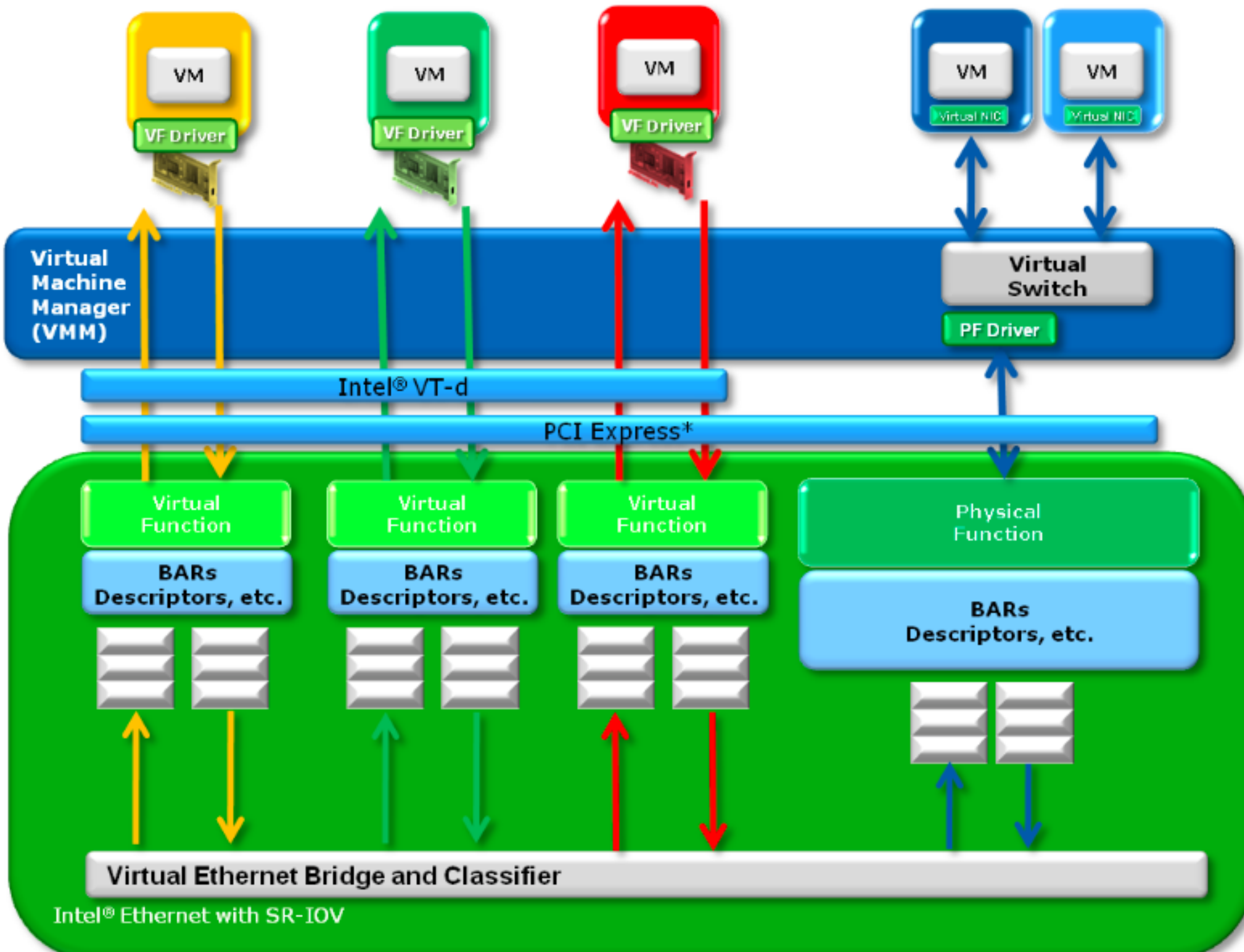
Background: Hardware



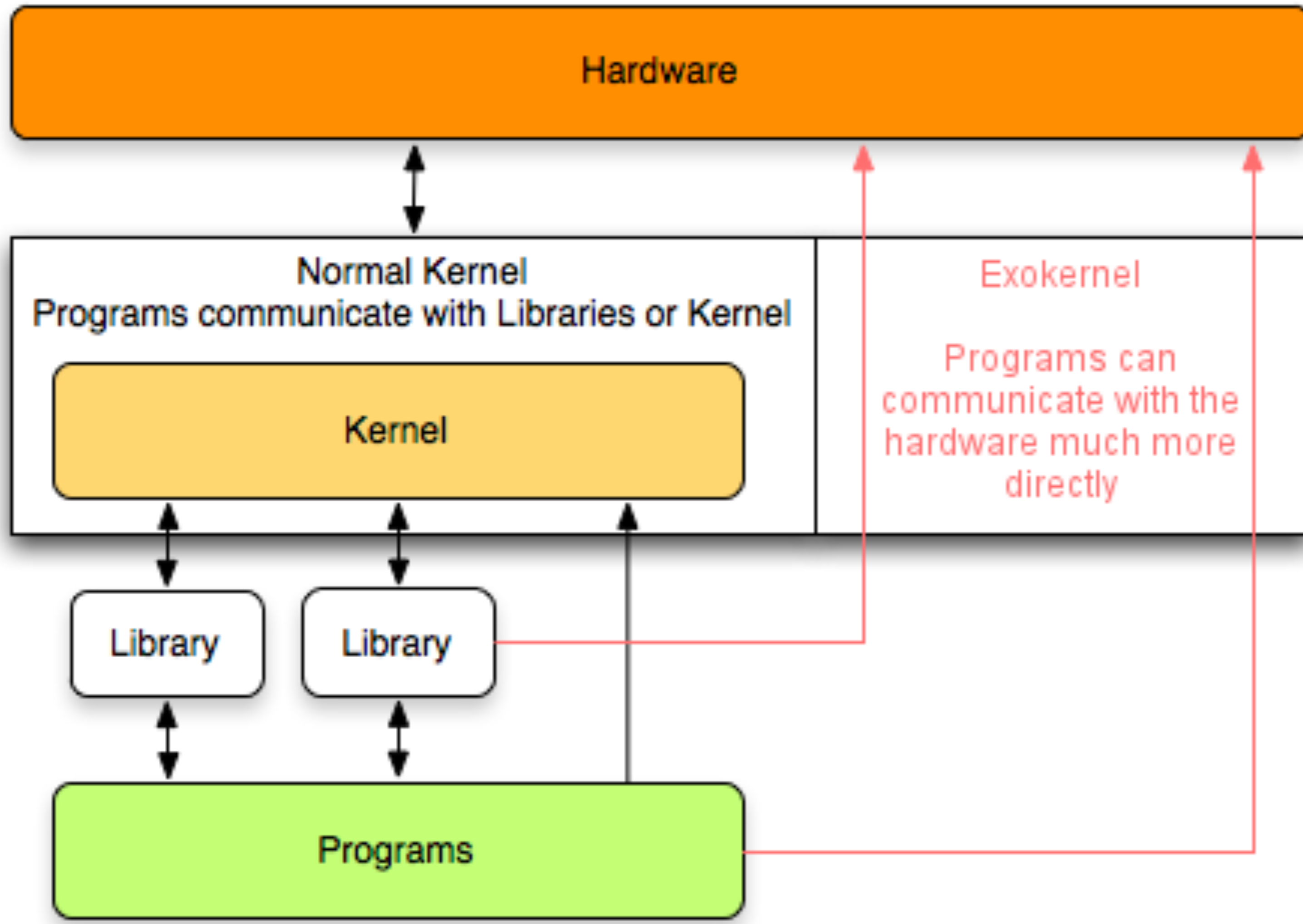
Background: Hardware



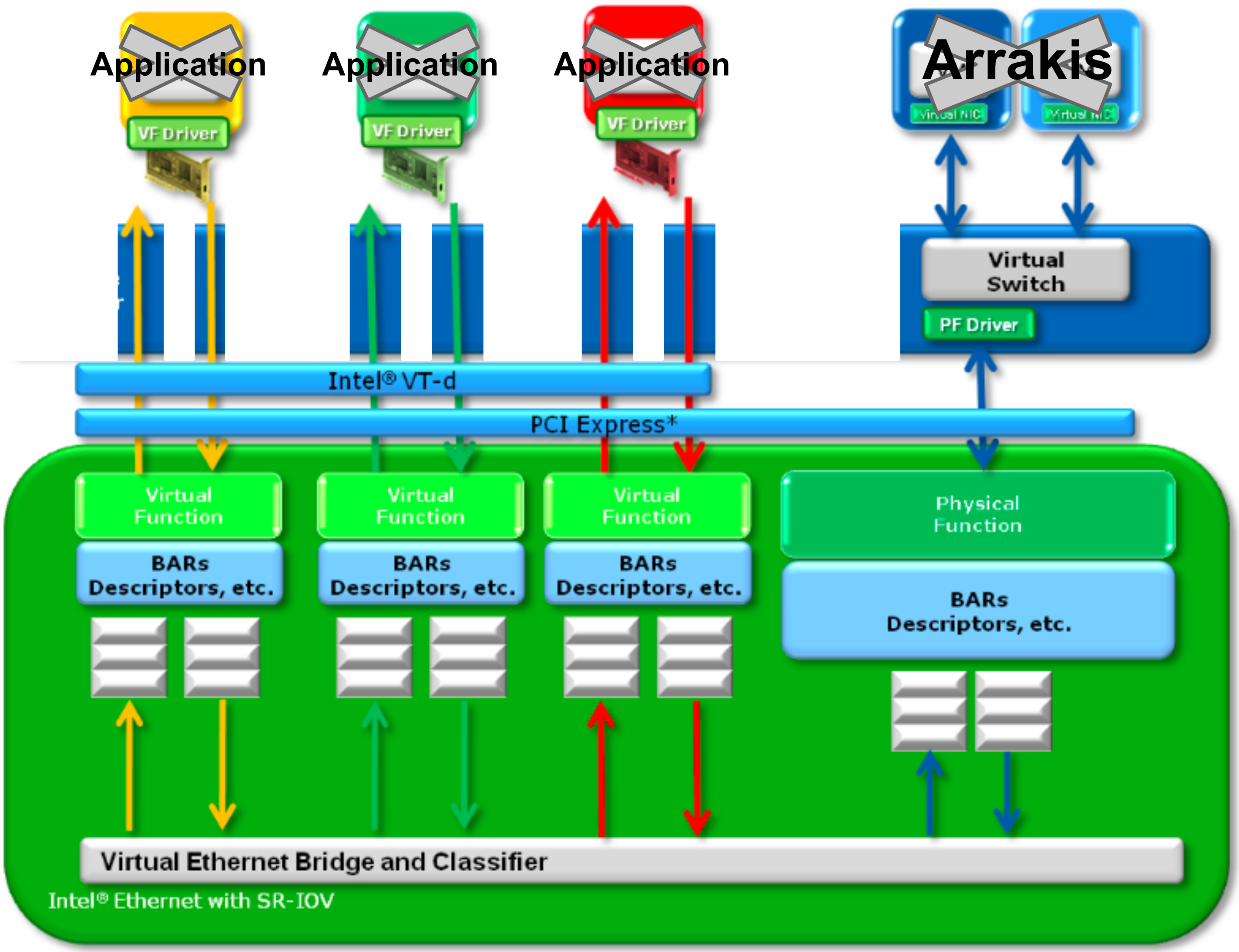
Background: Hardware



Background: ExoKernel

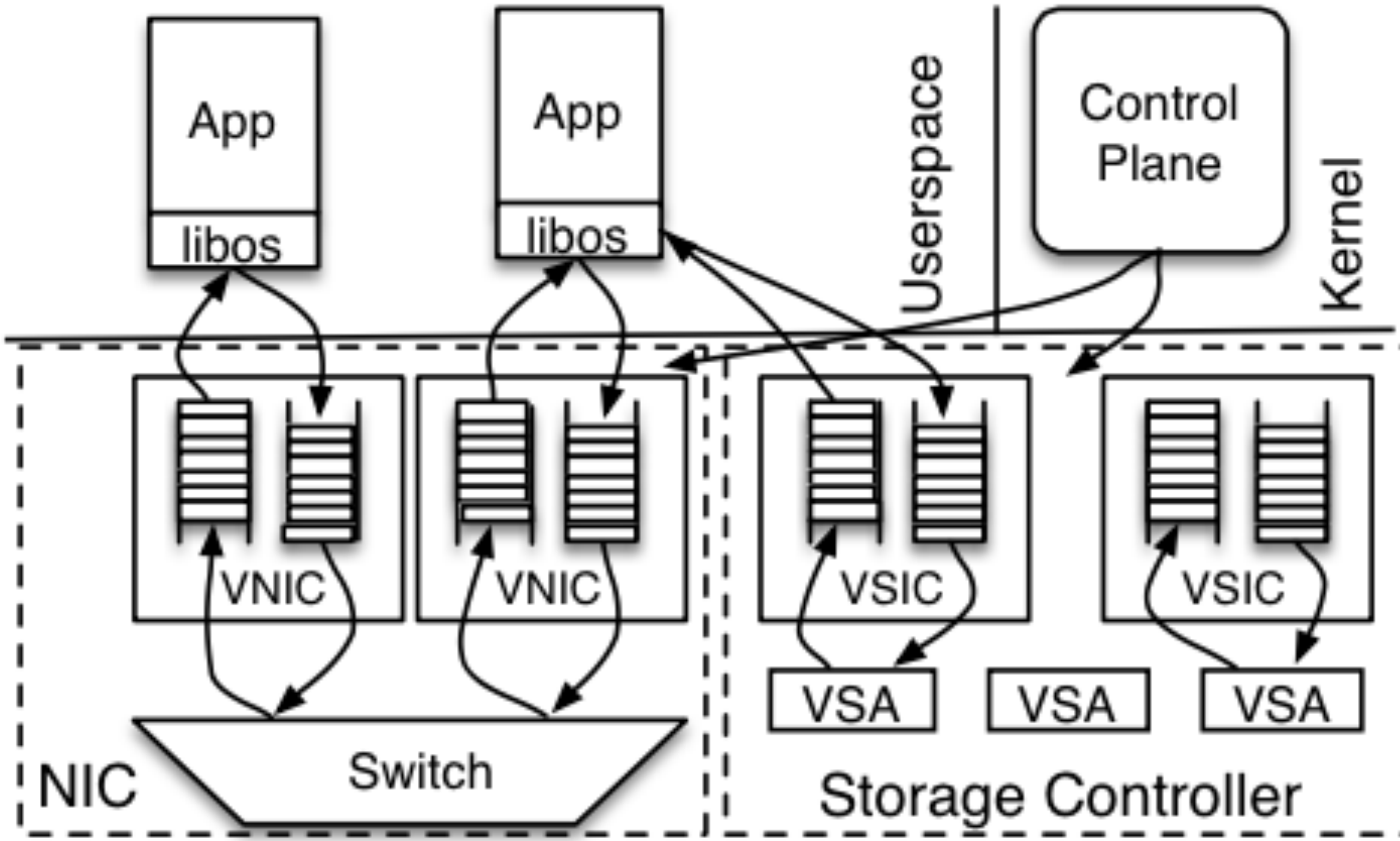


Arrakis!



Application
VF Driver
Application
VF Driver
Application
VF Driver
Arrakis
Virtual NIC
Virtual NIC
Virtual Switch
PF Driver
Intel® VT-d
PCI Express*
Virtual Function
BARs Descriptors, etc.
Virtual Function
BARs Descriptors, etc.
Virtual Function
BARs Descriptors, etc.
Physical Function
BARs Descriptors, etc.
Virtual Ethernet Bridge and Classifier
Intel® Ethernet with SR-IOV

Arrakis (again)!



Arrakis

Question: How do we solve the isolation and arbitration problems?

Answer: Hardware-based <u>capabilities</u>

Example: Network

1. Application X calls
   filter = create_filter(flags, peerlist, servicelist)
2. Control plane (Arrakis) is triggered
3. Arrakis creates a filter “capability” by configuring (via the PF) X’s VF to allow communication according to the specified filter
4. Arrakis returns the filter “pointer” to X
5. X assigns filter to a new network queue on its VNIC

Example: Storage

1. Application Y calls VSA = acquire_vsa(name)
2. Control plane (Arrakis) is triggered
3. Arrakis creates a VSA “capability” by setting up an entry in kernel memory containing a mapping of virtual storage blocks to physical ones
4. Arrakis configures Y’s VSIC to map in the new VSA area
5. Arrakis returns the VSA capability to Y
6. Y can then call resize_VSA(VSA, size)
7. Arrakis checks whether it can satisfy the request, then updates the mapping and hardware as needed

Doorbells

- Arrakis's way of handling asynchronous events
- Each queue has an associated doorbell
- When one of X's events is triggered, and X is running, the doorbell is a hardware-virtualized interrupt directly to X
- When one of X's events is triggered, and X is not running, the doorbell triggers a kernel interrupt, prompting the scheduler to switch to X
- Exposed to user applications like file descriptors, i. e. they can be polled by select()

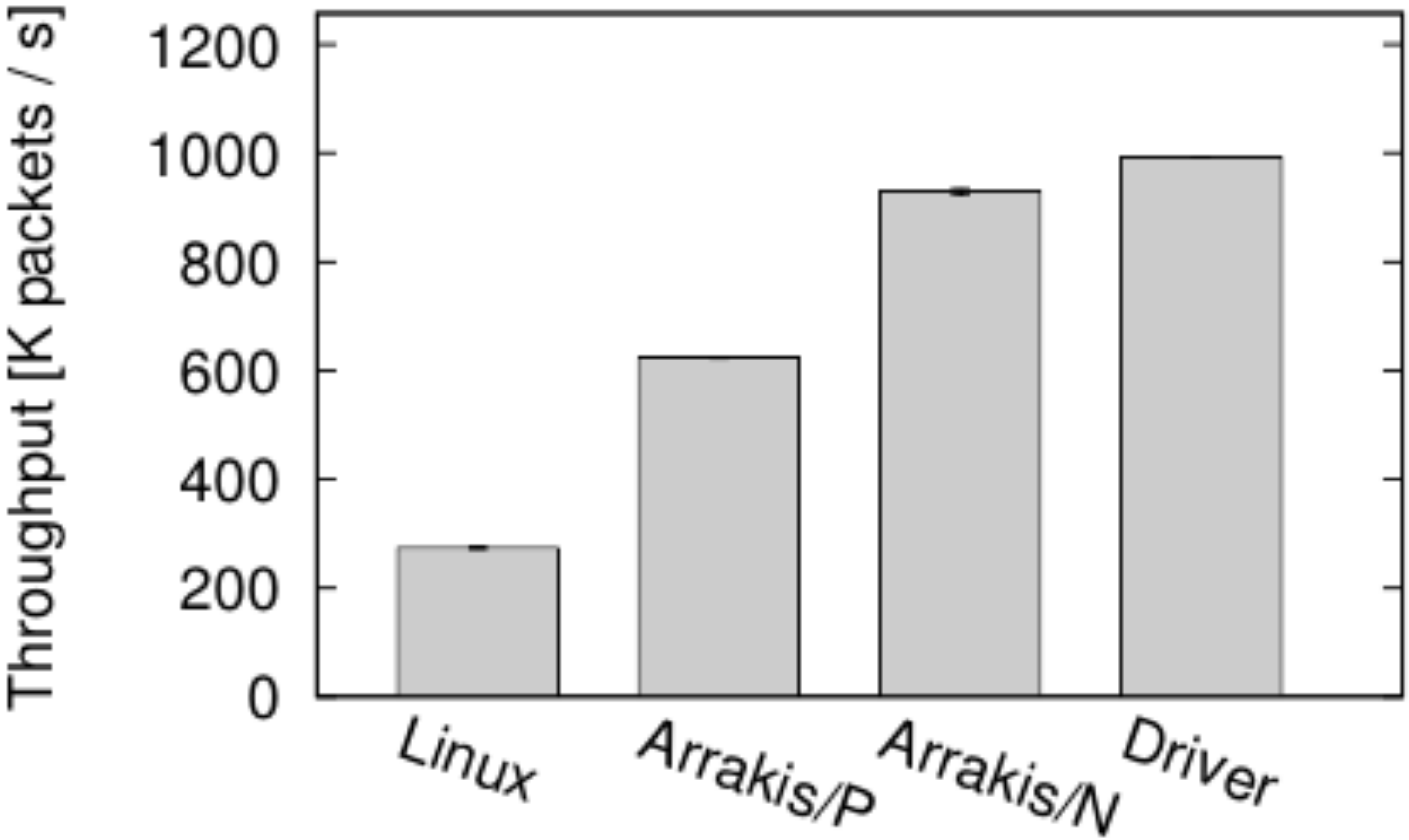
TenaciousD

- Persistent data structure framework to allow asynchronous persistent writes
- Operations are essentially “immediately persistent”
- Structure is robust to crash failures
- Operations have minimal latency
- Asynchronous API returns immediately, and may callback once the data is actually persistent
- Arrakis group modified Redis NoSQL to use TenaciousD

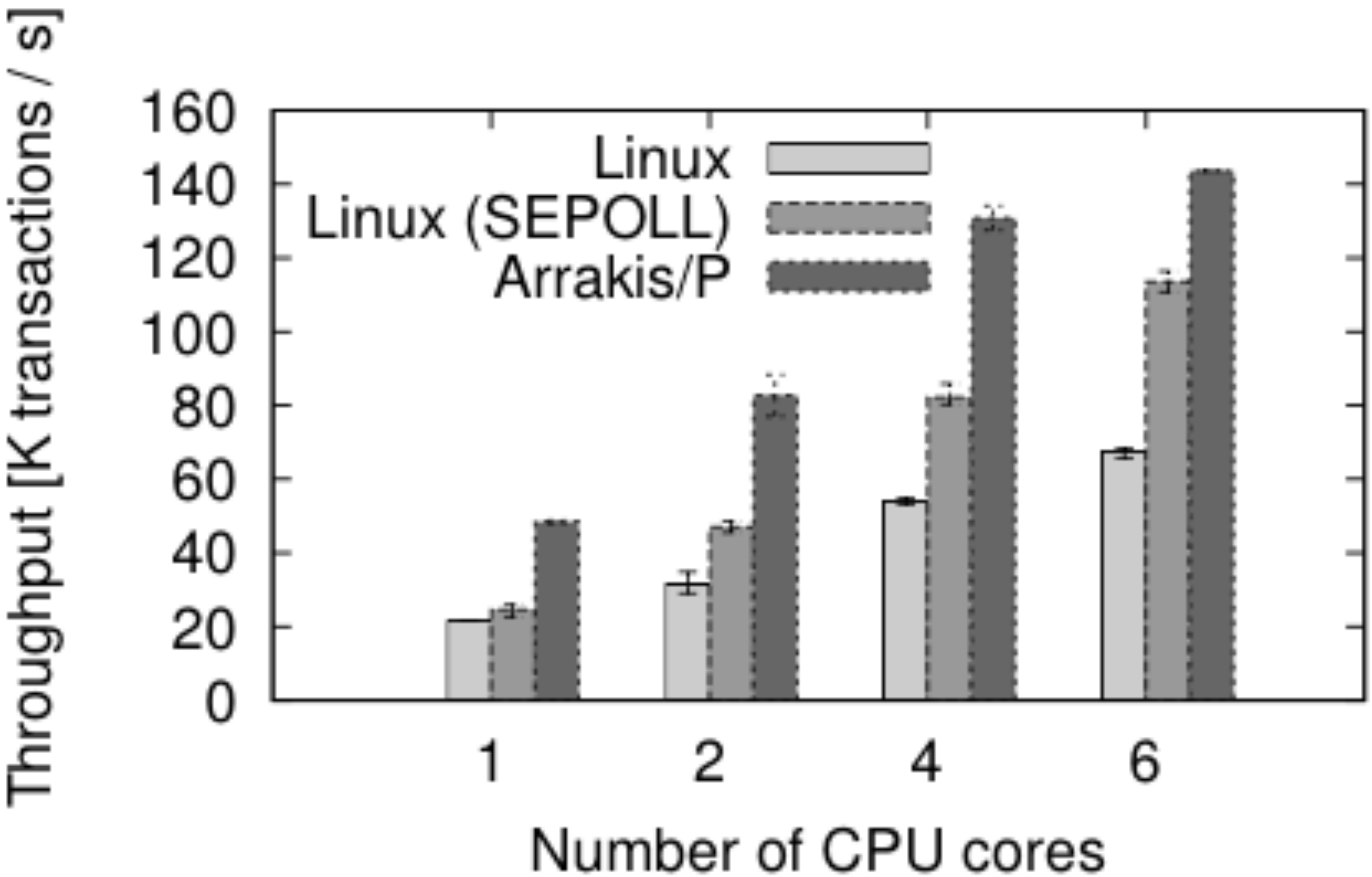
Performance (Network)

		Linux				Arrakis			
		Receiver running		CPU idle		POSIX interface		Native interface	
Network stack	in	1.26	(37.6%)	1.24	(20.0%)	0.32	(22.3%)	0.21	(55.3%)
	out	1.05	(31.3%)	1.42	(22.9%)	0.27	(18.7%)	0.17	(44.7%)
Scheduler		0.17	(5.0%)	2.40	(38.8%)	-		-	
Copy	in	0.24	(7.1%)	0.25	(4.0%)	0.27	(18.7%)	-	
	out	0.44	(13.2%)	0.55	(8.9%)	0.58	(40.3%)	-	
Kernel crossing	return	0.10	(2.9%)	0.20	(3.3%)	-		-	
	syscall	0.10	(2.9%)	0.13	(2.1%)	-		-	
Total		3.36	($\sigma = 0.66$)	6.19	($\sigma = 0.82$)	1.44	($\sigma < 0.01$)	0.38	($\sigma < 0.01$)

Performance (Network)



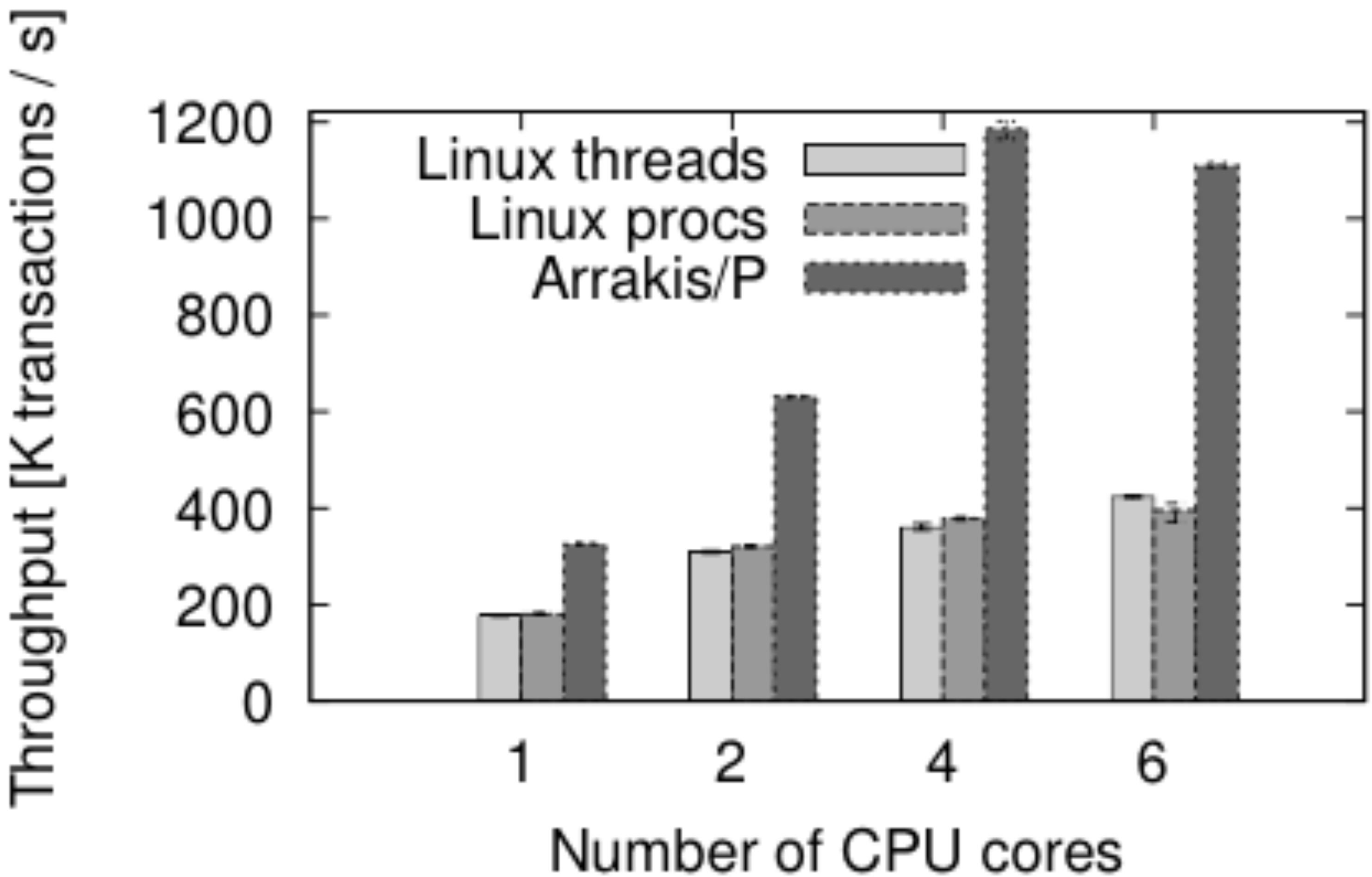
Performance (Network)



Performance (Storage)

	Read hit				Durable write			
	Linux		Arrakis/P		Linux		Arrakis/P	
epoll()	2.42	(27.91%)	1.12	(27.52%)	2.64	(1.62%)	1.49	(4.73%)
recv()	0.98	(11.30%)	0.29	(7.13%)	1.55	(0.95%)	0.66	(2.09%)
send()	3.17	(36.56%)	0.71	(17.44%)	5.06	(3.10%)	0.33	(1.05%)
Parse input	0.85	(9.80%)	0.66	(16.22%)	2.34	(1.43%)	1.19	(3.78%)
Lookup/set key	0.10	(1.15%)	0.10	(2.46%)	1.03	(0.63%)	0.43	(1.36%)
Prepare response	0.60	(6.92%)	0.64	(15.72%)	0.59	(0.36%)	0.10	(0.32%)
Log marshaling	-		-		3.64	(2.23%)	2.43	(7.71%)
Write log	-		-		6.33	(3.88%)	0.10	(0.32%)
Persistence	-		-		137.84	(84.49%)	24.26	(76.99%)
Other	0.55	(6.34%)	0.46	(11.30%)	2.12	(1.30%)	0.52	(1.65%)
Total	8.67	($\sigma = 2.55$)	4.07	($\sigma = 0.44$)	163.14	($\sigma = 13.68$)	31.51	($\sigma = 1.91$)

Performance (Storage)

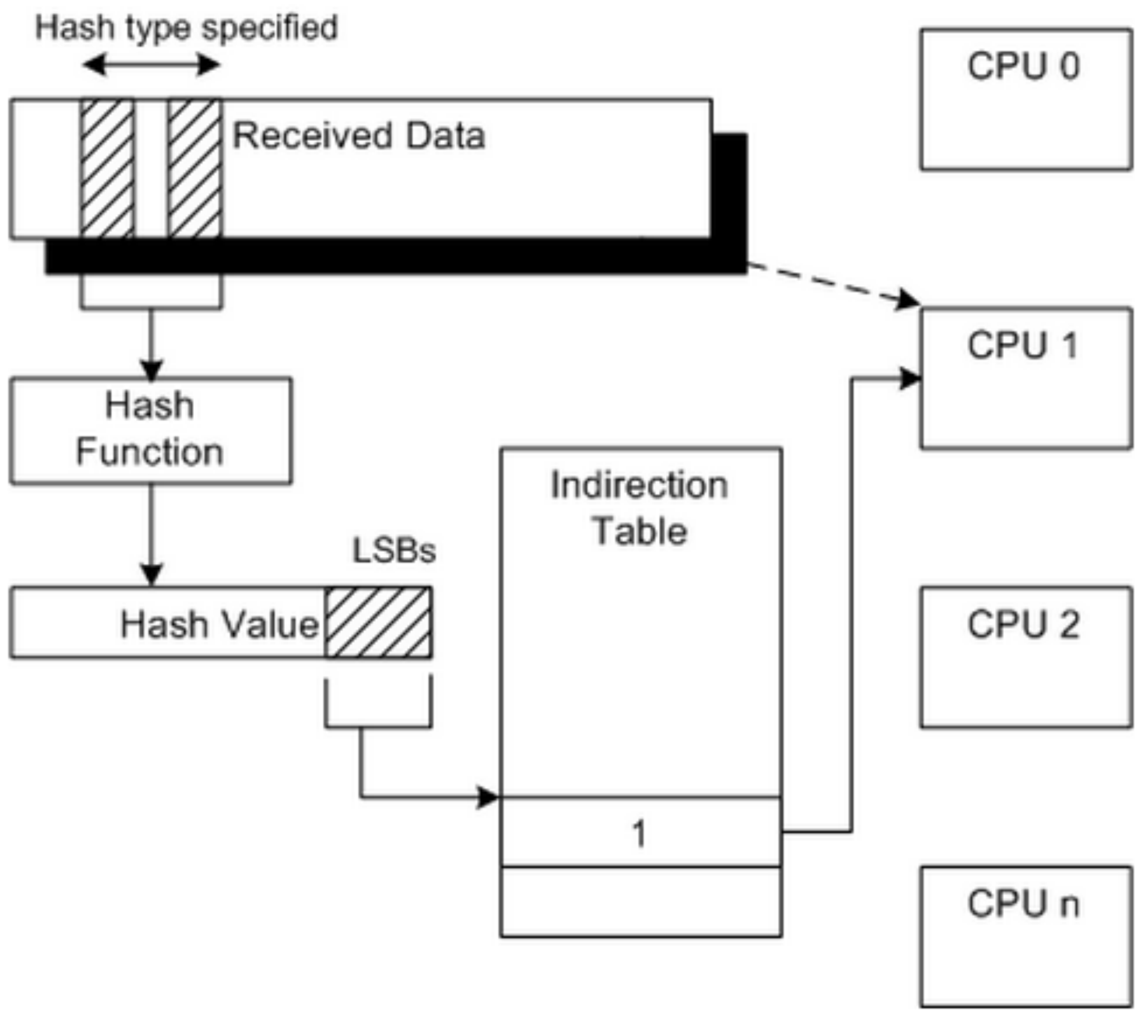


Comparison with IX

Differences:

- IX only optimized for network performance, not storage
- IX is a fork of the Linux kernel, whereas Arrakis is a fork of Barrelfish (an ExoKernel derivative)
- IX uses user-space rings to help enforce security
- Takes advantage of hardware-based RSS to reduce processing delays

Receive Side Scaling



Comparison with IX

Similarities:

- Both use a library OS approach with a user-space network stack
- Each data plane runs a single application in a single address space
- Data planes have associated capabilities
- comparable performance improvements over Linux

Thank You!