Understanding and Improving Device Access Complexity

Asim Kadav
(with Prof. Michael M. Swift)
University of Wisconsin-Madison

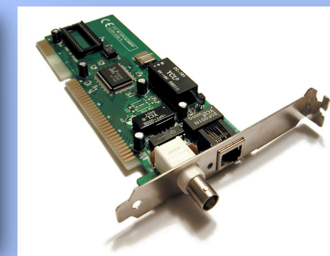
Devices enrich computers

* Keyboard
* Sound
* Printer
* Network
* Storage

* Keyboard
* Flash storage
* Graphics
* WIFI
* Headphones
* SD card
* Camera
* Accelerometers
* GPS
* Touch display
* NFC

Huge growth in number of devices

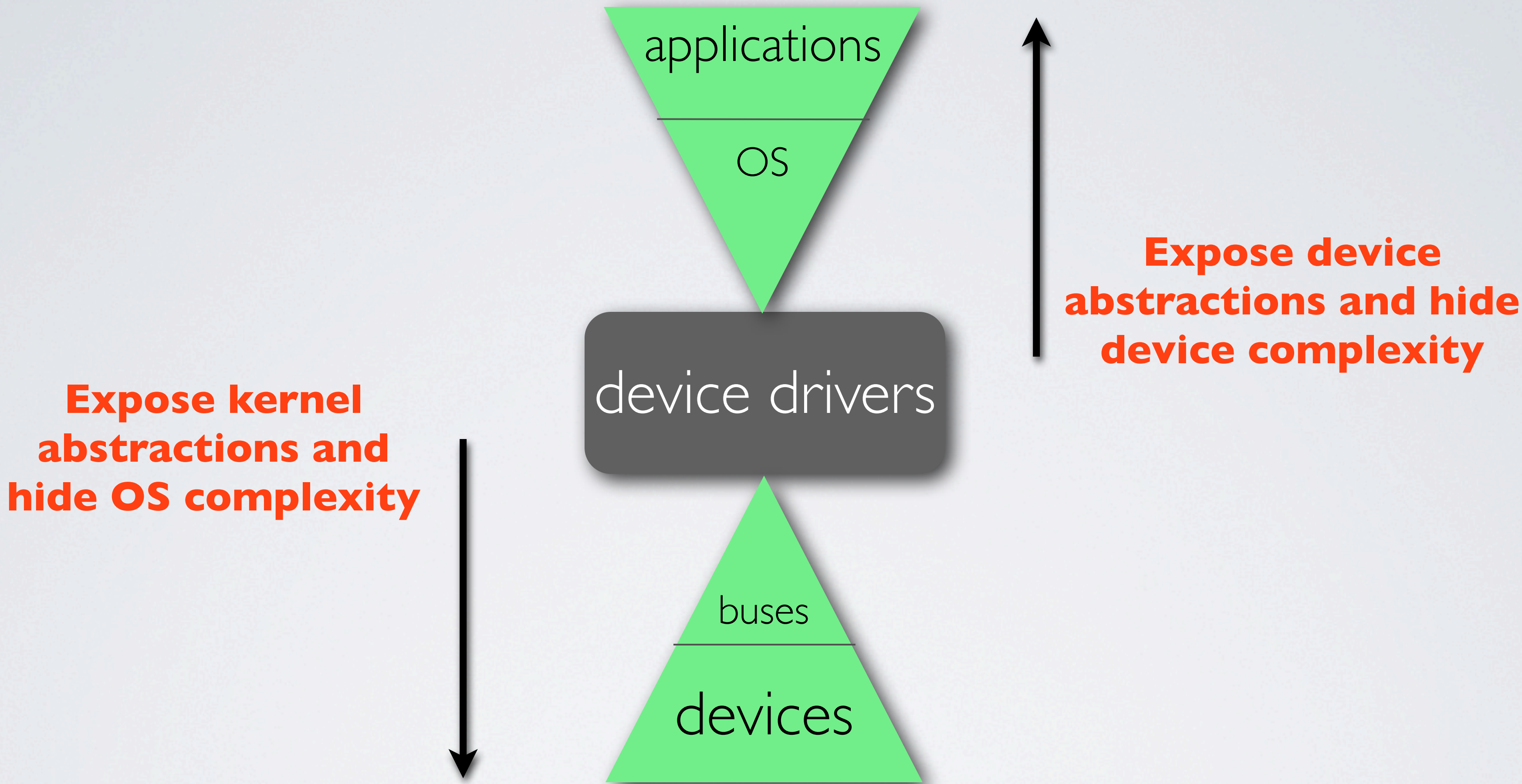
New I/O devices: accelerometers, GPUS, GPS, touch



Many buses: USB, PCI-e, thunderbolt

Heterogeneous O/S support: 10G ethernet vs card readers

Device drivers: OS interface to devices



Allow diverse set of applications and OS services to access diverse set of devices

Complexity hurts efficient device access

Tools and mechanisms to address increasing device complexity

Simplicity

Low latency

Reliability

Cost effective

Efficient device support in OS

Growth in number and diversity

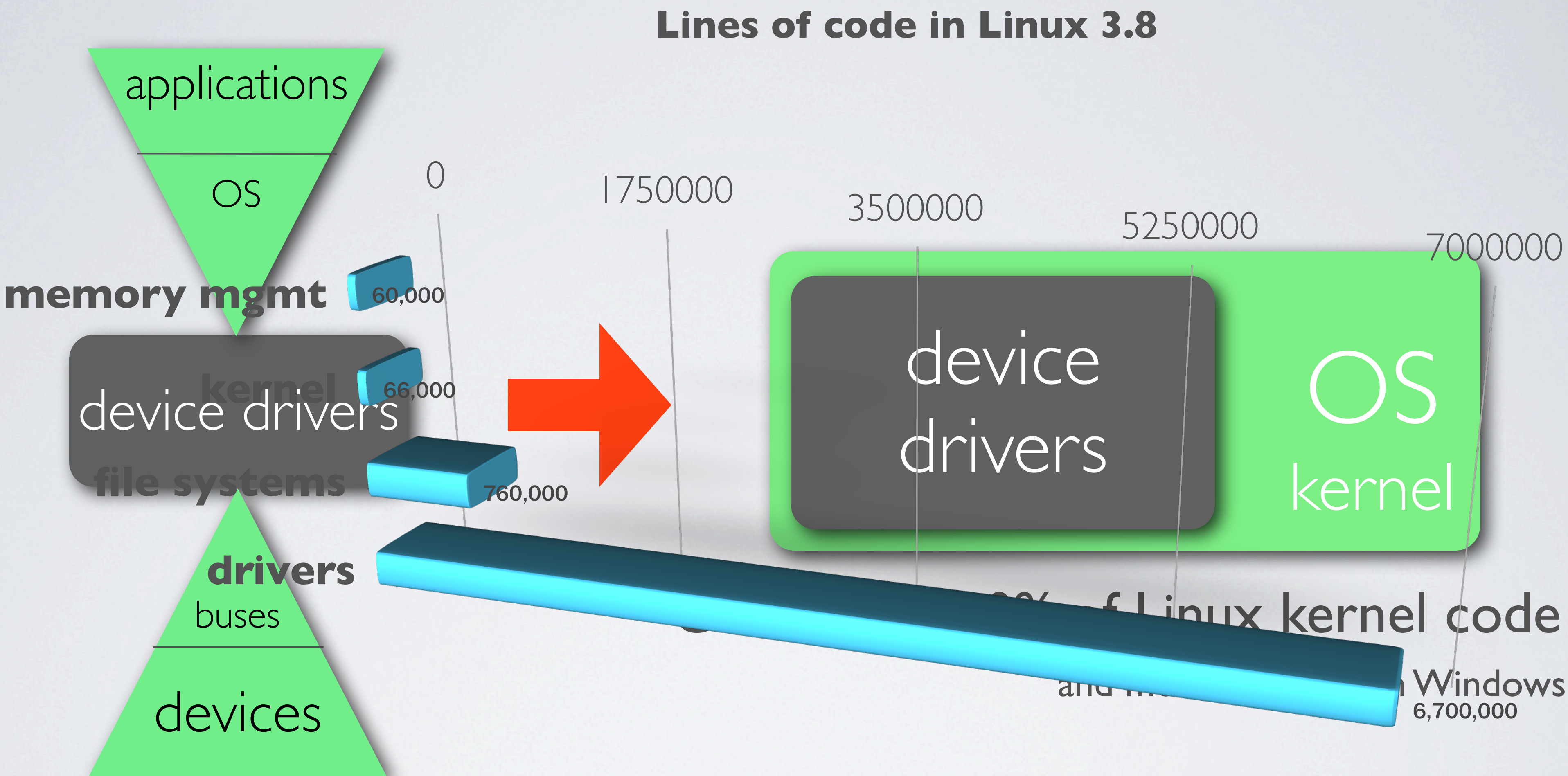
Run in challenging environments

Hardware failures (like CMOS issues)

Complex firmware and configuration modes

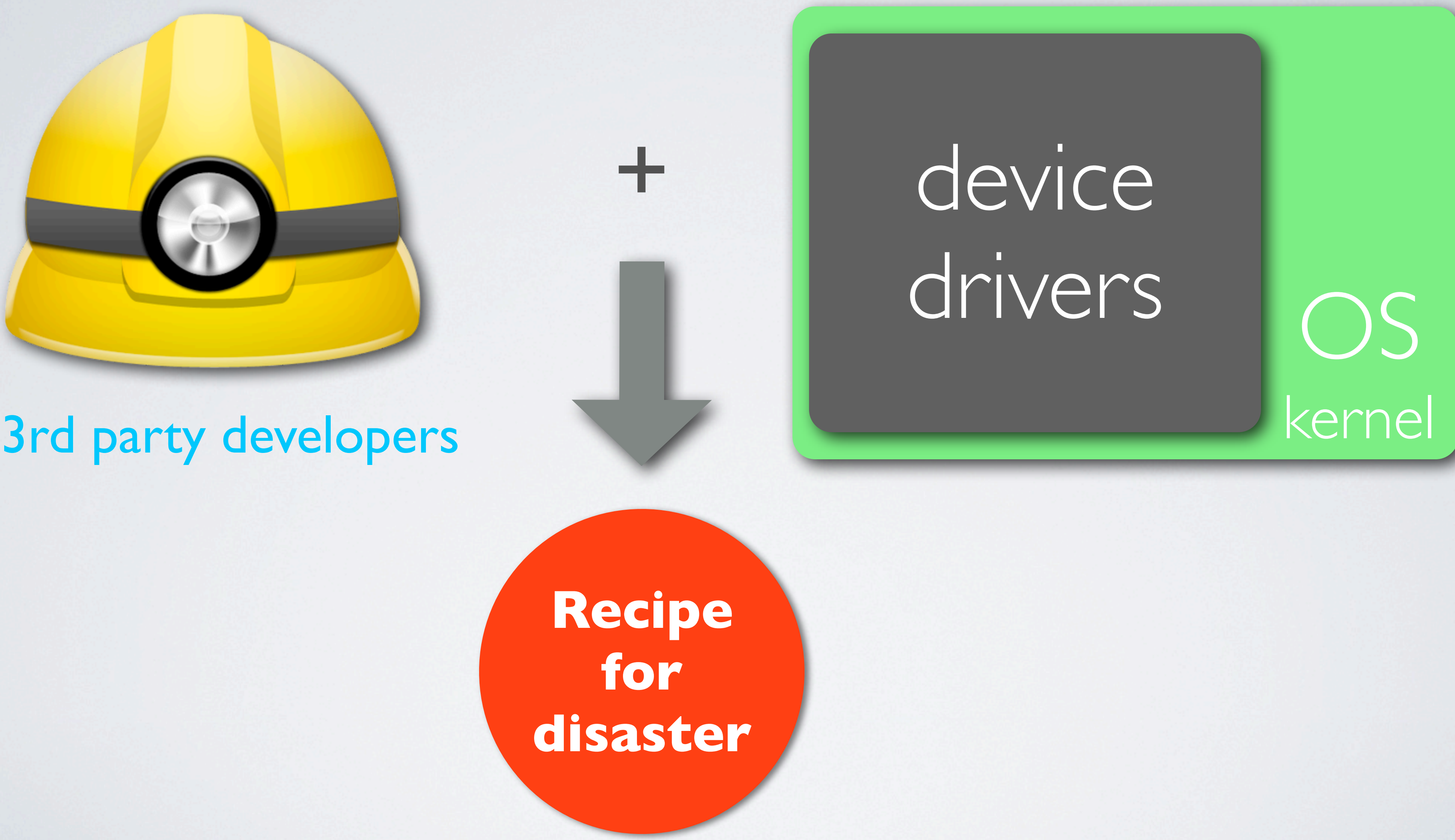
Evolution of devices

Complexity hurts understanding of drivers



Understand and improve this rapidly growing body of code

Last decade: Reliability of the driver-kernel interface



Re-use lessons from existing driver research

Improvement	System	Validation		
		Drivers	Bus	Classes
New functionality	Shadow driver migration [OSR09]	1	1	1
	RevNIC [Eurosys 10]	1	1	1
Reliability	Nooks [SOSP 03]	6	1	2
	XFI [OSDI 06]	2	1	1
	Singularity [Eurosys 06]	1	1	1
Specification	Nexus [OSDI 08]	2	1	2
	Termite [SOSP 09]	2	1	2

Limited kernel changes + Applicable to lots of drivers => Real Impact

Large kernel subsystems and validity of few device types result in limited adoption of research solutions

Goal

★ **Make device access efficient and reliable in the face of rising hardware and software complexity**

Increasing hardware complexity

Reliability against hardware failures

1

Increasing hardware complexity

Low latency device availability

2

Increasing software complexity

Better understanding of driver code

3

My approach

Take a narrow view and solve specific problems in all drivers

Tolerate device failures

Take a broad approach and have a holistic view of all drivers

Understand drivers and potential opportunities

Take a known approach and apply to all drivers

Transactional approach for low latency recovery

Minimize kernel changes and apply to all drivers

Contributions/Outline

First research consideration of hardware failures in drivers

SOSP '09

Tolerate device failures

Largest study of drivers to understand their behavior and verify research assumptions

ASPLOS '12

Understand drivers and potential opportunities

Introduce checkpoint/restore in drivers for low latency fault tolerance

ASPLOS '13

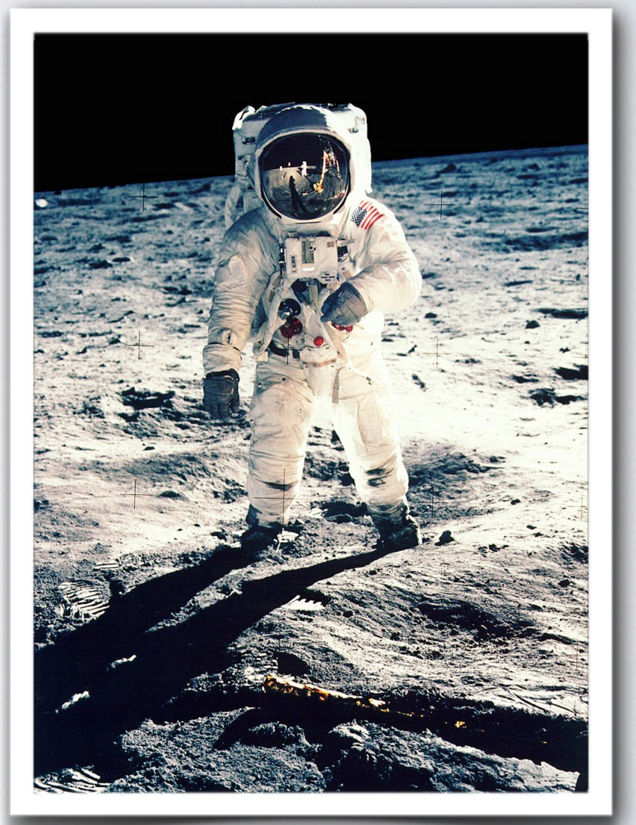
Transactional approach for low latency recovery

What happens when devices misbehave?

- ★ **Drivers make it better**
- ★ **Drivers make it worse**

Early example: Apollo 11 1969

- ★ **Hardware design bug almost aborted the landing**
- ★ **Assumptions about antenna in driver led to extra CPU**
- ★ **Scientists on-board had to manually prioritize critical tasks**



Current state of OS-hardware interaction
2013

★ **Many device drivers often assume device perfection**
  - Common Linux network driver: 3c59x.c

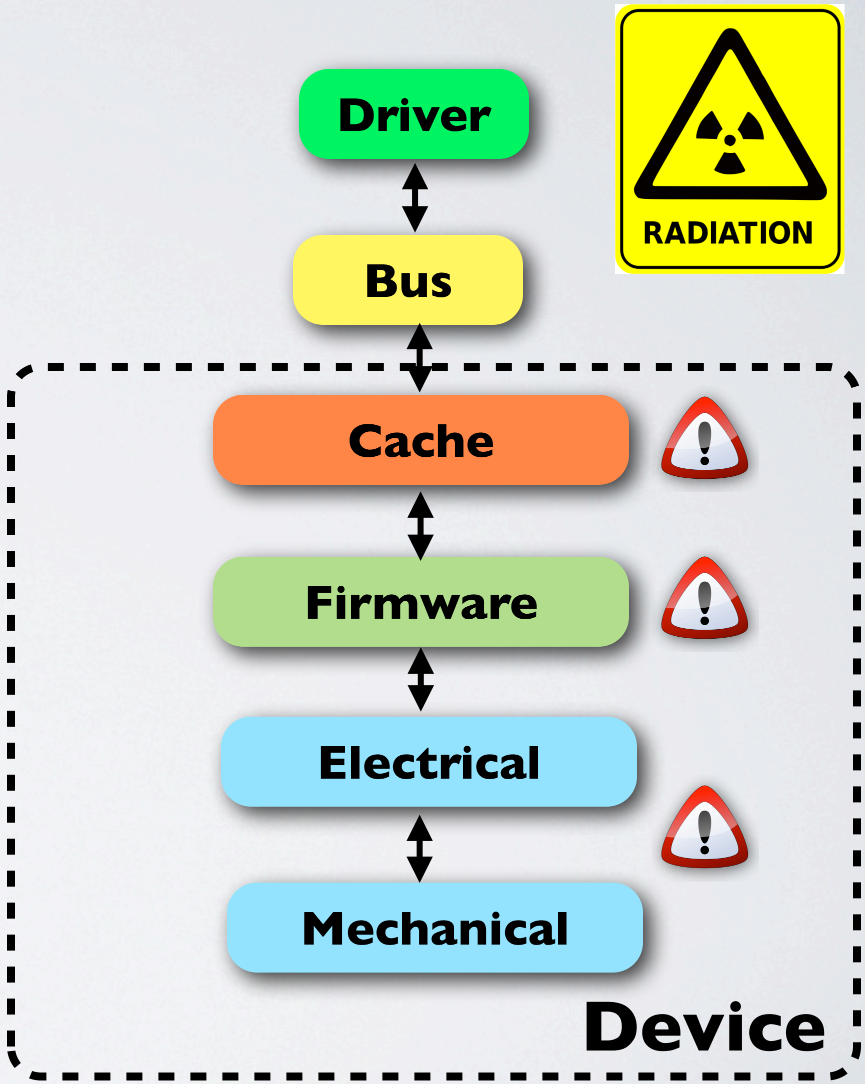
```
while (ioread16(ioaddr + Wn7_MasterStatus))
        & 0x8000);
```

Hardware dependence bug: Device malfunction can crash the system

Sources of hardware misbehavior

★ **Sources of hardware misbehavior**

- ★ **Firmware/Design bugs**
- ★ **Device wear-out, insufficient burn-in**
- ★ **Bridging faults**
- ★ **Electromagnetic radiation**



Sources of hardware misbehavior

- **Sources of hardware misbehavior**
  - **Firmware/Design bugs**
  - **Device wear-out, insufficient burn-in**
  - **Bridging faults**
  - **Electromagnetic radiation**

- **Results of misbehavior**
  - **Corrupted/stuck-at inputs**
  - **Timing errors/incorrect memory access**
  - **Interrupt storms/missing interrupts**

An evidence: Windows Server

Transient hardware failures caused **8%** of all crashes and **9%** of all unplanned reboots [1]

- ★ Systems work fine after reboots
- ★ Vendors report returned device was faultless

Existing solution is **hand-coded** hardened drivers

- ★ Crashes reduce from **8%** to **3%**

[1] Fault resilient drivers for Longhorn server, May 2004. Microsoft Corp.

How do hardware dependence bugs manifest?

1

Drivers use device data in critical control and data paths

```
printk("%s",msg[inb(regA)]);
```

2

Drivers do not report device malfunction to system log

```
if (inb(regA)!= 5)  {
  return; //do nothing
}
```

3

Drivers do not detect or recover from device failures

```
if (inb(regA)!= 5) {
   panic();
}
```

Vendor recommendations for driver developers

Recommendation	Summary	Recommended by			
		Intel	Sun	MS	Linux
Validation	Input validation	●	●	●	
	Read once& CRC data	●	●		●
	DMA protection	●	●		
Timing	Infinite polling	●	●	●	
Reporting	Report all failures	●	●	●	
Recovery	Handle all failures		●	●	
	Cleanup correctly	●	●		
	Do not crash on failure	●		●	●
	Wrap I/O memory access	●	●	●	●

Goal: Automatically implement as many recommendations as possible in commodity drivers

Carburizer [SOSP '09]

Goal: Tolerate hardware device failures in software through hardware failure detection and recovery

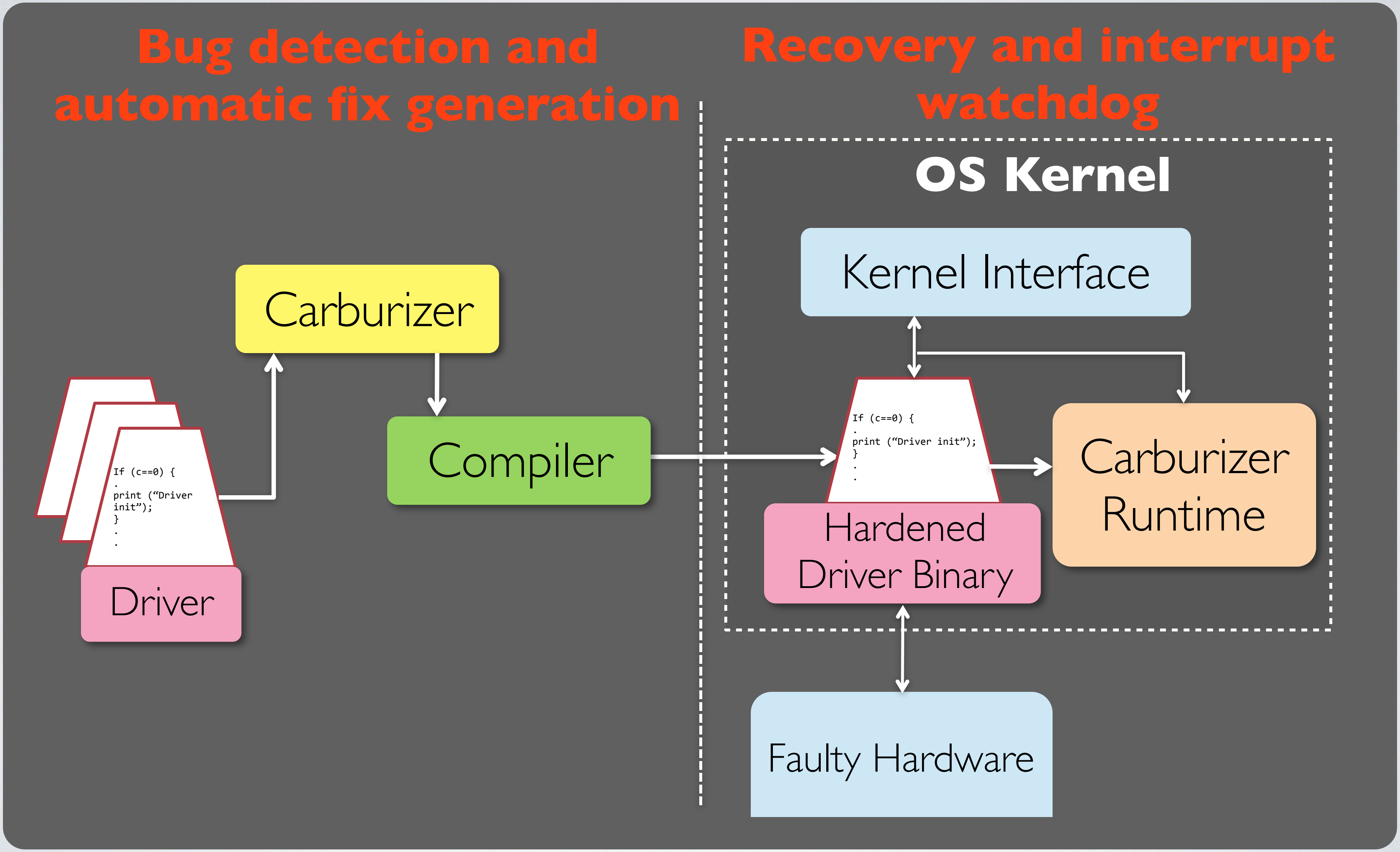
Static analysis component

- **Detect and fix hardware dependence bugs**
- **Detect and generate missing error reporting information**

Runtime component

- **Detect interrupt failures**
- **Provide automatic recovery**

Carburizer architecture



Hardening drivers

- **Goal: Remove hardware dependence bugs**
  - ⋆ **Find driver code that uses data from device**
  - ⋆ **Ensure driver performs validity checks**
- **Carburizer detects and fixes hardware bugs :**

Infinite polling

Unsafe pointer reference

Unsafe array reference

System panic calls

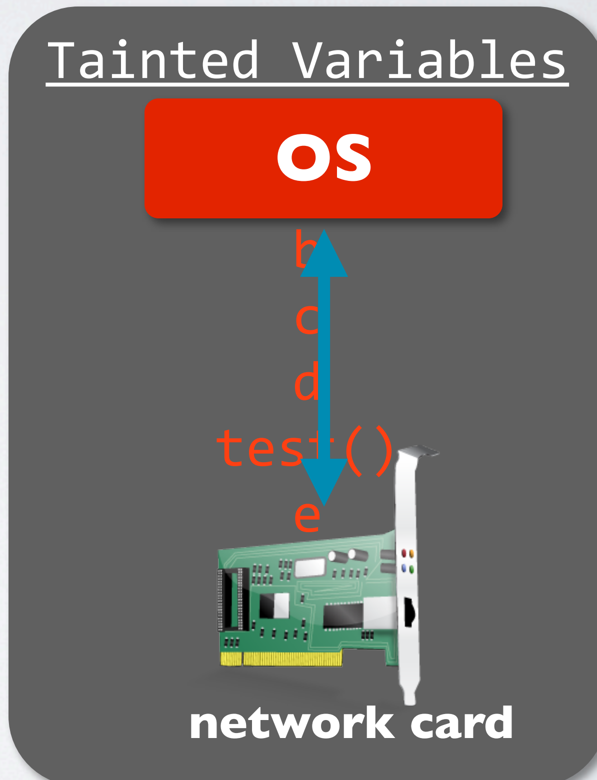
Finding sensitive code

- First pass: Identify tainted variables that contain data from device

```
int test() {
    a = readl();
    b = inb();
    c = b;
    d = c + 2;
    return d
}
int set() {
    e = test();
}
```

Types of device I/O

- **Port I/O :** inb/outb
- **Memory-mapped I/O :** readl/writel
- **DMA buffers**
- **Data from USB packets**



Detecting risky uses of tainted variables

- **Finding sensitive code**
  - **Second pass: Identify risky uses of tainted variables**
- **Example: Infinite polling**
  - **Driver waiting for device to enter particular state**
  - **Solution: Detect loops where all terminating conditions depend on tainted variables**

Infinite polling

★ **Infinite polling of devices can cause system lockups**

```
static int amd8111e_read_phy(………)
{
 ...
  reg_val = readl(mmio + PHY_ACCESS);
  while (reg_val & PHY_CMD_ACTIVE)
        reg_val = readl(mmio + PHY_ACCESS)
  ...
}
```

AMD 8111e network driver(amd8111e.c)

Hardware data used in array reference

★ **Tainted variables used as array indexes**

```
static void __init attach_pas_card(...)
{
   if ((pas_model = pas_read(0xFF88)))
   {
     ...
     sprintf(temp, "%s rev %d",
       pas_model_names[(int) pas_model], pas_read(0x2789));
     ...
}
```

Pro Audio Sound driver (pas2_card.c)

Experience with the Linux kernel

- **Extra analyses to reduce false positives**
  - **Detect counters, range and not NULL checks**
  - **Detect taint lifetimes**
- **Analyzed drivers in 2.6.18.8 Linux kernel**
  - **6300 driver source files**
  - **2.8 million lines of code**
  - **37 minutes to analyze and compile code**

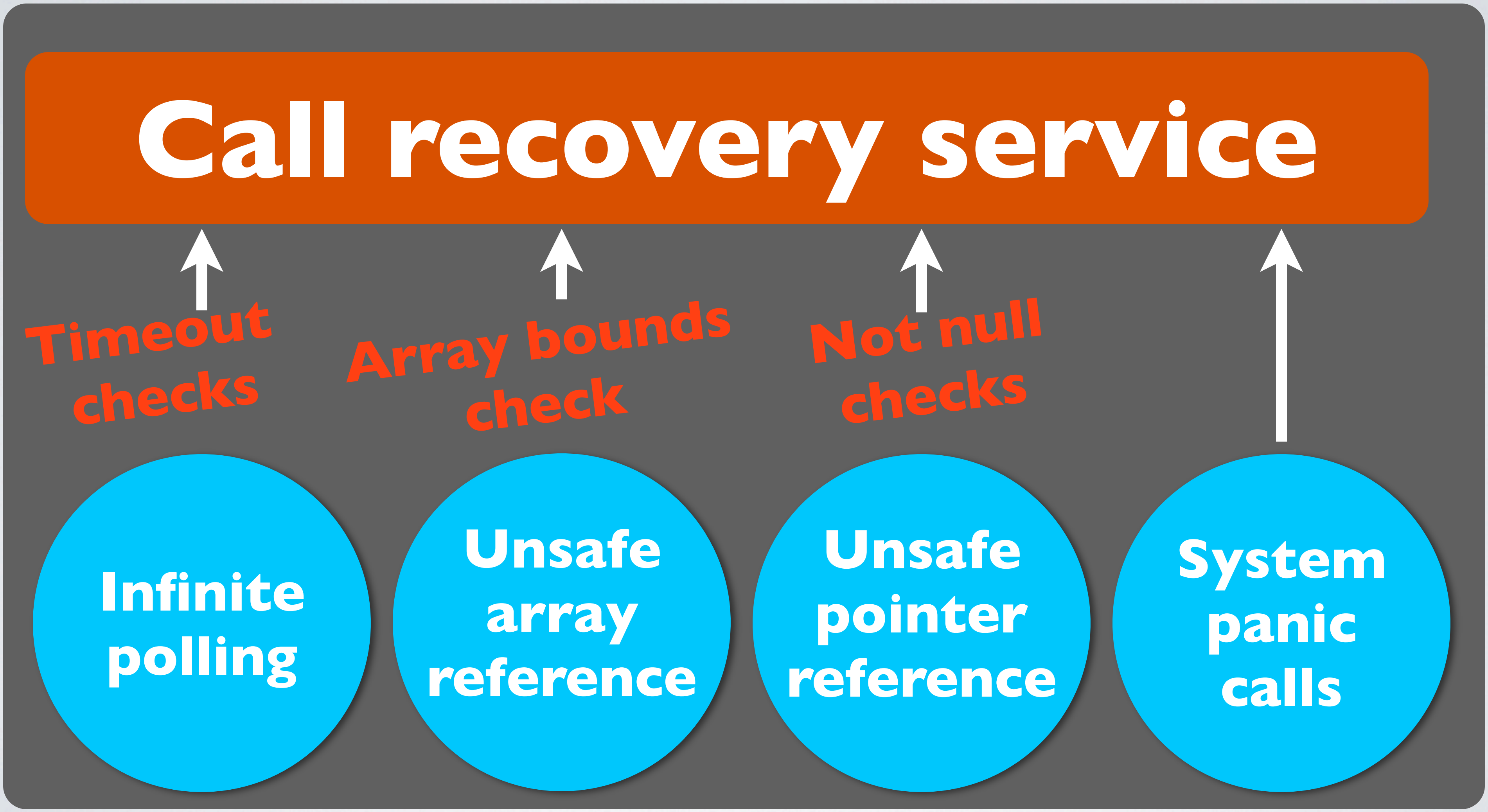
Analysis results over the Linux kernel

Driver class	Infinite polling	Static array	Dynamic array	Panic calls
net	117	2	21	2
scsi	298	31	22	121
sound				
video				2
other	381	9	57	32
Total	860	43	89	179

Lightweight and usable technique to find hardware dependence bugs

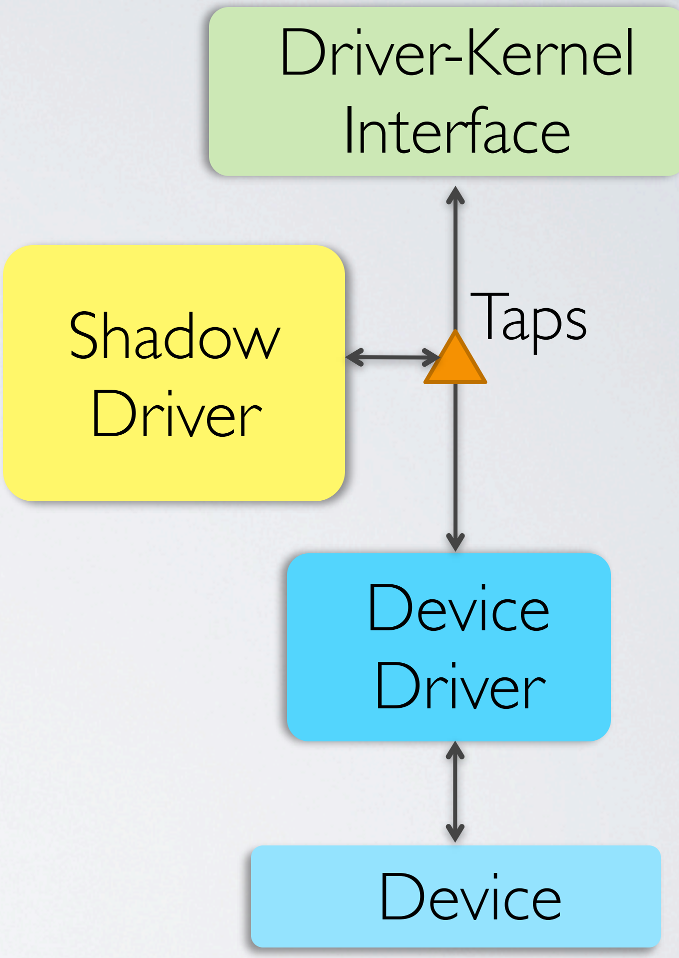
- **Found 992 hardware dependence bugs in driver code**
- **False positive rate: 7.4% (manual sampling of 190 bugs)**

Repairing drivers



Runtime fault recovery

- **Carburizer calls generic recovery service if check fails**
- **Low cost transparent recovery**
  - ★ **Based on shadow drivers**
  - ★ **Records state of driver**
  - ★ **Transparent restart and state replay on failure**
- **No isolation required (like Nooks)**



Swift [OSDI '04]

Carburizer automatically fixes infinite loops

```
timeout = rdtscll(start) + (cpu/khz/HZ)*2;
reg_val = readl(mmio + PHY_ACCESS);
while (reg_val & PHY_CMD_ACTIVE)       {
        reg_val = readl(mmio + PHY_ACCESS);

        if (_cur < timeout)
            rdtscll(_cur);
        else
            __recover_driver();


}
```

Timeout code added

AMD 8111e network driver(amd8111e.c)

*Code simplified for presentation purposes

Carburizer automatically adds bounds checks

```
static void __init attach_pas_card(...)
{

  if ((pas_model = pas_read(0xFF88)))
  {
    ...
    if ((pas_model< 0)) || (pas_model>= 5))
        __recover_driver();
     .
    sprintf(temp, “%s rev %d”,
      pas_model_names[(int) pas_model], pas_read(0x2789));

}
```

Array bounds detected and check added

Pro Audio Sound driver (pas2_card.c)

*Code simplified for presentation purposes

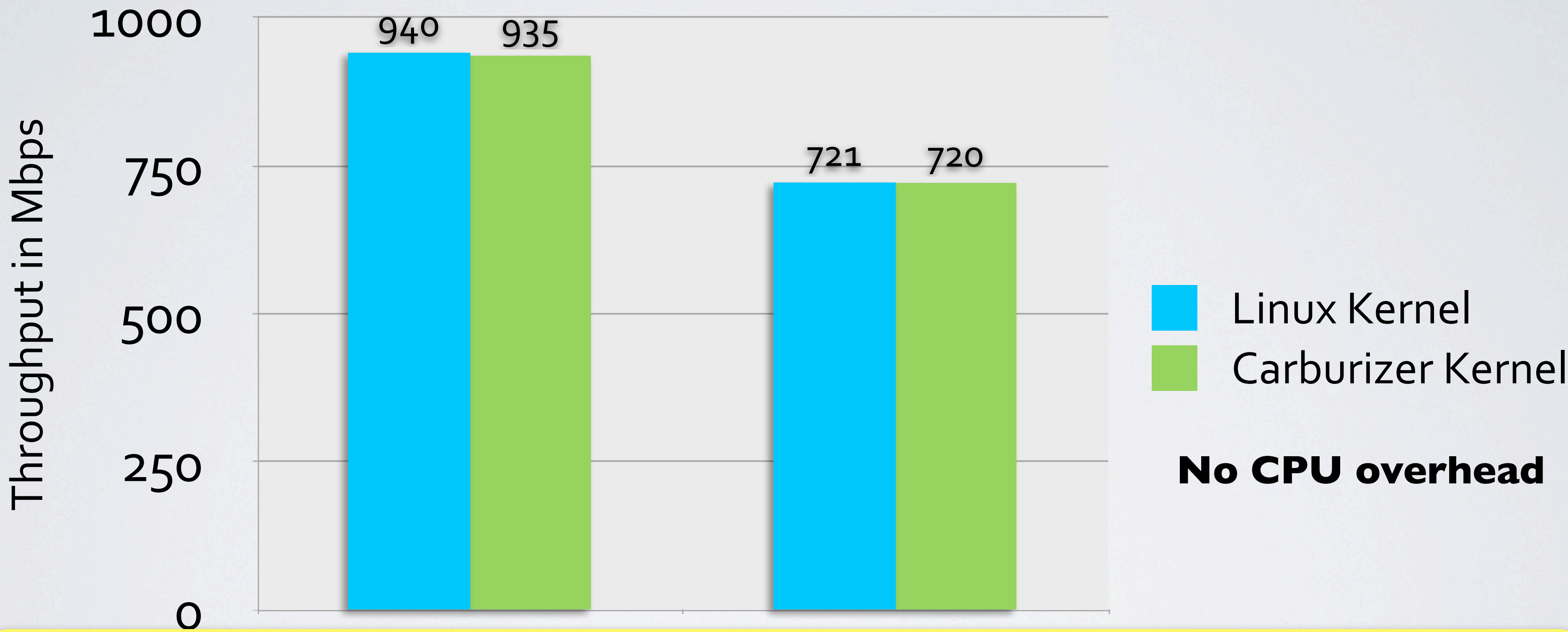
Fault injection validation

- ★ **Synthetic fault injection on network drivers**
- ★ **Results**

Device/ Driver	Original Driver		Carburizer		
	Behavior	Detection	Behavior	Detection	Recovery
3COM 3C905	CRASH	None	RUNNING	Yes	Yes
DEC DC 21x4x	CRASH	None	RUNNING	Yes	Yes

Carburizer failure detection and transparent recovery work well for transient device failures

Throughput overhead



No CPU overhead

Almost no overhead from hardened drivers and automatic recovery

Outline

Tolerate device failures

Hardening drivers
Reporting failures
Runtime Fault tolerance
Results

Understand drivers and potential opportunities

Transactional approach for cheap recovery

Outline

Tolerate device failures

- Hardening drivers
- Reporting failures
- **Runtime Fault tolerance**
- Results

Understand drivers and potential opportunities

Transactional approach for cheap recovery

Runtime failure detection

- Static analysis cannot detect all device failures

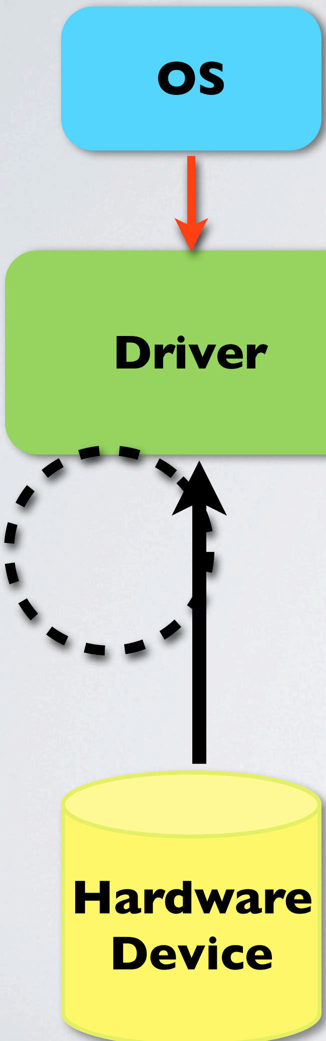
Missing interrupts

Interrupt expected but never arrives

Stuck interrupts

Interrupt cleared but continues to assert

Missing interrupts



- Device polling on interrupt failures
  - Polling frequently has high overhead
  - Polling infrequently results in throughput loss
- How frequently should we poll?
  - Increase frequency if interrupt invocation did useful work
- When are requests likely to come?
  - Driver invocation: Use reference bits to detect driver activity

Stuck interrupts

- Driver interrupt handler is called too many times
- Convert the device from interrupts to polling

Driver Type	Driver Name	Native	With Carburizer Runtime
Disk	ide-core,ide-disk, ide-generic	Hang	Reduced by 50%
Network	e1000	Hang	Reduced from 750 Mb/s to 130 Mb/s
Sound	ens1371	Hang	Sounds plays with distortion

Carburizer ensures system makes forward progress

Summary

Recommendation	Summary	Recommended by				Carburizer Ensures
		Intel	Sun	MS	Linux	
Validation	Input validation	●	●	●		●
	Read once& CRC data	●	●		●	
	DMA protection	●	●			
Timing	Infinite polling	●	●	●		●
	Stuck interrupt		●			●
	Lost request			●		●
	Avoid excess delay in OS			●		
	Unexpected events	●		●		
Reporting	Report all failures	●	●	●		●
	Wrap I/O memory access	●	●	●	●	

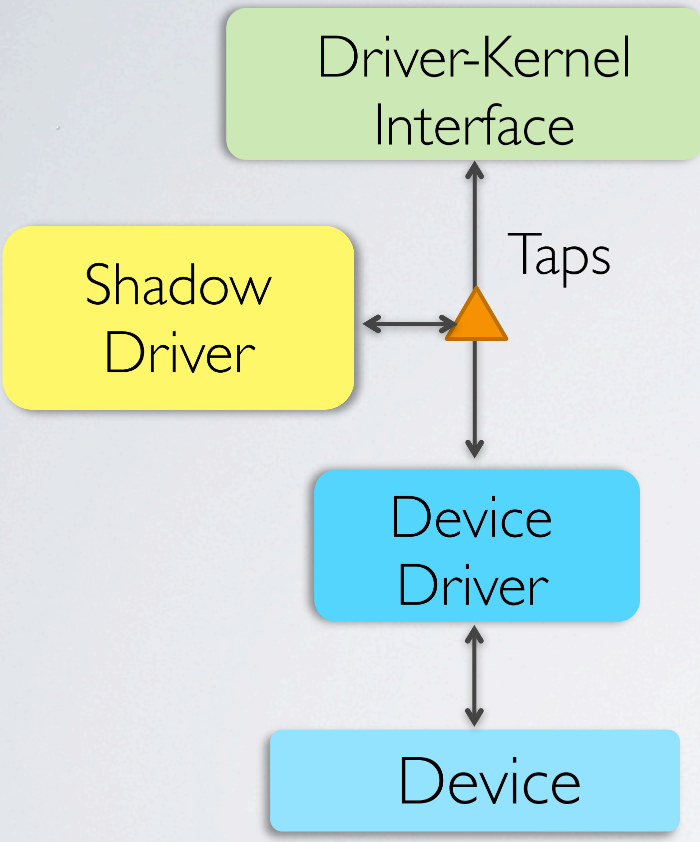
Carburizer improves system reliability by automatically ensuring that hardware failures are tolerated in software

Contributions beyond research

- ★ **Informed developers at Plumbers Conference [2011]**
- ★ **LWN Article with paper & list of bugs [Feb '12]**
- ★ **Released patches to the Linux kernel**
- ★ **Tool + source available for download at:**

`http://bit.ly/carburizer`

Functionality: Recovery assumes drivers follow class behavior

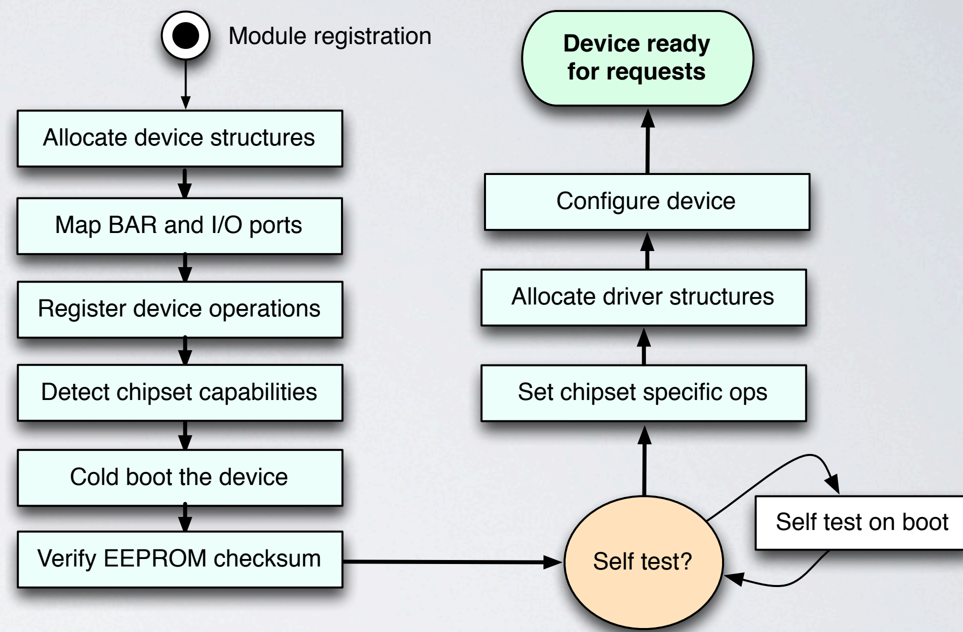


- ★ **Record state by interposing class defined entry points**
- ★ **Restart and replay state using class semantics when failure happens**

Non-class behavior can lead to incomplete restore after failure

Recovery Performance: Device initialization is slow

- **Multi-second device probe**
  - **Identify device**
  - **Cold boot device**
  - **Setup device/driver structures**
  - **Configuration/Self-test**



- **What does it hurt?**
  - **Fault tolerance: Driver recovery**
  - **Virtualization: Live migration, cloning, consolidation**
  - **OS functions: Boot, upgrade, NVM checkpoints**

Outline

Tolerate device failures

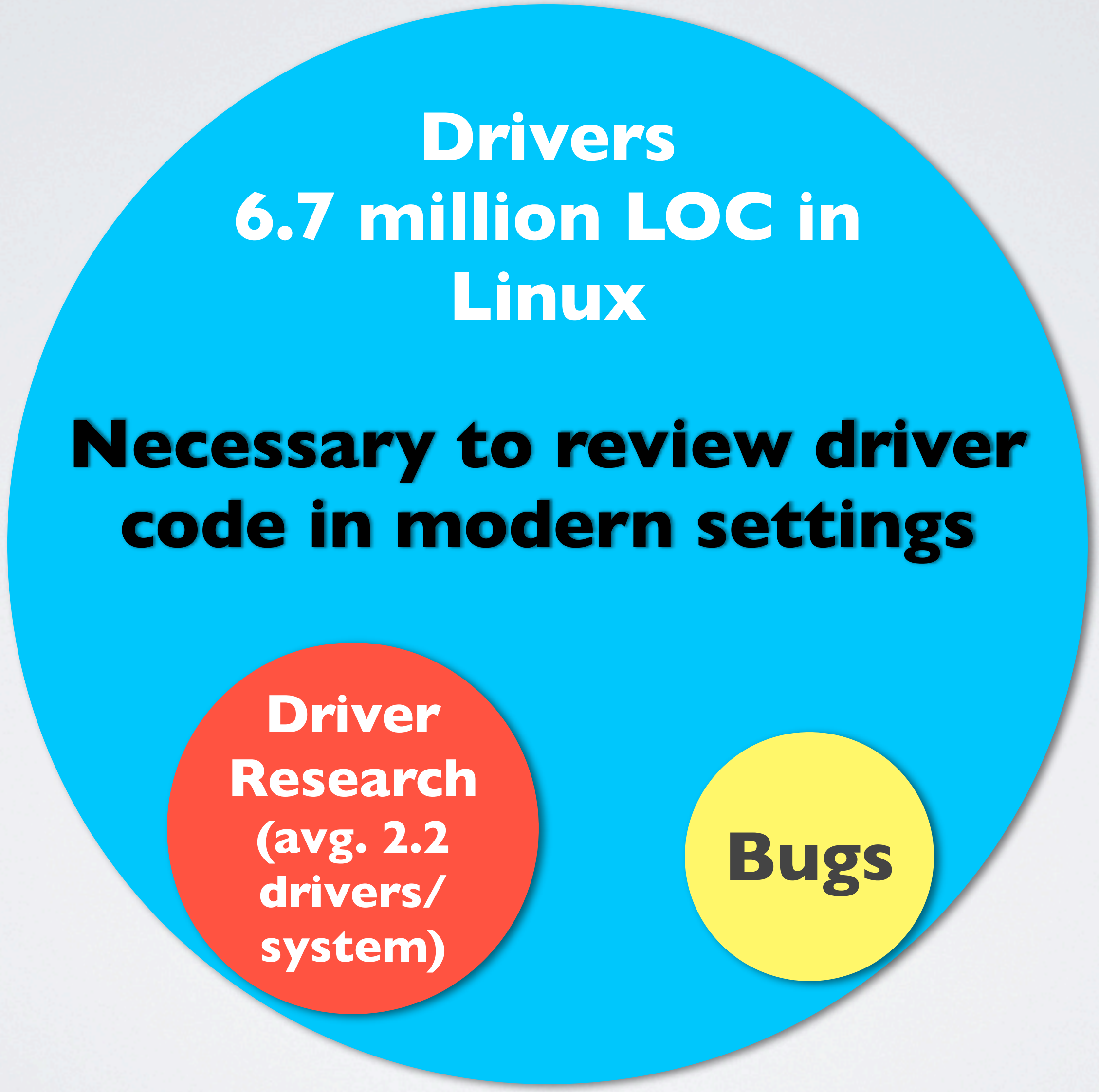
Understand drivers and potential opportunities

Overview

Recovery specific results

Transactional approach for cheap recovery

Our view of drivers is narrow



Understanding Modern Device Drivers[ASPLOS 2012]

Study source of all Linux drivers for x86 (~3200 drivers)

Driver properties

- Code properties
- Verify research assumptions

Driver interaction

- Driver kernel & device interaction
- Driver architecture

Driver similarity

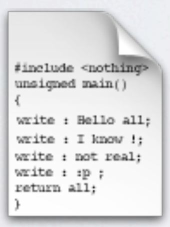
- 7 million lines of code needed?

Study methodology

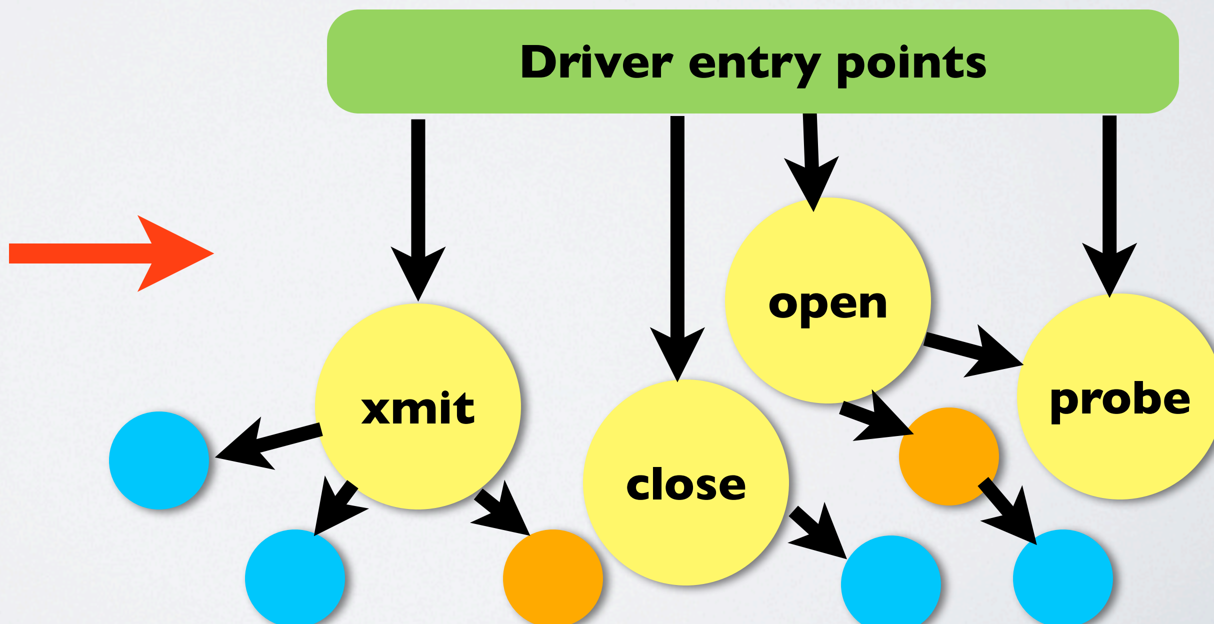
★ **Static source analysis of 3200 drivers in Linux 2.6.37.6 (May 2011)**

Driver properties

- ★ **Identify driver entry points, kernel and bus callouts**
  - ★ **Device class, sub-class**
  - ★ **Driver functions registered as entry points (purpose)**
  - ★ **Bus properties**
  - ★ **Other properties (module params)**



For every merged driver



Study methodology

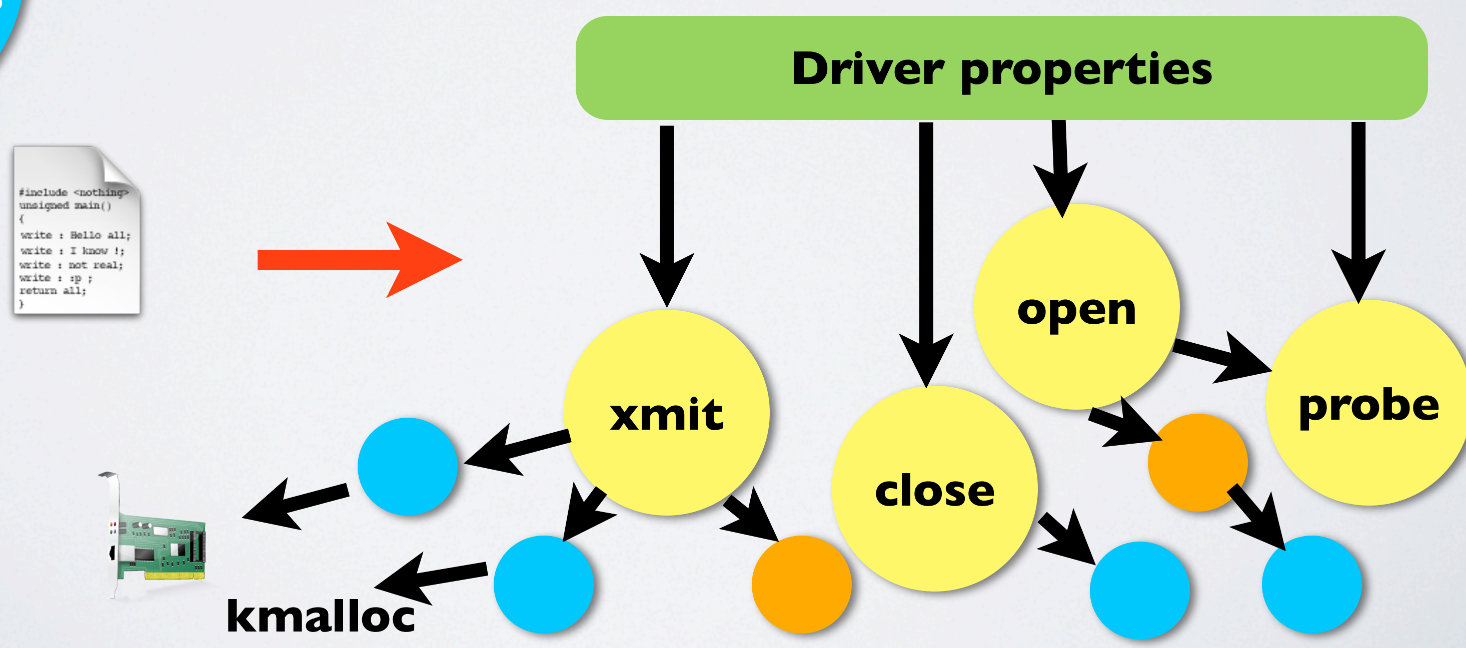
★ **Static source analysis of 3200 drivers in Linux 2.6.37.6 (May 2011)**

Driver properties

★ **Identify driver entry points, kernel and bus callouts**

Driver interactions

★ **Reverse propagate information to aggregate bus, device and kernel behavior**



Study methodology

★ **Static source analysis of 3200 drivers in Linux 2.6.37.6 (May 2011)**

Driver properties

★ **Identify driver wide and function specific properties of all drivers**

Driver interactions

★ **Reverse propagate information to aggregate bus, device and kernel behavior**

Driver similarity

★ **Use statistical clustering techniques and static analysis to identify similar code**

Some additional results

Driver properties

- ★ Many assumptions made by driver research does not hold:
  - ★ 15% drivers perform significant processing
  - ★ 28% drivers support multiple chipsets

Driver interactions

- ★ USB bus offers efficient access (as compared to PCI, Xen)
  - ★ Supports high # devices/driver (standardized code)
  - ★ Coarse-grained access

Driver similarity

- ★ 400, 000 lines of code similar to code elsewhere and ripe for improvement via:
  - ★ Procedural abstractions
  - ★ Better multiple chipset support
  - ★ Table driver programming

Contributions/Outline

Tolerate device failures

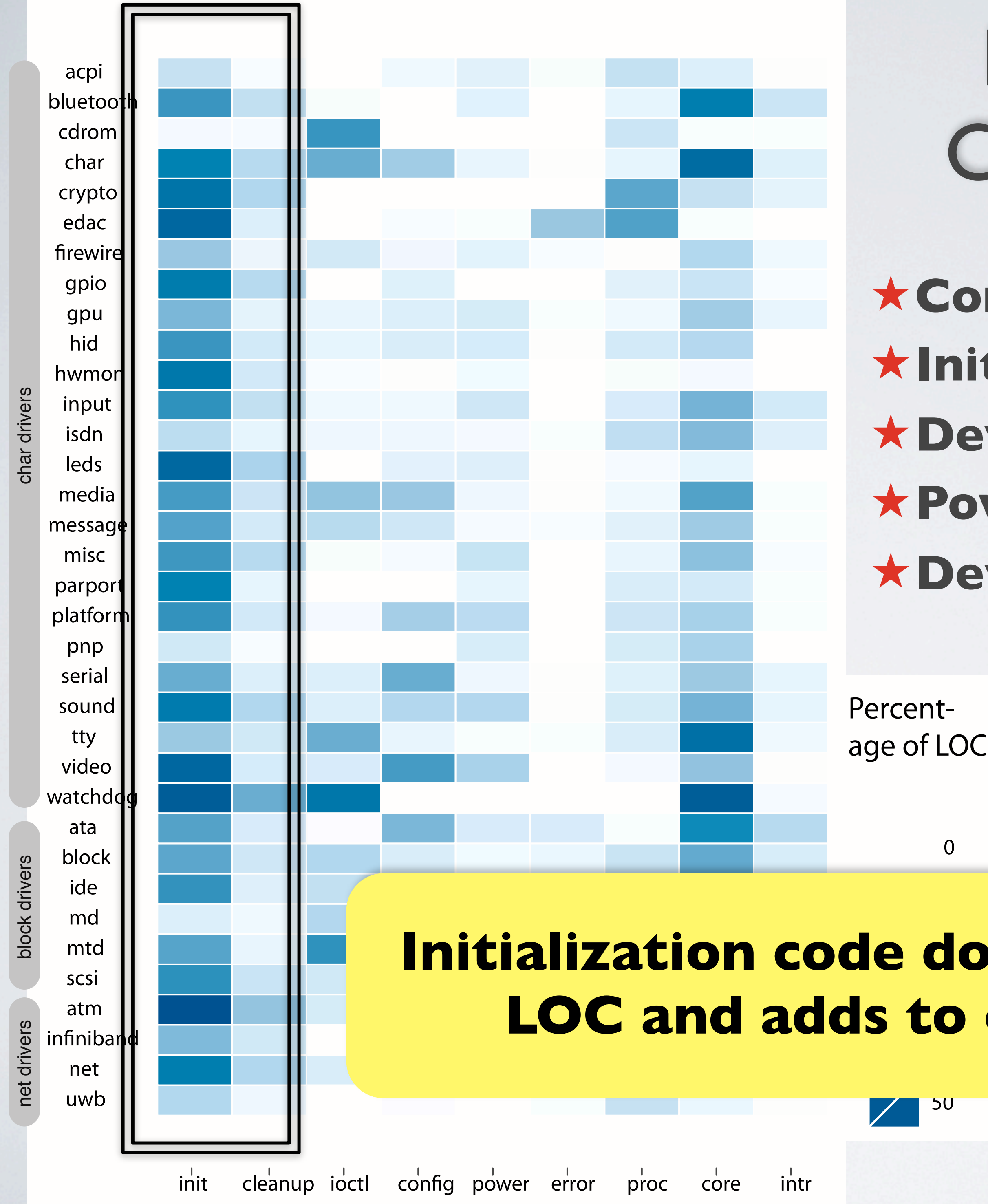
Understand drivers and potential opportunities

Overview

Recovery specific results

Transactional approach for cheap recovery

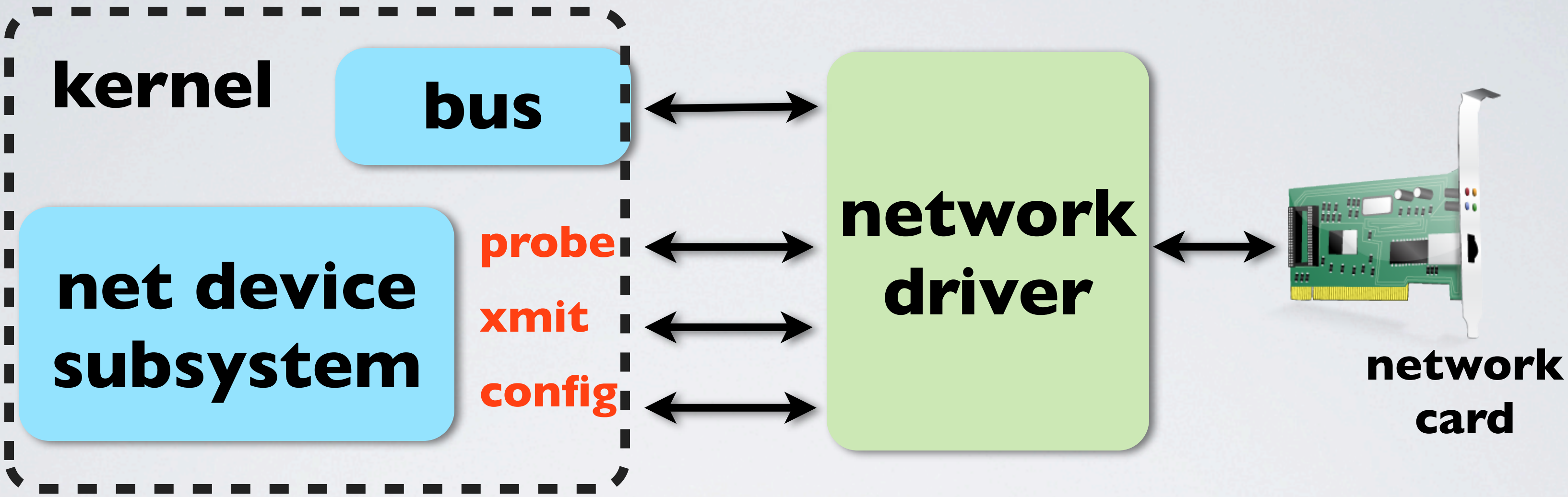
Driver Code Characteristics



- ★ **Core I/O & interrupts – 23%**
- ★ **Initialization/cleanup – 36 %**
- ★ **Device configuration – 15%**
- ★ **Power management – 7.4%**
- ★ **Device ioctl – 6.2%**

Initialization code dominates driver LOC and adds to complexity

Recovery assumes drivers follow class behavior



- Class definition includes:
  - Callbacks registered with the bus, device and kernel subsystem
  - Exported APIs of the kernel to use kernel resources and services

Does driver behavior belong to class definitions?

Do drivers belong to classes?

- **Non-class behavior stems from:**
  - **Load time parameters, unique ioctls, procfs and sysfs interactions**

- **Results as measured by our analyses:**
  - **16% of drivers use proc /sysfs support**
  - **36% of drivers use load time parameters**
  - **16% of drivers use ioctl that *may* include non-standard behavior**

- **Overall, 44% of drivers do not conform to class behavior**

Outline

Tolerate device failures

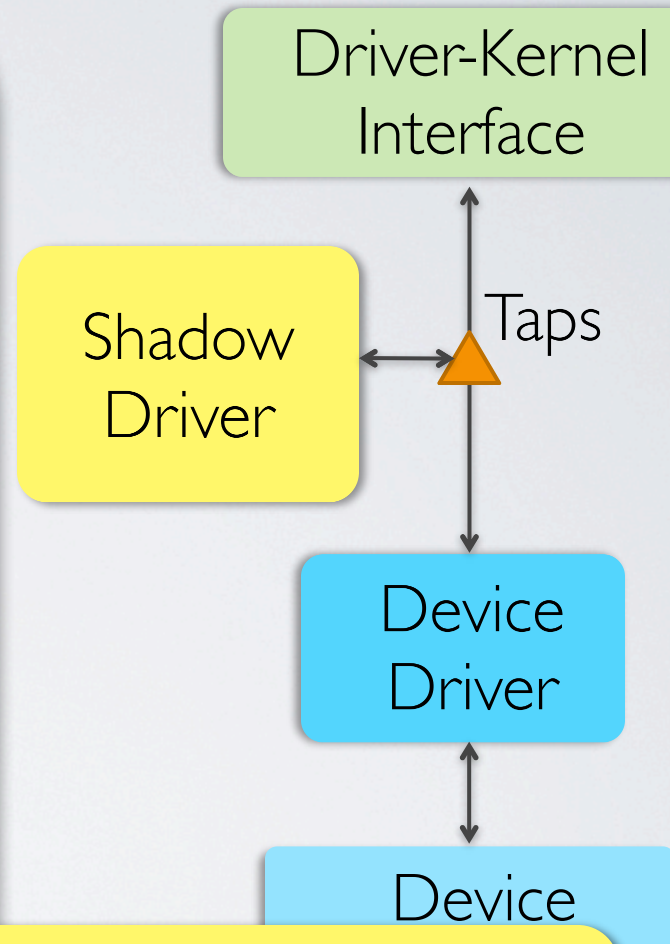
Understand drivers and potential opportunities

Transactional approach for cheap recovery

Checkpoint/restore
FGFT
Future work and conclude

Limitations of restart/replay recovery

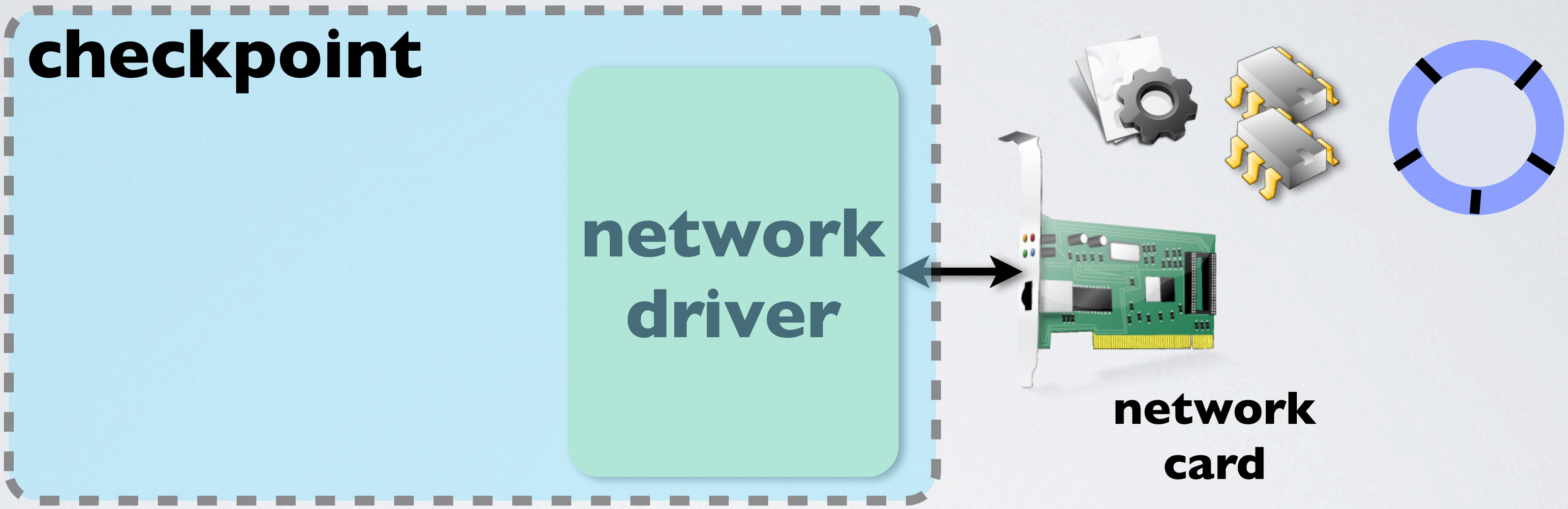
- **Device save/restore limited to restart/replay**
  - **Slow: Device initialization is complex (multiple seconds)**
  - **Not enough: Incomplete recovery due to unique semantics**
  - **Hard: Need to be written for every class of drivers**
  - **Expensive: Continuous logging of all driver operations**



Checkpoint/restore of device and driver state removes the need to reboot device and replay state

Checkpoint/Restore

- Checkpoints limited to capturing **memory** state



- Device state is not captured
  - Device configuration space
  - Internal device registers and counters
  - Memory buffer addresses used for DMA

Power management in drivers

- **Intuition: Power management code captures vendor specific state for every device**
  - **Our study: Present in 76% of all common classes**

- **Suspend to RAM: Save state and suspend processors and devices**

- **Refactor power management code for checkpoint/restore**
  - **Correct: Driver developer captures unique semantics**
  - **Fast: Avoids probe and latency critical for applications**

Checkpoint/Restore from PM code

Suspend

- Save config state
- Save device state
- Disable device
- Copy-out s/w state
- Susp

Resume

- Restore config state
- Restore register state
- Restore s/w state & reset
- Re-attach/Enable device
- dy

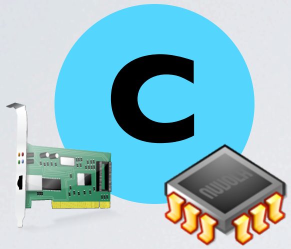
Suspend/resume code provides checkpoint functionality

Fine-Grained Fault Tolerance [ASPLOS 2013]

- **Use device checkpoints** to improve recovery
- Execute driver entry points as **transactions**
  - **Take a device checkpoint, run driver as memory transaction**
  - **If the driver fails, we abort memory transaction and restore the checkpoint**
- Provide memory safety and trap processor exceptions
- Recovery is simple and fast

- Developers export checkpoint/restore in all drivers

Fine-Grained Isolation



netdev->priv->rx_ring

netdev->priv->tx_ring

Range Table

Address	Access rights
0xffffa000	Read
0xffffa008	Write
0xffffa00a	Read

get ringparam

probe

xmit

get config

network driver

Kernel Log

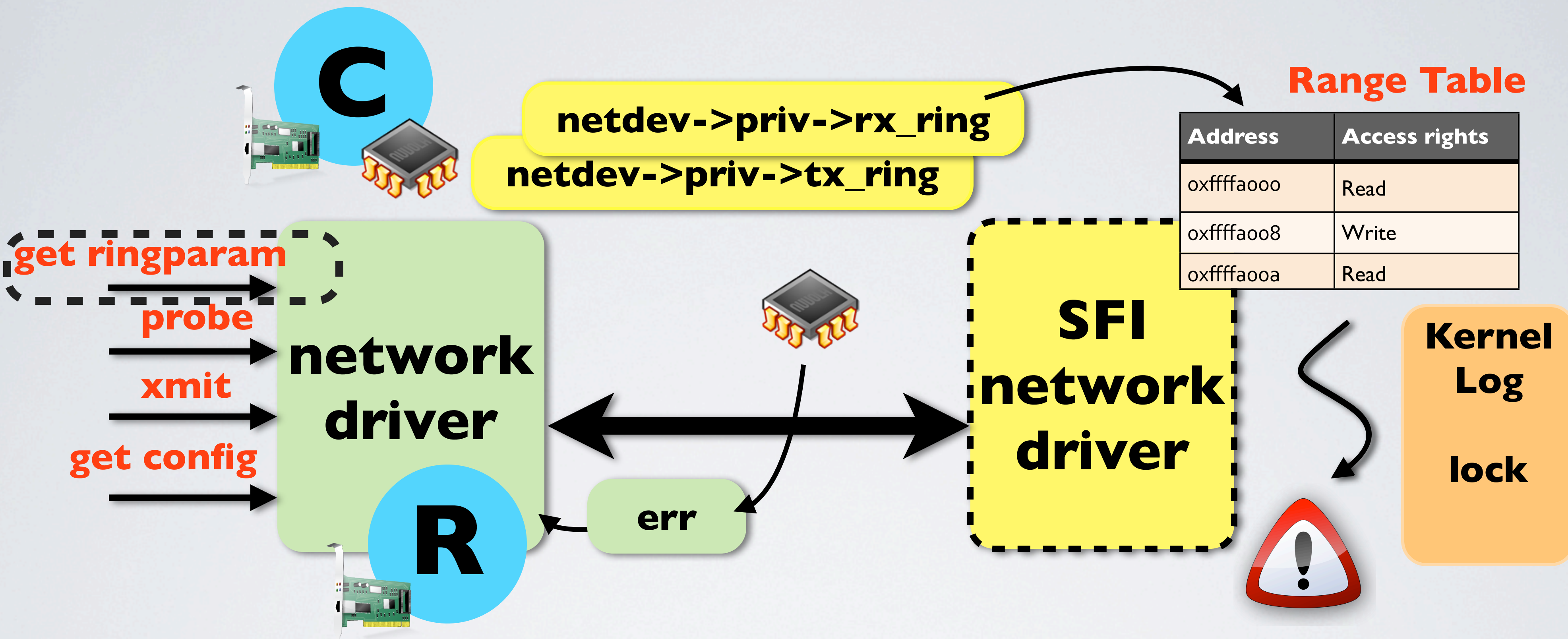
lock

Resource access

- I/O memory: Full access
- Locks: Read access & locks acquired via kernel
- Memory: Allocate & add to range table

- Suspect entry point arrives
- Checkpoint device
- Marshal required data in SFI
- Populate range table
- Execute & Populate compensation log
- Success: Copy back written data

Fine-Grained Isolation



Address	Access rights
oxffffaooo	Read
oxffffaoo8	Write
oxffffaooa	Read

- ★ Suspect entry point arrives
- ★ Checkpoint device and processor state

FGFT provides transactional execution of driver entry points

- ★ Fail: Restore processor and device state, release locks

Recovery speedup

Driver	Class	Bus	Restart recovery	FGFT recovery	Speedup
8139too	net	PCI	0.31s	70µs	4400
e1000	net	PCI	1.80s	295ms	6
r8169	net	PCI	0.12s	40µs	3000
pegasus	net	USB	0.15s	5ms	30
ens1371	sound	PCI	1.03s	115ms	9
psmouse	input	serio	0.68s	410ms	1.65

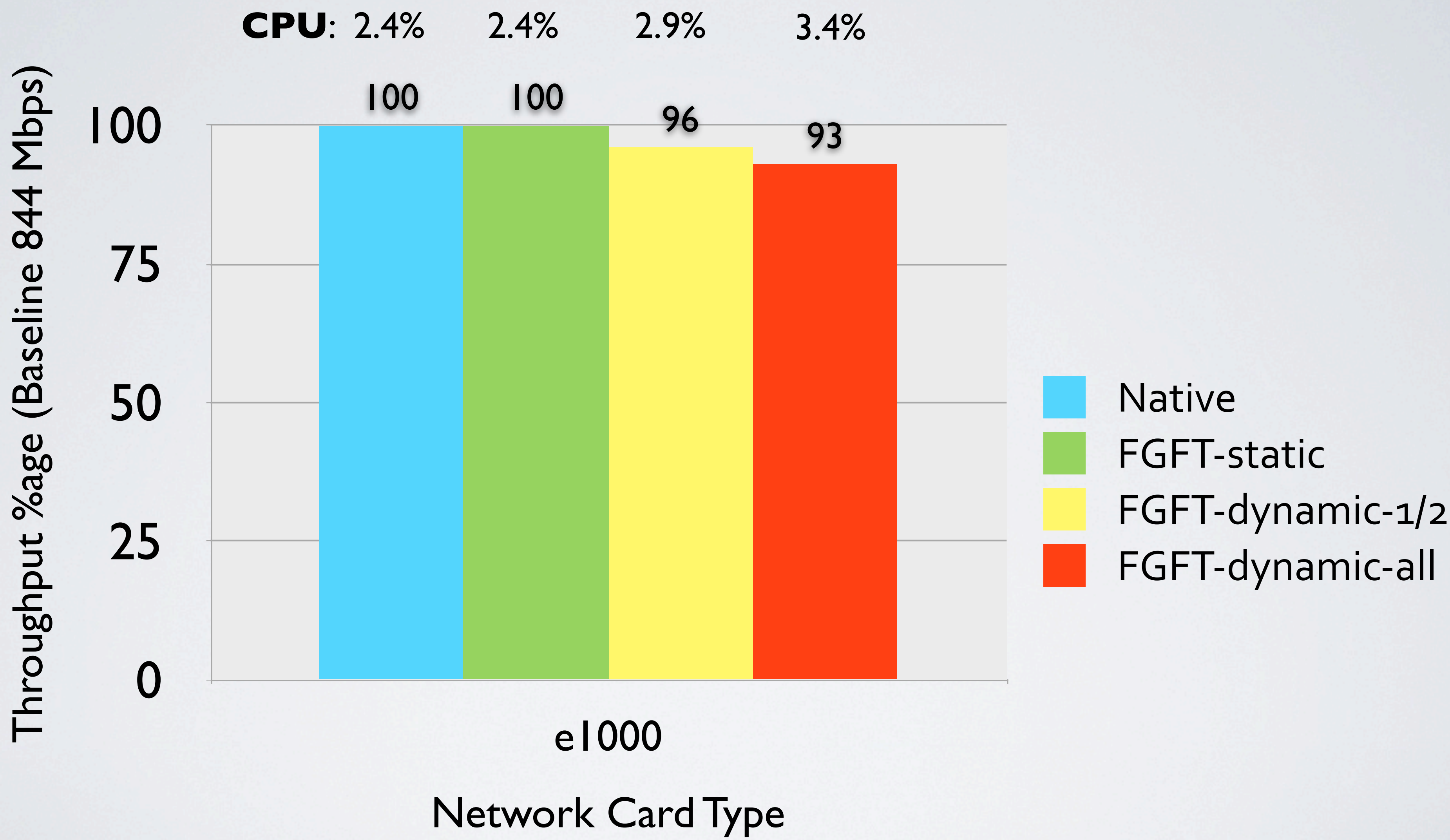
FGFT provides speedup in driver recovery

Programming effort

Driver	LOC	Recovery additions	
		LOC Moved	LOC Added
8139too	1, 904	26	4
e1000	13, 973	32	10
r8169	2, 993	17	5
pegasus	1, 541	22	5
ens1371	2, 110	16	6
psmouse	2, 448	19	6

FGFT requires limited annotation support and needs only 38 lines of new kernel code

Throughput overhead



netperf on Intel quad-core machines

Summary

- ★ **Investigated the problem of device failures in OS**
- ★ **Developed static and runtime solutions, contributed patches and a talk to developer community**
- ★ **Took a holistic view of research solutions and identified new research opportunities**
- ★ **Addressed one of these findings, and introduced checkpoint/restore in modern drivers for fast recovery**

Outline

Tolerate device failures

Understand drivers and potential opportunities

Transactional approach for cheap recovery

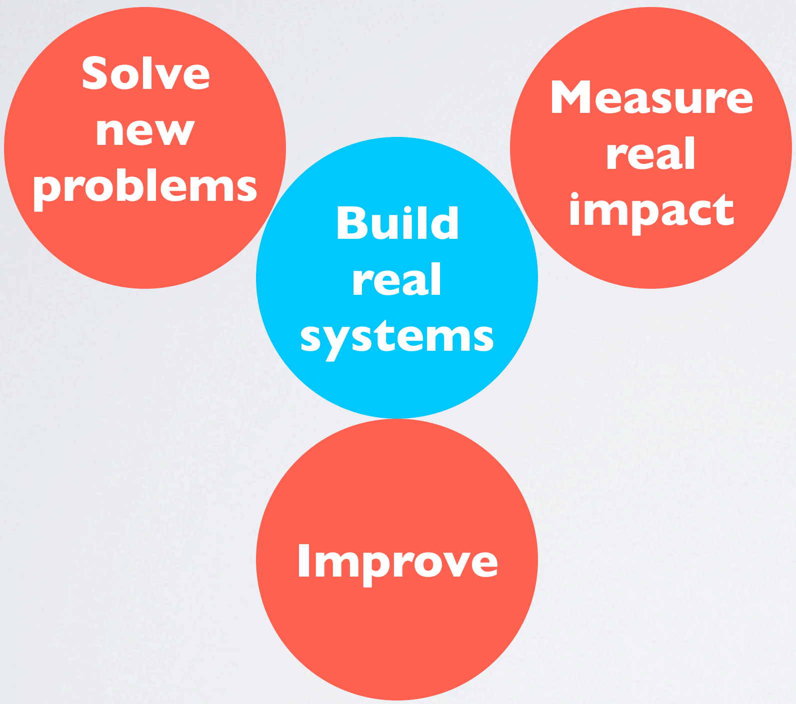
Checkpoint/restore
FGFT
Other/Future Work

Other work

	Reliability	Performance	Measurement
Storage	Differential RAID [Eurosys '10]	GPFS ThinCloud [Under Submission]	
Drivers	SymDrive [OSDI '12] FGFT [ASPLOS '13] Carburizer [SOSP '09]	Live Migration [OSR '09]	Driver study [ASPLOS '12]

Papers at http://cs.wisc.edu/~kadav

Future Work



- ★ **Use prior experience in**
  - ★ **Operating Systems**
  - ★ **Distributed Systems**
  - ★ **Software Reliability**
  - ★ **Program Analysis**

Future Work: Lessons from reliability research

★ **Distributed Systems: Identify and automatically fix cluster specific issues: expired leases, stale views, flooding (cascading failures)**

★ **Distributed Systems: How to create lightweight, broad and consistent checkpoints?**

★ **Automatically fix problems in other plugin based architectures like app stores, browsers**

Future Work: Investigate OS-hardware co-design

- **Co-design: Co-design OS and device abstractions**
  - **Integrating energy proportional DRAM in OS**
  - **Use special purpose workloads to accelerate cloud workloads**
  - **Re-design I/O in clusters for remote access**

- **Co-verification: Device protocol violations**
  - **Extend existing work on device failures to detect inconsistencies in software-device interaction**

Example: Energy Proportional DRAM

- **Goal: Co-design virtual memory and newer low power DRAM (such as Partial Array Self-Refresh)**
- **Evidence:**
  - **Workloads heterogenous show huge variance in memory demands (Google [SOCC '12])**
- **Problem: OS aggressively uses memory for performance**
  - **Consumes all memory as page cache**
  - **Fragments address space making consolidation difficult**
- **How do we re-design OS and DRAM chips to save power?**
  - **Where?: Reliable last level cache interface**
  - **Virtual memory integration: Ensure transparency**
  - **De-fragmentation: Energy-aware page migration**

Questions?

Asim Kadav

- **http://cs.wisc.edu/~kadav**

Thanks to

- **Michael Swift**
- **Matt Renzelmann**
- **Mahesh Balakrishnan**
- **Dahlia Malkhi**
- **Vijayan Prabhakaran**
- **Ed Nightingale**
- **Jeremy Elson**
- **James Mickens**
- **Rathijit Sen**