Understanding and Improving Device Access Complexity

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

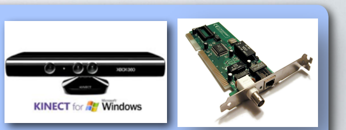
Devices enrich computers

- Keyboard
- Sound
- Printer
- Network
- Storage

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

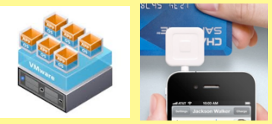
Huge growth in number of devices

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

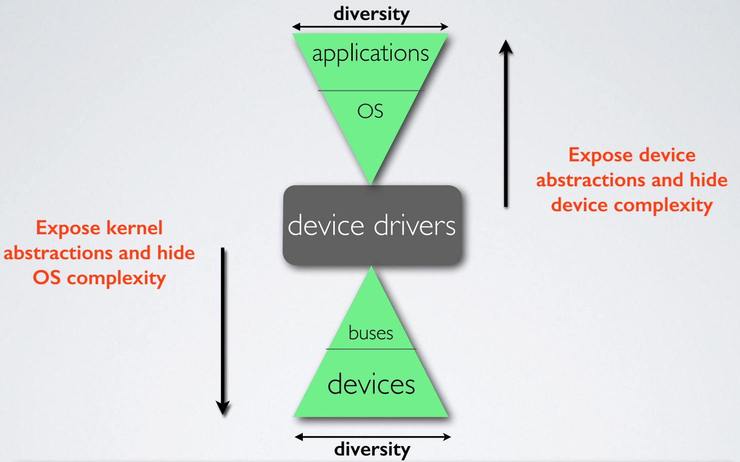


Many buses: USB, PCI-e, thunderbolt

Heterogeneous OS support: 10G ethernet vs card readers



Device drivers: OS interface to devices



Expose device abstractions and hide device complexity

Expose kernel abstractions and hide OS complexity

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

Evolution of devices hurts device access

Tools and mechanisms to address increasing device complexity

Simplicity

Low latency

Reliability

Efficient device support in OS

Cost effective

Growth in number and diversity

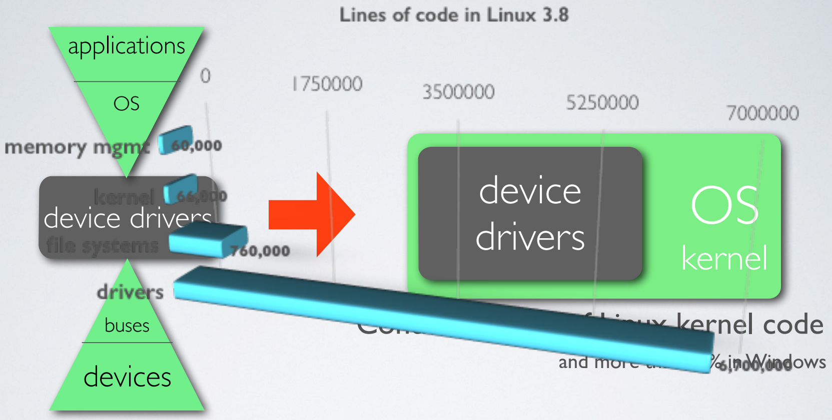
Run in challenging environments

Hardware failures (like CMOS issues)

Complex firmware and configuration modes

Evolution of devices

Growth in drivers hurts understanding of drivers



Understand the software complexity and improve driver code

Last decade: Focus on the driver-kernel interface

3rd party developers

+

Recipe for disaster

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
	CuriOS [OSDI 08]	2	1	2
Type Safety	SafeDrive [OSDI 06]	6	2	3

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

Design goal: Complete solution that limits kernel

Goal: Address software and hardware complexity

- ★ **Understand and improve device access 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

Narrow approach and solve specific problems in all drivers — Tolerate device failures

Broad approach and have a holistic view of all drivers — Understand drivers and potential opportunities

Known approach and apply to all drivers — Transactional approach for low latency recovery

Minimize kernel changes and apply to all drivers

Contributions/Outline

SOSP '09
First research consideration of hardware failures in drivers — Tolerate device failures

ASPLOS '12
Largest study of drivers to understand their behavior and verify research assumptions — Understand drivers and potential opportunities

ASPLOS '13
Introduce checkpoint/restore in drivers for low latency fault tolerance — Transactional approach for low latency recovery

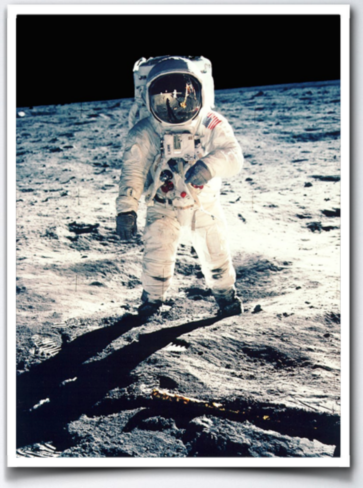
---

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**

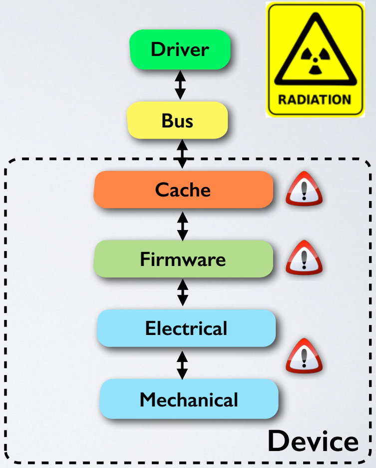
```
while (ioread16(ioaddr + Wn7_MasterStatus))
        & 0x8000);
```

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 interference, radiation, heat



Sources of hardware misbehavior

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

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

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	●	●	●	
	Stuck interrupt		●		
	Lost request			●	
	Avoid excess delay in OS			●	
	Unexpected events	●		●	
Reporting	Report all failures	●	●	●	
Recovery	Handle all failures		●	●	

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

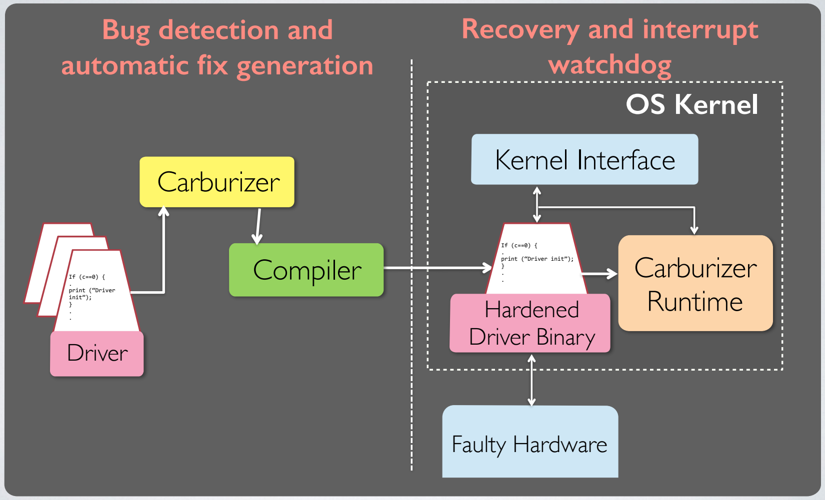
Static analysis component

- **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 array reference

Unsafe pointer reference

System panic calls

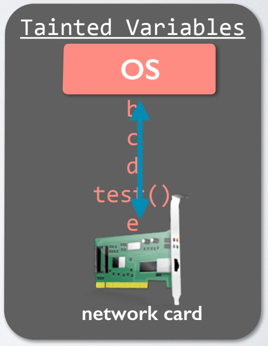
Finding sensitive code

⋆ **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/inw
- ⋆ **Memory-mapped I/O :** readl/readw
- ⋆ **DMA buffers**
- ⋆ **Data from USB packets**



Detecting risky uses of tainted variables

⋆ 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
  - ⋆ Extra analyses to existing timeouts

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**
- ⋆ **Detect existing range/not NULL checks**

```
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)

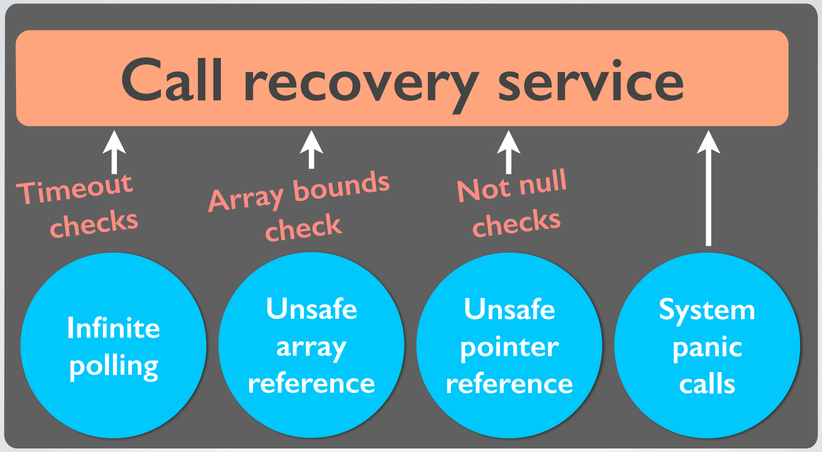
Analysis results over the Linux kernel

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

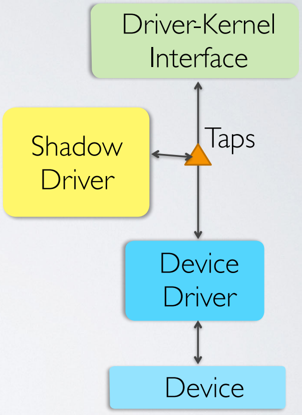
- ⋆ **Analyzed/Built 6300 driver files (2.8 million LOC) in 37 min**
- ⋆ **Found 992 hardware dependence bugs in driver code**
- ⋆ **False positive rate: 7.4% (manual sampling of 190 bugs)**

Repairing drivers



Runtime fault recovery : Shadow drivers

- **Carburizer calls generic recovery service if check fails**
- **Low cost transparent recovery**
  - ⋆ Based on shadow drivers
  - ⋆ Records state of driver at all times
  - ⋆ Transparently restarts and replays recorded state 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 and performance

- Synthetic fault injection on network drivers

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

- < 0.5% throughput overhead and no CPU overhead with network drivers

Carburizer failure detection and transparent recovery works and has very low overhead

---

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

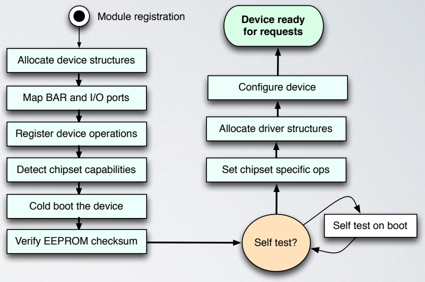
Contributions beyond research

- Linux Plumbers Conference [Sep ‘11]
- 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`

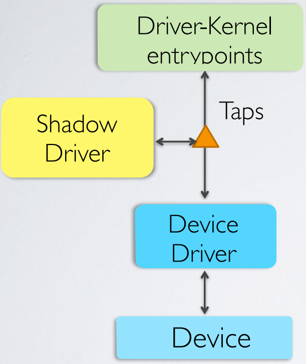
Recovery performance: device initialization is slow

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



- What does slow device re-initialization hurt?
  - Fault tolerance: Driver recovery
  - Virtualization: Live migration, cloning
  - OS functions: Boot, upgrade

Recovery functionality: assumes drivers follow class behavior



- Kernel exports standard entry points for every class (like “packet send” for network class)
- Shadow drivers records state by interposing class defined entry points
- Recovery = Restart and replay of captured state
- Do drivers have additional state?

How many drivers obey class behavior?

Outline

Tolerate device failures

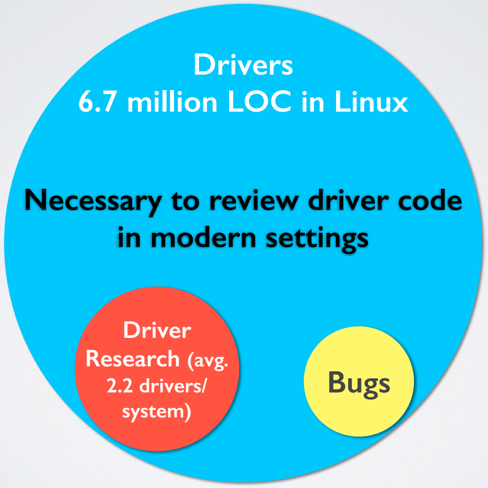
Understand drivers and potential opportunities

Overview

Recovery specific results

Transactional approach for cheap recovery

Our view of drivers is narrow



Drivers
6.7 million LOC in Linux

Necessary to review driver code in modern settings

Understanding Modern Device Drivers[ASPLOS 2012]

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

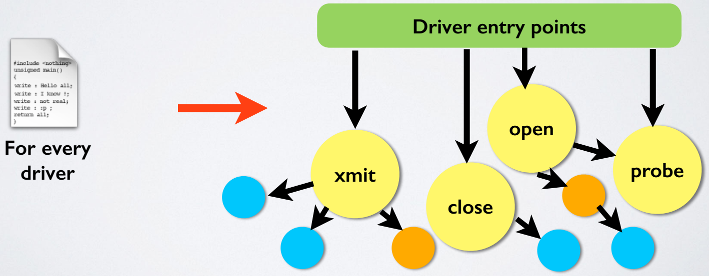
Driver properties	Driver interaction	Driver similarity
★ **Code properties** ★ **Verify research assumptions**	★ **Driver kernel & device interaction** ★ **Driver architecture**	★ **7 million lines of code needed?**

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, chipsets**
  - ★ **Bus properties & other properties (like module params)**
  - ★ **Driver functions registered as entry points (purpose)**



Study methodology

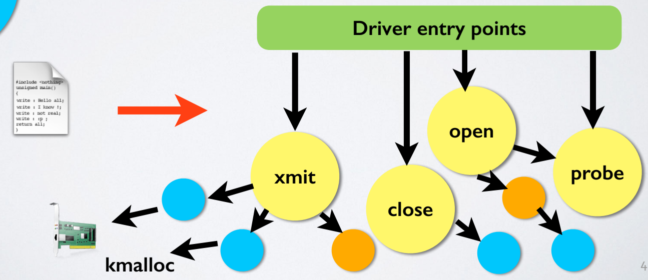
★ **Static source analysis of 3200 drivers in Linux 2.6.37.6 (May 2011)**

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

---

Contributions/Outline

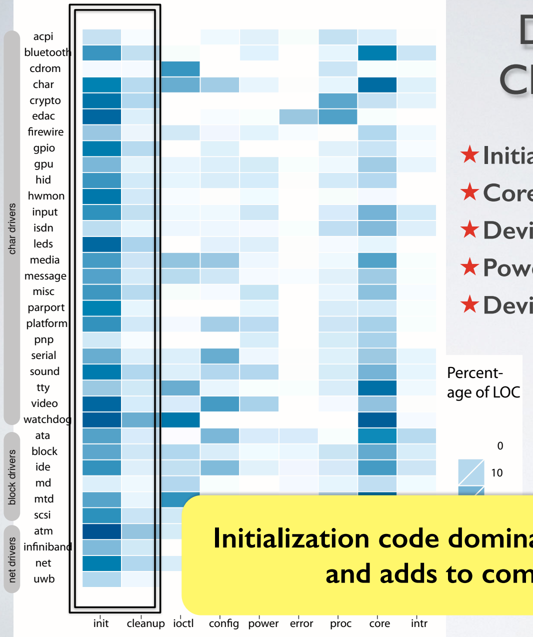
Tolerate device failures

Understand drivers and potential opportunities

Overview

Recovery specific results

Transactional approach for cheap recovery



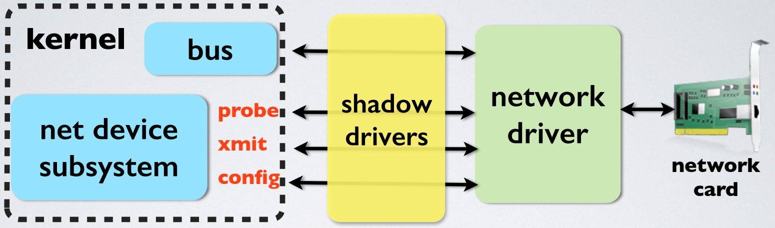
Driver Code Characteristics

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

Initialization code dominates driver LOC and adds to complexity

---

Problem 2: Shadow drivers assume drivers follow class behavior



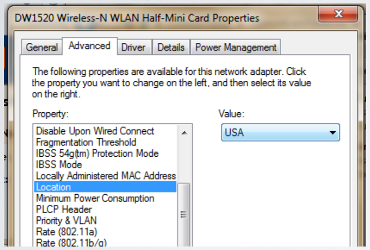
★ Class definition includes:

- ★ Callbacks registered with the bus, device and kernel subsystem

How many drivers follow class behavior and how much code does this add?

Problem 2(a): Drivers do behave outside class definitions

- ★ Non-class behavior in device drivers:
  - module parameters, unique ioctls, procfs/sysfs interactions



The following properties are available for this network adapter. Click the property you want to change on the left, and then select its value on the right.

Property:
Disable Upon Wired Connect
Fragmentation Threshold
IBSS 54g(tm) Protection Mode
IBSS Mode
Locally Administered MAC Address
Location
Minimum Power Consumption
PLCP Header
Priority & VLAN
Rate (802.11a)
Rate (802.11b/g)

Value: USA

Windows WLAN card
config via private ioctls

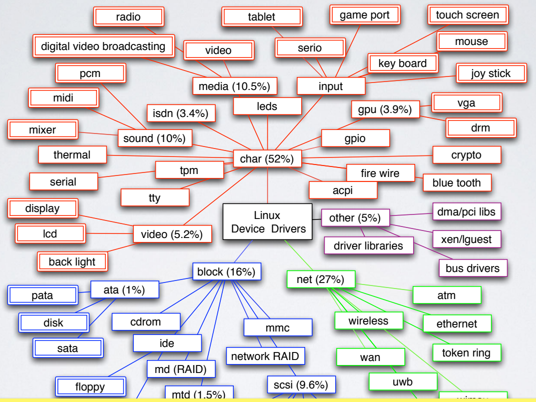
```
$ echo 1 > /sys/class/sound/mixer/
device/enable
```

Linux sound card config via sysfs

Overall 44% of drivers have non-class behavior and research making this assumption will not apply

---

Problem 2(b): Too many classes



Class-specific driver recovery leads to a large kernel recovery subsystem

★ "Understanding Modern Device Drivers" ASPLOS 2012

Few other results

Driver properties

- ★ Many assumptions made by driver research does not hold:
  - ★ 44% of drivers do not obey class behavior
  - ★ 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

★ More results in "Understanding Modern Device Drivers" ASPLOS 2012

Outline

Tolerate device failures

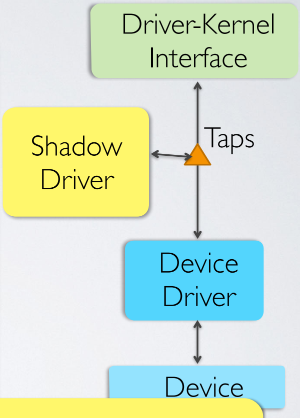
Understand drivers and potential opportunities

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)
  - Incomplete: Unique device semantics not captured
  - Hard: Need to be written for every class of drivers
  - Large changes: Introduces new, large kernel subsystem

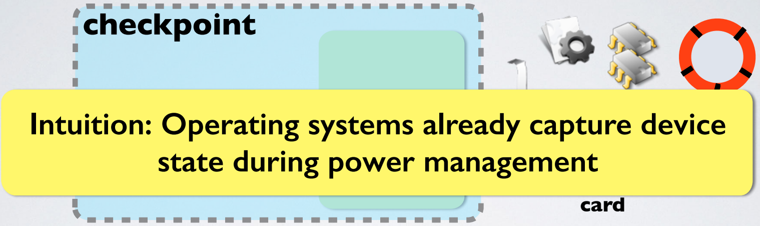


Checkpoint/restore of device and driver state removes the need to reboot device and replay state

---

Checkpointing drivers is hard

- Easy to capture **memory** state



Intuition: Operating systems already capture device state during power management

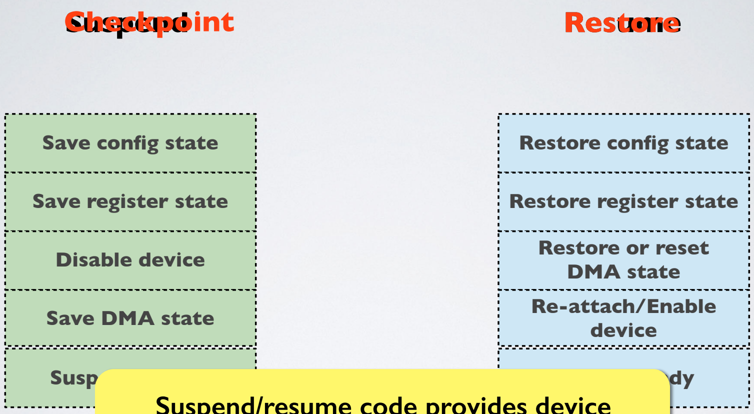
- Device state is not captured
  - Device configuration space
  - Internal device registers and counters
  - Memory buffer addresses used for DMA
- Unique for every device

Intuition with power management

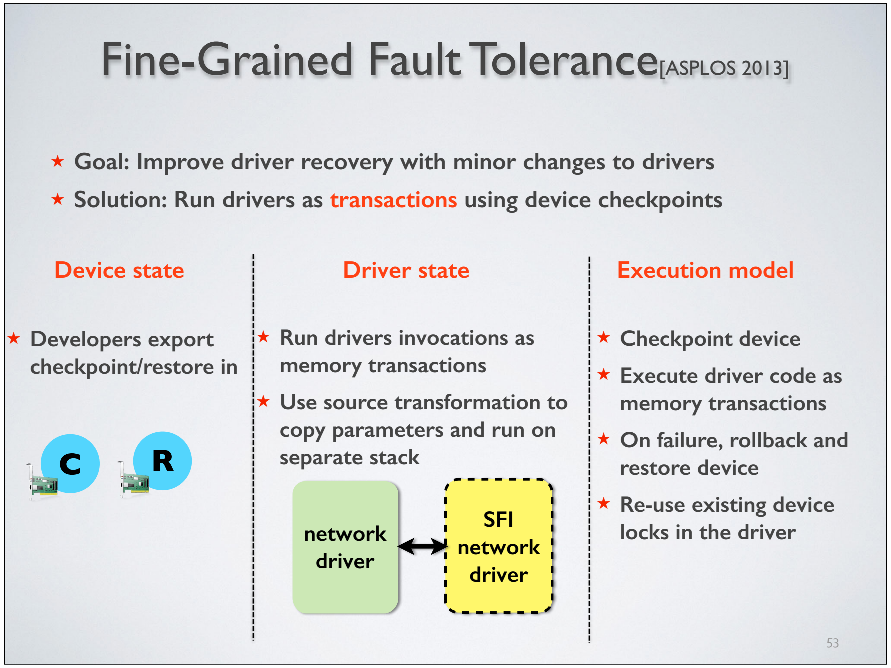
- Refactor power management code for device checkpoints
  - Correct: Developer captures unique device semantics
  - Fast: Avoids probe and latency critical for applications
- Ask developers to export checkpoint/restore in their drivers

---

Device checkpoint/restore from PM code



Suspend/resume code provides device checkpoint functionality



Fine-Grained Fault Tolerance[ASPLOS 2013]

- Goal: Improve driver recovery with minor changes to drivers
- Solution: Run drivers as transactions using device checkpoints

Device state

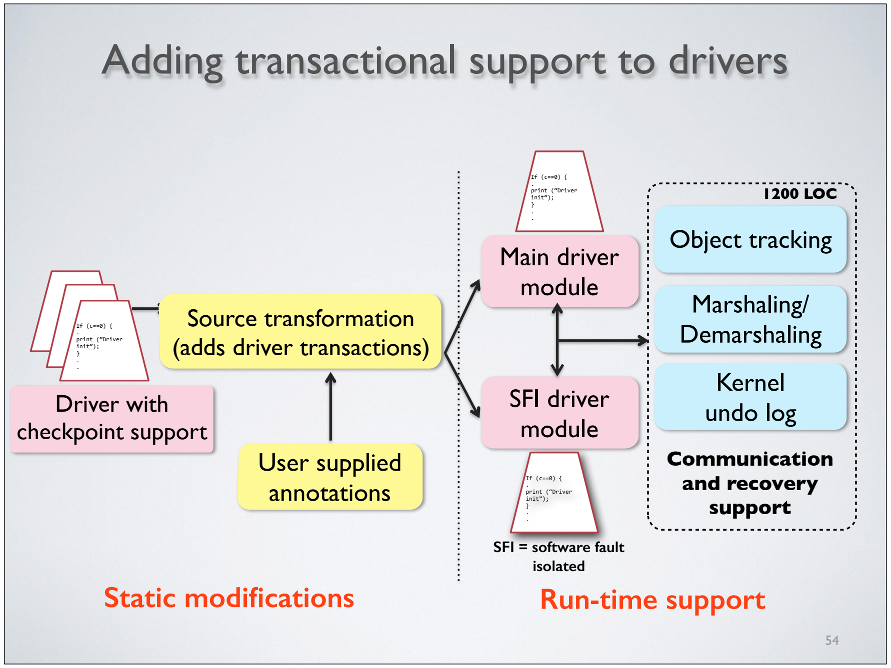
- Developers export checkpoint/restore in

Driver state

- Run drivers invocations as memory transactions
- Use source transformation to copy parameters and run on separate stack

Execution model

- Checkpoint device
- Execute driver code as memory transactions
- On failure, rollback and restore device
- Re-use existing device locks in the driver

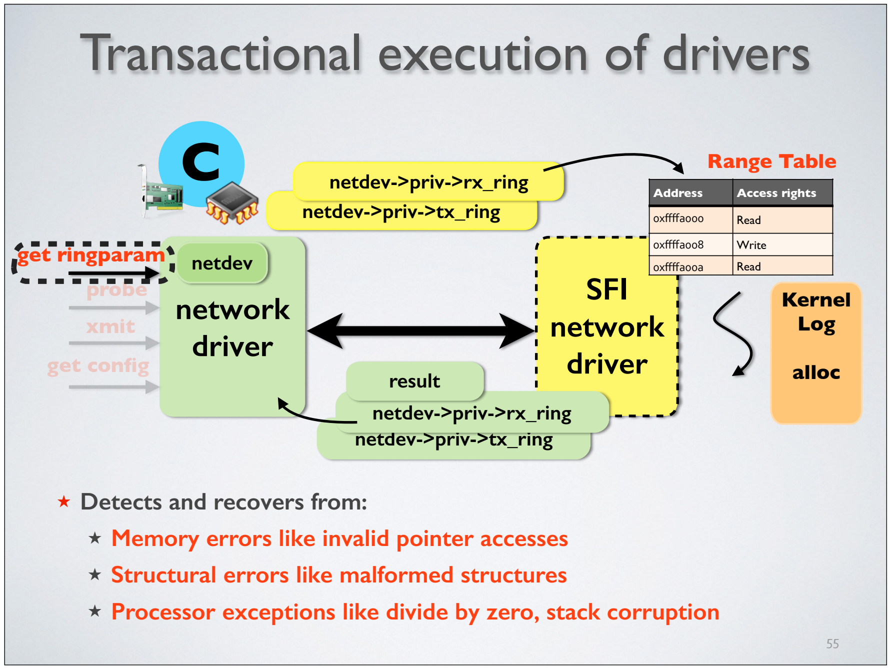


Adding transactional support to drivers

SFI = software fault isolated

Static modifications

Run-time support



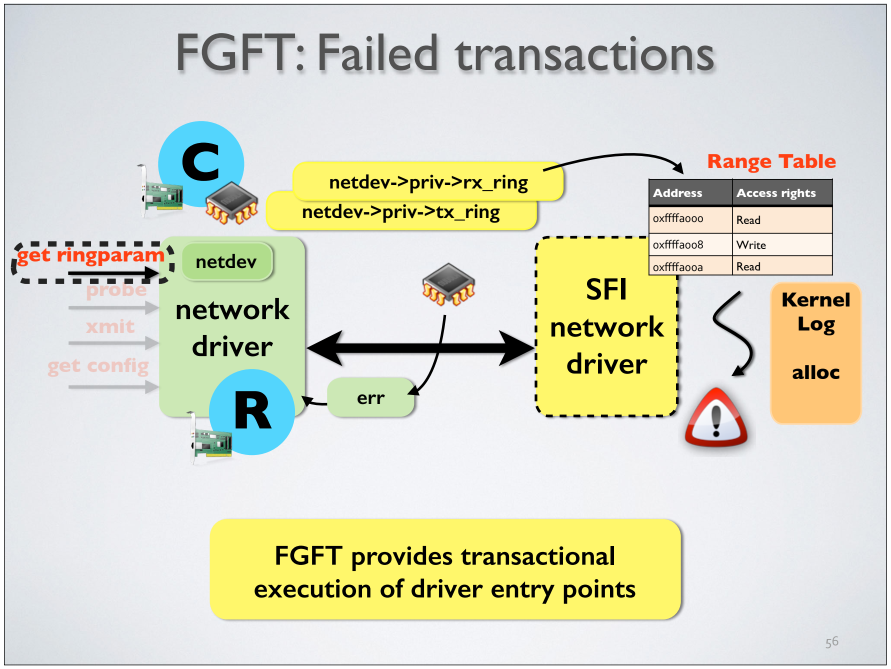
Transactional execution of drivers

Range Table

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

- Detects and recovers from:
  - Memory errors like invalid pointer accesses
  - Structural errors like malformed structures
  - Processor exceptions like divide by zero, stack corruption

---



FGFT: Failed transactions

Range Table

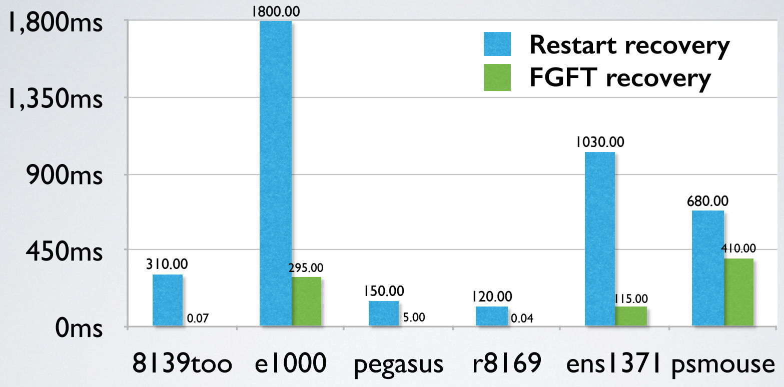
Address	Access rights
0xffffa000	Read
0xffffa008	Write
0xffffa00a	Read

FGFT provides transactional execution of driver entry points

How does this give us transactional execution?

- ★ Atomicity: All or nothing execution
  - ★ Driver state: Run code in SFI module
  - ★ Device state: Explicitly checkpoint/restore state
- ★ Isolation: Serialization to hide incomplete transactions
  - ★ Re-use existing device locks to lock driver
  - ★ Two phase locking
- ★ Consistency: Only valid (kernel, driver and device) states
  - ★ Higher level mechanisms to rollback external actions
  - ★ At most once device action guarantee to applications

Recovery speedup



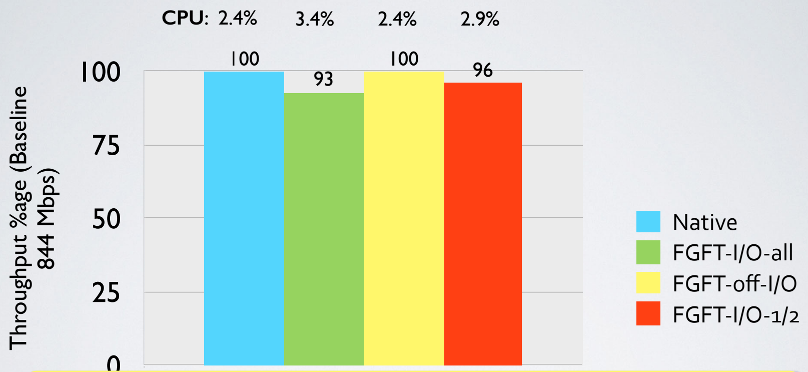
FGFT provides significant speedup in driver recovery and improves system availability

Programming effort

Driver	LOC	Checkpoint/restore effort	
		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 programmer effort and needs only 38 lines of new kernel code

Throughput with isolation and recovery



FGFT can isolate and recover high bandwidth devices at low overhead without adding kernel subsystems

netperf on Intel quad-core machines

Talk summary

First research consideration of hardware failures in drivers

SOSP '09

Released tool, patches & informed developers

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

ASPLOS '12

Measured driver behavior & identified new directions

Introduced checkpoint/restore in drivers for low latency fault tolerance

ASPLOS '13

Fast & correct recovery with incremental changes to drivers

Outline

Tolerate device failures

Understand drivers and potential opportunities

Transactional approach for cheap recovery

Checkpoint/restore
FGFT
Other Work

Future directions in device access

- **Most new devices will continue to be accessed using the traditional driver architecture ...**
  - **Complex bugs like device protocol violations**
- **But we will see new device architectures for specific device requirements (like low latency) ...**
  - **PCM does not require caching and scheduling from kernel**
- **or specific environments (like remote I/O for clusters).**
  - **Provide I/O architecture for single fabric computers**
- **OS researchers have an opportunity to think across layers**
  - **Co-design low power DRAMs with VM subsystem**

Other work

	Single system	Distributed systems	
Storage	**Differential RAID** **[Eurosys ‘10]**	**IBM GPFS** **ThinCloud** **[NSDI ‘14]**	
Drivers	**SymDrive** **[OSDI ‘12]** **FGFT** **[ASPLOS ‘13]** **Carburizer** **[SOSP ‘09]**	**Live Migration** **[OSR ‘09]**	**Driver study** **[ASPLOS ‘12]**
	Reliability	Performance	Measurement

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

Questions

Thanks to all my collaborators

Asim Kadav

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

Future work (II) : PL support for large scale processing

- Trends with large systems and workloads
  - Too hard : Difficult to get right
  - Too adhoc : Lack of structure for performance reasons
  - Too much data : Hard to stress test/test completely
  - Too whimsical : Hard to model w/o perturbation

- Opportunity for language support
  - Efficient representation, access and analysis
  - Programmability vs resource usage estimation
  - Reliability: Violations and debugging

Opportunity to provide language support to aid lack of structure, programmability and control

---

Future work (I): OS/hardware boundaries

- Hardware getting more specialized/interesting
  - New co-processors, low power modes, virtualization support, replicate OS functionality

- New device subsystems for specific device requirements (like low latency) ...
  - PCM does not require caching and scheduling from kernel

- or specific environments (like remote I/O for clusters).
  - Provide I/O architecture for single fabric computers

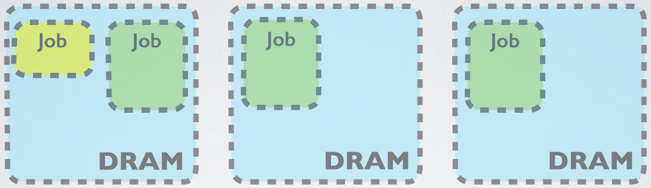
- OS researchers have an opportunity to think across layers
  - Co-design low power DRAMs with VM subsystem

Questions

Asim Kadav

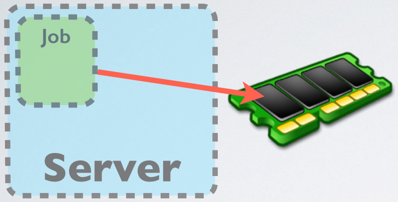
- **http://cs.wisc.edu/~kadav**
- **kadav@cs.wisc.edu**

Example: Energy Proportional DRAM



- Workloads show variance in memory needs (Google [SOCC '12])
- How do we integrate low power DRAM modes that can be turned off partially?

Integrate new DRAM power modes with OS



- Problem: OS aggressively uses DRAM 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

Some future directions

- Trends with new devices and new workloads
  - Faster co-processors, new memory technologies (performance and power), low latency network cards
  - OS vendors more open in adapting hardware to software
- Operating Systems: Develop OS and application abstractions
  - Scaling network performance
  - Integrating low power/latency devices in OS
  - Re-design I/O in clusters for remote access
- Software reliability in cloud services
  - Identify and automatically fix cluster specific issues: expired leases, stale views, flooding (cascading failures)
  - Debugging using replay techniques

Future Work: Better OS-hardware integration

- Trends with new devices
  - Fast co-processors, new memory, low latency network cards
  - OS vendors more open in adapting hardware to software
- Co-design: Develop OS and device abstractions
  - Integrating low power DRAM in OS
  - Re-design I/O in clusters for remote access
- Co-verification: Detect violation of device protocols
  - Thousands of devices with different models
  - Automatically detect inconsistencies in protocol implementation