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
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
Huge growth in number of devices

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

Many buses: USB, PCI-e, thunderbolt
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
Device drivers: OS interface to devices

Expose device abstractions and hide device complexity
Device drivers: OS interface to devices

- Expose kernel abstractions and hide OS complexity
- Expose device abstractions and hide device complexity

Allow diverse set of applications and OS services to access diverse set of devices
Evolution of devices hurts device access

Efficient device support in OS

Evolution of devices
Evolution of devices hurts device access

- Simplicity
- Low latency
- Reliability
- Cost effective

Efficient device support in OS
Evolution of devices hurts device access

- Simplicity
- Reliability
- Low latency
- Cost effective

Growth in number and diversity
Run in challenging environments
Complex firmware and configuration modes
Hardware failures (like CMOS issues)

Efficient device support in OS
Evolution of devices hurts device access

Tools and mechanisms to address increasing device complexity

- Growth in number and diversity
- Run in challenging environments
- Hardware failures (like CMOS issues)
- Complex firmware and configuration modes

Simplicity
Reliability
Low latency
Cost effective

Efficient device support in OS

Evolution of devices
Growth in drivers hurts understanding of drivers
Growth in drivers hurts understanding of drivers

Contribute 60% of Linux kernel code
and more than 90% in Windows
Growth in drivers hurts understanding of drivers

Lines of code in Linux 3.8

- memory mgmt: 60,000
- kernel: 66,000
- file systems: 760,000
- drivers: 6,700,000
Growth in drivers hurts understanding of drivers.

Lines of code in Linux 3.8:

- **Drivers**: 6,700,000
- **File systems**: 760,000
- **Kernel**: 66,000
- **Memory mgmt**: 60,000

Understand the software complexity and improve driver code.
Last decade: Focus on the driver-kernel interface
Last decade: Focus on the driver-kernel interface

3rd party developers + device drivers → OS kernel
Last decade: Focus on the driver-kernel interface

3rd party developers + device drivers = OS kernel

Recipe for disaster
Re-use lessons from existing driver research

<table>
<thead>
<tr>
<th>Improvement</th>
<th>System</th>
<th>Validation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Drivers</td>
<td>Bus</td>
</tr>
<tr>
<td>New functionality</td>
<td>Shadow driver migration [OSR09]</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>RevNIC [Eurosys 10]</td>
<td>1</td>
</tr>
<tr>
<td>Reliability</td>
<td>Nooks [SOSP 03]</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>XFI [OSDI 06]</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>CuriOS [OSDI 08]</td>
<td>2</td>
</tr>
<tr>
<td>Type Safety</td>
<td>SafeDrive [OSDI 06]</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>Singularity [Eurosys 06]</td>
<td>1</td>
</tr>
<tr>
<td>Specification</td>
<td>Nexus [OSDI 08]</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Termite [SOSP 09]</td>
<td>2</td>
</tr>
<tr>
<td>Static analysis tools</td>
<td>Windows SDV [Eurosys 06]</td>
<td>Many</td>
</tr>
<tr>
<td></td>
<td>Coverity [CACM 10]</td>
<td>All</td>
</tr>
<tr>
<td></td>
<td>Cocinelle [Eurosys 08]</td>
<td>All</td>
</tr>
</tbody>
</table>
## Re-use lessons from existing driver research

Large kernel subsystems and validity of few device types result in limited adoption of research solutions.
## Re-use lessons from existing driver research

<table>
<thead>
<tr>
<th>Improvement</th>
<th>System</th>
<th>Validation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Drivers</td>
</tr>
<tr>
<td>New functionality</td>
<td>Shadow driver migration [OSR09]</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>RevNIC [Eurosys 10]</td>
<td>1</td>
</tr>
<tr>
<td>Reliability</td>
<td>Nooks [SOSP 03]</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>XFI [OSDI 06]</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>CuriOS [OSDI 08]</td>
<td>2</td>
</tr>
<tr>
<td>Type Safety</td>
<td>SafeDrive [OSDI 06]</td>
<td>6</td>
</tr>
<tr>
<td>Static analysis tools</td>
<td>Windows SDV [Eurosys 06]</td>
<td>Many</td>
</tr>
<tr>
<td></td>
<td>Coverity [CACM 10]</td>
<td>All</td>
</tr>
<tr>
<td></td>
<td>Cocinelle [Eurosys 08]</td>
<td>All</td>
</tr>
</tbody>
</table>

"Limited kernel changes + Applicable to lots of drivers => Real Impact"
Re-use lessons from existing driver research

<table>
<thead>
<tr>
<th>Improvement</th>
<th>System</th>
<th>Validation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Drivers</td>
</tr>
<tr>
<td>New functionality</td>
<td>Shadow driver migration [OSR09]</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>RevNIC [Eurosys 10]</td>
<td>1</td>
</tr>
<tr>
<td>Reliability</td>
<td>Nooks [SOSP 03]</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>XFI [OSDI 06]</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>CuriOS [OSDI 08]</td>
<td>2</td>
</tr>
<tr>
<td>Type Safety</td>
<td>SafeDrive [OSDI 06]</td>
<td>6</td>
</tr>
</tbody>
</table>

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

Design goal: Complete solution that limits kernel changes and applies to all drivers
Goal: Address software and hardware complexity

★ Understand and improve device access in the face of rising hardware and software complexity
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
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
- Increasing hardware complexity
  - Low latency device availability
Goal: Address software and hardware complexity

- **Understand and improve device access in the face of rising hardware and software complexity**

1. Increasing hardware complexity
   - Reliability against hardware failures
2. Increasing hardware complexity
   - Low latency device availability
3. Increasing software complexity
   - Better understanding of driver code
First research consideration of hardware failures in drivers

Tolerate device failures

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

Understand drivers and potential opportunities

Introduce checkpoint/restore in drivers for low latency fault tolerance

Transactional approach for low latency recovery

SOSP '09

ASPLOS '12

ASPLOS '13
What happens when devices misbehave?
What happens when devices misbehave?

★ Drivers make it better
What happens when devices misbehave?

- Drivers make it better
- Drivers make it worse
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
Current state of OS-hardware interaction

2013

* Many device drivers often assume device perfection
  - Common Linux network driver: 3c59x.c
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);
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);
Current state of OS-hardware interaction

2013

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

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

Hardware dependence bug: Device malfunction can crash the system
Sources of hardware misbehavior

- Device
  - Bus
  - Cache
  - Firmware
  - Electrical
  - Mechanical
  - Driver
Sources of hardware misbehavior

- **Sources of hardware misbehavior**
- **Firmware/Design bugs**
Sources of hardware misbehavior

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

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

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

An evidence:

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

An evidence:

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%

How do hardware dependence bugs manifest?
How do hardware dependence bugs manifest?

Drivers use device data in critical control and data paths.

`printk("%s", msg[inb(regA)]);`
How do hardware dependence bugs manifest?

1. Drivers use device data in critical control and data paths
   ```c
   printk("%s", msg[inb(regA)]);
   ```

2. Drivers do not report device malfunction to system log
   ```c
   if (inb(regA)!= 5) {
       return; //do nothing
   }
   ```
How do hardware dependence bugs manifest?

1. Drivers use device data in critical control and data paths
   
   ```c
   printk("%s",msg[inb(regA)]);
   ```

2. Drivers do not report device malfunction to system log
   
   ```c
   if (inb(regA)!= 5) {
       return; //do nothing
   }
   ```

3. Drivers do not detect or recover from device failures
   
   ```c
   if (inb(regA)!= 5) {
       panic();
   }
   ```
# Vendor recommendations for driver developers

<table>
<thead>
<tr>
<th>Recommendation</th>
<th>Summary</th>
<th>Recommended by</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Intel</td>
</tr>
<tr>
<td><strong>Validation</strong></td>
<td>Input validation</td>
<td>●</td>
</tr>
<tr>
<td></td>
<td>Read once &amp; CRC data</td>
<td>●</td>
</tr>
<tr>
<td></td>
<td>DMA protection</td>
<td>●</td>
</tr>
<tr>
<td><strong>Timing</strong></td>
<td>Infinite polling</td>
<td>●</td>
</tr>
<tr>
<td></td>
<td>Stuck interrupt</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Lost request</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Avoid excess delay in OS</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Unexpected events</td>
<td></td>
</tr>
<tr>
<td><strong>Reporting</strong></td>
<td>Report all failures</td>
<td>●</td>
</tr>
<tr>
<td><strong>Recovery</strong></td>
<td>Handle all failures</td>
<td>●</td>
</tr>
<tr>
<td></td>
<td>Cleanup correctly</td>
<td>●</td>
</tr>
<tr>
<td></td>
<td>Do not crash on failure</td>
<td>●</td>
</tr>
<tr>
<td></td>
<td>Wrap I/O memory access</td>
<td>●</td>
</tr>
<tr>
<td>Recommendation</td>
<td>Summary</td>
<td>Intel</td>
</tr>
<tr>
<td>----------------</td>
<td>---------</td>
<td>-------</td>
</tr>
<tr>
<td>Validation</td>
<td>Input validation</td>
<td>✔️</td>
</tr>
<tr>
<td></td>
<td>Read once &amp; CRC data</td>
<td>✔️</td>
</tr>
<tr>
<td></td>
<td>DMA protection</td>
<td>✔️</td>
</tr>
<tr>
<td>Timing</td>
<td>Infinite polling</td>
<td>✔️</td>
</tr>
<tr>
<td></td>
<td>Stuck interrupt</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Lost request</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Avoid excess delay in OS</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Unexpected events</td>
<td></td>
</tr>
<tr>
<td>Reporting</td>
<td>Report all failures</td>
<td>✔️</td>
</tr>
<tr>
<td>Recovery</td>
<td>Handle all failures</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Cleanup correctly</td>
<td>✔️</td>
</tr>
<tr>
<td></td>
<td>Do not crash on failure</td>
<td>✔️</td>
</tr>
<tr>
<td></td>
<td>Wrap I/O memory access</td>
<td>✔️</td>
</tr>
</tbody>
</table>

Vendor recommendations for driver developers
## Vendor recommendations for driver developers

<table>
<thead>
<tr>
<th>Recommendation</th>
<th>Summary</th>
<th>Intel</th>
<th>Sun</th>
<th>MS</th>
<th>Linux</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Validation</strong></td>
<td>Input validation</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Read once &amp; CRC data</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td></td>
<td>DMA protection</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td><strong>Timing</strong></td>
<td>Infinite polling</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td></td>
<td>Stuck interrupt</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Lost request</td>
<td></td>
<td></td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Avoid excess delay in OS</td>
<td></td>
<td></td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Unexpected events</td>
<td>✔</td>
<td></td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td><strong>Reporting</strong></td>
<td>Report all failures</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td><strong>Recovery</strong></td>
<td>Handle all failures</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td></td>
<td>Cleanup correctly</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Do not crash on failure</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Wrap I/O memory access</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>
## Vendor recommendations for driver developers

<table>
<thead>
<tr>
<th>Recommendation</th>
<th>Summary</th>
<th>Recommended by</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Validation</strong></td>
<td></td>
<td>Intel</td>
</tr>
<tr>
<td>Validation</td>
<td>Input validation</td>
<td>•</td>
</tr>
<tr>
<td></td>
<td>Read once &amp; CRC data</td>
<td>•</td>
</tr>
<tr>
<td></td>
<td>DMA protection</td>
<td>•</td>
</tr>
<tr>
<td><strong>Timing</strong></td>
<td>Infinite polling</td>
<td>•</td>
</tr>
<tr>
<td></td>
<td>Stuck interrupt</td>
<td>•</td>
</tr>
<tr>
<td></td>
<td>Lost request</td>
<td>•</td>
</tr>
<tr>
<td></td>
<td>Avoid excess delay in OS</td>
<td>•</td>
</tr>
<tr>
<td></td>
<td>Unexpected events</td>
<td>•</td>
</tr>
<tr>
<td><strong>Reporting</strong></td>
<td>Report all failures</td>
<td>•</td>
</tr>
<tr>
<td><strong>Recovery</strong></td>
<td>Handle all failures</td>
<td>•</td>
</tr>
<tr>
<td></td>
<td>Cleanup correctly</td>
<td>•</td>
</tr>
<tr>
<td></td>
<td>Do not crash on failure</td>
<td>•</td>
</tr>
<tr>
<td></td>
<td>Wrap I/O memory access</td>
<td>•</td>
</tr>
</tbody>
</table>
## Vendor recommendations for driver developers

<table>
<thead>
<tr>
<th>Recommendation</th>
<th>Summary</th>
<th>Intel</th>
<th>Sun</th>
<th>MS</th>
<th>Linux</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Validation</strong></td>
<td>Input validation</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td></td>
<td>Read once &amp; CRC data</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td></td>
<td>DMA protection</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td><strong>Timing</strong></td>
<td>Infinite polling</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Stuck interrupt</td>
<td></td>
<td>•</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Lost request</td>
<td></td>
<td></td>
<td>•</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Avoid excess delay in OS</td>
<td></td>
<td></td>
<td></td>
<td>•</td>
</tr>
<tr>
<td></td>
<td>Unexpected events</td>
<td></td>
<td></td>
<td>•</td>
<td></td>
</tr>
<tr>
<td><strong>Reporting</strong></td>
<td>Report all failures</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td><strong>Recovery</strong></td>
<td>Handle all failures</td>
<td></td>
<td></td>
<td>•</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Cleanup correctly</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Do not crash on failure</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Wrap I/O memory access</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
</tbody>
</table>
## Vendor recommendations for driver developers

<table>
<thead>
<tr>
<th>Recommendation</th>
<th>Summary</th>
<th>Intel</th>
<th>Sun</th>
<th>MS</th>
<th>Linux</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Validation</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Input validation</td>
<td></td>
<td>●</td>
<td>●</td>
<td>MS</td>
<td>Linux</td>
</tr>
<tr>
<td>Read once&amp; CRC data</td>
<td></td>
<td>●</td>
<td>●</td>
<td></td>
<td>Linux</td>
</tr>
<tr>
<td>DMA protection</td>
<td></td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Timing</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Infinite polling</td>
<td></td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stuck interrupt</td>
<td></td>
<td></td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lost request</td>
<td></td>
<td></td>
<td></td>
<td>MS</td>
<td></td>
</tr>
<tr>
<td>Avoid excess delay in OS</td>
<td></td>
<td>●</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unexpected events</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Reporting</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Report all failures</td>
<td></td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Recovery</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Handle all failures</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Linux</td>
</tr>
</tbody>
</table>

**Goal:** Automatically implement as many recommendations as possible in commodity drivers
Goal: Tolerate hardware device failures in software through hardware failure detection and recovery
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
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

Bug detection and automatic fix generation

```c
if (c==0) {
  printf("Driver init\n");
}
```
Carburizer architecture

Bug detection and automatic fix generation

Carburizer

Compiler

Driver

If (c==0) {
  print ("Driver init");
}
If (c==0) {
  print ("Driver init");
}
Bug detection and automatic fix generation

Carburizer

Compiler

Driver

Recovery and interrupt watchdog

OS Kernel

Hardened Driver Binary

If (c==0) {
  print ("Driver init");
} ;

If (c==0) {
  print ("Driver Init");
} ;
Carburizer architecture

**Bug detection and automatic fix generation**

- Carburizer
- Compiler
- Driver

**Recovery and interrupt watchdog**

- OS Kernel
- Kernel Interface
- Carburizer Runtime
- Hardened Driver Binary
- Faulty Hardware
Hardening drivers

• **Goal: Remove hardware dependence bugs**
  - Find driver code that uses data from device
  - Ensure driver performs validity checks
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:**
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
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
**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
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

Types of device I/O

- Port I/O: `inb/inw`
- Memory-mapped I/O: `readl/readw`
- DMA buffers
- Data from USB packets
Finding sensitive code

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

```c
int test () {
```

Finding sensitive code

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

```c
int test () {
    a = read1();
}
```
Finding sensitive code

First pass: Identify tainted variables that contain data from device

```c
int test () {
    a = read1();
}
```

Tainted Variables

```
a
```
Finding sensitive code

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

```c
int test () {
    a = readl();
    b = inb();
}
```

Tainted Variables

- a
Finding sensitive code

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

```c
int test () {
    a = readl();
    b = inb();
}
```

Tainted Variables
```
    a
    b
```
Finding sensitive code

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

```c
int test () {
    a = readl();
    b = inb();
    c = b;
}
```

Tainted Variables

- a
- b
Finding sensitive code

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

```c
int test () {
    a = readl();
    b = inb();
    c = b;
}

Tainted Variables
    a
    b
    c
```
Finding sensitive code

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

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

Tainted Variables

- a
- b
- c
Finding sensitive code

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

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

Tainted Variables

- a
- b
- c
- d
Finding sensitive code

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

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

Tainted Variables
- a
- b
- c
- d
Finding sensitive code

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

```c
int test () {
    a = read1();
    b = inb();
    c = b;
    d = c + 2;
    return d;
}
```

Tainted Variables

- a
- b
- c
- d
- test()
Finding sensitive code

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

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

Tainted Variables:
- a
- b
- c
- d
- test()
Finding sensitive code

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

```c
int test () { 
    a = read1();
    b = inb();
    c = b;
    d = c + 2;
    return d;
}
int set() { 

Tainted Variables

a
b
c
d
test()
```
Finding sensitive code

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

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

Tainted Variables

- a
- b
- c
- d
- test()
Finding sensitive code

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

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

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

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

int set() {
    e = test();
}
```

Tainted Variables

- a
- b
- c
- d
- test()
- e
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

```c
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

```c
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

<table>
<thead>
<tr>
<th>Driver class</th>
<th>Infinite polling</th>
<th>Static array</th>
<th>Dynamic array</th>
<th>Panic calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>net</td>
<td>117</td>
<td>2</td>
<td>21</td>
<td>2</td>
</tr>
<tr>
<td>scsi</td>
<td>298</td>
<td>31</td>
<td>22</td>
<td>121</td>
</tr>
<tr>
<td>sound</td>
<td>64</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>video</td>
<td>174</td>
<td>0</td>
<td>22</td>
<td>22</td>
</tr>
<tr>
<td>other</td>
<td>381</td>
<td>9</td>
<td>57</td>
<td>32</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>860</strong></td>
<td><strong>43</strong></td>
<td><strong>89</strong></td>
<td><strong>179</strong></td>
</tr>
</tbody>
</table>

- 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)
Analysis results over the Linux kernel

<table>
<thead>
<tr>
<th>Driver class</th>
<th>Infinite polling</th>
<th>Static array</th>
<th>Dynamic array</th>
<th>Panic calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>net</td>
<td>117</td>
<td>2</td>
<td>21</td>
<td>2</td>
</tr>
<tr>
<td>scsi</td>
<td>298</td>
<td>31</td>
<td>22</td>
<td>121</td>
</tr>
<tr>
<td>sound</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>video</td>
<td></td>
<td></td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>other</td>
<td>381</td>
<td>9</td>
<td>57</td>
<td>32</td>
</tr>
<tr>
<td>Total</td>
<td>860</td>
<td>43</td>
<td>89</td>
<td>179</td>
</tr>
</tbody>
</table>

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

- Carburizer automatically generates repair code
  - Inserts failure detection and recovery service callout
Repairing drivers

★ Carburizer automatically generates repair code

★ Inserts failure detection and recovery service callout

- Infinite polling
- Unsafe array reference
- Unsafe pointer reference
- System panic calls
Repairing drivers

★ Carburizer automatically generates repair code
  ★ Inserts failure detection and recovery service callout

Timeout checks

- Infinite polling
- Unsafe array reference
- Unsafe pointer reference
- System panic calls
Repairing drivers

- Carburizer automatically generates repair code
  - Inserts failure detection and recovery service callout

- Timeout checks
- Array bounds check

- Infinite polling
- Unsafe array reference
- Unsafe pointer reference
- System panic calls
Repairing drivers

- Carburizer automatically generates repair code
- Inserts failure detection and recovery service callout

- Timeout checks
- Array bounds check
- Not null checks

- Infinite polling
- Unsafe array reference
- Unsafe pointer reference
- System panic calls
Repairing drivers

Call recovery service

- Timeout checks
- Array bounds check
- Not null checks

- Infinite polling
- Unsafe array reference
- Unsafe pointer reference
- System panic calls
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

```c
timeout = rdtsc111(start) + (cpu/khz/Hz)*2;
reg_val = read1(mmio + PHY_ACCESS);
while (reg_val & PHY_CMD_ACTIVE) {
    reg_val = read1(mmio + PHY_ACCESS);
    if (_cur < timeout)
        rdtsc111(_cur);
    else
        __recover_driver();
}
```

Timeout code added

*Code simplified for presentation purposes*
Carburizer automatically adds bounds checks

```c
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
Fault injection and performance

* Synthetic fault injection on network drivers

<table>
<thead>
<tr>
<th>Device/Driver</th>
<th>Original Driver</th>
<th>Carburizer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Behavior</td>
<td>Detection</td>
</tr>
<tr>
<td>3COM 3C905</td>
<td>CRASH</td>
<td>None</td>
</tr>
<tr>
<td>DEC DC 21x4x</td>
<td>CRASH</td>
<td>None</td>
</tr>
</tbody>
</table>
Fault injection and performance

- Synthetic fault injection on network drivers

<table>
<thead>
<tr>
<th>Device/Driver</th>
<th>Original Driver</th>
<th>Carburizer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Behavior</td>
<td>Detection</td>
</tr>
<tr>
<td>3COM 3C905</td>
<td>CRASH</td>
<td>None</td>
</tr>
<tr>
<td>DEC DC 21x4x</td>
<td>CRASH</td>
<td>None</td>
</tr>
</tbody>
</table>

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

- Synthetic fault injection on network drivers

<table>
<thead>
<tr>
<th>Device/Driver</th>
<th>Original Driver</th>
<th>Carburizer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Behavior</td>
<td>Detection</td>
</tr>
<tr>
<td>3COM 3C905</td>
<td>CRASH</td>
<td>None</td>
</tr>
<tr>
<td>DEC DC 21x4x</td>
<td>CRASH</td>
<td>None</td>
</tr>
</tbody>
</table>

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

Carburizer failure detection and transparent recovery works and has very low overhead.
<table>
<thead>
<tr>
<th>Recommendation</th>
<th>Summary</th>
<th>Intel</th>
<th>Sun</th>
<th>MS</th>
<th>Linux</th>
<th>Carburizer Ensures</th>
</tr>
</thead>
<tbody>
<tr>
<td>Validation</td>
<td>Input validation</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td></td>
<td>●</td>
</tr>
<tr>
<td></td>
<td>Read once &amp; CRC data</td>
<td>●</td>
<td>●</td>
<td></td>
<td>●</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DMA protection</td>
<td></td>
<td></td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Timing</td>
<td>Infinite polling</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td></td>
<td>●</td>
</tr>
<tr>
<td></td>
<td>Stuck interrupt</td>
<td></td>
<td></td>
<td>●</td>
<td></td>
<td>●</td>
</tr>
<tr>
<td></td>
<td>Lost request</td>
<td></td>
<td></td>
<td>●</td>
<td></td>
<td>●</td>
</tr>
<tr>
<td></td>
<td>Avoid excess delay in OS</td>
<td></td>
<td></td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Unexpected events</td>
<td></td>
<td></td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reporting</td>
<td>Report all failures</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td></td>
<td>●</td>
</tr>
<tr>
<td>Recovery</td>
<td>Handle all failures</td>
<td></td>
<td></td>
<td>●</td>
<td></td>
<td>●</td>
</tr>
<tr>
<td></td>
<td>Cleanup correctly</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td></td>
<td>●</td>
</tr>
<tr>
<td></td>
<td>Do not crash on failure</td>
<td></td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td></td>
<td>Wrap I/O memory access</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td></td>
<td>●</td>
</tr>
</tbody>
</table>
## Summary

<table>
<thead>
<tr>
<th>Recommendation</th>
<th>Summary</th>
<th>Intel</th>
<th>Sun</th>
<th>MS</th>
<th>Linux</th>
<th>Carburizer Ensures</th>
</tr>
</thead>
<tbody>
<tr>
<td>Validation</td>
<td>Input validation</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td></td>
<td>●</td>
</tr>
<tr>
<td></td>
<td>Read once &amp; CRC data</td>
<td>●</td>
<td>●</td>
<td></td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td></td>
<td>DMA protection</td>
<td>●</td>
<td></td>
<td>●</td>
<td></td>
<td>●</td>
</tr>
<tr>
<td>Timing</td>
<td>Infinite polling</td>
<td>●</td>
<td>●</td>
<td></td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td></td>
<td>Stuck interrupt</td>
<td></td>
<td>●</td>
<td></td>
<td></td>
<td>●</td>
</tr>
<tr>
<td></td>
<td>Lost request</td>
<td></td>
<td>●</td>
<td></td>
<td></td>
<td>●</td>
</tr>
<tr>
<td></td>
<td>Avoid excess delay in OS</td>
<td></td>
<td></td>
<td>●</td>
<td></td>
<td>●</td>
</tr>
<tr>
<td></td>
<td>Unexpected events</td>
<td></td>
<td></td>
<td>●</td>
<td></td>
<td>●</td>
</tr>
<tr>
<td>Reporting</td>
<td>Report all failures</td>
<td>●</td>
<td>●</td>
<td></td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td></td>
<td>Wrap I/O memory access</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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:
Recovery performance: device initialization is slow

- **Multi-second device probe**
  - Identify device
  - Cold boot device
  - Setup device/driver structures
  - Configuration/Self-test
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?
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
- Transactional approach for cheap recovery

Overview
Recovery specific results
Our view of drivers is narrow

Drivers
6.7 million LOC in Linux
Our view of drivers is narrow

Drivers

6.7 million LOC in Linux

Driver Research
(avg. 2.2 drivers/system)
Our view of drivers is narrow

Drivers
6.7 million LOC in Linux

Driver Research (avg. 2.2 drivers/system)

Bugs
Our view of drivers is narrow

Drivers
6.7 million LOC in Linux

Necessary to review driver code in modern settings

Driver Research
(avg. 2.2 drivers/system)

Bugs
Understanding Modern Device Drivers [ASPLOS 2012]
Study source of all Linux drivers for x86 (~3200 drivers)
Understanding Modern Device Drivers [ASPLOS 2012]

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

- Code properties
- Verify research assumptions
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
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)
Study methodology

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

★ Identify driver entry points, kernel and bus callouts
Study methodology

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

★ Identify driver entry points, kernel and bus callouts
  ★ Device class, sub-class, chipsets
Study methodology

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

★ Identify driver entry points, kernel and bus callouts
  ★ Device class, sub-class, chipsets
  ★ Bus properties & other properties (like module params)
Study methodology

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

★ 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)

- 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)

For every driver
Study methodology

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

★ 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)

For every driver

Driver entry points

xmit
open
probe
close
Study methodology

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

- Identify driver entry points, kernel and bus callouts

- 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)

★ Identify driver entry points, kernel and bus callouts

★ 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
- Transactional approach for cheap recovery

Overview
Recovery specific results
Driver Code
Characteristics
<table>
<thead>
<tr>
<th>Driver Code Characteristics</th>
<th>init</th>
<th>cleanup</th>
<th>ioctl</th>
<th>config</th>
<th>power</th>
<th>error</th>
<th>proc</th>
<th>core</th>
<th>intr</th>
</tr>
</thead>
<tbody>
<tr>
<td>acpi</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bluetooth</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cdrom</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>char</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>crypto</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>edac</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>firewire</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>gpio</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>gpu</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hid</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hwmon</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>input</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>isdn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>leds</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>media</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>message</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>misc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pnp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>serial</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sound</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tty</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>video</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>watchdog</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ata</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>block</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ide</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>md</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mtd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>scsi</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>atm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>infiniband</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>net</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>uwb</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Driver Code Characteristics

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

<table>
<thead>
<tr>
<th>Driver Code Characteristics</th>
<th>Percent-</th>
<th>age of LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>init</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>cleanup</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>ioctl</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>config</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>power</td>
<td>50</td>
<td></td>
</tr>
<tr>
<td>error</td>
<td></td>
<td></td>
</tr>
<tr>
<td>proc</td>
<td></td>
<td></td>
</tr>
<tr>
<td>core</td>
<td></td>
<td></td>
</tr>
<tr>
<td>intr</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Legend:**
- 0
- 10
- 20
- 30
- 40
- 50
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
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

- Windows WLAN card config via private ioctls
- Linux sound card config via sysfs

```
$ echo 1 > /sys/class/sound/mixer/device/enable
```
Problem 2(a): Drivers do behave outside class definitions

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

Windows WLAN card config via private ioctls

Linux sound card config via sysfs

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

```bash
$ echo 1 > /sys/class/sound/mixer/device/enable
```
Problem 2(b): Too many classes

Linux Device Drivers

★ “Understanding Modern Device Drivers” ASPLOS 2012
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

★ 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

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

★ 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
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
Checkpointing drivers is hard

★ Easy to capture memory state
Checkpointing drivers is hard

★ Easy to capture memory state

★ Device state is not captured

★ Device configuration space
Checkpointing drivers is hard

★ Easy to capture memory state

★ Device state is not captured
  ★ Device configuration space
  ★ Internal device registers and counters
Checkpointing drivers is hard

- Easy to capture memory state

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

- **Easy to capture memory state**

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

- **Unique for every device**
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**
- Save config state
- Save register state
- Disable device
- Save DMA state
- Suspend device

**Resume**
- Restore config state
- Restore register state
- Restore or reset DMA state
- Re-attach/Enable device
- Device Ready
### Device checkpoint/restore from PM code

<table>
<thead>
<tr>
<th>Suspend</th>
<th>Resume</th>
</tr>
</thead>
<tbody>
<tr>
<td>Save config state</td>
<td>Restore config state</td>
</tr>
<tr>
<td>Save register state</td>
<td>Restore register state</td>
</tr>
<tr>
<td>Save DMA state</td>
<td>Restore or reset DMA state</td>
</tr>
<tr>
<td>Suspend device</td>
<td>Re-attach/Enable device</td>
</tr>
<tr>
<td></td>
<td>Device Ready</td>
</tr>
</tbody>
</table>
Device checkpoint/restore from PM code

**Suspend**

- Save config state
- Save register state
- Save DMA state

**Resume**

- Restore config state
- Restore register state
- Restore or reset DMA state
- Re-attach/Enable device
- Device Ready
Device checkpoint/restore from PM code

**Suspend**
- Save config state
- Save register state
- Save DMA state

**Resume**
- Restore config state
- Restore register state
- Restore or reset DMA state
- Re-attach/Enable device
- Device Ready
Device checkpoint/restore from PM code

**Checkpoint**

- Save config state
- Save register state
- Save DMA state

**Resume**

- Restore config state
- Restore register state
- Restore or reset DMA state
- Re-attach/Enable device
- Device Ready
Device checkpoint/restore from PM code

**Checkpoint**

- Save config state
- Save register state
- Save DMA state

**Resume**

- Restore config state
- Restore register state
- Restore or reset DMA state
- Re-attach/Enable device
Device checkpoint/restore from PM code

**Checkpoint**

- Save config state
- Save register state
- Save DMA state

**Resume**

- Restore config state
- Restore register state
- Restore or reset DMA state
Device checkpoint/restore from PM code

**Checkpoint**
- Save config state
- Save register state
- Save DMA state

**Restore**
- Restore config state
- Restore register state
- Restore or reset DMA state
Device checkpoint/restore from PM code

**Checkpoint**
- Save config state
- Save register state
- Save DMA state

**Restore**
- Restore config state
- Restore register state
- Restore or reset DMA state

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
Goal: Improve driver recovery with minor changes to drivers

Solution: Run drivers as transactions using device checkpoints

Device state

Developers export checkpoint/restore in drivers
Goal: Improve driver recovery with minor changes to drivers

Solution: Run drivers as transactions using device checkpoints

- Developers export checkpoint/restore in drivers
- Run drivers invocations as memory transactions
- Use source transformation to copy parameters and run on separate stack

Device state

Driver state
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 drivers

### 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

Driver with checkpoint support

Static modifications
Adding transactional support to drivers

Driver with checkpoint support

Source transformation (adds driver transactions)

User supplied annotations

Static modifications
Adding transactional support to drivers

Driver with checkpoint support

Source transformation (adds driver transactions)

User supplied annotations

Main driver module

SFI driver module

SFI = software fault isolated

Static modifications
Adding transactional support to drivers

Source transformation (adds driver transactions)

Driver with checkpoint support

User supplied annotations

Main driver module

SFI driver module

SFI = software fault isolated

Static modifications

Run-time support
Adding transactional support to drivers

Driver with checkpoint support

User supplied annotations

Source transformation (adds driver transactions)

Static modifications

Main driver module

SFI driver module

Run-time support

Object tracking

Marshaling/Demarshaling

Kernel undo log

Communication and recovery support

1200 LOC

SFI = software fault isolated

Main driver module

User supplied annotations

Driver with checkpoint support
Transactional execution of drivers

network driver

SFI network driver

get ringparam
probe
xmit
get config
Transactional execution of drivers

- **get ringparam**
- **probe**
- **xmit**
- **get config**

**SFI network driver**

**netdev**

**network driver**

54
Transactional execution of drivers

network driver

get config

probe

xmit

netdev

get ringparam

SFI network driver
Transactional execution of drivers

netdev->priv->rx_ring
netdev->priv->tx_ring

C

get ringparam
probe
xmit
get config

netdev

network driver

SFI network driver
Transactional execution of drivers

<table>
<thead>
<tr>
<th>Address</th>
<th>Access rights</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xffffa000</td>
<td>Read</td>
</tr>
<tr>
<td>0xffffa008</td>
<td>Write</td>
</tr>
<tr>
<td>0xffffa00a</td>
<td>Read</td>
</tr>
</tbody>
</table>

Range Table

get ringparam
probe
xmit
get config

netdev
network
driver

netdev->priv->rx_ring
netdev->priv->tx_ring

SFI network driver
Transactional execution of drivers

- netdev->priv->rx_ring
- netdev->priv->tx_ring

Range Table

<table>
<thead>
<tr>
<th>Address</th>
<th>Access rights</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xffffa000</td>
<td>Read</td>
</tr>
<tr>
<td>0xffffa008</td>
<td>Write</td>
</tr>
<tr>
<td>0xffffa00a</td>
<td>Read</td>
</tr>
</tbody>
</table>

get ringparam
probe
xmit
get config

get config

SFI network driver

network driver

netdev

C

network driver
Transactional execution of drivers

![Diagram of network driver with SFI and Range Table]

- **Address**
  - 0xffffa000: Read
  - 0xffffa008: Write
  - 0xffffa00a: Read

**netdev->priv->rx_ring**
**netdev->priv->tx_ring**
Transactional execution of drivers

Range Table

<table>
<thead>
<tr>
<th>Address</th>
<th>Access rights</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xffffa000</td>
<td>Read</td>
</tr>
<tr>
<td>0xffffa008</td>
<td>Write</td>
</tr>
<tr>
<td>0xffffa00a</td>
<td>Read</td>
</tr>
</tbody>
</table>

get ringparam
probe
xmit
get config

netdev

network driver

netdev->priv->rx_ring
netdev->priv->tx_ring

SFI network driver

result

netdev->priv->rx_ring
netdev->priv->tx_ring

Kernel Log
alloc
Transactional execution of drivers

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

Range Table

<table>
<thead>
<tr>
<th>Address</th>
<th>Access rights</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xffffa000</td>
<td>Read</td>
</tr>
<tr>
<td>0xffffa008</td>
<td>Write</td>
</tr>
<tr>
<td>0xffffa00a</td>
<td>Read</td>
</tr>
</tbody>
</table>

Kernel Log alloc

**C**

- netdev
  - probe
  - xmit
  - get config

- netdev->priv->rx_ring
- netdev->priv->tx_ring
- result

SFI network driver

- get ringparam

Network driver

- netdev
FGFT: Failed transactions

network driver

SFI network driver

get ringparam
probe
xmit
get config
FGFT: Failed transactions
FGFT: Failed transactions

get ringparam
probe
xmit
get config

netdev

network driver

SFI
network driver
FGFT: Failed transactions

netdev->priv->rx_ring
netdev->priv->tx_ring

network driver

SFI network driver

C

get ringparam
probe
xmit
get config
FGFT: Failed transactions

Range Table

<table>
<thead>
<tr>
<th>Address</th>
<th>Access rights</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xffffa000</td>
<td>Read</td>
</tr>
<tr>
<td>0xffffa008</td>
<td>Write</td>
</tr>
<tr>
<td>0xffffa00a</td>
<td>Read</td>
</tr>
</tbody>
</table>

netdev->priv->rx_ring
netdev->priv->tx_ring

get ringparam
probe
xmit
get config

get config

netdev
network driver

SFI network driver
FGFT: Failed transactions

netdev->priv->rx_ring
netdev->priv->tx_ring

Range Table

<table>
<thead>
<tr>
<th>Address</th>
<th>Access rights</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xffffa000</td>
<td>Read</td>
</tr>
<tr>
<td>0xffffa008</td>
<td>Write</td>
</tr>
<tr>
<td>0xffffa00a</td>
<td>Read</td>
</tr>
</tbody>
</table>

get ringparam
probe
xmit
get config

netdev

network driver

C

SFI network driver
FGFT: Failed transactions

<table>
<thead>
<tr>
<th>Address</th>
<th>Access rights</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xffffa000</td>
<td>Read</td>
</tr>
<tr>
<td>0xffffa008</td>
<td>Write</td>
</tr>
<tr>
<td>0xffffa00a</td>
<td>Read</td>
</tr>
</tbody>
</table>
FGFT: Failed transactions

```
netdev->priv->rx_ring
netdev->priv->tx_ring
```

Range Table

<table>
<thead>
<tr>
<th>Address</th>
<th>Access rights</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xffffa000</td>
<td>Read</td>
</tr>
<tr>
<td>0xffffa008</td>
<td>Write</td>
</tr>
<tr>
<td>0xffffa00a</td>
<td>Read</td>
</tr>
</tbody>
</table>

Kernel Log alloc

netdev

network driver

probe

xmit

get config

get ringparam

C

netdev

network driver

SFI network driver

Kernel Log alloc
FGFT: Failed transactions

Range Table

<table>
<thead>
<tr>
<th>Address</th>
<th>Access rights</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xffffa000</td>
<td>Read</td>
</tr>
<tr>
<td>0xffffa008</td>
<td>Write</td>
</tr>
<tr>
<td>0xffffa00a</td>
<td>Read</td>
</tr>
</tbody>
</table>

get ringparam

netdev->priv->rx_ring
netdev->priv->tx_ring

SFI network driver

Kernel Log alloc

netdev

network driver

C

get config

probe

xmit
FGFT: Failed transactions

netdev->priv->rx_ring
netdev->priv->tx_ring

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

Range Table

Kernel Log alloc
FGFT: Failed transactions

<table>
<thead>
<tr>
<th>Address</th>
<th>Access rights</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xffffa000</td>
<td>Read</td>
</tr>
<tr>
<td>0xffffa008</td>
<td>Write</td>
</tr>
<tr>
<td>0xffffa00a</td>
<td>Read</td>
</tr>
</tbody>
</table>

**Range Table**

get ringparam

probe

xmit

get config

netdev

network driver

netdev->priv->rx_ring

netdev->priv->tx_ring

SFI network driver
FGFT: Failed transactions

```
netdev->priv->rx_ring
netdev->priv->tx_ring
```

Range Table

<table>
<thead>
<tr>
<th>Address</th>
<th>Access rights</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xfffffa000</td>
<td>Read</td>
</tr>
<tr>
<td>0xfffffa008</td>
<td>Write</td>
</tr>
<tr>
<td>0xfffffa0oa</td>
<td>Read</td>
</tr>
</tbody>
</table>

```
get ringparam
probe
xmit
get config
```
FGFT: Failed transactions

netdev->priv->rx_ring
netdev->priv->tx_ring

Range Table

<table>
<thead>
<tr>
<th>Address</th>
<th>Access rights</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xffffa000</td>
<td>Read</td>
</tr>
<tr>
<td>0xffffa008</td>
<td>Write</td>
</tr>
<tr>
<td>0xffffa00a</td>
<td>Read</td>
</tr>
</tbody>
</table>

netdev

C

network driver

R

get ringparam

probe

xmit

get config

err

SFI network driver

netdev

C

network driver

R

get ringparam

probe
	xmit

get config

err

SFI network driver
FGFT: Failed transactions

FGFT provides transactional execution of driver entry points

netdev->priv->rx_ring
netdev->priv->tx_ring

Range Table

<table>
<thead>
<tr>
<th>Address</th>
<th>Access rights</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xffffa000</td>
<td>Read</td>
</tr>
<tr>
<td>0xffffa008</td>
<td>Write</td>
</tr>
<tr>
<td>0xffffa00a</td>
<td>Read</td>
</tr>
</tbody>
</table>
How does this give us transactional execution?
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**
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**
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

<table>
<thead>
<tr>
<th>Recovery times</th>
<th>Restart recovery</th>
<th>FGFT recovery</th>
</tr>
</thead>
<tbody>
<tr>
<td>2,000ms</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1,500ms</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1,000ms</td>
<td></td>
<td></td>
</tr>
<tr>
<td>500ms</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0ms</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

8139too  e1000  pegasus  r8169  ens1371  psmouse
Recovery speedup

<table>
<thead>
<tr>
<th>Recovery times</th>
<th>0ms</th>
<th>500ms</th>
<th>1,000ms</th>
<th>1,500ms</th>
<th>2,000ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>8139-too</td>
<td>310.00</td>
<td>150.00</td>
<td>120.00</td>
<td>1030.00</td>
<td>1800.00</td>
</tr>
<tr>
<td>e1000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pegasus</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r8169</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ens1371</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>psmouse</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>680.00</td>
</tr>
</tbody>
</table>

- **Restart recovery**
- **FGFT recovery**
Recovery speedup

Restart recovery

FGFT recovery

0ms

8139too  e1000  pegasus  r8169  ens1371  psmouse

2,000ms

1,500ms

1,000ms

500ms

0ms
FGFT provides significant speedup in driver recovery and improves system availability.
Programming effort

<table>
<thead>
<tr>
<th>Driver</th>
<th>LOC</th>
<th>Checkpoint/restore effort</th>
<th>LOC Moved</th>
<th>LOC Added</th>
</tr>
</thead>
<tbody>
<tr>
<td>8139too</td>
<td>1,904</td>
<td></td>
<td>26</td>
<td>4</td>
</tr>
<tr>
<td>e1000</td>
<td>13,973</td>
<td></td>
<td>32</td>
<td>10</td>
</tr>
<tr>
<td>r8169</td>
<td>2,993</td>
<td></td>
<td>17</td>
<td>5</td>
</tr>
<tr>
<td>pegasus</td>
<td>1,541</td>
<td></td>
<td>22</td>
<td>5</td>
</tr>
<tr>
<td>ens1371</td>
<td>2,110</td>
<td></td>
<td>16</td>
<td>6</td>
</tr>
<tr>
<td>psmouse</td>
<td>2,448</td>
<td></td>
<td>19</td>
<td>6</td>
</tr>
</tbody>
</table>

FGFT requires limited programmer effort and needs only 38 lines of new kernel code.
Throughput with isolation and recovery

Netperf on Intel quad-core machines
Throughput with isolation and recovery

Throughput %age (Baseline 844 Mbps)

- Native
- FGFT-I/O-all
- FGFT-off-I/O
- FGFT-I/O-1/2

netperf on Intel quad-core machines
Throughput with isolation and recovery

Throughput %age (Baseline 844 Mbps)

CPU: 2.4%

Native
FGFT-I/O-all
FGFT-off-I/O
FGFT-I/O-1/2

e1000 Network Card

netperf on Intel quad-core machines
Throughput with isolation and recovery

Throughput %age (Baseline 844 Mbps)

CPU: 2.4%  2.4%

100

93

Native
FGFT-I/O-all
FGFT-off-I/O
FGFT-I/O-1/2

netperf on Intel quad-core machines

e1000 Network Card
Throughput with isolation and recovery

Throughput %age (Baseline 844 Mbps)

CPU: 2.4% 3.4% 2.4%

100 93 100

Native
FGFT-I/O-all
FGFT-off-I/O
FGFT-I/O-1/2

e1000 Network Card

netperf on Intel quad-core machines
Throughput with isolation and recovery

Throughput %age (Baseline 844 Mbps)

**CPU:** 2.4% 3.4% 2.4% 2.9%

Native: 100
FGFT-I/O-all: 93
FGFT-off-I/O: 100
FGFT-I/O-1/2: 96

**e1000 Network Card**

*netperf on Intel quad-core machines*
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
First research consideration of hardware failures in drivers

- Released tool, patches & informed developers

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

- Measured driver behavior & identified new directions

Introduced checkpoint/restore in drivers for low latency fault tolerance

- Fast & correct recovery with incremental changes to drivers
Questions

Asim Kadav
☆ http://cs.wisc.edu/~kadav
☆ kadav@cs.wisc.edu