Understanding and Improving Device Access Complexity

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

- Keyboard
- Sound
- Printer
- Network
- Storage

- Keyboard
- Flash storage
- Graphics
- WIFI
- Headphones
- SD card
- Camera
- Accelerometers
- GPS
- Touch display
- NFC
Huge growth in number of devices

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

Many buses: USB, PCI-e, thunderbolt

Heterogeneous O/S support: 10G ethernet vs card readers
Device drivers: OS interface to devices

- Applications
- OS
- 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
Simplicity

Reliability

Complexity hurts efficient device access

Tools and mechanisms to address increasing device complexity

Growth in number and diversity

Run in challenging environments

Efficient device support in OS

Low latency

Cost effective

Hardware failures (like CMOS issues)

Complex firmware and configuration modes
Complexity hurts understanding of drivers

Lines of code in Linux 3.8

- Applications: 60,000
- OS: 66,000
- Memory mgmt: 760,000
- Device drivers: 1,750,000
- File systems: 3,500,000
- Drivers: 5,250,000
- Buses: 6,700,000
- Devices: 7,000,000

Contribute 60% of Linux kernel code and more than 90% in Windows

Understand and improve this rapidly growing body of code
Last decade: Reliability of the driver-kernel interface

Recipe for disaster

3rd party developers + device drivers
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>Specification</td>
<td>Nexus [OSDI 08]</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Termite [SOSP 09]</td>
<td>2</td>
</tr>
</tbody>
</table>

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

**Singularity [Eurosys 06]**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Drivers</th>
<th>Bus</th>
<th>Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

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

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

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
My approach

- Take a narrow view and solve specific problems in all drivers
- Tolerate device failures

- Take a broad approach and have a holistic view of all drivers
- Understand drivers and potential opportunities

- Take a known approach and apply to all drivers
- Transactional approach for low latency recovery

Minimize kernel changes and apply to all drivers
Contributions/Outline

- Tolerate device failures
  - First research consideration of hardware failures in drivers
    - SOSP ’09
  - Introduce checkpoint/restore in drivers for low latency fault tolerance
    - ASPLOS ’13
- Understand drivers and potential opportunities
  - Largest study of drivers to understand their behavior and verify research assumptions
    - ASPLOS ’12

SOSP ’09
ASPLOS ’12
ASPLOS ’13
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

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

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

Results of misbehavior

- Corrupted/stuck-at inputs
- Timing errors/incorrect memory access
- Interrupt storms/missing interrupts
An evidence:

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?

1. Drivers use device data in critical control and data paths
   - `printf("%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

<table>
<thead>
<tr>
<th>Recommendation</th>
<th>Summary</th>
<th>Recommended by</th>
</tr>
</thead>
<tbody>
<tr>
<td>Validation</td>
<td>Input validation</td>
<td>Intel</td>
</tr>
<tr>
<td></td>
<td>Read once &amp; CRC data</td>
<td>Sun</td>
</tr>
<tr>
<td></td>
<td>DMA protection</td>
<td>MS</td>
</tr>
<tr>
<td>Timing</td>
<td>Infinite polling</td>
<td>Linux</td>
</tr>
<tr>
<td>Reporting</td>
<td>Report all failures</td>
<td>Intel</td>
</tr>
<tr>
<td>Recovery</td>
<td>Handle all failures</td>
<td>Sun</td>
</tr>
<tr>
<td></td>
<td>Cleanup correctly</td>
<td>MS</td>
</tr>
<tr>
<td></td>
<td>Do not crash on failure</td>
<td>Linux</td>
</tr>
<tr>
<td></td>
<td>Wrap I/O memory access</td>
<td>Intel</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

**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

Recovery and interrupt watchdog

OS Kernel

Carburizer

Compiler

Driver

Faulty Hardware

Hardened Driver Binary

Kernel Interface

Carburizer Runtime

Driver

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

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

• Carburizer detects and fixes hardware bugs:
  - Infinite polling
  - Unsafe pointer reference
  - Unsafe array reference
  - System panic calls
Finding sensitive code

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

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

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

Tainted Variables

```
int set() {
    int e = test();
}
```
Detecting risky uses of tainted variables

- **Finding sensitive code**
  - Second pass: Identify **risky uses** of tainted variables

- **Example: Infinite polling**
  - Driver waiting for device to enter particular state
  - Solution: Detect loops where **all** terminating conditions depend on tainted variables
Infinite polling

★ Infinite polling of devices can cause system lockups

static int amd8111e_read_phy(........)
{
    ...
    reg_val = readl(mmio + PHY_ACCESS);
    while (reg_val & PHY_CMD_ACTIVE)
    {
        reg_val = readl(mmio + PHY_ACCESS)
    }
    ...
}
Hardware data used in array reference

* Tainted variables used as array indexes

```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));
        ...
    }
```
Experience with the Linux kernel

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

<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></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>

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

Lightweight and usable technique to find hardware dependence bugs
Repairing drivers

Call recovery service

- Timeout checks
- Array bounds check
- Not null checks
- Infinite polling
- Unsafe array reference
- Unsafe pointer reference
- System panic calls
• **Carburizer** calls generic recovery service if check fails

• **Low cost transparent recovery**
  ★ Based on shadow drivers
  ★ Records state of driver
  ★ Transparent restart and state replay on failure

• **No isolation required** (like Nooks)

---

**Runtime fault recovery**

[Swift [OSDI ’04]]
Carburizer automatically fixes infinite loops

```c
timeout = rdtsc11(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)
        rdtsc11(_cur);
    else
        __recover_driver();
}
```

*Code simplified for presentation purposes*
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));
    }
}
Fault injection validation

- Synthetic fault injection on network drivers
- Results

<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>

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

Almost no overhead from hardened drivers and automatic recovery
Tolerate device failures

Hardening drivers
Reporting failures
Runtime Fault tolerance
Results

Understand drivers and potential opportunities

Transactional approach for cheap recovery
Outline

Tolerate device failures

Hardening drivers
Reporting failures
Runtime Fault tolerance
Results

Understand drivers and potential opportunities

Transactional approach for cheap recovery
Runtime failure detection

- Static analysis cannot detect all device failures

- **Missing interrupts**: Interrupt expected but never arrives

- **Stuck interrupts**: Interrupt cleared but continues to assert
Missing interrupts

- **Device polling on interrupt failures**
  - Polling frequently has high overhead
  - Polling infrequently results in throughput loss

- **How frequently should we poll?**
  - Increase frequency if interrupt invocation did useful work

- **When are requests likely to come?**
  - Driver invocation: Use reference bits to detect driver activity
Stuck interrupts

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

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

Carburizer ensures system makes forward progress
Carburizer improves system reliability by automatically ensuring that hardware failures are tolerated in software.
Contributions beyond research

- Informed developers at Plumbers Conference [2011]
- LWN Article with paper & list of bugs [Feb ‘12]
- Released patches to the Linux kernel
Functionality: Recovery assumes drivers follow class behavior

- **Non-class behavior can lead to incomplete restore after failure**

- **Record state by interposing class defined entry points**
- **Restart and replay state using class semantics when failure happens**
Recovery Performance: Device initialization is slow

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

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

- Tolerate device failures
- Understand drivers and potential opportunities
- Transactional approach for cheap recovery
- Overview
  Recovery specific results
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

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)

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

Driver properties

For every merged driver

Driver entry points

open

close

probe

xmit
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

Driver properties

Driver interactions

xmit
open
close
probe
kmalloc
Study methodology

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

★ Identify driver wide and function specific properties of all drivers

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

★ Use statistical clustering techniques and static analysis to identify similar code
Some additional results

- Many assumptions made by driver research does not hold:
  - 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
Contributions/Outline

- Tolerate device failures
- Understand drivers and potential opportunities
- Transactional approach for cheap recovery
- Overview
  Recovery specific results
 Initialization code dominates driver LOC and adds to complexity
Recovery assumes drivers follow class behavior

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

Does driver behavior belong to class definitions?
Do drivers belong to classes?

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

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

★ Overall, 44% of drivers do not conform to class behavior
Outline

- Tolerate device failures
- Understand drivers and potential opportunities
- Transactional approach for cheap recovery
  - Checkpoint/restore
  - FGFT
  - Future work and conclude
Limitations of restart/replay recovery

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

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

- Checkpoints limited to capturing **memory** state

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

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

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

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

**Suspend**
- Save config state
- Save device state
- Disable device
- Copy-out s/w state

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

*Suspend/resume code provides checkpoint functionality*
**Fine-Grained Fault Tolerance**

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

- **Developers export checkpoint/restore in all drivers**
Fine-Grained Isolation

Resource access
- I/O memory: Full access
- Locks: Read access & locks acquired via kernel
- Memory: Allocate & add to range table
- Suspect entry point arrives
- Checkpoint device
- Marshal required data in SFI
- Populate range table
- Execute & Populate compensation log
- Success: Copy back written data

Range Table

<table>
<thead>
<tr>
<th>Address</th>
<th>Access rights</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xffffffff</td>
<td>Read</td>
</tr>
<tr>
<td>0xfffffa08</td>
<td>Write</td>
</tr>
<tr>
<td>0xfffffa0a</td>
<td>Read</td>
</tr>
</tbody>
</table>

Kernel Log

lock
Fine-Grained Isolation

**SFI network driver**

- **netdev->priv->tx_ring**
- **netdev->priv->rx_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 lock**

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

**FGFT provides transactional execution of driver entry points**

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

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

**FGFT provides speedup in driver recovery**
## Programming effort

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

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

Throughput %age (Baseline 844 Mbps)

Network Card Type

netperf on Intel quad-core machines

CPU: 2.4%  2.4%  2.9%  3.4%

Native  FGFT-static  FGFT-dynamic-1/2  FGFT-dynamic-all
Summary

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

- Tolerate device failures
- Understand drivers and potential opportunities
- Transactional approach for cheap recovery
- Checkpoint/restore
- FGFT
- Other/Future Work
Other work

Storage

- Differential RAID [Eurosys '10]
- SymDrive [OSDI '12]
- FGFT [ASPLOS '13]
- Carburizer [SOSP '09]

Drivers

- GPFS
- ThinCloud [Under Submission]
- Live Migration [OSR '09]
- Driver study [ASPLOS '12]

Reliability

Performance

Measurement

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

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

- Solve new problems
- Build real systems
- Measure real impact
- Improve
Future Work: Lessons from reliability research

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

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

- Automatically fix problems in other plugin based architectures like app stores, browsers
Future Work: Investigate OS-hardware co-design

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

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

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

Asim Kadav
★ http://cs.wisc.edu/~kadav

Thanks to
★ Michael Swift
★ Matt Renzelmann
★ Mahesh Balakrishnan
★ Dahlia Malkhi
★ Vijayan Prabhakaran
★ Ed Nightingale
★ Jeremy Elson
★ James Mickens
★ Rathijit Sen