


## Understanding Modern Device Drivers



Asim Kadav and Michael M. Swift  
 University of Wisconsin-Madison 

## Why study device drivers?

- » Linux drivers constitute ~5 million LOC and 70% of kernel
  - » Little exposure to this breadth of driver code from research
  - » Better understanding of drivers can lead to better driver model
- » Large code base discourages major changes
  - » Hard to generalize about driver properties
  - » Slow architectural innovation in driver subsystems
- » Existing architecture: Error prone drivers
  - » Many developers, privileged execution, C language
  - » Recipe for complex system with reliability problems

2

## Our view of drivers is narrow

- » Driver research is focused on reliability
  - » Focus limited to fault/bug detection and tolerance
  - » Little attention to architecture/structure
- » Driver research only explores a small set of drivers
  - » Systems evaluate with mature drivers
  - » Volume of driver code limits breadth
- » Necessary to review current drivers in modern settings

3

## Difficult to validate research on all drivers

Improvement	System	Validation		
		Drivers	Bus	Classes
New functionality	Shadow driver migration [OSRes]	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
	Singularity [Eurosys 06]	1	1	1
Specification	Nexus [OSDI 08]	2	1	2
	Termite [SOSP 09]	2	1	2
Static analysis tools	SDV [Eurosys 06]	All	All	All

Device availability/slow driver development restrict our research runtime solutions to a small set of drivers

4

## Difficult to validate research on all drivers

“...Please do not misuse these tools! (Coverity)... If you focus too much on fixing the problems quickly rather than fixing them cleanly, then we forever lose the opportunity to clean our code, because the problems will then be hidden.”


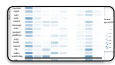
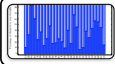

LKML mailing list <http://lkml.org/lkml/2005/3/27/131>

Static analysis tools		All	All	All
SDV [Eurosys 06]		All	All	All
Carburizer [SOSP 09]		All/1	All	All
Cocinelle [Eurosys 08]		All	All	All

## Understanding Modern Device Drivers

- » Study source of all Linux drivers for x86 (~3200 drivers)
- » Understand properties of driver code
  - » What are common code characteristics?
  - » Do driver research assumptions generalize?
- » Understand driver interactions with outside world
  - » Can drivers be easily re-architected or migrated ?
  - » Can we develop more efficient fault-isolation mechanisms?
- » Understand driver code similarity
  - » Do we really need all 5 million lines of code?
  - » Can we build better abstractions?


## Outline

	Methodology
	Driver code characteristics
	Driver interactions
	Driver redundancy

## Methodology of our study

- » Target Linux 2.6.37.6 (May 2011) kernel
- » Use static source analyses to gather information
- » Perform multiple dataflow/control-flow analyses
  - » Detect driver properties of the drive code
  - » Detect driver code interactions with environment
  - » Detect driver code similarities within classes

### Extract driver wide properties for individual drivers



Identify driver entry points, driver data structures, kernel registration

- Bus type registered with the kernel
- Device class registered with the kernel
- Device sub class registered with the kernel
- Driver functions mapped to entry points
- Number of devices registered with the bus
- Other properties (module parameters etc)
- Length of the driver functions

Step 1: Determine driver code characteristics for each driver from driver data structures registered with the kernel

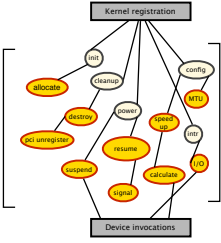
9

### Determine code characteristics of each driver function

- Bus type registered with the kernel
- Device class registered with the kernel
- Device sub class registered with the kernel
- Driver functions mapped to entry points
- Number of devices registered with the bus
- Other properties (module parameters etc)
- Length of the driver functions

Record driver properties using multiple control/data flow analysis

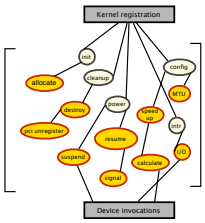
Tag each reachable driver function with entry point information



Step 2: Propagate the required information to driver functions and collect information about each function

10

### Determining interactions of each driver function



Undefined kernel functions classified by type (device/kernel libraries, memory management, synchronization, services)



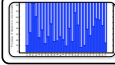

Device calls - port/memio, DMA, I2C operations, bus library invocations

- Kernel interaction across entry points
- Device interaction across entry points
- Difference in driver structures across buses
- Bus type registered with the kernel
- Threading/Synchronization models used

Step 3: Determine driver interactions from I/O operations and calls to kernel and bus for each function and propagate to entry points

11

### Outline

-  Methodology
-  Driver code characteristics
-  Driver interactions
-  Driver redundancy

12

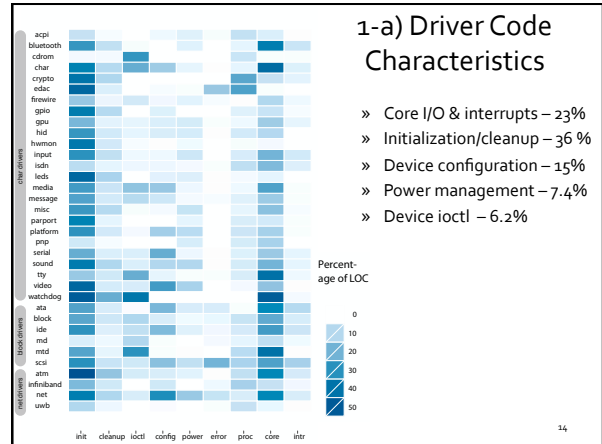
## Part 1: Driver Code Behavior

A device driver can be thought of as a translator. Its input consists of high level commands such as "retrieve block 123". Its output consists of low level, hardware specific instructions that are used by the hardware controller, which interfaces the I/O device to the rest of the system.

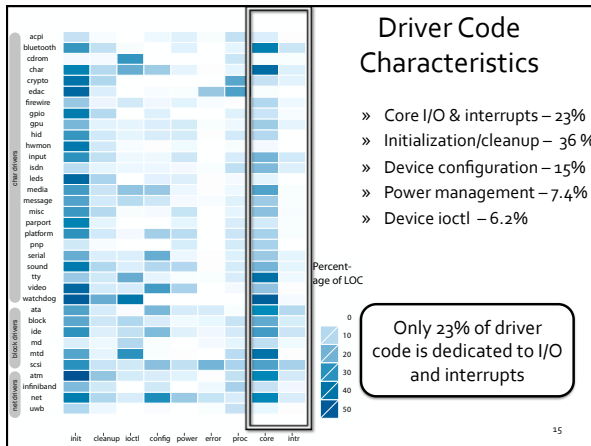
-- Operating System Concepts VIII edition

Driver code complexity and size is assumed to be a result of its I/O function.

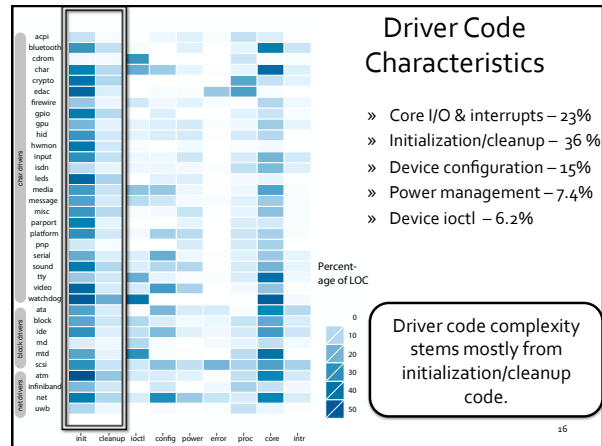
13



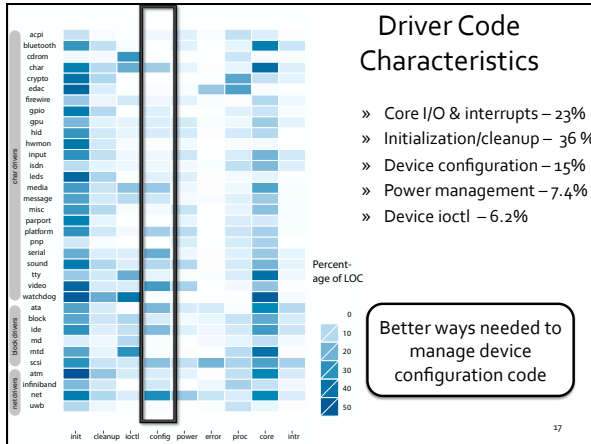
14



15



16



### 1-b) Do drivers belong to classes?

- » Drivers registers a class interface with kernel
  - » Example: Ethernet drivers register with bus and net device library
- » Class definition includes:
  - » Callbacks registered with the bus, device and kernel subsystem
  - » Exported APIs of the kernel to use kernel resources and services
- » Most research assumes drivers obey class behavior

### Class definition used to record state

- » Modern research assumes drivers conform to class behavior
- » Example: Driver recovery (Shadow drivers<sup>[OSDI 04]</sup>)

- » Driver state is recorded based on interfaces defined by class
- » State is replayed upon restart after failure to restore state

Figure 3: A sample shadow driver operating in active mode. The taps redirect communication between the kernel and the failed driver directly to the shadow driver.

Non-class behavior can lead to incomplete restore after failure

### Class definition used to infer driver behavior

- » Example: Reverse engineering of drivers - RevNIC<sup>[Eurosys 10]</sup>

- » Driver behavior is reverse engineered based on interfaces defined by class
- » Driver entry points are invoked to record driver operations
- » Code is synthesized for another OS based on this behavior

Figure 1: High-level architecture of RevNIC.

Figure from Revnic paper

Non-class behavior can lead to incomplete reverse engineering of device driver behavior

## Do drivers belong to classes?

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

```
... qlcnic_sysfs_write_esw_config (...) {
...
switch (esw_cfg[i].op_mode) {
case QLCNIC_PORT_DEFAULTS:
    qlcnic_set_eswitch_...(...,&esw_cfg[i]);
    ...
case QLCNIC_ADD_VLAN:
    qlcnic_set_vlan_config(...,&esw_cfg[i]);
    ...
case QLCNIC_DEL_VLAN:
    esw_cfg[i].vlan_id = 0;
    qlcnic_set_vlan_config(...,&esw_cfg[i]);
    ...
}
```

Drivers/net/qlcnic/qlcnic\_main.c: QLogic driver(network class)

## Many drivers do not conform to class definition

- » 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
- » Breaks systems that assume driver semantics can be completely determined from class behavior

Overall, 44% of drivers do not conform to class behavior  
Systems based on class definitions may not work properly  
when such non-class extensions are used

## 1-c) Do drivers perform significant processing?

- » Drivers are considered only a conduit of data
- » Example: Synthesis of drivers (Termite<sup>[SOSP09]</sup>)
  - » State machine model only allows passing of data
  - » Does not support transformations/processing
- » But: drivers perform checksums for RAID, networking, or calculate display geometry data in VMs

23

## Instances of processing loops in drivers

- » Detect loops in driver code that:
  - » do no I/O,
  - » do not interact with kernel
  - » lie on the core I/O path

```
static u8 e1000_calculate_checksum(...)
{
    u32 i;
    u8 sum = 0;
    ...
    for (i = 0; i < length; i++)
        sum += buffer[i];

    return (u8) (0 - sum);
}
```

drivers/net/e1000e/lib.c: e1000e network driver

24

## Many instances of processing across classes

```
static void _cx18_process_vbi_data(...)
{
    // Process header & check endianness
    // Obtain RAW and sliced VBI data
    // Compress data, remove spaces, insert mpg info.
}
void cx18_process_vbi_data(...)
{
    // Loop over incoming buffer
    // and call above function
}
drivers/media/video/cx18/cx18-vbi.c:cx18 IVTV driver
```

25



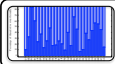

## Drivers do perform processing of data

- » Processing results from our analyses:
  - » 15% of all drivers perform processing
  - » 28% of sound and network drivers perform processing
- » Driver behavior models should include processing semantics
  - » Implications in automatic generation of driver code
  - » Implications in accounting for CPU time in virtualized environment

Driver behavior models should consider processing

26

## Outline

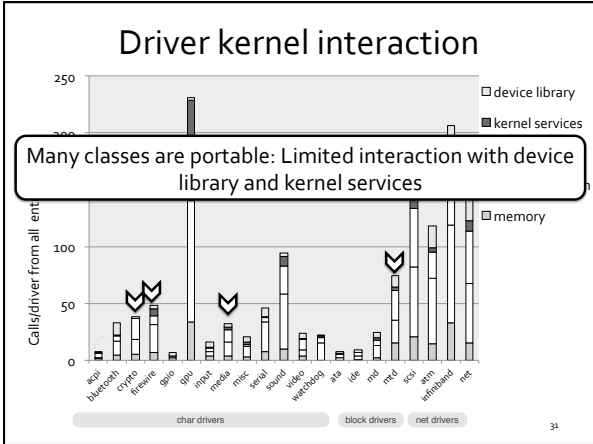
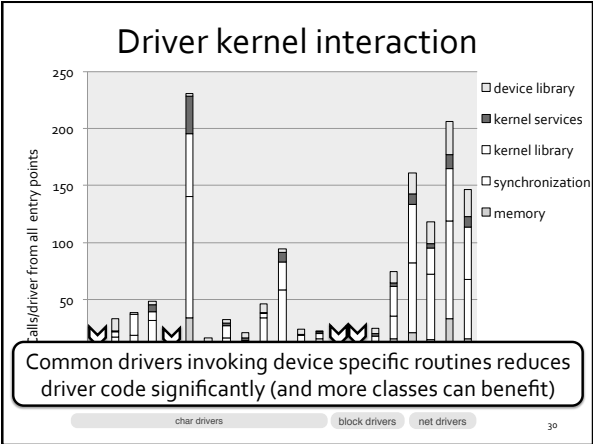
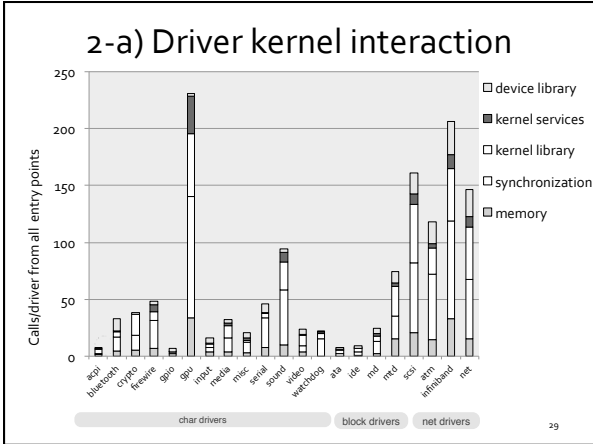
-  Methodology
-  Driver code characteristics
-  Driver interactions
-  Driver redundancy

27

## Part 2: Driver interactions

- a) What are the opportunities to redesign drivers?
  - » Can we learn from drivers that communicate efficiently?
  - » Can driver code be moved to user mode, a VM, or the device for improved performance/reliability?
- b) How portable are modern device drivers?
  - » What are the kernel services drivers most rely on?
- c) Can we develop more efficient fault-tolerance mechanisms?
  - » Study drivers interaction with kernel, bus, device, concurrency

28



- ### 2-b) Driver-bus interaction
- » Compare driver structure across buses
  - » Look for lessons in driver simplicity and performance
  - » Can they support new architectures to move drivers out of kernel?
    - » Efficiency of bus interfaces (higher devices/driver)
      - Interface standardization helps move code away from kernel
    - » Granularity of interaction with kernel/device when using a bus
      - Coarse grained interface helps move code away from kernel
- 32



### PCI drivers: Fine grained & few devices/driver

BUS	Kernel Interactions (network drivers)					Device Interactions (network drivers)				
	mem	sync	dev lib	kern lib	services	port/mmio	DMA	bus	Devices/driver	
PCI	29.3	91.1	46.7	103	12	302	22	40.4	9.6	

- » PCI drivers have fine grained access to kernel and device
  - » Support low number of devices per driver (same vendor)
  - » Support performance sensitive devices
  - » Provide little isolation due to heavy interaction with kernel
  - » Extend support for a device with a completely new driver

33

### USB: Coarse grained & higher devices/driver

BUS	Kernel Interactions (network drivers)					Device Interactions (network drivers)				
	mem	sync	dev lib	kern lib	services	port/mmio	DMA	bus	Devices/driver	
PCI	29.3	91.1	46.7	103	12	302	22	40.4	9.6	
USB	24.5	72.7	10.8	25.3	11.5	0.0	6.2*	36.0	15.5	

- » USB devices support far more devices/driver
  - » Bus offers significant functionality enabling standardization
  - » Simpler drivers (like, DMA via bus) with coarse grained access
  - » Extend device specific functionality for most drivers by only providing code for extra features

\* accessed via bus

34

### Xen : Extreme standardization, limit device features


BUS	Kernel Interactions (network drivers)					Device Interactions (network drivers)				
	mem	sync	dev lib	kern lib	services	port/mmio	DMA	bus	Devices/driver	
PCI	29.3	91.1	46.7	103	12	302	22	40.4	9.6	
USB	24.5	72.7	10.8	25.3	11.5	0.0	6.2*	36.0	15.5	
Xen	11.0	7.0	27.0	7.0	7.0	0.0	0.0	24.0	1/All	

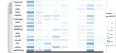
- » Xen represents extreme in device standardization
  - » Xen can support very high number of devices/driver
  - » Device functionality limited to a set of standard features
  - » Non-standard device features accessed from domain executing the driver

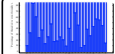
Efficient remote access to devices and efficient device driver support offered by USB and Xen

35

### Outline

 Methodology

 Driver code characteristics

 Driver interactions

 Driver redundancy

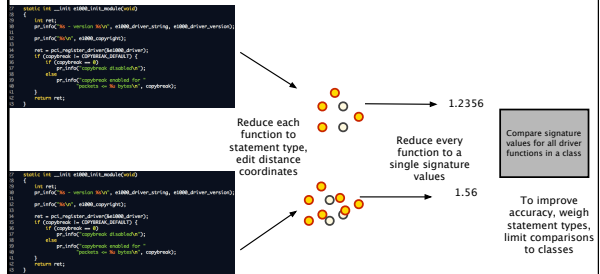
36

Part 3: Can we reduce the amount of driver code?

- » Are 5 million lines of code needed to support all devices?
  - » Are there opportunities for better abstractions?
  - » Better abstractions reduce incidence of bugs
  - » Better abstractions improve software composability
- » Goal: Identify the missing abstraction types in drivers
  - » Quantify the savings by using better abstractions
  - » Identify opportunities for improving abstractions/interfaces

37

Finding out similar code in drivers



Determine similar driver code by identifying clusters of code that invoke similar device, kernel interactions and driver operations

38

Drivers within subclasses often differ by reg values

```

.. nv_mcp55_thaw(...) {
void __iomem *mmio_base = ap->host->iomap[NV_MMIO_BAR];

int shift = ap->port_no *
NV_INT_PORT_SHIFT_MCP55;
...
writel(NV_INT_ALL_MCP55 << shift,
mmio_base+NV_INT_STATUS_MCP55);

mask = readl(mmio_base +
NV_INT_ENABLE_MCP55);
mask |= (NV_INT_MASK_MCP55 <<
shift);
writel(mask, mmio_base +
NV_INT_ENABLE_MCP55);
}

.. nv_ck804_thaw(...) {
void __iomem *mmio_base = ap->host->iomap[NV_MMIO_BAR];

int shift = ap->port_no *
NV_INT_PORT_SHIFT;
...
writel(NV_INT_ALL << shift,
mmio_base +
NV_INT_STATUS_CK804);
mask = readb(mmio_base +
NV_INT_ENABLE_CK804);
mask |= (NV_INT_MASK << shift);
writel(mask, mmio_base +
NV_INT_ENABLE_CK804);
}
    
```

drivers/ata/sata\_nv.c

39

Wrappers around device/bus functions

```

static int nv_pre_reset(...)
{
..
struct pci_bits
nv_enable_bits[] = {
{ 0x50, 1, 0x02, 0x02 },
{ 0x50, 1, 0x01, 0x01 }
};

struct ata_port *ap = link->ap;
struct pci_dev *pdev =
to_pci_dev(...);
if (!pci_test_config_bits
(pdev,&nv_enable_bits[ap->port_no]))
return -ENOENT;
return ata_sff_prereset(...);
}

static int amd_pre_reset(...)
{
..
struct pci_bits
amd_enable_bits[] = {
{ 0x40, 1, 0x02, 0x02 },
{ 0x40, 1, 0x01, 0x01 }
};

struct ata_port *ap = link->ap;
struct pci_dev *pdev =
to_pci_dev(...);
if (!pci_test_config_bits
(pdev,&amd_enable_bits[ap->port_no]))
return -ENOENT;
return ata_sff_prereset(...);
}
    
```

drivers/ata/pata\_amd.c

40



### Drivers repeat functionality around kernel wrappers

```

... delkin_cb_resume(...) {
    struct ide_host *host =
        pci_get_drvdata(dev);
    int rc;

    pci_set_power_state(dev, PCI_D0);
    rc = pci_enable_device(dev);
    if (rc)
        return rc;

    pci_restore_state(dev);
    pci_set_master(dev);

    if (host->init_chipset)
        host->init_chipset(dev);
    return 0;
}
    
```

drivers/ide/ide.c

```

... ide_pci_resume(...) {
    struct ide_host *host =
        pci_get_drvdata(dev);
    int rc;

    pci_set_power_state(dev, PCI_D0);
    rc = pci_enable_device(dev);
    if (rc)
        return rc;

    pci_restore_state(dev);
    pci_set_master(dev);

    if (host->init_chipset)
        host->init_chipset(dev);
}
    
```

drivers/delkin\_cb.c

45

### Drivers covered by our analysis

- All drivers that compile on x86 platform in Linux 2.6.37.6
- Consider driver, bus and virtual drivers
- Skip drivers/staging directory
  - Incomplete/buggy drivers may skew analysis
- Non x86 drivers may have similar kernel interactions
- Windows drivers may have similar device interactions
  - New driver model introduced (WDM), improvement over vxd

46

### Limitations of our analyses

- Hard to be sound/complete over ALL Linux drivers
- Examples of incomplete/unsound behavior
  - Driver maintains private structures to perform tasks and exposes opaque operations to the kernel

47

### Repeated code in family of devices (e.g initialization)

```

... asd_aic9405_setup(...) {
    int err = asd_common_setup(...);

    if (err)
        return err;

    asd_ha->hw_prof.addr_range = 4;
    asd_ha->hw_prof.port_name... = 0;
    asd_ha->hw_prof.dev_name... = 4;
    asd_ha->hw_prof.sata_name... = 8;

    return 0;
}
    
```

drivers/scsi/aic94xx driver

```

... asd_aic9410_setup(...) {
    int err = asd_common_setup(...);

    if (err)
        return err;

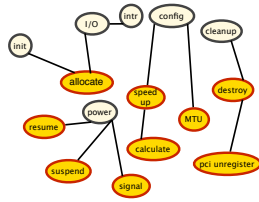
    asd_ha->hw_prof.addr_range = 8;
    asd_ha->hw_prof.port_name... = 0;
    asd_ha->hw_prof.dev_name... = 8;
    asd_ha->hw_prof.sata_name... = 16;

    return 0;
}
    
```

48

### How many devices does a driver support?

- Many research projects generate code for specific device/driver
- Example, safety specifications for a specific driver



49

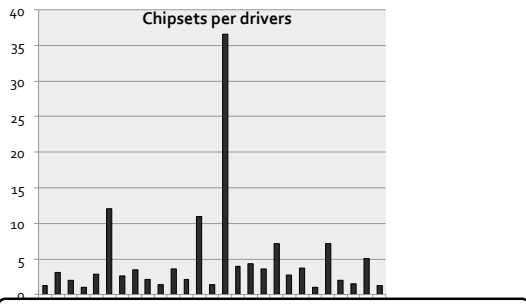
### How many devices does a driver support?

```
static int __devinit cy_pci_probe(...)
{
    if (device_id == PCI_DEVICE_ID_CYCLOM_Y_Lo) { ...
    if (pci_resource_flags(pdev,2)&IORESOURCE_IO){ ...
    if (device_id == PCI_DEVICE_ID_CYCLOM_Y_Lo ||
        device_id == PCI_DEVICE_ID_CYCLOM_Y_Hi) {...
    }else if (device_id==PCI_DEVICE_ID_CYCLOM_Z_Hi)
    ....
    if (device_id == PCI_DEVICE_ID_CYCLOM_Y_Lo ||
        device_id == PCI_DEVICE_ID_CYCLOM_Y_Hi) {
        switch (plx_ver) {
            case PLX_9050: ...
            default: /* Old boards, use PLX_9060 */
        }
    }
}
```

drivers/char/cyclades.c: Cyclades character driver

50

### How many devices does a driver support?



28% of drivers support more than one chipset

char drivers    block drivers    net drivers

51

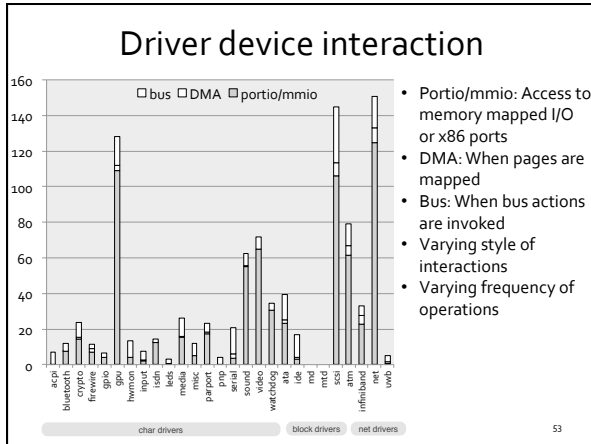
### How many devices does a driver support?

28% of drivers support more than one chipset

83% of the total devices are supported by these drivers

- Linux drivers support ~14000 devices with 3200 drivers
- Number of chipsets weakly correlated to the size of the driver (not just initialization code)
- Introduces complexity in driver code
- Any system that generates unique drivers/specs per chipset will lead in expansion in code

52



### Class definition used to record state

» Modern research assumes drivers conform to class behavior

**Figure 3:** A sample shadow driver operating in active mode. The taps redirect communication between the kernel and the failed driver directly to the shadow driver.

**Figure 1:** High-level architecture of RevNIC.

- » Driver behavior is reverse engineered based on interfaces defined by class
- » State is replayed upon restart
- » Code is synthesized for another OS based on this behavior<sup>54</sup>