Device drivers constitute 70% of the Linux code base. Several studies have shown that drivers are the dominant cause of OS crashes in desktop PCs.

This research focus on

- what driver code does
- the interaction of driver code with devices, buses, and the kernel
- new opportunities for abstracting driver functionality into common libraries or subsystems.

A device driver is a software component that provides an interface between the OS and a hardware device.

- The device configure and manage the device
- Convert requests from kernel into requests to the hardware

Drivers relies on three interfaces:

- Driver and kernel, for communicating requests and accessing OS services
- Driver and device, for executing operations
- Driver and bus, for managing communication with the device.

In Linux, a driver class refers to a category of devices that share a common interface and behavior.

In Linux, the three main categories(class) of drivers

- Character drivers, which are byte-stream oriented
- Block drivers, which support random-access to blocks
- Network drivers, support stream of packets.
- The class defines a standard set of operations that all drivers in that category must implement. For example, all network drivers must provide functions for sending packets, receiving packets, and handling network configuration.

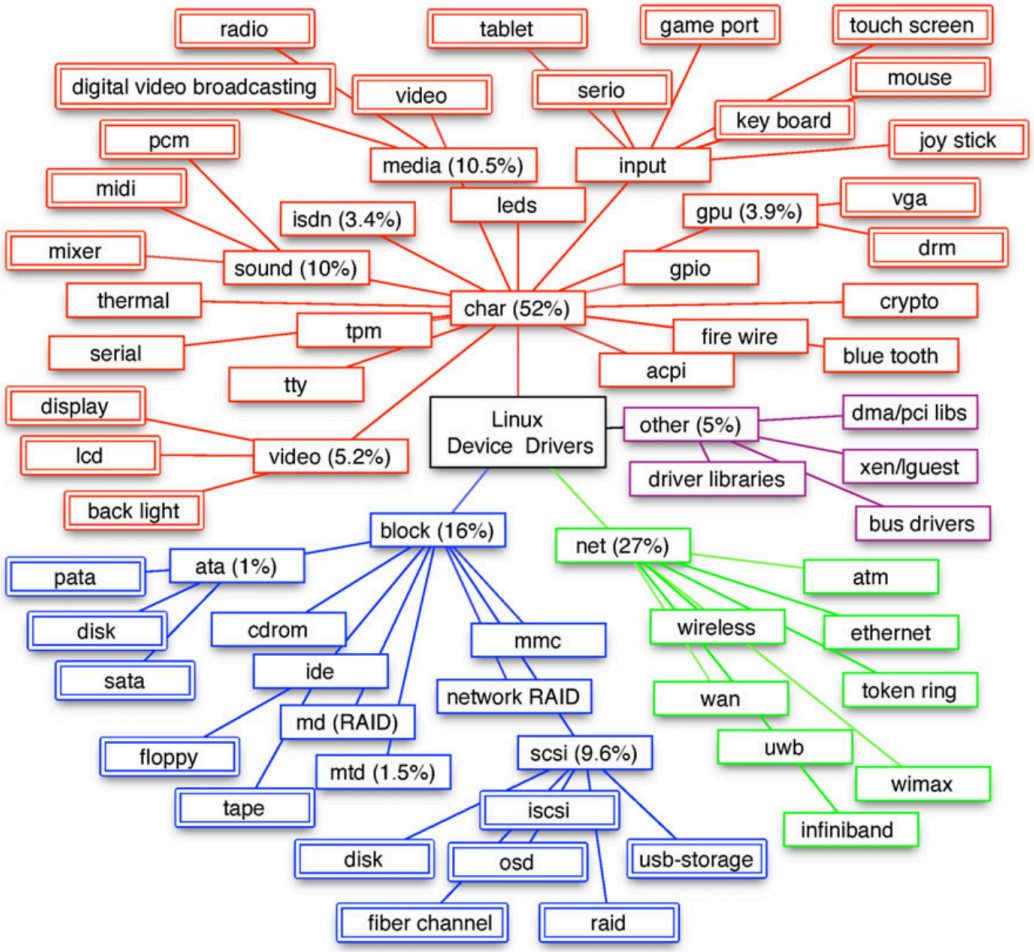

Figure 1. The Linux driver taxonomy in terms of basic driver classes. The size (in percentage of lines of code) is mentioned for 5 biggest classes. Not all driver classes are mentioned.

Most driver research neglects the heavy tail of character devices.

- video and GPU drivers contribute significantly towards driver code(9%) due to complex device with instruction set that change each generation.

Driver interactions
Class membership: Drivers belong to common set of classes, and the class completely determines their behavior.
Procedure calls: Drivers always communicate with the kernel through procedure calls.
Driver state: The state of the device is completely captured by the driver.
Device state: Drivers may assume that devices are in the correct state.
Driver architecture
I/O: Driver code functionality is only dedicated to converting requests from the kernel to the device.
Chipsets: Drivers typically support one or a few device chipsets.
CPU Bound: Drivers do little processing and mostly act as a library for binding different interfaces together.
Event-driven: Drivers execute only in response to kernel and device requests, and to not have their own threads.

Table 2. Common assumptions made in device driver research.

这个research提供了一些common assumptions, and then it will test and find whether all of them still holds

What do Drivers do

Device drivers are commonly assumed to primarily perform I/O.

- A device driver can be thought of 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.”

- However it does more than that.

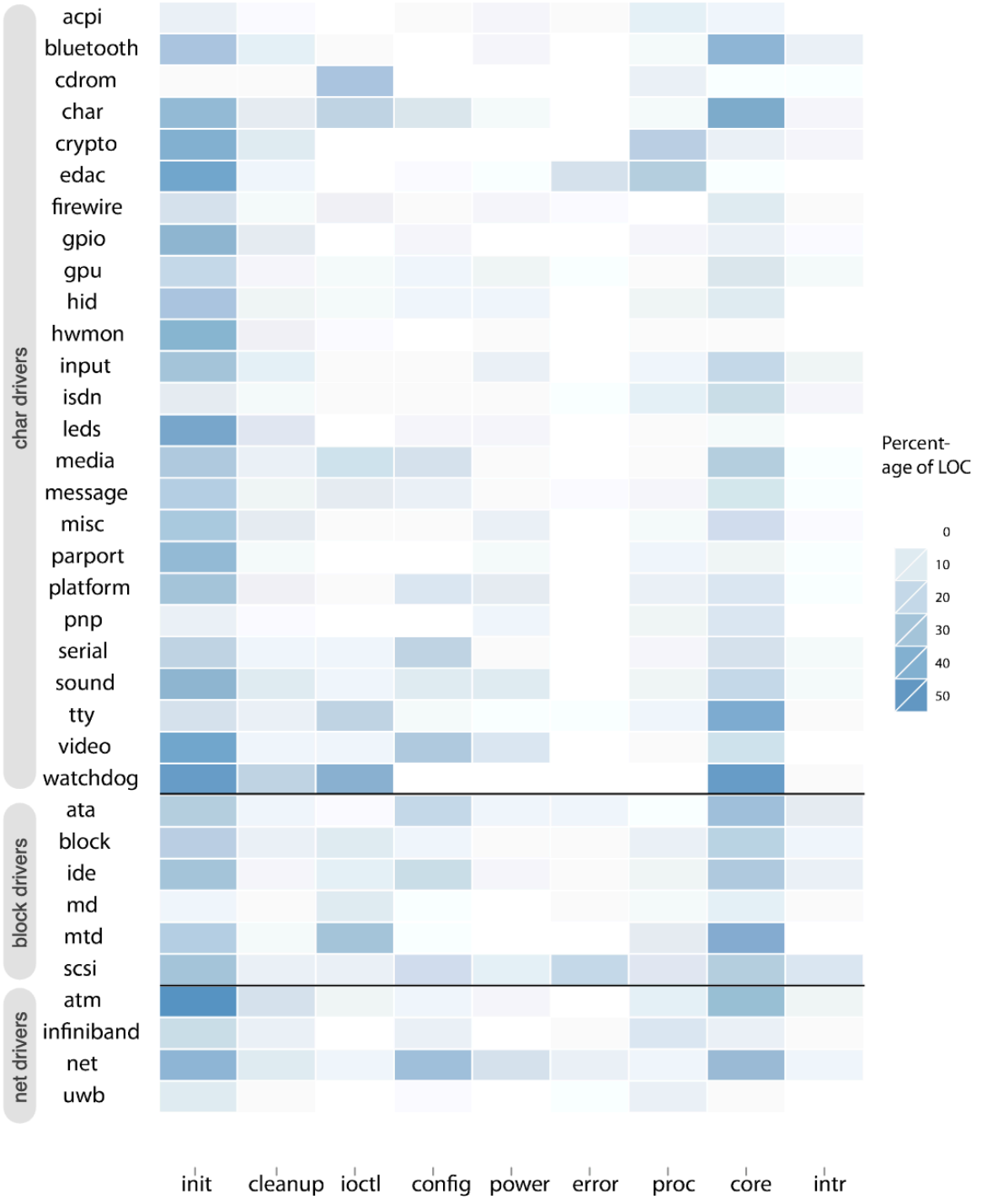

Figure 2. The percentage of driver code accessed during different driver activities across driver classes.

- The figure shows the fraction of driver code invoked during driver initialization, cleanup, ioctl processing, configuration, power management, error handling, /proc and /sys handling, and most importantly, core I/O request handling, and interrupt handling across different driver classes.

a simple driver for a single chipset may devote most of its code to processing I/O requests and have a simple initialization routine. In contrast, a complex driver supporting dozens of

chipsets may have more code devoted to initialization and error handling than to request handling.

These results indicate that efforts at reducing the complexity of drivers should not only focus on request handling, which accounts for only one fourth of the total code, but on better mechanisms for initialization and configuration.

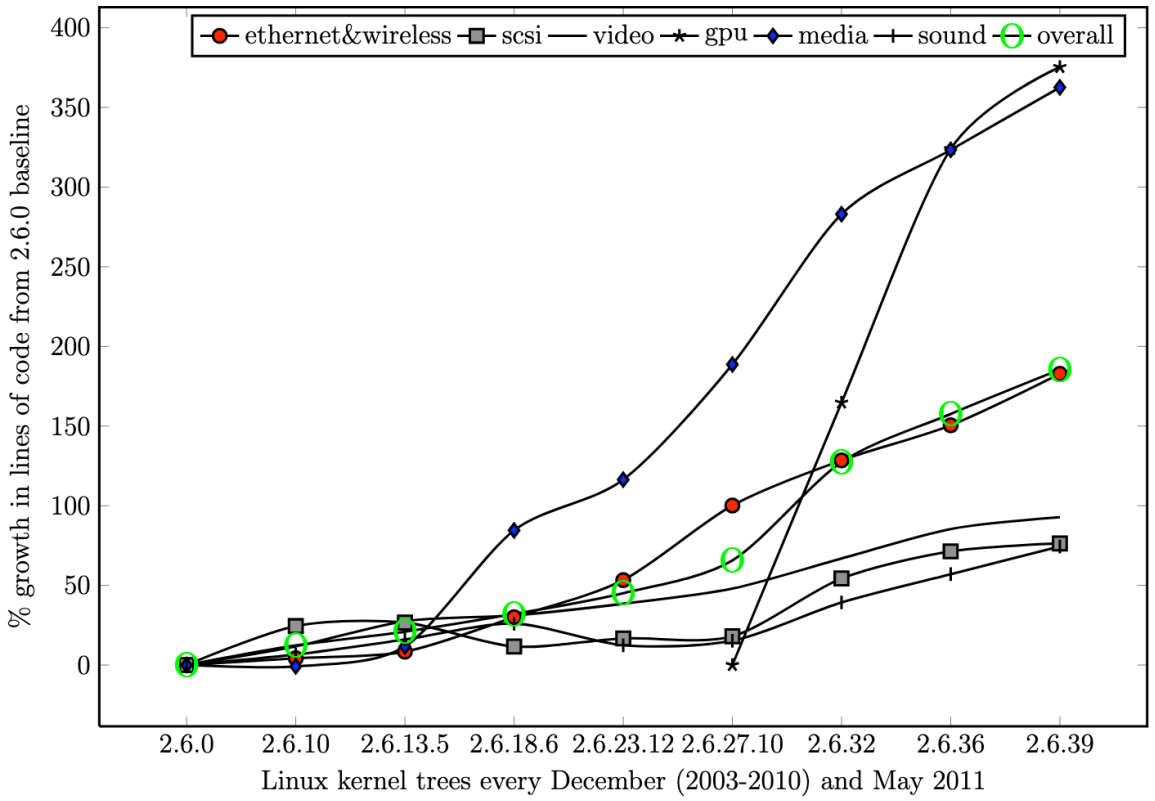

Figure 3. The change in driver code in terms of LOC across different driver classes between the Linux 2.6.0 and Linux 2.6.39 kernel.

Overall, driver code has increased by 185% over the last eight years.

- First, there is additional code for new hardware.
- Second, there is increasing support for certain class of devices, including network (driven by wireless), media, GPU and SCSI.
- While Ethernet and sound, the common driver classes for research, are important, research should look further into other rapidly changing drivers, such as media, GPU and wireless drivers.

Class Abstraction:

Many driver research projects assume that:

1. Every driver belongs to exactly one class
2. The class interface **completely determines** the driver's behavior
3. There's no functionality outside what the class defines

The authors discovered this assumption often fails:

- **44% of drivers** have behavior **outside** their class definition
- This "extra" behavior manifests as:
  - Private `ioctl` options (device-specific commands)
  - `/proc` or `/sys` entries (36% of drivers use these for load-time parameters)
    - `/proc` is the folder containing process and kernel information, like cpu info and stuff
    - `/sys` is a more modern and structured interface that exposes the **device model** — the hierarchy of buses, devices, and drivers. It's organized around the actual hardware topology.
  - Unique configuration options not part of the standard interface
  - This finding means research approaches that rely purely on class semantics won't work for nearly half of all drivers. Systems trying to automatically recover, synthesize, or verify drivers need to account for these class extensions.

Driver Compute

It is often assumed that drivers perform little processing and simply shuttle data between the OS and the device.

- If driver require substaintial CPU processing, then processing power must be reserved.
  - And if there's heavy I/O from one guest VM, it would substaintially reduce CPU availability for other guest VMs.

The research finds that 15% of the driver have at least one function that performs processing, and that processing occurs in 1% of all driver functions.

- 28% of sound and network do processing
- Wireless drivers perform processing to interpolate power levels of the device under different frequencies and other conditions.

Therefore a substantial fraction of drivers do some form of data processing.

- Thus, efforts to generate driver code automatically must include mechanisms for data processing, not just converting requests from the OS into requests to the device.
- New research opportunities: should all the ocmputation be offloaded to the device with embedded processors, and is there a performance or power benefits to doing so?

How many device chipsets does a single driver support?

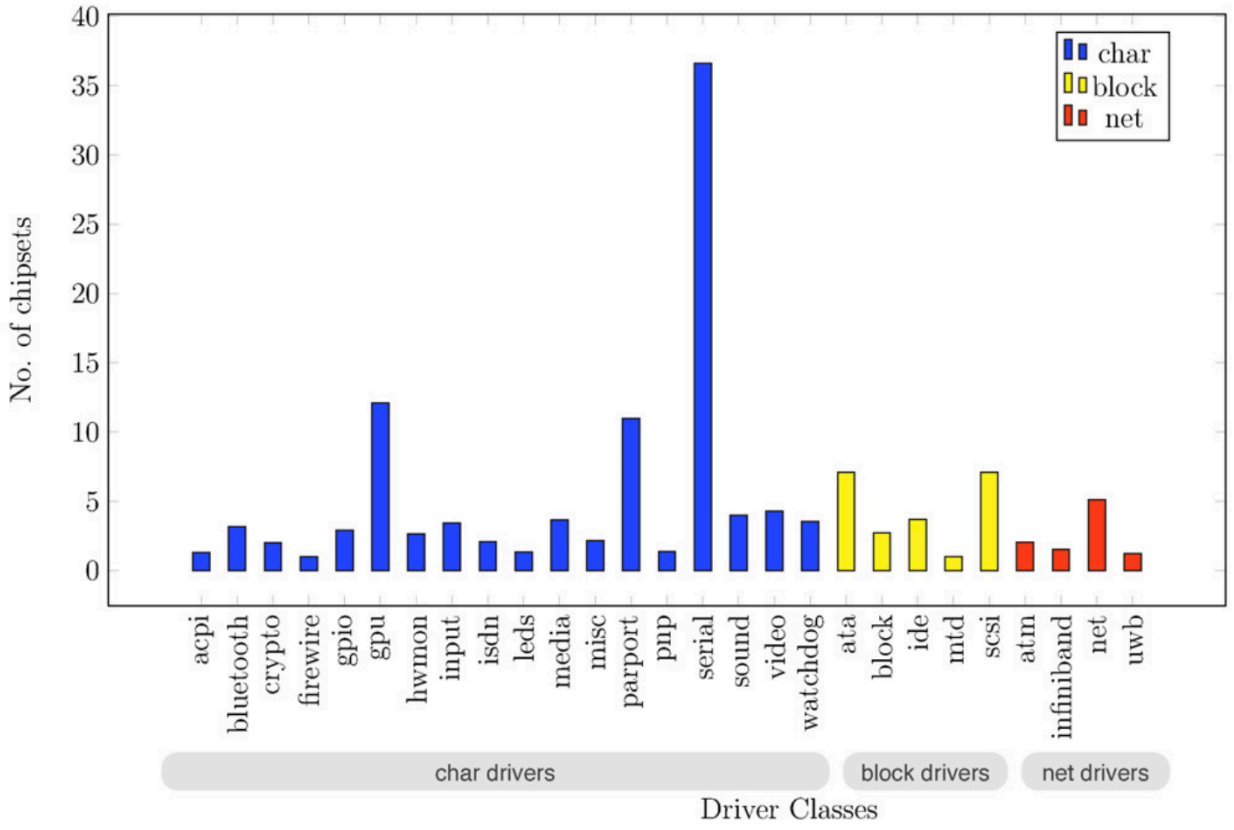

Figure 5. The average number of chipsets supported by drivers in each class.

The study measured how many chipsets each Linux driver supports by counting device IDs (PCI, USB, I2C, etc.) that each driver recognizes.

- Serial drivers support an average of 36 chipsets per driver
- Network drivers average 5 chipsets per driver
- Most other classes support only a few chipsets per driver
- Generic USB drivers like `usb-storage` and `usb-audio` support over **200 chipsets** each
- The `usb-serial` driver supports more than **500 chipsets**
- Linux supports **14,070 devices** with only **3,217 drivers** — roughly **400 lines of code per device**

```
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 */
   ...
}
```

Figure 6. The cyclades character drivers supports eight chipsets that behaves differently at each phase of execution. This makes driver code space efficient but extremely complex to understand.

Any system that generates unique drivers for every chipset or requires per-chipset manual specification may lead to a great expansion in driver code and complexity. Furthermore, there is substantial complexity in supporting multiple chipsets, as seen in Figure 6, so better programming methodologies, such as object-oriented programming and automatic interface generation, should be investigated.

Driver Interactions

how do drivers use the kernel, and how do drivers communicate with devices

Driver/Kernel Interaction

Driver use kernel functions in five categories:

- **Kernel library**: Generic utility routines. Timers, string manipulation, checksums, data structures
- **Memory management**: Allocation services `kmalloc`, `kfree`, page allocation
- **Synchronization**: Concurrency control. Locks, semaphores, spinlocks

- **Device library**: Class-specific I/O support. Network subsystem helpers, SCSI library functions
- **Kernel services**: Access to other subsystems. File system, scheduling, process management

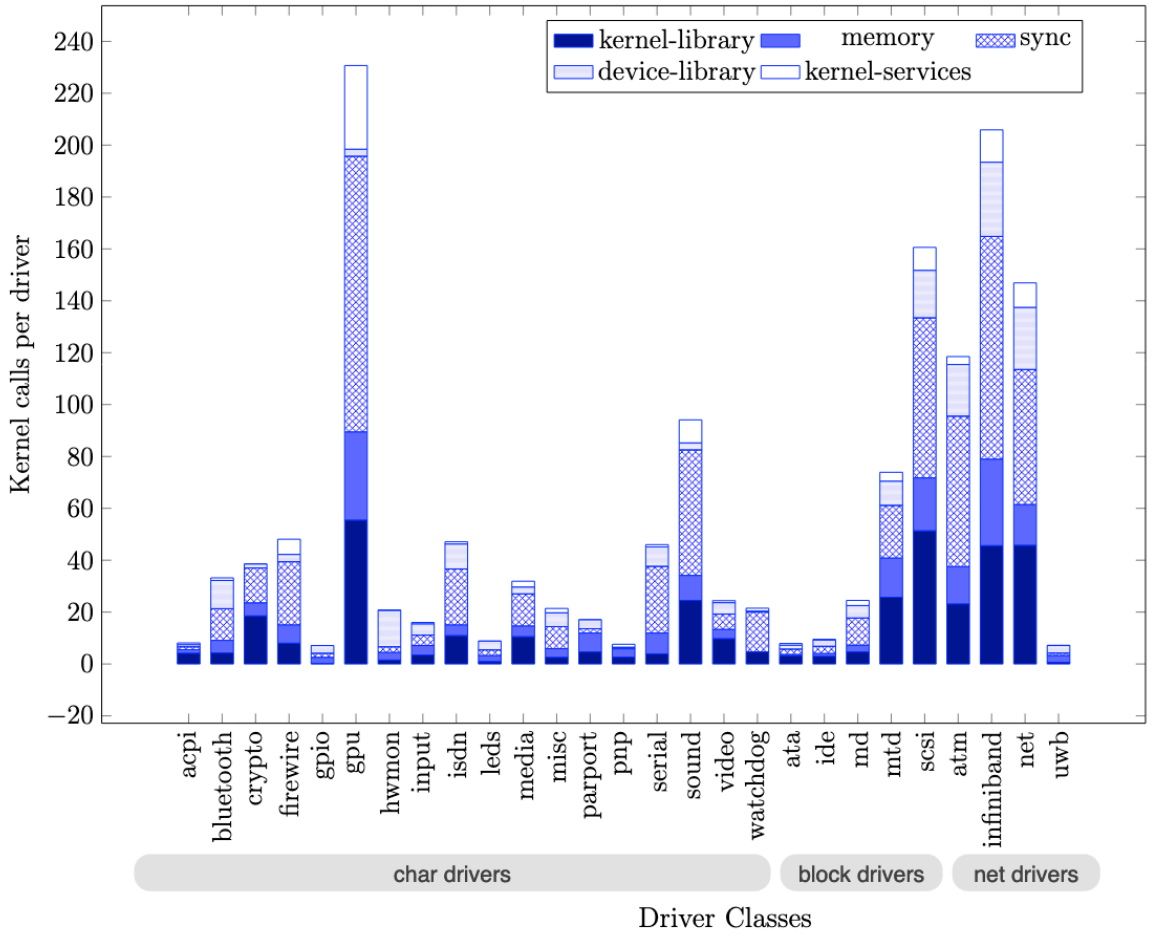

Figure 7. The average kernel library, memory, synchronization, kernel device library, and kernel services library calls per driver (bottom to top in figure) for all entry points.

- Finding 1: Most calls are for "local" operations
  - The majority of kernel invocations are for:
    - Kernel library routines
    - Memory management
    - Synchronization
  - These functions are **local to the driver** — they don't require interaction with other kernel subsystems. This means a driver running in a **separate execution context** (user mode, separate virtual machine, or on the device itself) could provide these services locally without calling into the kernel.
- Finding 2: Very few calls to kernel services
- Finding 3: Device library usage varies by abstraction level

    - Drivers with **rich library support** (network, SCSI) make many calls to device libraries
    - Drivers with **less library support** (GPU) primarily use generic kernel routines
- Finding 4: Some drivers use almost no kernel services
    - ATA, IDE, ACPI, and UWB drivers make very few kernel calls. These use a **"miniport" architecture** — the driver is just a small set of device-specific routines called by a larger common driver. Most functionality lives in the shared library, not the device-specific code.

The paper suggests that drivers can be isolated from kernel and can be run in a separate VM or OS running on the device.

The driver VM has its own small OS that provides:

- Memory allocation (its own `kmalloc`)
- Locks (its own synchronization primitives)
- Timers, string functions, etc.

The driver doesn't know or care that it's in a separate VM — these local services look identical to the real kernel services.

Driver/Device Interaction

Driver communicate with hardware devices in three categories:

- **Port I/O / MMIO(Memory mapped IO)**: Direct register access. CPU reads/writes specific addresses that map to device registers
- **DMA**: Direct Memory Access. Device reads/writes system memory directly without CPU involvement
- **Bus**: Bus protocol calls. Driver sends commands through a bus abstraction layer (USB, PCI)

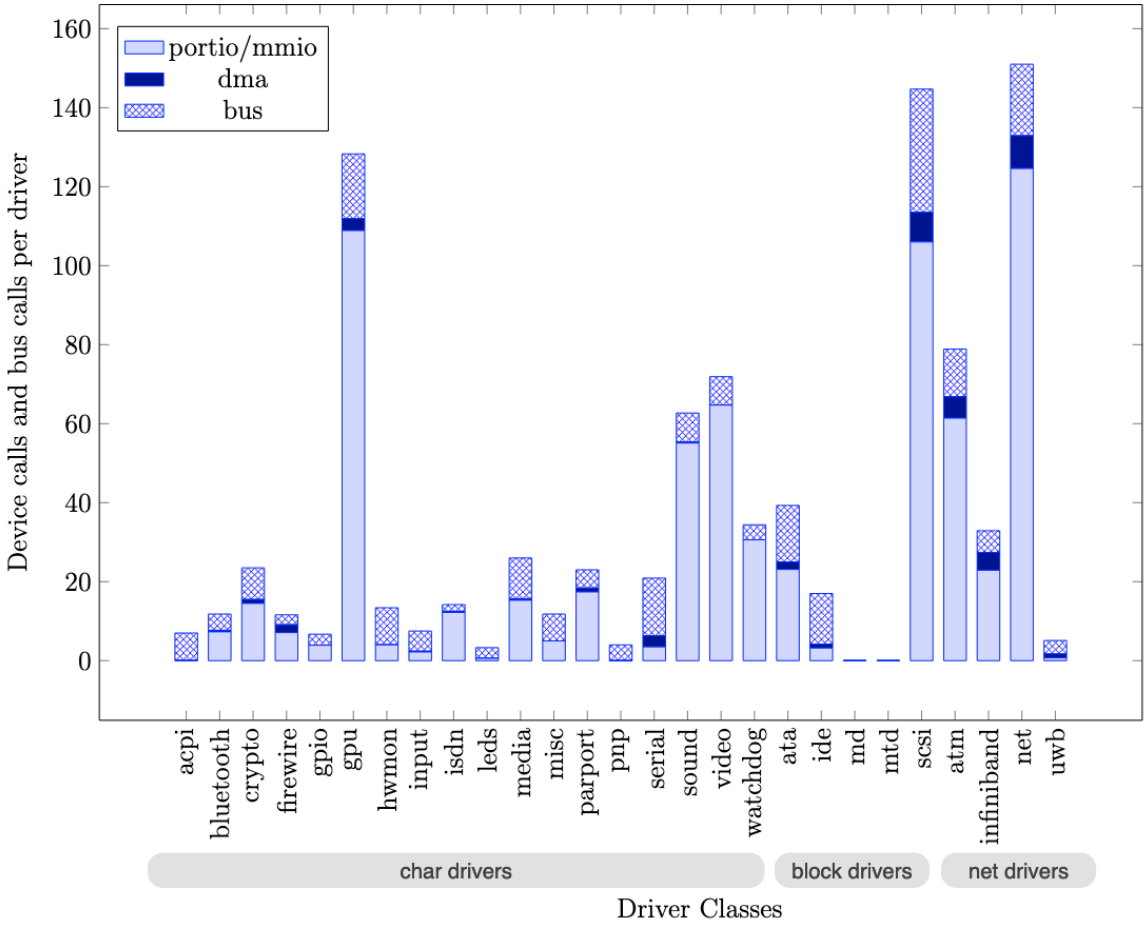

Figure 8. The device interaction pattern representing port I/O, memory mapped I/O, bus resources (bottom to top) invoked via all driver entry points.

Findings:

- Different classes use very different interaction styles:
- IDE and ATA drivers, although they're both block device
  - IDE do very little port or MMIO
    - They rely on PCI configuration space for accessing device registers
    - They show a greater proportion of bus operations
- Therefore Direct Interaction(Port IO/MMIO) device can use hardware protection by using separate VM or user space to isolate driver.
  - The driver can be isolated but still directly access the device
    because hardware (MMU, IOMMU) enforces which memory regions the driver can touch.
- However Bus calls driver requires software isolation, Every USB operation requires communicating with the kernel's USB subsystem, this means crossing protection boundaries, which adds overhead.
- Verifying a driver with 150 Port I/O calls is very different from verifying one with 15 bus calls
- The cost of verification varies by driver class

Driver/Bus Interaction

BUS	Kernel Interactions					Device Interactions			
	mem	*sync*	*dev lib.*	*kern lib.*	*kern services*	*port/mmio*	*dma*	*bus*	*avg devices/driver*
PCI	15.6	57.8	13.3	43.2	9.1	125.1	7.0	21.6	7.5
USB	9.6	25.5	5.6	9.9	3.0	0.0	2.2[2]	13.8	13.2
Xen	10.3	8.0	7.0	6.0	2.75	0.0	0.0	34.0	1/All

Table 3. Comparison of modern buses on drivers across all classes. Xen and USB drivers invoke the bus for the driver while PCI drivers invoke the device directly.

Although most Linux drivers attach to **PCI-class buses**, other buses like **USB** (for removable devices) and **XenBus** (virtual bus for Xen hypervisor) are increasingly important.

Device Support Efficiency:

- The PCI have the lowest average chipset supported(7.5), lacking of standardization, each vendor requires unique driver code.
- USB has the highest average chipset supported(13.2), very high standardization. For example, USB storage devices implement a standard interface, so one driver works for devices from many different manufacturers.
- XenBus is the most efficient one, because they don't access real hardware, they communicate with a backend driver in another VM that handles the actual device. So One XenBus network frontend driver works with **any** network card in the backend VM.

Kernel Interaction Comparison:

- XenBus has lowest kernel interaction count, because they need little minimal initialization, error handling. They're primarily just I/O request handling.
- PCI device has heavy Kernel interactions.
- Driver with few kernel interactions run more easily in other execution environments, such as the device itself.

Device Interaction Comparison：

- USB and XenBus drivers could efficiently access devices over a network because:
  - Fewer total interactions
  - Each interaction is coarse-grained (more work per request)
  - Lower overhead from batching

For moving drivers out of kernel:

- PCI buses drivers are harder, because they have many fine-grained interactions, and needs hardware protection.
- USB and XenBus drivers are easier, because they have fewer interactions, and already have standardized protocol codes. And XenBus drivers are already designed for virtualization and isolation.
  - However, USB device's standardization is incomplete, device-specific code still needed for initialization, suspend/resume, special features.

Driver Concurrency

when multiple applications or events need to use the same device simultaneously.

The authors identify two programming models:

- Threaded, use blocking and explicit synchronization, call `msleep()` waits for resources.
- Event-friendly, responds to event without blocking, use callbacks, completion routines.

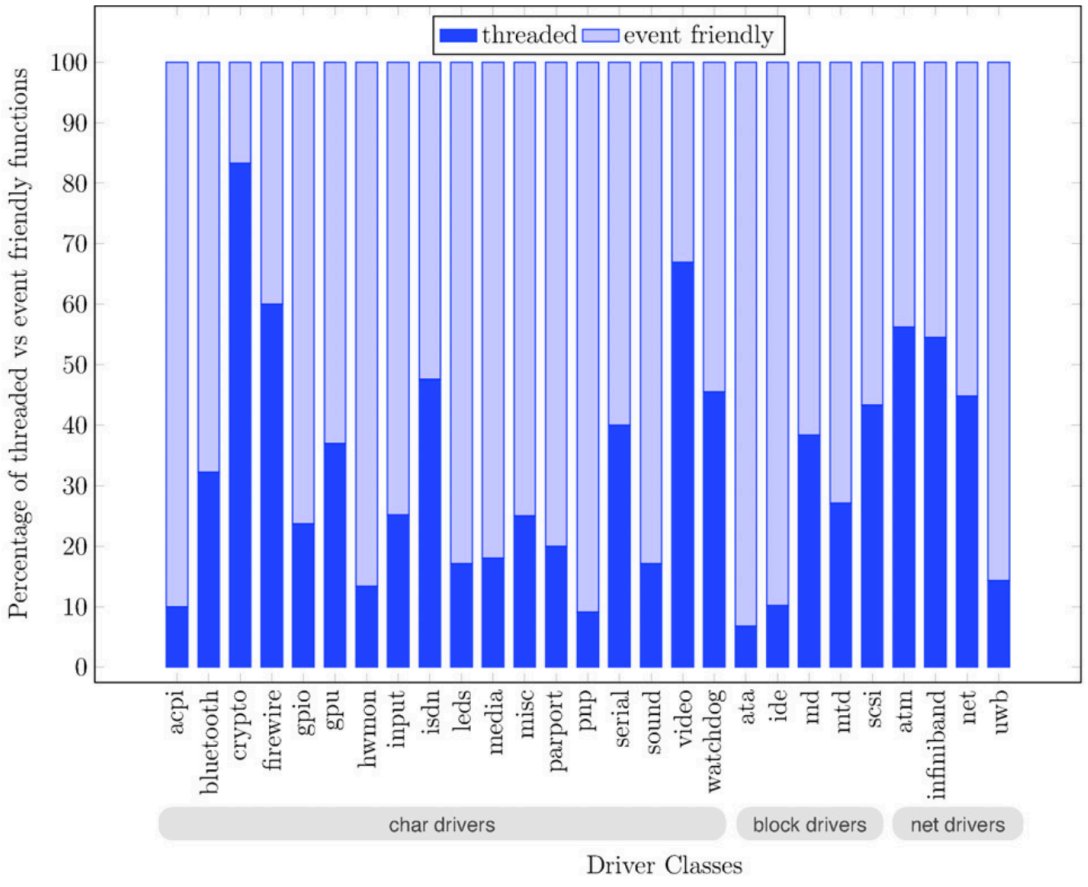

Figure 9. The percentage of driver entry points under coverage of threaded and event friendly synchronization primitives.

No single model dominates. Drivers use both methods for different purposes:

- **Threaded primitives**: Synchronizing driver/device operations during initialization, updating global data structures
- **Event-friendly primitives**: Core I/O request handling

For running the drivers outside the kernel, threaded code is difficult

- Blocking requires kernel thread support
- he paper mentions Microdrivers as an example, they execute all driver code in an event-like fashion, restricting invocation to single requests at a time.

For driver architecture improvement:

- The paper suggest vonverting drivers to use **only event-based synchronization** would:
  - Simplify running drivers outside the kernel
  - Match naturally with message-passing architectures (like XenBus)

Driver Redundancy

If drivers for similar devices share common patterns, this redundant code could potentially be abstracted into libraries to reduce overall code size and complexity.

The research find 8% of all driver code is substantially similar to code elsewhere.

Three similarity levels:

- Within a single driver
  - The most common form is **wrappers around device I/O or kernel functions**:
  - These wrappers either convert data into the appropriate format or perform an associated support operation that is required before calling the routines but differ from one another because they lie on a different code path.

```
DrComp signature:1.594751
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(..);
}
```

```
DrComp signature:1.594751
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(..);
 }
```

Figure 11. The above figure shows identical code that consumes different register values. Such code is present in drivers where multiple chipsets are supported as well as across drivers of different devices. The functions are copies except for the constants as shown in the boxes.

- Across entire driver classes
  - Many drivers differ only in **constant values** (device registers, flag values)
  - The functions are **structurally identical**, only the constants differ.
- Across subsets of drivers in a class
  - Some patterns appear in groups of related drivers
  - Refactoring these common interface to pull them into a library could simplify these drivers.

Reduce Redundancy

- Procedural Abstraction: Move shared code to a library and parameterize the differences
- Better Multiple Chipset Support: Instead of separate drivers with copied code, support multiple chipsets in one driver
- Table-Driven Programming: Replace code that differs only in constants with data tables

Driver class	Driver subclass	Similar code fragments	Fragment clusters	Fragment size (Avg. LOC)	Redundancy results and action items to remove redundant code
char	acpi	64	32	15.1	Procedural abstraction for centralized access to kernel resources and passing get/set configuration information as arguments for large function pairs.
	gpu	234	108	16.9	Procedural abstractions for device access. Code replicated across drivers, like in DMA buffer code for savage, radeon, rage drivers, can be removed by supporting more devices per driver.
	isdn	277	118	21.0	Procedural abstraction for kernel wrappers. Driver abstraction/common library for ISDN cards in hisax directories.
	input	125	48	17.23	Procedural abstraction for kernel wrappers. Driver abstraction/common driver for all touchscreen drivers. Procedural abstraction in Aiptek tablet driver.
	media	1116	445	16.5	Class libraries for all Micron image sensor drivers. Procedural abstraction in saa 7164 A/V decoder driver and ALI 5602 webcam driver.
	video	201	88	20	Class libraries for ARK2000PV, S3Trio, VIA VT8623drivers in init/cleanup, power management and frame buffer operations. Procedural abstraction in VESA VGA drivers for all driver information functions.
	sound	1149	459	15.1	Single driver for ICE1712 and ICE1724 ALSA drivers. Procedural abstraction for invoking sound libraries, instead of repeated code with different flags. Procedural abstraction for AC97 driver and ALSA driver for RME HDSPM audio interface.
block	ata	68	29	13.3	Common power management library for ALI 15x3, CMD640 PCI, Highpoint ATA controllers, Ninja32, CIL 680, ARTOP 867X, HPT3x3, NS87415 PATA drivers and SIS ATA driver. Table driven programming for device access in these drivers.
	ide	18	9	15.3	Procedural abstraction for the few wrappers around power management routines.
	scsi	789	332	25.6	Shared library for kernel/scsi wrappers for Qlogic HBA drivers; pmc sierra and marvell mvsas drivers. Large redundant wrappers in mp2sas firmware, Brocade FC port access code.
net	Ethernet/ wireless	1906	807	25.1	Shared library for wireless drivers for talking to device/kernel and wireless routines. Lot of NICs share code for most routines like configuration, resource allocation and can be moved to a single driver with support for multiple chipsets. A driver sub-class library for all or vendor specific Ethernet drivers.
	infiniband	138	60	15.0	Procedural abstraction for Intel nes driver.

Table 4. The total number of similar code fragments and fragment clusters across driver classes and action items that can be taken to reduce them.

Why is this matter?

- Duplicated code is a well-documented source of bugs. When a bug is fixed in one copy, other copies may be forgotte
- With 8% redundant code in 5 million lines, that's approximately **400,000 lines** that could potentially be eliminated or consolidated. Hence can reduce maintenance burden
- Shared libraries ensure consistent behavior across drivers and make it easier to update functionality across the entire driver class. Hence improve consistency.