Software Support for Improved Driver Reliability

Michael Swift
University of Wisconsin--Madison

The (old) elephant in the room

- Device drivers are:
  - The majority of kernel code written
  - A large fraction of kernel code in memory
  - Unreliable
    - 89% of Windows XP crashes are from drivers
    - Linux drivers had 7x bug density of other kernel code

  - Difficult to write, test, and maintain

Driver programming has not changed much

- Unix Version 3, 1973
  - (dn.c, a DN-11 modem)

```
dnwrite(dev) {
   struct dn *dp;
   register struct dn *rdp;
   int c;
   dp = &DNADDR->dn11[dev.d_minor];
   for(;;) {
     while (((rdp = dp)-
              >dn_stat&DONE)==0)
          sleep(DNADDR, DNPRI);
     rdp->dn_stat =& ~DONE;
     if (rdp->dn_reg&(PWI|ACR)) {
          u.u_error = EIO;
          return;
     }
     if (rdp->dn_stat&DSS) return;
     rdp = dp;
     rdp->dn_reg = c-'0';
     rdp->dn_stat =| DPR;
```

- Linux 2.6.23, 2007
  - esp.c, a serial port driver

```
static int rs_write(struct tty_struct * tty,
                    const unsigned char *buf,
                     int count) {
     int     c, t, ret = 0;
     struct esp_struct *info = (struct esp_struct
            *)tty->driver_data;
     unsigned long flags;
     while (1) {
      c = count;
      t = ESP_XMIT_SIZE - info->xmit_cnt - 1;
      memcpy(info->xmit_buf + info->xmit_head,
            buf, c);
      info->xmit_head = (info->xmit_head + c) &
                        (ESP_XMIT_SIZE-1);
       …
     }
    serial_out(info, UART_ESI_CMD1,
             ESI_SET_SRV_MASK);
    serial_out(info, UART_ESI_CMD2, info->IER);
```

Everything else has changed

- Unix Version 3, 1973
  - 16 drivers
  - 36 KB of driver code
  - Written by Dennis Ritchie

- Linux 2.6.31.6, 2009
  - 4254 driver variations
  - 204 MB of driver code
  - 4.5 million lines of code
  - Written by > 374 people
  - ~1400 person-years of effort

Writing quality drivers is hard

- Kernel programming is hard
  - Few tools
  - Hard to debug
- Many rules to follow
  - Which locks can be used
  - Which memory can be touched
  - Which order operations must follow
- Hardware unreliability
  - May fail independently and transiently

- USB Homer Simpson doll was an epiphany for Microsoft

Software Support for Driver Reliability

- State of the art:
  - Driver isolation: allow existing drivers to fail and clean up
    - Nooks
    - SafeDrive
  - Driver architecture: new designs for new drivers to reduce faults and tolerate failures
    - Minix 3
    - Windows UMDF, KMDF
    - UNSW Dingo & Termite
- Our approach: tool + runtime
  - Carburizer: detect and tolerate device failures in software
  - Decaf Drivers: simplify driver development through language

Outline

- Introduction
- Carburizer
  - Hardening Drivers
  - Reporting Errors
- Decaf Drivers
- Conclusions

Current state of OS-hardware interaction

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

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

HANG!

Hardware dependence bug: device malfunction can crash the system

Current state of OS-hardware interaction

- Hardware dependence bugs occur across driver classes
- Manifest as hangs, crashes, incorrect behavior

```
void hptitop_iop_request_callback(...)     {

arg = readl(...);
  ...
if (readl(&req->result) == IOP_SUCCESS)  {
    arg->result = HPT_IOCTL_OK;
  }
}
```

`Highpoint SCSI driver(hptiop.c)`

*Code simplified for presentation purposes

How do the hardware bugs manifest?

- Drivers often trust hardware to always work correctly
  - Drivers use device data in critical control and data paths
  - Drivers do not report device malfunctions to system log
  - Drivers do not detect or recover from device failures

An example: Windows servers

- 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 driver:
  - Crashes reduced from 8% to 3%

---

[1] Fault resilient drivers for Longhorn server, May 2004. Microsoft Corp.

Carburizer

- Goal: Tolerate hardware device failures in software through hardware failure detection and recovery

- Static analysis tool - analyze and insert code to:
  - Detect and fix hardware dependence bugs
  - Detect and generate missing error reporting information

- Runtime
  - Handle interrupt failures
  - Transparently recover from failures

Outline

- Introduction
- Carburizer
  - Hardening Drivers
  - Reporting Errors
- Decaf Drivers
- Conclusions

Hardware unreliability

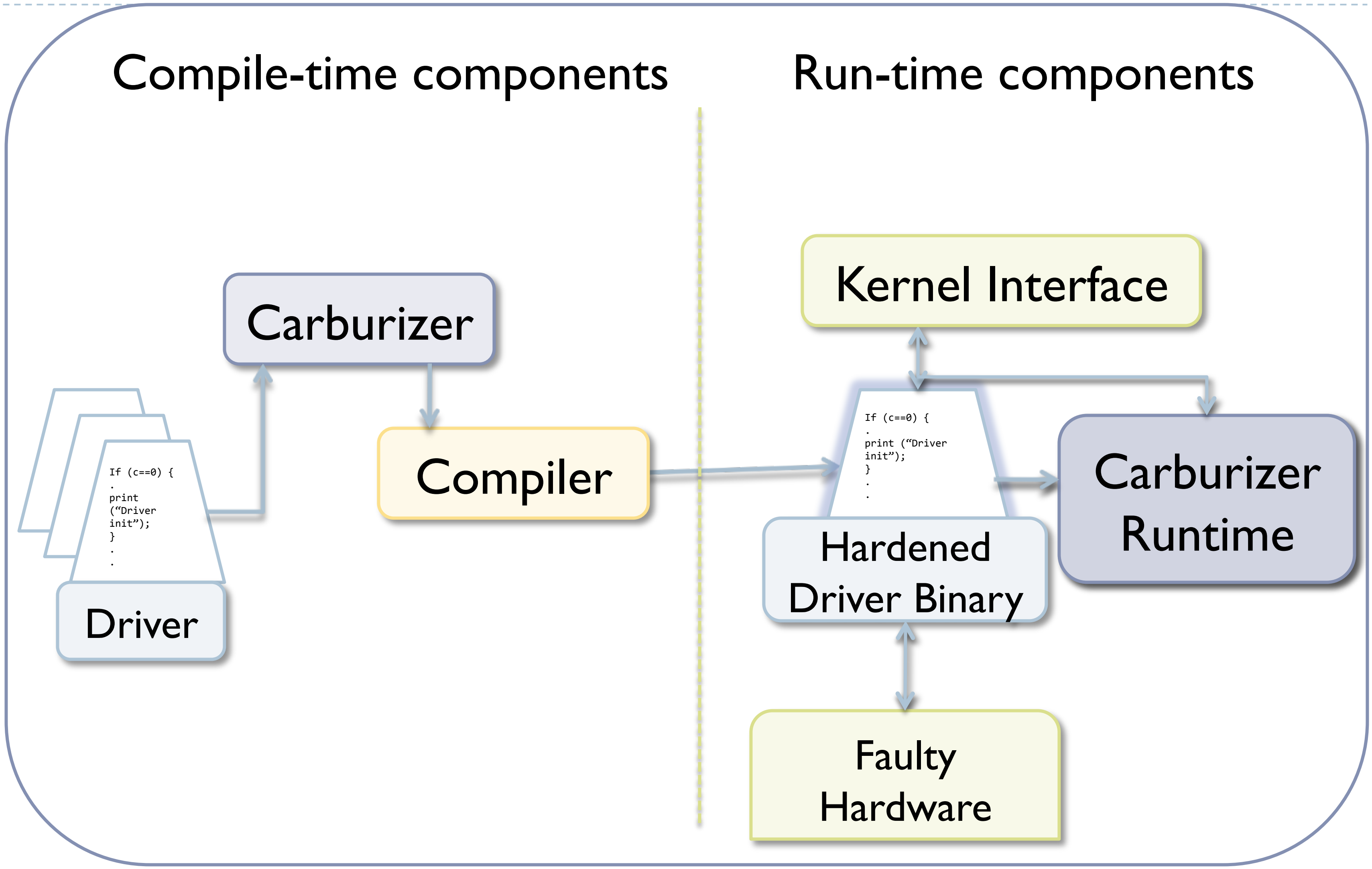
- Sources of hardware misbehavior:
  - Device wear-out, insufficient burn-in
  - Bridging faults
  - Electromagnetic radiation
  - Firmware bugs
- Result of misbehavior:
  - Corrupted/stuck-at inputs
  - Timing errors/unpredictable DMA
  - Interrupt storms/missing interrupts

Vendor recommendations for driver developers

Recommendation	**Summary**	**Recommended by**			
		Intel	Sun	MS	Linux
Validation	Input validation	●	●	●	
	Read once& CRC data	●	●		●
	DMA protection	●	●		
Timing	Infinite polling	●	●	●	
	Unexpected events	●		●	
Reporting	Report all failures	●	●	●	
Recovery	Handle all failures		●	●	
	Cleanup correctly	●	●		
	Do not crash on failure	●		●	●
	Wrap I/O memory access	●	●	●	●

Goal: *Automatically* implement as many recommendations as possible in commodity drivers

Carburizer architecture



Compile-time components

Run-time components

Driver

Carburizer

Compiler

Kernel Interface

Hardened Driver Binary

Carburizer Runtime

Faulty Hardware

Outline

- Introduction
- Carburizer
  - Hardening Drivers
    - Finding sensitive code
    - Repairing code
  - Reporting Errors
- Decaf Drivers
- Conclusions

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 from
  - Infinite polling
  - Unsafe static/dynamic array reference
  - Unsafe pointer dereferences
  - System panic calls

Hardening drivers

- Finding sensitive code
  - First pass: Identify tainted variables

Finding sensitive code

First pass: Identify tainted variables

```
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

- 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

Example: Infinite polling

Finding sensitive code

```
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)

Not all bugs are obvious

```
 while (DAC960_PD_StatusAvailableP(ControllerBaseAddress))
 {
      DAC960_V1_CommandIdentifier_T CommandIdentifier= DAC960_PD_ReadStatusCommandIdentifier
                                                                (ControllerBaseAddress);
      DAC960_Command_T *Command = Controller ->Commands [CommandIdentifier-1];
      DAC960_V1_CommandMailbox_T *CommandMailbox = &Command->V1.CommandMailbox;
      DAC960_V1_CommandOpcode_T CommandOpcode=CommandMailbox->Common.CommandOpcode;
      Command->V1.CommandStatus =DAC960_PD_ReadStatusRegister(ControllerBaseAddress);
      DAC960_PD_AcknowledgeInterrupt(ControllerBaseAddress);
      DAC960_PD_AcknowledgeStatus(ControllerBaseAddress);
      switch (CommandOpcode)
        {
        case DAC960_V1_Enquiry_Old:
           DAC960_P_To_PD_TranslateReadWriteCommand(CommandMailbox);
           …
}
```

DAC960 Raid Controller(DAC960.c)

Detecting risky uses of tainted variables

- Example II: Unsafe array accesses
  - Tainted variables used as array index into static or dynamic arrays
  - Tainted variables used as pointers

Example: Unsafe array accesses

Unsafe array accesses

```
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

- 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

- Additional analyses to detect existing validation code

Analysis results over the Linux kernel

Driver class	Infinite polling	Static array	Dynamic array	Panic calls
net	117	2	21	2
scsi	298	31	22	121
sound	64	1	0	2
video	174	0	22	22
other	381	9	57	32
Total	860	43	89	179

Many cases of poorly written drivers with hardware dependence bugs

Repairing drivers

- Hardware dependence bugs difficult to test
- Carburizer automatically generates repair code
  - Inserts timeout code for infinite loops
  - Inserts checks for unsafe array/pointer references
  - Replaces calls to panic() with recovery service
  - Triggers generic recovery service on device failure

Carburizer automatically fixes infinite loops

```
timeout = rdstcll(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)
           rdstcll(_cur);
       else
           __recover_driver();


}
```

Timeout code added

AMD 8111e network driver(amd8111e.c)

*Code simplified for presentation purposes

Carburizer automatically adds bounds checks

```
static void __init attach_pas_card(...)
{

  if ((pas_model = pas_read(0xFF88)))
  {
    ...
    if ((pas_model< 0)) || (pas_model>= 5))
       __recover_driver();
     .
    sprintf(temp, “%s rev %d”,
      pas_model_names[(int) pas_model], pas_read(0x2789));

}
```

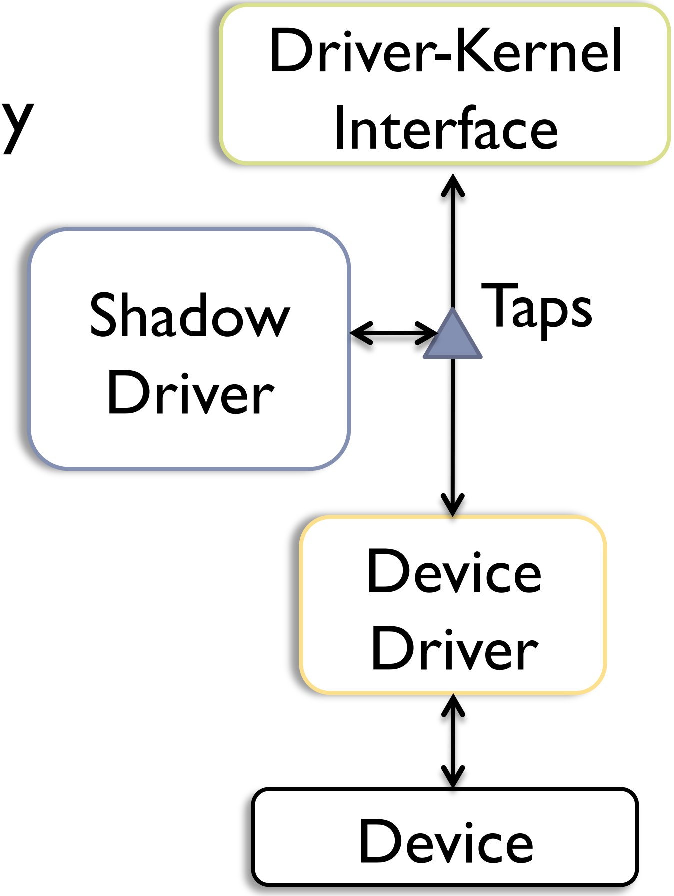
Array bounds check added

Pro Audio Sound driver (pas2_card.c)

*Code simplified for presentation purposes

Runtime fault recovery

- Low cost transparent recovery
  - Based on shadow drivers
  - Records state of driver
  - Transparent restart and state replay on failure
- Independent of any isolation mechanism (like Nooks)



Experimental validation

- Synthetic fault injection on network drivers
  - Modify I/O routines to return modified values
- Results

Device/ Driver	Original Driver		Carburized Driver		
	Behavior	**Detection**	**Behavior**	**Detection**	**Recovery**
3COM 3C905	CRASH	None	RUNNING	Yes	Yes
DEC DC 21x4x	CRASH	None	RUNNING	Yes	Yes

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

Outline

- Introduction
- Carburizer
  - Hardening Drivers
  - Reporting Errors
- Decaf Drivers
- Conclusions

Reporting errors

- Drivers often fail silently and fail to report device errors
  - Drivers should proactively report device failures
  - Fault management systems require these inputs

- Driver already detects failure but does not report them

- Carburizer analysis performs two functions
  - Detect when there is a device failure
  - Report unless the driver is already reporting the failure

Detecting driver detected device failures

- Detect code that depends on tainted variables
  - Perform unreported loop timeouts
  - Returns negative error constants
  - Jumps to common cleanup code

```
while (ioread16 (regA) == 0x0f)  {
  if (timeout++ == 200)  {
    sys_report(“Device timed out %s.\n”, mod_name);
    return (-1);
  }
}
```

Reporting code
added by
Carburizer

Detecting existing reporting code

Carburizer detects function calls with string arguments

```
static u16 gm_phy_read(...)
{
  ...
 if (__gm_phy_read(...))
   printk(KERN_WARNING "%s: ...\n”, ...);
```

Carburizer detects existing reporting code

`SysKonnect network driver(skge.c)`

Evaluation

- Manual analysis of drivers of different classes

Driver	**Class**	**Driver detected device failures**	**Carburizer reported failures**
ens1371	sound	10	9

Carburizer *automatically* improves the fault diagnosis capabilities of the system

- No false positives
- Fixed 1135 cases of unreported timeouts and 467 cases of unreported device failures in Linux drivers

Carburizer Summary

Recommendation	Summary	Recommended by			
		Intel	Sun	MS	Linux
Validation	Input validation	●	●	●	
	Read once& CRC data	●	●		●
	DMA protection	●	●		
Timing	Infinite polling	●	●	●	
	Stuck interrupt		●		
	Lost request			●	
	Avoid excess delay in OS			●	
	Unexpected events	●		●	
Reporting	Report all failures	●	●	●	
Recovery	Handle all failures		●	●	
	Cleanup correctly	●	●		
	Do not crash on failure	●		●	●
	Wrap I/O memory access	●	●	●	●

Carburizer Summary

Recommendation	Summary	Recommended by				Carburizer Ensures
		Intel	Sun	MS	Linux	
Validation	Input validation	●	●	●		●
	Read once& CRC data	●	●		●	
	DMA protection	●	●			
Timing	Infinite polling	●	●	●		●
	Unexpected events	●		●		
Reporting	Report all failures	●	●	●		●
Recovery	Handle all failures		●	●		●
	Cleanup correctly	●	●			●
	Do not crash on failure	●		●	●	●
	Wrap I/O memory access	●	●	●	●	

Carburizer improves system reliability by *automatically* ensuring that hardware failures are tolerated in software

Outline

- Introduction
- Carburizer
- Decaf Drivers
  - Intuition
  - Driver Slicer
  - Runtime
  - Evaluation
- Conclusions

Intuition

- Kernel programming is difficult and leads to driver unreliability
- For compatibility and performance, some driver tasks should remain in the kernel.
- Many driver tasks need not
  - Initialization/shutdown
  - Configuration
  - Error handling
- Thesis: there are better languages than C and better places than the kernel to run this code.

Why change drivers?

- Kernel environment is brittle
    - Sensitive to pointer problems
    - Difficult/cumbersome memory management
    - Little support for error handling
    - Difficult to debug

Kernel vs. Java development

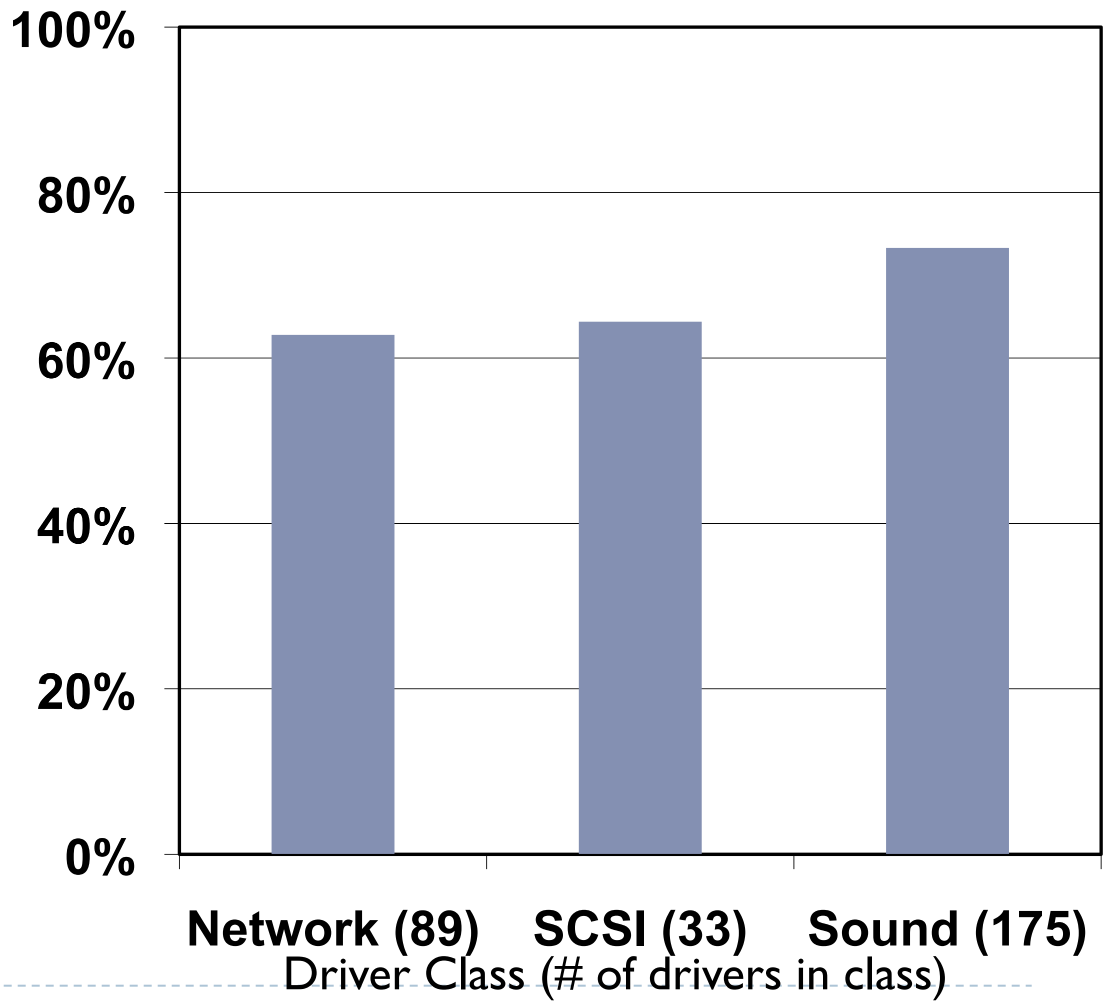
Feature	Kernel	Java
Memory management	Manual	Garbage collection
Type safety	Limited	Extensive
Debugging	Few tools / difficult	Many tools / easier
Data structure library	Subset of libc	Java class library
Error handling	Return values	Exceptions

Potential for change

How much code can *potentially* be moved from the kernel?

Up to 1.8 million lines of code

Portion of code removable in Linux 2.6.27

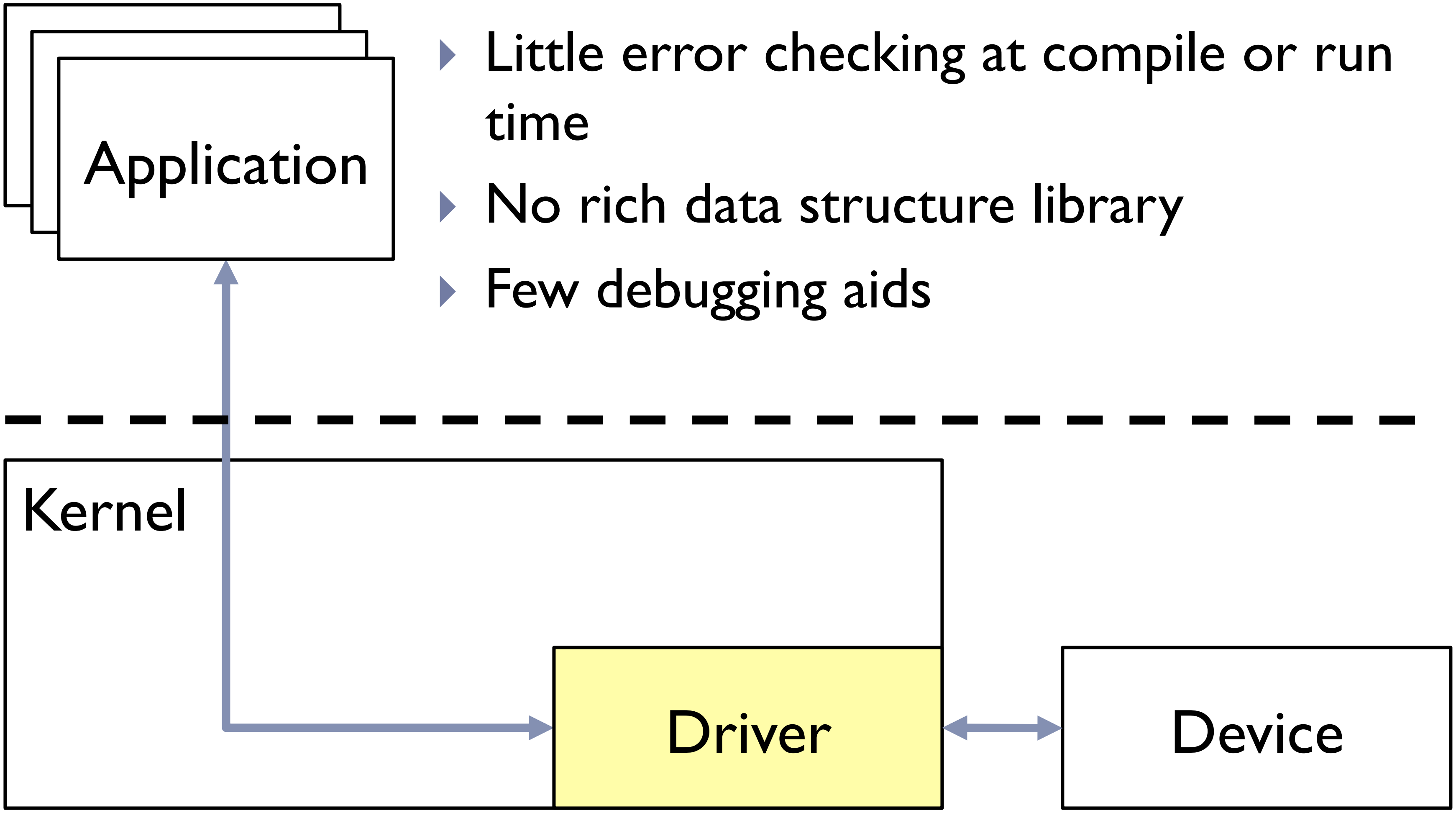


Decaf Drivers

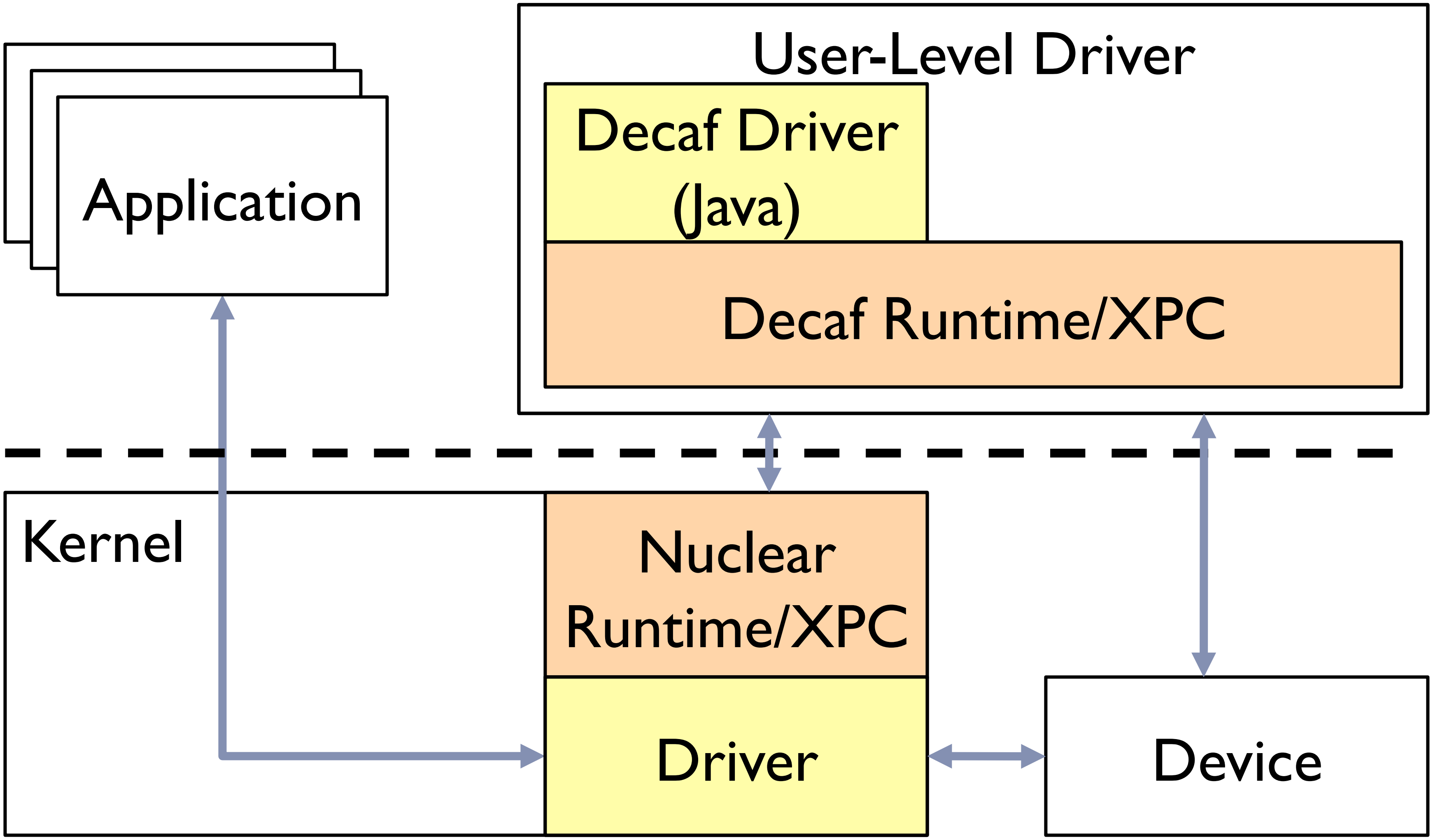
- *Decaf Drivers* execute most driver code in user mode Java
  - Performance critical code left in kernel
- The *Decaf System* provides support for
  1. Migrating driver code into a modern language (Java)
  2. Executing drivers with high performance
  3. Evolving drivers over time

Existing Driver Architecture

- Little error checking at compile or run time
- No rich data structure library
- Few debugging aids



Decaf Architecture



Application

User-Level Driver

Decaf Driver (Java)

Decaf Runtime/XPC

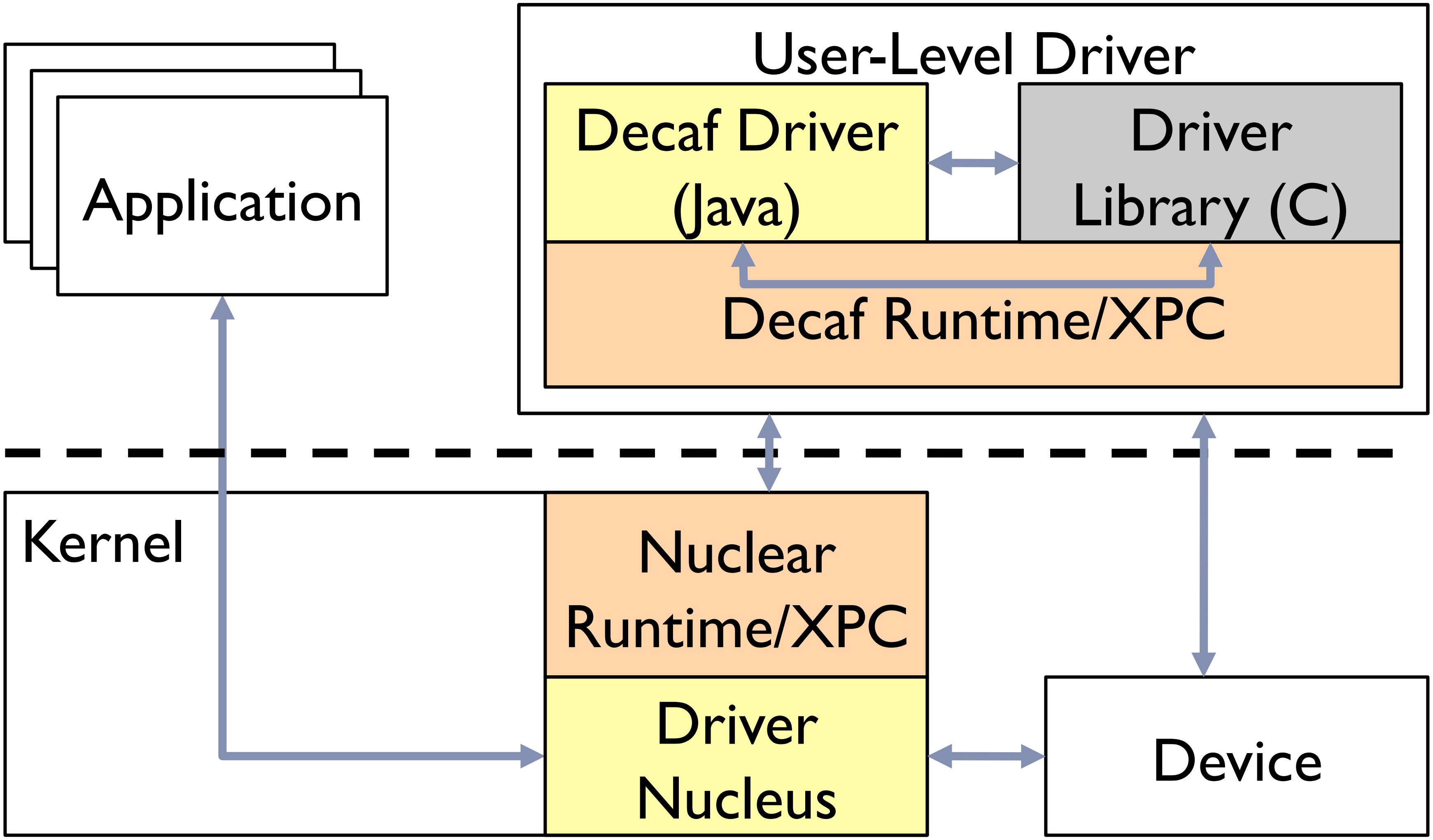
Kernel

Nuclear Runtime/XPC

Driver

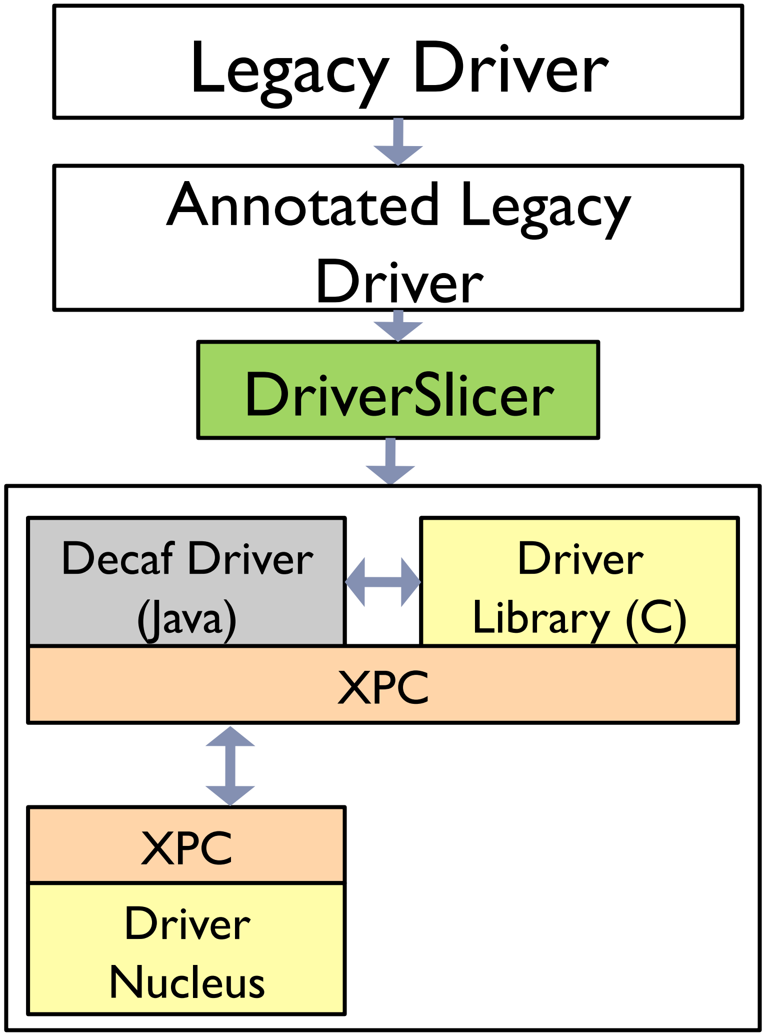
Device

Decaf Architecture



Creating Decaf Drivers

1. Annotate it
2. Run DriverSlicer to split the driver into a Driver Nucleus and Library
3. Migrate code from the Driver Library into the Decaf Driver



Splitting a driver

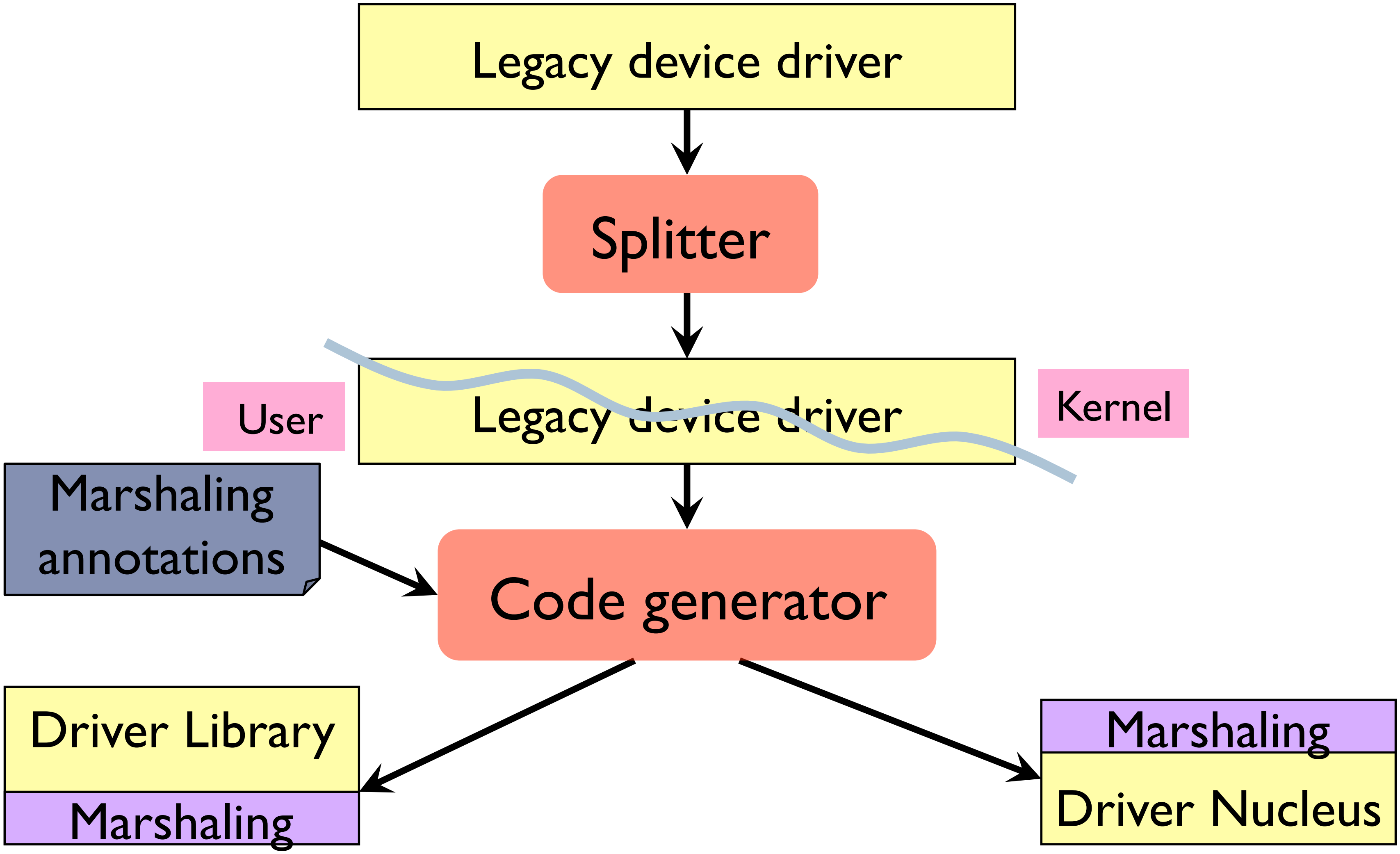
Goal: separate critical code from the rest

1. Low latency requirements
2. High bandwidth requirements
3. High priority requirements

Solution: leverage standard driver interfaces

1. Identify critical root functions for a driver from driver interface definition
2. Expand transitively through call graph
3. Identify all entry point functions where control passes between the driver library and nucleus

Splitting drivers



Generating marshaling code

- Goal: generate code for entry point functions to pass data structures between kernel and user
- Problems:
  - Kernel structures are highly linked
  - Types defined incompletely in C

Marshaling linked structures

- Solution: only copy fields actually accessed
  - Identify which fields are accessed from each entry point
  - Generate unique code for each entry point

Field analysis example

net_device before:

```
struct net_device
{
    char                              name[IFNAMSIZ];
    struct hlist_node       name_hlist;
    unsigned long                     mem_end; /* shared mem end
    */
    unsigned long                    mem_start;        /* shared mem start
    */
    unsigned long                     base_addr;       /* device I/O
address             */
    unsigned int            irq;      /* device IRQ number
    */
    unsigned char                     if_port; /* Selectable AUI, TP,..*/
    unsigned char                     dma;     /* DMA channel
    */
    unsigned long                     state;
    struct net_device                 *next;
    int                     (*init)(struct net_device *dev);
    unsigned long                     features;
    struct net_device                 *next_sched;
    int                     ifindex;
    int                     iflink;
    struct net_device_stats*          (*get_stats)(struct net_device
*dev);
    struct iw_statistics* (*get_wireless_stats)(struct net_device
*dev);
    const struct iw_handler_def * wireless_handlers;
    struct ethtool_ops      *ethtool_ops;
    unsigned short                    flags;   /* interface flags (a la
BSD)  */
    unsigned short                    gflags;
      unsigned short                  priv_flags; /* Like 'flags' but
invisible to userspace. */
    unsigned short                    padded;  /* How much padding added
by alloc_netdev() */
    unsigned                mtu;      /* interface MTU value
    */
    unsigned short                    type;    /* interface hardware
type */
    unsigned short                    hard_header_len; /* hardware hdr
length*/
    struct net_device                 *master;
    unsigned char                     perm_addr[MAX_ADDR_LEN]; /*
permanent hw address */
    unsigned char                     addr_len;        /* hardware
address length*/
    unsigned short                     dev_id;         /* for shared
network cards */
    struct dev_mc_list      *mc_list;         /* Multicast mac
addresses*/
    int                     mc_count;         /* Number of installed
mcasts*/
    int                     promiscuity;
    int                     allmulti;
      void                  *atalk_ptr;       /* AppleTalk link
    */
    void                    *ip_ptr; /* IPv4 specific data
    */
    void                              *dn_ptr;         /* DECnet
specific data */
    void                              *ip6_ptr;        /* IPv6 specific
data */
    void                    *ec_ptr; /* Econet specific data
    */
    void                    *ax25_ptr;        /* AX.25 specific data */
    struct list_head        poll_list ____cacheline_aligned_in_smp;
    int                     (*poll) (struct net_device *dev, int
*quota);
    int                     quota;
```

net_device after:

```
struct net_device
{
  char name[IFNAMSIZ];
  void *priv;
  unsigned long features;
  unsigned long trans_start;
}
```

Marshaling incomplete types

Extend C with 7 marshaling annotations:

- `Nullterm`
- `Array`
- `Combolock`
- `Opaque`
- `Sentinel`
- `Storefield`
- `Container`

Guides programmers in placing annotations

Annotation example

```
struct pcnet32 private {
  const char * name;
  int rx_ring_size;
  struct pcnet32_rx_head * rx_ring;
  spinlock_t lock; ...
}
```

Problem Pointers

Problem lock

Annotation example

```
struct pcnet32 private {
  const char * Nullterm name;
  int rx_ring_size;
  struct pcnet32_rx_head *
            Array(rx_ring_size) rx_ring;
  spinlock_t Combolock lock; ...
}
```

Java access to kernel data and functions

- Problem
  - Different type systems
  - No easy conversion
- Leverage: RPC systems solve this problem
- Phase one: DriverSlicer emits XDR
  - Extracts all data structure definitions and typedefs
  - Converts these definitions to an XDR specification
- Phase two: **`rpcgen`** and **`jrpcgen`** generate code
  - Create Java classes with public fields
  - Support features like recursive data structures

Phase 1: Example

From e1000.h

```
struct e1000_adapter { …
  struct e1000_rx_ring test_rx_ring;
  uint32_t * __attribute__((exp(PCI_LEN))) config_space;
  int msg_enable;
… };
```

Original C code

```
typedef unsigned int uint32_t;

struct uint32_256_t {
   uint32_t array_256[256];
};

typedef struct uint32_t_256 *uint32_t_256_ptr;

struct e1000_adapter { …
  struct e1000_rx_ring test_rx_ring;
  uint32_t_256_ptr config_space;
  int msg_enable;
… };
```

Automatically-generated XDR Definition

Phase 2: Continued

```
typedef unsigned int uint32_t;
struct uint32_256_t {
   uint32_t array_256[256];
};
typedef struct uint32_t_256 *uint32_t_256_ptr;
struct e1000_adapter { …
  struct e1000_rx_ring test_rx_ring;
  uint32_t_256_ptr config_space;
  int msg_enable;
… };
```

Automatically-generated XDR Definition

```
public class e1000_adapter … { …
  public e1000_rx_ring test_rx_ring;
  public uint32_t_256_ptr config_space;
  public int msg_enable;
…
  public e1000_adapter () { … }
  public e1000_adapter(XdrDecStream xdr) { … }
  public void xdrEncode(XdrEncStream xdr) { … }
  public void xdrDecode(XdrDecStream xdr) { … }
}
```

Automatically-generated Java

DriverSlicer Summary

- Splitter
  - Identifies kernel code from **critical root functions**
  - Identifies driver library/nucleus entry points
- Marshaler
  - Generates code to marshal/unmarshal structures
  - Identifies which fields are accessed in user mode
- Java Conversion
  - C → XDR conversion
  - XDR code generation

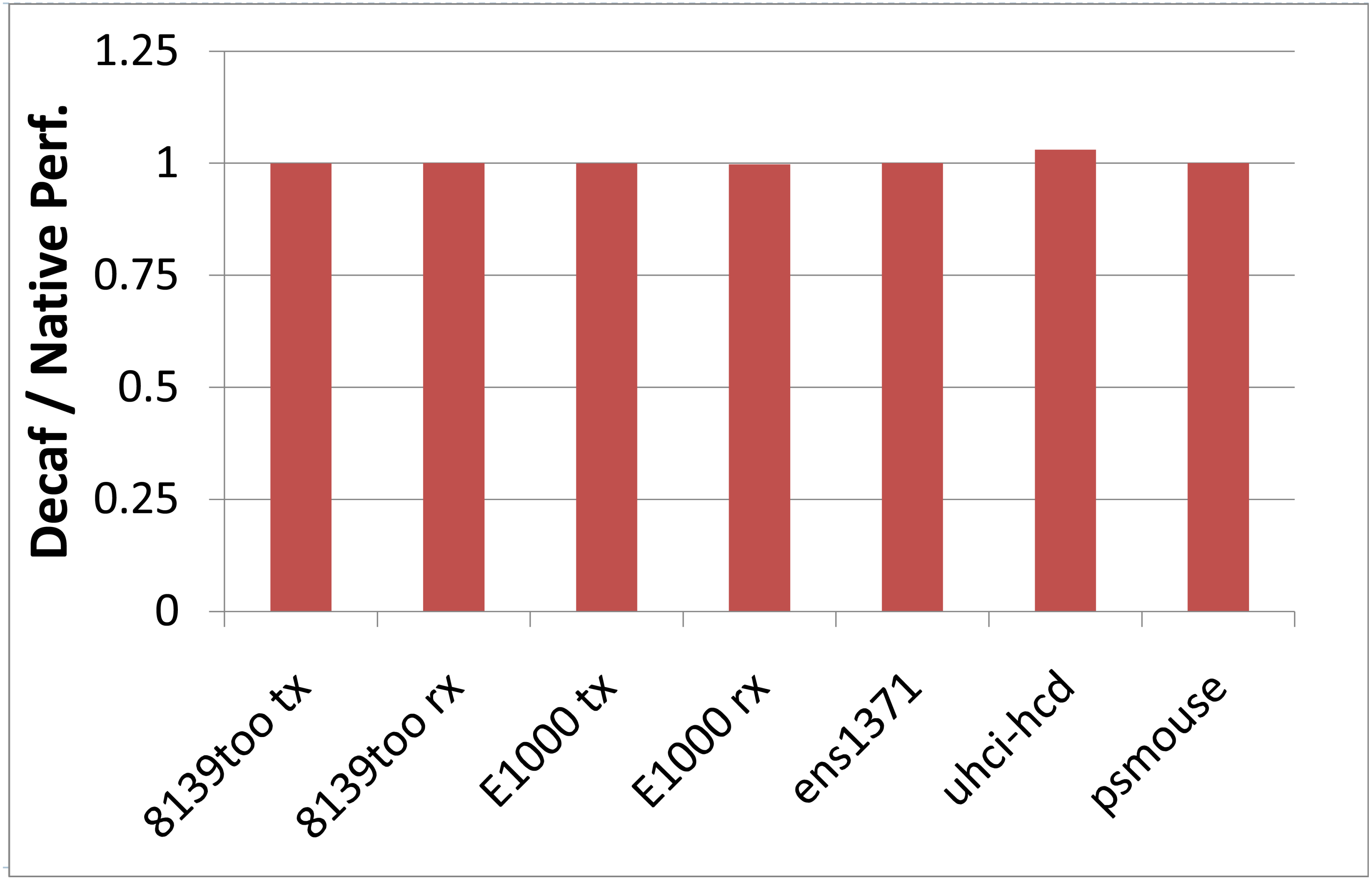
Outline

- Introduction
- Overview
- Design and Implementation
- Evaluation
  - Conversion effort
  - Performance analysis
  - Benefits of Decaf Drivers
    - Case study of E1000 gigabit network driver
- Conclusion

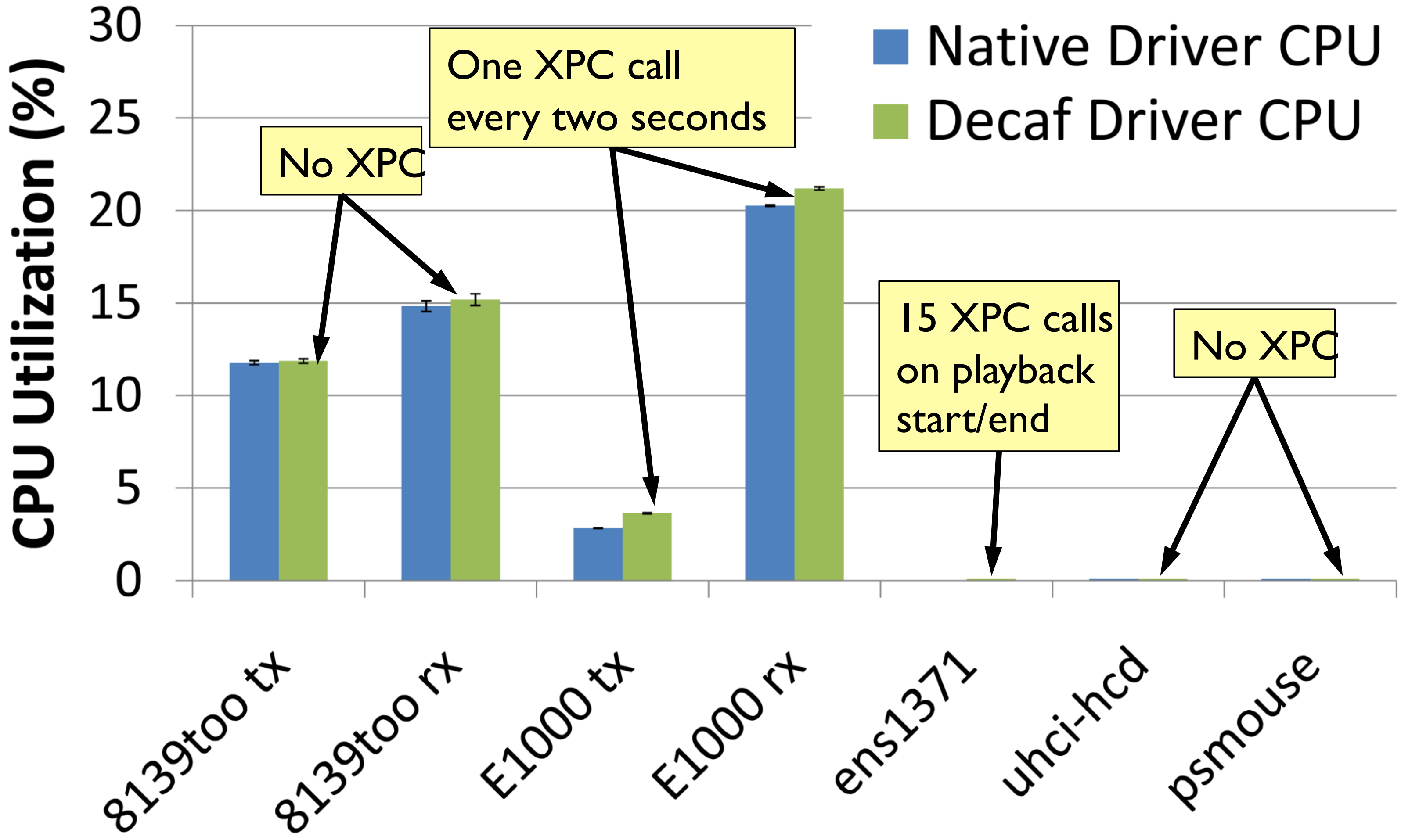
Conversion Effort

Driver	Original Lines of Code	Anno-tations	Functions		
			Driver Nucleus	Decaf Driver	Driver Library
e1000	14,204	64	46	236	0
8139too	1,916	17	12	25	16
ens1371	2,165	18	6	59	0
psmouse	2,448	17	15	14	74
uhci-hcd	2,339	94	68	3	12

Results: Relative Performance



Results: CPU Utilization



E1000: Core 2 Quad 2.4Ghz, 4GB RAM

All others: Pentium D 3.0Ghz, 1GB RAM

Experience Rewriting Drivers

- Step one: initial conversion
  - Largely mechanical: syntax is similar
  - Leaf functions first, then remainder
- Step two: use Java language features
  - Example benefit: E1000 exception handling

Java Error Handling

Original C, e1000_hw.c

```
if(hw->ffe_config_state == e1000_ffe_config_active) {
  ret_val = e1000_read_phy_reg(hw, 0x2F5B,
                &phy_saved_data);
  if(ret_val) return ret_val;

  ret_val = e1000_write_phy_reg(hw, 0x2F5B, 0x0003);
  if(ret_val) return ret_val;

  msec_delay_irq(20);
  ret_val = e1000_write_phy_reg(hw, 0x0000,
                IGP01E1000_IEEE_FORCE_GIGA);
  if(ret_val) return ret_val;
```

- Easy to miss an error condition

Java Error Handling

Java, e1000_hw.java

```
if(hw.ffe_config_state.value == e1000_ffe_config_active) {
  e1000_read_phy_reg(0x2F5B, phy_saved_data);
  e1000_write_phy_reg((short) 0x2F5B, (short) 0x0003);
  e1000_write_phy_reg((short) 0x2F5B, (short) 0x0003);
  DriverWrappers.Java_msleep (20);
  e1000_write_phy_reg((short) 0x0000,
    (short) IGP01E1000_IEEE_FORCE_GIGA);
```

- Uncovered at least 28 cases of ignored error conditions
- Resulting code 8% shorter *overall*

Decaf Summary

- Decaf Drivers simplify driver programming
  - Provide a migration path from C to Java
  - Allow driver code to run in user mode
  - Support continued driver and kernel evolution
  - Offer excellent performance

Conclusions

- Large-scale changes to driver architecture or development practices take time
- Drivers can be improved through program-analysis tools and a small amount of runtime code
  - Carburizer: detect/fix hardware dependency bugs
  - Decaf drivers: migrate code to Java
- Runtime costs can be low, by focusing on uncommon cases

Questions?

Reliability

- Kernel code is brittle

- 89% of XP crashes are from drivers
- Linux drivers had 7x bug density of other kernel code

Driver Evolution

- Example: E1000 network driver 2.6.18.1 to 2.6.27
  - e1000_adapter structure needs additional members
- Existing marshaling code does not transfer new fields automatically
- Solution: the driver is the specification
  1) Add new member definitions to original e1000.h
  2) Re-run DriverSlicer
  3) Use variables in Driver Nucleus or Decaf Driver

What has changed?

- Driver Classes in Unix, 1973
  1. Character
  2. Disk
  3. Tape
  4. Printer
  5. TTY

- Driver/Device Classes in Windows Vista/7
  1. 1394 device
  2. Auxiliary display (SideShow)
  3. Bluetooth L2CAP
  4. Bluetooth Radio Frequency Communications (RFCOMM)
  5. Cell phone, PDA, portable media player*
  6. Digital camera
  7. Display adapter
  8. Human input device (HID)
  9. Keyboard/Mouse filter
  10. Modem, cable modem
  11. Network Transport Driver Interface (TDI) client
  12. Network-connected device*
  13. Printer
  14. Scanner
  15. Secure digital (SD)
  16. Serial and parallel devices (legacy)
  17. Smart card device
  18. USB device
  19. Video capture