The Design and Implementation of Microdrivers

Vinod Ganapathy*, Matthew Renzelmann[+], Arini Balakrishnan[^], Michael Swift[+], Somesh Jha[+]

**Rutgers University,*
[+]University of Wisconsin-Madison,
[^]Sun Microsystems

Drivers 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 2.6.23, 2007
  - 3199 driver variations
  - 134 MB of driver code
  - 3 million lines of code
  - Written by > 312 people

Drivers are unreliable!

- Writing drivers is hard
  - Must handle asynchronous events
  - Must obey kernel programming rules
  - Many drivers written by non-kernel experts
- Debugging drivers is hard
  - Non-reproducible failures
  - Fewer advanced development tools

Existing solutions are not enough

- Driver isolation systems
  - Nooks [Swift, SOSP '03]
  - SafeDrive [Zhou, OSDI '06]
- User-level drivers
  - Minix 3 [Herder, DSN '07]
  - Windows UMDF
  - Linux User-Level Device Drivers [Leslie, JCST '05]

Microdriver Architecture

- Splits drivers into:
  - A *k-driver* containing performance-sensitive code
  - A *u-driver* containing everything else
- Simplifies driver programming by moving much of it to *user mode*.
- Improves reliability by *reducing kernel code size*.
- Maintains high performance.
- Can be written manually, or generated almost *automatically from existing drivers*.
- Is *compatible* with existing operating systems.

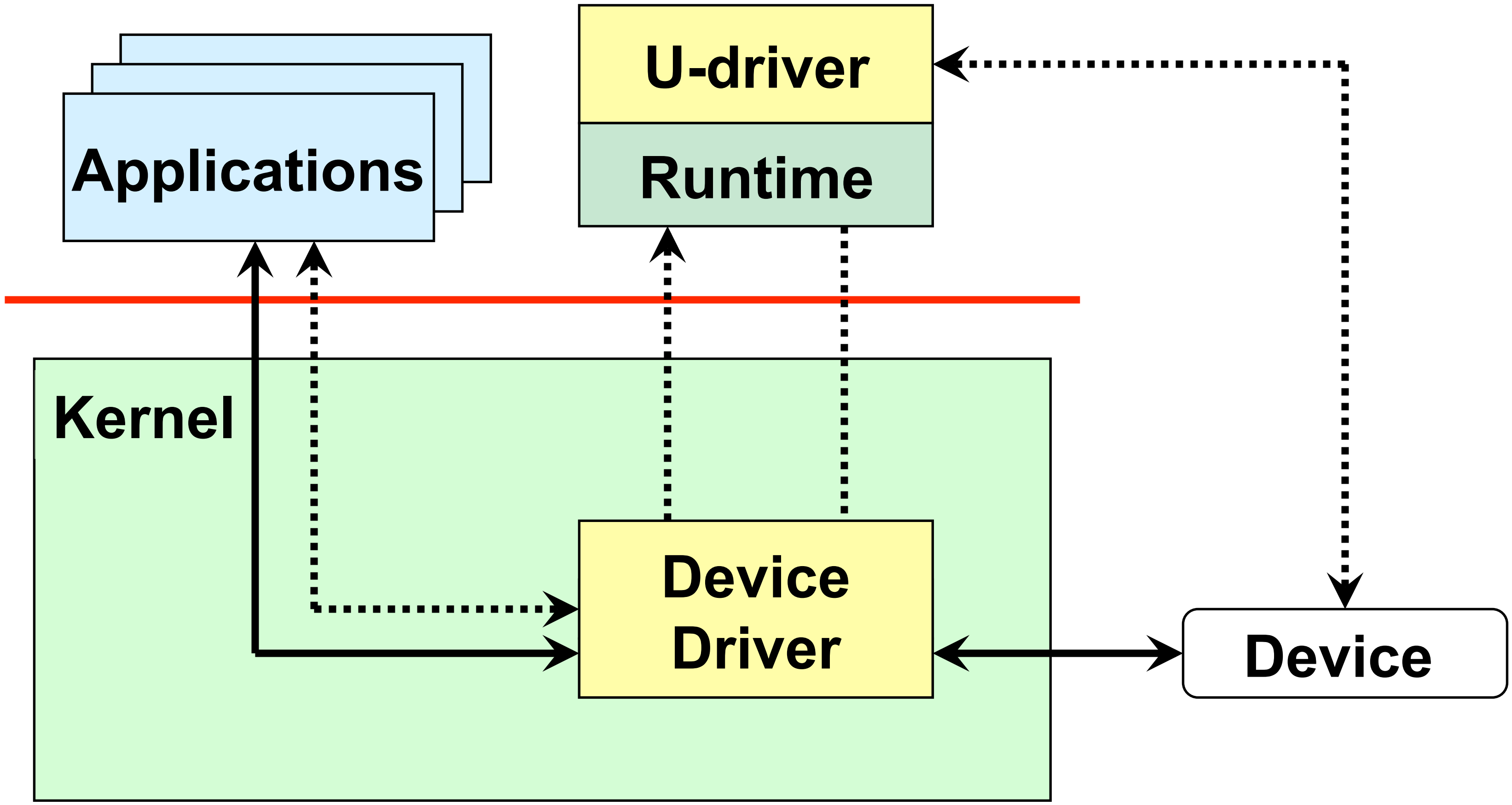
Outline

- Introduction
- Architecture
- DriverSlicer
- Evaluation
- Conclusions

Intuition

- For compatibility and performance, some driver tasks should remain in the kernel.
- Many driver tasks need not
  - Initialization/shutdown
  - Configuration
  - Error handling

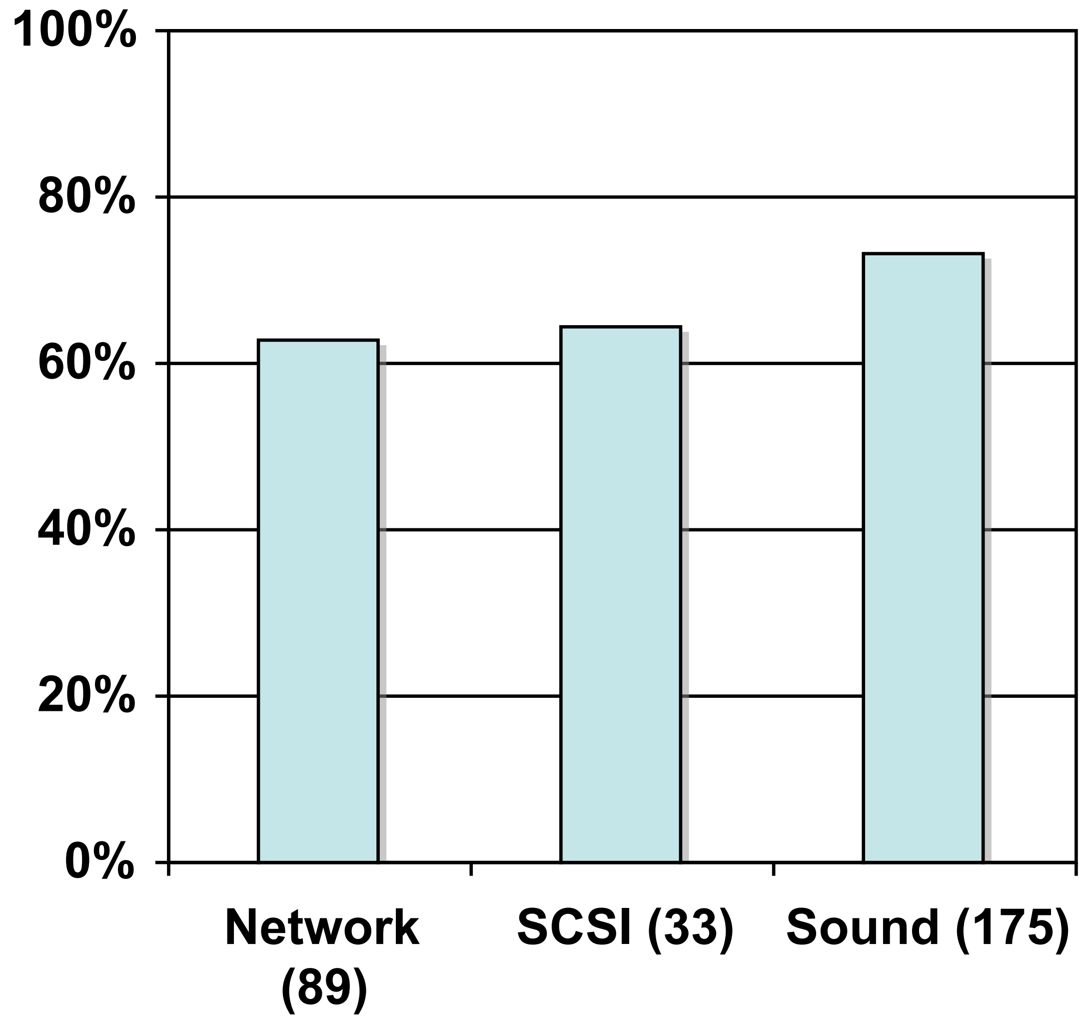
Microdrivers



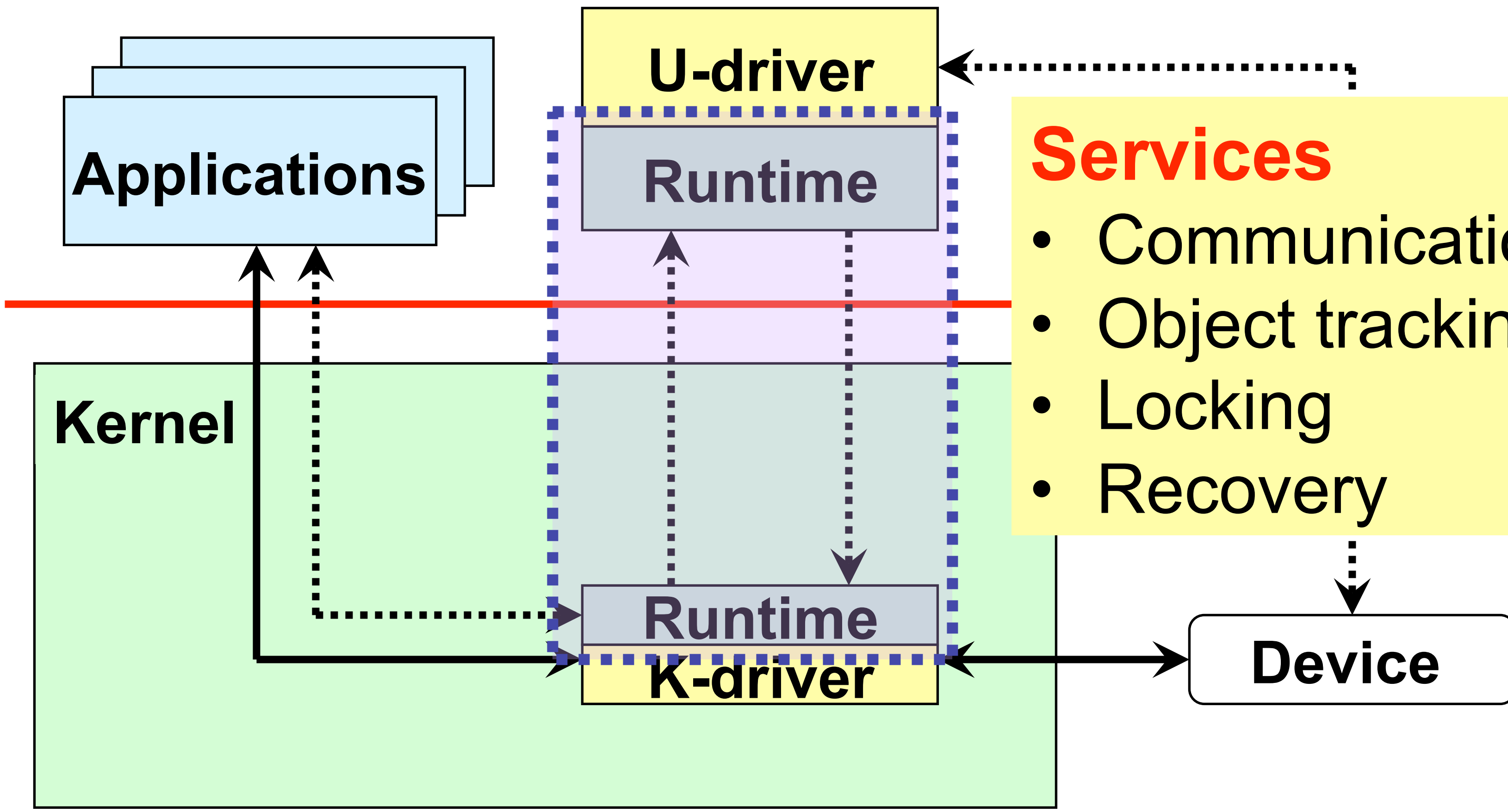
Microdriver Potential

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

Up to 1.8 million lines of code



Runtime services

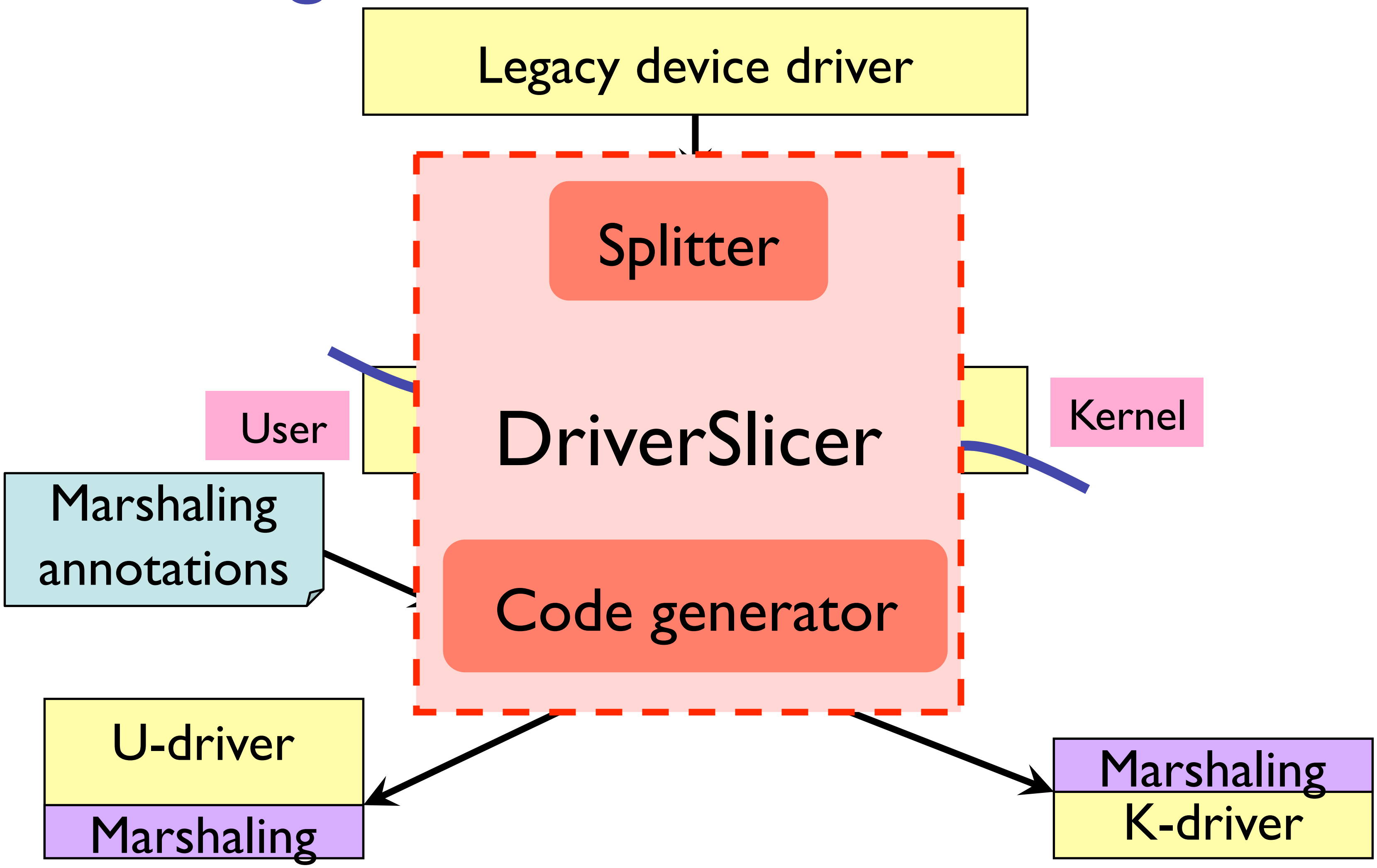


Services
- Communication
- Object tracking
- Locking
- Recovery

Outline

- Introduction
- Architecture
- DriverSlicer
- Evaluation
- Conclusions

Generating a microdriver



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 U- and K-driver

Generating marshaling code

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

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

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;
      int                    weight;
      unsigned long                      last_rx;    /* Time of last Rx      */
      unsigned char                      dev_addr[MAX_ADDR_LEN];
      spinlock_t                         queue_lock ____cacheline_aligned_in_smp;

      … 38 more fields …
}
```

After:

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

Experimental Results

Bytes transferred during 8139cp network driver initialization

- Without optimization: 2,931,212
- With optimization: 1,729

DriverSlicer Summary

- Splitter
  - Identifies kernel code from *critical root functions*
  - Identifies u/k-driver entry points
- Marshaler
  - Generates code to marshal/unmarshal structures
  - Identifies which fields are accessed in user mode

Outline

- Introduction
- Architecture
- DriverSlicer
- Evaluation
  - Moving Code
  - Performance
- Conclusions

Experience

- Implemented in unmodified Linux 2.6.18.1 kernel:
  - Kernel runtime: 4,951 lines of code
  - User runtime: 1,959 lines of code
  - DriverSlicer: 9,827 lines of OCaml in CIL [Necula et al. '02]
- Tested on 7 drivers:
  - Network: forcedeth, 8139cp, 8139too, pcnet32, ne2000
  - Sound: ens1371
  - USB: uhci-hcd
- Simplified debugging of u-drivers
  - Standard tools (gdb, valgrind) apply

Annotation Difficulty

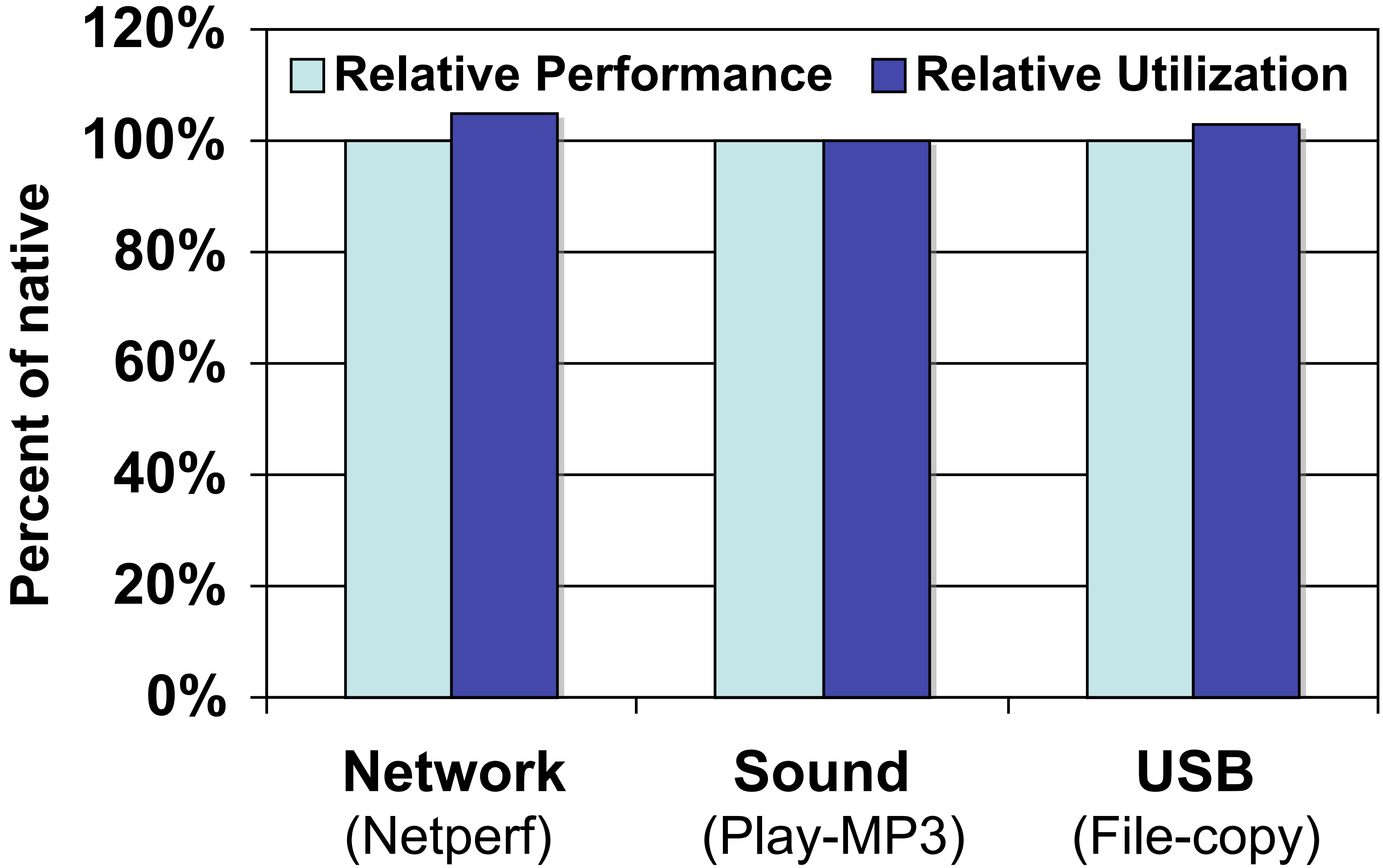
Driver	Code Size	Driver Annot.	
Network	3460	12	
Sound	1494	7	
USB	2517	146	

Code Motion

Driver	Code Size	Driver Annot.	User Code %
Network	3460	12	65
Sound	1494	7	46
USB	2517	146	19

Fraction of code changes (from BitKeeper) similar to fraction of code.

Performance



Conclusion

- Microdrivers:
  - Reduce the amount of code in the kernel
  - Permit the use of user-mode tools for driver development
  - Are compatible with commodity operating systems
  - Can be generated largely automatically from existing drivers
  - Have good common-case performance

Questions?

For more information:

swift@cs.wisc.edu
or visit

www.cs.wisc.edu/~swift/drivers

Additional Code

Driver	Marshaling Code Size
8139too	14,700
forcedeth	37,900
ens1371	6,100
uhci_hcd	12,000

Future Work

- Measure improve reliability from moving code to user
- Identify kernel changes to enable more code motion
- Generate user-editable driver code

- Convert user-level driver code to Java or Python
- Generate kernel driver code from a domain specific language

Recovery

- Detect and recover from failed u-driver
  - Ideally transparent to applications
- Detection done at interface
  - Parameter checks and timeouts
- Recovery – compatible with prior work
  - Shadow driver mechanism [Swift *et al.*, 2004]
  - SafeDrive recovery mechanism [Zhou *et al.*, 2006]

Splitting Example

pcnet32.c

```
irqreturn_t pcnet32_int(int,
   void *, struct pt_regs *) {
  …
  pcnet32_rx(dev)
  …
}
int pcnet32_start_xmit(struct
   sk_buff*, struct net_device*){
  …
   p->read_csr(ioaddr, 80);
  netif_stop_queue(dev);
  …
}
```

Latency roots:

- Interrupts
- Softirqs
- Timers

Bandwidth roots:

- Packet send

Priority roots:

- `set_mcast_list`

Marshaling Incomplete Types

Extend C with 7 marshaling annotations:

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

Guide 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; ...
}
```

Field Access Analysis Algorithm

- Given function F, field accesses are:
  - For each type of structure accessed in F, the fields accessed for that type
  - The field accesses for F's callees
- Complications
  - Void * fields
  - Indirect calls

Locking

- Problem: shared data structures require mutual exclusion
  - Spinlocks not safe outside kernel
  - Semaphores not safe at high priority
- Solution: ComboLocks
  - Spins when all requesters are in kernel
  - Devolves to semaphores when acquired from user level

ComboLocks

```
struct combolock {
   spinlock slock;
   semaphore sem;
   int sem_required;
};
```

- Kernel spinlock protects driver data and `sem_required`

Kernel:

```
cl_lock(combolock l) {
  lock (l.slock);
  if (l.sem_required != 0) {
    l.sem_required++;
    unlock (l.slock);
    sem_acquire(l.sem);
  }
}
```

ComboLocks from user-level

```
struct combolock {
   spinlock slock;
   semaphore sem;
   int sem_required;
};
```

User:

```
cl_lock(combolock l) {
  lock(l.slock);
  l.sem_required++;
  unlock (l.slock);
  sem_acquire (l.sem);
}
```

- Call into kernel to acquire lock
- Synchronize objects on lock/release