

# NDL: A Domain-Specific Language for Device Drivers

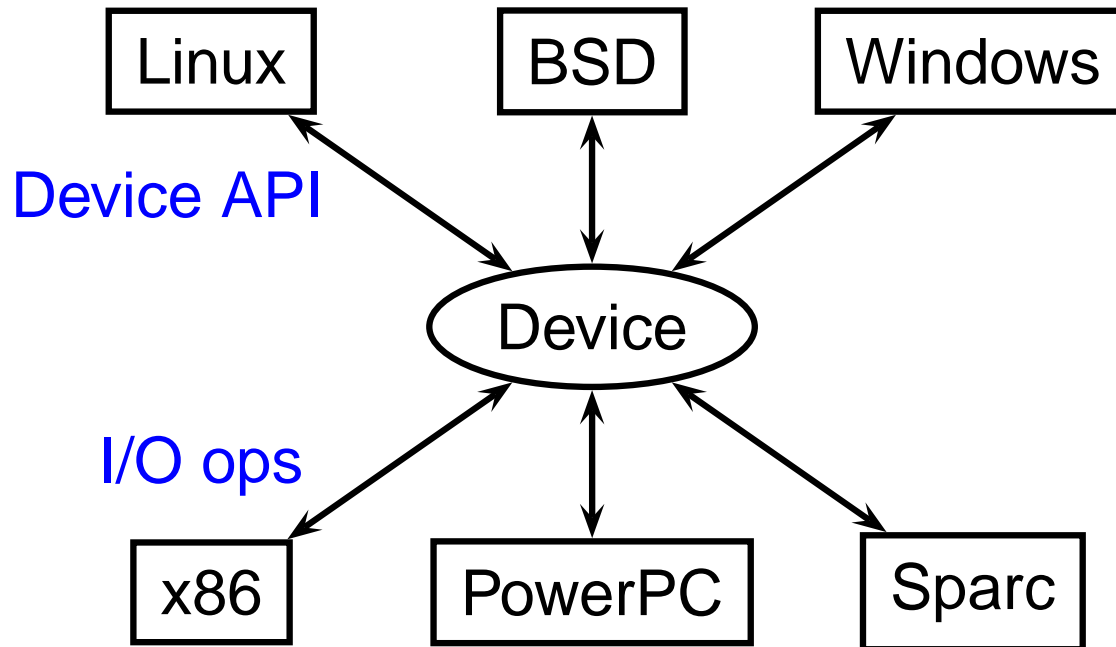
Christopher Conway & Stephen Edwards  
Columbia University

LCTES '04  
11 June 2004

# What's the Problem?

- Drivers are difficult to write and error prone.
- They are typically written in C: low-level, type unsafe.
- (Chou, et al. 2001): Drivers account for 70-90% of bugs in the Linux kernel and have an error rate 7x that of the rest of the kernel.
- Run in kernel mode. A driver bug can crash the entire system.

# Portability



- Driver APIs can be quite dissimilar.
- UDI (Uniform Device Interface) - common standard API, poorly supported.

# Related Work

- Static analysis and model checking: [SPIN: Holzmann 1997], [SLAM: Ball & Rajamani 2002]
- Domain-specific languages
  - GAL: language for X Windows video drivers, code 90% smaller than C. [Thibault, Marlet & Consel 1999]
  - Devil: an IDL for device drivers, translates into C macros. [Mérillon, et al. 2000]
  - Iris: a DSL for device drivers, FSM modelling, hardware/software co-design. [Wang, Malik & Bergamaschi 2003]

# Making Drivers Easier (And Better)

- A language that is easier to write and maintain.
- Add type knowledge to I/O operations.
- Provide tools for debugging and testing.
- Test case: NE2000 network card driver.

# NDL = “The NDL Device Language”

Typical NDL code (NE2000 network card driver):

```
start = true;  
dmaState = DISABLED;  
remoteDmaByteCount = count;
```

The equivalent C:

```
outb(E8390_NODMA + E8390_PAGE0 + E8390_START,  
     nic_base + NE_CMD);  
outb(count & 0xff, nic_base + EN0_RCNTLO);  
outb(count >> 8, nic_base + EN0_RCNTHI);
```

# NDL Syntax

```
remoteByteCount = count ;
remoteStartAddr = start_page*FRAME_LEN ;

trans DMA_WRITING ;

dataport =<16> buffer ;

wait 20ms for remoteDmaIrq else {
    print("ne2k: Timeout waiting for Tx RDC.") ;
    soft_reset() ;
    start_dev() ;
}

remoteDmaIrq=ACK ;
```

# Device Registers

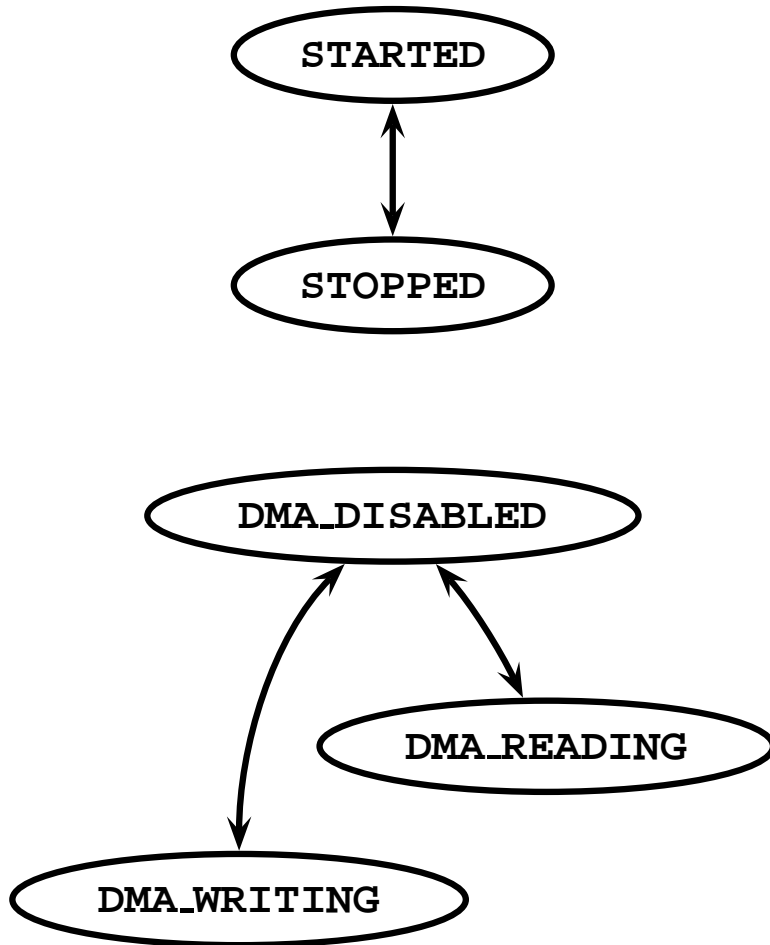
The device interface is usually a block of memory-mapped I/O locations with complex semantics.

```

ioports {
  command = {
    0: stop: trigger except 0,
    1: start: trigger except 0,
    2: transmit: trigger except 0,
    3..5:
    dmaState : {
      READING = #001
      WRITING = #010
      SENDING = #011
      DISABLED = #1**
    } volatile,
    6..7: registerPage: int{0..2}
  },
  0x01..0x0f: [
    /* predicated regs. */
    ( PAGE(0) ) => {
      write rxStartAddr,
      write rxStopAddr,
      boundaryPtr,
      [ /* overlay reg. */
        read txStatus = {
          0: packetTransmitted,
          1: _,
          2: transmitCollided,
          3: transmitAborted,
          4: carrierLost,
          5: fifoUnderrun,
          6: heartbeatLost,
          7: lateCollision
        } volatile
      ]
    }
    ||
    write txStartAddr
  ],
  /* eleven bytes elided */
}
|| /* predicated regs. */
( PAGE(1) ) => {
  physicalAddr : byte[6],
  currentPage : byte,
  multicastAddr : byte[8]
}
|| /* predicated regs. */
( PAGE(2) ) => {
  _ : byte[13],
  read dataConfig,
  read interruptMask
}
],
0x10: dataport : fifo[1] trigger,
  _ : byte[14],
0x1f: reset : byte trigger
}

```

# State Machines



```
state STOPPED {
    trans DMA_DISABLED;
    stop = true;
}
||
STARTED { start = true; }

state DMA_DISABLED {
    dmaState = DISABLED;
}
||
DMA_READING {
    trans STARTED;
    dmaState = READING;
}
||
DMA_WRITING {
    trans STARTED;
    dmaState = WRITING;
}
```

```

ioports {
  command = {
    0: stop: trigger except 0,
    1: start: trigger except 0,
    2: transmit: trigger except 0,
    3..5:
    dmaState : {
      READING = #001
      WRITING = #010
      SENDING = #011
      DISABLED = #1**
    } volatile,
    6..7: registerPage: int{0..2}
  },
  0x01..0x0f: [
    /* predicated regs. */
    ( PAGE(0) ) => {
      write rxStartAddr,
      write rxStopAddr,
      boundaryPtr,
      [ /* overlay reg. */
        read txStatus = {
          0: packetTransmitted,
          1: _,
          2: transmitCollided,
          3: transmitAborted,
          4: carrierLost,

```

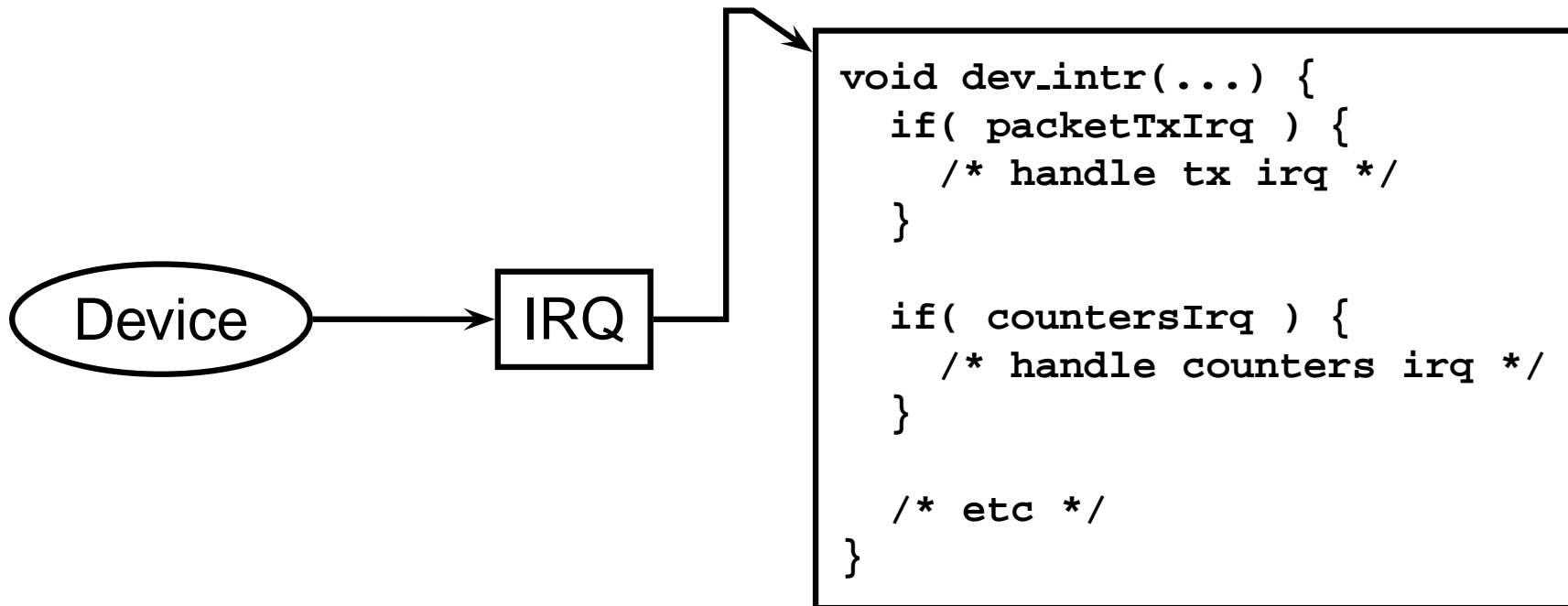
```

          5: fifoUnderrun,
          6: heartbeatLost,
          7: lateCollision
        } volatile
      ]
    ],
    /* eleven bytes elided */
  }
  || /* predicated regs. */
  ( PAGE(1) ) => {
    physicalAddr : byte[6],
    currentPage : byte,
    multicastAddr : byte[8]
  }
  || /* predicated regs. */
  ( PAGE(2) ) => {
    _ : byte[13],
    read dataConfig,
    read interruptMask
  }
  ],
  0x10: dataport : fifo[1] trigger,
    _ : byte[14],
  0x1f: reset : byte trigger
}

```

# Interrupts

Much of the interesting behavior of a device is driven by interrupt requests.



# Interrupt Handlers

Functions can be tagged with interrupt conditions they handle.

```
function tx_intr @(packetTxIrq) {  
    /* handle tx interrupt ... */  
}
```

```
/* anonymous interrupt handler */  
function @(countersIrq) {  
    rxFrameErrors += frameAlignErrors;  
    rxCrcErrors += crcErrors;  
    rxMissedErrors += packetErrors;  
    countersIrq = ACK;  
}
```

# Interrupt Handlers: cont'd

The compiler generates a top-level handler:

```
void dev_intr(...) {  
    if( packetTxIrq ) {}  
    if( countersIrq ) {}  
    /* etc */  
}
```

```
function tx_intr @(packetTxIrq) {  
    /* handle tx interrupt ... */  
}
```

```
function @(countersIrq) {  
    rxFrameErrors += frameAlignErrors;  
    rxCrcErrors += crcErrors;  
    rxMissedErrors += packetErrors;  
    countersIrq = ACK;  
}
```

# Debug Mode

Prints trace of I/O ops and dumps state at function boundaries.

```
/var/log/messages:  
Jun  9 20:04:31 prosser: ne2k: outb(0x42,base+0)  
Jun  9 20:04:31 prosser: ne2k: outb(0x49,base+14)  
Jun  9 20:04:31 prosser: ne2k: outb(0x00,base+10)  
Jun  9 20:04:31 prosser: ne2k: outb(0x00,base+11)  
Jun  9 20:04:31 prosser: ne2k: outb(0x00,base+15)  
Jun  9 20:04:31 prosser: ne2k: outb(0x7f,base+7)  
Jun  9 20:04:31 prosser: ne2k: outb(0x20,base+12)  
Jun  9 20:04:31 prosser: ne2k: outb(0x02,base+13)  
Jun  9 20:04:31 prosser: ne2k: outb(0x20,base+10)  
Jun  9 20:04:31 prosser: ne2k: outb(0x00,base+11)  
Jun  9 20:04:31 prosser: ne2k: outb(0x00,base+8)  
Jun  9 20:04:31 prosser: ne2k: outb(0x00,base+9)  
Jun  9 20:04:31 prosser: ne2k: outb(0x0a,base+0)  
Jun  9 20:04:31 prosser: ne2k: insb(base+0,dev_addr,0x06)  
Jun  9 20:04:31 prosser: ne2k: init(): crcErrors=0x00  
multicastAddr=0xff000000 boundaryPtr=0x7f packetErrors=0x00  
rxStatus=0x00 frameAlignErrors=0x00 interruptMask=0x7f  
physicalAddr=0x09ce95ba5000 txStatus=0x00 intStatus=0x00  
currentPage=0x4c dataConfig=0x49
```

# The NDL Compiler

- Written in Standard ML.
- Three-address IR with control flow.
- Special instructions from read and writing device registers:

```
LOAD n, register[a:b]
```

```
STORE register[a:b], n
```

- Code generation is syntax-directed translation into C.

# Optimization

- Performance is not our main focus.
- Goal: minimize performance advantage of C
- The key performance metric is I/O operations (minimize register read/writes)

Note: register access is slow.

- Leverage domain knowledge + rich semantic information
- Leave non-domain sensitive optimization to the C compiler

# Code Generation

The IR code is highly redundant. State predicates get expanded inline.

```
fifoThreshold=FILLED_8;
```

```
autoInitRemote=false;
```

```
loopbackDisabled=true;
```

```
dmaAddrMode=DUAL_16;
```

```
byteOrder=LITTLE_ENDIAN;
```

```
dmaTransferWidth=WORD_WIDE;
```

```
STORE command[6:7], 0
```

```
STORE dataConfig[5:6], 2
```

```
STORE command[6:7], 0
```

```
STORE dataConfig[4:4], 0
```

```
STORE command[6:7], 0
```

```
STORE dataConfig[3:3], 1
```

```
STORE command[6:7], 0
```

```
STORE dataConfig[2:2], 0
```

```
STORE command[6:7], 0
```

```
STORE dataConfig[1:1], 0
```

```
STORE command[6:7], 0
```

```
STORE dataConfig[0:0], 1
```

# Idempotence

Repeated writes to a state-holding register can be pruned.

```
STORE command[6:7], 0
STORE dataConfig[5:6], 2
STORE command[6:7], 0
STORE dataConfig[4:4], 0
STORE command[6:7], 0
STORE dataConfig[3:3], 1
STORE command[6:7], 0
STORE dataConfig[2:2], 0
STORE command[6:7], 0
STORE dataConfig[1:1], 0
STORE command[6:7], 0
STORE dataConfig[0:0], 1
```

```
STORE command[6:7], 0
STORE dataConfig[5:6], 2
STORE dataConfig[4:4], 0
STORE dataConfig[3:3], 1
STORE dataConfig[2:2], 0
STORE dataConfig[1:1], 0
STORE dataConfig[0:0], 1
```

# Field Aggregation

Orthogonal writes to sub-registers can be consolidated.

```
STORE command[6:7], 0
STORE dataConfig[5:6], 2

STORE dataConfig[4:4], 0

STORE dataConfig[3:3], 1

STORE dataConfig[2:2], 0

STORE dataConfig[1:1], 0

STORE dataConfig[0:0], 1
```

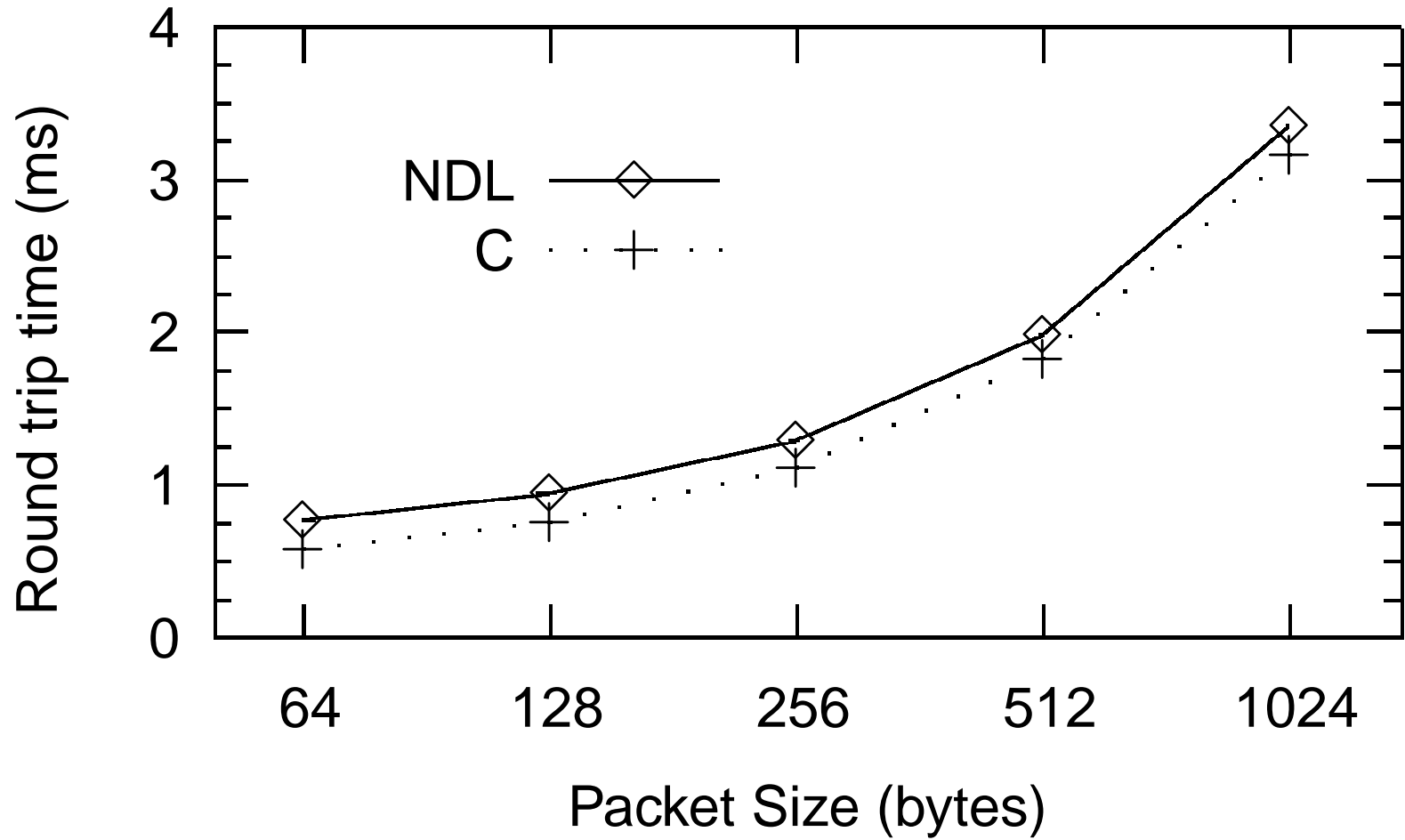
```
STORE command[6:7], 0
STORE dataConfig[0:6], 0x49
```

# Performance Tests

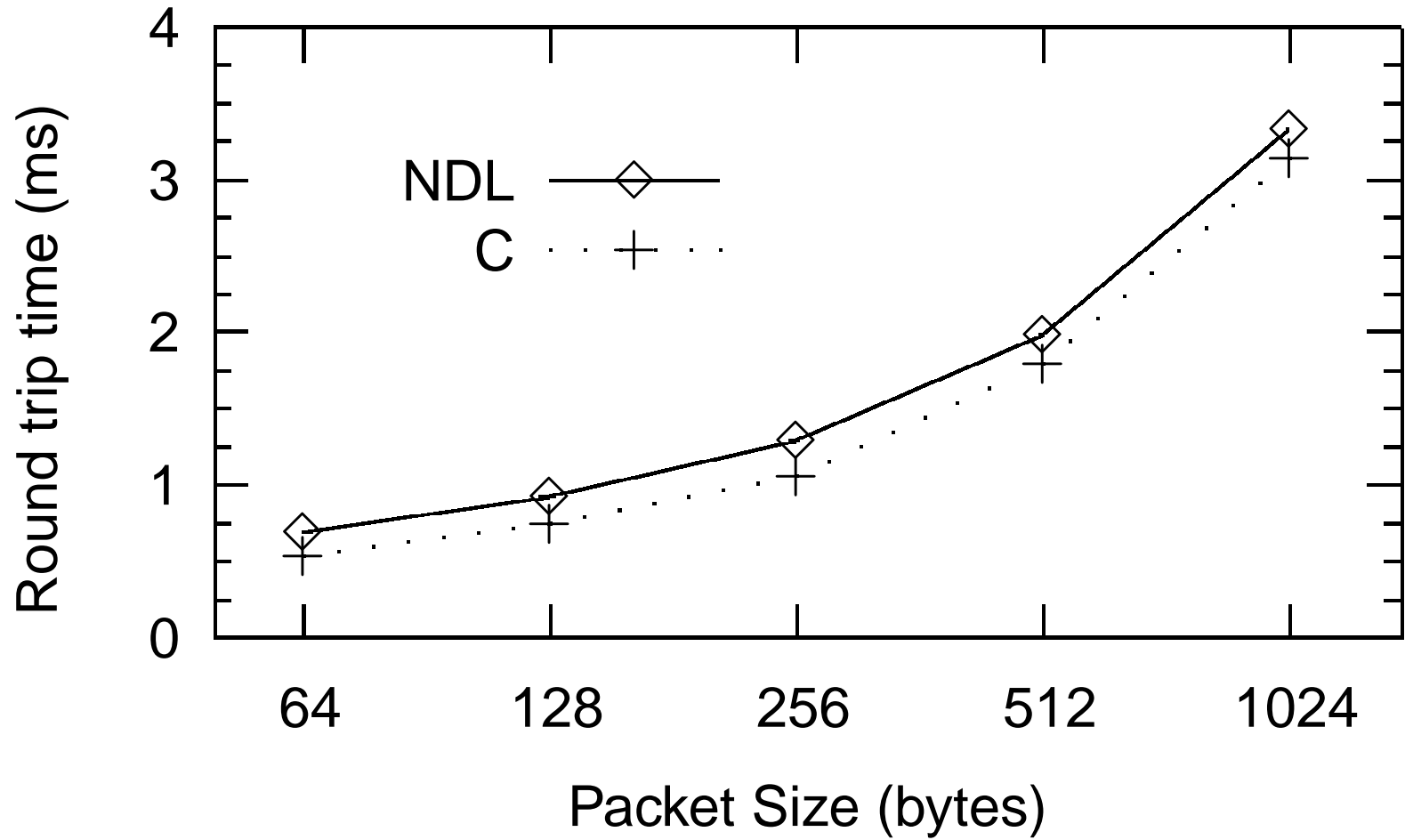
Flood the card with packets and measure latency.  
Stresses the critical path of most drivers: interrupt handling.

```
% ping -f -c 5000 prosser
PING localhost (127.0.0.1): 56 data bytes
.....
--- localhost ping statistics ---
5000 packets transmitted, 4094 packets received, 0.1% packet loss
round-trip min/avg/max = 0.513/0.774/1.581 ms
```

# Round trip time: Incoming



# Round trip time: Outgoing



# Results: Summary

Working drivers: NE2000 network card, `/dev/null`.

- Increase in executable size:  $> 25\%$
- Performance hit (vs. C): 4-35 %
- Reduction in lines of code:  $> 50\%$