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]
  - 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_RCNHTLO);
outb(count >> 8, nic_base + EN0_RCNTHI);
```
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 */
    },
    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.

```c
void dev_intr(...) {
    if (packetTxIrq) {
        /* handle tx irq */
    }
    if (countersIrq) {
        /* handle counters irq */
    }
    /* etc */
}
```
Interrupt Handlers

Functions can be tagged with interrupt conditions they handle.

```plaintext
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:

```c
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:
  
  \[
  \text{LOAD } n, \text{ register}[a:b] \\
  \text{STORE } \text{register}[a:b], n
  \]

- 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.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
<th>STORE command[6:7], 0</th>
<th>STORE dataConfig[5:6], 2</th>
<th>STORE command[6:7], 0</th>
<th>STORE dataConfig[4:4], 0</th>
<th>STORE command[6:7], 0</th>
<th>STORE dataConfig[3:3], 1</th>
<th>STORE command[6:7], 0</th>
<th>STORE dataConfig[2:2], 0</th>
<th>STORE command[6:7], 0</th>
<th>STORE dataConfig[1:1], 0</th>
<th>STORE command[6:7], 0</th>
<th>STORE dataConfig[0:0], 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>fifoThreshold</td>
<td>FILLED_8;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>autoInitRemote</td>
<td>false;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>loopbackDisabled</td>
<td>true;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dmaAddrMode</td>
<td>DUAL_16;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>byteOrder</td>
<td>LITTLE_ENDIAN;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dmaTransferWidth</td>
<td>WORD_WIDE;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Idempotence

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

<table>
<thead>
<tr>
<th>STORE command[6:7], 0</th>
<th>STORE command[6:7], 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>STORE dataConfig[5:6], 2</td>
<td>STORE dataConfig[5:6], 2</td>
</tr>
<tr>
<td>STORE command[6:7], 0</td>
<td>STORE command[6:7], 0</td>
</tr>
<tr>
<td>STORE dataConfig[4:4], 0</td>
<td>STORE dataConfig[4:4], 0</td>
</tr>
<tr>
<td>STORE command[6:7], 0</td>
<td>STORE command[6:7], 0</td>
</tr>
<tr>
<td>STORE dataConfig[3:3], 1</td>
<td>STORE dataConfig[3:3], 1</td>
</tr>
<tr>
<td>STORE command[6:7], 0</td>
<td>STORE command[6:7], 0</td>
</tr>
<tr>
<td>STORE dataConfig[2:2], 0</td>
<td>STORE dataConfig[2:2], 0</td>
</tr>
<tr>
<td>STORE command[6:7], 0</td>
<td>STORE command[6:7], 0</td>
</tr>
<tr>
<td>STORE dataConfig[1:1], 0</td>
<td>STORE dataConfig[1:1], 0</td>
</tr>
<tr>
<td>STORE command[6:7], 0</td>
<td>STORE command[6:7], 0</td>
</tr>
<tr>
<td>STORE dataConfig[0:0], 1</td>
<td>STORE dataConfig[0:0], 1</td>
</tr>
</tbody>
</table>
Field Aggregation

Orthogonal writes to sub-registers can be consolidated.

<table>
<thead>
<tr>
<th>STORE command[6:7], 0</th>
<th>STORE command[6:7], 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>STORE dataConfig[5:6], 2</td>
<td>STORE dataConfig[0:6], 0x49</td>
</tr>
<tr>
<td>STORE dataConfig[4:4], 0</td>
<td></td>
</tr>
<tr>
<td>STORE dataConfig[3:3], 1</td>
<td></td>
</tr>
<tr>
<td>STORE dataConfig[2:2], 0</td>
<td></td>
</tr>
<tr>
<td>STORE dataConfig[1:1], 0</td>
<td></td>
</tr>
<tr>
<td>STORE dataConfig[0:0], 1</td>
<td></td>
</tr>
</tbody>
</table>
Performance Tests

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

```bash
% 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

![Graph showing the relationship between packet size and round trip time. The graph includes two lines, one labeled NDL and the other labeled C, representing different data transfer speeds. Packet size (in bytes) increases from 64 to 1024 as the x-axis values. The y-axis represents round trip time (in ms) and varies from 0 to 4.]
Round trip time: Outgoing

![Graph showing round trip time vs. packet size](image)

- **NDL**
- **C**

Round trip time (ms) vs. Packet Size (bytes) for Outgoing packets.
Results: Summary


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