NDL:
A Domain-Specific Language for Device Drivers

Christopher Conway
Languages & Compilers Group

CS Graduate Student Seminar
Columbia University
22 April 2004
Background

- Drivers are difficult to write and error prone.
- Run in kernel mode. A driver bug can crash the entire system.
- 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.
- They are typically written in C: low-level, type unsafe.
Portability

• Driver APIs are different on each OS. (Can be quite dissimilar.)

• Low-level calls are different on each architecture.

• UDI (Uniform Device Interface) - common standard API
  – Complex and poorly supported.
  – Unix-centric
Related Work

- Holzmann (1997), Ball & Rajamani (2002) - static analysis and model checking

- Domain-specific languages
  - Thibault, et al. (1999) - language for X Windows video drivers, code 90% smaller than C.
The TAXI project

“The goal of this project is to make it possible to write a single device driver and have it automatically ported to as many operating systems as possible.”

**Front End:** Platform-neutral device specifications
(Me, Misha Litvin)

**Back End:** Device API translation between platforms
(Tom Heydt-Benjamin, John Rodriguez, Noel Vega)

- M4 macros that translate, e.g., a Linux driver into a BSD driver.
- Some success with relatively simple drivers.
NDL = “The NDL Device Language”

Typical C driver code:

```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);
```

The equivalent NDL:

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

The compiler generates C code with platform-appropriate API calls.
remoteByteCount = count ;
remoteStartAddr = start_page*FRAME_LEN ;

goto 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.
- NDL provides a structured view of device registers:
  - Fields are laid out sequentially with no implicit padding.
  - The compiler computes offsets automatically.
  - Offset and range assertions ensure consistency.
- Registers can be treated like plain variables - the compiler generates the correct access code.
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: {
        ( PAGE(0) ) => {
            /* predicated regs. */
            write rxStartAddr,
            write rxStopAddr,
            boundaryPtr,
            [ read txStatus = { /* overlay reg. */
                0: packetTransmitted,
                1: _,
                2: transmitCollided,
                3: transmitAborted,
                4: carrierLost,
                5: fifoUnderrun,
                6: heartbeatLost,
                7: lateCollision
            } volatile
            /* ... eleven bytes elided ... */
        }
        ( PAGE(1) ) => {
            /* predicated regs. */
            physicalAddr : byte[6],
            currentPage : byte,
            multicastAddr : byte[8]
        }
        ( PAGE(2) ) => {
            /* predicated regs. */
            _ : byte[13],
            read dataConfig,
            read interruptMask
        }
        0x10: dataport : fifo[1] trigger,
            _ : byte[14],
        0x1f: reset : byte trigger
    }
}
Sub-Registers

Registers often have independent sub-values as small as 1 bit.

- Each bit or group of bits can be named and typed.
- Sub-registers are treated like ordinary variables.
- The compiler generates the correct bit masking and shifting to ensure consistent access.
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: [
    ( PAGE(0) ) => { /* predicated regs. */
    write rxStartAddr,
    write rxStopAddr,
    boundaryPtr,
    [ read txStatus = { /* overlay reg. */
      0: packetTransmitted,
      1: _,
      2: transmitCollided,
      3: transmitAborted,
      4: carrierLost,
    },
    0x10: dataport : fifo[1] trigger,
    _ : byte[14],
    0x1f: reset : byte trigger
  }volatile
  5: fifoUnderrun,
  6: heartbeatLost,
  7: lateCollision
  }
}
/* ... eleven bytes elided ... */

( PAGE(1) ) => { /* predicated regs. */
  physicalAddr : byte[6],
  currentPage : byte,
  multicastAddr : byte[8]
}

( PAGE(2) ) => { /* predicated regs. */
  _ : byte[13],
  read dataConfig,
  read interruptMask
}
}
State Machines

Device behavior can often be described using state machines.
State Machines: cont’d

State machines interact in complex ways.

```
+-----------------+                  +-----------------+
| STARTED/         |                  | STARTED/         |
| DMA_DISABLED     |                  | DMA_WRITING      |
|                  +-----------------+                  +-----------------+
|                  |                      |                  |
| STOPPED/         |                  | STARTED/         |
| DMA_DISABLED     |                  | DMA_READING      |
```
NDL supports this behavior using state declarations. Legal transitions are not explicitly defined.

<table>
<thead>
<tr>
<th>state STOPPED</th>
</tr>
</thead>
<tbody>
<tr>
<td>goto DMA_DISABLED;</td>
</tr>
<tr>
<td>stop = true;</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>state DMA_DISABLED</td>
</tr>
<tr>
<td>dmaState = DISABLED;</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>state DMA_READING</td>
</tr>
<tr>
<td>goto STARTED;</td>
</tr>
<tr>
<td>dmaState = READING;</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>state DMA_WRITING</td>
</tr>
<tr>
<td>goto STARTED;</td>
</tr>
<tr>
<td>dmaState = WRITING;</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>
Predicated and Overlaid Registers

- Access to certain registers must be preceded by a transition to a particular state.
- The compiler generates the appropriate transitions.
- Predicate registers may form mutually exclusive sets (“pages” or “windows”) which may be overlaid.
- Registers may also be overlaid on disjoint read/write.
iports {
    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: [
        ( PAGE(0) ) => {
            /* predicated regs. */
            write rxStartAddr,
            write rxStopAddr,
            boundaryPtr,
            [ read txStatus = {
                /* overlay reg. */
                0: packetTransmitted,
                1: _,
                2: transmitCollided,
                3: transmitAborted,
                4: carrierLost,
                5: fifoUnderrun,
                6: heartbeatLost,
                7: lateCollision
            } volatile
        },
        /* ... eleven bytes elided ... */
    ]
    ( PAGE(1) ) => {
        /* predicated regs. */
        physicalAddr : byte[6],
        currentPage : byte,
        multicastAddr : byte[8]
    }
    ( PAGE(2) ) => {
        /* predicated regs. */
        _ : byte[13],
        read dataConfig,
        read interruptMask
    },
    0x10: dataport : fifo[1] trigger,
    _ : byte[14],
    0x1f: reset : byte trigger
}
Interrupts

- Functions can be marked to handle specific interrupt conditions.
- The compiler generates a top-level interrupt handler with branches.

```c
critical function @(countersIrq) {
    rxFrameErrors += frameAlignErrors;
    rxCrcErrors += crcErrors;
    rxMissedErrors += packetErrors;
    countersIrq = ACK;
}
```
The NDL Compiler

- Written in Standard ML, using ML-Lex and ML-Yacc.
- 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

- 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
- Pitfall: Device functions are sensitive to instruction reordering
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.

```
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
```
Debug Mode

The compiler can generate “debug” driver code

- Dumps device state at the top and bottom of every function.
- Allows programmer to verify correct operation
- Future research direction (automated validation)
Results: Summary


- Reduction in lines of code: > 50%
- Performance hit (vs. C): \(\approx 15\%\)
- Increase in executable size: > 25\%
Round trip time: Incoming

![Graph showing round trip time vs packet size](image-url)
Round trip time: Outgoing

![Graph showing the relationship between Round trip time (ms) and Packet Size (bytes). The graph compares NDL and C. The x-axis represents Packet Size (bytes) ranging from 64 to 1024, and the y-axis represents Round trip time (ms) ranging from 0.5 to 3.5. The graph indicates an increasing trend in Round trip time as Packet Size increases.]
Future Work

- More aggressive optimization (e.g., DFA)
- Formal verification
- Back-end for multiple platforms.
- Hardware donations welcome.