NDL: A Domain-Specific Language for Device Drivers

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;
```
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.

NDL Syntax
```ndl
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;
```
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.

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.

State Machines

Device behavior can often be described using state machines.

- STARTED
- DMA_READING
- DMA_DISABLED
- DMA_WRITING
- STOPPED

State Machines: cont’d

State machines interact in complex ways.

- STARTED/ DMA_DISABLED
- STOPPED/ DMA_DISABLED
- STARTED/ DMA_READING
- STARTED/ DMA_WRITEING

State Declarations

NDL supports this behavior using state declarations. Legal transitions are not explicitly defined.

```
state STOPPED {
  goto DMA_DISABLED;
  dmaState = DISABLED;
  return;
}

| STARTED | start = true;
| OFF | DMA_ENABLED {
  goto DMA_DISABLED;
  dmaState = DISABLED;
  return;
}
```

Interrupts

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

```
critical function handleInterrupts {
  if (interrupt.type == 0) {
    // Interrupt handler code
  }
  else if (interrupt.type == 1) {
    // Interrupt handler code
  }...
```
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:
  \[
  \text{LOAD} \ n, \ register[a:b] \\
  \text{STORE} \ register[a:b], \ n
  \]
- 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

Field Aggregation

Orthogonal writes to sub-registers can be consolidated.

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)

Idempotence

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

Results: Summary


- Reduction in lines of code: > 50%
- Performance hit (vs. C): ≈ 15%
- Increase in executable size: > 25%

Round trip time: Incoming

Round trip time: Outgoing

Future Work

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