NDL: A Domain-Specific Language for Device Drivers

Christopher L. Conway
conway@cs.columbia.edu

Stephen A. Edwards
sedwards@cs.columbia.edu

Department of Computer Science, Columbia University
1214 Amsterdam Ave., New York, NY 10027

ABSTRACT

Device drivers are difficult to write and error-prone. They are usually written in C, a fairly low-level language with minimal type safety and little support for device semantics. As a result, they have become a major source of instability in operating system code.

This paper presents NDL, a language for device drivers. NDL provides high-level abstractions of device resources and constructs tailored to describing common device driver operations. We show that NDL allows for the coding of a semantically correct driver with a code size reduction of more than 50% and a minimal impact on performance.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*domain-specific languages*; D.1.2 [**Programming Techniques**]: Automatic Programming—*program synthesis*; D.2.3 [**Software Engineering**]: Coding Tools and Techniques

General Terms

Design,Languages,Reliability

Keywords

Device drivers, systems programming, domain-specific languages

1. INTRODUCTION

Device drivers are critical, low-level systems code. Traditionally, they have been written in C due to its efficiency and flexibility. Unfortunately, sophisticated device interaction protocols and C's lack of type safety make driver code complex and prone to failure. Indeed, device drivers have been noted as a major source of faults in operating system code [2]. However, no other high-level language is widely accepted for device programming.

LCTES'04, June 11–13, 2004, Washington, DC, USA.

To address this challenge, we introduce NDL[1], a domain-specific language for device drivers. The language includes direct support for the semantics of device drivers, leading to shorter, more maintainable driver code. NDL can be used to write drivers for a wide variety of devices in a platform-neutral fashion. The compiler is designed to be readily ported to a broad class of operating systems.

In this paper, we demonstrate the features that make NDL device drivers simple and concise and show that the NDL compiler produces code which is only marginally less efficient than the equivalent C. Throughout this paper, we will illustrate the use of NDL using examples from the driver for an NE2000-compatible Ethernet adapter.

2. RELATED WORK

A variety of approaches have been suggested to improve the reliability of low-level software and device driver software in particular. Crary and Morrissett [3] propose typed assembly language as a compiler target for preserving type information from higher-level languages. Unfortunately, C, the most common systems programming language, is not much more strongly typed than a traditional assembly language; there is little the compiler can do to improve the type safety of C code.

Deline and Fähndrich [4] use a similar typing system in the C-like programming language VAULT. The use of variables is controlled through type guards that describe when an operation on a variable is valid. In order for the compiler to accept the program, it must respect the type guards' access specifications and types must match at program join points. VAULT has shown some success in preventing common programmer errors, but its limitations on alias types prevent it from being generally applicable to device driver development.

Holzmann [5] and Ball and Rajamani [1] have addressed the use of static analysis in the verification of traditional C systems software. Static analyses can find subtle bugs and increase confidence in the code, but the types of detectable errors are restricted and the quality of the analysis depends on programmer-written property specifications.

A group at the University of Rennes has done perhaps the most work on domain-specific languages for device drivers. Thibault et al.'s GAL [8] is a domain-specific language for X Windows video drivers. Their compiler combines a partial evaluation framework with a language tailored to video driver operations to produce driver code that is nearly 90%

[1]With apologies to GNU, NDL stands for "The NDL Device Language".

smaller than the equivalent C code and just as fast. This work is promising, but the methodology may not be applicable to a wider variety of device drivers.

Also at Rennes, Mérillon et al. [6] developed Devil, an interface definition language designed to be a more general solution for device driver development. A Devil specification describes hardware components such as I/O ports and memory-mapped registers. The specification is compiled into a set of C macros for device manipulation; the macros are called from traditional C driver code, allowing the driver programmer to avoid writing the lowest-level code by hand. This approach prevents certain common programming errors, but it does not specify the protocol for using the device, and it does not provide the type safety of a higher-level solution. Nevertheless, portions of NDL borrow liberally from Devil's interface definition syntax.

O'Nils and Janstch [7] and Wang et al. [9] have also developed tools for device driver synthesis, but they are primarily concerned with hardware/software codesign for embedded systems. NDL provides a more flexible tool for a wider range of devices.

3. NDL

NDL is a language for device driver development that provides high-level constructs for device programming, describing the driver in terms of its operational interface. Its declarations are designed to closely resemble the specification document for the device it controls. An NDL driver is typically composed of a set of register definitions, protocols for accessing those registers, and a collection of device functions. The compiler translates register definitions and access protocols into an abstract representation of the device interface. Device functions are then translated into a series of operations on that interface.

Device drivers are systems-level code that interact directly with the operating system. The NDL compiler generates C that makes appropriate operating system calls. Platform-specific functions are provided through compiler libraries and device templates; platform dependencies are minimal.

NDL's main abstraction is a representation of the state of the peripheral device being controlled. Internally, devices can be in a variety of states, e.g., receiving data, waiting for a certain condition, or having encountered an error. From software, this state can be fairly awkward to access. At its simplest, it is spread out across bits in I/O memory packed carefully into bytes. Typically it is even more complicated, often consisting of an address and data register pair that provide indirect access to internal device registers. In both cases, NDL provides an abstraction layer that provides transparent access to this state information.

For example, a typical sequence of I/O memory accesses in C might be coded

```
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);
```

Bit flags are combined and the integer `count` is shifted and masked to force its value into a disjoint register format. The write operations are a series of low-level I/O calls, using predefined offsets on the device's base address. The equivalent NDL code uses a simple assignment syntax:

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

NDL also provides a mechanism for describing groups of mutually exclusive states and procedures for switching among them. As a result, with a single line of NDL code the programmer is able to effect a state transition that would require many I/O operations in an equivalent C program.

Copying large amounts of data to and from buffers is another frequent device operation. There are common idioms for a buffer-to-buffer copy in C, but they are complicated by the need to support compatible devices with different data bus widths in the same driver. NDL treats buffer copying as a special case of assignment. The following fragment reads `count` bytes read from the FIFO `dataport` (defined elsewhere) and placed in the array `buffer` using the appropriate I/O operations.

```
var buffer: byte[count];
buffer = dataport;
```

3.1 Device Inheritance

An NDL driver may begin by declaring a parent device from which it should inherit interface and template information. For example, the NE2000 driver declares itself as an extension of `NetworkDevice`, which is an abstract device that defines functions and variables common to all Ethernet adapters, such as the `start_transmit` function, which takes a packet buffer and queues it for transmission.

In addition, `NetworkDevice` is associated with a template that specifies the boilerplate code required to implement a network driver on a particular platform (e.g., initialization and release of operating system data structures and requests for system resources). Templates provide the glue layer between the high-level device specification and platform-specific C driver code.

3.2 Device States

Often, aspects of a device's behavior are most easily modeled as finite-state machines. For example, the direct memory access system in the NE2000 can be either transferring data to the card, from the card, or can be idle. NDL supports this view through explicitly defined state machines, such as those in Figure 1. A state machine is defined as a list of mutually exclusive states separated by '`||`'. Each state may have a an associated list of statements intended to switch the machine to that state, invoked when a corresponding `goto` statement is executed.

The states `STOPPED` and `STARTED` are mutually exclusive. When the device needs to enter the `STOPPED` state, the statements "`goto DMA_DISABLED`" (indicating a transition to state `DMA_DISABLED`) and "`stop = true`" will be executed. When the device needs to enter the `STARTED` state, the statement "`start = true`" will be executed.

The states `DMA_DISABLED`, `DMA_READING` and `DMA_WRITING` also form a mutually exclusive set. Notice that `DMA_READING` and `DMA_WRITING` induce a transition to the state `STARTED`; the two machines can interact provided the constraints are not cyclic.

The state `PAGE` is parameterized and takes an integer argument in the range 0 to 2, inclusive. When a state definition is parameterized, each parameter value forms a separate mutually exclusive state.

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

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

  state PAGE(i : int{0..2}) {
    registerPage = i;
  }
```

Figure 1: State declarations for the NE2000.

NDL's state machine declarations differ substantially from those in a hardware description language such as Verilog. The main difference is that NDL's machines are only slightly constrained models whereas those in an HDL must be completely specified. For example, NDL assumes that any transition between states is possible, whereas many would be illegal in the actual machine. In the future, we plan to allow a programmer to prohibit certain transitions and automatically check that they can never be taken.

The syntax for state definitions is shown in Figure 2, beginning with the nonterminal *StateDecl*.

3.3 Memory-mapped I/O

Figure 3 illustrates one of the key features of NDL: a structured definition of memory-mapped I/O locations and their meaning. Drivers primarily interact with devices by reading and writing memory-mapped I/O locations, but the layout and behavior of these locations is often quite baroque and is one of the main sources of complexity in device drivers. One of NDL's key contributions is a structured way of describing these locations and their meanings, simplifying code that interacts with the device and reducing errors through increased transparency and automatic consistency checks.

The `ioport` declaration defines a block of memory-mapped I/O locations. In addition to giving names to words, bytes, and even bits within these ranges, the `ioport` declaration may also generate code that requests exclusive access to these locations from the operating system. The syntax for the `ioports` declaration and register definitions is shown in Figure 2, beginning with the nonterminal *IoportDecl*.

An `ioports` declaration resembles a C `struct`, but has more precise semantics and facilities for ensuring consistency. Each field must consume an integral number of bytes and each bit of every byte must be accounted for. The fields are laid out sequentially with no implicit padding.

StateDecl → `state` *StateDesc* (`||` *StateDesc*)*

StateDesc → *Id*
| *Id* (`(` *FParams*? `)`)? `{` *SimpleStmts* `}`

IoportDecl → `ioports` `{` *RegList* `}`

RegList → *RangeAssert*? *RegDecl* (`,` *RegList*)?

RegDecl → *RW*? *SimpleReg*
| *RW*? *Id* `=` `{` *SimpleRegList* `}` *RegProp*
| `[` *RegDeclList* (`||` *RegDeclList*)+ `]`
| `(` *StateAssert* `)=>` *RegDeclList*

RegDeclList → *RegDecl* (`,` *RegDecl*)*

RangeAssert → *Num* (`..` *Num*)? `:`

StateAssert → *Id* (`(` *Params* `)`)?

RW → `read` | `write`

SimpleReg → *Id*
| *Id* `:` *RegProp*
| *Id* `:` *TypeDesc* *RegProp*?
| *Id* `:` `{` (*Id* `=` (*Num* | *Mask*))+ `}`

SimpleRegList →
RangeAssert? *SimpleReg* (`,` *SimpleRegList*)?

RegProp → `trigger` (`except` *Num*)?
| `volatile`
| `nopred`

Figure 2: A portion of the NDL grammar. Productions for some nonterminals are omitted.

While the compiler computes the offset from the base I/O address of each field, the programmer may also include an offset or range for any field to ensure consistency. The compiler signals an error if any user-provided offset is inconsistent with the computed addresses or sizes of the fields.

Figure 3 shows a portion of the `ioports` declaration for the NE2000 network card. The one-byte register `command` (Lines 2-15) occupies the first position in the device memory, followed by fifteen bytes of additional registers. The registers `dataport` and `reset` (Lines 55 and 57) are preceded by a range assertions (e.g., "`0x10:`"). If the compiler has not allocated 16 bytes of device memory before it encounters the definition of `dataport`, compilation will fail. To pad the unused space between `dataport` (byte 16) and `reset` (byte 31), a 14-byte register is labeled with the don't-care symbol '`_`' (Line 56).

The definition of `command` register illustrates other constructs for defining memory-mapped I/O behavior. The byte-wide register is subdivided into five fields, or sub-registers: `stop`, `start`, `transmit`, `dmaState`, and `registerPage`. Each field is preceded by an optional bit offset assertion; just as with byte offsets, the compiler uses this redundant information to assure the consistency of the register definition.

The values `stop`, `start` and `transmit` are of the default

```
1  ioports {
2    command = {
3      0:  stop : trigger except 0,
4      1:  start : trigger except 0,
5      2:  transmit : trigger except 0,
6      3..5:
7          dmaState : {
8            READING = #001
9            WRITING = #010
10           SENDING = #011
11           DISABLED = #1**
12         } volatile,
13     6..7:
14         registerPage : int{0..2}
15   },
16
17 0x01..0x0f:
18   [
19     ( PAGE(0) ) => {   /* predicated regs. */
20       write rxStartAddr,
21       write rxStopAddr,
22       boundaryPtr,
23
24       [
25         read txStatus = {  /* overlay reg. */
26           0: packetTransmitted,
27           1: _,
28           2: transmitCollided,
29           3: transmitAborted,
30           4: carrierLost,
31           5: fifoUnderrun,
32           6: heartbeatLost,
33           7: lateCollision
34         } volatile
35       ||
36         write txStartAddr
37       ],
38
39       /* ... eleven bytes elided ... */
40     }
41   ||
42     ( PAGE(1) ) => {  /* predicated regs. */
43       physicalAddr : byte[6],
44       currentPage : byte,
45       multicastAddr : byte[8]
46     }
47   ||
48     ( PAGE(2) ) => {  /* predicated regs. */
49       _ : byte[13],
50       read dataConfig,
51       read interruptMask
52     }
53   ],
54
55 0x10: dataport : fifo[1] trigger,
56       _ : byte[14],
57 0x1f: reset : byte trigger
58 }
```

Figure 3: A portion of the `ioports` declaration for the NE2000.

type for a sub-register (one bit) and are marked as `trigger` fields, meaning each write to the field causes a side effect. The three-bit `dmaState` field has an enumeration type; its values are defined using the binary literal syntax (i.e., preceded by '`#`'). The value `DISABLED` is a bit-mask—only its most significant bit is relevant and the rest take don't-care values. The `registerPage` field occupies the remaining two bits of the eight-bit `command` register. The field is defined to range over the integers 0, 1, and 2.

Note that the fields of `command` occupy exactly eight bits. Every register must occupy a whole number of bytes and every bit within the register must be accounted for; otherwise the NDL compiler signals an error. Unused or ignored bits in a register must be labeled with the don't-care symbol '`_`'.

Registers can also be given the attributes `read` and `write`, indicating they are read- or write-only. If a register is doubly defined—once with `read` and once with `write`—the definitions will be unified into a single register with disjoint read and write ports.

3.3.1 Overlaid and Predicated Registers

NDL provides explicit support for device registers that occupy the same I/O addresses under different circumstances. The `txStatus` and `txStartAddr` fields in Figure 3 (Lines 24-37) illustrate a common case of such overlay registers (our term): they share an I/O location that means one thing when read and something else when written.

Multiple registers may be included in an overlay (e.g., one 2-byte register may be overlaid with two 1-byte registers, or three 1-byte read-only registers maybe be overlaid with three 1-byte write-only registers using only one parallel operator). Each memory location thus defined must contain at most one `read` and one `write` definition.

As is fairly common in many devices, most I/O locations in the NE2000 take on different roles depending on the setting of an address field (in the NE2000, this is the two-bit `registerPage` field on Line 14). We model such behavior in NDL with the concept of predicated registers.

Predicated registers, declared with the syntax "`(`*state*`) => {`*reg-list*`},`" are accessible only when their predicate is true, i.e., when a particular state machine is in the prescribed state. The declaration in Figure 3 contains three groups of predicated registers. The registers `rxStartAddr`, `rxStopAddr`, `boundaryPtr`, `txStatus`, and `txStartAddr` are governed by the predicate `PAGE(0)` (Lines 19-40). By definition, any attempt to access these registers must be preceded by a transition to state `PAGE(0)`. The compiler will generate this transition automatically. Similarly, the registers `physicalAddr`, `currentPage` and `multicastAddr` are governed by the predicate `PAGE(1)` (Lines 42-46). The compiler will precede any attempt to access these registers with a transition to state `PAGE(1)`. Similarly, the `dataConfig` and `interruptMask` registers are governed by the predicate `PAGE(2)` (Lines 48-52).

The predicated registers in Figure 3 are also overlaid. In this case, the overlay registers are distinguished by their mutually-exclusive state predicates rather than by their read or write access modifiers. Any number of register lists may be overlaid in this fashion, provided their predicates are mutually exclusive. (Recall the parameterized definition of the state `PAGE(i)` from Figure 1.) Each register list in a predicated register overlay must occupy exactly the same number of bytes in the device memory as the other register lists in

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

Figure 4: An anonymous interrupt function for the NE2000.

the overlay. The `PAGE(1)` and `PAGE(2)` register lists both occupy 15 bytes. Several registers have been omitted from the `PAGE(0)` group for space reasons, but together they must also total 15 bytes.

3.4 Functions and Interrupts

Device functions look much like functions in any imperative, procedural language. Local variables are allowed, as are reads from and writes to registers. Function calls are available, but recursion is prohibited. Loops are bounded by timers or by constant integers.

Functions marked with `@(`*bool-expr*`)` handle interrupts. Such functions are invoked by a compiler-generated master interrupt handling routine dispatched by the operating system in response to interrupts from the device. When an interrupt occurs, each interrupt function's boolean expression is evaluated and, if true, the associated function is executed.

Figure 4 shows an anonymous interrupt-handling function. In this example, the function will be invoked when the `countersIrq` register is true, i.e., when a counter interrupt occurs. The function accumulates the device counters and acknowledges the interrupt, clearing the execution condition for future evaluation. (`ACK` is a built-in constant equal to 1, reflecting the common interrupt-handling convention.) By not naming the function, the programmer indicates that it is to be called only from the interrupt handler and not from elsewhere in the driver.

3.5 Library Functions

NDL provides a library of functions that wrap platform-specific system calls or perform generally useful operations. For example, Ethernet cards usually have a transmit buffer which is filled from an operating system queue of outgoing packets. When the transmit buffer is full, the card must issue a signal to stop pulling packets off the queue until there is room in the buffer. The nature of this signal will vary between operating systems. The NDL library function `os_stop_queue` provides access to this signal in a platform-neutral way. Other library functions include `print` (which wraps the platform kernel-mode logging facility) and `systemTimeMillis`.

4. IMPLEMENTATION

The NDL compiler is written in Standard ML. The front-end uses ML-Lex and ML-Yacc to generate an abstract syntax tree. The semantic analyzer produces a three-address-style intermediate code with C-like control-flow structures. The back-end performs a straightforward syntax-directed translation from intermediate code to C.

The intermediate representation targets an abstract machine with a C-like imperative programming model. Its main distinguishing feature is operations for loading from and storing to a device register, e.g.,

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

These operations are parameterized by bit ranges, allowing arbitrary bit-fields to be read from and written to a register. These bit ranges allow sub-register accesses to be translated directly into intermediate form.

The intermediate code generated during semantic analysis is unoptimized. Every register access is preceded by a state-change predicate, if necessary, and these state-change predicates are expanded inline. If two adjacent register accesses share a state predicate, the predicate-setting code will be included once for each register. In the semantic analysis phase, the compiler is concerned primarily with correctness and simplicity; it is expected that an optimization phase will reduce or eliminate undesirable redundancy.

4.1 Optimizations

We have implemented two optimizations on the NDL IR: we remove redundant writes to idempotent registers and aggregate multiple writes to adjacent fields. Since the C compiler also performs optimization on the generated code, we focus on higher-level optimizations that depend on domain knowledge that would be unavailable to a C compiler. At the same time, we strive to produce code that will not complicate the optimization phase of the C compiler. Our goal is to produce target code that corresponds to the best practices of C programmers with respect to execution efficiency.

The optimization passes described may make unsafe assumptions about the behavior of the device. To avoid incorrect code generation, optimizations are turned off by default and each pass may be enabled independently.

4.1.1 Idempotent Register Values

Many registers behave like memory in that writing the same value to them repeatedly has the same effect as writing the value once. We term such registers idempotent, and can eliminate all redundant write operations to such a register without changing the behavior of the device. This is particularly useful with paged or bank-switched registers. In this case, several sets of logical registers, called pages, are overlaid on the same memory area. To access a particular register, the page is selected by first writing to an index register. (The registers on Lines 18-53 in Figure 3 are paged by the index register `registerPage`.) If multiple registers on the same page are accessed in sequence, the naive algorithm will generate a write to the index register before each access. If the index register holds its value (i.e., is idempotent), as is typical, these redundant operations can be eliminated. NDL considers registers idempotent by default; a programmer must mark non-idempotent registers with the `trigger` keyword.

The idempotence code transformation looks for a constant assignment to an idempotent register value. It then searches forward, deleting subsequent assignments of the same value to the same register, until it reaches a program join point or the assignment of a different value to the same register.

4.1.2 Field Aggregation

As described in Section 3.3, NDL allows the programmer to directly address groups of bits within bytes. These ref-

erences are translated by the compiler into reads from and writes to bit-fields within the address. To protect adjoining values from interference, writing a sub-register is typically implemented using a read-modify-write pattern, effectively doubling the number of load and store operations necessary to perform a single write. However, if several values contained within the same register are accessed in sequence and those values do not depend on the sequence of access, the sequence of read-modify-write operations may be condensed into a single operation.

The sub-register aggregation code transformation looks for adjacent assignments to contiguous bit-fields within the same register. The two store operations are transformed into a single assignment to the merged bit-fields by shifting, masking and combining the two assignment values. For example, the following intermediate code sequence could be optimized by combining the second and third instructions:

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

The resulting intermediate code would be:

```
STORE command[0:0], 1
STORE command[3:7], 4
```

In fact, bits 1 and 2 of the `command` register (the `stop` and `transmit` sub-registers, respectively) each have a write-neutral value of 0. Writing to the full register does not require preserving those bits. Taking this into consideration, the above code can be transformed into:

```
STORE command[0:7], 0x21
```

4.2 Debugging Mode

The NDL compiler can generate driver code that includes a debugging feature. In this mode, the driver dumps the device state (i.e., the value of every non-volatile register) to the system log on entrance to and exit from every device function. This level of debugging information is rather coarse, but we have found it extremely useful while coding drivers.

5. RESULTS

We wish to demonstrate the utility of NDL with respect to three criteria: the concision of NDL device specifications, the efficiency of the code generated by the NDL compiler, and the size of the final object code. We measure concision as the number of non-comment lines of source code. As a measure of efficiency, we present the average response time under heavy load.

We present results from two drivers: a null driver (i.e., a character device implementing the standard Unix `/dev/null` functionality) and a driver for the NE2000-compatible D-Link DE-220P network card. The NE2000 driver is compared to the equivalent driver from the Linux 2.4 kernel source distribution. The null driver is compared to a hand-coded C driver which differs from the standard Linux driver chiefly in using the loadable kernel module facility.

5.1 Lines of Code

Table 1 shows the number of lines of code for equivalent C and NDL device drivers. Lines of code were counted using David A. Wheeler's SLOCCount tool. The figure for the NE2000 C driver is the total of the lines-of-code figures for three files—`ne.c`, `8390.c`, and `8390.h`—each of which is essential to the proper operation of the driver.

	Lines of code		Executable size (bytes)	
Driver	C	NDL	C	NDL
null	74	7	495	1119
NE2000	1370	677	13623	16934

Table 1: Code measurements for NDL vs. C.

The NDL code for the null driver is more than 90% shorter than the equivalent C code, but the null driver is an exceptional case since the C code is dominated by boilerplate code that the NDL compiler generates itself. For moderately complex drivers, the expected gains in code length are probably closer to those seen with the NE2000 driver: somewhat more than 50%. Even so, this is a dramatic increase in concision.

Our results are not directly comparable to those of the Devil project—the Devil NE2000 driver was written for a PCI variant with a separate C driver (`ne2k-pci.c`). However, it may be useful to note that Devil showed an increase of about 25% in lines of code relative to the comparable C driver. The primary contribution of Devil was convenience rather than concision; NDL provides both.

5.2 Performance

We compared the performance of our NE2000 driver generated by NDL with the stock Linux driver to evaluate the performance of the code generated by the NDL compiler. We connected a machine with an NE2000 card to a 10Mb Ethernet hub with only one other machine connected and then invoked `ping` in "flood mode" to send 5000 uniform packets in a short period of time (i.e., "`ping -f -c 5000`"). In individual tests we instructed the driver to send and receive packets ranging from 64 to 1024 bytes (the size is that of the ICMP (ping) packet; the actual Ethernet packet, including headers, is somewhat larger). Figures 5 and 6 show the results of these tests.

For both incoming and outgoing packets, the round trip time for the C driver is less than 15% faster on average, and the relative difference diminishes as the packets get larger. This suggests NDL will be practical for all but the most performance-sensitive drivers; further optimizations could narrow the gap.

5.3 Code Size

We compared the size of executable from the NDL compiler with the stock Linux driver by running the `size` command. The figure for the NE2000 C driver is the sum of the sizes of `ne.o` and `8390.o`. Together, the two modules comprise the standard driver.

For the NE2000 driver, the object code generated from the native C code driver is less than 20% smaller. For the null driver, the C code is more than 55% smaller, reflecting a large static overhead in the generated code. To date, we have not focused on optimizing for executable size, but further improvements are possible.

6. FUTURE WORK

NDL improves the ease and reliability of driver development at a slight cost in performance. The results of the

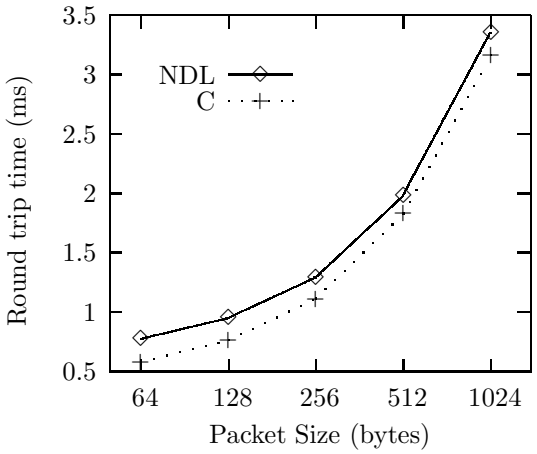

Figure 5: Average round trip time for incoming packets.

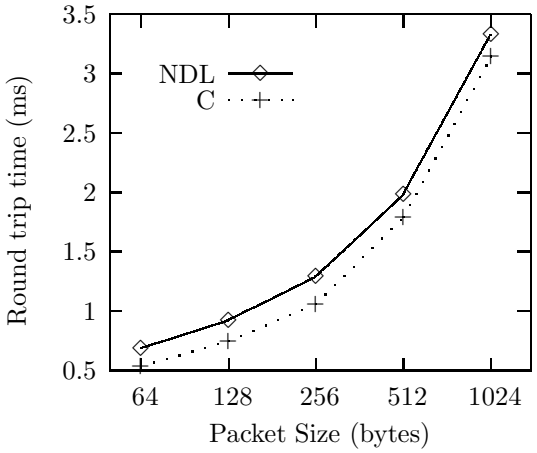

Figure 6: Average round trip time for outgoing packets.

GAL project [8] suggest this trade-off can be mitigated using clever compilation techniques. Current optimization strategies are able to improve the performance of the driver by only a few percent. We believe that more aggressive optimizations could lead to generated drivers that match or exceed the performance of hand-coded C drivers.

NDL provides a high-level description of a device interface, with information about underlying state-based behavior and usage protocols maintained. Currently, this information is used only to increase the concision of the language. In the future, we will augment the compiler with static verification functions in order to improve the correctness and robustness of the driver.

Currently, the NDL compiler generates C code for the x86 Linux platform. Future versions will target additional architectures and operating systems.

7. REFERENCES

[1] T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Symposium on Principles of Programming Languages (POPL)*, pages 1–3, Portland, Oregon, January 2002.

[2] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. R. Engler. An empirical study of operating system errors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, volume 35 of *Operating System Review*, pages 73–88, Banff, Alberta, Canada, October 2001.

[3] K. Crary and G. Morrisett. Type structure for low-level programming languages. In *International Colloquium on Automata, Languages, and Programming (ICALP) 1999*, volume 1644 of *Lecture Notes in Computer Science*, pages 40–54, Prague, Czech Republic, July 1999. Springer Verlag.

[4] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 59–69, Snowbird, Utah, June 2001.

[5] G. J. Holzmann. Software analysis and model checking. In *Computer Aided Verification, 14th International Conference (CAV)*, volume 2404 of *Lecture Notes on Computer Science*, pages 1–16, Copenhagen, Denmark, July 2002. Springer Verlag.

[6] F. Mérillon, L. Réveillère, C. Consel, R. Marlet, and G. Muller. Devil: An IDL for hardware programming. In *Proceedings of the 4th USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 17–30, San Diego, California, October 2000.

[7] M. O'Nils and A. Jantsch. Operating system sensitive device driver synthesis from implementation independent protocol specification. In *Proceedings of Design Automation and Test in Europe (DATE)*, pages 562–567, Munich, Germany, March 1999.

[8] S. Thibault, R. Marlet, and C. Consel. Domain-specific languages: from design to implementation—application to video device drivers generation. *IEEE Transactions on Software Engineering*, 25(3):363–377, May-June 1999.

[9] S. Wang and S. Malik. Synthesizing operating system based device drivers in embedded systems. In *Proceedings of the First International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, Newport Beach, CA, October 2003.