Devil: An IDL for Hardware Programming

Fabrice Mérillon Laurent Réveillère*
Charles Consel* Renaud Marlet† Gilles Muller

Compose Group, IRISA / INRIA, University of Rennes I
Campus Universitaire de Beaulieu, F-35042 Rennes Cedex, France

E-mail: `{merillon,lreveill,consel,marlet,muller}@irisa.fr`

Abstract

To keep up with the frantic pace at which devices come out, drivers need to be quickly developed, debugged and tested. Although a driver is a critical system component, the driver development process has made little (if any) progress. The situation is particularly disastrous when considering the hardware operating code (i.e., the layer interacting with the device). Writing this code often relies on inaccurate or incomplete device documentation and involves assembly-level operations. As a result, hardware operating code is tedious to write, prone to errors, and hard to debug and maintain.

This paper presents a new approach to developing hardware operating code based on an Interface Definition Language (IDL) for hardware functionalities, named Devil. This IDL allows a high-level definition of the communication with a device. A compiler automatically checks the consistency of a Devil definition and generates efficient low-level code.

Because the Devil compiler checks safety critical properties, the long-awaited notion of robustness for hardware operating code is made possible. Finally, the wide variety of devices that we have already specified (mouse, sound, DMA, interrupt, Ethernet, video, and IDE disk controllers) demonstrates the expressiveness of the Devil language.

*Author's current address: LaBRI / ENSERB, 351 cours de la Libération, F-33405 Talence Cedex, France.

†Author's current address: Trusted Logic, 5 rue du Bailliage, F-78000 Versailles, France. E-mail: Renaud.Marlet@trusted-logic.fr.

1 Introduction

A device driver is a key system component that makes hardware innovation available to end users. Device drivers are critical both in general-purpose computers and in the fast-evolving domain of appliances. If driver development falls behind, product competitiveness can be compromised. If a device driver is faulty, a hardware innovation may turn into a disaster instead of improving competitiveness.

Still, ever since the first device drivers have been written, their development process has made little (if any) progress. This situation has particularly disastrous effects when considering *hardware operating code* (*i.e.,* code communicating with the hardware). This layer of code is well-known to be *low level* and *error prone.*

Hardware operating code is low level because it consists of many bit operations. Indeed, we have found that bit operations can represent up to 30% of driver code[1]. Such low-level programming is obviously prone to errors and requires tedious debugging. In fact, advances in programming languages have had no impact on the development of hardware operating code: there is no syntactic support for low-level operations, there is no verification support to identify incorrect usage of these operations, and there is no tool support to facilitate debugging.

Additionally, hardware documentation typically contains imprecise or inaccurate information. Therefore, writing hardware operating

[1] This measurement was performed on various Linux 2.2-12 drivers.

code typically involves laboriously searching for obscure incantations aimed at performing specific operations on the device. Not only can this sometime cause unexpected behavior, but it also makes re-use of hardware operating code difficult.

Finally, there are no recognized methodologies for structuring device drivers. Even worse, a driver is often written by modifying an existing one. As a result, the code quickly becomes tangled, which makes debugging and maintenance complex.

Our proposal

This paper describes a new approach to developing the hardware operating layer of a driver. Our approach allows drivers to be written in a high-level language, allows important safety properties to be checked, and allows low-level code to be automatically generated.

We introduce an Interface Definition Language (IDL) to describe hardware functionalities, named Devil. IDLs are extensively used in modern OSes, either to hide heterogeneity and intricacies of message construction in distributed systems [3, 13], or to glue together components in modular operating systems [2, 9, 10]. Just as RPC IDLs conventionally define operations and their input/output types, Devil specifies the functional interface of the device. To do so, it provides the programmer with abstractions and syntactic constructs that are specific to describing devices. From a Devil specification, a compiler automatically generates stubs containing low-level code to operate the device. Furthermore, verification tools enable critical safety properties to be checked at compile time, and at run time if necessary.

Just as an IDL typically allows code to be re-used, a Devil specification can be re-used in different contexts (*e.g.,* various operating systems). More generally, our vision is that Devil specifications either should be written by device vendors or should be widely available as public domain libraries in order to ease driver development.

Our contributions are as follows.

- We have designed and implemented an IDL for devices. This language is an alternative to assembly-language-like programming of devices.
- We propose tools to verify critical safety properties of hardware operating code. These tools enable us to provide the long-awaited notion of *robustness* for device drivers.
- We present a comparison between Devil specifications and existing driver code. This comparison is based on experimental data which demonstrate that a Devil specification is up to 5.9 times less prone to errors than C code, with almost no loss in performance.

The rest of this paper is organized as follows. Section 2 presents the Devil language. Section 3 describes the safety properties that can be verified both statically on Devil specifications and dynamically by the generated interface. Section 4 assesses the benefits of our approach by comparing hand-crafted drivers with equivalent ones written using Devil. Section 5 describes related work. Section 6 concludes and suggests future work.

2 Devil

Devil is an IDL for specifying the functional interface of a device. To design Devil, we have studied a wide spectrum of devices and their corresponding drivers, mainly from Linux sources: Ethernet, video, sound, disk, interrupt, DMA and mouse controllers. This study was supported by literature about driver development [7, 16], device documentation available on the web, and discussions with device driver experts for Windows, Linux and embedded operating systems. Devil has proved expressive enough to describe even devices having a contorted interface such as the Crystal CS4236B sound controller.

Concretely, a device can be described by three layers of abstraction: *ports*, *registers*, and *device variables*. The entry point of a Devil specification is the declaration of a device, parameterized by *ports* or ranges of ports, which

```
1  device logitech_busmouse (base : bit[8] port @ {0..3})
   {
      // Signature register (SR)
4     register sig_reg = base @ 1 : bit[8];
5     variable signature = sig_reg, volatile, write trigger : int(8);

      // Configuration register (CR)
8     register cr = write base @ 3, mask ’1001000.’ : bit[8];
9     variable config = cr[0] : { CONFIGURATION => ’1’, DEFAULT_MODE => ’0’ };

      // Interrupt register
12    register interrupt_reg = write base @ 2, mask ’000.0000’ : bit[8];
13    variable interrupt = interrupt_reg[4] : { ENABLE => ’0’, DISABLE => ’1’ };

      // Index register
16    register index_reg = write base @ 2, mask ’1..00000’ : bit[8];
17    private variable index = index_reg[6..5] : int(2);

19    register x_low  = read base @ 0, pre {index = 0}, mask ’****....’ : bit[8];
20    register x_high = read base @ 0, pre {index = 1}, mask ’****....’ : bit[8];
21    register y_low  = read base @ 0, pre {index = 2}, mask ’****....’ : bit[8];
22    register y_high = read base @ 0, pre {index = 3}, mask ’...*....’ : bit[8];

24    structure mouse_state = {
25       variable dx = x_high[3..0] # x_low[3..0], volatile : signed int(8);
26       variable dy = y_high[3..0] # y_low[3..0], volatile : signed int(8);
27       variable buttons = y_high[7..5], volatile : int(3);
      };
   }
```

Figure 1: Logitech Busmouse Specification

abstract physical addresses. Ports then allow device *registers* to be declared; these define the granularity of interactions with the device. Finally, *device variables* are defined from registers, forming the functional interface to the device.

These three layers of abstraction are illustrated by the following fragment of the Devil description of the Logitech Busmouse controller (see Figure 1 for a complete description).

```
device logitech_busmouse(base : bit[8] port@{0..3})
{
  register sig_reg   = base @ 1 : bit[8];
  variable signature = sig_reg, ... : int(8);
  ...
}
```

The `logitech_busmouse` declaration is parameterized by a range of ports specified as the main address `base` and a range of offsets (from 0 to 3). An eight-bit register `sig_reg` is declared at port `base`, offset by `1`. Finally, the device variable `signature` is the interpretation of this register as an eight-bit unsigned integer. This fragment declares a device whose functional interface consists of a device variable (`signature`). Only device variables are visible from outside a Devil description ports and registers are hidden. In fact, for each variable the Devil compiler generates two C stubs that permit to write or read the variable by emitting the proper I/O operations.

In the rest of this section, we first describe the basic Devil constructs, and then present advanced Devil features that allow the description of devices with contorted addressing modes.

2.1 Basic Devil

Ports, registers, and device variables are the basic layers of abstraction that describe the interface of a device. We now present their usage by describing in detail the Devil specification of the Logitech Busmouse (see Figure 1), and a fragment of the NE2000 Ethernet controller.

Ports. The port abstraction is at the basis of the communication with the device. A port hides the fact that, depending on how the device is mapped, it can be operated via either I/O or memory operations. A device often has several communication points whose addresses are derived from one or more base addresses. Therefore, the port constructor, denoted by `@`, takes as arguments a ranged port and a constant offset (*e.g.,* `base@1` as illustrated by line 4 of the Busmouse specification). To enable verification, the range of valid offsets must be specified within the entry point declaration (*e.g.,* `port@{0..3}` as illustrated by line 1 of the Busmouse specification).

Registers. Registers define the granularity of interaction with a device; as such register size (in number of bits) must be explicitly specified. Registers are typically defined using two ports: one for reading and one for writing. Only one port needs to be provided when reading and writing share the same port, or when the register is read-only or write-only.

A register declaration may be associated with a mask to specify bit constraints. An element of this mask can either be '`*`' to denote a relevant bit, '`0`' or '`1`' to denote a bit that is irrelevant when read but has a fixed value (`0` or `1`) when written, or '`-`' to denote a bit that is irrelevant whether read or written. As an example, consider the declaration of the `write-only` register `index_reg` in line 16 of the Busmouse specification.

```
register index_reg =
   write base@2, mask '1..00000' : bit[8];
```

This mask indicates that only bits 6 and 5 are relevant. Also, bit 7 is forced to `1` when written while bits 4 through 0 are forced to `0`. Proper register masking is performed as part of the stubs generated by the Devil compiler.

Device variables. In order to minimize the number of I/O operations required for communicating with a device, hardware designers often group several independent values into a single register. Accessing these values requires bit mask and shift operations which are error-prone in a general programming language such as C. Devil abstracts values as device variables, which are defined as a sequence of bit registers. Device variables are strongly typed in order to detect potential misuses of the device. Possible types are booleans, enumerated types, signed or unsigned integers of various sizes, and ranges or sets of integers. In line 17 of the Busmouse specification, the 5th and 6th bit of the `index_reg` register make up a two-bit unsigned integer variable (*i.e.,* a variable that can take a value from 0 to 3). The `private` attribute means that the `index` variable is not defined in the functional interface of the Busmouse controller and can not be directly accessed by the driver programmer.

```
private variable index = index_reg[6..5] : int(2);
```

Access pre-actions. Device functionalities are often extended by mapping multiple registers to a single physical address. Examples are index-based addressing mode and banks of registers. As a result, accessing such registers requires the setting of a specific context which may involve several I/O operations. To capture this situation, Devil allows pre-actions to be attached to a register. Lines 19 and 20 of the Busmouse specification declare two read-only registers on the same port `base@0`, provided that the variable `index` is set either to `0` or `1` prior to the port access.

```
register x_low  = read base@0, mask '****....',
   pre {index = 0} : bit[8];
register x_high = read base@0, mask '****....',
   pre {index = 1} : bit[8];
```

Register concatenation. Device variables can be spread over several registers. As illustrated by line 25 of the Busmouse specification, constructing the `dx` variable requires concatenation of the two registers `x_high` and `x_low`. The 8-bit variable `dx` is obtained by concatening the four lower bits of register `x_high` with the four lower bits of register `x_low`.

```
variable dx = x_high[3..0] # x_low[3..0], ...
```

Enumerated types. Devil allows defining an enumerated type to abstract the concrete representation of bit values. The symbols `<=`, `=>` and `<=>` define read, write and read-write constraints, respectively. Enumerated types are used to specify the valid values of a device variable. As an example, the `config` variable declaration shown in line 9 of the Busmouse specification declares the two modes (`CONFIGURATION` and `DEFAULT_MODE`) that can be written to the `config` variable.

```
variable config = cr[0] : {
   CONFIGURATION => '1', DEFAULT_MODE  => '0' };
```

Caching and synchronization. Sharing one or more registers between variables induces cache and synchronization problems. When one variable needs to be written independently from the others, the Devil compiler has to determine a value to assign to the other variables. The choice of value depends on whether the access to that variable is idempotent. A Devil variable can be associated with a *behavior* qualifier that specifies the access semantics. No qualifier (the default case) means that the access is idempotent and thus can be redone without side effect; consequently, the variable value can be cached. Such a behavior is often associated with variables that serve as parameters.

A `trigger` behavior means that a write (or read) access to the variable induces a side effect on the controller. Since the side effect cannot be re-done, multiple trigger variables cannot be defined on a register unless a neutral value is provided. Command variables usually have a trigger behavior. The following fragment from an NE2000 Ethernet controller presents examples of the trigger behavior.

```
register cmd = base@0 : bit[8];
variable st  = cmd[1..0],
   write trigger except NEUTRAL;
variable txp = cmd[2],
   write trigger except NOP;
variable rd  = cmd[5..3],
   write trigger except NODMA;
private variable page = cmd[7..6] : int(2);
```

In this example, the register `cmd` is split into four variables. While the `page` variable has an idempotent behavior, the variables `st`, `txp` and `rd` trigger an action when written, except for specific values (`NEUTRAL`, `NOP` and `NODMA`).[2]

Finally, a `volatile` behavior specifies that a read operation is not idempotent; two successive reads may deliver different values. When one needs to get a consistent value of several volatile variables, it is necessary to read them together in one or multiple read operations and cache the result for later use. To do so, Devil allows several variables to be grouped using a `structure`. The use of a structure is demonstrated by the `dx, dy` and `buttons` variables of the Busmouse specification (lines 19 to 22).

```
structure mouse_state = {
   variable dx =
      x_high[3..0] # x_low[3..0], volatile :...
   variable dy =
      y_high[3..0] # y_low[3..0], volatile :...
   variable buttons = y_high[7..5], volatile : ...
};
```

[2]These values are defined using an enumerated type, not shown here.

To access field variables `dy` and `buttons`, the programmer first has to read the `mouse_state` structure. Stubs generated for the structure perform the effective I/O operations, while stubs for the field variables access only the cache. It should be noted that since `dy` and `buttons` share the `y_high` register, `y_high` is read only once. Use of the stubs by the driver programmer is detailed in section 4.1.

Cache and synchronization issues are usually only informally documented by hardware vendors. When programming controllers in a general programming language, cache and synchronization issues are typically solved in an ad-hoc manner that limits code re-use and driver evolution. In fact, the lack of a rigorous description of variable behaviors often leads to laborious testing until the expected functionality is obtained. Also, without specific language support, no verification of the correct usage of variables is possible; this opens opportunities for undetected errors.

Assessment. By clearly defining the semantics of variable behavior, a Devil specification serves as knowledge repository for the correct use of a device. In fact, the driver programmer is guided by the interface generated from the Devil specification. This simplifies driver development and improves re-use. Furthermore, verification is possible at two design stages: (i) on the Devil specification itself so as to check consistency of declarations, (ii) on the correct usage of interface procedures generated by the Devil compiler. These advantages are even more crucial when the device interface is awkward and contorted. The next section presents advanced Devil constructions which permit to handle these situations.

2.2 Advanced Devil

To maximize performance, most modern devices offer a simple, flat interface to registers. However, devices are rarely built from scratch and many of them are evolutions or supersets of previous controllers. For example, today's PCs still rely on DMA, interrupt and graphics controllers that were designed more than twenty years ago.

Design constraints of older devices were guided not only by performance but also by technology and the size of the available I/O address space. Adding functionalities to a device while maintaining backward compatibility induces tricks for addressing additional registers. These issues result in contorted addressing modes, making the programming of such devices even more complex and error-prone. Devil has been specifically targeted towards supporting such devices. Let us now present some of the advanced Devil features using fragments from the Devil specifications of the 8237A DMA, the 8259A interrupt, the Crystal CS4236B, and the IDE controllers.

Register serialization. The 8237A DMA controller provides 16-bit counters through a single 8-bit port. As illustrated by the following example, constructing the counter `x` requires concatenation of the two registers `cnt_high` and `cnt_low`. Since these registers are accessed through the same port, a reading order has to be specified (`cnt_low` then `cnt_high`). Finally, a pre-action attached to `cnt_low` (write any value to the `flip-flop` variable) permits to reset an internal pointer to this register.

```
register cnt_low =
   data, pre {flip_flop = *} : bit[8];
register cnt_high = data : bit[8];
variable x = cnt_high # cnt_low : int(16)
   serialized as {cnt_low; cnt_high};
```

Control-flow based serialization. The 8259A interrupt controller possesses various execution modes that depend on the hardware configuration (processor type, cascaded/single controller) [12]. Initialization of the controller is performed by writing to configuration variables defined over four initialization registers. The initialization sequence varies with the actual values of configuration variables. Additionally, three of the configuration registers (e.g., `icw2, icw3, icw4`) are mapped to a single port and their addressing is implicitly done by previously written configuration values. The following example shows how such an addressing mode can be specified in Devil: configuration variables are grouped together within the `init` structure. Writing variables of this structure into registers is ordered using tests on variable values.

```
register icw1 =
   write base@0, mask '...1....' : bit[8];
register icw2 = write base@1 : bit[8];
register icw3 = write base@1 : bit[8];
register icw4 =
   write base@1, mask '000.....' : bit[8];

structure init = {
   variable sngl = icw1[1] : {
      SINGLE => '1', CASCADED => '0' };
   variable ic4 = icw1[0] : bool;
   ...
   variable microprocessor = icw4[0] : {
      X8086 => '1', MCS80_85 => '0' };
} serialized as {
   icw1;
   icw2;
   if (sngl == SINGLE) icw3;
   if (ic4 == true) icw4;
};
```

Automata based addressing mode. Among the chips we have studied, the Crystal CS4236B sound chip is one of the most complex. This chip is compatible with the Windows Sound System standard [5], but possesses 18 additional registers. These registers are doubly indexed through the `I23` index. Writing a specific device variable converts `I23` from an extended address register into an extended data register. To convert I23 back to an address register, the control register must be written. In order to specify this automata, Devil offers the notion of private variables that are not mapped to a specific register (`xm` in the following example). These variables can be used as memory cells and can be updated when writing a register or a device variable. The code below shows how the extended registers of the CS4236B can be specified using Devil.

```
private variable xm : bool;
register control =
   base@0, set {xm = false} : bit[8];
variable IA = control : int{0..31};

// Indexed Registers I0 - I31
register I(i : int{0..31}) =
   base@1, pre {IA = i} : bit[8];
register I23 = I(23), mask '......0.';

variable ACF = I23[0] : bool;
structure XS = {
   variable XA = I23[2,7..4] : int(5);
   variable XRAE = I23[3], set {xm = XRAE},
      write trigger for true : bool;
};

// Extended Registers X0-X17,X25
register X(j : int{0..17,25}) = base@1,
   pre {XS = {XA=>j; XRAE=>true}} : bit[8];
```

Block transfer. On some processors, such as those of the Pentium family, replacing a

C loop over a variable read/write by a dedicated looping instruction (*e.g., rep* on the Pentium) is often more efficient. Variables with a block transfer usage have to be identified with a `block` keyword. For those variables, the Devil compiler generates two processor-specific block transfer stubs in addition to the single access stubs. The `Ide_data` variable declaration from the IDE specification shown below illustrates the use of the block attribute.

```
variable Ide_data =
   ide_data, trigger, volatile, block : int(16);
```

Other features of Devil are not detailed here. These features include access post-actions, arrays, register constructors and conditional declarations depending on device modes. A complete description of Devil can be found in [17].

3 Property Verification

Devil has been designed to express domain-specific information about the functional interface of devices. Because this information is made explicit, Devil enables a variety of verifications that are beyond the scope of general programming languages. As a result, more errors can be caught earlier in the driver development process. In turn, debugging is easier and less time-consuming. Finally, the robustness of the driver is improved since the programmer has guarantees over the correctness of low-level interactions.

This section summarizes the properties that can be verified both when a Devil description is compiled and when the resulting interface implementation is used.

3.1 Verification of Devil specifications

Due to the declarative nature of the Devil language, it is possible to verify the following properties that ensure the consistency of a specification:

Strong typing. Devil abstractions (*e.g.,* ports, registers, variables) are strongly typed: all uses of these abstractions can be matched against their definition to check type correctness. Types describe usage constraints for registers and variables that are read or write only. Also, various size checks can be performed: the size of data accesses on ports, the size of registers, the size of variables derived from conversion functions, the size of bit masks, and the size of bit patterns that are associated a symbolic name in enumerated types, port ranges, and bit ranges for register fragments.

No omission. All declared entities in a Devil specification must be used at least once. This constraint concerns port arguments in a device declaration, values of ranged port offsets, registers, and register bits (although some bits can be declared irrelevant using bit masks). Read elements of a type mapping must be exhaustive. Also, a type for reading (as well as possibly writing) must be used with a readable variable. The same holds for writing.

No double definition. All entities in a Devil specification must be declared at most once. This constraint concerns port arguments in a device declaration, ports, registers, types, symbolic names and bit patterns in enumerated types and variables.

No overlapping definitions. Port and register descriptions must not overlap. More precisely, each port must appear only once in the register definitions, except when registers are defined using disjoint pre-actions or masks. However, the same port may be used for reading from one register and writing to another. No bit of a single register can be used in the definition of two different variables.

3.2 Verification of interface usage

Verification of the correct usage of the generated interface can be both static and dynamic. In the latter case, run-time checks are optionally included in the code for debugging purposes.

When writing to a variable, a check can be performed to verify that the written value falls within the range specified by the variable type. If the value is constant, the check can generally be done at compile time. However, because the type system of C is not powerful enough to express all Devil types, not all such verifications can be implemented at compile time. In this situation, checks have to be implemented in debug mode using run-time checks. Finally, run-time checks can optionally be generated after variable reads. Such checks are useful for verifying that a device behaves accordingly to its Devil specification.

Our experience in re-engineering drivers showed that dynamic checks allow the early detection of usage errors, preventing them from becoming insidious bugs. This is particularly valuable for kernel-mode drivers, which are tricky to step through with a debugger. Moreover, since the checks are automatically and systematically inserted and removed by the compiler, their use is easy and safe.

4 Comparison with Hand-Crafted Drivers

To assess our approach, we now compare the use of Devil and C. First, we analyse issues related to code development. Then, we report on a study based on mutation analysis to evaluate the robustness of Devil and C implementations. Finally, we discuss the performance of drivers that use the C library automatically generated from a Devil specification.

4.1 Driver development

To illustrate the benefits of Devil in terms of separation of concerns and readability, we compare a fragment of the original C implementation of the Logitech Busmouse driver (see Figure 2) with the use of the interface (see Figure 3) generated from the equivalent Devil specification.

In a traditional C driver, the programmer writes code that accesses the device with assembly-language-level operations (*e.g.,* bit manipulations). For example, the C code needed to express the concatenation of the four lower bits of registers `y_high` and `y_low` is tedious. As shown in Figure 2-a, macros are often defined so as to factorize common expressions or associate names with commands. Nevertheless, it is rather difficult to understand the behavior of the device from the implementation; maintenance of this code is error-prone and not easy.

Using Devil, driver development is a two stage process: first the chip is specified in Devil, then code is written using the stubs generated from the specification. *Describing* the device as opposed to *coding* improves readability. For instance, the Devil description of the variable `dy` in the Busmouse specification (see line 26 of Figure 1) consists of a straightforward concatenation of two bit-fragments. The Devil specification is so close to a device description that it can be used for documentation purposes.

When writing the driver code, the programmer first has to include Devil-generated stubs and to speficy configuration information. For instance, in Figure 3-a, Busmouse stubs are used in debug mode and in a single device configuration (`#define DEVIL_NO_REF`). Further communication with the device is encapsulated in stubs (see Figure 3-b). Therefore, the driver programmer only has to focus on operating the device using abstract values. Writing the hardware operating code becomes a very simple task, especially if the programmer can use an existing Devil specification.

4.2 Robustness

As discussed in Section 3, Devil exposes properties that can be automatically checked. This section evaluates the benefits of these checks in terms of software robustness.

Detecting bugs as early as possible is crucial during the development process. A study by DeMillo and Mathur found that simple errors (*e.g.,* typographic errors, inattention errors) represent a significant fraction, though not the majority, of the errors in production programs. This study also revealed that such errors can remain hidden for a long time. Even though their study was concerned with the development of TeX, which differs from device drivers, these observations remain pertinent, and are even more important considering the permissive nature of a language such as C, es-

```
#define MSE_DATA_PORT          0x23c
#define MSE_CONTROL_PORT       0x23e
...
#define MSE_READ_Y_LOW         0xc0
#define MSE_READ_Y_HIGH        0xe0
```

2a. Macro definition

```
dy = (inb(MSE_DATA_PORT) & 0xf);
outb(MSE_READ_Y_HIGH, MSE_CONTROL_PORT);
buttons = inb(MSE_DATA_PORT);
dy |= (buttons & 0xf) << 4;
buttons = ((buttons >> 5) & 0x07);
```

2b. Macro usage

Figure 2: Fragment of the original Linux driver for the Logitech Busmouse

```
#define DEVIL_NO_REF
#define dev_name bm
#define DEVIL_DEBUG
#include "busmouse.dil.h"
```

3a. Interface usage

```
bm_get_mouse_state();
dy = bm_get_dy();
buttons = bm_get_buttons();
```

3b. Stub usage

```
#define bm_get_mouse_state() ( \
  outb(1, bm_cache.__dil_base__+2); bm_cache.cache_mouse_state.cache_get_x_high = inb(bm_cache.__dil_base__); \
  outb(0, bm_cache.__dil_base__+2); bm_cache.cache_mouse_state.cache_get_x_low  = inb(bm_cache.__dil_base__); \
  outb(3, bm_cache.__dil_base__+2); bm_cache.cache_mouse_state.cache_get_y_high = inb(bm_cache.__dil_base__); \
  outb(2, bm_cache.__dil_base__+2); bm_cache.cache_mouse_state.cache_get_y_low  = inb(bm_cache.__dil_base__) )

#define bm_get_dy() ( \
  (bm_cache.cache_mouse_state.cache_get_y_high & 0xfu) << 4 | bm_cache.cache_mouse_state.cache_get_y_low & 0xfu)

#define bm_get_buttons() ((bm_cache.cache_mouse_state.cache_get_y_high & 0xe0u) >> 5)
```

3c. Generated stubs (after inlining)

Figure 3: Fragment of the Devil based driver for the Logitech Busmouse

pecially when used to write low-level code.

In order to evaluate the impact of Devil on driver robustness, we have estimated the number of errors that can be detected automatically by the C and Devil compilers/checkers.[3] The *error-detection coverage* is computed using a mutation analysis technique [1, 8].

For a program P, mutation analysis produces a set of alternate programs, each generated by modifying a single statement of P, according to mutation rules. In our experiment, the mutation rules introduce errors in operators, identifiers and literal constants. Such errors are generated by inserting, replacing or removing a character from the targeted token. For example, the logical operator || can be replaced by the bit operator |, the number 121 can be replaced by 21, etc. Mutation rules are defined so as to ensure that the resulting mutant is syntactically correct, and actually modifies the semantics of the program. Therefore, detection of the mutation introduced error by the compiler occurs only if the mutant violates a property of the language (*e.g.,* C or Devil).

In a C driver, we are only interested in testing the hardware operating code. Accordingly, we manually insert tags to mark the corresponding regions in the original C code, and only apply mutations to the tagged regions. In a Devil-based driver, mutations have to be applied both to the Devil specification of the device, and to procedure calls to the generated interface (this C code is denoted by C_{Devil} in the rest of the paper).

Our experiments compare the error-detection coverage of C against the error-detection coverages of the Devil specification and C_{Devil}. It should be noted that our measurements reflect the worst case for Devil for the following reasons. First, the mutation rules for C and Devil have been chosen so that C is always favored. Second, since a driver often uses a subset of a device, the Devil specification offers more mutation sites (possible errors) than the original C driver. Finally, Devil specifications should ideally come from the device manufacturer or widely

[3] In our current experiments, the benefit of run-time checks in Devil generated interfaces are not taken into account.

Device	Language	lines	Number of mutation sites	Mutants per site	Undetected mutants per site	Mutation Sites with undetected mutants	Ratio to C
Logitech Busmouse	C	36	62	36.6	26.8	**45.3**	-
	Devil	21	81	15.9	0.2	**1.0**	-
	C_{Devil}	18	21	13.5	5.0	**7.7**	**5.9**
	Devil+C_{Devil}		102	15.4	1.2	**8.7**	**5.2**
IDE (Intel PIIX4)	C	64	95	29.0	18.8	**61.8**	-
	Devil	127	277	17.1	1.6	**26.6**	-
	C_{Devil}	81	42	22.6	7.4	**13.3**	**4.6**
	Devil+C_{Devil}		319	17.5	2.0	**39.9**	**1.6**
Ethernet (NE2000)	C	204	247	14.7	12.6	**212.4**	-
	Devil	144	456	15.0	1.1	**33.7**	-
	C_{Devil}	137	258	48.7	12.5	**66.1**	**3.2**
	Devil+C_{Devil}		714	27.2	4.7	**99.8**	**2.1**

Table 1: Language Error-Detection Coverage Analysis

available public-domain libraries. Thus, one can expect them to be bug-free and errors only to appear in C_{Devil}.

Measurement analysis. Our study focuses on three different devices (*e.g.,* Logitech Busmouse, NE2000 Ethernet, and IDE controllers) and their corresponding Linux 2.2-12 drivers. Table 1 presents the results of the mutation analysis. Overall, the experiments show that the probability of undetected errors is 1.6 to 5.2 times higher in C hand-crafted drivers than in Devil-based driver (Devil + C_{Devil}). When comparing C to C_{Devil} only (assuming that the specification is correct), the propensity of undetected errors 3.2 to 5.9 times higher in C. Finally, it can also be observed that mutation errors in Devil specifications are nearly always detected.

The first column of Table 1 represents the number of possible mutation sites (s). The second column shows the number of mutants (*i.e.,* errors) which can be injected for each site (m_s). For example, given an integer of two digits in base ten, 50 mutants can be generated (2 for removing a digit, 30 for inserting a new digit, and 18 for replacing a digit). The third column shows, for each mutation site, the number of mutants not detected by the compiler/checker (um_s).

To enable the comparison between C, Devil and C_{Devil} we are interested in measuring the number of mutation sites that have undetected mutants (s_{um}). To compute this value, we have to balance the number of undetected mutants per site by the number of mutation sites ($s_{um} = um_s/m_s * s$). For example, consider the Logitech Busmouse C driver. It has 62 mutation sites. For each site, 36.6 mutants are generated (on average) and 26.8 are not detected by the compiler. This give us 45.3 sites with undetected mutants.

4.3 Performance

It is well-recognized that the performance of drivers is critical for the overall system performance. Furthermore, as demonstrated by Thekkath and Levy for high-performance RPCs [18], the performance of the hardware operating code has a significant impact on the overall driver performance. While Devil can improve readability and robustness of driver hardware operating code, its usefulness depends on the efficiency of the generated code: using Devil must not induce significant execution overhead.

In order to evaluate the benefit and impact of Devil on driver development, we are reengineering various Linux drivers and testing them on a bi-processor PC.[4] Among the drivers and devices in a Unix system, we chose to implement first the IDE and the accelerated X11 drivers for two reasons: (i) they are representative of performance intensive drivers and they illustrate totally different device access behavior.

In the rest of this section, we first identify

[4]The PC is a DELL Precision 210 with the following configuration: two Pentium II 450 MHz, Intel PIIX4 PCI chipset, Maxtor model 91000D8 UDMA2 19.5Gb disk with 512Kb cache, 3Dlabs Permedia2 graphic controller.

Transfer mode	Sectors per interrupt	I/O Size in bits	Standard driver: I/O Operations	Standard driver: Throughput in Mb/s	Devil driver: I/O Operations	Devil driver: Throughput in Mb/s	Devil/Stand. throughput ratio
DMA	-	-	14	14.25	20	14.25	**100 %**
PIO	16	32	$7 + \frac{\#s(1+128)}{16}$	8.17	$10 + \frac{\#s(3+128)}{16}$	7.36	**90 %**
		16	$7 + \frac{\#s(1+256)}{16}$	4.45	$10 + \frac{\#s(3+256)}{16}$	3.94	**88 %**
	8	32	$7 + \frac{\#s(1+128)}{8}$	8.09	$10 + \frac{\#s(3+128)}{8}$	7.28	**89 %**
		16	$7 + \frac{\#s(1+256)}{8}$	4.42	$10 + \frac{\#s(3+256)}{8}$	3.91	**88 %**
	1	32	$7 + \#s(1 + 128)$	6.93	$10 + \#s(3 + 128)$	6.36	**91 %**
		16	$7 + \#s(1 + 256)$	4.06	$10 + \#s(3 + 256)$	3.63	**89 %**

Table 2: IDE Linux driver comparative performance results (using C loops)

the possible penalties induced by Devil, and then we compare the performance of the IDE and accelerated X11 Devil-based drivers with the original ones.

Micro-analysis Interface procedures generated by the Devil compiler contain I/O as well as bit-shift and bit-mask instructions. These procedures are optimized by the Devil compiler and implemented as pre-processor macros or inlined functions. Therefore, there is no execution overhead for a single Devil interface procedure as compared to hand-crafted C instructions.

In one situation, we observed that Devil could induce an execution penalty. Accessing independent device variables (*i.e.,* variables not grouped in a structure) defined over a single register, requires multiple Devil interface calls. Each additional call induces additional I/O, as compared to an hand-crafted driver. Nevertheless, as we found in our re-engineering of the IDE and Permedia2 driver, such variables are often parameters and rarely affect the performance of the critical path.

IDE driver Table 2 compares the performance of a Devil-based IDE driver with that of the original C driver. IDE throughput measurements were obtained using the standard Linux `hdparm` utility. We wrote two Devil specifications for this driver: a specification of the IDE controller and a specification of the Intel PIIX4 PCI busmaster IDE.

We have run the IDE driver in both Ultra DMA-2 and several PIO modes, varying the size of I/O (16 or 32 bits) and the number of sectors transfered per interrupt. In DMA mode, Devil induces 6 additional I/O operations to prepare the command. Because of the long duration of the DMA transfer, there is no impact on the available throughput. In the PIO modes, there are 3 additional I/O operations to prepare the command, plus 2 for each interrupt (`#s` denotes the total number of sectors accessed). When using a C loop over a single variable read, we measured a 10% throughput penalty. When using block transfer stubs that use a *rep* instruction, we did not observe an impact on the available throughput.

Permedia2 X11 driver Tables 3 and 4 show the performance Devil-based X11 driver for the 3Dlabs Permedia2 graphics controller. Throughput measurements were obtained using the `xbench` utility. We have modified the 3Dlabs X11 server, which is based on a Xfree86-3.3.6 implementation. Although the Permedia2 chip provides acceleration for both 2D and 3D, the X11 server does not support 3D operations. Additionally, to minimize device-dependant code, many 2D primitives are implemented in software in Xfree86. In fact, hardware acceleration is only used for implementing the `fill rectangle` and `screen area copy` primitives.

Unlike many I/O devices, the Permedia2 controller maps registers into the memory address space. In fact, processor accesses are decoded by the controller and stored in a FIFO. Before accessing the chip, the driver must wait for free entries in the FIFO. This wait loop in-

Display Mode (bits/pixel)	Rectangle Size (pixels)	Standard Driver		Devil Driver		Devil/Stand. Throughput Ratio
		I/O Operations	Throughput (rect./s)	I/O Operations	Throughput (rect./s)	
8	2x2	$3(\#w) + 15$	984838	$3(\#w) + 17$	949052	**96 %**
	10x10		589621		585350	**99 %**
	100x100		38472		38438	**100 %**
	400x400		3762		3762	**100 %**
16	2x2	$3(\#w) + 15$	982338	$3(\#w) + 17$	945916	**96 %**
	10x10		333670		332499	**100 %**
	100x100		21022		21033	**100 %**
	400x400		2221		2221	**100 %**
24	2x2	$2(\#w) + 10$	978605	$2(\#w) + 10$	945884	**97 %**
	10x10		235119		234716	**100 %**
	100x100		3693		3693	**100 %**
	400x400		244		243	**100 %**
32	2x2	$3(\#w) + 15$	957534	$3(\#w) + 17$	929833	**97 %**
	10x10		251522		251584	**100 %**
	100x100		10466		10466	**100 %**
	400x400		899		899	**100 %**

Table 3: Comparative Performance of Permedia2 Xfree86 Driver: Rectangle Test

Display Mode (bits/pixel)	Copy Size (pixels)	Standard Driver		Devil Driver		Devil/Stand. Throughput Ratio
		I/O Operations	Throughput (copies/s)	I/O Operations	Throughput (copies/s)	
8	2x2	$3(\#w) + 15$	149553	$3(\#w) + 17$	144494	**97 %**
	10x10		123584		122300	**99 %**
	100x100		10662		10638	**100 %**
	400x400		764		764	**100 %**
16	2x2	$3(\#w) + 15$	145084	$3(\#w) + 17$	136755	**94 %**
	10x10		85994		85561	**99 %**
	100x100		3502		3512	**100 %**
	400x400		238		238	**100 %**
24	2x2	$2(\#w) + 9$	144385	$2(\#w) + 9$	144521	**100 %**
	10x10		77443		77605	**100 %**
	100x100		1716		1716	**100 %**
	400x400		114		114	**100 %**
32	2x2	$2(\#w) + 9$	142335	$2(\#w) + 9$	142598	**100 %**
	10x10		69762		69804	**100 %**
	100x100		1703		1701	**100 %**
	400x400		111		111	**100 %**

Table 4: Comparative Performance of Permedia2 Xfree86 Driver: Screen Copy Test

duces one I/O operation per iteration. In Tables 3 and 4, $\#w$ denotes the number of iterations per wait loop. In the driver we modified, 2 or 3 wait loops are performed per primitive call.

The time for execution of a drawing command by the Permedia2 controller is proportional to the number of drawn pixels and their depth. Therefore, the overhead induced by Devil is more perceptible for shortest commands. The worst case is reached for 2x2 pixel commands in 8 or 16 bit mode, where Devil induces a performance penalty of up to 6%. For primitive calls involving more than 100 pixels (which are the most common in practice), 99% to 100% of the performance of the original server is obtained (always 100% in 24 bit mode).

5 Related Work

Our work on device drivers started with a study of graphic display adaptors for a X11 server. We developed a language, called GAL, aimed at specifying device drivers in this context [19]. Although successful as a proof of concept, GAL covered a very restricted domain.

The goal of the UDI project[5] is to make device drivers source-portable across OS platforms. To do so, they have normalized the API between the OS and the lower part of device drivers [14]. Besides showing the timeliness of our work, UDI focuses only on the high-level

[5]The UDI (Uniform Driver Interface) project is the result of a collaboration of several computer companies including Compaq, HP and IBM.

part of drivers and their interaction with the OS.

Windows-specific driver generators like BlueWater System's WinDK [4] and NuMega's DriverWorks [6] provide a graphical interface for specifying the main features of a driver. They produce a driver skeleton that consists of invocations of coarse-grained library functions. To our knowledge, no existing driver generators cover the communication with the device.

Languages for specifying digital circuits and systems have existed for many years. The VHDL standard [11], widely used in this domain, is one of the most expressive. It addresses several aspects of chip design such as documentation, simulation and synthesis. VHDL provides both high-level and low-level abstractions: arrays and loops are supported, as well as bit-vector literals and bit extraction. However, all VHDL abstractions focus on the inner workings of circuits, not their high-level programming interface. As a consequence, chip interfaces are not explicitly denoted, and VHDL compilers perform limited consistency checks. Interestingly, VHDL allows attaching arbitrary strings to variables. Using them to add interface-specific information is possible, but would require a normalized syntax and compiler support, which in some way amounts to embedding Devil concepts in VHDL.

The New Jersey Machine-Code Toolkit [15] helps programmers write applications that process machine code at an assembly-language level of abstraction. Guided by a instruction set specification, the toolkit generates the code for reading or generating binary. Some simple verifications iare also done at the specification level.

6 Conclusion and Future Work

This paper has presented a new approach to developing hardware operating code that is based on an IDL named Devil. This IDL enables hardware communication to be described using high-level, domain-specific constructs instead of being written with assembly-language-like operations. Raising the implementation level of this layer of a device driver dramatically reduces the risk of errors. Devil has shown to be expressive enough to specify a wide variety of devices such as the DMA, interrupt, Ethernet, IDE disk, sound, mouse and video controllers.

Because Devil significantly raises the level of abstraction of communication with the hardware, Devil specifications are more readable, maintainable and re-usable than equivalent C code.

We have developed a compiler that checks the consistency of a Devil specification and automatically generates low-level code that is mostly comparable to hand-crafted code. We have assessed our approach by conducting experiments aimed at comparing hardware operating code in C or Devil for robustness and performance. We have demonstrated that our approach enables hardware operating code to be more robust than C, with mostly comparable performance.

Our future work aims to improve the performance of the output of our Devil compiler. Specifically, we want to enhance performance by factorizing and scheduling device communications and by better exploiting special-purpose assembly-level instructions. The key advantage of introducing optimizations at the compiler level is that these advanced techniques are transparently available to any Devil programmer. As a result, our work reduces the need to have a highly experienced programmer to write hardware operating code since part of this expertise is captured by the compiler.

We are currently building a public domain library of Devil specifications for common devices such as those found in PCs. Our purpose is to setup a WWW repository that would help dissemination of expertise about hardware and facilitate the development of device drivers.

Acknowledgment.

We thank Julia Lawall from DIKU and the other members of the Compose group for helpful comments on earlier versions of this paper. We also thank Timothy Roscoe and the anonymous reviewers for their valuable inputs.

This work has been partly supported by France Telecom under the CTI contract 991B726, the French Ministry of Research and Technology under the Phenix contract 99S0362, and the French Ministry of Education and Research.

Availability

The Devil compiler, Devil specifications and Devil-based drivers mentioned in the paper are available at the following web page `http://www.irisa.fr/compose/devil`.

References

[1] A. T. Agree, T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Mutation analysis. Technical Report GIT-ICS-79/08, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, USA, September 1979.

[2] B.N. Bershad, T.E. Anderson, E.D. Lazowska, and H.M. Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems*, 8(1):37–55, February 1990.

[3] A. Birrell and B. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[4] BlueWater Systems, Inc. *WinDK Users Manual*. URL: www.bluewatersystems.com.

[5] Cirrus Logic, Inc, P.O. Box 17847, Austin, TX 78760. *CrystalClear^{TM} Single Chip Audio System (CS4236B)*, September 1997. URL: www.cirrus.com.

[6] Compuware NuMega. *DriverWorks User's Guide*. URL: www.numega.com.

[7] E. N. Dekker and J. M. Newcomer. *Developing Windows NT device drivers : A programmer's handbook*. Addison-Wesley, first edition, March 1999.

[8] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: help for the practicing programmer. *Computer*, 11(4):34–41, April 1978.

[9] R. Draves, M. Jones, and M. Thompson. *MIG - The MACH Interface Generator*. School of Computer Science, Carnegie Mellon University, July 1989.

[10] E. Eide, K. Frei, B. Ford, J. Lepreau, and G. Lindstrom. Flick: A flexible, optimizing IDL compiler. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 44–56, Las Vegas, NV, USA, June 15–18, 1997.

[11] IEEE Standards. *1076-1993 Standard VHDL Language Reference Manual*, 1994. URL: standards.ieee.org.

[12] H. P. Messmer. *The Indispensable PC Hardware Book*. Addison-Wesley, third edition, 1997. page 669, figure 26.6.

[13] S. O'Malley, T. Proebsting, and A.B. Montz. USC: A universal stub compiler. In *Proceedings of Conference on Communication Architectures, Protocols and Applications*, London (UK), September 1994.

[14] Project UDI. *UDI Specifications, Version 1.0*, September 1999. URL: www.project-udi.org.

[15] Norman Ramsey and Mary F. Fernandez. The new jersey machine-code toolkit. In *Proceedings of the Winter USENIX Conference*, New Orleans, LA, January 1995.

[16] A. Rubini. *Linux Device Drivers*. O'Reilly, first edition, February 1998.

[17] L. Réveillère, F. Mérillon, C. Consel, R. Marlet, and G. Muller. The Devil language. Research Report 1319, IRISA, Rennes, France, May 2000.

[18] C.A. Thekkath and H.M. Levy. Limits to low-latency communication on high-speed networks. *ACM Transactions on Computer Systems*, 11(2):179–203, May 1993.

[19] 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.