HAIL: A Language for Easy and Correct Device Access

Jun Sun, Wanghong Yuan, Mahesh Kallahalla, and Nayeem Islam
DoCoMo Communication Laboratories USA, Inc.
San Jose, CA 95110
+1-408-573-1050
{ jsun, yuan, kallahalla, nayeem}@docomolabs-usa.com

ABSTRACT
It is difficult to write device drivers. One factor is that writing low-level code for accessing devices and manipulating their registers is tedious and error-prone. For many system-on-chip based systems, buggy hardware, imprecise documentation, and code reuse worsen the situation further. This paper presents *HAIL (Hardware Access Interface Language)*, a language-based approach to simplify device access programming and generate error checking code against bugs in software, hardware, and documentation. HAIL is a domain-specific language that specifies all aspects of a device's programming interface and the access methods in a particular system and OS. A compiler automatically checks the specification and translates it into C code for device access, with optional debugging code. The generated code can be included directly into device driver code. In the paper, we argue that HAIL lowers development effort, incurs minimal runtime overhead, and reduces device access related bugs. We also show that the HAIL specification can be reused for different operating systems, thereby reducing porting costs.

Categories and Subject Descriptors
D.2.4 [**Software Engineering]:** Software/Program Verification. D.3.2 [**Programming Languages**]: Language Classifications – *domain-specific languages.* D.4.4 [**Operating Systems**]: Communications Management – *device access in drivers.*

General Terms
Design, Languages, Verification.

Keywords
Device drivers, domain-specific languages, automatic code generation, register access, embedded systems, system-on-chip, invariant specification and verification, software reuse.

1. Introduction
Device drivers are notoriously difficult to write and often cause operating system failures [1]. Many factors contribute to this, including interrupt handling, buffer management, synchronization, and timing constraints. In particular, simply accessing the device and manipulating its registers involves a significant amount of work and is an inviting ground for bugs.

Today programmers typically scan through device documentations and define a long list of macros for various parts of the device interface such as register names and offsets, bit field masks, and enumeration of valid values. Hard-coded offsets and bit-manipulations based on these macros are used to access the device. This programming style is cumbersome and error prone. Defining macros and performing register bit manipulation represent a significant amount of development effort. For example, in the Linux 2.6.9 kernel, the e1000 driver for Intel eepro1000 devotes 14% of lines of code to macro definitions and 8% to bit manipulations. In general, for low-level device driver code we typically see anywhere from 15% to 25% of code devoted to macro definitions and bit manipulations.

Bit manipulations are inherently error-prone due to the mismatch between the logic operation unit and the software manipulation unit. Bit fields within a register represent the logic operation units of the device. The software manipulation unit is, however, the whole register because there is no way to address individual bits. This disparity adds unwelcome complexities during the flow of programming. For example, to manipulate bit fields in a register, one generally uses operations such as masking, shifting, and OR'ing, which do not directly reflect the underlying logic of the program.

In many embedded systems, where new chips and system-on-chip style are widely used, the situation becomes worse for two reasons:

- First, the hardware and documentation of new chips often have bugs. Prudent programmers frequently use assert statements to verify the hardware state, the document specification, and their understanding of the document. However, there is no a systematic approach to do this. Programmers often subconsciously trade off between the benefit and the overhead of writing additional checks.
- Second, with system-on-chips widely used and reused in embedded systems, the same IP core can be used on many chips and many more systems; a device can have a different base address in one bus or even live on a totally different bus. For example the SMC91C111 chip [2] is used by at least nine different SoC development boards. As a result of such reuse and subtle differences in reuse, the programming effort often multiplies; #ifdef statements are used which leads to unmaintainable code.

To address the second problem, programmers typically abstract the access code and adapt it to individual systems. Such an abstraction for device access, however, is driver-specific and hence cannot be easily shared with another device on the same bus. It is also difficult, even impossible sometimes, to abstract device access for future versions of the same chip and for future buses the device may attach to.

In this paper we present a domain-specific language, called HAIL (Hardware Access Interface Language), that simplifies device access and provides guards against access-related bugs in software, hardware or documentation. HAIL allows programmers to specify devices and their associated buses in a structured descriptive language that mostly consist of attribute/value pairs. A complete specification consists of four parts: register map, invariants, address space, and device instantiation. The first two parts describe device registers and constraints on their access, the third part describes one or more buses the device is attached to, and the last part describes an instance of the device by associating it with the bus. The decomposition of the specification into four parts improves the reuse of the specification. For example, the same device can live in different address spaces while multiple devices can share the same address space. The HAIL compiler then checks for the consistency of the speciation and translates the specification into C code for accessing registers and their bits. The compiler also generates optional debugging code that checks for any violation of device-access constraints at run-time.

HAIL contributes to advancing the domain-specific language approach of synthesizing hardware access code, especially for embedded systems. Specifically HAIL has three main unique features.

- HAIL decomposes hardware access specification into four orthogonal parts, not only making it possible for multiple parties to supply the specification but also increasing the reuse of the specification. The specification reuse is easier and safer than code reuse.
- HAIL introduces invariant specification and generates run-time debugging code for a wide variety of software bugs and hardware bugs.
- HAIL's syntax and semantics are strongly motivated by and closely mirror typical hardware specification, which should lower the effort of adoption and usage.

The rest of the paper is organized as follow. In next section, we compare HAIL with related work. HAIL is presented and detailed in Section 3, followed by a case study in Section 4. Section 5 presents some additional features of HAIL. Section 6 describes the experimental evaluation. Finally, Section 7 concludes this paper.

2. Related Work

A device driver has three logical programming interfaces: hardware access, kernel service access, and driver service export to applications or kernel. The driver code encodes two run-time models: the device hardware run-time model and the OS driver run-time model. A very ambitious approach towards easy and correct driver programming is to synthesize the driver completely from a specification, encompassing all the above programming and run-time aspects. Wang et al. [3] proposed a formal specification of device behavior so that a complete OS-aware driver can be synthesized. Conway et al. [4] devised a high-level domain-specific language for driver development. Both pieces of work assume device access through memory-mapped IO at a fixed or variable base address. In the preliminary results, many aspects of programming interfaces and run-time models are simplified in order to facilitate implementation.

Previous work on hardware/software co-design focuses on generating functional-level device access routines for software and VHDL specification for hardware [5]. This approach is typically used for ASIC design, where the actual register-level device access is trivialized. By contrast, HAIL focuses solely on generating device access code at the register (or bit-field) level and addresses the handling of much more complex bus structures.

In many regards HAIL is similar to Devil, which also translates high-level device description into register-level device access code [6, 7]. HAIL, however, differs from Devil in four ways. First, Devil targets PC systems, while HAIL targets embedded systems, which make wider use of system-on-chips and have more varieties of buses. Second, a Devil specification is programming oriented and uses C style description, while a HAIL specification is hardware attribute description oriented. Third, HAIL captures additional important attributes of registers such as fixed values for read/write and read with side effect. Finally and more importantly, HAIL introduces a formal invariant specification that generates run-time debugging code to catch a wide variety of software and hardware bugs.

Commercial tools exist to aid driver programming, especially for device access. Examples are WinDriver and KernelDriver [8] and DriverStudio [9]. These products usually target one particular OS and one class of devices (such as PCI devices). While they facilitate driver development for the intended applications, they don't apply for general driver programming.

HAIL uses invariant specification to express inherent constraints in device access and device state. HAIL compiler automatically generates run-time debugging code to ensure that invariants are preserved at run-time. This approach is related to defensive programming research such as formal method for assertion by Rosenblum [10] and contract checking wrappers by Edwards et al. [11].

HAIL is also remotely related to code verification in simulated run-time environments [12, 13, 14]. In both cases the generated run-time debugging code performs consistency check against a run-time model or a valid state transition machine. The latter work, however, typically use hand-crafted code, instead of invariant specification, to perform more extensive static analysis and check for more complex and more specific constraints.

3. Hail Specification

3.1 Overview

HAIL specification consists of four parts: (1) register map description, which describes the various device registers and bit fields, (2) address space description, which describes the mechanisms for accessing a bus, (3) device instantiation, which describes the actual instantiation of the device in the particular system, and (4) invariant specification, which describes the constraints on accessing the device.

The register map description and invariant specification in HAIL specification are usually translated from device documentation directly. The address space specification depends on the system /CPU architecture and sometimes the OS. Device instantiation is tied to the specific driver's needs.

A HAIL compiler translates the specification into C code for accessing registers. In a simplified view, the generated code is a list of get_xxx()/set_xxx() for all registers and bit fields, put together in a C header file. Optionally the generated code can also have run-time debugging code that catches any violations of the specified invariants.

The device driver includes the generated header file and uses the get_xxx()/set_xxx() function to access the device registers, without concerning about the underlying device base addresses, bus attributes, data width, endianess and etc.. Additionally, the device driver can manipulate bit fields directly using these functions and hence avoid many explicit bit manipulations.

Figure 1 illustrates the HAIL approach. In the next subsections, we describe each of the four parts of HAIL specification. In Section 4, we present a case study of HAIL.

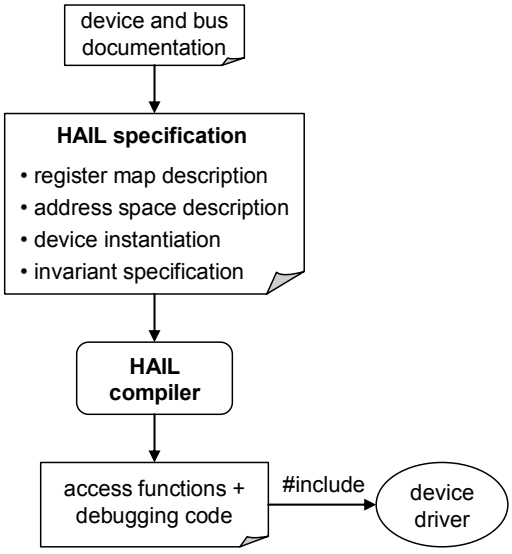

Figure 1 - Overview of HAIL

3.2 Register map description

Registers are the software interface of a device. A *register map* is composed of a set of logically coherent registers that reside on the same bus. A device can have one or more register maps. For example, a typical PCI device has three register maps, one each for PCI configuration, PCI memory, and PCI IO space.

Each register consists of a set of bit fields. In register map description both register and bit field have a list of attributes such as name, size, and accessibility (read-only, write-only, or read-write). A readable register or bit field can be *static, volatile* (i.e., two adjacent reads may return different values), *volatile_se* (volatile with side effect for reading), *fixed* (always the same return value), or *dont_matter* (as in reserved bits). A writable bit field may require a default value to write when its neighboring fields are modified. This attribution is called *default_write,* and it can be *fixed, dont_change* (i.e., the value of this bit should be preserved), or *no_default.*

In general, register map description can be easily translated from the device documentation. For example, Figure 2 shows a segment document for register LCR (line control register) in universal asynchronous receiver/transmitter (UART) device [15], and Figure 3 shows its corresponding description in HAIL.

LCR Bit Definitions

Physical Address 0xBASE_000C

Bits	Access	Name	Description
31:8	——	——	reserved
7	R/W	DLAB	Divisor Latch Access 0 = access TX, RX, and IER 1 = access DLL and DLM
…			
1:0	R/W	WLS	Word Length Select 0b00 = 5-bit character 0b01 = 6-bit character 0b10 = 7-bit character 0b11 = 8-bit character

Figure 2 - Documentation for Register LCR [16]

```
.=0x000C { name=LCR; access=RW; size=4;
    [31:8]: reserved;
    [7]   : name=DLAB;
    ...
    [1:0] : name=WLS;
            enum={WLEN5=0b00,WLEN6=0b01,
                  WLEN7=0b10, WLEN8=0b11};
}
```

Figure 3 – HAIL Specification for Register LCR

Note that in Figure 3 we do not have all attributes for the register and bit fields (such as read_value and default_write). HAIL allows one to specify a set of default attributes. Any unspecified attributes will implicitly take the default values.

The register map description is only related to the device itself and is independent of the bus, system and OS. Therefore, we envision that the device vendor writes the register map description and distributes it along with the device documentation

3.3 Address space description

In a system a device is always attached to a bus, or *address space*, which dictates how the CPU can access the device registers. The CPU accesses an address space through certain instructions. A CPU with a memory management unit (MMU) can access the CPU virtual space through regular memory read and write instructions. Some CPUs can also directly access additional address spaces. For example, in the i386 architecture, the CPU can access an IO space through special instructions such as inb and outb; in the Sparc64 architecture, the CPU can directly access 256 different spaces.

If a device is attached to an address space that is not directly accessible to the CPU, the address space must be mapped into one that is. The most common method is to map a window in such an address space to the CPU virtual space. Suppose a window `[x,x+s]` is mapped at base address, `b`, and the register in the

space has an offset, y, where x<=y<x+s, then the CPU can read and write the register through *(unsigned int *)(b+y-x).

In HAIL, the address space description is separated from the device description (i.e., register maps of the device). As a result, devices on the same bus can share the same address space description. In fact the system vendor or the OS porting engineer can provide the address space description for the whole system before driver development starts.

Further, in HAIL address spaces and mappings between them are specified separately. An address space has attributes such as name, supported data width, and endian-ness that are independent of the mapping. For address space mapping, we define a list of attributes including name, base address, mapping window, and endian swapping. Figure 4 illustrates the relationship among devices, address space, and space mapping in a simple one-level mapping scenario. In more general and complex cases, there can be multiple levels of mapping in order for the CPU to access an remote address space.

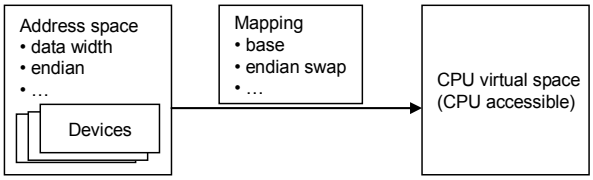

Figure 4 - Address space and space mapping

The separation of address space description and space mappings not only makes specification simpler but also makes it more flexible and expressive. For example, consider a MIPS system running in big endian mode with a little endian PCI bus. If the host-PCI controller does automatic endian swapping, the mapping will have an endian swapping attribute. As a result HAIL will not generate endian swapping code in register access functions. If the host-PCI controller does not perform endian swapping, a one-line change in the HAIL specification will yield the correct code with proper endian swapping.

In addition to memory mapped address spaces, HAIL also supports *gated spaces.* A typical such space is usually accessed through a pair of address/data registers such as many implementations of i2c bus. For gated space, HAIL resorts to a set of externally defined read/write functions.

Sometimes it is useful to treat a memory mapped space as a gated space. For example, PCI configuration space is typically mapped into CPU virtual space. However, Linux already defines a set of functions for accessing PCI configuration registers. This makes it preferable to specify the PCI configuration space as a gated space and supply the proper wrapper functions to call the underlying Linux functions.

3.4 Device instantiation

Device instantiation defines an instance of a device by associating every device's register map to a specific address space. When associating a register map with an address space, the instantiation specifies a base address for the register map (which can be a statically fixed address), a literal (which is defined as a macro or a variable by the driver or OS environment), or a HAIL variable (which needs to be set by driver at run-time).

Figure 5 shows the instantiation of the UART device (Figure 3) in Arcom Viper board.

```
instantiate myuart as UART {
  multi_instance = no;
  serial_reg_map => cpu_virtual {
     base_address = static(0xF8100000);
  };
}
```

Figure 5 – Device instantiation for UART

3.5 Invariant specification

Invariants are constraints that the device must satisfy at runtime. In general, constraints fall into two categories: logical constraints and sequential constraints. We next describe them and the resulted debugging code in detail.

3.5.1 Logical constraint

Logical constraints are boolean expressions on register values. For example, in an UART the DLAB bit must be set to 1 to access the DLL and DLM registers. A violation of this constraint indicates the programmer may have forgotten to set DLAB bit before accessing DLL and DLM registers. Logical constraints also help to catch hardware bugs. For example, in an Ethernet controller the transmit overflow error bit and the transmit underflow error bit should never be set at the same time. A violation of this constraint indicates a hardware error.

In HAIL, we use *(pre_condition) {actions} (post_condition)* to describe a logical constraint, where actions are register read and write and the *pre_condition* and *post_condition* are the pre- and post- assertions for the actions. For the UART example in Figure 3, there are two logical constraints

```
(r(DLAB) == 0) {r(RX), w(TX), rw(IER)}
(r(DLAB) == 1) {rw(DLL), rw(DLM)}
```

indicating that the DLAB bit needs to be 0 to access register RX, TX, and IER and be 1 to access DLL and DLM (see the description of register LCR in Figure 2).

3.5.2 Sequential constraint

If each read/write access to a register is viewed as an event, sequential constraints specify the acceptable order of the events. For example, when a NEC Ethernet controller needs to send a command to the transceiver, it first puts the write command code to its own command register and then puts the value into its data register. This event sequence -- writing a write command into command register immediately followed by writing a data into the data register -- is the only allowed sequence to send a command to transceiver.

In HAIL, we describe sequential constraints with a syntax inspired by research on temporal logic [16]. Specifically, each sequential constraint is a sequence of events connected by sequence connectors,

```
event =x> event =x> … =x>event
```

where event is a logical constraint (i.e., access of a register with optional pre- and post-condition) and the connector =x> can be one of the following three:

1. A =i> B : Event A is immediately followed by event B; no other event can occur in between.
2. A =e> B : Event A is eventually followed by event B; no other event in the sequence can occur in between.
3. A =o>B : Event A is optionally followed by event B; no other event in the sequence can occur in between, except for the first event whose occurrence terminates the current sequence and restarts the same sequence.

For the above NEC Ethernet controller example, we can express the sequence constraint as follows;

```
{w(MADR)} =i> {w(MWTD)}
```

where w(MADR) represents the write to the command register and w(MWTD) represents the write to the data register. The following is a more complex example for reading codec register in an AC97 controller.

```
{w(CODEC_WR)} =o>
  (r(CODEC_WR_WRC)==1 && r(CODEC_RDRRDYA)==1)
  { r(CODEC_RD_RRDYD) }
  (m(CODEC_WR_WADDR)== m(CODEC_RD_RADDR))
```

There are two events in the example: the first one is simply setting the WR register. The second event is reading the data back from RD register. However, two pre-conditions must satisfy: the preceding write indicates data read command (`WR_WRC==1`) and the data is ready for fetching (`RDRRDYA==1`). Furthermore it has a post-condition that register offset passed in when setting WR register (`CODEC_WR_WADDR`) must be the same as the register offset read back (`CODEC_RD_RADDR`). The =o> connector means that writing WR is a necessary but not sufficient predecessor to reading RD.

3.5.3 Run-time debugging code

The HAIL compiler generates optional debugging code for both logical and sequential constraints. In particular, for each logical constraint the compiler inserts the pre- and post-assertion code before and after, respectively, the corresponding register access. For each sequential constraint the compiler generates a state machine for the whole sequence.

At run-time, when the driver accesses a register, the debugging code first checks the pre-conditions. It then checks all state machines. If any machine reaches an error state, it indicates a violation for the corresponding sequential constraint. Finally, after accessing the register, the debugging code checks the post-condition. The following pseudo-code illustrates the structure of run-time debugging code.

```
type get_reg(){
   HAIL_ASSERT(pre);
   HAIL_CHECK_STATES();
   x = *(unsigned int*)(base + offset);
   HAIL_ASSERT(post);
   return x;
}
```

4. A Case Study

In this section we look at another UART example which illustrates all four elements in HAIL specification together. We also examine the generated code and its usage by the driver.

This UART has similar register layout as the one discussed earlier [15] but lives in PCI IO address space. The system is a MIPS board running in big endian mode. The OS is Linux, where PCI IO space is mapped into the CPU virtual space at the base address denoted by a global variable `mips_io_port_base`.

```
/* part 1: register map */
device UART {
  register_map map {
    .=0x0 {
      RX;
      size=4; access=RO; read_value=volatile;
      [31:8]: reserved;
      [7:0] : RX;
    }
          }
    .=0xc {
      LCR;
      size=4; access=RW;
      read_value=static;
      default_write=dont_change;
      [31:8]: reserved;
      [7]   : DLAB;
      [6]   : SBC;
      ...
    }
  }
}

/* part 2: address space */
address_space PCIIO {
  data_width = {1,2,4};
  endian = little;
  ...
}

address_mapping PCIIO => CPU_VIRTUAL {
  base = literal(mips_io_port_base);
  window = {0x0, 0x200000};
  endian_swapping = no;
  ...
}

/* part 3: instantiation */
instantiate myuart as UART {
  multi_instance=NO;
  reg_map => PCIIO {base=0x1000};
}

/* part 4: invariant specification */
invariant {
  (r(LCR_DLAB) == 0) {r(RX)}
}
```

Figure 6 - Segment of HAIL specification for UART

Figure 6 is an excerpt from the complete HAIL specification for the UART controller. Part 1, the register map description, defines register names, bit field names, and their accessibility (read-only, read-write, etc). In this example we only show the definition of RX register and the partial definition of LCR register. The RX register is defined as being at offset 0x0 in the register map. It is 4 byte in size, it is read-only, and successive reads can return different values (volatile). The upper 24 bits are reserved while the lower 8 bits contains the received byte during receiving. In a similar fashion, the LCR register is defined as being at offset 0xc, has read/write access, static read value and a "dont_change" default write value.

In essence, except for the "size" attribute, the attribute values specified at register level are the default values for the bit fields within the register (with the exception of "reserved" bit field, obviously). For example in the case of LCR register, default_write is an attribute for bit fields only, indicating the

value of a bit field should be preserved during the modification of its neighboring bit fields. Specifying this attribute at the register level gives the default values for all bit fields in LCR. If a bit field has a different value for an attribute, it can simply re-set the attribute following its definition (Not shown in Figure 6). In practice, for example, it is quite common for one bit to be volatile while the rest of the register is static.

Part 2 of the HAIL specification is the address space description. In this example the UART controller lives in the PCI IO space, a window of which, ranging from 0x0 to 0x20000, is mapped to CPU virtual address space at a base address denoted by *mips_io_port_base*. The total mapped region is 2MB in size. The first part of the specification specifies the characteristics of the PCIIO space such as supported data widths and endian-ness. The address_mapping description captures the mapping of the PCIIO space to the CPU virtual space. Specifically we notice that the "endian_swapping" attribute is "no", which indicates no endian swapping in hardware (the host-PCI controller). Since the host is running in big endian mode and PCI IO space is little endian, we will need to swap the data in software when accessing PCI IO space.

Part 3, the device instantiation, is a simple matter of associating register maps with address spaces. In this case, a device "myuart" is instantiated as type UART. Its attributes specify that there is to be only one instance of the device, and that its register map is at the base address 0x1000 in the PCI IO space.

Part 4, the invariant specification, lists a simple invariant clause: DLAB bit must be 0 when the RX register is read. In specifying the complete UART, we have 10 such invariant clauses. In general simple invariants are typically derived from limitations on register manipulations. Complex invariants are usually embedded in device behavior models described in the device documentation.

```
...
#define LCR_DLAB_MASK 0x80
#define LCR_DLAB_SHIFT 0x7
#define RX_RX_MASK 0xFF
#define RX_RX_SHIFT 0x0
...
inline unsigned int get_RX(void) {
    HAIL_ASSET (get_LCR_DLAB() == 0);
    return HAIL_SWAP(* (unsigned int*)
            (mips_io_port_base+0x1000-0+0)));
}
inline unsigned char get_RX_RX(void) {
    unsigned int reg_val;
    reg_val = get_RX();
    return (reg_val & RX_RX_MASK) >>
            RX_RX_SHIFT;
}
...
inline unsigned int get_LCR(void) {
    return HAIL_SWAP(*(unsigned int*)
          (mips_io_port_base+0x1000 -0+0x0c));
}
inline unsigned char get_LCR_DLAB(void) {
    unsigned int reg_val;
    reg_val = get_LCR();
    return (reg_val & LCR_DLAB_MASK) >>
            LCR_DLAB_SHIFT;
}
inline void set_LCR_DLAB(unsigned char val) {
    unsigned int reg_val;
    reg_val = get_LCR();
    reg_val = (reg_val & ~LCR_DLAB_MASK) |
((val << LCR_DLAB_SHIFT) & LCR_DLAB_MASK);
    set_LCR(reg_val);
}
...
```

Figure 7 - Segment of generated code for UART

Figure 7 shows a segment of the generated code for the UART. We can see that a set of get_/set_ functions are generated for each register and bit field. In case of get_RX(), a piece of debugging code is generated to check the invariant condition. Such debugging code can be turned on and off through a HAIL compiler option. Since the host endian format is different from the PCI IO endian format, the generated code correctly performs swapping at appropriate places.

```
(before)
    /* disable break condition */
    serial_out(info, UART_LCR,
serial_inp(info, UART_LCR) & ~UART_LCR_SBC);

(after)
    /* disable break condition */
    set_LCR_SBC(0);
```

Figure 8 - Serial driver using HAIL generated code

Figure 8 shows the change to one line of code in the Linux serial driver before and after using HAIL generated code. We would like to point out the improved clarity in the driver code.[1]

5. Implementation and Discussions

The HAIL compiler is written in C. The front end uses Lex and Yacc to generate an abstract syntax tree from the HAIL specification. The back end checks the specification (e.g., missing bit fields) and generates C code for accessing devices, which is used by the device driver. We next discuss some additional features of HAIL and various error checking capabilities in HAIL.

5.1 Simplifying specification

As we discussed in Section 3, a HAIL specification is often a straightforward and simple translation from the device and bus documentation. To further ease the task of writing the specification, we added the following convenience features:

- **Default attributes:** In the register map, users can supply default attributes for all registers that otherwise must be specified for individual registers.
- **Shorthand**: HAIL defines shorthand macros for many commonly used attributes. For example, ro means access=read_only.
- **Macros:** HAIL allows users to define macros for valid bit field values. Using these macros improves the readability of the driver code. In addition, HAIL automatically generates run-time code to check for invalid write arguments and returned read data.

5.2 Chip revision support

In embedded systems devices often evolve rapidly. For parameter-type changes such as register stride (the default offset between two consecutive registers), register size, bus width,

[1] Note in the transformation we also lost the instance parameter, info, because in this particular case we instantiated myuart as a single-instance device. HAIL supports multi-instance instantiation which can correct this problem.

HAIL already has adequate support. For register or bit field removal, modification or addition, HAIL provides a simple supplementary revision specification.

Figure 9 illustrates revision specification with a hypothetical example. A new revision of the chip can inherit the register map description from the existing revision through "derived from" keyword. One can delete an existing register through "remove" keyword (such as reg_X). One can add a new register through a regular register definition (such as reg_Z). To modify an existing register one can simply remove it and then add it back with the same name at the same offset (such as reg_Y).

```
device A derived from B {
    remove reg_X;
    remove reg_Y;
    .=off_Y {reg_Y; …}
    .=off_Z {reg_Z; …}
};
```

Figure 9 – Illustration of revision specification in HAIL

5.3 Supporting multi-bit-field manipulation

HAIL tries to promote the separation between logical bit manipulations from actual register accesses. Sometimes, however, it is desirable or even mandatory to manipulate multiple bits together. For example, we may need to disable or enable certain interrupt bits of an interrupt mask register at the same time. To support such multi-bit-field manipulation, the HAIL compiler generates memory-based bit manipulation functions in addition to the regular register access functions. The driver code then looks like the following.

```
x = get_reg();
mem_set_reg_bitfield_A(&x, A_val);
mem_set_reg_bitfield_B(&x, B_val);
…
set_reg (x);
```

5.4 Error checking with HAIL

HAIL helps programmers catch various errors that are common in conventional driver development. Many of these have been discussed in the previous sections; we list them here for a complete overview.

5.4.1 Inconsistent specification

In addition to basic syntax errors, HAIL compiler can catch various semantic errors during HAIL specification compilation time. For example, when we worked on a SMC91C111 Ethernet controller, we found a potential bug in the specification: The TXENA bit has a "volatile" read attribute and a "dont_change" default write attribute. "dont_change" means that writing its neighboring bits requires reading TXENA to preserve its old value, which is impossible due to the "volatile" read attribute. In this case, the HAIL compiler generated a warning.

In general, several kinds of inconsistencies can be present in the HAIL specification. For example, one must not specify the default_write attribute for a read-only register. For another example, when an address space is mapped to CPU virtual space through a cascading list of address spaces, all the spaces in the list must support at least one common data width. Otherwise there is no appropriate CPU instruction to access the device in the remote address space.

5.4.2 Correct by void construction

Based on certain attribute values HAIL compiler may intentionally generate void functions that simply have a "#error" statement. If such functions are used in a driver, a C-compile time error will happen with a detailed explanation for the error.

For example, if a register is read-only, the "set" function for this register will be such a void function. More subtly, if a bit field has the default_write attribute being "no_default", all bit field setting functions for its neighboring bit fields will be void functions since one cannot simply set them individually.

5.4.3 Hardware state checking

For register or bit fields with fixed read values, HAIL generates code to verify their return for each read. If any mismatch happens, a run-time error is reported.

5.4.4 Driver parameter checking

The HAIL specification may put limitations on the values of a register or a bit field, set either explicitly through "fixed" read attribute or implicitly through "enum" macro definition. In either cases the HAIL compiler generates checking code to catch any violations during run-time.

5.4.5 Invariant checking

As discussed before, HAIL generates run-time code to check for logical invariants and sequential invariants. Any violation will generate a run-time error.

To implement sequential invariant checking, we recognize that each sequential invariant in essence specifies a state transition diagram. Each state transition diagram has an error state, which indicates a violation of the invariant. In this sense, each invariant corresponds to a finite state machine. At run-time we feed each register access as an event to all the finite state machines corresponding to all the invariant clauses. If any of them enters the error state a run-time error is thrown; it indicates a violation of the corresponding invariant clause.

5.4.6 Miscellaneous

HAIL can perform several other checks that does not fall into the above categories. For example, HAIL can generate run-time code to check whether a register map in a remote address space is mapped within the boundaries of the mapped window.

In addition, the arguments and return values of HAIL generated inline functions are strongly typed. Any violation of data typing can be caught by the C compiler.

6. Experiments and Results

HAIL promises to lower development effort, to help debug code, and to improve portability. To validate these claims, we did various experiments on an Xscale PXA255 based development board called Viper from Arcom. We did the experiments on both Linux and NetBSD. Two devices were chosen as the experiment subjects, the on-board UART serial controller [15] and the SMC91C111 Ethernet controller [2].

We wrote the HAIL specifications for both devices from the corresponding device documentation. The HAIL compiler generated device access functions. We then modified the existing drivers in Linux and NetBSD to use the HAIL-generated access

functions. In this section, we share the results and provide some insight into the strength and possible weaknesses of HAIL.

In all test cases we observe very similar data between NetBSD and Linux. For clarity we only show the data for Linux here. Complete experiment data are available in [17].

6.1 Impact to development effort

Impact on the development effort is a subjective measure. We attempt to shed some light on this topic by comparing the code sizes and the nature of code changes in our experiments.

The following table, Table 1, shows the various source code sizes for the Linux smc91x.c driver before and after we modified it (based on Linux 2.4.26). All numbers are numbers of lines, excluding comments and blank lines.

Table 1 - Size comparison

	Original driver	HAIL driver	size reduction
Macro definitions	386	68	82.4%
Bit manipulations	215	108	49.8%
Driver code	1984	1474	25.7%

We were able to remove 82.4% of hand-written macro lines and 49.8% of bit manipulations. In exchange, we added HAIL specification which has roughly the same number of lines as the reduced macro lines. This may suggest a saving of pure programming effort. But we believe that the HAIL specification is a simple translation from the device documents, which should incur less effort for the same number of lines. When chip vendor supplies the register map description and OS vendor provides the address space description, the saving is even more substantial.

The smc91x.c driver is used by at least ten different boards. Some have 8-bit buses, some have 16-bit buses and others have 32-bit buses with 4-byte register stride. We notice quite a bit of code reduction happens in the multiple boards support area. With HAIL multiple boards can be accommodated by modifying only the address space description or the device instantiation.

In addition to the size reduction in macro definitions and bit manipulations, there is also a small amount of reduction (85 lines) in other areas. This reduction comes from more compact and logical programming when using HAIL. For example, a typical read-modify-set three line sequence can often be replaced with one-line set_xxx() function in HAIL.

In our experiment we modified existing drivers. It would be a more accurate experiment if we could develop a new driver from scratch where we can experience the full cycle of a driver development with HAIL. We believe in that case the reduction in development effort will be more obvious due to the worry-free approach of register access and bit manipulations in HAIL.

6.2 Impact on driver performance

To evaluate the performance impact of HAIL, we compare the HAIL-based Ethernet driver with the original Linux driver. In particular, we connect a Viper machine to a PC through a dedicated 100base-T connection. Both machines run Linux. The Viper machine has two versions of the Ethernet driver, the original Linux driver and the HAIL-based driver.

We then run netperf to test TCP streaming bandwidth from Viper to the PC. For each of the two drivers in Viper, we measure the throughput and CPU usage of netperf in the Viper machine. Each experiment is repeated 20 times, and we report the results with a 99% confidence interval being less than 1% of the means.

Table 2 - Performance comparison

	Original driver	HAIL driver
Throughput	64.20 Mbps	64.15 Mbps
CPU usage	99.89%	99.69%

The results (Table 2) show that HAIL has minimal impact on the Ethernet performance. The main reason is that HAIL generates access functions as inline functions. Most modern C compilers, such as recent versions of gcc, can generate very efficient object code for inline functions and yield almost identical performance as embedded code.

Nonetheless we still see a small and yet consistent degradation in performance (about 0.1% in the example). There are two possible reasons. First one is inherent in HAIL. When retrieving a bit field the HAIL generated code will perform a bit shifting in addition to bit masking to normalize the return value. Depending on the usage of the value, the bit shifting operation may be unnecessary in hand-crafted driver code. Second, in spite of the improvements, compilers still lose a little performance when compiling inline function code. That loss could well translate into the tiny performance degradation here.

We have thought about increasing performance by aggressively caching static register values in memory. That idea turns out to be disappointing. While a memory read is certainly faster than a device register read, a conditional branch checking for the validity of caching data adds overhead. Depending on the speed difference between reading memory and reading device registers, there may not be any performance gain at all. In addition, caching data could cause race conditions for registers that are accessed in both process context and interrupt context.

6.3 Bug findings

Both UART and SMC91C111 drivers in Linux and NetBSD are mature drivers in existence for many years. They are used by multiple boards and have been well tested. Nevertheless, with HAIL we still found one potential bug in both Linux and NetBSD. Driver normally enables packet transmission by setting TXENA bit to 1. However the device can change the bit from 1 to 0 on error conditions. The Linux driver always set this to 1, which could potentially discard transmission errors. The NetBSD driver always writes back the value read from the bit, which could accidentally disable transmission under a race condition when an error happens.

When one develops a new device driver, we expect the bug finding capability in HAIL to be more valuable. In that case, HAIL can help identify misunderstanding of documents, inconsistent or wrong documentation, hardware bugs or software bugs in the early stage of the driver development.

6.4 Improved portability

There are two aspects of the portability of a HAIL driver: the portability of the HAIL specification and the portability of the driver source code. For both UART and SMC91C111 cases we were able to use the same HAIL specification with little modification for both NetBSD and Linux. In general we expect only changes in address space description when the same device is used in different OSes and systems.

We also notice that the SMC91C111 is used in many other boards with different buses. For example, the Viper board uses 16-bit only bus, while the Mainstone board supports 8-, 16- and 32-bit data access. A single line change in the specification, about bus_width attribution, would enable the same driver to be reused for the Mainstone board.

7. Summary and Future Work

HAIL is a domain-specific language that specifies attributes related to device access and generates device access functions for driver to use. It promises to expedite driver development, help identity bugs in software and hardware, and increase code portability. HAIL attempts to do so with none or little run-time overhead.

We like to promote its usage among driver programmers and let it mature for practical applications. We expect the syntax and semantics of HAIL to be expanded during this process. For current implementation of HAIL compiler, language specification and user guide, please visit the web site [17].

One related area which HAIL can be extended is the manipulation of DMA descriptors. A DMA descriptor can be viewed as a device with one register map whose instances are created dynamically at run-time. To support this, HAIL needs to support dynamic instances. In addition, we need to take care of some DMA specific issues, such as cache flashing.

We are currently addressing other aspects of driver programming, such as synchronization, buffer management, and device modeling. Our goal is to come up with a complete set of practical solutions for easy, correct and portable driver programming, of which HAIL is an important part.

8. Acknowledgements

Our thanks go to Cliff Neighbours for implementing the HAIL based drivers on NetBSD and performing various evaluation tests.

9. References

[1] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, "An empirical study of operating systems errors," in *Proceedings of the eighteenth ACM symposium on Operating systems principles (SOSP '01)*, pp. 73--88, Banff, Alberta, Canada, 2001, ACM Press.

[2] SMSC. *LAN91C111 - 10/100 Non-PCI Ethernet Single Chip MAC + PHY*. Web site: http://www.smsc.com/main/datasheets/91c111.pdf.

[3] S. Wang and S. Malik, "Synthesizing operating system based device drivers in embedded systems," in *Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pp. 37-44, Newport Beach, CA, USA, 2003, ACM Press.

[4] C. L. Conway and S. A. Edwards, "NDL: a domain-specific language for device drivers," in *Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools*, Washington, DC, USA, 2004, ACM Press.

[5] S. A. Edwards, "SHIM: A Language for Hardware/Software Integration," in *Proceedings of the Synchronous Languages, Applications, and Programming (SLAP)*, Edinburgh, Scotland, April 3, 2005.

[6] F. Merillon, L. Reveillere, C. Consel, R. Marlet, and G. Muller, "Devil: An IDL for Hardware Programming," in *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI 2000)*, pp. 17-30, San Diego, CA, 2000.

[7] L. Reveillere, F. Merillon, C. Consel, R. Marlet, and G. Muller, "The Devil language," IRISA, Rennes, France Reserach Report 1319, 2000.

[8] Jungo Software Technologies. *WinDriver and KernelDriver*. Web site: http://www.jungo.com.

[9] Compuware Corporation. *DriverStudio*. Web site: http://www.compuware.com/products/driverstudio/.

[10] D. S. Rosenblum, "Towards a method of programming with assertions," in *Proceedings of the 14th international conference on Software engineering*, pp. 92--104, Melbourne, Australia, 1992.

[11] S. H. Edwards, M. Sitaraman, B. W. Weide, and J. Hollingsworth, "Contract-Checking Wrappers for C++ Classes," *IEEE Transactions on Software Engineering*, vol. 30, pp. 794--810, 2004.

[12] Coverity Inc. *Coverity Prevent for C and C++*. Web site: http://www.coverity.com.

[13] Microsoft. *Static Driver Verifier: Finding Bugs in Device Drivers at Compile-Time*. Web site: http://download.microsoft.com/download/5/b/5/5b5bec17-ea71-4653-9539-204a672f11cf/SDV-intro.doc.

[14] L. d. Alfaro, M. Faella, T. A. Henzinger, R. Majumdar, and M. Stoelinga, "Model Checking Discounted Temporal Properties," in *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*.

[15] Intel. *Intel® PXA27x Processor Family Developer's Manual*. Web site: http://www.intel.com/design/pca/applicationsprocessors/manuals/280000.htm.

[16] A. Galton. *Temporal Logic*. Web site: http://plato.stanford.edu/archives/win2003/entries/logic-temporal/.

[17] W. Yuan and J. Sun. *HAIL - Hardware access interface language (compiler, specification, user guide and other related information)*. Web site: http://www.docomolabsresearchers-usa.com/~jsun/hail.