A Formal Model of Memory Peculiarities for the Verification of Low-Level Operating-System Code

Hendrik Tews [1, 2] Tjark Weber [1, 3]

Institute for Computing and Information Sciences
Radboud Universiteit Nijmegen, The Netherlands

Marcus Völp [1, 4]

Institute for System Architecture
Technische Universität Dresden, Germany

Abstract

This paper presents our solutions to some problems we encountered in an ongoing attempt to verify the micro-hypervisor currently developed within the Robin project. The problems that we discuss are (1) efficient automatic reasoning for type-correct programs in virtual memory, and (2) modeling memory-mapped devices with alignment requirements. The discussed solutions are integrated in our verification environment for operating-system kernels in the interactive theorem prover PVS. This verification environment will ultimately be used for the verification of the Robin micro-hypervisor. As a proof of concept we include an example verification of a very simple piece of code in our environment.

Keywords: operating-system kernel, micro-hypervisor, virtual memory, memory-mapped devices, formal verification

1 Introduction

The programming environment of operating-system kernels differs in essential ways from that of application programs. The most prominent differences are direct hardware access and privileged processor instructions. In addition, certain situations that are absurd from an application-programming point of view are possible (and sometimes even very common) in kernel programming. For instance, many kernels see some piece of main memory at different virtual addresses. (Other important

[1] The authors have been supported by the European Union through PASR grant 104600.

[2] Homepage: `http://www.cs.ru.nl/~tews`

[3] Homepage: `http://www.cs.ru.nl/~weber`

[4] Homepage: `http://os.inf.tu-dresden.de/~voelp`

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

ways in which kernel programming differs are the use of casts and pointer arithmetic, however, those are outside the main scope of this paper.)

The additional hardware features that are exploited in a kernel programming environment are usually subject to very specific programming rules, which are described in the hardware architecture's technical documentation. Typically the rules are not enforced and, when not obeyed, one might get bugs that are hard to reproduce. As a consequence certain kinds of bugs can only occur in kernels (or similar kinds of low-level systems).

In the following we use the term *kernel-programming features* to refer to the additional phenomena of kernel programming just described. A verification environment for kernel code must of course model the kernel-programming features in order to give a semantics to the code. Less immediate but equally important is, however, that the verification environment is faithful with respect to the possible bugs associated with kernel-programming features. With faithful we mean here that any such bug must with certainty lead to a verification failure. The subtle differences between valid and erroneous code make the design of the verification environment very challenging.

In this paper we discuss our approach to model certain kernel-programming features that are used in the Robin micro-hypervisor. The solutions presented here are already implemented in our verification environment. The presentation includes a discussion of the kinds of programming errors that our models are able and are not able to catch. We focus on the memory peculiarities in this paper and present the following two points:

- A model of virtual memory that captures virtual-memory aliases, permits the verification of page-table modifications and efficient reasoning about well-behaved code (Section 3).
- A general model for memory-mapped devices and the phenomena of reserved bits in certain hardware registers (Section 4).

In addition, Section 2 gives a general overview of our verification environment, and Section 5 contains a verification example.

2 Overview of the Robin Verification Environment

This section provides some details about our verification environment for the Robin micro-hypervisor, see Figure 1 for illustration. More technical information can be obtained from [8], while the context of the Robin project is described in [6]. Our approach relies on source-code verification in the interactive theorem prover PVS [5]. The input language of PVS is higher-order logic enriched with predicate subtyping and some other forms of dependent types. For the verification we model parts of the IA32 hardware and the semantics of C++ data types inside PVS. These two models provide the basic operations for a model of the micro-hypervisor. Then we use the prover

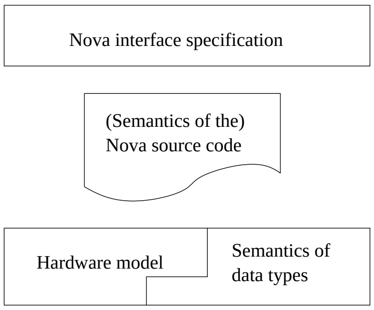

Fig. 1: Verification approach

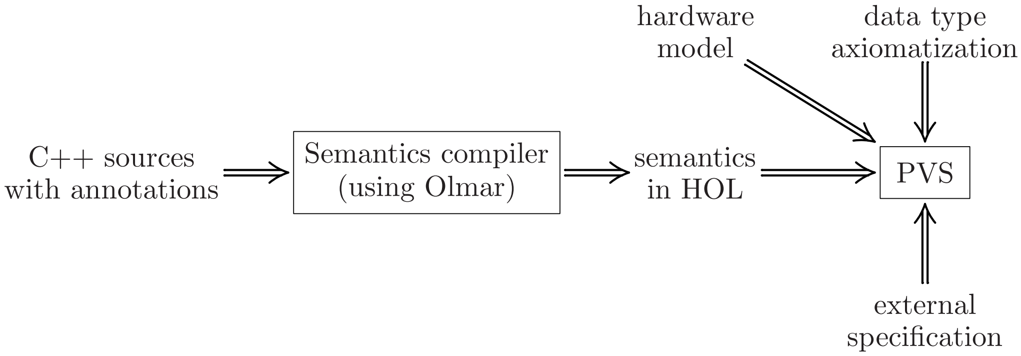

Fig. 2. Approach for source code verification

component of PVS to establish theorems about the model. Technically we show

$$\Phi_{data\_types},\quad \Phi_{hardware}\quad \vdash\quad \varphi(\text{hypervisor}),$$

where φ is one property from the hypervisor specification, such as *termination without runtime type errors*. Our verification results will describe properties of the source code. A formal lifting of the results to object code (which would eliminate the correctness of the compiler from our assumptions) is planned for the future.

Figure 2 depicts the data flow of our verification approach. A semantics compiler translates the C++ source code into its semantics in higher-order logic in PVS. The semantics compiler is currently developed on the basis of the Elsa/Olmar C++ front-end [7] and will be described elsewhere. The basic building blocks of the C++ semantics are provided by the hardware model and the data-type semantics.

The operations in the hardware model as well as the data-type semantics and the semantics of C++ fragments are uniformly modeled as *state transformers*. State transformers come in two flavors: statement state transformers (for C++ statements) and expression state transformers (for C++ expressions and everything else). An expression state transformer is a function of type

$$\mathsf{State} \rightarrow \mathsf{ExprResult[State,\ Data]}.$$

Here State is the type of all possible states of the hardware model, and Data is a type parameter for the result of the state transformer (if it terminates successfully). Both types are theory parameters in our PVS formalization, making them effectively polymorphic. In the verification of concrete C++ programs however, Data is instantiated with a fixed type for each state transformer. The State parameter is either instantiated as well (if we verify against a concrete hardware model), or left polymorphic if we verify against the plain-memory specification (see Section 3) in general. The type ExprResult is defined as follows:

```
ExprResult[State, Data : Type] : Datatype
BEGIN
  OK(state: State, data: Data) : OK?
  Exception(ex_type : Exception_type, state : State) : Exception?
  Fatal : Fatal?
  Hang : Hang?
END ExprResult
```

This piece of PVS code defines ExprResult as a disjoint union with four variants tagged OK, Exception, Fatal and Hang. The identifiers with question marks are recognizer predicates for the corresponding variants (e.g., OK? is true on OK(···) and false on the other three variants). The identifiers state, data, ex_type are (partial) accessor functions (e.g., state(OK(s, −)) = s).

A state-transformer result of the form OK(s, d) models successful termination with successor state s and result d. Hang stands for non-termination, for instance because of a while loop or a page fault that keeps occurring at the same instruction. Fatal is reserved for unrecoverable errors, which we want to rule out by verification. A result of the form Exception models hardware exceptions and interrupts that will be handled by the micro-kernel.[5]

Statement state transformers have the form [State → StmtResult[State, Data]] with a very similar type StmtResult. The main difference between ExprResult and StmtResult is that StmtResult contains abnormalities like Break and Return to model the corresponding C++ control-flow statements, very similar to [3]. Further, OK does not carry a data element. The parameter of type Data is used inside Return to model the return type of C++ functions.

State transformers can be composed in the obvious way. For two state transformers f and g their composition f ## g is a state transformer that performs the effect of g on the successor state of f if f returns OK. Otherwise g is discarded and the result of f is the result of the composition. If f is an expression transformer, the data in any OK result is discarded.

The first base component of our verification environment, the hardware model, formalizes an abstract model of the IA32 hardware in PVS. It provides physical memory, virtual memory with address translation via page tables, and much more. The hardware model does not blindly implement the behavior of the real hardware. Instead certain subtle programming errors that would cause the real CPU to do nonsense yield unprovable proof obligations in the hardware model. For instance, the attempt to interpret a string as a page-table entry yields a proof obligation that remains unprovable unless a suitable data-type conversion is formalized as an axiom. (Of course such an axiom is not justified.) This kind of error checking even works for hardware-initiated page-table traversals during address translation.

The memory formalization in the hardware model is split into different layers of memory models, for instance for physical and virtual memory. These different memory models share a common interface. Every memory model defines a type State of possible states and the following record of operations.

```
Memory_struct : Type = [#
    memory_read : [Address → [State → ExprResult[State, Byte]]],
    memory_write : [Address, Byte → [State → ExprResult[State, Unit]]],
    ... #]
```

The two operations respectively read and write one byte at the given address. Note that reading in memory can change the memory (for instance set the accessed bits in the page table). The memory structure contains two additional operations for

[5] Exception does *not* model C++ exceptions. We consider C++ exceptions too heavyweight to be used in an operating-system kernel. Therefore they are outside the fragment of C++ treated in this paper.

modeling memory-mapped devices and reserved bits, see Section 4.

In order to get a uniform treatment of memory-mapped devices and the special effects of feature flags in CPU control registers, we decided to have a uniform address space for both memory and registers. The type Address is therefore defined as a record consisting of a Register_Id and an offset:

Address : **Type** = [# type_of : Register_Id, offset : nat #]

Real memory appears as a (rather big) special register with Register_Id Mem. For memory the offset is the real address. For hardware registers the offset will most often be 0, however, for processor architectures featuring partial register access, the offset might be positive (e.g., for accessing `AH` one would use offset 1 in `EAX`). The sizes and possible offsets of hardware registers are enforced with suitable side effects, see Section 4.

On top of the memory structure we define the following two functions for accessing contiguous blocks of memory for every memory model in the obvious way.

memory_write_list :
  [Memory_struct → [Address, list[Byte] → [State → ExprResult[State, Unit]]]]

memory_read_list :
  [Memory_struct → [Address, nat → [State → ExprResult[State, list[Byte]]]]]

Our second base component, the semantics of data types, provides a suitable semantics for all C++ types and all necessary hardware data types (such as page-table entries). It exploits under-specification to make the detection of erroneous type casts and wrong implicit type conversions possible (like, for instance, reading data from a union with the wrong type) [2]. The data-type semantics is independent of any memory model. It provides operations to convert data from and to their object representation, which is of type list[Byte]. Writing and reading the object representation into and out of memory is done with the above functions memory_write_list and memory_read_list.

In our design, the three base components—hardware model, data types, and C++ semantics—are relatively independent of each other. It is therefore possible

- to add new operations to the hardware model,
- to use different versions of the hardware model for different parts of the hypervisor; the boot code of the hypervisor can, for instance, be verified against physical memory,
- to add additional axioms to the data types, e.g. to model compiler-specific assumptions about the size of some data types or the precise behavior of some type casts, and
- to adopt the semantics of new C++ features or compiler-specific C++ constructs.

Our hardware model and the C++ semantics are necessarily incomplete. Many of the omissions however do not lead to global assumptions on the validity of our verification. The hardware model, for instance, does not contain virtual 8086 mode, but the validity of our verification does not hinge on the absence of instructions that enable virtual 8086 mode. Instead the `VM` flag, which controls this mode, is protected with a suitable side effect (see Section 4). Any attempt to enable virtual 8086 mode will yield a Fatal result. Hence a proof of normal termination suffices to show that virtual 8086 mode is never enabled. Similarly, the use of missing features in the C++ semantics will trigger an assertion in the semantics compiler.

For a number of features currently not present in our verification environment, we plan inclusion in the future. These features are (1) the Translation Lookaside Buffer (TLB)[6], (2) cache policy checking for devices, (3) segment offsets and segment size checking, (4) linking object code and instruction fetch to the abstract C++ semantics. Because of their absence we are currently unable to detect certain kinds of errors, namely

- TLB errors, e.g. inconsistencies between the TLB and the page tables, or implicit assumptions about the TLB size and structure,
- segment violations (the Robin micro-hypervisor uses a flat memory model where no segment violations can occur, however, currently we do not check that the segment descriptors are filled with the proper values),
- cache policy errors for memory-mapped devices, and delayed side effects for cachable memory-mapped devices,[7]
- discrepancies between our C++ semantics and the compiled object code, which (apart from compiler bugs) could occur for the following reasons: volatile-related errors in the source code[8], certain compiler optimizations (e.g. delayed write-back to memory), or self-modifying code (however, no self-modifying code is contained in our current verification target).

Once these missing features have been added, our verification will build on the following general assumptions:

- The software to be verified will be executed on a single-processor system.
- Caches for real memory are working completely transparent and can be ignored. This should be guaranteed by the hardware on single-processor systems.
- The involved software tools—the C++ compiler used to compile the Robin micro-hypervisor, our semantics compiler (including the Elsa C++ parser and type checker), and PVS—produce correct results.

3 The Plain-Memory Abstraction

The memory in an IA32 system is a sophisticated device: segments and page tables specify access rights, a given piece of memory might be visible in different virtual-address ranges, the address translation in the CPU from virtual to physical addresses might differ from what is specified in the page table because of bogus TLB entries,

[6] The TLB is a special CPU-internal cache for virtual-to-physical address translations.

[7] The source code that we currently target does not involve any devices. In general, cache policy checking for memory-mapped devices is trivial to add with our mechanism for side effects (see Section 4). To model cache effects on cachable devices, the model of the device should include the relevant cache effects.

[8] A C++ compiler is permitted to perform arbitrary optimizations with respect to non-volatile data. Memory accesses to such data are not part of the *observable behavior* of a C++ program, which makes a correct semantics difficult. At the moment our C++ semantics treats all data as volatile. A verification based on the current semantics will therefore not catch missing volatile annotations or missing memory fences.

and much more. We cannot ignore all these effects, not even for most innocent kernel code, because of the errors that they might cause.

As a consequence we designed the *plain-memory* abstraction for the verification of those parts of the kernel that require only the standard C++ memory model. It deals with the following issues.

- Writing or reading a single byte in memory can have devastating effects if one hits a memory-mapped device, a page table or simply an unmapped address. For correctness, the verification must therefore be carried out against a faithful model of IA32 memory. Plain memory provides a (comparatively) simple abstraction that can be used for those parts of the sources that only need well-behaved memory without special effects.
- The IA32 hardware provides several memory configurations: real-address mode, protected mode with and without paging. Our hardware model multiplies the number of different memories because we prefer to model different memory features (such as the TLB or execution of the page-fault handler) in different memory models. Most of the code does not depend on a concrete memory model and should consequently be verified against a suitable set of memory models. Plain memory permits precisely this because every memory model of interest will give rise to a model of plain memory.

Technically plain memory is a specification that provides byte-wise read and write access to memory, where special properties are guaranteed for *read-blessed* and *read/write-blessed* address regions. (There are additional operations to access side effects of memory-mapped devices and to enforce reserved bits. We ignore these things here, see Section 4 for details.) The general idea is simple. Memory at blessed addresses is sane: read access does not change anything in the blessed address range, and write access only changes the bytes written (in the expected way). Moreover, these special properties are maintained as long as only blessed addresses are accessed. No guarantees are made however for a memory access outside the blessed address regions. We have shown in PVS that these properties are satisfied by normal virtual memory (that is outside memory-mapped devices) under the following preconditions:

- All blessed addresses are mapped in the page table, and no two different read/write-blessed addresses are mapped to the same physical address. (Different read-blessed addresses might refer to the same physical address, and there might also exist other mappings for addresses outside the blessed address regions.)
- The page tables can only be read-blessed if the accessed and dirty bits are set. (This condition can be relaxed for page-table entries of read-blessed or unblessed memory.)

We want the plain-memory specification to be usable with all concrete memory models (including physical real-address memory). Therefore the specification must describe all properties only with the observations that can be made by reading and writing single bytes. In PVS the specification is split into a record of functions (capturing the plain-memory signature) and a predicate for the required properties. With this technique the axioms of the plain-memory specification do not show up as axioms in the PVS formalization, hence they do not affect consistency. Instead, any use of a plain-memory property in a proof will spawn a subgoal requiring the proof of the plain-memory axioms. The plain-memory signature is defined as follows:

```
Plain_Memory : Type = [#
  mem : Memory_struct[State],          % see page 4
  states  :  PRED[State],              % states fulfilling the plain memory properties
  ro_addr  :  PRED[Address],           % read−blessed addresses
  rw_addr :  PRED[Address]             % write−blessed addresses
#]
```

The properties of plain memory are specified as follows.

```
plain_memory?(pm) : bool =
   unchanged_memory_invariant?(pm‘mem, pm‘states,
      union( all permitted state transformers except write access to pm‘rw_addr ),
      union(pm‘ro_addr, pm‘rw_addr)) ∧
   unchanged_memory_invariant?(pm‘mem, pm‘states,
      memory_write_transformers(pm‘mem, pm‘rw_addr),
      pm‘ro_addr) ∧
   unchanged_memory_write_invariant?(pm‘mem, pm‘states, pm‘rw_addr) ∧
   changed_memory_invariant?(pm‘mem, pm‘states, pm‘rw_addr) ∧
   transformers_ok?(pm‘states,
      union( all permitted state transformers )) ∧
   side_effect_content_unchanged(union(pm‘ro_addr, pm‘rw_addr), pm‘states,
      memory_read_side_effect(pm‘mem)) ∧
   side_effect_content_unchanged(pm‘rw_addr, pm‘states,
      memory_write_side_effect(pm‘mem))
```

We have omitted the involved expression for the union of all permitted state transformers. This set contains all read accesses to read- and read/write-blessed addresses, all write accesses to read/write-blessed addresses, and all corresponding side effects.

The first clause states that read accesses to blessed addresses and all possible side effects do not change the contents of any of the blessed addresses. The second clause expresses the same for write accesses and the read-blessed addresses. The third clause requires that a write access to one address leaves all other read/write-blessed addresses intact. The fourth clause states that write accesses actually change the written address in the right way. The utility predicates used in the first four clauses additionally require that the set of states forms an invariant with respect to the respective set of state transformers. This makes the plain-memory property an invariant: permitted state transformers must stay in the set of plain-memory states, in which all the nice properties hold. The fifth clause makes all memory accesses terminate with OK (which prohibits e.g. unhandled page-faults). The last two clauses require that side effects do not change the data read or written.

The plain-memory specification entails that only explicit writes change a memory cell. This property enables us to prove the following lemma.

```
plain_memory_read_write_other_res : Lemma
  plain_memory?(pm) ∧
  pm‘states(s) ∧
  in_blessed_memory?(dt1, addr1, pm‘rw_addr) ∧
  in_blessed_memory?(dt2, addr2, union(pm‘ro_addr, pm‘rw_addr)) ∧
```

```
blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) ∧
valid_in_mem(pm,dt2)(addr2)(s)
  ⟹
    data( (write_data(pm,dt1)(addr1, data1) ##
           read_data(pm,dt2)(addr2))(s) )
      =
    data( read_data(pm,dt2)(addr2)(s) )
```

It expresses that for two variables of type dt1 and dt2 that lie disjoint in blessed memory, writing the first one does not change the contents of the second. This lemma is used in a rewrite engine that enables PVS to symbolically compute the value of a variable by going back to the last write access to that variable.

Our hardware model contains physical memory (RAM) as a base of all memory models. Physical memory provides one byte of storage for every address up to a certain maximum. Accesses above the maximum yield Fatal as result. Unsurprisingly we can prove that all states of physical memory form a plain memory, with all addresses below the maximum read/write-blessed.

Our model of linear memory contains page-table based address translation, but no TLB or page-fault handler yet. The linear memory is stacked on top of a plain memory using the general memory-structure interface (see Section 4.2 for more on the stacking of memory models). This plain memory is typically instantiated with physical memory (possibly containing devices). We have shown in PVS that the plain-memory properties are obtained for linear memory under the following preconditions:

- The code segment register (CS), determining the code privilege level, the page-table base register, and the page tables themselves remain unchanged (with the exception of access bits).
- Any translation for read or execute accesses succeeds for the entire blessed range of virtual addresses. Translations for writes succeed for the writable subset.
- Blessed writable virtual addresses map to blessed writable physical addresses, blessed read-only addresses map to blessed readable or writable physical addresses.
- Page tables reside in a memory area that is disjoint from the targets of the above mappings.
- There is no blessed shared-memory alias to a writable virtual address. (Virtual read-only regions may be shared arbitrarily.)

One point to highlight is that we only have to require those page-table entries to remain unchanged that are used in the translation of the virtual blessed address range. This allows us to modify unrelated page-table entries without loosing the blessing properties. We achieve this by requiring

```
disjoint?(virt_to_phys_range(s, union(pm`ro_addr, pm`rw_addr)),
          address_in_pt_range?(s, union(pm`ro_addr, pm`rw_addr))),
```

where virt_to_phys_range translates all addresses in the virtual blessed address range, and address_in_pt_range? returns the corresponding physical addresses containing the page-table entries for this range.

4 Memory-Mapped Devices and Reserved Bits

Although most device drivers reside outside the micro-hypervisor, some devices (e.g. the interrupt controller) must remain under kernel control to prevent malicious code from monopolizing the system. To program these devices, the micro-hypervisor code accesses certain device registers. Unlike normal RAM, these registers show very special behavior when accessed. Special-purpose processor registers (such as the IA32 control registers [IA32-3a-2.5] [9]) are similar to device registers in that read or write accesses to them may cause special effects. For our verification attempt, the following special effects are important.

Reserved bits The value of reserved bits must not be modified, or otherwise the processor behavior is undefined. For example, bits 0–2 and bits 5–11 of the IA32 page-table base register (`CR3`) are reserved [IA32-3a-2.5].

Access type restrictions Some device registers are read-only respectively write-only accessible, or they allow no instructions to be fetched. ROM is a prominent example of a read-only accessible device.

Alignment restrictions Some devices require that registers are accessed only at certain offsets relative to the register base address. Furthermore, each access must read or write a certain amount of data at once. For example, the registers of the IA32 advanced programmable interrupt controller (APIC) are aligned on 16-byte boundaries. They must be accessed with 4-byte wide and 4-byte aligned reads and writes [IA32-3a-8.4.1].

Side effects Reading or writing causes side effects on some devices. For example, writing to the IA32 APIC *end of interrupt* register signals completion of the interrupt-handling procedure [IA32-3a-8.8.5]. This may cause the immediate delivery of the next pending interrupt.

More abstractly, we can summarize these behaviors as follows: reading and writing certain registers and certain locations in memory may result in modifications to the system state, to the memory (or register) contents, and to the value read or written. Furthermore, the behavior of an operation may be undefined; in this case the verification should fail.

4.1 Modeling Devices and Reserved Bits

Instead of modeling memory and additional devices as parallel abstract machines, we prefer the following approach. We extend the Memory_struct of each memory model with two *side-effect transformers* memory_read_side_effect and memory_write_side_effect of the following type:

$$\mathsf{Address},\ \mathsf{list[Byte]},\ \mathsf{bool} \rightarrow [\mathsf{State} \rightarrow \mathsf{ExprResult[State,\ list[Byte]]}].$$

Unlike memory_read and memory_write, these side-effect transformers take the entire list of bytes read from (or written to) the given address. This is necessary to enforce alignment restrictions, which require knowledge about the amount of data that is

[9] The notation [IA32-3a-2.5] refers to Volume 3a, Section 2.5 of the *Intel 64 and IA-32 Architectures Software Developer's Manual* [4].

read (respectively written) at once. The third parameter is an indicator whether the access crossed a page boundary in a higher virtual memory layer. We elaborate on the use of this parameter in Section 4.2.

In the following we use a simple memory-mapped random number generator (RNG) to illustrate how one can model side effects and access restrictions. The RNG device provides one read-only memory-mapped register, rnd_val, that contains an unspecified (supposedly "random") value. The internal state of the device (which is kept in addition to the memory state) contains two natural numbers seed and access_count. The value of access_count is incremented as a side effect with each memory access. The seed is left unspecified.

To obtain the random value we apply a completely unspecified function random to the seed and the current access_count. Because access_count is strictly increasing, we get potentially different values for every access to rnd_val. Under-specified and non-deterministic behavior of more complicated devices can be modeled in a similar fashion.

4.1.1 Access Type and Alignment Restrictions

Modeling devices with access-type and alignment restrictions is straightforward by checking these restrictions for overlapping accesses. As an example, we impose that rnd_val must be accessed with machine-word granularity (as `unsigned int`).

```
unaligned_access(a : Address, bl : list[Byte], cp : bool) :
      [Random_device_memory → ExprResult[Random_device_memory, list[Byte]]] =
  If disjoint?(address_block(a, length(bl)), address_block(rnd_val, size(uidt(dt_uint)))) ∨
      (¬ cp ∧ length(bl) = size(uidt(dt_uint)) ∧ a = rnd_val)
  Then ok_result(bl)
  Else fatal_result Endif
```

Here size(uidt(dt_uint)) comes from the C++ data-type model and gives the number of bytes of an `unsigned int` (which is usually 4 on the IA32 architecture, although this may vary with the C++ implementation). To require the read-only behavior of rnd_val we use the following after the above alignment check has been passed:

```
write_rnd_dev(a : Address, bl : list[Byte], cp : bool) :
      [Random_device_memory → ExprResult[Random_device_memory, list[Byte]]] =
  If a = rnd_val
  Then fatal_result
  Else ok_result(bl) Endif
```

The memory_read_side_effect and memory_write_side_effect transformers of the RNG device are set to compositions of the above two transformers and other checks, which we discuss below.

4.1.2 Reserved Bits

Reserved-bit restrictions come in two flavors, depending on whether the value of reserved bits is specified. For example, the IA32 processor manuals [IA32-3a-2.5] specify bits 11–31 of the CR4 register as reserved. The value of these bits must be 0. As long as the value of reserved bits is specified, we merely have to check that the value in the byte list passed to the memory_write_side_effect transformer is according to the specification. Registers whose reserved bits all have specified values can then be modified simply by writing to these registers.

Other special-purpose processor registers and device registers leave the value of certain reserved bits undefined. In this case we match against an unspecified value in the memory_write_side_effect transformer. Because the initial register contents are not specified, we can establish that reserved bits are unaltered by a write access only when the written data originates from a previous read of this register.

Reserved bits can also be used to restrict the processor modes in which the micro-hypervisor may execute. For example, we fix the mode bits in the IA32 control registers `CR0` and `CR4` to the setting for 32-bit paged, protected mode. This prevents the kernel from switching back to real mode. Consequently it suffices to model only those parts of real mode that are required for the verification of the kernel's boot-strapping code.

4.1.3 Side Effects

Side effects on reads (respectively on writes) cause additional parts of the system state to be updated. Here, one can either update the memory state itself, or add an additional device state. For the random number generator, we decided to use the latter approach. As described earlier, our RNG device implements a side effect to count all memory accesses in its internal state.

```
access_count(a : Address, bl : list[Byte], cp : bool)(s) :
                                    ExprResult[Random_device_memory, list[Byte]] =
  OK(increase_access_count(s)(length(bl)), bl)
```

Another side-effect transformer generates the "random" (i.e. unspecified) value when the rnd_val register is read.

```
random : [nat, nat → { l : list[Byte] | length(l) = size(uidt(dt_uint)) }]

read_rnd_val(a : Address, bl : list[Byte], cp : bool) :
      [Random_device_memory → ExprResult[Random_device_memory, list[Byte]]] =
  If a = rnd_val
  Then λ(s : Random_device_memory) :
                                    ok_result(random(seed(s), access_count(s))(s)
  Else ok_result(bl) Endif
```

4.2 Stacking Memory Layers

For greater modularity we split different hardware features into separate memory models where possible. For instance, segment based and page-table based address translation (which are both part of the virtual memory in the IA32 architecture) can nevertheless be split into two independent memory models. To obtain the overall effect we stack different memory models, exploiting the general memory_struct interface. Every memory layer performs its functionality before invoking the underlying layer via the abstract interface. The bottom layer is a model of physical memory that contains a byte array to store the memory contents.

A good example of the stacking of memory layers is provided by linear memory, which adds page-table based address translation to an underlying physical memory. In order to keep the stacking flexible, the linear memory is not based on our model of physical memory directly, but is instead parameterized with an arbitrary plain-memory model pm. (This model can be instantiated with physical memory, and with

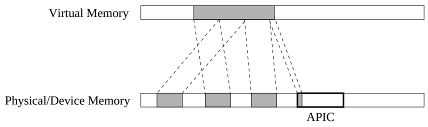

Fig. 3. Splitting of side effects when stacking virtual memory on top of device memory.

any other memory model that satisfies the plain-memory properties.) The linear memory layer defines page-table based lookup and address translation as described for 32-bit page tables in [IA32-3a-3.7]. Finally it defines functions for filling the memory_struct of the linear memory model. The function for memory_read is given by

linear_read(a : Address) : [Linear_memory → ExprResult[Linear_memory, Byte]] =
  If in_memory(max_linear)(a) **Then**
    If Mem?(type_of(a)) **Then**
      linear_resolve(a, Read) ## λ(pa : Address) : memory_read(pm‘mem)(pa)
    Else
      memory_read(pm‘mem)(a)
    Endif
  Else fatal_result **Endif**

The state-space type Linear_memory is equal to the state space of the plain-memory parameter pm. The function linear_read first checks if the access is within the memory or register bounds. For a real memory access, the virtual address a is then translated into a physical address pa, which is used to access the underlying memory of pm. Any abnormal result of the address translation (because of page faults or data-type errors in a page table) is propagated to the outside via the composition of state transformers, ##. Register accesses are passed to pm without address translation, of course.

Our random number generator from the previous section was also implemented as an independent memory layer, which adds the functionality of the RNG device to the underlying memory layer.

However, when combining virtual memory with memory-mapped devices, one must deal with new problems. The virtual memory performs address translation and may thereby split an access to a contiguous block of memory into several accesses to noncontiguous blocks. By coincidence the splitting might result in an access to a memory-mapped device that seemingly satisfies all alignment and granularity requirements of the device. For instance, in Figure 3 the very last piece of the virtual address block is mapped precisely to the first APIC register. We view such an access as an error, because it is only part of a larger memory access. Moreover, the IA32 architecture gives no guarantees that in the case depicted in Figure 3 the access to the APIC is performed with 4-byte granularity [IA32-3a-7.1.1], as required by the APIC.

To detect this kind of error, we introduce a crossed-page indicator cp as argument to our side-effect transformers. Initially being false, this indicator is set to true when the address translation splits a contiguous memory access. The side effect transformer unaligned_access (see page 11) of our RNG device always checks the cp flag and delivers an error if it is true.

5 Example Verification

To illustrate how kernel-code verification works in our environment, we have proved two partial correctness properties of a simple linear search (in an array of N unsigned integers) in PVS. The C++ implementation of the search algorithm uses pointers and pointer arithmetic:

```
unsigned int a[N], value;
unsigned int *first = &a[0];
unsigned int *last = &a[N];
unsigned int *current;

current = first;
while(current < last) {
  if (*current == value) break;
  current++;
}
```

More precisely, we have shown that the current pointer refers to an array element containing the search value if the value is contained in the array, and to the element one beyond the array bounds (last) if no such element is present. The verification of these properties proceeds according to the approach that was outlined in Figure 2 on page 3. First, (1) the C++ sources for the search program are translated into their semantics in PVS. Second, (2) the correctness properties are formulated as pre- and postconditions, and then verified against the plain-memory specification. Our verification thus shows that, under suitable assumptions, the example program runs correctly both in physical and virtual memory. Finally, to avoid vacuous results, (3) the plain-memory preconditions were validated for concrete stacks of memory models (e.g. linear memory on top of the RNG device on top of physical memory).

Because we have established parts (2) and (3) separately, changing the underlying hardware/memory model only requires that one repeats the validation of the plain-memory assumptions (part (3)) for the new memory model.

5.1 C++ to Semantics Translation

The semantics compiler translates the above C++ code into its PVS semantics. Expression-to-statement and lvalue-to-rvalue conversions are made explicit. We only show the translation of the second part (lines 6–10) of the above code snippet.

```
e2s[State, Address, Semantics_void]
   (assign(pm, dt_pointer)(id(current), l2r(pm, dt_pointer)(id(first)))) ##
while(l2r(pm, dt_pointer)(id(current)) < l2r(pm, dt_pointer)(id(last)),
  if_else(
    l2r(pm, dt_uint)(deref(pm)(l2r(pm, dt_pointer)(id(current)))) == literal(value),
    break,
    skip) ##
  e2s[State, Semantics_pointer, Semantics_void](postinc(pm)(id(current)))),
```

Here first, last and current are addresses of disjointly allocated pointer variables. Currently we require disjointness of these variables in a precondition, but once the formalization of memory allocation is complete, disjointness will be derived from the allocator model.

5.2 Verification Against the Plain-Memory Abstraction

Verification is currently done by automatic loop unrolling and simplification according to the plain-memory rewriting rules. For this, the necessary preconditions in_blessed_memory and valid_in_mem for not previously written variables have been added to the precondition. in_blessed_memory states that the variable is correctly allocated in blessed memory, valid_in_mem that the memory contains a valid bit representation for the variable. Typically the latter is established by a previous write to this variable.

In our C++ semantics, all expressions and most statements are expressed using a combination of only four different state transformers: read_data, write_data, ok_result(data) (which returns OK(s, data)), and fatal_result (which produces Fatal). It is therefore possible to simplify expressions by first expanding them to sequences of these transformers, and then simplifying these sequences using the plain-memory rewriting rules (e.g. the rule in Section 3).

Similarly, we evaluate statements up to the point where only expressions remain in the code sequence. For example, under the precondition OK?((expr ## b_ex)(s)) the sequence

(e2s(expr) ## if_else(b_ex, stmt_if, stmt_else))(s)

is rewritten into

(e2s(expr ## b_ex) ## **If** data(expr ## b_ex)(s) **Then** stmt_if **Else** stmt_else **Endif**)(s)

This simplifies with an appropriate rewriting rule for e2s to either expr ## b_ex ## stmt_if or to expr ## b_ex ## stmt_else, depending on the value of data((expr ## b_ex)(s)). Likewise, we rewrite the statements stmt_if and stmt_else to expression sequences containing only read_data, write_data, and the above result transformers. Because of this transformation it suffices to define and prove the plain-memory rewriting rules only for data reads and writes. All other rules of the rewriting system operate independently from the data-type or memory model.

5.3 Establishing Plain Memory

For each memory model we established the plain-memory property for a certain range of addresses. As stated above, for physical memory this is the entire address range. Stacked models contain preconditions which require the blessed address range to be contained in the blessed range of the underlying memory model. It is therefore sufficient to show that the addresses used in the code to be verified all reside in blessed memory. Accesses outside the blessed-memory address range automatically violate the plain-memory assumption and cannot be simplified with the plain-memory rewriting rules. In such a case the plain-memory property must be reestablished before one can proceed with the automatic simplification.

For our verification example, one needs the following preconditions:

- The variables are allocated so that they do not overlap with the registers of the random device.
- The variables are allocated so that they do not overlap with a page-table entry

used to access some of the variables.

- All page-table entries of these variables are writable (because reference bits may be written back to memory).
- The memory of current is not virtually aliased with any of the other variables used in the search program.

Note that it is possible to have virtual aliases in the array or for the first and last pointers, as these are read only.

6 Conclusions

In this paper we have presented two details of our approach to model the memory of an IA32 system. The first detail is a specification of well-behaved *plain memory*. This specification allows us to maintain an abstract level of reasoning with reasonable efficiency on top of a complex model of paged virtual memory. The second detail is our modeling of memory-mapped devices and reserved bit restrictions. We use side-effect state transformers that are performed before and after memory access to uniformly model both, reserved bits and memory-mapped devices. As demonstrations we have included the formalization of a (memory-mapped) random number generator, and an example verification of a simple C++ code fragment.

Related Work. There have been extensive attempts to reconcile the untyped memory model of C with a typed view, see e.g. [10]. At the other end of the spectrum, complete micro-processors have been formally verified at the gate level [1]. Our current work is located in-between those efforts. We take a correct IA32 processor for granted. Our aim then is to establish that the view on memory provided by the Robin micro-hypervisor, despite various peculiarities present in the architecture's hardware (virtual vs. physical memory, memory-mapped devices, etc.), is a model of C++ memory, which is well-behaved in the sense that one does not need to worry about low-level features like virtual address aliasing anymore. Perhaps most closely related is the verification of page table algorithms in [9], which still uses a rather abstract memory model however.

Acknowledgments We would like to thank the anonymous referees for their valuable suggestions.

References

[1] Beyer, S., C. Jacobi, D. Kröning, D. Leinenbach and W. J. Paul, *Putting it all together—formal verification of the VAMP*, STTT Journal, Special Issue on Recent Advances in Hardware Verification **8** (2006), pp. 411–430.

[2] Hohmuth, M. and H. Tews, *The semantics of C++ data types: Towards verifying low-level system components*, in: D. Basin and B. Wolff, editors, *16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2003). Emerging Trends Proceedings* (2003), pp. 127–144, technical Report No. 187.

[3] Huisman, M. and B. Jacobs, *Java program verification via a Hoare logic with abrupt termination*, in: T. Maibaum, editor, *Fundamental Approaches to Software Engineering*, Lecture Notes in Computer Science **1783** (2000), pp. 284–303.

[4] Intel Corporation, Denver, CO, "Intel 64 and IA-32 Architectures Software Developer's Manual," (2007), order Number: 25366[5-9]-023US.

[5] Owre, S., S. Rajan, J. Rushby, N. Shankar and M. Srivas, *PVS: Combining specification, proof checking, and model checking*, in: R. Alur and T. Henzinger, editors, *Computer Aided Verification*, Lecture Notes in Computer Science **1102** (1996), pp. 411–414.

[6] Tews, H., *Micro hypervisor verification: Possible approaches and relevant properties*, in: *NLUUG Voorjaarsconferentie 2007: Virtualisatie*, 2007, pp. 96–109.

[7] Tews, H., *Olmar: manipulating C and C++ abstract syntax trees in OCaml*, in: H. Tews, editor, *Proceedings of the C/C++ Verification Workshop*, 2007, pp. 103–113, technical report ICIS-R07015, Radboud University Nijmegen.

[8] Tews, H., B. Jacobs, E. Poll, M. van Eekelen and P. van Rossum, *Specification and verification of the Nova microhypervisor* (2007), deliverable D.6 of the Robin project, available at http://www.cs.ru.nl/~tews.

[9] Tuch, H. and G. Klein, *Verifying the L4 virtual memory subsystem*, in: G. Klein, editor, *Proc. NICTA Formal Methods Workshop on Operating Systems Verification*, NICTA Technical Report 0401005T-1 (2004), pp. 73–97.

[10] Tuch, H., G. Klein and M. Norrish, *Types, bytes, and separation logic*, in: M. Hofmann and M. Felleisen, editors, *Proc. 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'07)*, Nice, France, 2007, pp. 97–108.