J Autom Reasoning (2009) 42:189–227
DOI 10.1007/s10817-009-9122-0

Formal Memory Models for the Verification of Low-Level Operating-System Code

Hendrik Tews · Marcus Völp · Tjark Weber

Received: 26 February 2009 / Accepted: 26 February 2009 / Published online: 23 April 2009

Abstract This article contributes to the field of operating-systems verification. It presents a formalization of virtual memory that extends to memory-mapped devices. Our formalization consists of a stack of three detailed formal memory models: physical memory (i.e., RAM), physically-addressable memory-mapped devices (including their respective side effects, access and alignment requirements), and page-table based virtual memory. Each model is formally shown to satisfy the plain-memory specification, a memory abstraction that enables efficient reasoning for type-correct programs.

This stack of memory models was developed in an attempt to verify Nova, the Robin micro-hypervisor. It is a key component of our verification environment for operating-system kernels based on the interactive theorem prover PVS.

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. Certain guarantees, such as that memory behaves well without strange effects, hold for application programs, because they are provided

This work has been supported by the European Union through PASR grant 104600.

Hendrik Tews
Institute for Computing and Information Sciences
Radboud Universiteit Nijmegen, The Netherlands
Homepage: `http://www.cs.ru.nl/~tews`

Marcus Völp
Institute for System Architecture
Technische Universität Dresden, Germany
Homepage: `http://os.inf.tu-dresden.de/~voelp`

Tjark Weber
Computer Laboratory
University of Cambridge, UK
Homepage: `http://www.cl.cam.ac.uk/~tw333/`

by the operating system. However, these guarantees are not in general valid inside operating-system kernels. Consequently, a verification environment for kernels must differ from that of application programs. Consider, for instance, the property that variables change only when explicitly modified (directly or via an alias). For application programs this property can be built into the verification environment. For operating systems, however, the property is not a priori true. Therefore, a verification environment for operating systems must model many more details of the execution environment to allow this property to fail, thereby permitting a sound verification of the kernel code.

In this article, we focus on two points that are different in kernel programming: control over virtual memory, and direct hardware access. For the first point, virtual memory, note that operating-system kernels must manage the data structures that control the mapping from virtual addresses to physical addresses. Simple errors can therefore lead to a situation where the contents of a relatively large address range change without accessing it. Additionally, kernels often map the same memory contents to different virtual address ranges for optimization purposes. We use the term *virtual alias* to refer to this phenomenon, where a certain memory area is available at different (virtual) addresses. Certainly, a verification environment for kernels must model virtual aliases correctly.

For the second point, direct hardware access, note that hardware registers and memory-mapped devices have very special behavior when read or written. Reading or writing might have side effects, that is, trigger special actions in the hardware that change the state of the system substantially. For instance, whether interrupts are enabled or disabled is controlled by writing to the `EFLAGS` register on an IA32 platform [14]. Apart from side effects, one has to pay attention to alignment rules (the read or write access must be done with a certain base address and a certain access size) and reserved bits (when writing, certain bits must not be changed). Violating these requirements might result in a wide range of possible behavior, ranging from correct execution to an immediate crash of the processor. The precise behavior often depends on the processor model. The verification environment must therefore very carefully model all side effects and implement all alignment and reserved-bit checks.

In this article we present our solutions for modeling virtual memory and access to devices in a kernel verification environment. The material has been developed for the verification of a micro-hypervisor running on the IA32 platform. Therefore, when it comes to details (e.g., register names or page-table formats) our descriptions are IA32 specific. The general setup and our modeling techniques, however, apply to all platforms that have memory-mapped devices and where the kernel runs in virtual memory.

The main contribution of this article is a stack of three detailed formal memory models:

– *physical memory*, i.e., RAM (Section 3.2.1);
– *device memory*, an abstract memory model whose concrete instances contain memory-mapped devices. Device memory models their side effects, access and alignment restrictions, and reserved bits in device registers (Section 4); and
– *paged virtual memory*, which performs address translation and corresponding access-permission checks via multi-level page tables (Section 5).

All memory models presented are formally shown to satisfy the *plain-memory* specification, a memory abstraction that captures virtual aliases, permits the verification of page-table modifications, and supports efficient reasoning about well-behaved code

(Section 3). The stack of memory models remains flexible though, and further layers may easily be added, because each plain-memory proof (with the exception of the proof for physical memory, which is the lowermost layer) assumes as one of its preconditions the plain-memory property for the respective underlying layer. The device memory model makes plain-memory proofs particularly easy for memory-mapped devices that have visible side effects and restrictions only on their device registers.

In addition to the core contributions, Section 2 provides an overview of our verification environment, and Section 6 contains a verification example. Section 7 discusses related work, and Section 8 concludes. This article is an extended and revised version of [25].

The solutions that we present in this article rely on a combination of over-approximation and underspecification, where especially underspecification plays a very important role. Over-approximation is used for features that are present on the IA32 platform but never used in our verification target. For instance, the bit that controls virtual x86 mode is modeled in such a way that any verification attempt fails when it is switched on.

Underspecification is typically used where the relevant hardware manuals do not precisely specify the behavior. In addition, we use underspecification to ensure datatype correctness for hardware operations and the software that we verify. For instance, the page-table walks that are implemented in a specific part of our hardware model (see Section 5) use typed memory accesses for reading and writing page-table entries. These typed accesses leave some important details underspecified. As a result, every read access generates a proof obligation that data of the right type is contained in the memory (see Section 2.3 for details). This prevents our hardware model from, for instance, interpreting a string as a page-table entry, which we consider an error (although the resulting behavior is precisely defined).

The specification and modeling techniques presented here are key components of the Robin verification environment. Robin was a European project that ended in April 2008, with partners from Germany, France, and the Netherlands. The Robin project focused on the further development of the Nizza security architecture [12]. One major goal was the design and implementation of Nova, a new micro-hypervisor for the IA32 architecture, in parallel with the development of formal methods for the verification of this new micro-hypervisor [24,26,27]. For more information on Robin see [22, 33].

The formal methods part of the Robin project builds on the earlier VFiasco project [10,32]. The formalization of IA32 hardware and the plain memory specification in the interactive theorem prover PVS [21] were started in VFiasco, but had not reached a state that permitted verification of actual code. The work was continued in Robin, finally building a detailed verification environment for operating-system kernels (and other low-level code) in PVS.

The results of this work are collected at `http://www.cs.ru.nl/~tews/Robin`. Besides the project deliverables, this web page also contains all relevant software, including our PVS repository. All PVS code in this article is taken directly from this repository. All lemmas and theorems contained in this article have been proven in PVS.

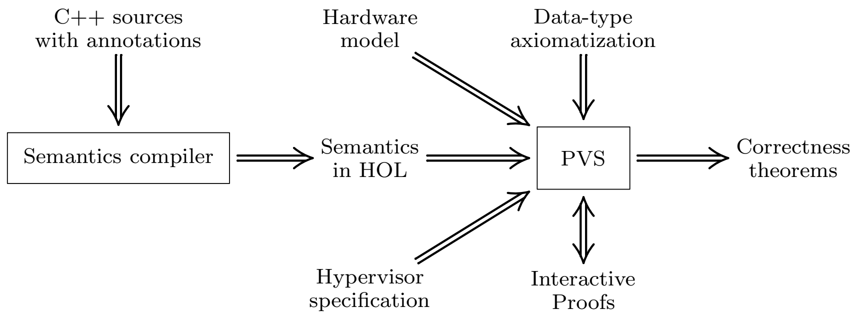

Figure 2 Source code verification (data flow)

2 Overview of the Robin Verification Environment

This section provides some background information about our verification environment for the Nova micro-hypervisor, which has been developed in Robin. According to the classification given in [13], Nova is a type 1 hypervisor (i.e., running directly on the hardware platform, with no operating system underneath) for the common IA32 architecture. It is written in C++ with some inline assembly. More technical information can be obtained from [26] and [27].

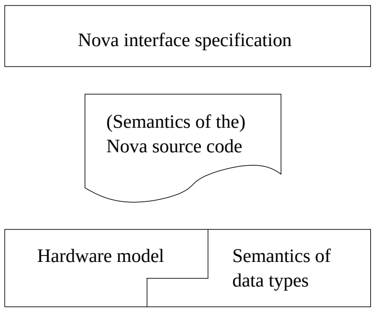

Fig. 1 Robin verification approach

Figure 1 illustrates our verification approach. We perform source-code verification (i.e., C++ code with inline assembly statements) in the interactive theorem prover PVS [21]. The input language of PVS is higher-order logic enriched with predicate subtyping and dependent record types. As Figure 1 shows, we have modeled parts of the IA32 hardware and of the semantics of C++ data types in PVS. These two blocks provide the basic operations for the semantics of the Nova source code. On top of the generated semantics, specifications of Nova's behavior can be written as formulas in PVS. One can then use the prover component of PVS to establish the validity of these formulas. Technically we show

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

where φ is a property of the Nova source code, such as *absence of runtime type errors*, or that a particular function or system call behaves as specified in the Nova specification [26]. Our verification results describe properties of the source code. A formal lifting of the results to object code (which would eliminate the correctness of the C++ compiler from our assumptions) was not part of the Robin project.

Figure 2 depicts the data flow of our verification approach. A semantics compiler translates Nova's C++ sources into their semantics in higher-order logic in PVS. Using this semantics, which draws upon the underlying hardware model and data-type axiomatization, the theorem prover then generates verification conditions (i.e., proof obligations required to show that Nova meets its formal specification). These are solved by interactive proof, ultimately establishing correctness of the Nova implementation.

In the following subsections we elaborate in more detail on important aspects of the two base blocks of our verification environment, the hardware model and the data type axiomatization.

2.1 State Transformers

The IA32 hardware is modeled in PVS as a set of its possible states, where each state contains the contents of the physical memory and the hardware registers. The operations on 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

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

Here Data is a type parameter for the result of the state transformer (if it terminates successfully), and State is the type of all possible states of the hardware model. Hence, expressions may have side effects. For example, reading a value in virtual memory sets the page-accessed bit in the corresponding page-table entries.

Because both State and Data are type parameters of our PVS theory (as opposed to types that have a concrete definition), our formalization of the C++ semantics is polymorphic in the underlying hardware model and datatype semantics. In the verification of concrete C++ programs, 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. In the latter case, the verification results automatically hold for all hardware models for which the plain-memory abstraction was established.

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. Non-termination is an unrecoverable error. (Operating-system kernels should not terminate, but only in a very general sense. Looking at details, every system call should terminate, possibly by returning to a different thread of execution.) Fatal is reserved for other unrecoverable errors, e.g., attempting to write 3 bytes into the register `EAX`. We want to rule out both Hang and Fatal by verification. A result of the form Exception models interrupts that will be handled by the micro-hypervisor. It also models exceptions [IA32-1-6.4][1] that IA32 instructions—and consequently C++ expressions implemented with these instructions—may generate: e.g., the `div` instruction may cause a divide error.[2]

Statement state transformers are functions of type

$$\mathsf{State} \to \mathsf{StmtResult}[\mathsf{State}],$$

where StmtResult is defined as follows:

StmtResult[State : **Type**] : **Datatype**
Begin
  OK(state: State) : OK?
  Break(state: State) : Break?
  Continue(state: State) : Continue?
  Return(state: State) : Return?
  Switch(state: State, case: int) : Switch?
  Default(state: State) : Default?
  Exception(ex_type: Exception_type, state: State) : Exception?
  Fatal : Fatal?
  Hang : Hang?
End StmtResult

The main difference between ExprResult and StmtResult is that the latter additionally contains the variants Break, Continue, Return, Switch and Default to model C++ control-flow statements, similar to [11]. Statements that follow after one of these control-flow statements in the body of a loop or function, and likewise **case** statements that do not match the selected case, must be skipped in the semantics. We collectively refer to these variants as *abnormalities*, indicating that they disrupt the normal (i.e., sequential) control flow. Moreover, since C++ statements do not denote a value, OK does not have a data argument.

Note that Return has no data argument either. In an earlier version StmtResult had a Data type parameter for the return value of C++ functions, which was stored in the Return abnormality. However, for certain compound types (e.g., classes), the C++ compiler allocates memory for the return value before the function call, and passes an additional pointer to the allocated memory into the function. To detect incorrect stack frame manipulations we have to model this behavior in our verification environment. Therefore, the Return abnormality does not carry the function return value any longer. For scalar types (e.g., integers, pointers), the compiler (and our verification environment) only allocates a register (usually `EAX`) for the return value. We leave this register address underspecified to remain independent of the calling convention of the C++ compiler.

[1] The notation [IA32-1-6.4] refers to Volume 1, Section 6.4 of the *Intel 64 and IA-32 Architectures Software Developer's Manual* [14].

[2] Exception does *not* model C++ exceptions. The Nova developers consider C++ exceptions too heavyweight to be used in an operating-system kernel. Therefore, they are outside the fragment of C++ treated in the Robin project.

State transformers can be composed. 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. For convenience we overload the ##-operator for functions g that take an additional argument of type Data (and return a state transformer): f ## λ(x: Data): g(x) means that the state transformer g(d) is applied to state s if f returns OK(s, d), i.e., (f ## λ(x: Data): g(x)) = (f ## g(d)) in this case. Otherwise, g is discarded as described before.

In various contexts statement and expression state transformers must be mixed. For instance, to describe an invariant, all statement and expression state transformers that maintain the invariant must be collected in one set. We therefore define SuperResult as a common supertype of both ExprResult and StmtResult. SuperResult only distinguishes normal termination (generalizing OK), abnormal termination with a successor state (generalizing Exception and the C++ control-flow abnormalities, such as Return), and abnormal termination without successor state (generalizing Hang and Fatal). The PVS definition of SuperResult is as follows.

```
SuperResult[State : Type] : Datatype
Begin
  OK(state: State)         : OK?
  Abnormal(state: State)   : Abnormal?
  Bottom                   : Bottom?
End SuperResult
```

A function of type State $\rightarrow$ SuperResult[State] is called a *super transformer*. Functions expr_2_super and stmt_2_super convert expression and statement state transformers, respectively, into super transformers by a case distinction on their result. The conversion into super transformers is only done for typing reasons, semantically it is irrelevant.

2.2 The Robin Hardware Model: A Collection of Stackable Memory Models

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, all general-purpose registers, selected special-purpose registers, and a general formalization of memory-mapped devices.

The hardware model does not fully implement the behavior of the real hardware. Instead a combination of over-approximation and underspecification is employed. Certain operations, such as enabling virtual x86 mode, directly yield a verification error, i.e., Fatal. For other operations, the semantics is constructed in such a way that any use of it yields an unprovable proof obligation during the verification. 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 unjustified, and not available in our verification environment.) This latter kind of error checking even works for hardware-initiated page-table traversals during address translation.

On the IA32 architecture, both application programs and the operating-system kernel typically run in virtual memory. This means that the addresses that are contained

Memory model	Features
segmented memory	segment checking, segment-based address translation
TLB	TLB consistency check
linear memory	page-table based address translation (see Section 5)
APIC device	IA32 Advanced Programmable Interrupt Controller (APIC)
physical memory	finite physical memory, processor registers

Figure 3 The stack of memory models that forms the Robin IA32 hardware model

in the object code are first translated into physical addresses before the main memory is accessed. Special data structures, namely segment registers and page tables, which are automatically traversed by the processor, control the translation from virtual to physical addresses. If the address translation fails (because some region of virtual addresses is not backed up by physical memory), the processor transfers control to the page-fault handler, which typically resides in the operating-system kernel. Eventually the page-fault handler solves the problem by either adjusting the segment registers and/or the page tables, or by killing the execution thread that caused the page fault.

The address translation on IA32 processors involves two translation steps: (i) segmentation, and (ii) page-table translation. In the first translation step, segment-local addresses, which are called *logical addresses* in the hardware manuals [IA32-3a-3.1], are translated into *linear addresses* of a flat 32-bit address space by addition of the segment base address. In the second translation step, linear addresses are translated into physical addresses with the help of multi-level page tables. This second step involves a Translation Lookaside Buffer (TLB) [15]. The TLB is a special CPU-internal cache. It stores results from the linear-address translation and requires explicit invalidation when the corresponding page-table entries are modified.

To reduce the overall complexity, we have decided to split the modeling of the IA32 hardware into several *memory models*. For a list of all our memory models (current and planned) see Figure 3. Each such memory model formalizes a particular feature set of the IA32 hardware. For instance, the model of physical memory formalizes access to the main memory via physical addresses. As its internal state, it maintains a function from Address to Byte to model the contents of the main memory and the processor registers.

To provide the complete functionality of the hardware model, the different memory models are stacked on top of each other, with the model of physical memory at the bottom. The generated semantics of the Nova source code and our specifications only access the topmost memory model. To provide its service, each memory model typically performs some computation (e.g., address translation), collects data from the underlying model, and combines everything into a result.

The *linear memory model* is named after the term *linear address* that is used in the IA32 manuals. The linear memory model contains page tables and the translation from linear to physical addresses. It is explained in detail in Section 5. For the actual contents of the memory it relies on some underlying memory model, e.g., physical memory.

The TLB memory model and the segment model are future work. They will cover TLB consistency checking and segment-based translation from logical to linear addresses, respectively. Note that on the IA32 architecture one can disable neither the

TLB nor segment-based address translation. The Robin hardware model is therefore somewhat incomplete until these two memory models have been implemented.

It is convenient to also model certain devices, especially memory-mapped devices, as memory models that are inserted at the appropriate place into the stack of memory models. For instance, we are currently building a model of the Advanced Programmable Interrupt Controller (APIC) that contains the memory-mapped APIC control registers. Accesses to addresses outside of the APIC registers are simply passed on to the underlying memory model, which should typically be physical memory. In this article, we first focus on the plain-memory abstraction (Section 3). The modeling of devices is described afterwards in Section 4.

All memory models share a common interface. Before we can discuss this interface, we have to introduce the type Address. We observe that reserved-bit checks must not only be done for memory-mapped devices, but also for registers. Further, accessing typed data in registers behaves the same as accessing typed data in main memory. Therefore, we combine register addresses and memory addresses into a uniform address space. This effectively halves the number of functions required in the abstract memory interface. Interestingly, such a uniform address space also provides the core of an assembly semantics for free, see Section 2.3. Because of all these reasons we define Address 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, but also other small numbers are permitted (e.g., 1 for accessing register `AH` inside `EAX`). The sizes and possible offsets of hardware registers are enforced with suitable side effects (see Section 4).

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]]],
  memory_read_side_effect    : [Address, list [Byte], bool →
                                      [State →  ExprResult[State,  list [Byte]]]],
  memory_write_side_effect   : [Address, list [Byte], bool →
                                      [State →  ExprResult[State,  list [Byte]]]]
#]
```

The operations in the memory structure form a suitable abstraction on top of which functions for reading and writing typed data can be defined polymorphically for all memory models (see Section 2.3). Our specifications and the generated Nova semantics refer to these derived functions only; they never use the above operations directly. We define a memory structure to contain these operations (rather than the derived functions) to ease the implementation of memory models.

The first two operations in the memory structure read and write one byte at the given address, respectively. Note that reading in memory can change the memory contents (for instance, set reference bits in the page table). The last two operations perform side effects. Collectively they are referred to as *side-effect state transformers.* Side-effect transformers model real side effects when accessing memory-mapped devices. Alignment, reserved bits, and access restrictions are also checked with appropriate side-effect transformers. If such a check fails, the side-effect transformer yields Fatal as result to abort any verification attempt. Side-effect transformers are always used in combination with a series of memory_read or memory_write operations (see below). Thereby the side-effect transformers are solely responsible for performing the side effect, and for checking the relevant constraints.

In general, the phenomena that we model with side effects depend on the whole memory block that is read or written. Therefore, the side-effect transformers take the start address and the whole memory block (of type list[Byte]) as arguments. The third (Boolean) argument indicates whether the memory access has been split on a virtual-memory page boundary. This Boolean makes it possible to check for a subtle error condition, see Section 4.2. As part of their operation the side-effect transformers can change the memory block, or discontinue the execution by returning a result different from OK. We further elaborate on the use and potential of side-effect transformers in Section 4.

On top of the memory structure we define the following two functions. They combine side-effect-free byte-wise memory access with side-effect transformers to memory-block access functions.

```
memory_write_list(pm : Memory_struct)(addr : Address, bl : list[Byte]) :
                                        [State → ExprResult[State, Unit]] =
  memory_write_side_effect(pm)(addr, bl, false) ##
    λ(bl1 : list[Byte]) : memory_write_list_nse(pm)(addr, bl1)

memory_read_list(pm : Memory_struct)(addr : Address, size : nat) :
                                        [State → ExprResult[State, list[Byte]]] =
  memory_read_list_nse(pm)(addr, size) ##
    λ(bl1 : list[Byte]) : memory_read_side_effect(pm)(addr, bl1, false)
```

The functions memory_write_list_nse and memory_read_list_nse lift the byte-wise memory access from the memory structure to whole memory blocks (of type list[Byte]), performing one byte-wise access for each byte in the memory block. For writing, the side effect is performed before accessing the real memory. For reading, the side effect comes last.

2.3 C++ Data Type Semantics

The second base component of our verification environment is a formalization in PVS of various data types. It provides a suitable semantics for all C++ types used in Nova, and for all necessary hardware data types (such as page-table entries). The data-type semantics is independent of any memory model. Every scalar[3] data type has a semantic domain Data, and an object representation of type list[Byte]. The number of bytes in the object representation (i.e., the length of the byte list) is fixed. It is given by the data type's size field. A valid? predicate checks whether a given byte list is a valid object representation of some value of the type. Moreover, functions to_byte and from_byte convert data to and from their object representation. Writing and reading the object representation to and from memory is done with the above functions memory_write_list and memory_read_list, respectively, that are provided by

[3] In the C++ standard, arithmetic types, enumeration types and pointer types are collectively called scalar types [16, §3.9(10)].

an arbitrary underlying memory model. The function from_byte has a result of type lift[Data], because not every byte list forms a valid object representation. The type constructor lift adds a constant bottom to its type argument and keeps all elements of the type constructor in the form up(...).

```
Uninterpreted_data_type : Type = [#
  size : nat,
  valid? : [list[Byte], Address → bool]
#]

Interpreted_data_type : Type = [#
  uidt : Uninterpreted_data_type,
  to_byte : [Data, Address → list[Byte]],
  from_byte : [list[Byte], Address → lift[Data]]
#]
```

These two PVS types correspond to the different signatures of the semantics of non-scalar and scalar C++ types. Scalar types, such as integers and pointers, but also page-table entries, are modeled with Interpreted_data_type. Hence, they have a semantic domain with associated to_byte and from_byte functions. Non-scalar types, e.g., classes, unions, and arrays, are modeled as Uninterpreted_data_type. Consequently, they have no semantic domain, and no to_byte and from_byte functions. The following reasons motivate our unusually simple semantics for non-scalar types:

- A precise semantics for C++ structures, classes, and unions is very complex. Structures and classes can be partially initialized. Under certain circumstances unions can be used to perform implicit type casts (i.e., writing field a of the union, and then reading field b). Therefore, the semantic domain of a C++ union would have to be a certain quotient of the disjoint union of its fields.
- In C++ (and consequently also in our semantics), there are no operations that store or retrieve entire objects of, e.g., class type in or from memory at once. The language standard prescribes that assignment and copying is done field-by-field. Field access does not access the whole compound data object, instead it only accesses the portion of memory containing the field.
- For virtual method dispatch the validity of classes (as provided by valid?) is important to ensure that the virtual function table has not been corrupted. In our approach the semantics of virtual methods is relatively independent of the semantics of types.
- For structures, classes and unions one might often prefer a custom semantics anyway. For instance, for a class S with an **int** field and a pointer S∗, one might want to use list[int_semantics] as the semantics of properly initialized elements, rather than int_semantics × pointer_semantics.

A discussion of further details is beyond the scope of this article.

A number of properties must be satisfied by the semantics of every scalar data type. These properties are combined in a predicate interpreted_data_type? on Interpreted_data_type[Data]. We require that valid? must fail on object encodings whose length does not equal size, to_byte must produce valid encodings only, from_byte must succeed on valid encodings, and from_byte must be a left-inverse of to_byte (i.e., from_byte(to_byte(d, a), a) = d). Otherwise, however, to_byte and from_byte are largely underspecified. This underspecification is exploited by our data-type semantics to make the detection of erroneous type casts and wrong implicit type conversions possible. For instance, reading data from a C++ union with a wrong type will cause an unprovable proof obligation [9]. Because of the underspecification there is no property in the data-type formalization that ensures success of from_byte of type t_1 on the result of to_byte of type t_2.

The semantics of scalar data types is combined with memory access in the following two functions, which are parametric in the memory model pm and the data type dt:

```
read_data(pm : Memory_struct[State], dt : (interpreted_data_type?[Data]))
                                  (addr : Address) : [State → ExprResult[State, Data]] =
  (memory_read_list(pm)(addr, size(uidt(dt))) ##
     λ(bl : list[Byte]) : ok_lift(from_byte(dt)(bl, addr)))

write_data(pm : Memory_struct[State], dt : (interpreted_data_type?[Data]))
                      (addr : Address, data : Data) : [State → ExprResult[State, Unit]] =
  memory_write_list(pm)(addr, to_byte(dt)(data, addr))
```

The function read_data reads a scalar data type at address addr by reading the correct memory block and passing it to from_byte. The result of from_byte, which is of the PVS option type lift[Data], is mapped to ExprResult by the function ok_lift as follows: results of the form up($-$) are mapped to OK, and bottom is mapped to Fatal. The function write_data writes the memory block returned by to_byte to memory.

Because the Address type contains register addresses, read_data and write_data can be used to provide a semantics for many inline assembly statements, for which suitable data-type declarations can be inferred. For a lot of untyped assembly, that is, where no data-type declarations are available, memory_read_list and memory_write_list suffice to give a semantics. Note that important assembly instructions, such as loading the page-table base register (CR3) or disabling interrupts, are covered. We noticed this assembly semantics only long after the Address type and the above functions had been implemented in PVS. The uniform address space therefore provided us truly for free with the core of an assembly semantics.

In our design, the hardware model, the data-type semantics, and the C++ semantics are relatively independent of each other. Because of the underspecification of these models 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, for instance, can be verified against physical memory,
- to add additional axioms to the data-type semantics, e.g. to model compiler-specific assumptions about the size of data types or the precise behavior of certain type casts, and
- to adopt the semantics of new C++ features or compiler-specific C++ constructs.

2.4 Limitations and Assumptions

Formalizing the entire IA32 architecture, or the entire C++ standard, would be tremendous tasks well beyond the scope of the Robin project. Moreover, many features of a complete formalization would not be used anywhere in the Nova sources (or in any other contemporary operating-system kernel). Therefore, our hardware model and the C++ semantics are incomplete. The hardware model, for instance, does not contain virtual 8086 mode.

Most omissions, however, do *not* lead to global assumptions that would restrict the validity of our verification results. For example, we do not assume 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 C++ features that are missing in our semantics would trigger an assertion in the semantics compiler.

A number of features have been considered, but were not implemented within the limited time-frame of the Robin project. These features are (i) the Translation Lookaside Buffer (TLB), (ii) segment offset and segment size checking, (iii) cache policy checking for devices, and (iv) linking object code and instruction fetch to the C++ semantics. Because of their absence the Robin verification environment is currently unable to detect certain kinds of errors, namely

1. TLB errors, e.g., inconsistencies between the TLB and page tables, or implicit assumptions about the TLB's size and structure,
2. 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 registers are initialized with correct descriptor values),
3. cache policy errors and delayed side effects for cached memory-mapped devices,[4] and
4. discrepancies between our C++ semantics and the compiled object code. Apart from compiler bugs, these could occur for the following reasons: volatile-related errors in the source code (e.g., missing volatile annotations, missing memory fences),[5] certain compiler optimizations (e.g., delayed write-back to memory), or self-modifying code (however, no self-modifying code is contained in the Robin micro-hypervisor).

Moreover, our verification is based 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. On single-processor systems, this should be guaranteed by the hardware.
- The software tools involved—the C++ compiler used to compile the Robin micro-hypervisor, our semantics compiler, and PVS—produce correct results.

Note that in the Nova design, drivers for hard disks and other common devices are located in user space. Thus, apart from the Advanced Programmable Interrupt Controller (APIC) no devices need to be modeled for a complete verification of Nova.

[4] It is in general straightforward to add side effects to the device model that check whether the device is accessed with the proper caching policy (see Section 4). The source code that we are currently targeting does not involve devices that require a specific caching policy.

[5] C++ compilers are 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. This makes a correct semantics difficult. At the moment our C++ semantics treats all data as volatile.

3 Plain Memory

The memory in an IA32 system is a sophisticated device: segments and page tables specify access rights, a given region of memory might be visible in different virtual-address ranges, bogus TLB entries might cause the address translation from virtual to physical addresses in the CPU to differ from what is specified in the page table, and much more. When verifying kernel code we cannot ignore these effects, not even for most innocent code, because of the errors that they might cause.

As a consequence we have designed the *plain-memory* abstraction for the verification of those parts of the kernel that require only the standard C++ memory model. The plain-memory abstraction 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 detailed model of IA32 memory. Plain memory provides a (comparatively) simple abstraction that can be used for those parts of the sources that only access 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 implements these modes in multiple different memory models, see Section 2.2. Most of the code, however, 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.

3.1 The Plain-Memory Specification

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. The general idea is simple. Memory at blessed addresses is well-behaved: a read access does not change anything in the blessed address range, and a write access only changes the bytes written (in the expected way). The side-effect transformers must neither change the memory block nor any memory at blessed addresses. Moreover, these special properties are maintained as long as only blessed addresses are accessed. However, no guarantees are made for the memory contents at non-blessed addresses (even when only accessing blessed addresses), and for memory accesses outside the blessed address regions.

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 its properties with observations that can be made by reading and writing single bytes only, by referring to the Memory_struct interface that is common to all memory models. 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 verification proof will spawn a subgoal requiring a proof of the plain-memory axioms for the underlying memory model. The plain-memory signature is given by the following record.

```
Plain_Memory : Type = [#
  mem      : Memory_struct[State],   % see page 9 (Section 2.2)
  states   : PRED[State],            % states fulfilling the plain memory properties
  ro_addr  : PRED[Address],          % read−blessed addresses
  rw_addr  : PRED[Address]           % read/write−blessed addresses
#]
```

Here, the type constructor PRED constructs the type of all predicates over its argument type. For instance, PRED[State] is equal to [State → bool]. Record fields in PVS can be accessed with two equivalent syntaxes. If pm is of type Plain_Memory, then both mem(pm) and pm‘mem denote its first field.

The properties of plain memory are specified as follows.

```
plain_memory?(pm) : bool =
   unchanged_memory_invariant?(pm‘mem, pm‘states,
      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,
      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))
```

For better readability the rather involved expression denoting the union of all permitted state transformers is only shown once below. Its definition relies on the utility functions memory_read_transformers and memory_write_transformers to collect all state transformers that perform a read or write access, respectively, and on memory_read_side_effect_super_transformers / memory_write_side_effect_super_transformers to collect all side effects of the underlying memory model at a given set of addresses.

```
all permitted state transformers ≡
   union(union(memory_read_transformers(pm‘mem, union(pm‘ro_addr, pm‘rw_addr)),
               memory_write_transformers(pm‘mem, pm‘rw_addr)),
         union(memory_read_side_effect_super_transformers(pm‘mem,
                                                 union(pm‘ro_addr, pm‘rw_addr)),
               memory_write_side_effect_super_transformers(pm‘mem, pm‘rw_addr)))
```

The first clause in the definition of plain_memory? 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 read-blessed addresses (this implies that read-blessed and read/write-blessed addresses must be disjoint). The third clause requires that a write access to one address leaves the memory at all other read/write-blessed addresses unchanged. The fourth clause states that write accesses actually change the memory at the written address in the expected 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 desirable properties hold. The fifth clause requires that all memory accesses to blessed addresses terminate with OK. This prohibits, e.g., unhandled page-faults. The last two clauses require that side effects (which we discuss in Section 4) do not change the data read or written.

The plain-memory specification entails that only explicit writes change a memory location. 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) )
```

The lemma expresses that for two variables (of type dt1 and dt2, respectively) that lie disjoint in blessed memory, writing the first one does not change the contents of the second. The proof uses induction on the size of dt1 to show that the memory_write_list call that is inside write_data does not change the result of the memory_read_list call that is inside read_data. Then the result follows.

For program verification, the lemma is part of a rewriting engine that allows PVS to symbolically compute the value of a variable by going back to the last write access to that variable, while ignoring memory accesses to other variables.

3.2 Establishing Plain Memory

In Section 6, we show in more detail how the plain-memory abstraction simplifies the verification of C++ programs. To avoid vacuous results, however, the plain-memory property must be established for the underlying memory model. For each memory model contained in our PVS formalization of the IA32 architecture, we have established the plain-memory property for a suitable range of addresses. These results discharge the plain-memory assumptions on our verification results.

The Robin hardware formalization contains plain-memory results for physical memory (RAM), for memory containing devices, and for linear memory. In the following subsection we detail the result for physical memory. The plain-memory result for devices is shown in Section 4.3. A description of the linear memory model with its plain-memory result follows in Section 5.

3.2.1 Establishing Plain Memory for Physical Memory

As a base of all other memory models, physical memory provides one byte of storage for every address between a certain minimum and maximum. Accesses above the maximum or below the minimum yield Fatal as result. Unsurprisingly we can prove in PVS that all states of physical memory form a plain memory, with all addresses between the minimum and the maximum read/write-blessed. The plain-memory structure for physical memory is thus given by

```
phy_pm : Plain_Memory[Physical_memory] = (#
  mem     := phy_mem,                 % Memory_struct of the physical memory model
  states  := fullset [Physical_memory],
  ro_addr := emptyset[Address],       % all addresses are read/write blessed
  rw_addr := in_memory(min, max),
#)
```

We establish the plain-memory property with the following lemma.

phy_mem_plain_memory : **Lemma** plain_memory?(phy_pm)

The proof is rather trivial because our physical memory is simply a byte array, without any side effects.

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.

Many aspects that occur when reading and writing memory-mapped device registers can also be found in special-purpose processor registers (such as the IA32 control registers [IA32-3a-2.5]). In our verification environment, we therefore treat both memory-mapped device registers and special-purpose processor registers in a similar way. For verification, the following effects are important.

Access type restrictions. Some device registers and also some special-purpose processor registers are read-only or 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's base address. Furthermore, each access must read or write a certain amount of data at once. The registers of the IA32 Advanced Programmable Interrupt Controller (APIC) provide an example. They are aligned on 16-byte boundaries, and they must be accessed with 4-byte wide and 4-byte aligned reads and writes [IA32-3a-8.4.1].

Reserved bits. The value of reserved bits must not be modified, or otherwise the processor's behavior is undefined. For instance, bits 0–2 and bits 5–11 of the IA32 page-table base register (`CR3`) are reserved [IA32-3a-2.5]. Bits of memory-mapped device register may also be marked as "reserved", and modifying these bits may cause similar side effects.

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 immediate delivery of the next pending interrupt. Similar effects can be observed for special-purpose processor registers.

More abstractly, we can summarize these effects as follows: reading and writing certain registers and certain locations in memory may cause 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

One popular way to describe external devices is to model the CPU and the device as separate entities that execute in parallel. This approach is well-suited for complex devices that perform some computation in parallel and asynchronously with the processor. However, this approach makes the verification much more complicated, because one has to reason about all possible interleavings of state changes of the processor and the device.

For the verification of the Nova micro-hypervisor we only need to model relatively simple devices. We therefore aimed at a modeling approach that would avoid the additional verification costs for parallel executing entities. Our key insight here is that the devices that we need can be modeled in a synchronous way. That is, these devices only perform state changes when the processor accesses them. They can therefore be modeled sufficiently precise with side-effect transformers, which can perform a state change in the device on every access. A device can then conveniently be modeled as a separate memory model. With this approach reasoning about interleavings of state changes is not necessary, and the verification does not become substantially more difficult.

To some extent, we can even model asynchronous state changes in devices. To model external interrupt sources, the APIC model can define a side-effect transformer that is executed at *every* memory access and that non-deterministically signals an external interrupt. The memory access counter described in Section 4.1.3 below provides another example for an asynchronous device.

In the following we illustrate our approach by providing several examples for the side-effect transformers memory_read_side_effect and memory_write_side_effect. First, we recall their type from Section 2.2 (on page 9):

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

As explained before, the second argument of these side-effect transformers is the entire memory block read from (or written to) the given address. Having the address and the complete memory block available provides high flexibility and makes it possible to implement all the effects discussed above. Checking alignment, for instance, requires the address and the length of the memory block, while checking for reserved bits requires the contents of the memory block.[6] The third parameter indicates whether the access crossed a page boundary in a memory layer stacked on top. We elaborate on the use of this parameter in Section 4.2.

We use a simple memory-mapped random number generator (RNG) as an example to illustrate how one can model side effects and access restrictions. The RNG device

[6] In principle one could check for reserved bits inside memory_write and use side-effect transformers only for, e.g., alignment checks. Then it would be sufficient to pass the length of the memory block instead of the block itself into the side-effect transformer. We did not investigate such a solution, because, for modularity reasons, we prefer to separate the conventional memory interface from the side-effect interface and implement side effects in side-effect transformers only.

provides one read-only memory-mapped register, rnd_val, that contains an unspecified (supposedly "random") value. We have implemented this device as a separate memory model in PVS, which adds its functionality to an arbitrary underlying memory model (cf. Section 2.2). The memory state is augmented with the internal state of the RNG device, which contains two non-negative integers, 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, a completely unspecified function random is applied to the seed and the current access_count. Since access_count is strictly increasing, we obtain potentially different values for every access to rnd_val in our verification environment.

Underspecified and nondeterministic behavior of more complicated devices could be modeled in a similar fashion. For example, interrupt-generating devices could be modeled through underspecified side-effect transformers that set the interrupt pending flag in the interrupt controller. Masking and demasking of interrupts in the CPU and in the interrupt controller correspond to writes to the respective registers (e.g., the I-Flag in the `EFLAGS` CPU register [IA32-1-3.4.3] and the Mask Flags in the APIC Local Vector Table [IA32-3a-8.5.1]). The actual delivery is then simply a state transformer that evaluates these flags and changes the system state accordingly. [IA32-1-6.4] describes these state changes in detail.

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 a certain granularity.

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, word_size)) ∨
    (¬ cp ∧ length(bl) = word_size ∧ a = rnd_val)
  Then ok_result(bl)
  Else fatal_result **Endif**

Here word_size 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). The term ¬cp ensures that the memory access is not part of a larger memory access that crossed a page boundary (see Section 4.2).

To require the read-only behavior of the rnd_val register, we use the following side-effect state transformer 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 defined as compositions of the above two transformers and other checks, which we present below.

4.1.2 Reserved Bits

Reserved-bit restrictions come in two flavors, depending on whether the value of reserved bits is specified. For instance, the IA32 manual says that bits 11–31 of the `CR4` register must have value 0 [IA32-3a-2.5]. 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 adheres to the specification. One can then simply write to those registers whose reserved bits are completely specified.

Other special-purpose processor registers and device registers leave the value of certain reserved bits undefined, yet, the value of these bits may not be altered. 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 the register.

Reserved bits can also be used to restrict the processor modes in which the micro-hypervisor may execute. For instance, 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 hypervisor 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 hypervisor's bootstrapping code.

4.1.3 Side Effects

Side effects on memory reads or 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 (see Section 4.1, page 19), the RNG device implements a side effect to count all memory accesses in its internal state. Here we show its definition:

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

The following side effect read_rnd_val is part of the complete read side-effect transformer of our RNG device. It exploits the completely underspecified function random, which, for two arguments, returns an arbitrary byte list of length word_size. The function read_rnd_val checks whether the memory-mapped register rnd_val is accessed. If this is the case it discards the value from the underlying memory model and returns the result of random instead. Note that alignment restrictions are checked before in unaligned_access, see Section 4.1.1 (on page 19). When read_rnd_val is evaluated, the memory block read is either precisely the memory-mapped register rnd_val or completely disjoint from it.

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

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(get_seed(s), get_access_count(s))(s)
  Else ok_result(bl) Endif
```

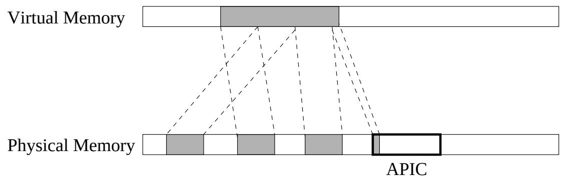

Figure 4 Splitting of side effects when stacking virtual memory on top of device memory

4.2 Stacking Memory with Side Effects

In Section 2.2, we described the stacking of different memory models. However, the combination of virtual memory with memory-mapped devices yields two complications. First, address translation must also be applied to the memory blocks that are supplied to one of the side-effect transformers. Thereby the memory block might be split at page boundaries into several smaller, non-contiguous blocks. For each such smaller block the side-effect transformer of the underlying memory must be called, and the results of all these calls must be suitably combined. The side-effect transformers of the linear memory model (see Section 5.2) perform this splitting and recombination of results.

Second, 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 4 the very last piece of the virtual address block is mapped precisely to the first APIC register. Such an access should be considered an error, because it is only part of a larger memory access. Moreover, the IA32 architecture gives no guarantee that in the case depicted in Figure 4, the access to the APIC is performed with 4-byte granularity [IA32-3a-7.1.1], as required by the APIC.

Checking for this kind of error is problematic because the memory layer that models the APIC is independent of the linear memory model, and therefore has no information whether splitting at page boundaries has occurred or not. One possible solution would be to pass a list of memory blocks as argument to the side-effect transformers. This list would contain more than one element if splitting had occurred. The downside of this list-of-memory-blocks approach is that every side-effect transformer has to perform its own iteration over the memory-block argument list.

In our solution to the problem, we introduce a Boolean crossed-page indicator cp as argument to the side-effect transformers of our memory models. Initially being false, this indicator is set to true when the address translation splits a contiguous memory access.

The side-effect transformer unaligned_access of the example RNG device (see Section 4.1.1, page 19) checks the cp flag every time the rnd_val register is accessed, and delivers an error if the flag is true. Use of the cp flag is further exemplified by the side-effect transformers for linear memory, see Section 5.2.

4.3 Establishing Plain Memory for Device Memory

A device typically keeps its state internally where it cannot directly be read or written. Furthermore, many devices cause side effects only on their memory-mapped registers and on their I/O ports.

In order to assist writing device models that only implement side-effect transformers (and simply forward memory accesses to their underlying memory layer), we developed Device_memory. In this memory layer, we maintain an additional Device_state next to the state of the underlying memory, and we lift (in em_lift below) a state transformer on the latter to a state transformer on Device_memory, keeping the Device_state unchanged. Device_memory gives rise to the following Memory_struct, which takes the device side effects as arguments.

```
device_pm(device_read_side_effect,
          device_write_side_effect) : Memory_struct[Device_memory] =
(#
   memory_read := λ(a : Address) :
     em_lift[Physical_memory, Device_state, Byte](memory_read(pm)(a)),
   memory_write := λ(a : Address, b : Byte) :
     em_lift[Physical_memory, Device_state, Unit](memory_write(pm)(a, b)),
   memory_read_side_effect :=
     λ(a : Address, bl : list[Byte], cp : bool) :
       em_lift[Physical_memory, Device_state, list[Byte]](
         memory_read_side_effect(pm)(a, bl, cp)) ## λ(bl1 : list[Byte]) :
         device_read_side_effect(a, bl1, cp),
   memory_write_side_effect :=
     λ(a : Address, bl : list[Byte], cp : bool) :
       em_lift[Physical_memory, Device_state, list[Byte]](
         memory_write_side_effect(pm)(a, bl, cp)) ## λ(bl1 : list[Byte]) :
         device_write_side_effect(a, bl1, cp)
#)
```

Like our other memory models, device memory is parametric in the underlying memory model pm. It can therefore be stacked on top of any other memory model.

The following lemma establishes that device memory is plain memory:

```
device_memory_plain_memory : Lemma
  is_device_plain_memory?(device_read_side_effect, device_write_side_effect)(pm)
    ⟹plain_memory?(pm)
```

Here is_device_plain_memory? encodes the necessary preconditions. It is defined as follows, where pm_phy is a plain-memory structure for the underlying memory model pm:

```
is_device_plain_memory?(device_read_side_effect, device_write_side_effect)
                         (pm) : bool =
    pm`mem = device_pm(device_read_side_effect, device_write_side_effect) ∧
    % the underlying memory is plain memory
    plain_memory?(pm_phy) ∧
    pm`states = em_lift(pm_phy`states) ∧
    subset?(pm`rw_addr, pm_phy`rw_addr) ∧
    subset?(pm`ro_addr,
      union(pm_phy`ro_addr, difference(pm_phy`rw_addr, pm`rw_addr))) ∧
    % properties about device side effect transformers
    transformers_ok?(pm`states, union(
      drse_super_transformers(union(pm`ro_addr, pm`rw_addr),
```

```
                                   device_read_side_effect),
      dwse_super_transformers(pm`rw_addr, device_write_side_effect))) ∧
    unchanged_memory_invariant?(pm`mem, pm`states, union(
      drse_super_transformers(union(pm`ro_addr, pm`rw_addr),
                                   device_read_side_effect),
      dwse_super_transformers(pm`rw_addr, device_write_side_effect)),
                                     union(pm`ro_addr, pm`rw_addr)) ∧
    side_effect_content_unchanged(union(pm`ro_addr, pm`rw_addr), pm`states,
                                       device_read_side_effect) ∧
    side_effect_content_unchanged(pm`rw_addr, pm`states,
                                       device_write_side_effect)
```

The first clause links the memory structure of this plain memory to the device memory structure. The clauses 2–5 establish that the underlying memory is plain memory, blessed states are underlying blessed states, write-blessed addresses are write-blessed in the underlying memory, and read-blessed addresses are read-blessed or write-blessed in the underlying memory (but not write-blessed in the device memory), respectively. Clauses 6–8 establish the required properties for the device side effects: side effects terminate with OK on blessed addresses, do not change blessed addresses, and change no data read to or written from blessed addresses.

To exemplify the use of device memory, we proved that the RNG device gives rise to a plain memory for all underlying physical states, and for all physically blessed addresses with the exception of the device register.

```
random_device_plain_memory : Lemma
  plain_memory?(pm_phy) ⟹ plain_memory?(random_device_pm(pm_phy))
```

The proof follows from the fact that side effects are only allowed to modify the extended device state. Proving this lemma in PVS (by establishing the above preconditions) took us less than an hour.

5 Linear Memory

Following Intel's terminology of *linear address* [IA32-3a-3.1], we use *linear memory* to denote a memory model that provides page-table based address translation, but no Translation Lookaside Buffer (TLB) or segment-based address translation. (Therefore, linear memory is merely an abstraction that does not exist in reality. On an IA32 platform, one can neither disable the TLB nor the segment-based address translation.)

Like all our memory models, linear memory defines a memory structure, where the address arguments of the functions memory_read and memory_write are now considered to be linear addresses. To access page-table entries and the contents of the translated physical address, these two functions call the underlying memory model. To keep the stacking flexible, the linear memory model is not directly based on a model of physical memory, but is instead parametrized with the Memory_struct pm of an arbitrary memory model. Therefore, linear memory can be stacked on physical memory, on device memory, or on any other memory model. Throughout this section the constant pm refers to the memory structure of the underlying memory. (In PVS, pm is a theory parameter that is instantiated when the stack of all memory models is formed, see Section 2.2.)

5.1 Address Translation

Functions memory_read and memory_write of the linear memory model translate their linear address argument to a physical address, check access rights in the corresponding page-table entries, update reference bits in these entries and finally access the memory content in the underlying 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)(pa)
    Else
      memory_read(pm)(a)
    Endif
  Else fatal_result Endif
```

The state-space type Linear_memory is equal to the state space of the underlying memory model. The constant Read, the second argument of linear_resolve, comes from an enumeration type Memory_access, which contains one element for each kind of memory access (namely Read, Write, and Execute for instruction fetch). 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 translated into the physical address pa, which is used in memory_read to access the underlying memory. Register accesses are passed to the underlying memory model without address translation.

The function linear_write, which fills the memory_write slot in the linear memory structure, is defined analogously to linear_read. We now introduce in more detail the function linear_resolve, which performs address translation and page-table updates. Readers who are not interested in the technical details of virtual memory on the IA32 architecture may wish to skip forward to Section 5.2 (on page 27), which discusses the side-effect transformers of our linear memory model.

We model linear_resolve precisely as described in [IA32-3a-3.7] for a configuration in which the two address extensions—physical address extension (PAE flag), and page-size extension (PSE-36 feature flag)—are disabled, and large pages (PSE flag) are enabled. In this mode, an IA32 processor implements two levels of page tables with 1024 entries each, and supports up to 4 GB of physical memory, with 4 KB and 4 MB pages.

Address translation follows the standard multi-level page-table algorithm, which is described in [IA32-3a-3.7.1] and [IA32-3a-3.7.2]. From the virtual address, the processor extracts the topmost 10 bits to index into the first-level page table. The entry at this index may either be not present (or be marked as read only for a write operation), in which case the processor generates a page fault, or it may refer to either a page (of size 4 MB) or to a second-level page-table, for which this translation step is repeated with the respective next highest 10 bits of the virtual address. The entries in this second table may either be not present, or they may refer to a page of size 4 KB. [IA32-3a-3.7.6] describes the layout and semantics of individual bits in page-directory (first level) and page-table (second level) entries.

Address translation involves similar checks and updates at each of the two levels. Hence, we formalize the updates and the translation required for a single level in the following generic function.

```
translate(lvl : Level, base, addr : Memory_Address_4G,
```

```
          access : Memory_access, priv : Memory_privilege)
            : [Physical_memory → ExprResult[Physical_memory, [bool, Address]]] =
  Let pe_addr = xlat_idx(lvl, base, addr) In
    (read_data(pm, paging_data_type(lvl))(pe_addr) ##
      λ(pe : (range_pt(lvl))) :
        If present?(pe) Then
          If accessible?(pe, access) ∧ privileged?(pe, priv) Then
            write_data(pm, paging_data_type(lvl))(pe_addr, set_reference(pe, access)) ##
              ok_result( (is_leaf?(pe), base(pe)) )
          Else
            raise_fault(priv, access, true, addr)
          Endif
        Else
          raise_fault(priv, access, false, addr)
        Endif)
```

Here the type Level is a two-element subtype of the non-negative integers, containing just 1 (for first-level page directories) and 2 (for second-level page tables). First, xlat_idx combines the page-table base address base, the virtual address addr, and the level lvl to determine the address pe_addr of the relevant page-table entry. Reading this entry with read_data will only terminate normally if a page-table entry has been previously written at the address pe_addr. Otherwise, because of the underspecification of from_byte, an unprovable proof obligation will remain. The function paging_data_type selects either the data type for page-directory entries (lvl = 1) or page table entries (lvl = 2). After reading the entry, translate performs two orthogonal privilege-level checks: first, whether the entry is present (a present entry implicitly grants read and execute access), and whether the entry is writable if the access mode was Write (accessible?(pe, access)); second, whether the page is user accessible when the processor executes in user mode (in privileged?(pe, priv)). In case any such check fails, translate generates a page fault with the function raise_fault, which we describe below. Otherwise, translate sets the accessed bit, and if the access was a write also the dirty bit. The function returns either false and the base address of the page table of the next level, or true and the base address of the memory page. In the first case, the translation is incomplete yet, and a further translation step indexing into the second level page table is required. translate also models this second translation step when invoked with lvl = 2 and with the base address of the second-level page table that the first translation step returns.

Page faults are reported with the abnormal ExprResult Exception. It carries all relevant information about the page fault, except for the page-fault address, which is written to the special register CR2 before returning the Exception result.

```
raise_fault(priv : Memory_privilege, access : Memory_access, present : bool,
            pfa : Memory_Address_4G)
    : [Physical_memory → ExprResult[Physical_memory, [bool, Address]]] =
  write_data(pm, address_data_type)(CR2, pfa) ##
  exception_result(Page_fault(
    (# reserved_bit_violation := false,
       privilege := priv,
       access := add(Read, access),
       present := present #)))
```

Because the PAE flag is clear, the processor will not generate page faults when a reserved bit is not set to 0 [IA32-3a-3.7.6].

We can now already establish important results about page-table traversals. Provided the base address of the page directory (or page table) is properly aligned to 4 KB [IA32-3a-3.6.2], and provided translate terminates with OK, it holds that:

1. translate returns a positive address below 4 GB,
2. translate returns a 4 KB aligned address if the page-directory entry refers to a page table, and
3. if translate returns a page base address (as opposed to the base address of the next-level page table), this address is properly aligned (i.e., it is either a 4 KB aligned address if translate is invoked on a page-table, or a 4 MB aligned address if translate is invoked on a page-directory).

For illustration, the lemma formalizing property 3 looks as follows.

```
translate_result_leaf : Lemma
  Forall (lvl : Level, base, addr : Memory_Address_4G, access : Memory_access,
          priv : Memory_privilege, s : Linear_memory) :
    aligned?(bits_per_level + pe_size)(offset(base)) ∧
    OK?(translate(lvl, base, addr, access, priv)(s)) ∧
    Proj_1(data(translate(lvl, base, addr, access, priv)(s)))
  ⟹
    aligned?(bus_width − bits_per_level ∗ lvl)
            (offset(Proj_2(data(translate(lvl, base, addr, access, priv)(s)))))
```

Here Proj_1 and Proj_2 respectively project out the Boolean and the address in the result of translate.

With bus_width = 32 bit and bits_per_level = 10 (as described above), the term bus_width − bits_per_level ∗ lvl evaluates to the exponent 22 (thus 4 MB) for the first level, and to the exponent 12 (thus 4 KB) for the second level.

On the basis of translate, the traversal function for one level in a page-directory structure, we can now define linear_resolve, which translates virtual into physical addresses.

```
linear_resolve(addr : Memory_Address_4G, access : Memory_access)
    : [Linear_memory → ExprResult[Linear_memory, Address]] =
  % 1. get current privilege level from cs
  (read_data(pm, segment_reg_data_type)(CS) ## λ(cs : Segment_Reg_type) :
     Let priv = segment_to_priv(cs) In
       % 2. read page−directory base register PDBR
       (read_data(pm, pdbr_data_type)(PDBR) ## λ(pdbr : Pdbr_type) :
          % 3. first level page−directory lookup
          Let lvl = pdir_lvl In
            (translate(lvl, pdbr‘base_addr, addr, access, priv) ##
               λ(leaf : bool,
                  base : {b : Memory_Address_4G |
                             If leaf
                             Then aligned?(bus_width − bits_per_level ∗ lvl)(offset(b))
                             Else aligned?(bits_per_level + pe_size)(offset(b))
                             Endif}) :
```

```
        If leaf Then
          ok_result(xlat_ofs(lvl, base, addr))
        Else
          % 4. second level page−table lookup
          Let lvl = ptab_lvl In
            (translate(lvl, base, addr, access, priv) ##
               λ(leaf : bool, base : Memory_Address_4G) :
                 ok_result(xlat_ofs(lvl, base, addr)))
        Endif)))
```

As explained before, the Memory_access is either Read, Write, or Execute, depending on the access kind to be performed on the virtual address addr. The constant pdir_lvl holds the page-directory level 1, and ptab_lvl the page-table level 2. The rather involved type expression for base, i.e., {b : Memory_Address_4G | . . . }, is a dependent type, containing just those b's that satisfy the predicate after the vertical bar |. In other words, base is 4 MB aligned if leaf is true, and 4 KB aligned if leaf is false.

The current privilege level (CPL) [IA32-3a-4.5] of the memory access, that is user (ring 3) or supervisor (rings 0–2) [IA32-3a-4.11.2], is stored in the code segment descriptor CS [IA32-3a-3.4.5]. Because the processor caches the results of the segment descriptor tables (GDT and LDT) when loading a segment [IA32-3a-3.5.1], and because writes to these tables may modify the descriptor without simultaneously updating the descriptor cache, we have to read the current privilege level from the hidden part of the code segment register [IA32-3.4.3]. The page table base register CR3 (or PDBR) [IA32-3a-2.5] contains (besides further cache-control bits) the physical address of the page directory. The individual fields of this register are formalized in the record type Pdbr_type. After translate returns the base address of a 4 MB (resp., 4 KB) page, we obtain the physical address by adding the lower 22 (resp., 12) bits of the virtual address to this base address.

5.2 Side Effects

Recall from Section 4.2 (on page 21) that the side-effect transformers for linear memory must also perform address translation in order to invoke the side-effect transformers of the underlying memory model. This may cause a split of the memory block along page boundaries. In that case the crossed-page indicator cp must be set to true to enable the relevant checks in the underlying memory models.

The read side-effect transformer for linear memory looks as follows.

```
linear_read_side_effect(a : Address, bl : list[Byte], cp : bool)
                    : [Linear_memory → ExprResult[Linear_memory, list[Byte]]] =
  If null?(bl) ∨
     . . . % access inside memory/register address range
  Then
    If Mem?(type_of(a)) Then
      % split list in page contained lists
      apply_side_effects(split(min_page, a, bl),
          linear_read_side_effect_in_page(length(split(min_page, a, bl)) > 1 ∨ cp))
    Else
      memory_read_side_effect(pm)(a, bl, cp)
```

```
    Endif
  Else
    fatal_result
  Endif
```

The definition of linear_write_side_effect is analog. Splitting along page boundaries and recombining the results is done by apply_side_effects. Its first argument is the list of split memory blocks as produced by split (which we show below). The second argument (here linear_read_side_effect_in_page(...)) is the function to be applied to each split block. The first argument of linear_read_side_effect_in_page is the crossed-page indicator to pass down to the underlying memory layer. It is true if split returns more than one block (because there was a page boundary in the original memory block), or if the cp flag that linear_read_side_effect received was true already.

```
linear_read_side_effect_in_page(cp : bool)(a : Memory_Address_4G, bl : list[Byte])(s) :
                                              ExprResult[Linear_memory, list[Byte]] =
  (linear_resolve(a, Read) ##
    Lambda(pa : Memory_Address_4G)(ns : Linear_memory) :
      memory_read_side_effect(pm)(pa, bl, cp)(s))(s)
```

The function linear_read_side_effect_in_page, which is applied to each block, performs address translation before calling the side-effect transformer of the underlying memory layer. Care must be taken that the additional address translation that our model introduces for side effects does not contribute anything to the final result state. This is achieved by passing the state s explicitly into the underlying side-effect transformer memory_read_side_effect(pm), thereby discarding the successor state of linear_resolve.

The definition of apply_side_effects is as follows.

```
apply_side_effects(l : list[[Memory_Address_4G, list[Byte]]],
                   f : [Memory_Address_4G, list[Byte] →
                        [Linear_memory →
                   ExprResult[Linear_memory, list[Byte]]]])(s)
    : ExprResult[Linear_memory, list[Byte]] =
  reduce(ok_result(null),
        λ(e : [Memory_Address_4G, list[Byte]],
                   r : [Linear_memory → ExprResult[Linear_memory, list[Byte]]]) :
            (r ## λ(tail : list[Byte]) :
               (f(e) ## λ(head : list[Byte]) : ok_result(append(head, tail))))
    )(l)(s)
```

Recall that the second argument f is the function to be applied to every split block in the list l. For reading, f is linear_read_side_effect_in_page(...). The function reduce is the standard algebraic folding function over lists, sometimes called fold_right or foldr. On the empty list it returns its first argument, i.e., ok_result(null). For a non-empty list it applies its second (functional) argument to the head, and to the recursively reduced tail of the list.

Finally we show the definition of split:

```
split(size : nat, a : Address, bl : list[Byte])
    : Recursive list[[Address, list[Byte]]] =
  Let e2size = expt(2, size) In
    If null?(bl) Then null Else
```

```
      If a + length(bl) ≤ floor(e2size)(a) + e2size Then
        cons( (a, bl), null)
      Else
        Let delta = rem(e2size)(offset(a)) In
          cons( (a, head(bl, e2size − delta)),
                split(size, floor(e2size)(a) + e2size, tail(bl, e2size − delta)))
      Endif
    Endif
  Measure length(bl),
```

This function splits a list of bytes starting at address a into a list of lists of bytes whose elements no longer range across 2^{size}-aligned addresses.

5.3 Establishing Plain Memory for Linear Memory

We can now establish that the plain-memory property is satisfied for our model of linear memory. Linear memory is plain memory under the following preconditions (which we combine in a predicate is_linear_plain_memory?):

- In all states, the code segment register (CS) determining the code privilege level, the page-table base register, and the page-directory and page-table entries that are accessed when translating blessed virtual addresses are identical up to reference bits.
- Any address 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 in the underlying (e.g., physical) plain memory, blessed read-only addresses map to blessed readable or writable physical addresses.
- There is no blessed shared-memory alias to a writable virtual address. (Virtual read-only regions may be shared arbitrarily.)
- Page-table entries that are accessed when translating blessed virtual addresses are outside of the physical memory area that can be accessed from blessed addresses.

Although these preconditions seem rather restrictive, they hold for most addresses and for most parts of the kernel code. Situations in which these preconditions are temporarily violated and for which they must be established again afterward (possibly for a different set of addresses) include: context switches changing the page directory (e.g., when switching to another process), allocation and deallocation of thread control blocks (a data structure that the kernel memory allocator accesses at a different address than the remaining code—only one location may remain in the blessed address range), device accesses (because the underlying address range is not blessed), and some page-table manipulations. We only exclude those page-table entries from the blessed addresses that are used to translate the blessed addresses themselves. When verifying the manipulation of other page-table entries, one can fully benefit from plain memory.

The following lemma contains the plain-memory result for linear memory:

```
linear_memory_plain_memory : Lemma
  is_linear_plain_memory?(pm) ⟹plain_memory?(pm)
```

We prove this lemma by first showing (in auxiliary lemmas) that the individual clauses of the plain-memory predicate hold. The proof of these lemmas requires about 18 additional nontrivial lemmas, in which we first establish the clauses of the plain-memory property, and additional results for linear_resolve. The proofs of all these lemmas proceeds in a straightforward way by expanding the relevant definitions and by case splitting. Here, the application of an interactive verification tool proved to be useful, as we could have easily overlooked one of these cases in a traditional pen-and-paper proof.

For space reasons, we clearly cannot show the whole proof in this article. We therefore present two of the intermediate lemmas for illustration. Please refer to our published PVS sources for more details of the complete proof. The first example here is a special result about translate, which is needed for both the page-directory level and the page-table level.

```
translate_unchanged_pm_phy_except_pe : Lemma
  Forall (base : PTab_Address, lin_a : Memory_Address_4G,
          access : Memory_access, priv : Memory_privilege) :
    union(pm‘ro_addr, pm‘rw_addr)(lin_a) ∧
    is_linear_plain_memory?(pm) ∧
    subset?(address_block(xlat_idx(lvl, base, lin_a), expt(2, pe_size)), pm_phy‘rw_addr) ∧
    (Forall(s : (pm‘states)) :
       OK?(read_data(pm_phy, paging_data_type(lvl))(xlat_idx(lvl, base, lin_a))(s))) ∧
    (Forall(s1, s2 : (pm‘states)) :
      set_reference(data(read_data(pm_phy, paging_data_type(lvl))
                            (xlat_idx(lvl, base, lin_a))(s1)), Write) =
        set_reference(data(read_data(pm_phy, paging_data_type(lvl))
                              (xlat_idx(lvl, base, lin_a))(s2)), Write))
  ⟹
    Let pe_addr = xlat_idx(lvl, base, lin_a) In
      unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
        singleton(expr_2_super(translate(lvl, base, lin_a, access, priv))),
        difference(union(pm_phy‘ro_addr, pm_phy‘rw_addr),
                   address_block(pe_addr, expt(2, pe_size))))
```

The lemma uses the utility predicate unchanged_memory_invariant? to express that translate (under various preconditions) leaves all the read- and write-blessed addresses unchanged, except for the page-directory or page-table entry at pe_addr. This lemma directly gives rise to a similar result about linear_resolve, which states that linear_resolve only changes the relevant page-directory and page-table entries, leaving the remaining blessed addresses alone. Consequently, read-blessed addresses will not change their value during address translation, and of the blessed addresses only the address written will change (to the value that was written). Side effects such as the setting of accessed and dirty bits in the page-table entries occur outside the blessed address range. Clauses 1 to 4 of the plain-memory specification (see Section 3.1) follow immediately from this lemma and from the respective facts of the underlying memory model.

The second example lemma that we show here is an intermediate result about linear_resolve. It is important with regard to our formalization of side effects for linear memory (Section 5.2).

```
linear_resolve_same_page_address : Lemma
  Forall (addr : Memory_Address_4G, delta : nat, ac : Memory_access) :
```

$$
\begin{aligned}
&\quad \mathsf{rem(expt(2,\ min\_page))(offset(addr))} + \mathsf{delta} < \mathsf{expt(2,\ min\_page)} \wedge \\
&\quad \mathsf{OK?(linear\_resolve(addr,\ ac)(s))} \\
&\Longrightarrow \\
&\quad \mathsf{data(linear\_resolve(addr,\ ac)(s))} + \mathsf{delta} = \mathsf{data(linear\_resolve(addr} + \mathsf{delta,\ ac)(s))}
\end{aligned}
$$

The lemma states that the address translation essentially remains unchanged when increasing the offset by delta, as long as one stays below the next page boundary. Consequently, when writing a list of n bytes to a contiguous memory block within a page, it is possible to deduce the translation of the remaining $n-1$ addresses from the translated base address.

6 Example Verification

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

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

value = rnd_device.rnd_val;
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 the search value is not present.

The verification of these properties proceeds according to the approach that was outlined in Figure 2 (on page 4). First, the C++ sources for the search program are translated into their semantics in PVS. This is done automatically by the semantics compiler, see Section 6.1. Second, 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 in any memory model that satisfies the plain-memory property (see Section 6.2). Third, to avoid vacuous results, 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), see Section 6.3. Because we have established program correctness and plain-memory properties separately, changing the memory model only requires validation of the plain-memory property for the new model.

6.1 C++ to Semantics Translation

The semantics compiler translates the above C++ code into its PVS semantics. Expression-to-statement (e2s) and lvalue-to-rvalue (l2r) conversions are made explicit. We only show the translation of the second part (lines 6–11) of the above code snippet.

```
e2s[State, Address]
   (assign(pm, dt_uint)(id(value), l2r(pm, dt_uint)(id(rnd_val)))) ##
e2s[State, Address]
   (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))))
          == l2r(pm, dt_uint)(id(value))),
    break,
    skip) ##
  e2s[State, Semantics_pointer](postinc(pm)(id(current)))),
```

Here first, last and current are addresses of disjointly allocated pointer variables; value is an unsigned integer variable, which we initialize with a random value obtained from the random device's rnd_val register. Currently we require disjointness of these variables in a precondition. With a complete formalization of memory allocation (which we were not able to finish within the Robin project), however, disjointness would be derived from the allocator model.

6.2 Verification Against the Plain-Memory Abstraction

Verification is currently done by fixing the size of the array to a small number and by unrolling loops automatically. A key point is the automatic simplification of variable read and write operations with the plain-memory rewriting rules. These are mostly independent of the methods used for the verification of the C++ program. It is therefore straightforward to replace loop unrolling by more sophisticated verification techniques (e.g., loop invariants), while still benefiting from plain-memory rewriting.

For a simplification with the plain-memory rewriting rules, the following preconditions were added for the program variables: in_blessed_memory and valid_in_mem. Condition in_blessed_memory states that a variable is allocated in blessed memory, valid_in_mem states that the memory contains a valid bit representation for a variable. Typically the latter is established by a previous write to that variable. For uninitialized variables it is part of the precondition.

In our C++ semantics, all expressions and most statements are expressed using a combination of only four different state transformers: read_data, write_data (for reading and writing typed data from and to memory, respectively), 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 lemma plain_memory_read_write_other_res shown in Section 3).

Similarly, suitable rewrite lemmas simplify 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. If data((expr ## b_ex)(s)) is true, we obtain expr ## b_ex ## stmt_if, otherwise expr ## b_ex ## stmt_else.

Likewise, statements stmt_if and stmt_else are rewritten 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 of the data-type or memory model.

6.3 Validation of the Plain-Memory Precondition

For each memory model we have established the plain-memory property for a certain range of addresses. As stated in Section 3.2, 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 cases the plain-memory property must be reestablished before one can proceed with the automatic simplification.

For the verification example shown, one needs the following preconditions to establish the plain-memory property for a memory stack consisting of linear memory on top of the RNG device on top of physical memory:

- The variables are allocated so that they do not overlap with a register of the RNG 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 location 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.

7 Related Work

For an overview of current and past verification efforts on operating-system kernels see [17]. Early efforts on operating system verification include the UCLA Secure Unix [34] and KIT [2]. The project on UCLA Secure Unix mainly focused on specifications. Functions that perform hardware access were axiomatized as well, but not proven correct with respect to a model of the underlying hardware. KIT featured a very rudimentary kernel of only a few hundred lines of assembly code that provides very basic operating-system features, but no virtual memory. KIT was the first formally verified kernel. However, it cannot be used as a basis of a realistic computer system.

In September 2008, the commercial INTEGRITY-178B Separation Kernel became the first operating system to receive the Common Criteria EAL-6+ certification. This involved a formal information-flow analysis on an abstraction of the C source code [4], carried out in the ACL2 theorem prover. The details are still to be published.

Even closer to our work are two still ongoing large-scale verification efforts: the L4.verified and the Verisoft project. The L4.verified project, originally scheduled to complete at the end of 2008, is formally establishing correctness of the seL4 microkernel in a verification environment based on the interactive theorem prover Isabelle/HOL [23]. In his Ph. D. thesis [20], Norrish models C memory as a map from addresses to bytes. In [30], Tuch and Klein impose a typed heap abstraction on this untyped, byte-oriented memory to simplify reasoning about type-correct programs. Tuch et al. [28, 29, 31] later combined this typed view with separation logic to address the problem of aliasing between variables of the same type. The problem of virtual-memory aliases, however, has been considered in the context of the L4.verified project only recently. Kolanski and Klein [18, 19] propose a lifting of separation logic to virtual memory. Their approach is more abstract than ours and uses a different model of "sliced" heaps, but has requirements roughly similar to plain memory to ensure the frame rule of separation logic. It has not been integrated into the seL4 verification environment yet. To the best of our knowledge, there has been no work on devices and their special requirements in the context of L4.verified yet.

The Verisoft project [1, 6, 8], aims at the complete verification of a computer system from an e-mail client down to the gate level of the processor. Verification of an operating system kernel is therefore one part of the Verisoft project. Besides the bigger scale, one important difference between Verisoft on the one hand, and L4.verified and Robin on the other hand, is that for Verisoft it is acceptable if the verified system runs orders of magnitude slower than comparable unverified software. The hardware basis of Verisoft is the completely verified VAMP processor [3], an academic 32-bit RISC CPU design that can be implemented on an FPGA. In contrast, L4.verified and Robin aim at verified kernels that run on contemporary PC hardware at competitive speeds.

The VAMP is considerably simpler than the IA32 architecture. In kernel mode the VAMP, much like the Power PC processor when translation is not re-enabled, runs without address translation. Thus, the problem of virtual address aliases—one of the main motivations for our work on the plain-memory abstraction—simply does not exist. Also our second motivation for plain memory—to verify code independently of whether it runs in real or virtual memory—is not of interest on the VAMP.

VAMOS, the operating-system kernel considered in Verisoft [5, 6], is written in a high-level language (C0, a simplified subset of C) with some inline assembly. The C0 code is verified in an abstract C0 machine, achieving independence of hardware details similar to our plain-memory rewrite system. However, inline assembly is verified directly with the VAMP assembly semantics. The results of the assembly verification are propagated upwards into the C0 model via a simulation relation. In contrast our plain-memory abstraction is also suited for assembly code (provided it only accesses blessed memory).

In Verisoft also external devices are handled, for instance a hard disk in [7]. Such devices can be accessed by port or memory-mapped I/O. The variables of the abstract C0 machine, however, can change only because of C0 assignments. Therefore, an important property of the simulation relating the C0 machine to the VAMP model is that no C0 variables are allocated at addresses that contain memory-mapped device registers. In contrast, in our model high-level language variables can be allocated in memory that is not blessed, either because it contains page tables, or because it belongs to a device and can therefore change spontaneously.

In summary, some problems of the IA32 architecture considered in this article do not exist on the VAMP and are, therefore, not relevant in Verisoft. For high-level code, both our plain-memory abstraction and the C0 machine enable verification on a suitable abstraction level. For assembly code and memory-mapped devices, however, the plain-memory model appears to be more flexible.

8 Summary

In this article we have presented an approach to formally model the memory of an IA32 system. The first contribution is a stack of formal memory models, ranging from physical to paged virtual memory. For each memory model, a *plain-memory* property was established in PVS. This specification of well-behaved memory allows to maintain an abstract level of reasoning with reasonable efficiency on top of complex memory models, as found in an IA32 system.

The second contribution 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, devices and reserved bits. Memory models with devices and reserved bits have been integrated into our stack of memory models, and the plain-memory property has been established for suitable subsets of their addresses.

Both techniques have been integrated into our verification environment based on the interactive theorem prover PVS. To illustrate their application, we have presented the formalization of a memory-mapped random number generator, and an example verification of a simple C++ code fragment.

Acknowledgements We would like to thank the anonymous referees for their time and their extremely valuable comments, which helped to improve this article substantially.

References

1. Eyad Alkassar, Mark A. Hillebrand, Dirk Leinenbach, Norbert W. Schirmer, and Artem Starostin. The Verisoft approach to systems verification. In Natarajan Shankar and Jim Woodcock, editors, *Verified Software: Theories, Tools, Experiments Second International Conference, VSTTE 2008, Toronto, Canada, October 6–9, 2008. Proceedings*, volume 5295 of *Lecture Notes in Computer Science*, pages 209–224, Toronto, Canada, October 2008. Springer.
2. W.R. Bevier. Kit: A study in operating system verification. *IEEE Transactions on Software Engineering*, 15(11):1382–1396, 1989.
3. Sven Beyer, Christian Jacobi, Daniel Kroening, Dirk Leinenbach, and Wolfgang Paul. Putting it all together: Formal verification of the VAMP. *International Journal on Software Tools for Technology Transfer*, 8(4–5):411–430, August 2006.
4. Science Applications International Corporation. Green Hills Software INTEGRITY-178B Separation Kernel security target version 1.0, May 2008. Available from `http://www.niap-ccevs.org/cc-scheme/st/st_vid10119-st.pdf`. Retrieved February 11, 2009.

5. Matthias Daum, Jan Dörrenbächer, Burkhart Wolff, and Mareike Schmidt. A verification approach for system-level concurrent programs. In Jim Woodcock and Natarajan Shankar, editors, *Verified Software: Theories, Tools, Experiments. Second International Conference, VSTTE 2008, Toronto, Canada, October 6–9, 2008. Proceedings*, volume 5295 of *Lecture Notes in Computer Science*, pages 161–176, Toronto, Canada, October 2008. Springer.
6. Matthias Daum, Jan Dörrenbächer, and Sebastian Bogan. Model stack for the pervasive verification of a microkernel-based operating system. In Bernhard Beckert and Gerwin Klein, editors, *5th International Verification Workshop (VERIFY'08)*, volume 372 of *CEUR Workshop Proceedings*, pages 56–70. CEUR-WS.org, 2008.
7. M. Hillebrand, T. In der Rieden, and W.J. Paul. Dealing with I/O Devices in the Context of Pervasive System Verification. In *23nd IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD 2005), 2-5 October 2005, San Jose, CA, USA, Proceedings*, pages 309–316. IEEE, 2005.
8. Mark A. Hillebrand and Wolfgang J. Paul. On the architecture of system verification environments. In Karen Yorav, editor, *Hardware and Software: Verification and Testing, Third International Haifa Verification Conference, HVC 2007, Haifa, Israel, October 23–25, 2007, Proceedings*, volume 4899 of *Lecture Notes in Computer Science*, pages 153–168. Springer, 2008.
9. Michael Hohmuth and Hendrik Tews. The semantics of C++ data types: Towards verifying low-level system components. In David Basin and Burkhart Wolff, editors, *16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2003). Emerging Trends Proceedings*, pages 127–144. Institut für Informatik, Universität Freiburg, 2003. Technical Report No. 187.
10. Michael Hohmuth and Hendrik Tews. The VFiasco approach for a verified operating system. In *2nd ECOOP Workshop on Programming Languages and Operating Systems (ECOOP-PLOS)*, Glasgow, UK, July 2005.
11. M. Huisman and B. Jacobs. Java program verification via a Hoare logic with abrupt termination. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering*, volume 1783 of *Lecture Notes in Computer Science*, pages 284–303. Springer, Berlin, 2000.
12. H. Härtig, M. Hohmuth, N. Feske, C. Helmuth, A. Lackorzynski, F. Mehnert, and M. Peter. The Nizza Secure-System Architecture. In *First International Conference on Collaborative Computing: Networking, Applications and Worksharing*, San Jose, California, USA, Dec. 2005.
13. IBM Systems virtualization version 2 release 1, December 2005. Available from `http://publib.boulder.ibm.com/infocenter/eserver/v1r2/topic/eicay/eicay.pdf`. Retrieved December 18, 2008.
14. Intel Corporation, Denver, Colorado. *Intel 64 and IA-32 Architectures Software Developer's Manual*, May 2007. Order Number: 25366[5-9]-023US.
15. Intel Corporation. *TLBs, Paging-Structure Caches, and Their Invalidation*, April 2008. Application Note 317080-002.
16. ISO/IEC JTC1/SC22/WG21 C++ Standards Committee. *Programming Languages—C++*, 1998. ISO/IEC 14882:1998.
17. Gerwin Klein. Operating system verification — an overview. Technical Report NRL-955, NICTA, Sydney, Australia, June 2008.
18. Rafal Kolanski. A logic for virtual memory. *Electr. Notes Theor. Comput. Sci.*, 217:61–77, 2008.
19. Rafal Kolanski and Gerwin Klein. Mapped separation logic. In Jim Woodcock and Natarajan Shankar, editors, *Proceedings of VSTTE 2008 — Verified Software: Theories, Tools and Experiments*, volume 5295 of *Lecture Notes in Computer Science*, pages 15–29, Toronto, Canada, Oct 2008. Springer.
20. Michael Norrish. C formalised in HOL. Technical Report UCAM-CL-TR-453, Computer Laboratory, University of Cambridge, December 1998.
21. S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking, and model checking. In R. Alur and T.A. Henzinger, editors, *Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 411–414. Springer, Berlin, 1996.
22. Robin. Open robust infrastructures. project webpage, Feb 2006. http://robin.tudos.org.
23. Norbert Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL*. PhD thesis, Technische Universität München, 2006.

24. H. Tews. Micro hypervisor verification: Possible approaches and relevant properties. In *NLUUG Voorjaarsconferentie 2007: Virtualisatie*, pages 96–109, May 2007.
25. H. Tews, T. Weber, and M. Völp. A Formal Model of Memory Peculiarities for the Verification of Low-Level Operating-System Code. In R. Huuck, G. Klein, and B. Schlich, editors, *Proceedings of the 3rd international workshop on System Software Verification (SSV08)*, volume 217 of *Electronic Notes in Theoretical Computer Science*, pages 79–96, Sydney, 2008.
26. Hendrik Tews, Tjark Weber, Erik Poll, Marko van Eekelen, and Peter van Rossum. Formal Nova interface specification. Technical Report ICIS–R08011, Radboud University Nijmegen, May 2008.
27. Hendrik Tews, Tjark Weber, Marcus Völp, Erik Poll, Marko van Eekelen, and Peter van Rossum. Nova micro–hypervisor verification. Technical Report ICIS–R08012, Radboud University Nijmegen, May 2008.
28. Harvey Tuch. *Formal memory models for verifying C systems code*. PhD thesis, University of NSW, Sydney 2052, Australia, 2008.
29. Harvey Tuch. Structured types and separation logic. *Electr. Notes Theor. Comput. Sci.*, 217:41–59, 2008.
30. Harvey Tuch and Gerwin Klein. A unified memory model for pointers. In Geoff Sutcliffe and Andrei Voronkov, editors, *12th International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR-12)*, volume 3835 of *Lecture Notes in Computer Science*, pages 474–488, Jamaica, December 2005.
31. Harvey Tuch, Gerwin Klein, and Michael Norrish. Types, bytes, and separation logic. In Martin Hofmann and Matthias Felleisen, editors, *Proc. 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'07)*, pages 97–108, Nice, France, January 2007.
32. VFiasco. Verified Fiasco. project webpage, Jan 2001. http://os.inf.tu-dresden.de/vfiasco.
33. Marcus Völp, Stephan Courcambeck, and Christian Schwarz. Final activity report. Robin project deliverable D.8, Technische Universität Dresden, Germany, April 2008.
34. Bruce J. Walker, Richard A. Kemmerer, and Gerald J. Popek. Specification and verification of the UCLA Unix security kernel. *Commun. ACM*, 23(2):118–131, 1980.