Types, Maps and Separation Logic

Rafal Kolanski and Gerwin Klein

Sydney Research Lab., NICTA*, Australia
School of Computer Science and Engineering, UNSW, Sydney, Australia

{rafal.kolanski|gerwin.klein}@nicta.com.au

Abstract. This paper presents a separation-logic framework for reasoning about low-level C code in the presence of virtual memory. We describe our abstract, generic Isabelle/HOL framework for reasoning about virtual memory in separation logic, and we instantiate this framework to a precise, formal model of ARMv6 page tables. The logic supports the usual separation logic rules, including the frame rule, and extends separation logic with additional basic predicates for mapping virtual to physical addresses. We build on earlier work to parse potentially type-unsafe, system-level C code directly into Isabelle/HOL and further instantiate the separation logic framework to C.

1 Introduction

Virtual memory is a mechanism in modern computing systems that usual programming language semantics gloss over. For the application level, the operating system (OS) is expected to provide an abstraction of plain memory and details like page faults are handled behind the scenes. While, strictly speaking, the presence of virtual memory is still observable via sharing, ignoring virtual memory is therefore defendable for the application level.

For verifying lower-level software such as the operating system itself or software for embedded devices without a complex OS layer, this is no longer true. On this layer, virtual memory plays a prominent and directly observable role. It is also the source of many defects that are frequently very frustrating to debug. A wrong, unexpected mapping from virtual to physical addresses in the machine can lead to garbled, unrecognisable data at a much later, seemingly unrelated position in the code. A wrong, non-existing mapping will lead to a page fault: if the machine attempts to read a code instruction or a data value from a virtual address without valid mapping, on most architectures, a hardware exception is raised and execution branches to the address of a registered page fault handler (which often is virtually addressed itself). Defects in the page fault handler may lead to even more obscure, non-local symptoms. The situation is complicated by the fact that these virtual-to-physical mappings are themselves encoded in

* NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program

memory, usually in a hardware-defined page table structure, and they are often manipulated through the virtual memory layer.

As an example, the completion of the very first C implementation (at the time untried and unverified) of the formally verified seL4 microkernel [8] in our group was celebrated by loading the code onto our ARMv6[1] development board and starting the boot process to generate a hello-world message. Quite expectedly, nothing at all happened. The board was unresponsive and no debug information was forthcoming. It took 3 weeks to write the C implementation following a precise specification. It took 5 weeks debugging to get it running. It turned out that the boot code had not set up the initial page table correctly, and since no page fault handler was installed, the machine just kept faulting. This was the first of a number of virtual-memory related bugs. What is worse, our verification framework for C would, at the time, not have caught any of these bugs. We have since explictly added the appropriate virtual memory proof obligations. They are derived, in part, from the work presented in this paper.

We present a framework in Isabelle/HOL for the verification of low-level C code with separation logic in the presence of virtual memory. The framework itself is abstract and generic. In earlier work [16], we described a preliminary version of it, instantiated to a hypothetical simple page table and a toy language. In that work we concentrated on showing that the logic of the framework is indeed an instance of abstract separation logic [5] and that it supports the usual separation logic reasoning, including the frame rule. Here, we concentrate on making the framework applicable to the verification of real C code. We have instantiated the framework to the high-fidelity memory model for C by Tuch et al [24] and connected it with the same C-parsing infrastructure for Isabelle/HOL that was used there. On the hardware side, we have instantiated the framework to a detailed and precise model of ARMv6 2-level hardware page tables. To our knowledge, this is the first formalisation of the ARMv6 memory translation mechanism. The resulting instantiation is a foundational, yet practical verification framework for a large subset of standard C99 [13] with the ability to reason about the effects of virtual memory when necessary and the ability to reason abstractly in the traditional separation logic style when virtual memory is not the focus.

The separation logic layer of the framework makes three additional basic predicates available: mapping from a virtual address to a value, mapping from a physical address to a value, and mapping from a virtual to a physical address. For the user of the framework, these integrate seamlessly with other separation logic formulae and they support all expected, traditional reasoning principles. Inside the framework, we invest significant effort to provide this nice abstraction, to support the frame rule, and to shield the verification user from the considerable complexity of the hardware page table layout in a modern architecture.

Our envisaged application area for this framework is low-level OS kernel code that manipulates page tables and user-level page fault handlers in microkernel systems. To stay in the same, foundational framework, it can also be used

[1] The ARMv6 is a popular processor architecture for embedded systems, such as the iPhone or Android.

for the remaining OS kernel without any significant reasoning overhead in a separation logic setting. Our direct application area is the verification of the seL4 microkernel [8].

The remainder of this paper is structured as follows. After introducing notation in Sect. 2, we describe in Sect. 3 an abstract type class for encoding arbitrary C types in memory. Sect. 4 describes our abstract, generic page table framework and Sect. 5 instantiates this to ARMv6. Sect. 6 integrates virtual memory into our abstract separation logic framework, first at the byte level, and then at the structured types level. Sect. 7 makes the connection to C, and, finally, Sect. 8 discusses how translation caching mechanisms can be integrated into the model.

2 Notation

This section introduces Isabelle/HOL syntax where different from standard mathematical notation.

The space of total functions is denoted by $\Rightarrow$. Type variables are written $'a$, $'b$, etc. The notation $t :: \tau$ means that HOL term t has HOL type τ.

Pairs come with the two projection functions $\mathsf{fst} :: 'a \times 'b \Rightarrow 'a$ and $\mathsf{snd} :: 'a \times 'b \Rightarrow 'b$. *Sets* (type $'a$ set) follow the usual mathematical convention. *Lists* support the empty list [] and cons, written $x{\cdot}xs$. The list of natural numbers from a to (excluding) b is $[a..<b]$. We also use the standard zip and map from functional programming. The *option* type

$$\textbf{datatype}\ 'a\ \mathsf{option} = \mathsf{None} \mid \mathsf{Some}\ 'a$$

adjoins a new element None to a type $'a$. We use $'a$ option to model partial functions, writing $\lfloor a \rfloor$ instead of Some a and $'a \rightharpoonup 'b$ instead of $'a \Rightarrow 'b$ option. The Some constructor has an underspecified inverse called the, satisfying the $\lfloor x \rfloor = x$. Lifting functions to the option type is achieved by

$$\mathsf{option\text{-}map} = (\lambda f\ y.\ \mathsf{case}\ y\ \mathsf{of}\ \mathsf{None} \Rightarrow \mathsf{None} \mid \lfloor x \rfloor \Rightarrow \lfloor f\ x \rfloor)$$

Function update is written $f(x := y)$ where $f :: 'a \Rightarrow 'b$, $x :: 'a$ and $y :: 'b$ and $f(x \mapsto y)$ stands for $f\ (x := \mathsf{Some}\ y)$. *Finite integers* are represented by the type $'a$ word where $'a$ determines the word length in bits. The type supports the usual bit operations like left-shift (<<) and bitwise *and* (&&). The function unat converts to natural numbers (u for unsigned). Separation logic uses the concepts of disjoined maps $\perp$ and map addition ++. They are defined below.

$$m_1 \perp m_2 \quad \equiv \mathsf{dom}\ m_1 \cap \mathsf{dom}\ m_2 = \emptyset$$
$$m_1 +\!+ m_2 \equiv \lambda x.\ \mathsf{case}\ m_2\ x\ \mathsf{of}\ \mathsf{None} \Rightarrow m_1\ x \mid \lfloor y \rfloor \Rightarrow \lfloor y \rfloor$$

3 Types and Value Storage

Our aim of reasoning about C programs requires a representation of the storage of C values in memory. Similarly to Tuch et al [24], we define a mem-type type class to represent these types. This section describes the abstract operations of

this class and its axioms. The first such operations are serialising and restoring a value into and from bytes:

to-bytes :: $'t$::mem-type ⇒ byte list from-bytes (to-bytes v) = v
from-bytes :: byte list ⇒ $'t$::mem-type

For a particular type, all values occupy the same, non-zero number of bytes in memory. We will refer to the number of these bytes as the size. The length of a type's serialisation is equal to its size. The term TYPE($'t$) of type $'t$ itself makes an Isabelle type avaiable as term.

size-of :: $'t$::mem-type itself ⇒ nat length (to-bytes v) = size-of TYPE($'t$)
$0 <$ size-of TYPE($'t$)

For treating types as first-class values, we require each to map to a unique tag:

type-tag :: $'t$::mem-type itself ⇒ type-tag

In order to respect the alignment requirements of C types, mem-type instances carry alignment information. Types may only be aligned to sizes which are divisors of both the physical and virtual address space sizes:

align-of :: $'t$::mem-type itself ⇒ nat
align-of TYPE($'a$) dvd memory-size ∧ align-of TYPE($'a$) dvd addr-space-size

The model we present in this paper allows representation of all packed C types, i.e. atomic types such as int, array, and structs without padding. Tuch's work on structured C types [23] demonstrates how to extend this model to allow padding.

4 Virtual Memory

This section defines addressing and pointer conventions and describes our abstract interface to page table encodings.

4.1 Pointers and Addressing

Virtual memory is an abstraction layer on top of the physical memory in a machine. Each executing process gets its own view of physical memory, wherein each virtual address *may* be mapped to a physical address. We will henceforth refer to the function translating virtual addresses to physical ones as the *virtual map* and the application of the virtual map to a virtual address as a *lookup*.

The virtual map is partial and many-to-one — updates at one virtual address may affect values appearing at another. As in our previous work [16] memory is a partial function. Unlike our previous work [16], the work presented here is a realistic representation of physical memory and maps physical addresses to bytes. The virtual map is encoded in memory in a structure called a *page table*. Programs usually only have access to the virtual address layer, but devices may access physical memory directly. We define addresses as:

datatype ($'a$, $'p$) addr-t = Addr of $'a$

where $'a$ is the underlying address size (e.g. 32 word for 32-bit) and $'p$ is a tag: one of physical or virtual. For particular architectures, we instantiate addr-t into

$$\frac{\mathsf{ptable\text{-}lift}\ h_0\ r\ vp = \lfloor p \rfloor \qquad h_0 \perp h_1}{\mathsf{ptable\text{-}lift}\ (h_0 +\!\!+ h_1)\ r\ vp = \lfloor p \rfloor} \qquad \frac{\mathsf{ptable\text{-}lift}\ h\ r\ vp = \lfloor p \rfloor \qquad h \perp h'}{\mathsf{ptable\text{-}trace}\ (h +\!\!+ h')\ r\ vp = \mathsf{ptable\text{-}trace}\ h\ r\ vp}$$

$$\frac{p \notin \mathsf{ptable\text{-}trace}\ h\ r\ vp \qquad \mathsf{ptable\text{-}lift}\ h\ r\ vp = \lfloor p \rfloor}{\mathsf{ptable\text{-}trace}\ (h(p \mapsto v))\ r\ vp = \mathsf{ptable\text{-}trace}\ h\ r\ vp}$$

$$\frac{p \notin \mathsf{ptable\text{-}trace}\ h\ r\ vp \qquad \mathsf{ptable\text{-}lift}\ h\ r\ vp = \lfloor p \rfloor}{\mathsf{ptable\text{-}lift}\ (h(p \mapsto v))\ r\ vp = \lfloor p \rfloor}$$

$$\frac{\mathsf{ptable\text{-}lift}\ (h_0 +\!\!+ h_1)\ r\ vp = \lfloor p \rfloor \qquad h_0 \perp h_1}{\mathsf{ptable\text{-}lift}\ h_0\ r\ vp = \lfloor p \rfloor \vee \mathsf{ptable\text{-}lift}\ h_0\ r\ vp = \mathsf{None}}$$

Fig. 1. Abstract page table interface.

specific virtual and physical addresses. For the ARMv6 both virtual and physical addresses are 32-bit words, yielding the instantiations:

$\mathsf{vaddr} = (32\ \mathsf{word}, \mathsf{virtual})\ \mathsf{addr\text{-}t} \qquad \mathsf{paddr} = (32\ \mathsf{word}, \mathsf{physical})\ \mathsf{addr\text{-}t}$

ARMv6 is capable of natively addressing 8, 16 and 32 bit values in memory (corresponding to *char*, *short* and *int* in C). We have shown that these are instances of mem-type. We use addr-val (Addr a) $= a$ to extract the address.

4.2 Page Table

We now introduce our abstract interface to page table encodings. There are many such possible encodings: one-level tables, fixed multi-level tables, variable-depth guarded page tables or even just hash tables. Usually, mappings are encoded in blocks of addresses (pages, superpages, etc.), which are hardware-defined. The page table also encodes extra information such as permissions and hardware-defined flags. We generalise our previous abstract page table interface [16] slightly to accomodate multiple page sizes and briefly summarise the other definitions.

ptable-lift :: $('paddr \rightharpoonup 'val) \Rightarrow 'base \Rightarrow 'vaddr \rightharpoonup 'paddr$
ptable-trace :: $('paddr \rightharpoonup 'val) \Rightarrow 'base \Rightarrow 'vaddr \Rightarrow 'paddr\ \mathsf{set}$
get-page :: $('paddr \rightharpoonup 'val) \Rightarrow 'base \Rightarrow 'vaddr \Rightarrow 'a$

We use ptable-lift to extract a virtual map from memory, ptable-trace to find all the physical addresses used looking up a virtual to a physical address, and get-page to find which page a virtual address is on including any machine-specific flags (such as permissions) that might be attached to it. The types *'paddr* and *'vaddr* represent physical and virtual pointers, while *'base* says where we can find the page table in physical memory (e.g. the root of a two-level page table). We leave *'a* for a generic representation of what a page is.

In order to reason about memory access in the presence of a page table, we require page table functions to conform to the rules in Fig. 1. Firstly, changing memory in areas not related to a page table lookup must not affect the lookup: if evaluation of ptable-lift and ptable-trace succeeds on smaller heap , it will also succeed on a larger one. This corresponds to the safety monotonicity property of separation logic [5]. Furthermore, a successful lookup must be unaffected by any heap updates outside that lookup's trace. Finally, corresponding to the frame

monotonicity property [5] of separation logic, removal of information from the heap must either not affect ptable-lift or cause it to fail. Heap reduction must not return a different successful result.

5 A Formal Model of ARMv6 Page Tables

In this section, we instantiate the abstract interface described above to ARMv6 2-level page tables. We support multiple page sizes, but we omit handling of permissions — in our seL4 target setup, the ARM supervisor mode ignores permissions. Adding them would be simple.

Following ARM nomenclature [3], the first level table is called the *page directory* and the second level the *page table*. Individual *entries* at these levels are called PDEs and PTEs respectively, 32 bits wide in both cases. There is one page directory with potentially many page tables. The base of the entire structure is the physical address of the page directory. Our model uses the common ARMv6 page table format where subpages are disabled. In this mode, the hardware supports mappings in four granularities: small (4Kb) and large (64Kb) pages, as well as sections (1Mb) and supersections (16Mb):

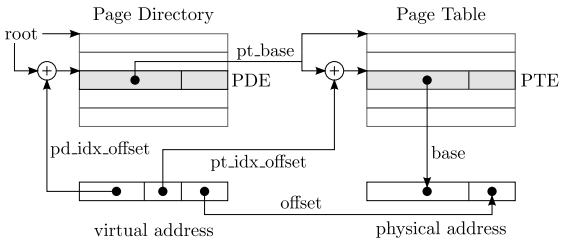

Fig. 2. ARMv6 page table lookup for SmallPage.

datatype page-size = SmallPage | LargePage | Section | SuperSection

Apart from invalid/reserved, a PDE either encodes the physical base address of a section, supersection or a second-level table. Within a second-level table, a valid PTE encodes the physical base address of a large or small page:

datatype pde = InvalidPDE | ReservedPDE | PageTablePDE of paddr
| SectionPDE of paddr | SuperSectionPDE of paddr

datatype pte = InvalidPTE | LargePagePTE of paddr | SmallPagePTE of paddr

The idea of looking up a virtual address is shown in Fig. 2: figure out the base address of the appropriate structure and its size, then add the virtual address divided by that size. The get-frame function calculates the base and size:

get-frame :: heap ⇒ paddr ⇒ vaddr ⇀ (paddr × page-size)
get-frame *h root vp* ≡
let *vp-val* = addr-val *vp*; *pd-idx-offset* = *vp-val* >> 20 << 2
in case decode-pde *h* (*root* + *pd-idx-offset*) of None ⇒ None
| ⌊PageTablePDE *pt-base*⌋ ⇒ get-frame-2nd *h pt-base vp*
| ⌊SectionPDE *base*⌋ ⇒ ⌊(*base*, Section)⌋
| ⌊SuperSectionPDE *base*⌋ ⇒ ⌊(*base*, SuperSection)⌋ | ⌊-⌋ ⇒ None

The function works by looking up a virtual address just like the ARM hardware. First, we look at the top 12 bits of the address as an index into the page directory.

We then shift the index by 2 as each PDE is 4 bytes in size, add it to the base address of the page directory (*root*). We decode the PDE at this address to decide what to do next: fail on invalid/reserved, pass through the base address for sections/supersections, and go look in the second-level table in the case of a PTE pointer. We omit the definitions of decode-pde and decode-pte; they work as described in the ARMv6 manual [3]. Second-level lookup is defined similarly:

```
get-frame-2nd :: heap ⇒ paddr ⇒ vaddr ⇀ (paddr × page-size)
get-frame-2nd h pt-base vp ≡
let vp-val = addr-val vp; pt-idx-offset = (vp-val >> 12) && 0xFF << 2
in case decode-pte h (pt-base + pt-idx-offset) of None ⇒ None
   | ⌊InvalidPTE⌋ ⇒ None | ⌊LargePagePTE base⌋ ⇒ ⌊(base, LargePage)⌋
   | ⌊SmallPagePTE base⌋ ⇒ ⌊(base, SmallPage)⌋
```

Starting at the physical address of the second-level table, we use the next 8 bits of the virtual address (bits 12-19) as an index, decode the PTE there, fail on invalid or return the base address of the frame along with its size.

Using get-frame, we can then implement the main lookup function ptable-lift by masking out the appropriate bits from the virtual address and adding them to the physical address of the frame:

```
vaddr-offset p w ≡ w && mask (page-size-bits p)

ptable-lift h pt-root vp ≡
let vp-val = addr-val vp
in option-map (λ(base, pg-size). base + vaddr-offset pg-size vp-val)
   (get-frame h pt-root vp)
```

where page-size-bits is log_2 of the page size.

Similarly, we can get the page a virtual address is on by masking out the offset bits. Also, since ARM allows multiple page sizes, the concept of a page must involve its size, instantiating the page type $'a$ to (vaddr × page-size) option:

```
addr-base sz w ≡ w && (0xFFFFFFFF << page-size-bits sz)

get-page h root vp ≡
let vp-val = addr-val vp
in option-map (λ(base, pg-size). (Addr (addr-base pg-size vp-val), pg-size))
   (get-frame h root vp)
```

We define a sequence of n addresses starting at p as:

```
addr-seq p 0 = []
addr-seq p (Suc n) = p·addr-seq (p + 1) n
```

The final function needed to instantiate the abstract page table model from Sect. 4 is ptable-trace. The trace contains the bytes in any page directory or table entry which has successfully contributed to looking up the virtual address:

```
ptable-trace h root vp ≡
let vp-val = addr-val vp; pd-idx-offset = vaddr-pd-index vp-val << 2;
    pt-idx-offset = vaddr-pt-index vp-val << 2;
    pd-touched = set (addr-seq (root + pd-idx-offset) 4);
    pt-touched = λpt-base. set (addr-seq (pt-base + pt-idx-offset) 4)
in case decode-pde h (root + pd-idx-offset) of None ⇒ ∅
   | ⌊PageTablePDE pt-base⌋ ⇒ pd-touched ∪ pt-touched pt-base
   | ⌊-⌋ ⇒ pd-touched
```

We have proved that the ptable-lift, ptable-trace and get-page functions in this section instantiate the abstract model from Sect. 4, including the axioms of Fig. 1.

6 Typed, Mapped Separation Logic

Based on our abstact page table interface of Sect. 4, we can now construct a separation logic framework for reasoning about pointer programs with types. This framework is independent of the particular page table instantiation.

Separation logic [18] is a tool for conventiently reasoning about memory and aliasing. It views memory as a partial *heap* from addresses to values, allowing for predicates which precisely state which part of the heap they hold on. At its core is the concept of separating conjunction: when the assertion $P \wedge^* Q$ holds on a heap, the heap can be split into two disjoint parts, where P holds on one part and Q on the other. Predicates which precisely define the domain of the heap they hold on allow for convenient local reasoning. This leads to the concept of local actions and the frame rule: for an action f, we can conclude $\{P \wedge^* R\}\, f\, \{Q \wedge^* R\}$ from $\{P\}\, f\, \{Q\}$ for *any* R. This expresses that the actions of f are local to the heaps described by P and Q, and therefore cannot affect any separate heap described by R. We also say that predicates *consume* parts of the heap under separating conjunction, because other predicates cannot depend on the same parts of this heap.

The basic assertion of separation logic is the maps-to arrow, holding on a heap containing only one address-value pair. From this simple assertion, more complex ones can be built. For a simple heap ($\mathsf{paddr} \rightharpoonup \mathsf{byte}$) it takes the form:

$$(\mathit{address} \mapsto \mathit{value})\ h \equiv h\ \mathit{address} = \mathit{value} \wedge \mathsf{dom}\ h = \{\mathit{address}\}$$

Under separating conjunction, it consumes *address* in the heap. Tuch et al extend this basic concept all the way to reasoning about C code with structures [23].

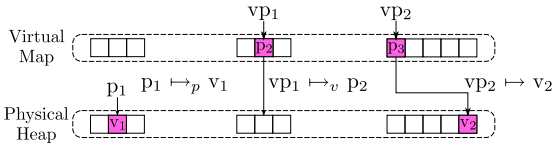

Fig. 3. The three maps-to assertions.

A naive addition of virtual memory to separation logic breaks the concept of separating conjunction, the frame rule, as well as the assumption of Tuch's work of values being stored contiguously in the heap. In previous work [16], we addressed the first two in a simplified setting. In this section, we solve them in a realistic setting and extend them to reasoning about typed pointers. We introduce new maps-to arrows, as well as a new, more complex state that we use instead of a simple heap.

Our eventual goal is to be able to write the new arrows of Fig. 3 with physical or virtual addresses on the left and complex, typed C values on the right. The new arrows in Fig. 3 describe (from left to right): mappings from physical address to value, from virtual to physical address, and from virtual address to value. The next section will introduce arrows that allow raw, single bytes and explicit type information on the right. The section after that will lift this information to allow structured C types on the right.

6.1 At the bytes level

Following Tuch et al and our own previous work, to support both types and virtual memory, we annotate the heap with extra information, extending the state for our assertions in a first step to:

$$(\mathsf{paddr} \rightharpoonup \mathsf{type\text{-}data} \times \mathsf{byte}) \times \mathsf{ptable\text{-}base}$$

where ptable-base is any extra information needed by the virtual memory subsystem, such as the page table root (paddr in the case of ARMv6); type-data annotates which higher-level type a byte is part of. On this level it is just passed through, we will explain its purpose in Sect. 6.2.

For our maps-to assertions to be useful in separation logic, we must define which parts of the heap they consume (what their domain is). Here we run into a problem, illustrated in Fig. 4: two distinct virtual addresses map to two values via distinct physical addresses, but using the same page table entry for the lookup. Writing to one virtual pointer does not affect the value at the other, so in this sense the two maps-to predicates are separate. However, a single page table entry is involved in the lookup of both virtual pointers. Under separating conjunction we can allow the entry to be consumed by either mapping or neither mapping, but not both mappings. If one consumes it, the other lacks information for a successful lookup. If neither consumes it, we lose locality: we could state the entry is separate from both mappings even though updating the entry can affect both virtual addresses!

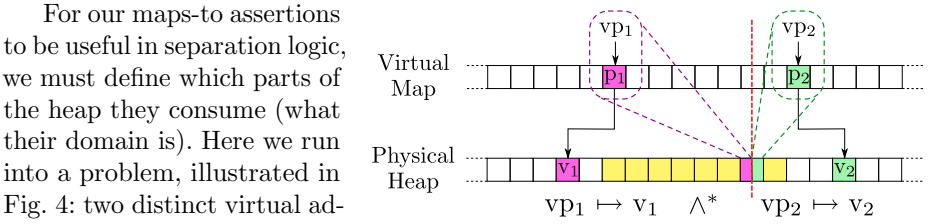

Fig. 4. Two virtual addresses resolving through the same page table entry.

The solution to this problem is to divide the page table entry up into two parts and share the slices between the maps-to predicates involved in the separating conjunction. This idea is similar to that of the fractional permission model of Bornat [4], with three important differences. Firstly, we do not wish to perform any explicit accounting of fractions in the most common case of the page table not being modified. Secondly, the number of virtual addresses an entry can map varies with the type of page table and the size of the mapped page. Thirdly, we want to utilise rather than recreate the proofs about partial maps and map disjunction in Isabelle/HOL. These issues are addressed by using a constant, large-enough number of slices for entries in the heap and placing them in the domain. The maximum useful number of slices is one entry mapping all virtual addresses. Thus our final state for assertions is:

$$\mathsf{fheap\text{-}state} = (\mathsf{paddr} \times \mathsf{vaddr} \rightharpoonup \mathsf{type\text{-}data} \times \mathsf{byte}) \times \mathsf{ptable\text{-}base}$$

We refer to the first component of this state as the *typed, fragmented heap tfh*. With this new state, our physical memory maps-to predicate becomes:

$$p :\!\!\rightarrow_p v \equiv \lambda(h, r).\ (\forall\, vp.\ h\ (p, vp) = \lfloor v \rfloor) \wedge \mathsf{dom}\ h = \{p\} \times \mathcal{U}$$

Like the simple maps-to predicate shown earlier, the heap at address p evaluates to value v. In the new state, it does so for all vp slices. The domain covers all slices of p, i.e. the universal set $\mathcal{U}$. This arrow works for the physical-to-value level. To define the virtual-to-physical arrow, we use our abstract page table interface. Unfortunately, this page table model knows nothing about slices and type annotations. So, to perform a lookup on vp, we derive a view of the heap tfh containing only slices associated with vp and discard type annotations:

$$\mathsf{h\text{-}view}\ tfh\ vp \equiv \mathsf{option\text{-}map\ snd} \circ (\lambda p.\ tfh\ (p,\ vp))$$

We can now define the virtual-to-physical arrow for mapping vp to p. It is just a ptable-lift on a heap made of slices associated with vp. The assertion consumes the vp slice of each byte used in its lookup, i.e. in ptable-trace:

$$\begin{aligned} vp :\rightarrow_v p \equiv \lambda(h,r).\ &\mathsf{let\ heap} = \mathsf{h\text{-}view}\ h\ vp;\ vmap = \mathsf{ptable\text{-}lift\ heap}\ r \\ &\mathsf{in}\ vmap\ vp = \lfloor p \rfloor \wedge \mathsf{dom}\ h = \mathsf{ptable\text{-}trace\ heap}\ r\ vp \times \{vp\} \end{aligned}$$

The virtual-to-value mapping is then just the separating conjunction of virtual-to-physical and physical-to-value.

$$vp :\rightarrow v \equiv \lambda s.\ \exists p.\ (vp :\rightarrow_v p \wedge^* p :\rightarrow_p v)\ s$$
$$P \wedge^* Q \equiv \lambda(h,\ r).\ \exists h_0\ h_1.\ h_0 \perp h_1 \wedge h = h_0 +\!\!+ h_1 \wedge P\ (h_0,\ r) \wedge Q\ (h_1,\ r)$$

For any of these levels, we can define the usual arrow variations [18]:

$$(p :\rightarrow -)\ s \equiv \exists v.\ (p :\rightarrow v)\ s \qquad (p :\hookrightarrow v)\ s \equiv (p :\rightarrow v \wedge^* \mathsf{sep\text{-}true})\ s$$
$$(p :\hookrightarrow -)\ s \equiv \exists v.\ (p :\rightarrow v \wedge^* \mathsf{sep\text{-}true})\ s \qquad \mathsf{sep\text{-}true} \equiv \lambda s.\ \mathsf{True}$$

One property of this framework is that it is mostly independent of the value space, the right-hand side of the maps-to arrows. Only in the interface to the page table have we touched it at all, and then only to discard additional type information. The basic assertions we get from this section are of the form $vp :\rightarrow (b,\ t)$ where b is the byte at virtual address vp, and t is the associated type annotation.

6.2 At the types level

This section uses the arrows for bytes and type information we have just defined to higher-level, typed assertions for any mem-type values. We define the concept of pointers to typed values by wrapping our existing concept of addresses and adding a phantom type, like Tuch et al [24]:

$$\textbf{datatype}\ ('a,\ 'p,\ 't)\ \mathsf{ptr\text{-}t} = \mathsf{Ptr\ of}\ ('a,\ 'p)\ \mathsf{addr\text{-}t}$$

Instantiated to the ARMv6:

$$'t\ \mathsf{pptr} = (32\ \mathsf{word},\ \mathsf{physical},\ 't)\ \mathsf{ptr\text{-}t} \qquad 't\ \mathsf{vptr} = (32\ \mathsf{word},\ \mathsf{virtual},\ 't)\ \mathsf{ptr\text{-}t}$$

Like Tuch et al we mark locations belonging to mem-type values in the heap with a type tag. The addition of virtual memory creates a new complication: if a value crosses a page boundary in virtual memory, it is not guaranteed to be contiguous at the physical level, nor even entirely loaded into memory. This means we must not only tag each byte in the heap, but also note which offset it is within the larger structure it belongs to. Our type information associated with each byte is:

$$\mathsf{type\text{-}data} = \mathsf{type\text{-}tag} \times \mathsf{nat}$$

We implement maps-to predicates at the typed level as a sequence of byte-level maps-to predicates, folded over separating conjunction in the usual way. For instance, we write *vps* $[:\rightarrow]$ *vs* for a sequence *vps* of virtual pointers mapping to a sequence of values *vs*. Note that these values are each of the from (b,t).

A value of type $'t$::mem-type seen in memory at either the virtual or physical level is a sequence of bytes (to-bytes) where each byte is tagged by the type-tag of $'t$ and its offset in the list:

value-seq *val* ≡
zip (map ($\lambda seq.$ (type-tag TYPE($'t$), *seq*)) [0..<size-of TYPE($'t$)]) (to-bytes *val*)

We can now define maps-to predicates on typed pointers. Like Tuch et al [24] we employ an arbitrary guard on the pointer itself to enforce constraints such as alignment. We have not found it necessary yet to let the guard depend on the state, but this could be added easily. Compared to Tuch et al, lifting sequences of bytes to structured values is much simpler, because we already have byte-level assertions available. Between virtual and physical levels only the arrows differ.

$g \vdash p \rightarrow_p v$ ≡ ptr-seq p TYPE($'t$) $[:\rightarrow_p]$ value-seq v $\lfloor\wedge\rfloor$ ($\lambda s.\ g\ p$)
$g \vdash vp \rightarrow_v p$ ≡ ptr-seq vp TYPE($'t$) $[:\rightarrow_v]$ ptr-seq p TYPE($'t$) $\lfloor\wedge\rfloor$ ($\lambda s.\ g\ vp$)
$g \vdash vp \rightarrow v$ ≡ ptr-seq vp TYPE($'t$) $[:\rightarrow]$ value-seq v $\lfloor\wedge\rfloor$ ($\lambda s.\ g\ vp$)

where ptr-seq p T ≡ addr-seq (ptr-val p) (size-of T) and addr-seq is defined in Sect. 4. Using these predicates, we can now make separation logic assertions describing the presence of typed values on the heap, visible as contiguous in either physical or virtual memory. In the common case, i.e. when not modifying the page table, our model keeps the virtual memory mechanism under the hood. We can just state, for instance, $p \rightarrow (\!|\ x = 10;\ y = 7\ |\!)$ where the right hand side is an Isabelle record of class mem-type corresponding to a C struct and the left hand side is a virtual address.

7 Connecting with C

In this section, we will connect the framework to C and define loading and storing of typed values in the program state. In the previous section, we have enriched the usual C heap with additional information: slices for specifying the domain of predicates under separating conjunction and type annotation information. We therefore need to be careful to not introduce unwanted dependencies on the additional information in the state and we need to make sure that C updates operate consistently on the extended state. We formalise load and store for virtually addressed access. Direct physical access would be similar, but simpler. In C, loading and storing are total functions. Loading from a wrongly typed or unmapped address or storing to it will produce garbage. For our intended application (seL4), we do not need to model page faults directly, but we annotate the C program with guards that make sure no page faults will occur. These annotations are added automatically during the translation into Isabelle/HOL and will produce proof obligations. Should a page-fault model be required for

different applications, it is easy to add: an access to an unmapped page, instead of a guard, simply produces a branch to the page fault handler.

For a generic map h from pointers p to values, loading a mem-type value at p is merely loading its size's worth of sequential bytes starting at p (load-list-basic), making sure h contains no gaps in that range (deoption-list) and passing it to from-bytes from the type class interface.

load-list-basic h 0 p = []
load-list-basic h (Suc n) p = h p·load-list-basic h n (p + 1)

deoption-list xs ≡ if None ∈ set xs then None else ⌊map the xs⌋
load-list h n p ≡ deoption-list (load-list-basic h n p)
load-value h p ≡ option-map from-bytes (load-list h (size-of TYPE('t)) p)

A pointer access in C is then just an application of load-value to the address-space view of memory, ignoring any read failures. We drop the additional type information that is only used in assertions, not in C, resulting in the heap type load-value expects. The as-view function is similar to h-view, but uses ptable-lift to arrive at a map from virtual addresses to values.

load-value-c s vp ≡ the (load-value (as-view s) (ptr-val vp))

As mentioned above, this function is total. The guard generated for each such access is c-guard ⊢ vp ↪ –, ensuring that the load-value-c will produce a valid result. The predicate c-guard p ensures that p is not Null and is correctly aligned for its type size.

Heap updates are similar. For a single physical address, we update all slices at that address and we leave the type annotation untouched. We can ignore entries with None, because, again, the generated guard c-guard ⊢ vp ↪ – will ensure this case does not occur. We then lift the single-byte update first to the virtual layer to provide address translation via vmap-view, and then like in Tuch et al to byte sequences to accomodate structured types.

tfheap-update tfh p v ≡
λppv. if fst ppv = p then option-map (λ(td, v'). (td, v)) (tfh ppv)
else tfh ppv

state-update-v s vp v ≡
case vmap-view s vp of None ⇒ s | ⌊p⌋ ⇒ (tfheap-update (fst s) p v, snd s)

state-update-v-list s [] = s
state-update-v-list s ((vp, v)·us) = state-update-v-list (state-update-v s vp v) us

c-state-update vp v s ≡ state-update-v-list s (zip (ptr-seq vp TYPE('a1)) (to-bytes v))

For interfacing to C code, we have adapted the C parser of Tuch et al [24]. It translates a significant subset of the C99 programming language into SIMPL [19], a generic, imperative language framework in Isabelle/HOL.

As in the framework by Tuch et al we cannot prove the frame rule generically, but we can prove it automatically for each individual program. This automatic proof ultimately reduces everything to valid memory accesses and updates, based on the following rule:

$$\frac{(\text{c-guard} \vdash vp \rightarrow - \wedge^* P)\ s}{(\text{c-guard} \vdash vp \rightarrow v \wedge^* P)\ (\text{c-state-update}\ vp\ v\ s)}$$

```
void mapUserFrame(pde_t *pd, paddr_t paddr, vptr_t vptr) {
    pde_t *pdSlot; pte_t *ptSlot, *pt, pte; unsigned int ptIndex;
    pdSlot = lookupPDSlot(pd, vptr);
    ptIndex = ((unsigned int)vptr >> ARMSmallPageBits) & MASK(PT_BITS);
    pt = ptrFromPAddr(pde_coarse_ptr_get_address(pdSlot));
    ptSlot = pt + ptIndex;
    pte = pte_small_new(paddr,1,0,0,0,3,1,1,1,0);
    *ptSlot = pte;
}
```

Fig. 5. Page table code extracted from seL4.

With $P =$ sep-true, this rule becomes the state update rule by Tuch et al, corresponding to the assignment axiom in standard separation logic. The corresponding rule for memory access holds as well, of course:

$$\frac{(g \vdash vp \rightarrow v)\ s}{\text{load-value-c}\ s\ vp = v}$$

Fig. 5 shows an excerpt of typical page table manipulation code that this framework can handle. The last line of this code, for instance, would be translated into the following SIMPL statement with guard:

Guard C-Guard {|c-guard ´*ptSlot*|}
(´*globals* :== heap-upd (c-state-update ´*ptSlot* ´*pte*))

The heap-upd function updates the C heap (our extended state) which is merely a global variable in the semantics of the C program. The guard statement Guard throws the guard error C-Guard if the condition {|c-guard ´*ptSlot*|} is false, and otherwise executes the statement. In previous work [16], we have conducted a detailed case study demonstrating how page table manipulations can be verified in this framework for a simple, one-level page table. Reasoning on the C and ARM level has precisely the same structure, it just involves more detail.

8 Translation Caching

Page table lookups are expensive; they potentially involve multiple memory reads. To decrease this cost, these lookups are cached in most architectures in a translation lookaside buffer (TLB). Abstractly, the TLB can be seen as a finite, small set of virtual-to-physical mappings. They may include lookups for code instructions as well as data. It is architecture-dependent whether these are handled separately from each other or not, how large the TLBs are, and when a mapping is removed from the TLB and replaced by another. Most architectures provide assembler instructions for explicitly removing all or specific mappings from the TLB, which is called *flushing*.

Although the page table should ultimately define what a mapping is, the hardware will always first consult the TLB and ignore the contents of the page table if a TLB entry is found. When we change the page table and the TLB contains the mapping being changed, we may introduce an inconsistency. This inconsistency can be resolved by flushing the TLB such that the new page

table contents will be loaded for future lookups. However, indiscriminate TLB flushes are expensive, because they will incur additional memory reads. Kernel programmers like to optimise by deferring TLB flushes as far as possible and by making them as specific as possibly.

In our model, we can add the TLB by reducing it to its safety-relevant content: whether the lookup for any specific virtual address may be inconsistent or not. What makes a TLB entry inconsistent is a change to the page table. We can turn this view around and instead keep track of inconsistent page table entries — those that have been written to since the last flush. We can reduce machinery by not caring whether a memory location currently is a page table entry or not, we just keep track of *all* locations that have been changed since the last TLB flush. If any memory read or write involves a page table entry whose location is in this set, the TLB might be inconsistent for this lookup. We can now generate guards that test for this case and require us to prove its absence.

This TLB model intergrates nicely with separating conjunction, because the set mentioned above can be implemented as an additional boolean next to the type information on the right-hand side of the maps-to arrow. Apart from the type, none of the generic framework definitions would need to change.

9 Related work

Our work touches three main areas: separation logic, virtual memory, and C verification. For an overview on OS verification in general, see Klein [14].

Separation logic was originally conceived by O'Hearn and Reynolds et al. [12,18] and has been formalised in mechanised theorem proving systems before [25,1]. We enhance these models with the ability to reason about properties on virtual memory, adding new basic predicates, but preserving the feel and reasoning principles of separation logic.

Virtual memory formalisations have appeared in the context of OS kernel verification before [15,7,11]. Reasoning about programs *running* under virtual memory, however, especially the operating systems which control it, remains mostly unexplored. Among the exceptions is the development of the Nova micro-hypervisor [20,21]. Like our work, the Nova developers aim to use a single semantics to describe all forms of memory access which simplifies significantly in the well-behaved case. They focus on reasoning about "plain memory" in which no virtual aliasing occurs and split it into read-only and read-write regions, to permit reading the page table while in plain memory. They do not use separation logic. Our work is more abstract. We do not explicitly define "plain memory". Rather the concept emerges from the requirements and state. Tews et al also include memory-mapped devices. The necessary alignment restrictions would intergrate seamlessly into our framework via the guard mechanism. Alkassar et al. [2] have proved the correctness of a kernel page fault handler, albeit not at the separation logic level. They use a single level page table and prove that the page fault handler establishes the illusion to the user of a plain memory abstraction, swapping in pages from disk as required. We instantiate our model

to an extensive, realistic model of ARMv6 2-level page tables. We are not aware of other realistic formalisations of ARM page tables; Fox [10] formalises the ARM instruction set, but omits memory translation, while Tews et al [21] formalise memory translation for IA32.

In the C verification space, we build directly on the work by Tuch et al [24,22,23] who employ separation logic to reasoning in a precise, foundational model for C memory with Isabelle/HOL infrastructure to reason about low-level, potentially type-unsafe C programs nicely and abstractly. This framework which in turn builds on Schirmer's SIMPL environment [19] is used in the verification of the seL4 microkernel [8]. We enhance the fidelity of the framework with a virtual memory layer for ARMv6 while inheriting its nice type-lifting and reasoning principles. Other work in C verification includes Key-C [17], VCC [6], and Caduceus [9]. Key-C treats only on a type-safe subset of C. VCC, which also supports concurrency, uses a memory model [6] that axiomatises a weaker version of what Tuch proves [23] and what we extend to virtual memory. Caduceus supports a large subset of C, but does not include virtual memory.

10 Conclusion and Future Work

We have presented an abstract framework for separation logic under virtual memory and have instantiated it to the C programming language as well as to ARMv6 page tables. We have shown in previous work that this framework supports one-level page tables as well as traditional separation logic reasoning, including the frame rule. We have shown here that the new instantiation supports the same basic rules for heap updates that Tuch et al provide for their C verification framework that is used in the verification of the seL4 microkernel.

Next to applying the framework to seL4 page table code in a verification case study, future work includes an Isabelle/HOL model for the translation caching mechanism that is an interesting and correctness-relevant part of most virtual memory architectures. We have sketched how the mechanism could be added to the presented model without fundamental changes. We are not aware of any other virtual memory frameworks that include TLB modelling.

The framework presented here makes the foundational verification of OS-level C code practical. It brings a significant source of errors into the realm for formal, machine-checked verification that otherwise formally verified code would ignore and fail on embarrassingly. Only when reasoning about page table modifications directly, the complexities of their encoding become visible. For reasoning on plain memory, no additional verification overhead must be paid.

Acknowledgements We thank Thomas Sewell for commenting on a draft of this paper and Michael Norrish for help with integrating the C parser.

References

1. R. Affeldt and N. Marti. Separation logic in Coq. `http://savannah.nongnu.org/projects/seplog`, 2008.

2. E. Alkassar, N. Schirmer, and A. Starostin. Formal pervasive verification of a paging mechanism. In C. Ramakrishnan and J. Rehof, editors, *Proc 14th TACAS'08*, volume 4963 of *LNCS*, pages 109–123. Springer, 2008.
3. ARM Limited. *ARM Architecture Reference Manual*, June 2000.
4. R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In *Proc. 32nd POPL*, pages 259–270. ACM, 2005.
5. C. Calcagno, P. W. O'Hearn, and H. Yang. Local action and abstract separation logic. In *Proc. 22nd LICS*, pages 366–378. IEEE Computer Society, 2007.
6. E. Cohen, M. Moskał, W. Schulte, and S. Tobies. A precise yet efficient memory model for C. `http://research.microsoft.com/apps/pubs/default.aspx?id=77174`, 2008.
7. I. Dalinger, M. A. Hillebrand, and W. J. Paul. On the verification of memory management mechanisms. In D. Borrione and W. J. Paul, editors, *CHARME*, volume 3725 of *LNCS*, pages 301–316. Springer, 2005.
8. K. Elphinstone, G. Klein, P. Derrin, T. Roscoe, and G. Heiser. Towards a practical, verified kernel. In *Proc. 11th HOTOS*, pages 117–122, 2007.
9. J.-C. Filliâtre and C. Marché. Multi-prover verification of C programs. In *Proc. 6th ICFEM*, volume 3308 of *LNCS*, pages 15–29. Springer, 2004.
10. A. Fox. Formal specification and verification of ARM6. In D. Basin and B. Wolff, editors, *TPHOLs '03*, volume 2758 of *LNCS*, pages 25–40. Springer, 2003.
11. M. Hillebrand. *Address Spaces and Virtual Memory: Specification, Implementation, and Correctness*. PhD thesis, Saarland University, Saarbrücken, 2005.
12. S. S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *Proc. 28th POPL*, pages 14–26. ACM, 2001.
13. *Programming languages—C*, 1999. ISO/IEC 9899:1999.
14. G. Klein. Operating system verification—An overview. *Sādhanā*, 34(1):27–69, 2009.
15. G. Klein and H. Tuch. Towards verified virtual memory in L4. In K. Slind, editor, *TPHOLs Emerging Trends '04*, Park City, Utah, USA, 2004.
16. R. Kolanski and G. Klein. Mapped separation logic. In J. Woodcock and N. Shankar, editors, *VSTTE*, volume 5295 of *LNCS*, pages 15–29. Springer, 2008.
17. O. Mürk, D. Larsson, and R. Hähnle. KeY-C: A tool for verification of C programs. In *Proc. 21st CADE*, volume 4603 of *LNCS*, pages 385–390. Springer, 2007.
18. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. 17th IEEE Symposium on Logic in Computer Science*, pages 55–74, 2002.
19. N. Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL*. PhD thesis, Technische Universität München, 2006.
20. H. Tews. Formal methods in the Robin project: Specification and verification of the Nova microhypervisor. In *C/C++ Verification Workshop, Technical Report ICIS-R07015*, pages 59–68, Oxford, UK, July 2007. Radboud University Nijmegen.
21. H. Tews, T. Weber, and M. Völp. Formal memory models for the verification of low-level operating-system code. *JAR*, 42(2–4):189–227, 2009.
22. H. Tuch. *Formal Memory Models for Verifying C Systems Code*. PhD thesis, School Comp. Sci. & Engin., University NSW, Sydney 2052, Australia, Aug. 2008.
23. H. Tuch. Formal verification of C systems code: Structured types, separation logic and theorem proving. *JAR*, 42(2–4):125–187, 2009.
24. H. Tuch, G. Klein, and M. Norrish. Types, bytes, and separation logic. In M. Hofmann and M. Felleisen, editors, *POPL '07*, pages 97–108. ACM, 2007.
25. T. Weber. Towards mechanized program verification with separation logic. In J. Marcinkowski and A. Tarlecki, editors, *Computer Science Logic – 18th Int'l Workshop, CSL 2004*, volume 3210 of *LNCS*, pages 250–264. Springer, 2004.