# Automated Verification of a Small Hypervisor

Eyad Alkassar[1,⋆], Mark A. Hillebrand[2,⋆], Wolfgang J. Paul[1], and Elena Petrova[1,⋆]

[1] Saarland University, Computer Science Dept., Saarbrücken, Germany
{eyad, wjp, petrova}@wjpserver.cs.uni-saarland.de
[2] German Research Center for Artificial Intelligence (DFKI), Saarbrücken, Germany
mah@dfki.de

**Abstract.** Hypervisors are system software programs that virtualize the architecture they run on. They are typically small, safety-critical, and hard to debug, which makes them a feasible and interesting target for formal verification. Previous functional verifications of system software were all based on interactive theorem proving, requiring substantial human effort complemented by expert prover knowledge. In this paper we present the first functional verification of a small hypervisor using VCC, an automatic verifier for (suitably annotated) C developed at Microsoft. To achieve this goal we introduce necessary system verification techniques, such as accurate modeling of software/hardware interaction and simulation proofs in a first-order logic setting.

## 1 Introduction

Hypervisors are small system software programs that virtualize the underlying architecture, allowing to run a number of guest machines (also called partitions) on a single physical host. Invented in the 1970s for use in mainframes, hypervisors are becoming more and more important today with shared multi-threading and shared multiprocessing being part of computer mainstream. Because they are hard to debug, and because of their small size yet high criticality, hypervisors make a viable and interesting target for (system) software verification. Hypervisor verification is also challenging: a hypervisor functions correctly if it simulates the execution of its guest systems. Thus, functional correctness of a hypervisor cannot be established by only proving shallow properties of the code.

In this paper we present the formal verification of a simple hypervisor, which we call baby hypervisor, using VCC, an automatic verifier for concurrent C (with annotations) developed at Microsoft [5]. The verification of the baby hypervisor is part of the Verisoft XT project, which also aims at the verification of the hypervisor of Microsoft's Hyper-V™. In comparison, the baby hypervisor and the architecture it virtualizes are very simple (e.g., neither a multi-core architecture nor concurrent code are considered). In the project, the baby hypervisor has played an important role of driving the development of the VCC technology and

⋆ Work funded by the German Federal Ministry of Education and Research (BMBF) in the framework of the Verisoft XT project under grant 01 IS 07 008.

G.T. Leavens, P. O'Hearn, and S.K. Rajamani (Eds.): VSTTE 2010, LNCS 6217, pp. 40–54, 2010.

applying it to system verification. For example, the region-based memory model of previous VCC versions exhibited serious performance problems in an earlier verification attempt of the baby hypervisor, which also led to the development of VCC's current memory model [6]. The baby hypervisor can serve a similar purpose outside the Verisoft XT project, and act as a benchmark or challenge for other verification tools.[1]

Our contribution is twofold: (i) We present the first verification of a hypervisor. It includes the initialization of the guest partitions and a simple shadow page table algorithm for memory virtualization. We verified the simulation of the guest partitions, and, since our modeling starts at host reset time, the assumptions are few and well-defined. (ii) We demonstrate how to apply automated verification to system software. In particular, we show a way to do simulation proofs in a first-order prover setting, how to model the underlying hardware, and reason about its interaction with the (mixed C and assembly) code.

The remainder of this paper is structured as follows. In Sect. 2 we give an overview of related work. In Sect. 3 we introduce VCC. In Sect. 4 we present an overview of our architecture, called *baby VAMP*, which we have formalized in VCC. In Sect. 5 we introduce the framework we use here for the verification of system software with VCC. System behavior is modeled by the execution of a *system program* that consists of an infinite loop of steps of the host architecture; system correctness is an invariant of this loop. In Sect. 6 we present an overview of the baby hypervisor data structures and invariants, and instantiate the simulation framework. The two main proof obligations are (i) the correctness of the hypervisor's top-level function and (ii) the simulation of guest steps by steps on the host in which no host exceptions. In Sects. 7 and 8 we evaluate and conclude.

## 2 Related Work

There are several projects with substantial results in the system verification area. The seminal work in pervasive systems verification was the CLI stack, which included the (very simple) KIT operating system [4]. More recent work was done in the projects FLINT, L4.verified, and Verisoft. FLINT focuses on the development of an infrastructure for the verification of systems software [8]. In L4.verified, the functional correctness of a high-performance C implementation of a microkernel was proven [9]. In Verisoft [13] large parts of the 'academic system', comprising hardware, system software, and applications, have been verified (e.g., cf. [1, 2] for work on the lower system software layers). In contrast to the present work, all the work above was based on interactive theorem proving, requiring significant human interaction and expertise. Of the above work, only KIT and Verisoft take into account user processes like we do. For microkernels this might be an acceptable compromise, assuming that a provably correct implementation of the kernel API already covers a substantial portion of overall system correctness. For hypervisors, however, the major part of its functionality is the simulation of the base architecture, and its verification should not be dodged.

[1] Verified source is available at `http://www.verisoftxt.de/PublicationPage.html`

There is also related but in-progress work in hypervisor verification. The Robin project aimed at verifying the Nova microhypervisor using interactive theorem proving [12]. Although on the specification side much progress has been made, only small portions of the actual hypervisor code are reported to be verified. In the Verisoft XT project [14], which the baby hypervisor verification was also part of, the verification of the hypervisor of Microsoft's Hyper-V$^{TM}$ using VCC is being attempted (cf. [10] for verification status). No code or specs are shared between these hypervisors; our work drove VCC development early on and thus helped empower VCC for more complex tasks.

## 3 VCC Overview

The Verifying C Compiler (VCC) is a verifier for concurrent C being developed at Microsoft Research, Redmond, USA, and the European Microsoft Innovation Center (EMIC), Aachen, Germany. Binaries and source are openly available and free for academic use. The VCC methodology aims at a broad class of programs and algorithms. Our overview here is focussed on our verification target (e.g., we do not consider concurrency); more information on VCC is available from [5,11].

*Workflow.* VCC supports adding specifications (e.g., in the form of function contracts or data invariants) directly into the C source code. During regular build, these annotations are ignored. From the annotated program, VCC generates verification conditions for (partial) correctness, which it then tries to discharge (under the hoods using the Boogie verifier [3] and the automatic theorem prover Z3 [7]). Ideally, all verification conditions can be proven. Otherwise, a counter example is produced or a violation of resource bounds is reported.

*Memory Model.* The C standard defines memory as a collection of byte sequences. Using this model directly proved to be inefficient. Instead, as a sound abstraction of the standard model, VCC now implements a typed memory model [6]. Pointers are modeled as pairs of types and addresses. We distinguish pointers to primitive types (e.g., integer) from pointers to non-primitive types (e.g., structs); the latter are also called 'object pointers'. Memory content is a mapping from primitive pointers to data. To ensure that differently typed pointers (and fields of different objects) do not overlap, VCC maintains a typedness predicate on pointers together with appropriate invariants, and inserts verification conditions that only typed memory is referenced. Typedness of pointers is inferred along structural type dependencies, e.g., a pointer to a field of a structure is known to be typed if the pointer to the structure is typed, and vice versa. These dependencies are also used to infer non-aliasing of pointers. For example, two typed references $\&p \rightarrow f$ and $\&q \rightarrow g$ can be disambiguated if $p \neq q$ or $f \neq g$.

To allow for the framing of function calls, VCC maintains a predicate for writability. At the beginning of a function, all pointers given by the function contract's writes clause are writable (and typed). For function calls, writability of the writes clauses is checked, and after the call memory contents, typedness and writability for non-written pointers are preserved. Pointers in the writes

clauses are still writable if the function ensures them to remain typed. Moreover, pointers may also be ensured as 'freshly typed', which will make them writable.

*Ownership and Invariants.* On top of the memory model, VCC implements and enforces an ownership model. For each object (i.e., non-primitive pointer), VCC maintains an owner field and a closed bit. The owner field of an object indicates which object it is owned by. If an object is closed, all objects it owns must be closed. The *domain* of a closed object is the set of all objects it transitively owns (and their fields). The currently executing thread (denoted as ***me***) is a special owner, which plays a role in memory reference checking. Pointers owned by it are called *wrapped* if closed and *mutable* otherwise. Extending the earlier checks, reading of a pointer is allowed if it is mutable or in the domain of a wrapped object. Writing to a pointer is allowed, if it is mutable and marked writable. Thus, while a domain remains unopened, its data cannot change. This allows for extended framing of calls. Ownership information is manipulated via ghost operations (setting the owner, wrapping, and unwrapping). These operations require write permissions on the objects they manipulate. Unwrapping a wrapped object opens it; its fields and owned objects become writable and wrapped. Wrapping a mutable object with wrapped closed objects closes it; its fields and owned objects will lose writability.

Objects can be annotated with invariants, which are meant to hold while the object is closed. Since invariants are only checked when wrapping they cannot talk about arbitrary state. Rather they may only refer to their domain, which is checked by VCC in an invariant admissibility check.

*Ghosts.* In addition to C types, VCC provides ghost types for use in annotations. The additional primitive types include unbounded integers, records, and maps. Moreover, VCC supports non-primitive ghost structures and ghost unions, which are fully-fledged objects with invariants. Ghost functions and data can be used for abstraction and for overcoming limitations of VCC's first order prover setting. For example, a linked list may be abstracted as a set of pointers to list elements with reachability relations being maintained with maps. The modification of implementation data in such scenarios typically involves doing a suitable ghost update. Such updates are done in *ghost code* inserted by the annotator. For soundness reasons, ghost code needs to be terminating and may not modify implementation variables (preventing information flow from ghost to implementation data). VCC checks this by a mixture of static and dynamic conditions.

*Syntax.* Object invariants are added to struct declarations using ***invariant*** clauses. Function contracts are given after the function signature. Writes clauses are given by ***writes***. Pre- and postconditions are specified using ***requires*** and ***ensures***; the clause ***maintains*** means both. In postconditions, ***old*** can be used to evaluate an expression in the prestate, ***result*** refers to the function's return value, and ***returns***(*x*) abbreviates ***ensures***(***result***≡ *x*). Listing 1 shows a small program with annotations. In the function *triple*, the bound requirement is needed to pass the overflow check generated for the arithmetic, the ***writes***

```
struct Even {
  unsigned v; invariant(v%2≡ 0)
};
unsigned triple(struct Even ∗e)
  requires(e→v < 4711)
  maintains(wrapped(e))
  writes(e) returns(old(e→v))
  ensures(e→v≡ 3∗old(e→v))
{
  unsigned x = e→v;
  unwrap(e);
  e→v = x + x + x;
  wrap(e);
  return x;
}
```

**Listing 1.** VCC Syntax Example

clause is needed to allow unwrapping the object $e$, and the proof that the object invariant holds when wrapping $e$ requires that validity of the invariant after unwrapping.

## 4 Architecture

The *baby VAMP* architecture is a 32-bit RISC architecture with 44 instructions, two privilege levels (user and system mode), single-level address translation (without TLB), and interrupt handling. In this section we describe the specification model of the baby VAMP hardware architecture. In VCC, architecture state and steps are defined using ghost types and functions, respectively. We use the model to specify (i) transitions of the *VAMP simulator* modeling steps of the system (cf. Sect. 5), (ii) effects of assembly code execution, and (iii) transitions of the abstract guest machines.

*Configuration.* Words and addresses of the machine are encoded as 32-bit integers. The memory $m$ is modeled as a map from addresses to words. A page is an aligned chunk of *PG_SZ:=1024* words in memory. Accordingly, an address *a* is decomposed into a page and a word index, *PX(a):=a / PG_SZ* and *WX(a):=a % PG_SZ*. For a page index *px* we define *PA(px):=px∗PG_SZ*.

Listing 2 shows the configuration of the VAMP machine (making use of VCC maps and records). It consists of the word addressable memory and the processor configuration. The latter consists of the normal and delayed program counters *pcp* and *dpc* (there is one delay slot) and the general- and special-purpose register files *gpr* and *spr*. For this paper the following special-purpose registers are

```
typedef unsigned _int32 v_word;
typedef v_word v_mem[unsigned];
typedef struct vcc(record) v_proc {
  v_word gpr[unsigned],
    spr[unsigned], dpc, pcp;
} v_proc;

typedef struct vcc(record) v_mach {
  v_mem m;
  v_proc p;
} v_mach;
```

**Listing 2.** Baby VAMP Configuration

of interest: (i) *PTO* and *PTL* for the page table's origin and maximum index, (ii) *MODE*, equal to *0* and *1* in system / user mode, respectively, and (iii) *EPC*, *EDPC*, *EMODE*, *EDATA* for registers used to save the current program counters, mode, and additional data (as exception cause and data) in case of an interrupt.

*Memory Access.* In user mode a memory access to a virtual address is subject to address translation. This translation is done via a page table in main memory, which maps virtual page indices to page table entries (PTEs). The start of the currently active page table is designated by the (physical) page index of its first entry and the length by its maximum index, stored in the *PTO* and *PTL* registers. Each page table entry *pte* is a machine word encoding three components: (i) the valid bit *V(pte):=pte & 1024* indicating whether the entry can be used for any access, (ii) the protection bit *P(pte):=pte & 2048* indicating whether the entry cannot be used for write access, and (iii) the physical page index *PPX(pte):=PX(pte)* indicating the location of the page in physical memory. To compute the translation of a virtual address *a* via a page table at a certain origin *pto*, we first compute the address of the corresponding page table entry as *v_ptea(pto, a):=PA(pto)+ PX(a)*. Second, we look up the entry in the memory, *v_pte(m, a, pto):=m[v_ptea(pto,a)]*. Finally, the translated address is obtained by concatenating the PTE's page index and the word index of the input address, *v_ta(m, a, pto):=PA(PPX(v_pte(m,a,pto)))+ WX(a)*. Given a memory, an input adress, a translation flag, and a page table origin a memory read result is defined as *v_mem_read(m, a, t, pto):=m[(t ? v_ta(m,a,pto): a)]*. Likewise, writing *v* is formalized with the function *v_mem_write(v, a, t, pto, m)*, which returns an updated memory.

Memory accesses may fail and cause an interrupt (a bus error or page fault). Untranslated accesses fail in case of a bus error, i.e., if the accessed address lies outside the physical memory whose size is given by the maximum physical page index *max_ppx*, a machine parameter. Translated accesses fail if (i) the virtual address is outside virtual memory (virtual page index outside the page table), (ii) the page-table entry address is outside physical memory, (iii) the page-table entry is invalid, (iv) the page-table entry is protected and a write is attempted, or (v) the translated address is outside physical memory. Given the memory, the maximum physical page index, the address to access, the translation flag, the page table origin and a flag indicating write or read access, the predicate *v_pf(m, max_ppx, a, t, pto, ptl, w)* indicates the presence of such a failure.

*Interrupt Handling.* Interrupts may trigger due to internal events like page faults, illegal / unprivileged instructions, and traps, or due to external events like reset and device interrupts. For the baby hypervisor verification, the only external interrupt considered is reset. Interrupts may be masked by the programmer, i.e., disabled, by setting corresponding bits in the special-purpose status register *SR*. We distinguish between maskable, i.e., interrupts which can be disabled, and non-maskable interrupts (in our case only reset).

If an interrupt occurs, the interrupt service routine (ISR) is invoked by: (i) setting exception cause and data registers and saving program counters, mode, and status registers to the SPR, (ii) setting the program counters to the start of the ISR, which handles the interrupt, and (iii) masking all maskable interrupts by setting the status register to zero.

*Semantics.* The architecture's main function is the step function with the following signature: ***v_mach*** *v_step(**v_mach** mach, **v_word** max_ppx, **bool** reset)*. It takes as input the current machine state, the maximum physical memory space, and a reset signal. It returns an updated machine state which is either computed by fetching and executing a single instruction of the machine, or by jumping to the interrupt service routine. Given a machine configuration, instruction fetch is defined as a simple memory read *v_mem_read(mach.m, mach.p.dpc, mach.p.spr[MODE], mach.p.spr[PTO]))*. An interrupt is triggered by the reset signal, by a page fault during instruction fetch, or during instruction execution.

## 5 Simulation Framework

In this section we show how to bring VCC and system verification together, allowing to reason on overall system correctness in an efficient and pervasive manner. To do this, we model the architecture state and steps in VCC such that later program verification is not made hard. We then show how to express top-level system correctness as a *system program*.

*Representing the Architecture.* Since C memory is sufficiently low-level, we use a prefix of it, starting from address zero, to represent the architecture's memory. We hold the processor state in a separate structure ***proc_t*** $*h$ located outside this region; it matches its abstract counterpart ***v_proc*** with the exception of arrays being used instead of maps. We define *abs_p(h)* to abstract the processor state and *abs_m(max_ppx, m):=λ(a; a < PA(max_ppx+1)? m[a] : 0)* to abstract memory. Using these definitions, the function *abs0(h)* abstracts from a processor configuration and a zero-based memory, returning a state of type ***v_mach***.

We perform machine steps by statements of the form *il = sim_vamp(h,reset)*. The function *sim_vamp* takes as inputs the processor state, a Boolean flag indicating reset, and, implicitly, the zero-based memory. It operates directly on the processor state and the VCC memory, simulating a single instruction. For simplicity, it also returns the interrupt level associated with the instruction (or *IL_NONE* if the instruction did not cause an exception and no reset occurred).

The contracts of *sim_vamp* have to be chosen carefully. Contradictory postconditions would make VCC unsound. We ensures consistency of the *sim_vamp* contracts by verifying them (in VCC) against a concrete implementation, which we refer to as the *baby VAMP simulator*. Moreover, the contracts have to comply to the (trusted) architecture specification described in the previous section. The obvious way to achieve this is by describing the effects of *sim_vamp* using the abstraction *abs0* and the transition function *v_step* in a postcondition ***ensures****(abs0(h)≡ **old**(v_step(abs0(h),max_ppx,reset)))*. However, this straightforward contract is impractical, since the caller of *sim_vamp* would have to ensure

that the complete memory is uniformly typed and writable. We realize a less invasive approach: only the specific memory cells that the architecture accesses in a certain step need to be typed and mutable (and in case of the store operand also writable).

*Basic Simulation Pattern.* A convenient way to show system correctness is to prove a simulation theorem between the concrete system and some abstraction of it. The simplest description of a system (which we call *system program*) is given by an infinite loop executing steps of the architecture modeled by *sim_vamp*. The simulation property is stated as a loop invariant. This invariant relates the initial (i.e., before entering the loop) and the current states of the system under the abstraction, and claims that the current abstract state is reachable from the initial abstract state taking steps of the abstract transition system.

In general, reachability cannot be stated in first-order logic. We avoid this problem by introducing a counter increased in each step. Let ***mathint*** denote the type of unbounded integers. Given an abstract step relation $S$, the predicate $R(s,t,n) \iff (s \equiv t \wedge n \equiv 0) \vee (n>0 \wedge \exists(\textbf{mathint}\ u;\ R(s,u,n-1) \wedge S(u,t)))$ is an exact first-order definition of $n$-step reachability.

With the abstraction denoted as $A(\textbf{proc\_t}\ *h)$ and additional implementation invariants as $I$, a basic simulation pattern in VCC has the following form (where ***old()*** here evaluates an expression in the state before entering the loop):

```
sim_vamp(h, true);
while (1)
  invariant(I ∧ ∃(mathint n; R(old(A(h)), A(h), n)))
  sim_vamp(h, false);
```

*Extended Simulation Pattern.* Next we want to combine code verification with the execution of the architecture. Note, that in the system program above we can first unroll the ***while***-loop, group chunks of *sim_vamp* computations together, and finally describe their effects by contracts. Such a chunk may be given, e.g., by the execution of assembly instructions, where the semantics of each instruction can be expressed in terms of *sim_vamp*. Similarly a chunk may consist of compiled C code. Since we use the same VCC memory model for the architecture and the C code verification, and by assuming compiler correctness, we can describe these chunks by their corresponding C implementation.

We show how to unroll and divide the ***while***-loop into four parts as they might occur in a typical (sequential) OS kernel verification. (i) The architecture executes under *sim_vamp* until some interrupt occurs. (ii) The kernel is entered and the interrupted state (here: the processor registers) must be saved, which must be implemented in assembly rather than in pure C. The semantics of assembly instructions can be fully expressed by *sim_vamp* and the effects of the complete code by the contracts of a function *kernel_entry(h)*. (iii) The kernel's main function *kernel_main(il)* implemented in C then handles the interrupt. (iv) Exiting the kernel and switching to the user again requires an assembly implementation, the effects of which we specify by the contracts of the function *kernel_exit(h)*. Accordingly, we define a more elaborate simulation pattern, which combines

reasoning on C code, assembly code, and user steps into a single, pervasive correctness proof:

```
il = sim_vamp(h, true);
while (1)
  invariant(I ∧ ∃(mathint n; R(old(A(h)), A(h), n)))
{ kernel_entry(h); kernel_main(il); kernel_exit(h);
  do il = sim_vamp(h, false); while (il≡ IL_NONE); }
```

## 6 Hypervisor Implementation and Correctness

We present the implementation and verification of a simple hypervisor, which we call *baby hypervisor*. The baby hypervisor virtualizes the architecture defined in Sect. 4, and its correctness is expressed and verified using the previously presented simulation framework.

The recipe for virtualizing the different guest architecture is simple. We always make the guests run in user mode on the host, regardless of the mode they think they are in. Hence, we obtain full control over guests, as they run translated and unprivileged. Under hardware address translation, we virtualize guest memory by setting up so-called host and shadow page tables for the guest running in system and user mode, respectively. The host page tables will map injectively into host memory with different regions allocated for each guest. The shadow page table of a guest is set up as the 'concatenation' of the guest's own page table and its host page table. To make sure that the guest cannot break this invariant, we map all host or shadow page table entries to the guest page table as read-only. Thus, attempts of a guest to edit its page table and also to perform a privileged operation (e.g., change the page-table origin) will cause an exception on the host and be intercepted by the hypervisor. The hypervisor will then emulate the exception operation in software.

We express the whole system verification scenario following the pattern of our simulation framework introduced in Sect. 5. The state of the system we intend to simulate consists of a vector of architecture states (cf. Sect. 4), where each state represents the state of a guest partition. The transition function is almost identical to the architecture's transition function; deviations are that guest traps in system mode are used to issue hypercalls (we only implement a simple 'yield' call for cooperative partition scheduling), and (for implementation reasons) the guest's page table length needs to be bounded by a maximum virtual page index parameter. Based on the hypervisor's data structure we then define an abstraction from the implementation state into the simulated, i.e., the guest's, state. The instantiation of the system program features the different code parts of the hypervisor (the hypervisor main function implemented in C, and assembly portions for hypervisor entry and exit), and architecture steps to model guest execution on the host. For simulation, we state the reachability of (abstracted) guest configurations from their initial state as an invariant of the main loop.

```
typedef struct guest_t {
  proc_t pcb;
  v_word max_ppx, max_vpx,
    *gm, gmo, *hpt, hpto, *spt, spto;
} guest_t;

typedef struct hv_t {
  v_word ng;
  guest_t *g, *cg;
} hv_t;
```

**Listing 3.** Hypervisor Data Structures

*Parameters and Memory Map.* The hypervisor is configured at boot time by four parameters encoded in four bytes at the beginning of its data segment at address *DS_START*: (i) the data segment size *DS_SIZE*, (ii) the number of guests *NG*, (iii) the maximum physical page index for each guest *MAX_PPX*, and (iv) the maximum virtual page index for each guest *MAX_VPX*. At boot time, the data segment is a byte array *DS:=**as_array**((**uint8_t***)DS_START,DS_SIZE)* (where ***as_array**(a,n)* denotes an array object with base *a* and size *n*). Given the last three parameters, we can compute the size to allocate and align all global data structures of the hypervisor. This size must not be larger than *DS_SIZE*, a condition that we abbreviate as *VALID_DS*.

*Data Structures.* Listing 3 shows the two main data structures of the hypervisor implementation without invariants. The structure ***guest_t*** holds all data for a single partition: (i) the maximum physical and virtual page index for the guest (that remain constant after initialization), (ii) the processor registers when suspended (modeled by the struct type ***proc_t*** already used in Sect. 5), (iii) pointers to the guest's memory as well as the host and shadow page tables, and (iv) for each of these, the index of the first (host) page they are stored in. The top-level data structure ***hv_t*** holds the number of guests, a pointer to an array of their data structures, and the current guest pointing into that array. When the boot procedure completes, a wrapped ***hv_t*** data structure is returned at the beginning of the data segment; we abbreviate *HV:=((**hv_t***)DS_START)*.

*Invariants.* All data structures have been annotated with invariants on subtypes, typedness / non-aliasing, ownership, and, in general, data. Importantly, by the non-aliasing and ownership invariants, all of the guests' data structures are separated. The most complex invariants are those for the host and shadow page tables entries, declared in the ***guest_t*** structure. Let *in_pt* indicate if a page index falls into a page table given by *pto* and *ptl*, i.e., *in_pt(x, pto, ptl):=pto* $\leq x \wedge (x \leq pto + (ptl\ /\ 1024))$ where *1024* is the number of page-table entries per page. A host PTE with index *x* must be valid, point to the guest page *x* as stored on the host, and be protected if covered by the guest's page table:

```
bool inv_hpt_entry(guest_t *g, v_word x)
  returns(V(g→hpt[x]) ∧ PPX(g→hpt[x])≡ g→gmo + x ∧
    (P(g→hpt[x]) ⟺ in_pt(x, g→pcb.spr[PTO], g→pcb.spr[PTL])));
```

A shadow PTE with index *x* is valid iff the translation yields no guest bus error (i.e. if the address is not in range) and the guest PTE is valid. Validity of the

shadow PTE then implies that it points to the host page designated by guest and host translation and it is protected if the guest page is protected or part of the guest page table. We define

```
bool inv_spt_entry(guest_t *g, v_word x)
  returns((V(g→spt[x]) ⟺
    ¬v_bus_error(v_ptea(o, PA(x)),g→max_ppx) ∧ PPX(e) ≤ g→max_ppx ∧ V(e)) ∧
   (V(g→spt[x]) ⟹ (PPX(g→spt[x]) ≡ PPX(g→hpt[PPX(e)]) ∧
     (P(e) ∨ in_pt(PPX(e),o,l) ⟹ P(g→spt[x])))));
```

where *o:=g→pcb.spr[PTO]*, *l:=g→pcb.spr[PTL]*, and *e:=g→gm[v_ptea(o, x)]*.

*Abstraction.* Given a guest's implementation data structure, we can construct an abstract machine state in terms of the architecture definition from Sect. 4. This involves constructing the guest's memory and processor state. The latter is done in two contexts. When the hypervisor executes, all registers are held in the guest's *pcb* structure. In this case, the abstraction just operates on a guest pointer *g* (the expression *(R){...}* defines a record constant):

```
v_mach abs_g(guest_t *g)
  returns((v_mach) { .p = abs_p(&g→pcb), .m = abs_m(g→max_ppx,g→gm) });
```

When a guest executes on the host, the partition control block only stores the guest SPR (accesses to those are emulated) while the GPR and program counters are stored in the actual host's processor state. With the host processor state denoted as *h* we define guest abstraction in this case as follows (the expression $r$ / {.$a$ = $b$} denotes record update):

```
v_mach abs_gh(guest_t *g, proc_t *h)
  returns((v_mach) { .p = abs_p(h) / { .spr = abs_p(&g→pcb).spr },
            .m = abs_m(g→max_ppx, g→gm) });
```

Let us define how the host's SPR are set up while the guest runs on it. When guests execute, the SPR is set up to use address translation and the shadow or host page table depending on the guest mode; in guest system mode the host page table origin and the maximum physical page index are used, in guest user mode the shadow page table origin and the guest's page table length index are used. Given a guest structure *g* and a host processor state *h* we define

```
bool spr_inv(guest_t *g, proc_t *h)
  returns(H.spr[MODE] ∧ (G.spr[MODE]
    ? H.spr[PTO]≡ g→spto ∧ H.spr[PTL]≡ G.spr[PTL]
    : H.spr[PTO]≡ g→hpto ∧ H.spr[PTL]≡ g→max_ppx));
```

where *H:=abs_p(h)* and *G:=abs_p(g)*. Given this setup, we can (in VCC) prove the equivalence of memory operations: (i) the absence of host page faults implies the absence of guest page faults and (ii) if there's no host page fault the result of reads (a data word) and writes (an updated memory) are equal.

*Simulation Loop.* Listing 4 shows the instantiatation of the system program for the baby hypervisor, where the relation $R$ is $n$-step reachability of guest transitions. As a boot requirement (i.e., a *sim* precondition) a writable, large-enough

```
void sim(proc_t *h)
  requires(VALID_DS ∧ wrapped(h)) writes(h, extent(DS))
{
  spec(v_word ng = NG, max_ppx = MAX_PPX;)
  v_il il;
  sim_vamp(h, true);
  hv_dispatch(h→spr[ECA],h→spr[EDATA]);
  spec(v_mach G0[unsigned] = λ(unsigned i; i < ng; abs_g(HV→g+i));)
  do
    invariant(∀(unsigned i; i < ng; ∃(mathint n;
      R(G0[i],abs_g(HV→g+i),max_ppx,n))))
    invariant(wrapped(HV) ∧ wrapped(h))
  {
    restore_guest(h,&HV→cg→pcb);
    do
      invariant(∀(unsigned i; i < ng; ∃(mathint n; R(G0[i],
        HV→g+i≡ HV→cg ? abs_gh(HV→cg, h) : abs_g(HV→g+i), max_ppx, n))))
      il = sim_vamp(h, false);
    while (il≡ IL_NONE);
    save_guest(&HV→cg→pcb,h);
    hv_dispatch(h→spr[ECA], h→spr[EDATA]);
  } while (1);
}
```

**Listing 4.** Hypervisor Simulation Loop

data segment and a wrapped and writable host processor state are assumed (note that we have not yet formalized remaining memory layout, such as the stack and code segment). Ghost code and declarations are given using the VCC keyword ***spec*** (e.g., the map of initial abstract guest machine configurations). The code that runs on the host architecture is represented by assembly portions *save_guest* and *restore_guest* (used on kernel entry and exit), *hv_dispatch*, and (for generic host execution) by *sim_vamp* (cf. Sect. 5). The hypervisor's main function *hv_dispatch* takes the architecture's exception cause and data as parameters *eca* and *edata*. The main code paths of the hypervisor are initialization (function *handle_reset*) and handling exceptions caused by guests (functions *handle_illegal* and *handle_pf*); calling *hv_dispatch* with *eca*≡ *IL_NONE* is not allowed. We sketch the individual components in a little more detail below.

*Assembly Parts.* A single assembly routine is located at the start of the interrupt service routine (i.e., address 0). It consists of three parts: (i) save the registers of the current guest (unless on reset), (ii) set up the stack and call the main C function of the hypervisor, *hv_dispatch*, (iii) restore the registers of the (possibly new) current guest. In the simulation loop (which shows dynamic instances of this code in the execution) we have represented this code in slightly abstracted form; calls to the dispatcher (including setting up the stack and the parameters) are represented as a regular call to *hv_dispatch*, we only specify the effects of save

and restore via function contracts *save_guest* and *restore_guest*, and we simply omit the save part after reset.[2] The specifications of save and restore describe the copying of register between a guest's partition control block and the host.

*Boot.* Calling *hv_dispatch* with *eca*≡ *IL_RESET* will lead to the execution of the reset procedure of the hypervisor. Just as *sim*, the function in this case requires a writable and valid data segment. Using a boot-time allocator, the reset code of the hypervisor allocates and initializes its data structures in the data segment, i.e., the hypervisor structure, the individual guest structures, and for each guest its memory, shadow page table, and host page table. For verifying this code we make use of VCC's memory reinterpretation feature, which allows us to split and convert the data segment (initially a byte array) into the necessary (page-aligned) typed objects. As a postcondition, *hv_dispatch* guarantees to return *HV* wrapped and with properly initialized guests.

*Host Exceptions.* On host interrupts *hv_dispatch* gets called after saving the guest state with a non-reset exception cause *eca*. Unless the guest issues a hypercall (which it does by executing a trap in system mode), the dispatcher has to emulate an interrupt, a privileged operation, or a page table writes of a guest. Interrupt injection is mostly done in *hv_dispatch* directly, while the latter two cases are implemented by the functions *handle_illegal* and *handle_pf*, respectively.

The non-reset contracts for *hv_dispatch* requires the hypervisor structure *HV* to be wrapped and writable and that *eca* contains an actual interrupt level that occurred on the host while executing the current guest *HV*→*cg*. The function ensures to return a wrapped hypervisor structure. For non-continue type interrupts (in our case every interrupt but a trap), it also guarantees to emulate a guest step. Trap interrupts are special for two reasons: (i) Traps that the guest executes in system mode are used to issue hypercalls. There is only a single hypercall in the baby hypervisor, which is a *yield* call to switch execution round-robin fashion to the next guest partition. From a guest's point of view, that call is just a nop (i.e., it only increments program counters). (ii) Traps reach the hypervisor with the guest's program counters already pointing to the next instruction. When emulating traps the hypervisor must not increment them again. We currently express these peculiarities by a special post condition describing the guest processor updates. In the context of the simulation loop, the combined effect of hardware and hypervisor updates give the complete semantics of the guest step.

## 7 Evaluation

Our work consists of four components (VAMP spec and simulator, hypervisor implementation, and system program) comprising 2.5k C code tokens and 7.7k annotation tokens, which comprise data invariants, function contracts (including loop invariants), ghost code, and (proof) assertions. Roughly a quarter of the

[2] Justifying the abstractions is the subject of future work. Note that almost identical assembly code has been verified previously [1].

**Table 1.** Annotation Effort (Tokens) and Runtimes (Average, Standard Deviation)

| | Code | Contract | Ghost Code | Proof | ∅ | $\sigma$ |
|---|---|---|---|---|---|---|
| *hv_dispatch* | 158 | 198 (1.3) | 90 (0.6) | 441 (2.8) | 930s | 574s |
| *handle_reset* | 60 | 150 (2.5) | 30 (0.5) | 42 (0.7) | 545s | 295s |
| *handle_pf* | 102 | 96 (0.9) | 40 (0.4) | 208 (2.0) | 473s | 414s |
| *reset_guest* | 125 | 131 (1.0) | 46 (0.4) | 53 (0.4) | 213s | 38s |
| *handle_movi2ptl* | 93 | 238 (2.6) | 6 (0.1) | 66 (0.7) | 98s | 31s |
| *handle_movi2pto* | 69 | 162 (2.3) | 6 (0.1) | 57 (0.8) | 74s | 20s |
| *handle_illegal* | 90 | 47 (0.5) | 48 (0.5) | 149 (1.7) | 53s | 8s |
| *update_spt* | 62 | 175 (2.8) | 0 (0.0) | 140 (2.3) | 14s | 6s |
| System program | 54 | 587 (10.9) | 205 (4.6) | 378 (7.0) | 1241s | 794s |

annotation tokens belong to the architecture specification and another quarter to the system program. Overall proof time is ca. 1 hour on one core of a 2.66 GHz Intel Core Duo machine, depending on random seeds for Z3's heuristics. The hypervisor C code consists of 9 loops in 45 functions. In Table 1 we list token counts and runtimes (for 20 runs) of its 8 most complex functions and the system program. Particularly, we give ratios of contract, ghost code, and proof tokens versus C tokens in brackets. The latter ratio may be considered an upper bound, and will likely decrease with better automation. Measuring person effort is hard, because VCC has undergone major development since we started, and therefore annotations had to be often revised. The hypervisor was never tested, and a number of bugs could be found and fixed during the verification. Notably, some of these became apparent when doing the proof of the system program, at the HW/SW boundary (e.g., emulating guest traps, cf. Sect. 6).

## 8 Future Work and Conclusion

We have presented a technique and framework for pervasively verifying system correctness based on automated methods. Our approach precisely models the underlying system architecture and represents it in a first-order logic based program prover (in our case VCC), which is then used to prove the mixed-language system software correct. We used this framework to show for the first time (i) the functional correctness of a small hypervisor, expressed as the architecture-conforming simulation of guest machines by the host system, and (ii) the feasibility of applying automated methods in the context of functional (systems) verification. We are confident that much of the presented methods can be used as a basis for further extensions, and that our verification may serve as a valuable benchmark for the automated reasoning and software verification community. Meanwhile, the feedback (in form of bug reports and optimization suggestions) provided by the baby hypervisor verification as first complex and completed proof target substantially contributed to the development of VCC.

There are several directions of future work, some of them ongoing. The assembly portions of the hypervisor have been specified but not yet verified against their implementation which almost only consists of straightforward code to copy register contents. Our framework allows for a seamless integration of assembly

verification into VCC (using the VAMP simulator). Still, a detailed soundness proof of the presented approach (including compiler calling convention and correctness), is ongoing work. Moreover, the presented framework should be extended to more complex software and hardware designs. On the hardware side, this should cover the modeling of multi-core systems, buffers (e.g., TLBs), and devices. On the software side, the step to the verification of multi-threaded or preemptive kernels is crucial. Parallelism in the code and in the architecture can be dealt with by using VCC's concurrency features, as e.g., two-state invariants. We plan to use two-state invariants for a more general way to express simulation, which should also scale to a concurrent setting. Adaptation to this new form will presumably require only little additional proof effort.

*Acknowledgments.* We wish to thank Ernie Cohen and Michał Moskal for many helpful comments and discussions of our work.

## References

1. Alkassar, E., Hillebrand, M.A., Leinenbach, D.C., Schirmer, N.W., Starostin, A., Tsyban, A.: Balancing the load: Leveraging a semantics stack for systems verification. JAR 42(2-4), 389–454 (2009)
2. Alkassar, E., Paul, W., Starostin, A., Tsyban, A.: Pervasive verification of an OS microkernel: Inline assembly, memory consumption, concurrent devices. In: Leavens, G.T., O'Hearn, P., Rajamani, S. (eds.) VSTTE 2010. LNCS, vol. 6217, pp. 71–85. Springer, Heidelberg (2010)
3. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006)
4. Bevier, W.R.: Kit and the short stack. JAR 5(4), 519–530 (1989)
5. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: A practical system for verifying concurrent C. In: Urban, C. (ed.) TPHOLs 2009. LNCS, vol. 5674, pp. 1–22. Springer, Heidelberg (2009)
6. Cohen, E., Moskal, M., Schulte, W., Tobies, S.: A precise yet efficient memory model for C. In: SSV 2009. ENTCS, vol. 254, pp. 85–103. Elsevier, Amsterdam (2009)
7. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
8. Feng, X., Shao, Z., Guo, Y., Dong, Y.: Certifying low-level programs with hardware interrupts and preemptive threads. JAR 42(2-4), 301–347 (2009)
9. Klein, G., Elphinstone, K., Heiser, G., et al.: seL4: Formal verification of an OS kernel. In: SOSP 2009, pp. 207–220. ACM, New York (2009)
10. Leinenbach, D., Santen, T.: Verifying the Microsoft Hyper-V Hypervisor with VCC. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 806–809. Springer, Heidelberg (2009)
11. Microsoft Corp. VCC: A C Verifier, `http://vcc.codeplex.com/`
12. Tews, H., Völp, M., Weber, T.: Formal memory models for the verification of low-level operating-system code. JAR 42(2-4), 189–227 (2009)
13. The Verisoft Project (2003), `http://www.verisoft.de/`
14. The Verisoft XT Project (2007), `http://www.verisoftxt.de/`