Completing the Automated Verification of a Small Hypervisor - Assembler Code Verification*

Wolfgang Paul, Sabine Schmaltz, and Andrey Shadrin

Saarland University, Germany
{wjp,sabine,shavez}(at)wjpserver.cs.uni-saarland.de

Abstract. In [1] the almost complete formal verification of a small hypervisor with the automated C code verifier VCC [2] was reported: the correctness of the C portions of the hypervisor and of the guest simulation was established; the verification of the assembler portions of the code was left as future work. Suitable methodology for the verification of Macro Assembler programs in VCC was given without soundness proof in [3]. A joint semantics of C + Macro Assembler necessary for such a soundness proof was introduced in [4]. In this paper i) we observe that for two instructions (that manipulate stack pointers) of the hypervisor code the C + Macro Assembler semantics does not suffice; therefore we extend it to C + Macro Assembler + assembler, ii) we argue the soundness of the methodology from [3] with respect to this new semantics, iii) we apply the methodology from [3] to formally verify the Macro Assembler + assembler portions of the hypervisor from [1], completing the formal verification of the small hypervisor in the automated tool VCC.

1 Introduction

Kernels and Hypervisors: kernels and hypervisors for an instruction-set-architecture (ISA) M run on processors with ISA M and have basically two roles

- the simulation/virtualization of multiple *guests* or *user virtual machines* of ISA M
- the provision of services for the users via system calls (e.g. inter process communication)

The salient difference between kernels and hypervisors is that under kernels guests only run in user mode, whereas under hypervisors guests are also allowed to run in system mode. Thus, hypervisors must implement two levels of address translations (either supported by hardware features like nested page tables or in software using shadow page tables), whereas kernels must only realize one such level.

* Work partially 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. Authors in alphabetic order.

Kernel and hypervisor verification comes in 3 flavours:

- Verification of the C code alone.

  A famous example is seL4 [5]. Because kernels and hypervisors cannot be written exclusively in C such a proof is necessarily incomplete, but we will see shortly that closing this gap is not hard.

- Complete verification at the assembler level.

  Examples are the pioneering work on KIT [6] and the current effort in the FLINT project [7]. Complete code coverage can be reached in this way, but, due to the exclusive use of assembler language, productivity is an issue.

- Verification of both the C portion and the non C portion based on a joint semantics of C + inline assembler [8, 9] or C + Assembler functions.

  As already shown in [10] C portions and non-C portions of a kernel can be verified separately and the correctness proofs can then be joined into a single proof in a sound way. The same could be done to cover the non C portions of seL4. In an interactive prover like Isabelle, which is used in [9] and [5], formal work can directly follow the paper and pencil mathematics. A small extra effort is needed if we want to perform such work in an automated C code verifier like VCC.

For work applying formal methods specifically to hypervisors consider the Nova microhypervisor [11], the recent MinVisor verification effort [12], or the partial verification of the Microsoft Hyper-V hypervisor [13].

The baby hypervisor [1] virtualizes a number of simplified VAMP [14, 15] (called *baby VAMP*, see Fig. 2) guest processors (*partitions*) on a sequential baby VAMP *host processor*. The baby VAMP ISA is a simplified DLX-ISA (which is basically MIPS). The simplified VAMP architecture this work is based on does not offer any kind of virtualization support. Privileged instructions of guests (running in system mode) cannot be executed natively on the host. Instead, any potentially problematic instruction (e.g. write to the page-tables, change of page-table origin) causes an interrupt on the host machine and is subsequently virtualized by the baby hypervisor (see Fig. 3).

The baby hypervisor guarantees memory separation of guests by setting up an address translation from *guest physical addresses* to *host physical addresses* by defining a *host page table* [16] for each guest. A host page table is composed with the respective guest page table (if the guest itself is running in user mode) by the baby hypervisor to form a *shadow page table* that provides the direct translation from *guest virtual addresses* to *host physical addresses*. Then, running the host processor in user mode with the page table origin pointing to the shadow page table is sufficient to correctly virtualize the guest – as long as the guest does not perform changes to its own page table. To detect this case, the baby hypervisor marks those pages containing the guest page table as read only in the shadow page table. In case of a write access to the guest page table,

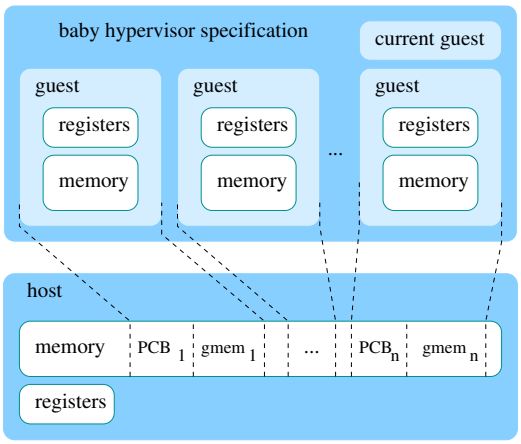

Fig. 1: Baby hypervisor overview

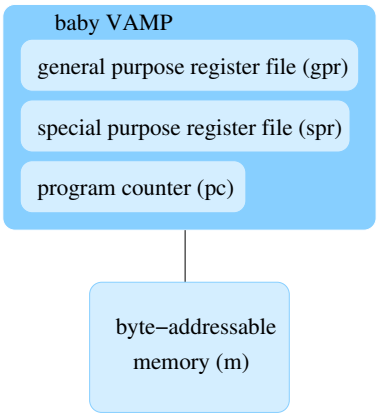

Fig. 2: Overview of the baby VAMP model

a page-fault interrupt occurs, which allows the baby hypervisor to correctly virtualize the guest updating its page table.

As illustrated in Fig. 1, guest machines virtualized by the baby hypervisor are represented by memory regions (data structures of the baby hypervisor implementation) of the host machine. *Process-control-blocks* (PCBs) correspond to register contents of not-currently-running guests. At the beginning of the interrupt handler of the baby hypervisor, guest registers are saved to their corresponding PCB, the function `hv_dispatch()` is called to virtualize the instruction causing the interrupt, and, at the end of the handler, guest registers are restored to the host machine registers.

Specifying assembler portions of code in a C verifier: C is a universal language, hence it can simulate any other language. In [1], in order to verify the correctness of the baby hypervisor, a baby VAMP interpreter is implemented in C. For the verification, the execution of hardware steps of the host processor that directly emulate guest steps are replaced by calls of the interpreter (which is implemented in such a way that it performs a single step of the baby VAMP model, i.e. executes a single machine instruction). The data structures of the interpreter are the obvious ones: i) hardware memory, which is fortunately already part of the C memory model of VCC and ii) processor registers, which are stored in a struct in a straightforward way. Based on this *and a specification of the effects of process save and restore*, the authors of [1] succeed to prove process separation of the guests. We are left with the problem to verify process save and restore in VCC and to integrate this proof with the existing formal proofs in a sound way. Consider that in the context of system verification, the notion of soundness encompasses that the resulting integrated formal model forms a sound abstraction of the physical machine's execution.

Verifying Macro Assembler in C: Theoretically, one could now try to prove properties of assembler programs in VCC by proving the properties for the interpreter running the re-

sulting machine code programs. The expected bad news is that this turned out to be inefficient. The good news is that modern hypervisors tend to use Macro Assembler instead of assembler. The control operations of Macro Assembler (stack operations, conditional jumps to labels) fit C much better than the (unstructured) jump and branch instructions of assembler language. This in turn permits to perform a semantics-preserving translation of Macro Assembler code to C code. That this works in an extremely efficient way was shown in [3]. Indeed, in [17], the author reports about the (isolated) verification of all Macro Assembler portions of the Microsoft hypervisor Hyper-V. Thus, it seems that we are left with i) the task to formally verify the Macro Assembler portions of the small hypervisor from [1], ii) the task to integrate this into the formal proof reported in [1], and iii) to show that this is sound relative to a joint semantics of C + Macro Assembler presented in [4]. We achieve the first two tasks by extending the VCC proof of [1], and we provide a pencil-and-paper proof for the third task.

The last two instructions: It turns out that two instructions of the hypervisor code are *not* compatible with the chosen Macro Assembler semantics: in our Macro Assembler, there is a built-in abstract notion of the stack. The corresponding stack pointer registers are not visible anymore on Macro Assembler level; they belong to the implementation. In order for the C + Macro Assembler code of the baby hypervisor to run properly when an interrupt is triggered on the host machine, however, they must be set up with appropriate values so that the stack abstraction for C + Macro Assembler is established. The good news is that this is done without using control instructions of the ISA, thus, the method from [3] can still be used. The moderately bad news is that in the end soundness has to be argued relative to a joint semantics of 3 languages: C + Macro Assembler + assembler.

Outline In Section 2, we introduce a rather high-level stack-based assembler language that we call *Macro Assembler* (*MASM*, not to be confused with Microsoft's MASM) and merge it with a very low-level intermediate language for C, *C-IL*, yielding an integrated semantics of *C-IL* and *MASM* – which is amenable to verification with VCC. In order to justify that the integration of semantics is done correctly, we state compiler correctness simulation relations for the two languages as pencil-and-paper theory in Section 3.

Since the *MASM* semantics defined before in [4] is only applicable when the stack pointers have been set up correctly, we remedy this shortcoming by extending *MASM* semantics in a simple way suited specifically to the situation occuring in the baby hypervisor in Section 4. Having achieved full coverage of the baby hypervisor code with our stack-based semantics, we proceed by translating the *MASM* code portions to *C* code according to the Vx86-approach in Section 5 – using VCC to verify correctness of the translated code. We conclude with a brief discussion of the verification experience.

All details of theories presented in this paper can be found in [18].

2 Models of Computation

2.1 Macro Assembler – A Stack-Based Assembler Language

In [1], it is stated that the assembler code verification approach to be used for verification of the missing assembler portions should be the same that has already been used

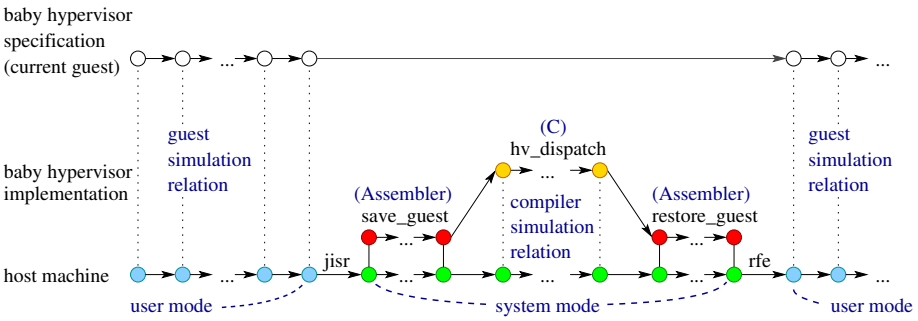

Fig. 3: Overview of baby hypervisor execution

in the Verisoft project [8, 15]: Correctness of assembler code execution is argued step-by-step on Instruction-Set-Architecture (ISA) level. Overall correctness in combination with the baby hypervisor C code is to be established by applying a compiler correctness specification that relates ISA and C configurations.

The assumption in [1] is that it would be quite simple to use the baby VAMP interpreter from [1] to verify the assembler code portions of the baby hypervisor by performing steps until the code has been executed. However, while this approach works decently in an interactive prover, this does not work so nicely in an automated prover. Running the baby VAMP interpreter for more than a few specific consecutive steps easily leads to huge verification times.

With the tool Vx86 [3] it has been demonstrated that VCC can efficiently be used to verify (isolated) non-interruptible x86 Microsoft Macro Assembler code – by translating Macro Assembler to C which is verified with VCC. Non-interruptability can safely be assumed for the baby hypervisor code, thus, we decided to follow this approach. In order to formally argue soundness, we need an assembler code execution model for which simulation with a C code execution model is simple to establish. In this light comes our custom high-level stack-based assembler language we call *Macro Assembler* in the following brief summary.

Macro Assembler is a language with the following features: Jumps are expressed as (different flavors of) gotos to locations in functions, function calls and return are always made with the *call* and *ret* macros, and the memory region that holds the stack is abstracted to an abstract stack component (a list of stack frames) on which all stack accesses are performed. The first two choices restrict the applicability of MASM-semantics to well-structured assembler code. For baby VAMP, the following instructions are implemented as macros: *call*, *ret*, *push*, *pop*. A macro is simply a shorthand for a sequence of assembler instructions.

Configuration A *Macro-Assembler* configuration

$$c = (c.\mathcal{M}, c.regs, c.s) \in conf_{MASM}$$

consists of a byte-addressable memory $\mathcal{M} : \mathbb{B}^{8k} \to \mathbb{B}^8$ (where k is the number of bytes in a machine word and $\mathbb{B} \equiv \{0, 1\}$), a component $regs : \mathcal{R} \to \mathbb{B}^{8k}$ that maps register

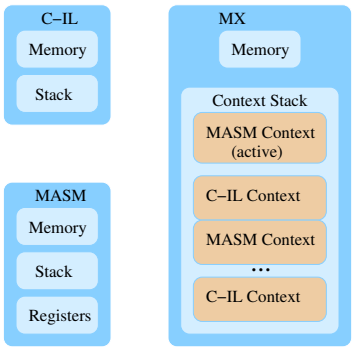

Fig. 4: Integrated semantics of C-IL and Macro Assembler with alternating execution contexts

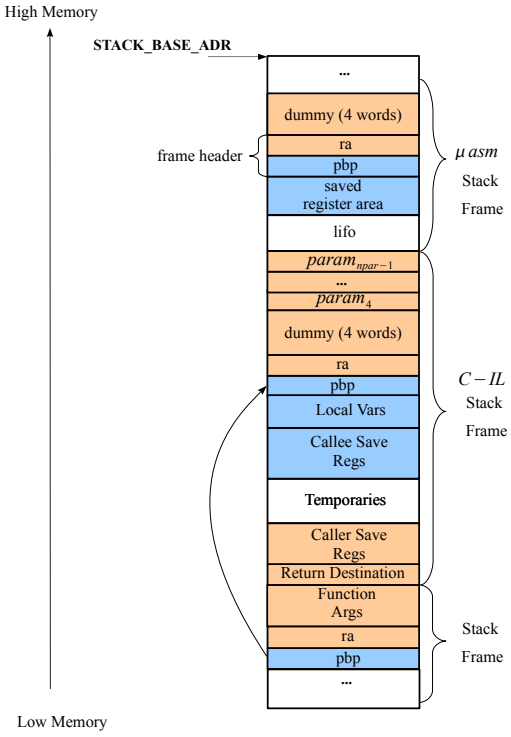

Fig. 5: *MX*-semantics stack layout

names to their values, and an abstract stack $s : frame^*_{MASM}$. Each frame

$$s[i] = (p, loc, saved, pars, lifo)$$

contains the name p of the assembler function we are executing in, the location *loc* of the next instruction to be executed in p's function body, a component *saved* that is used to store values of callee-save registers specified by a so-called *uses list* of the function, a component *pars* that represents the parameter region of the stack frame, and a component *lifo* that represents the part of the stack where data can be $push$ed and popped to/from. For a detailed description of *Macro Assembler* semantics, see [18].

2.2 Integrated Semantics – Merging C Intermediate Language and Macro Assembler

In our verification effort, we are particularly interested in the correct interaction between assembler and C at function call boundaries. (Note that in contrast to the Verisoft project, we do not have inline assembler code here, but assembler functions calling C

functions and vice versa.) Thus, we define an integrated semantics of a simple C intermediate language and *Macro Assembler* which allows function calls between those languages to occur according to the compiler's calling conventions.

In order to describe this integrated semantics, we first give a very short overview of the features of our C intermediate language *C-IL*. *C-IL* is a very simple language that only provides the following program statements: assignment, goto, if-not-goto, function call, procedure call, and corresponding return statements. Goto statements specify destination labels. Pointer arithmetic on local variables and the global memory is allowed. There is no inherent notion of a heap in *C-IL*.

C-IL Configuration A *C-IL* configuration

$$c = (\mathcal{M}, s) \in \mathit{conf}_{\mathit{C\text{-}IL}}$$

consists of a global, byte-addressable memory $\mathcal{M} : \mathbb{B}^{8k} \rightarrow \mathbb{B}^8$ and a stack $s \in \mathit{frame}^*_{\mathit{C\text{-}IL}}$ which is a list of *C-IL*-frames. Similar to *MASM*, a frame contains control information in form of location and function name – further, it contains a local memory that maps local variable and parameter names to their values as well as a return destination field for passing return parameters.

Integrated Semantics In [4], we provide a more detailed report on the integrated semantics – a defining feature of which is its call stack of alternating *C-IL* and *MASM* execution contexts (see Fig. 4). Exploiting that both semantics use the same byte-addressable memory, we obtain a joint semantics in a straightforward way by explicitly modeling the compiler calling conventions and by calling the remaining parts of a *C-IL*- or *MASM*-configuration an execution context for the respective language.

Configuration A mixed semantics (*MX*-) configuration

$$c = (\mathcal{M}, ac, sc) \in \mathit{conf}_{\mathit{MX}}$$

consists of a byte-addressable memory $\mathcal{M} : \mathbb{B}^{8k} \rightarrow \mathbb{B}$, an active execution context $ac \in \mathit{context}_{\mathit{C\text{-}IL}} \cup \mathit{context}_{\mathit{MASM}}$, and a list of inactive execution contexts $sc \in (\mathit{context}^{\mathit{inactive}}_{\mathit{C\text{-}IL}} \cup \mathit{context}^{\mathit{inactive}}_{\mathit{MASM}})^*$ which, in practice, is alternating between *C-IL* and *MASM*.

Here, an inactive execution context always contains information on the state of callee-save registers, which, before returning from *MASM* to *C-IL*, must be restored in order to guarantee that execution of compiled *C-IL* code will proceed correctly – or, respectively, the state of callee-save registers which will be restored automatically by the compiled *C-IL* code when returning from *C-IL* to *MASM*.

3 Compiler Correctness Specification

We assume a compiler correctness specification in the spirit of the C0 compiler [19] from the Verisoft project. We state a consistency relation that we expect to hold at certain points between a baby VAMP ISA- and a MX-computation (Figs. 6, 7).

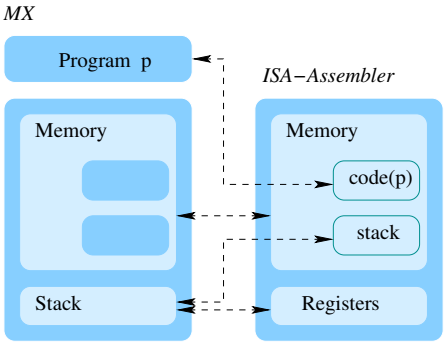

Fig. 6: Mapping abstract configuration to physical machine configuration by compiler consistency relation

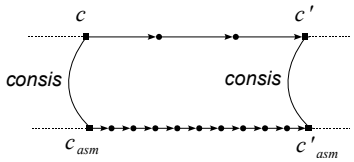

Fig. 7: Maintaining a compiler consistency relation between ISA- and MX-computation

Definition 1 (Code consistency). *The code region of the physical baby VAMP machine d is occupied by the compiled code of program p.*

$$consis_{code}(p,d) \equiv d.m_{\textbf{len}(code(p))}(code_{base}) = code(p)$$

where $code(p)$ denotes the compiled code of program p represented as a byte-string, **len** returns the length of such a string, $m_n(a)$ denotes reading a byte-string of length n starting at address a from byte-addressable memory m and $code_{base}$ denotes the address in memory where the code resides.

Definition 2 (Memory consistency). *The global memory content of the MX-machine c is equal to that of the physical baby VAMP machine d except for the stack and code region.*

$$consis_{mem}(c,d) \equiv \forall a \in \mathbb{B}^{32} \setminus code_{region} \setminus stack_{region} \quad : \quad c.\mathcal{M}(a) = d.m(a)$$

Definition 3 (Stack consistency). *The stack component of the MX-machine c is represented correctly in registers and stack region of the baby VAMP machine d.*

$$\begin{aligned} consis_{stack}(c,d,p) \equiv\ & d.m_{\textbf{len}(flatten_{stack}(c))}(\texttt{STACK\_BASE\_ADR}) = flatten_{stack}(c) \\ & \wedge consis_{regs}(c,d,p) \end{aligned}$$

where $flatten_{stack}$ denotes a function that, given a MX-configuration returns a list of bytes that represent the stack in the physical machine according to the compiler definitions and $consis_{regs}$ specifies that all machine registers have the values expected for the given abstract stack configuration of c. These can only be defined when additional information about the compiler is given: E.g. in order to compute the return address field of a stack frame, we need to know the address in the compiled code where execution must continue after the function call returns. The calling conventions detail where parameters are passed (e.g. in registers and on the stack), while the *C-IL*-compiler defines the order of local variables on the stack (and whether they are cached in registers for faster access). For an exemplary stack layout of our integrated semantics, see Fig. 5.

Definition 4 (Compiler consistency). *An MX-configuration c and a baby VAMP configuration d are considered to be consistent with respect to a program p iff code consistency, memory consistency and stack consistency are fulfilled.*

$$\mathit{consis}(c, d, p) \equiv \mathit{consis}_{\mathit{code}}(p, d) \wedge \mathit{consis}_{\mathit{mem}}(c, d) \wedge \mathit{consis}_{\mathit{stack}}(c, d, p)$$

Definition 5 (Optimizing compiler specification). *The compiler relates MX computations (c^i) and baby VAMP ISA computations (d^i) via two step functions $s, t : \mathbb{N} \to \mathbb{N}$ with the meaning that, for all i, MX-configuration $c^{s(i)}$ and ISA-configuration $d^{t(i)}$ are consistent*

$$\forall i : \mathit{consis}(c^{s(i)}, d^{t(i)})$$

in such a way that the step function $s(i)$ at least describes those states from the computation (c^i) which are about to perform an externally visible action or where such an action has been performed in the previous step. Further, both s and t are strictly monotonically increasing.

In our current sequential setting without devices, the only externally visible action possible is an external function call. From the viewpoint of *C-IL*-semantics, calls to *MASM* functions are external, and vice versa.

For a well-structured example of a simulation proof for compiler correctness of a multi-pass optimizing compiler, see [20]. In the compiler specification sketched in the thesis of A. Shadrin [18], we expect compiler consistency to hold additionally at the beginning and at the end of function bodies, which – while restricting the extent of compiler optimization – simplifies the inductive proof significantly.

4 Extending the Semantics for Stack Pointer Setup

While we saw that the formalism of *Macro Assembler* is nicely suited to serve as a basis for justification of a translation-based assembler verification approach, it became obvious that the restrictions of *Macro Assembler* as described so far prevents the use of *Macro Assembler* for some parts of the baby hypervisor code. In fact, *Macro Assembler* semantics is a sound abstraction for execution of all but the first 46 assembler instructions of the baby hypervisor code – the last two of those 46 set up the stack pointer registers in order to establish the stack abstraction of *Macro Assembler* (see Fig. 8). Before those two instructions that set up the stack pointers occur, the stack pointers are uninitialized and we cannot establish compiler consistency for the preceeding *Macro Assembler* execution.

The root of the problem In order to run the *MX*-machine (which makes use of a rather high-level stack abstraction), we need to establish the stack abstraction correctly on the physical machine in the first place. In order to apply our compiler correctness specification, we need to establish initial compiler consistency, of which stack consistency is one part. The first part of the baby hypervisor's interrupt handler implementation actually has to set up the stack abstraction for the baby hypervisor code to run by writing the

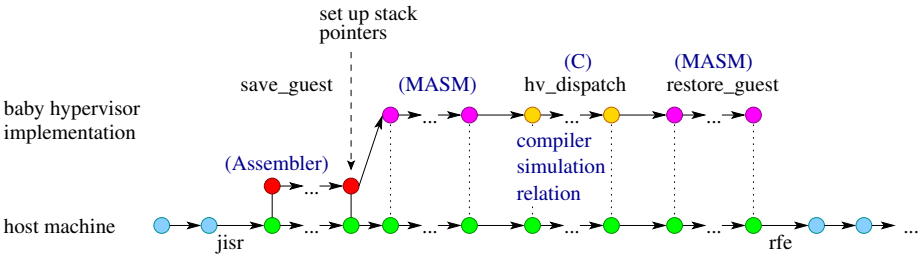

Fig. 8: The semantics stack applied to the baby hypervisor's interrupt service routine after introduction of the first *Macro Assembler* version

stack pointer registers of the baby VAMP machine (see Fig. 8). The stack abstraction of the original *Macro Assembler* semantics, however, abstracts the stack pointers away.

While we could have proven the correctness of the assembler instructions up to the initialization of stack pointers using the baby VAMP ISA model in an interactive prover (which would have been bearable after rewriting the assembler code to perform stack initialization much earlier), we instead looked at the problem from the other side: What changes do we need to make to *Macro Assembler* semantics in order to "lift" these instructions to the *Macro Assembler* formalism so that we can verify their correctness with VCC? Considering the baby hypervisor implementation closely, we are in a situation where the stack pointers are always set up in the same way: by writing the stack base address `STACK_BASE_ADR` of the baby hypervisor to both stack pointer registers, effectively resulting in an empty initial stack configuration (i.e. there are neither parameters nor saved register values nor temporaries that can be *pop*ped from the stack). This is the case since a baby hypervisor execution context is always created by an interrupt, then the baby hypervisor performs emulation of a guest step and then the execution context perishes by giving up control to the guest.

Our proposed solution For the situation in question, a simple band-aid is to just extend *Macro Assembler* semantics in such a way that there are two *execution modes*:

- *abstractStack*: The existing one with stack abstraction (stack pointer registers are hidden), and
- *noStack*: a mode without stack (stack pointer registers are accessible while function calls and stack operations are prohibited).

Defining the transitions between execution modes for this case is simple: When executing in *noStack*-mode, writing `STACK_BASE_ADR` to both stack pointer registers immediately results in an equivalent configuration in *abstractStack*-mode with empty stack content, whereas accessing the stack pointers in *abstractStack*-mode while the stack content is empty leads to an equivalent *noStack*-mode configuration (see Fig. 9).

With these definitions, we achieve full code coverage on the baby hypervisor with the resulting improved *MX*-semantics. While the chosen solution is rather specific, its simplicity raises the question whether there are more cases in which we can lift assembler instructions incompatible with stack abstraction to the *Macro Assembler*-level

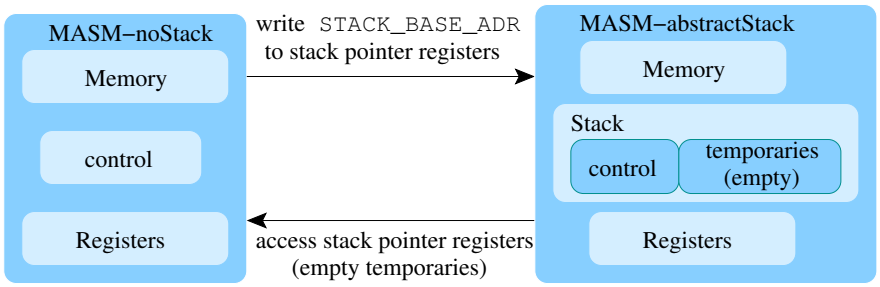

Fig. 9: Switching between mode without abstract stack and mode with abstract stack

under certain conditions. We think that the stack switch operation present in thread switch implementations (substituting the stack pointers of the physical machine with the stack pointers associated with the next thread to run) is such a candidate, which will enable the sound verification of the code of thread switch implementations using automated C verifiers.

5 Assembler Verification Approach

For the assembler code verification of the baby hypervisor, we follow the general idea used in Vx86 [3]: Assembler code is translated to C code which is verified using VCC. Since there is very little assembler code in the baby hypervisor, we do not implement a tool that performs the translation (e.g., like Vx86) – instead, we formally define the translation rules and translate the code by hand according to the rules. We state this translation using *Macro Assembler* and *C-IL* semantics in [18].

The translation, in general, works as follows: we model the complete *MASM* state in *C-IL* using global variables and translate each *MASM*-instruction to one or several *C-IL*-statements. Recall that a *MASM*-configuration contains three main parts: the byte-addressable memory, a register component, and a stack of *MASM*-frames which each consist of control information, *saved* registers, parameters, and a component *lifo* of temporaried pushed to the stack. The byte-addressable memory is represented by *C-IL*'s own byte-addressable memory. For register content, we introduce global variables `gpr` and `spr` as arrays of 32-bit integer values of appropriate size. Control information is translated implicitly, by preserving the structure of function calls and jumps of the *MASM*-program during the translation. We model each of the stack components *saved*, *pars* and *lifo* by a corresponding global 32-bit integer array variable and a 32-bit unsigned integer variable that counts the number of elements occupied in the array. For an example of representing *MASM*-state, see Fig. 10.

Translation We define a function $\tau_{MX2IL} : Prog_{MX} \rightarrow Prog_{C\text{-}IL}$ which, given a program π_{MX} of the integrated semantics returns a *C-IL* program $\pi_{C\text{-}IL} = \tau_{MX2IL}(\pi_{MX})$. *MASM*-functions are translated to *C-IL*-functions; every *MASM* instruction of π_{MX} is translated to one or several *C-IL* statements while *C-IL* statements of π_{MX} are simply preserved in $\pi_{C\text{-}IL}$.

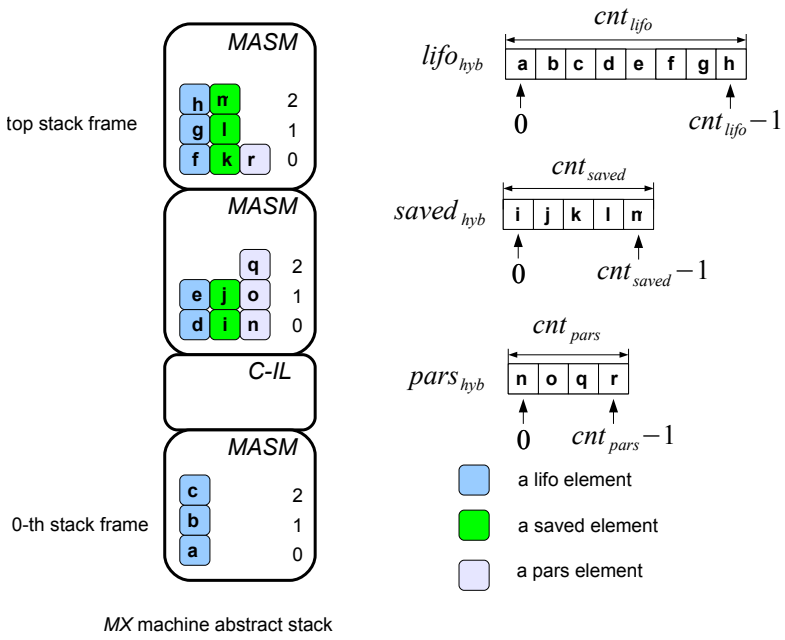

Fig. 10: Example of representing a *Macro Assembler* stack using C data structures

Consider the translation of the *push*-instruction:

$$\texttt{push}\, r \quad \Rightarrow \quad \mathit{lifo}_{\mathit{hyb}}\texttt{[}\mathit{cnt}_{\mathit{lifo}}\texttt{]} \;=\; \texttt{gpr[}r\texttt{];}\;\; \mathit{cnt}_{\mathit{lifo}} \;=\; \mathit{cnt}_{\mathit{lifo}} \;+\; 1$$

In *MASM* semantics, the *push*-instruction simply appends the value of register r to the *lifo*-component (which is modeled as a list in *MASM*-semantics and as an array with a counter in the translated program). Other examples are the translation of the *sw*-instruction or the *add*-instruction below:

$$\texttt{sw}\; rd\; rs_1\; imm \quad \Rightarrow \quad *((\texttt{int}*)(\texttt{gpr}[rs_1] + imm)) = \texttt{gpr}[rd]$$

$$\texttt{add}\; rd\; rs_1\; rs_2 \quad \Rightarrow \quad \texttt{gpr}[rd] = \texttt{gpr}[rs_1] + \texttt{gpr}[rs_2]]$$

Here, rd, rs_1, rs_2 are register indexes, and imm is a 16-bit immediate-constant – and `sw` is a *store word* instruction that stores the register content of register rd at offset imm of the memory address in register rs_1, while `add` is an instruction that adds the values of registers rs_1 and rs_2, storing the result in rd.

Soundness A soundness proof for this approach is given by proving a simulation between the *MX*-execution of the original program π_{MX} and the *C-IL* program $\pi_{C\text{-}IL}$ that results from the translation. The simulation relation and a pencil-and-paper proof are given in [18].

To achieve a clear separation between the original *C-IL*-code and the translated *MASM*-code, we place the data structures that model *MASM*-state at memory addresses which do not occur in the baby VAMP, i.e. addresses above 2^{32}. We call this memory region *hybrid memory*, since it is neither memory of the physical machine (which is covered by our compiler correctness specification) nor ghost state (VCC prevents information flow from ghost state to implementation state). Thus, the translation itself does not affect the execution of the original *C-IL* code parts. The actual proof is a case distinction on the step made in the *MX*-model: due to the way the translation is set up, correctness of pure C-IL steps is quite simple to show, while inter-language and pure MASM-steps have to explicitly preserve the simulation relation between the translated and the original program.

In order to formally transfer properties proven with VCC on the verified C code to the *MX*-execution we still lack a proof of property transfer from VCC C to C-IL. However, this gap should be straight-forward to close as soon as a formal model of VCC C is established.

The main advantage of how we implement this approach over how it has been done in the case of Vx86 is that we have a formal semantics for *Macro Assembler* that is quite similar to the C intermediate language (for which we also have formal semantics) we translate to. In [3], the closest formal basis for assembler code execution is given by the x86-64 instruction-set-architecture model developed by U. Degenbaev [21] – the authors apply the Microsoft Macro Assembler compiler to generate x86-64 code which is then translated to C. A monolithic simulation proof for this approach appears to be quite complex due to the big formal gap between the ISA model and the C semantics – thus, we deliberately chose *Macro Assembler* semantics as an abstraction of the ISA assembler code execution model in such a way that *Macro Assembler* semantics is structurally very similar to *C-IL* semantics.

6 Results & Future Work

Code Verification The practical part of this work extends the code verification of the baby hypervisor by proving the central interrupt service routine correct – following its implementation in *Macro Assembler* (consisting of 99 instructions, most of them memory accesses to store/restore register values to/from the corresponding PCB). Translation of the *MASM* code results in approximately 200 additional C code tokens – the remainder of the baby hypervisor implementation consists of about 2500 tokens. For verification, an additional number of 500 annotation tokens were needed. Originally, about 7700 annotation tokens were present. With the formal models we have now, we believe that the actual code verification effort comes down to about one person week. It is quite obvious, that, from a practical verification engineering point of view, completing the baby hypervisor code verification was a minor effort compared to what already had been done. Our main contribution is the justification of this code verification.

Using the *verification block* feature of VCC, it was possible to keep verification times quite low (e.g. 41,48 seconds for the `restore_guest` function, 75,68 seconds for the `save_guest` function). This feature allows to split the verification of large C functions into blocks with individual pre- and postconditions. The total proof checking

time of the completed baby hypervisor codebase is 4571 seconds (approx. $1\frac{1}{4}$ hours) on a single core of a 2.4 GHz Intel Core Duo machine.

Future work Possible extensions to this work include the generalization of calling conventions between *Macro Assembler* and *C-IL*. It appears desirable to have a semantic framework that can support many different compilers. For this, it could also be interesting to replace *C-IL* by a more mainstream intermediate language or a different flavor of C semantics. Similarly, *Macro Assembler* could be improved and generalized.

A work in progress deals with lifting the stack switch operation occuring in thread switch implementations to the *Macro Assembler* level. Extending the high-level semantics with a notion of active and inactive stacks (which is justified by a simulation with the ISA implementation layer), it should be possible to prove in an automated verifier that a given thread switch implementation based on switching stack pointers is correct.

7 Summary

In this work, we used an integrated semantics of *Macro Assembler*, a high-level assembler language, and *C-IL*, a simple C intermediate language, to give a pencil-and-paper justification of a translation-based assembler verification approach in the spirit of Vx86 [3]. In contrast to the original work, the translation is expressed rigorously based on formal semantics. We solved the problem of stack pointer setup by lifting a part of the ISA assembler semantics to our improved *Macro Assembler* semantics which we used as starting point for the translation of *Macro Assembler* code to *C-IL* code. The baby hypervisor implementation was completed by implementing the central interrupt service routine in *Macro Assembler* code, which was subsequently translated to C and verified with VCC – completing the formal verification of the baby hypervisor.

References

1. Alkassar, E., Hillebrand, M., Paul, W., Petrova, E.: Automated verification of a small hypervisor. In: Third International Conference on Verified Software: Theories, Tools, and Experiments (VSTTE'10). Volume 6217 of LNCS., Springer (2010) 40–54
2. 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 Berghofer, S., Nipkow, T., Urban, C., Wenzel, M., eds.: International Conference on Theorem Proving in Higher Order Logics (TPHOLs-09), August 17-20, Munich, Germany. Volume 5674 of LNCS, LNCS., Springer (2009) 23–42
3. Maus, S., Moskal, M., Schulte, W.: Vx86: x86 assembler simulated in C powered by automated theorem proving. In: AMAST. (2008) 284–298
4. Schmaltz, S., Shadrin, A.: Integrated semantics of intermediate-language C and macro-assembler for pervasive formal verification of operating systems and hypervisors from VerisoftXT. In Joshi, R., Müller, P., Podelski, A., eds.: Verified Software: Theories, Tools, Experiments. Volume 7152 of LNCS. Springer Berlin / Heidelberg (2012) 18–33
5. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: Formal verification of an OS kernel. In: Proc. 22nd ACM Symposium on Operating Systems Principles (SOSP), Big Sky, MT, USA, ACM (2009) 207–220

6. Bevier, W.R.: Kit and the Short Stack. J. Autom. Reasoning **5**(4) (1989) 519–530
7. Feng, X., Shao, Z., Dong, Y., Guo, Y.: Certifying low-level programs with hardware interrupts and preemptive threads. In: 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08), New York, NY, USA, ACM (2008)
8. Verisoft Consortium: The Verisoft Project. (`http://www.verisoft.de/`)
9. Alkassar, E., Paul, W.J., Starostin, A., Tsyban, A.: Pervasive verification of an OS microkernel: inline assembly, memory consumption, concurrent devices. In: Proceedings of the Third international conference on Verified software: theories, tools, experiments. VSTTE'10, Berlin, Heidelberg, Springer-Verlag (2010) 71–85
10. Gargano, M., Hillebrand, M., Leinenbach, D., Paul, W.: On the correctness of operating system kernels. In Hurd, J., Melham, T.F., eds.: 18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2005). Volume 3603., Springer (2005) 1–16
11. Tews, H., Weber, T., Völp, M., Poll, E., Eekelen, M., Rossum, P.: Nova micro-hypervisor verification formal, machine-checked verification of one module of the kernel source code (Robin deliverable d.13). (`http://robin.tudos.org/`) (2008)
12. Dahlin, M., Johnson, R., Krug, R.B., McCoyd, M., Young, W.D.: Toward the verification of a simple hypervisor. In Hardin, D., Schmaltz, J., eds.: ACL2. Volume 70 of EPTCS. (2011)
13. Leinenbach, D., Santen, T.: Verifying the Microsoft Hyper-V hypervisor with VCC. In: 16th International Symposium on Formal Methods (FM 2009). Volume 5850 of LNCS., Eindhoven, the Netherlands, Springer (2009) 806–809
14. Tverdyshev, S.: Formal Verification of Gate-Level Computer Systems. PhD thesis, Saarland University, Computer Science Department (2009)
15. Tsyban, A.: Formal Verification of a Framework for Microkernel Programmes. PhD thesis, Saarland University, Computer Science Department (2009)
16. Alkassar, E., Cohen, E., Hillebrand, M., Kovalev, M., Paul, W.: Verifying shadow page table algorithms. In: Formal Methods in Computer Aided Design (FMCAD) 2010, Lugano, Switzerland, IEEE (2010) 267–270
17. Maus, S.: Verification of Hypervisor Subroutines written in Assembler. PhD thesis, Freiburg University, Computer Science Department (2011)
18. Shadrin, A.: Mixed Low- and High Level Programming Language Semantics and Automated Verification of a Small Hypervisor. PhD thesis, Saarland University, Computer Science Department (to appear 2012)
19. Leinenbach, D.: Compiler Verification in the Context of Pervasive System Verification. PhD thesis, Saarland University, Computer Science Department (2007)
20. Leroy, X.: Formal verification of a realistic compiler. Communications of the ACM **52**(7) (2009) 107–115
21. Degenbaev, U.: Formal Specification of the x86 Instruction Set Architecture. PhD thesis, Saarland University, Computer Science Department (2011)