Pervasive Verification of an OS Microkernel

Inline Assembly, Memory Consumption, Concurrent Devices

Eyad Alkassar[⋆,⋆⋆⋆], Wolfgang J. Paul,
Artem Starostin[⋆⋆,⋆⋆⋆], and Alexandra Tsyban[⋆⋆⋆]

Computer Science Department - Saarland University
{eyad,wjp,starostin,azul}@wjpserver.cs.uni-saarland.de

Abstract. We report on the first formal *pervasive* verification of an operating system microkernel featuring the correctness of inline assembly, large non-trivial C portions, and concurrent devices in a single seamless formal proof. We integrated all relevant verification results we had achieved so far [21,20,2,5,4] into a single top-level theorem of microkernel correctness. This theorem states the simulation of user processes with own, separate virtual memories — via the microkernel — by the underlying hardware with devices. All models, theorems, and proofs are formalized in the interactive proof system Isabelle/HOL.

1 Introduction

Pervasive Verification: Why Bother? A program proven correct in a high-level programming language may not execute as expected on a particular computer. Such correctness proof ignores irregular patterns of control flow which take place due to multitasking and interrupts on the computer. High-level data types and operations used to implement the program and formulate its correctness criteria differ from flip-flops and signals that occur in the hardware. The gap between what has been proven about the program in the high-level language semantics and what is actually executed on the underlying hardware may be a source of errors. The solution to the problem is to verify the execution environment of the program: the operating system to ensure correct assignment of hardware resources to the program and non-interference with other programs, the compiler and assembler to guarantee correct translation from high-level data types and operations to the machine instruction level, the actual hardware implementation to make certain that it meets the instruction set architecture. This is known as *pervasive* verification [18].

Pervasive verification of complete computer systems stacks from the gate-level hardware implementation up to the application level is the aim of the

⋆ Work was supported by the German Research Foundation (DFG) within the program 'Performance Guarantees for Computer Systems'.

⋆⋆ Work was supported by the International Max Planck Research School for Computer Science (IMPRS-CS).

⋆⋆⋆ Work was supported by the German Federal Ministry of Education and Research (BMBF) within the Verisoft project under grant 01 IS C38.

G.T. Leavens, P. O'Hearn, and S.K. Rajamani (Eds.): VSTTE 2010, LNCS 6217, pp. 71–85, 2010.

German Verisoft project[1]. The context of the current work is 'academic system', a subproject of Verisoft, which covers, among others, a processor with devices, a microkernel, and an operating system.

In order to ensure that interfaces of all components of the program's execution environment fit together a common formal framework — like Isabelle/HOL [23] in our project — has to be used. By choosing the implementation model of each layer to be the specification of the next lower layer it is possible to combine the components into a verified stack. With the program on top of the stack one achieves the *highest* degree of assurance in program correctness.

The Challenges in Pervasive Verification of an OS Microkernel. It is fair to put an operating system microkernel at the heart of a hardware-software stack. By design, a microkernel inevitably features (i) inline assembly portions — to access resources beyond the visibility of C variables, e.g., hardware registers, (ii) large sequential C parts — to implement resource management policies, e.g., user process scheduling, and (iii) interleaving communications of the processor with devices — to support, e.g., demand paging. Hence, the task of formal pervasive verification of a microkernel requires a feasible technology for efficient reasoning — in a single proof context — about the aforementioned features. In our experience, the complexity of the problem turns out to be not in verification of individual components comprised by a system, but rather in formal integration of different correctness results achieved separately. For instance, integration of functional correctness properties of sequential C code into an interleaved hardware computation requires additional reasoning about the memory consumption.

Contributions. This paper gives a bird's eye view on the first pervasive correctness proof of an operating system microkernel including such challenging components as demand paging, devices communications, and process-context switch. We report on the top-level correctness theorem and its completed (modulo symmetric cases) formal proof. This proof has motivated development of formal theories to reason, among others, about inline assembly, memory consumption, and concurrent devices.

Related Work. As Klein's article [16] provides an excellent and comprehensive overview of the history and current state of the art in operating systems verification we limit this paragraph to highlight the peculiarities of our work. We extend the seminal work on the CLI stack [6] by integrating devices into our model and targeting a more realistic system architecture regarding both hard and software. The project L4.verified [11] focuses on the verification of an efficient microkernel, rather than on formal pervasiveness, as no compiler correctness or an accurate device interaction is considered. The project produced a 200k-line formal correctness proof of the seL4 microkernel implementation. In the FLINT project, an assembly code verification framework is developed and a preemptive thread implementation together with synchronization primitives on a x86 architecture were formally proven correct [13]. A program logic for assembly

[1] `www.verisoft.de`

code as well as techniques for combining domain-specific and foundational logics are presented [12], but no integration of results into high-level programming languages is undertaken. The relevant references to our own previous work covering single pieces of the overall correctness puzzle are as follows: [21] reports on process-context switch verification, [20] describes the correctness of microkernel primitives, [2] elaborates on interleaved driver verification, and [5] shows verification of a page-fault handler. The semantic stack used to verify the microkernel is covered in large detail in [4]. The current paper reports for the first time on integrating all mentioned previous results into a single formal top-level correctness theorem of a microkernel.

Outline. In Sect. 2 we introduce CVM, our programming model for microkernels. Sect. 3 states its top-level correctness theorem. Sect. 4 introduces a semantics stack used for the theorem's proof outlined in Sect. 5. We conclude in Sect. 6.

2 CVM Programming Model

We discuss operating system microkernels built following the model of communicating virtual machines (CVM) [14], a programming model for concurrent user processes interacting with a microkernel and devices. The purpose of this model is to provide to the programmer of a microkernel a layer implementing separate virtual user processes.

CVM is implemented in C with inline assembly as a framework [15,22] featuring isolated processes, virtual memory, demand paging [5,19], and low-level inter-process and devices communications [2,1]. Most of these features are implemented in the form of so called microkernel primitives [20]. Primitives are functions with inline assembly parts realizing basic operations which constitute the kernel's functionality. The framework can be linked on the source code level with an abstract kernel, an interface to users, in order to obtain a concrete kernel, a program that can be translated and run on a target machine, the VAMP processor [7] with devices in our case. As CVM is only parametrized with an abstract kernel, its computations do not depend on particular shapes of abstract kernels. Two different abstract kernels were used in Verisoft: a general purpose microkernel VAMOS [9] and an OSEKtime-like microkernel OLOS [10].

Specification. The state space of the CVM comprises components for (i) user processes, (ii) the abstract kernel, (iii) devices, (iv) shared interrupt mask, and (v) the current process identifier. User processes are modeled as a vector of separate VAMP assembly machines (cf. Sect. 4) with own, large virtual memories. The abstract kernel is a pure C program with no inline assembly portions. Its computations are modeled by the small-step semantics of C0 (cf. Sect. 4), a C-dialect used in Verisoft [17]. Besides ordinary C0 functions the abstract kernel can call a number of special functions, called CVM primitives [20]. These functions have no implementation within the abstract kernel, and are therefore called externally. CVM primitives can alter states of user processes and devices and implement basic means needed for a microkernel programmer: copy data

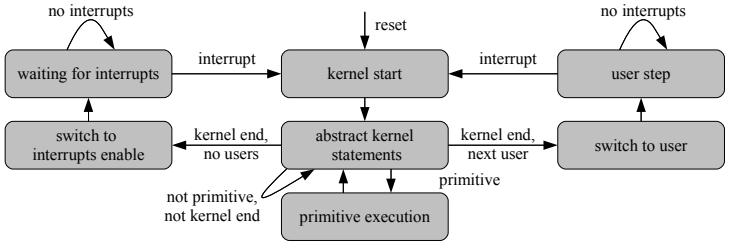

Fig. 1. States and transitions of CVM model

between processes, manage size of virtual memory given to processes, send data to devices, etc. Devices are modeled as deterministic transition systems communicating with an external environment and the processor via a specified memory interface. The external environment is used to model non-determinism and communication.

The transition function of the CVM model (cf. Fig. 1) distinguishes, therefore, three top-level cases of an execution corresponding to the mentioned CVM components: an *user step*, a *kernel step*, and a *devices step*.

A user step distinguishes three cases: (i) an uninterrupted step, (ii) an interrupted step with an abort of user execution, and (iii) a step with interrupt which nevertheless allows us to perform a step of the user machine before interrupt handling. In the first case the step boils down to an update of the user process configuration according to the VAMP assembly semantics. In case an interrupt occurs during the user step, the user has to be suspended and the kernel's interrupt handling routine has to be invoked. The actions taken in the third case are simply a composition of the first and the second case.

Kernel steps come in three flavors: (i) the kernel stays in the idle 'wait' state, (ii) the kernel finishes its execution by switching to the idle state or to a user process, and (iii) the kernel performs a step of the abstract kernel component. The last step distinguishes between an ordinary C0 small-step semantics step of the abstract kernel and a primitive execution. For the case of a primitive invocation the CVM transition function defines which effect the primitive has on the user processes and/or the kernel.

A devices step boils down to an external step of the specified device. The effect of a device step is to update the devices component of the CVM model.

Target Hardware Platform. The main purpose of a microkernel is to provide multiple users access to shared computational resources like physical memory and devices. Therefore, a particular target hardware model has to be considered while reasoning about microkernel correctness. We use the VAMP [7] with devices hardware platform to run the compiled microkernel.

The VAMP instruction set architecture is a sequential specification of the VAMP gate-level hardware. The model is defined by a transition function over the states which comprise bit-vector representations of components for the

program counters, general and special purpose registers, and memory. The model features two execution modes, system and user, and address translation in user mode. VAMP ISA computations could be broken by interrupt signals, either internal, or external. In this case the execution is switched to the system mode, and the program counters are set to the start address of the compiled kernel.

Below, we briefly highlight how this hardware platform allows us to implement some fundamental features of a microkernel. Physical memory sharing is realized in CVM by memory virtualization: the kernel ensures that each user process has a notion of its own large address space. User processes access memory by virtual addresses which are translated to physical ones by a memory management unit [8] on the hardware side, or by the kernel on the software. We allow address spaces of user processes to exceed real memory of the physical hardware. This feature is supported by means of demand paging [5]: we partition available physical memory into small consecutive portions of data, called pages, which are stored either in fast but strongly limited in size *physical memory*, or in large but slower auxiliary memory, called *swap memory*. We store the swap memory on a hard disk [2]. The address translation algorithm of VAMP can determine where a certain page lies. In case the desired pages resides in the physical memory the kernel can provide an immediate access. Otherwise, the page is on the hard disk. The processor signals it by raising a data or instruction page-fault interrupt. The kernel's page-fault handler reacts to this interrupt by transferring the page from the hard disk to the main memory.

Communication of user processes with devices is supported by memory-mapped devices of the VAMP with devices. Devices are modeled as deterministic transition systems communicating with an external environment and the processor. The processor accesses a device by reading or writing special addresses. The devices, in turn, can signal interrupts to the processor. Interaction with the external environment is modeled by non-deterministic input/output. Several devices and a processor model could be coupled into a combined system which interleaves devices and processor steps. We refer to this system as VAMP ISA with devices. Computations of this system are guided by an external oracle, called an execution sequence, which defines for each point of time which of the computational sources, either the processor or some device, makes a step.

Implementation. CVM provides a microkernel architecture consisting of two layers. The general idea behind this layering is to separate a kernel into two parts: the abstract kernel that can be purely implemented in a high-level programming language, and the framework that inevitably contains inline assembly code because it provides operations which access hardware registers, devices, etc. The CVM framework is implemented with approximately 1500 lines of code from which 20% constitute inline assembly code. The implementation contains the following routines.

Process-context switch procedures `init_()` and `cvm_start()` are used for saving and restoring of contexts of user processes, respectively. The function `init_()` distinguishes a reset and non-reset cases. The former occurs after the power was switched on on the VAMP processor — the kernel memory structure is created.

In a non-reset case the procedure saves by means of inline assembly code the content of hardware registers into a special kernel data structure and invokes the elementary dispatcher of the CVM framework. The `cvm_start()` procedure is basically an inverse of the context save in a non-reset case. Context-switch procedures are almost fully implemented in inline assembly.

The page-fault handler of CVM `pfh_touch_addr()` features all operations on handling page faults, software address translation, and guaranteeing for a certain page to reside in the main memory for a specified period. The necessarily needed assembly code for talking to the hard disk is isolated in elementary hard-disk drivers `write_to_disk()` and `read_from_disk()`.

The function `dispatcher()` is an elementary dispatcher of the CVM framework. It handles possible page faults by invoking `pfh_touch_addr()` and calls `dispatcher_kernel()`, the dispatcher of the abstract kernel. This dispatcher returns to the CVM framework an identifier of the next-scheduled process or a special value in case there is no active processes. In the former case the elementary dispatcher starts the scheduled process by means of `cvm_start()`. In the latter case `cvm_wait()` is called which implements the kernel idle state.

The remaining part of the CVM framework contains 14 primitives for different operations for user processes.

3 Verification Objective

The CVM verification objective is to justify correctness of (pseudo-)parallel executions of user processes and the kernel on the underlying hardware (cf. Fig. 2). This is expressed as a simulation theorem between the VAMP ISA with devices and virtual machines interleaving with the kernel. States of the CVM and VAMP ISA with devices models are coupled by a simulation relation which is a conjunction of the claims like (i) the kernel relation which defines how an abstract kernel is related to the concrete one and — through the C0 compiler correctness

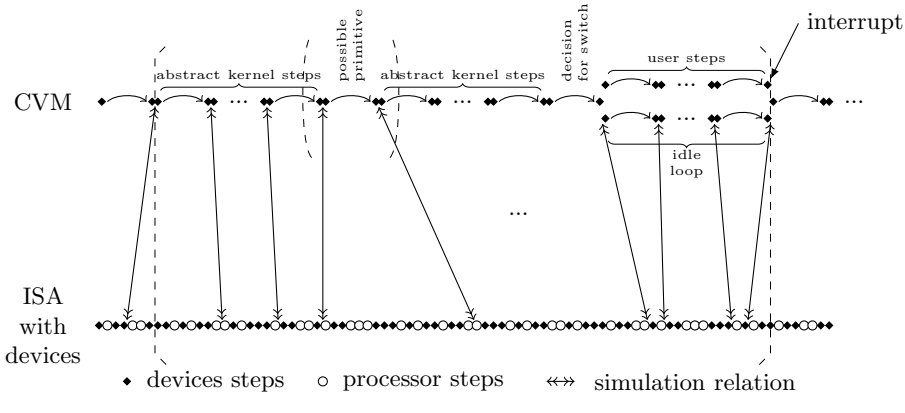

Fig. 2. Different cases of the CVM correctness theorem

statement [17] — how the latter is mapped to the VAMP ISA machine, (ii) the devices relation which claims that devices in CVM and hardware configurations are equal except for the swap hard disk which is invisible in the CVM model, (iii) the relation for user processes which states that the user process configurations are encoded in the configuration of the hardware with devices model. Thus, the top-level correctness theorem of the CVM can be stated as follows.

Theorem 1 (CVM correctness). *Given an execution of the VAMP ISA with devices model we can construct an execution of the CVM model such that the simulation relation between both holds after each CVM step.*

To prove that the CVM abstraction relation holds throughout CVM executions a number of invariants over the CVM implementation as well as the underlying hardware model have to hold. The reader can find complete definitions of the simulation relation and invariants in [22].

4 Semantics Stack

Ultimately, the right level to express overall correctness of system software, as the CVM kernel, is VAMP ISA with devices; only there all relevant components, as for example the mode register, become visible. Still, conducting all the code verification (or even the implementation) at this level seems to be infeasible. Rather, we introduced a pervasive semantics stack (depicted in Fig. 3), reaching from the high-level programming language C0, down to VAMP ISA with devices. On the one hand this semantics stack should provide for each single verification target with the most adequate reasoning environment. On the other hand the stack must be sound, i.e. allowing to integrate results of different levels into a single proof and finally propagate correctness to the low-level hardware model.

The overwhelming part of the microkernel is written in the language C0. C0 was designed as a subset of C which is expressive enough to allow implementations of all encountered system code in the Verisoft project, while remaining handy enough for verification. Therefore, we restricted ourselves to a type-safe fragment of C, without pointer arithmetic. However in the context of system-code verification we also have to deal with portions of inline assembly code that break the abstraction of structured C0 programs: low-level hardware intrinsics as processor registers, explicit memory model and devices become visible.

The semantics stack of the Verisoft project comprises three flavors of C0 reasoning [4]: Hoare logic, small-step semantics and an intermediate big-step semantics. The Hoare logic provides sufficient means to reason about pre- and postconditions of sequential, type-safe, and assembly-free C0 programs. In contrast to small-step semantics, the Hoare logic features split heap, compound variables, and implicit typing. The heap model we use excludes explicit address arithmetic but it is capable to efficiently represent heap structures like lists.

Compiler correctness allows to transfer properties proven in the C0 small step semantics to the so called VAMP Assembly with devices model [17]. While attempting to show correctness of assembly portions we have concluded that

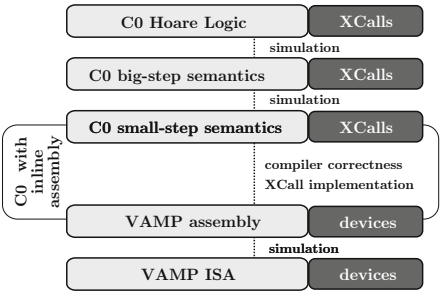

Fig. 3. Semantics stack

reasoning about the code in the VAMP ISA semantics is unnecessarily hard for a number of reasons like bit-vector representation of operands and presence of unwanted interrupts. As a response to this issue we introduced a convenient abstraction, the VAMP assembly model [22].

Having in mind that devices are executed in parallel with the processor, and that computations of the processor may be interrupted, only a C0 small-step semantics is adequate for verifying drivers and interleaved applications. Still, conducting all the code verification at the level of small step semantics or even below is not intended. Otherwise, one would abdicate the whole power of Hoare logic and the corresponding verification condition generator. The solution is to abstract low-level components by an extended state and to encapsulate the effects of inline assembly code by so called XCalls, which are atomic specifications manipulating both the extended state and the original C0 machine. First, by enriching the semantics stack with XCalls, we lift assembly code and driver semantics into Hoare logic. Then, by proving implementation correctness of XCalls we transfer results proven in Hoare logic down to VAMP assembly with devices [1].

All verification levels are glued together by respective simulation theorems [3,4]. This gives us freedom to choose the most efficient level of verification for each individual parts of the kernel and subsequently — via the simulation theorems — combine the results at the lowest level, the VAMP ISA.

Next we will describe in more detail three important extensions to the basic semantics stack which enable us to reason on system software correctness.

Concurrent Devices and Reordering. Device drivers are often an integral part of operating system kernels. For instance, since CVM features demand paging it needs correctly implemented hard-disk drivers. Hence, any approach to pervasive verification of operating system kernels should deal with driver correctness. Nonetheless, when proving functional driver correctness it does not suffice to reason only about code running on a processor. Devices themselves and their interaction with the processor also have to be formalized.

Obviously, when proving correctness of a concrete driver, an interleaved semantics of all devices is extremely cumbersome. Integration of results into traditional Hoare logic proofs also becomes hardly manageable. Preferably, we would

like to maintain a sequential programming model or at least, only bother with interleaved steps of those devices controlled by the driver we attempt to verify. A basic observation of our overall model is that device and processor steps that do not interfere with each other can be swapped. For a processor and a device step, this is the case if the processor does not access the device and the device does not cause an interrupt. Similarly, we can swap steps of devices not communicating with each other. Utilizing this observation we reorder execution sequences into parts where the processor accesses no device or only one device. All interleaved and non-interfering device steps are moved to the end of the considered part and hence a (partially) sequential programming model is obtained.

Note that compiled, pure C0 programs never access devices, because data and code segments must not overlap with device addresses. Hence, all interleaved device steps can be delayed until some inline assembler statement is encountered. More generally, the execution of drivers controlling different (non-interfering) devices can also be separated, enabling modular verification of device drivers [1].

Inline Assembly and XCalls. The simulation theorems described in Sect. 4 allow us to transfer program properties from the Hoare logic down to the assembly level. Recall that CVM contains large chunks of C code involving heap data structures and, at the same time, rare calls to functions with inline assembly. It is highly desirable to verify these parts in the Hoare logics, however inline assembly portions break the abstraction of structured C0 programs: low-level entities (like the state of a device) may become visible even in the specification of code that is only a client to the inline assembly parts. To avoid doing all the verification in the lower semantic levels we extend the Hoare logic to represent the low-level actions on an abstract extension of the state space by the concept of XCalls.

XCalls capture the semantical effects of function calls by atomic specifications. Particularly, when specifying functions with inline assembly portions, as for example drivers, the use of XCalls is appealing. First, we extend the C0 configuration by additional *meta variables*, representing those parts of the processor which are accessed by the assembly code. More general, these ghost variables — in the following called *extended state* — may abstract from arbitrary low-level entities which lie outside the scope of C0, e.g. memory, registers or even device states. An XCall describes the effect of a function call on the C0 configuration and on the extended state by one atomic state update. Compiler correctness remains applicable only in case implementation correctness proofs are provided for each of the XCalls.

The main charm of XCalls is that they enable us to argue on effects of inline assembly portions without caring about assembly semantics. Thus, by enriching the semantics stack with XCalls, we can lift assembly code and driver semantics up to the Hoare logic level. Then, by proving implementation correctness of XCalls we transfer results proven in Hoare logic down to VAMP assembly with devices. Note, that for drivers XCalls abstract interleaved executions to sequential atomic specifications. This is justified by the reordering theory [5,1].

Memory Consumption. Any code verification effort claiming the label *pervasive* has to deal with the concrete memory consumption of the given program and with memory restrictions of the target machine. The compiler correctness theorem, for example, is only applicable if in each step sufficient heap and stack memory is available in the assembly machine. Often such assumptions are silently ignored because they are not visible in the semantics of the given high-level language. As in the C0 small-step semantics, those models assume some infinite memory. Memory restrictions do not emerge until results are propagated down to lower levels, as in the case of the driver correctness.

Conditions on memory consumption should be formalized and verified in a modular way, i.e., in form of function contracts which do not depend on the invocation context. Moreover, they should be discharged for the high-level programming language, rather than at the level of the target machine.

For the kernel verification we have applied and verified the soundness of two different approaches to deal with memory restrictions:

- Static (syntactical) program code analysis. An upper bound of the stack consumption of functions with no recursive calls and no pointers to functions can be computed by a static analysis of the program. In short, this approach determines the deepest path (in terms of stack consumption) in the invocation tree of the given code. The soundness of this approach is established by verifying that executing the analyzed function (in any context) will never consume more stack memory than the computed upper bound.
- Extending the programming logic. In this approach we extend the Hoare logic by a ghost variable, which keeps track of the so far consumed heap memory. Moreover we have to adapt the inference rules for memory allocation to check and update the meta-variable. The soundness proof of this approach is part of the soundness proof of the transfer theorems from Hoare logic to C0 small step semantics. A similar approach could also be used to compute the stack memory consumption of a program. Note, that this methodology exploits our knowledge about the C0 compiler, in particular the sizes of its types.

5 Verifying CVM

The proof of the CVM correctness theorem (Theorem 1) is split according to various types of steps that can be made within the model (cf. Fig. 1). The statement of this theorem is formulated for all possible interleavings of the VAMP hardware with devices. Many steps of the hardware might be mapped to a single step of the CVM model (or even be invisible in it). The execution sequence of the CVM model must be constructed from the hardware one taking care about points where the processors detects interrupts. The theorem is proven by induction on the number of processor steps in the CVM execution sequence. The construction of this sequence is the starting point in proofs of all cases of the CVM correctness theorem. We partition all possible cases from Fig. 1 into two groups on verification of which we elaborate below: the kernel step and the

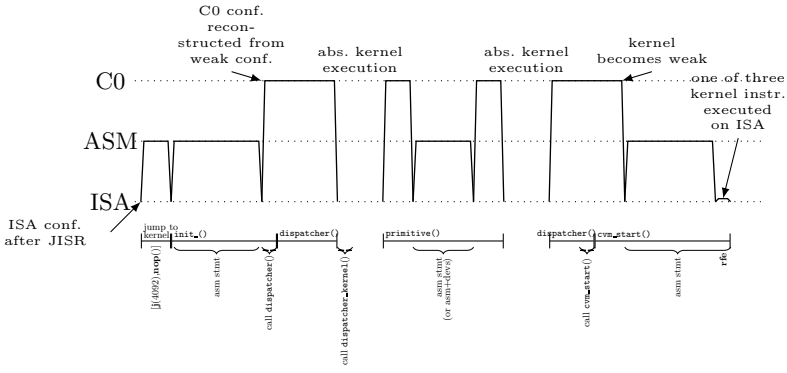

Fig. 4. Verification of a kernel step

user step. For each of them we examine the relevant code parts and show their functional correctness. From that the top-level simulation relation is inferred.

Kernel Step. The kernel step corresponds to such parts of the CVM implementation as (i) kernel initialization (after reset), (ii) process-context switch: switch to user and kernel initialization after user interrupt, (iii) primitives execution, (iv) waiting for interrupts, and (v) abstract kernel step.

Since this part of the CVM implementation features — beside the C code — many inline assembly portions, its verification proceed in the C0 small-step semantics and VAMP assembly semantics. Moreover, reasoning about instructions which switch process execution modes involves even the VAMP ISA semantics because the VAMP assembly model by design lacks support of modes. For each part of the code we would like to do the formal verification on the highest possible level of abstraction. Fig. 4–5 reflect this approach and depict correspondence between the CVM implementation parts and the semantics of verification.

The kernel step (cf. Fig. 4) starts right after the processor detects an interrupt and the JISR (jump to interrupt service routine) signal is activated. The latter sets the program counters to the start address of the *zero page* which, essentially, is used as a draft to store intermediate results of assembly computations. The first two instructions of the zero page, however, implement a jump to the kernel and are verified in the VAMP assembly semantics. The target of this jump is the function `init_()` which is responsible for the process-context save as well as kernel initialization and is implemented in inline assembly. The last statement of `init_()` is a C-call to the CVM's elementary dispatcher. We verify this call and the body of the dispatcher in C0 small-step semantics. In order to proceed with that we have to reconstruct the C0 configuration from the state of VAMP assembly. For that we maintain an invariant which states that the parts of C0 configuration — called the *weak* C0 configuration— are permanently encoded in the VAMP assembly memory. This technique is described in detail in [21]. Next, the CVM's dispatcher invokes the abstract kernel whereas the latter might

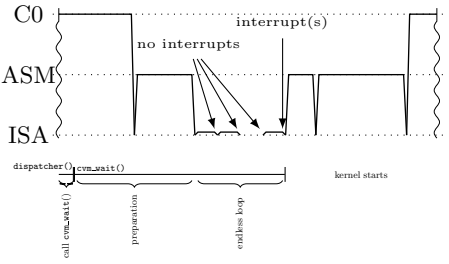

Fig. 5. Verification of the 'wait' case (idle loop)

invoke some CVM primitive. Primitives are verified [20] on C and assembly levels as they access user memory regions and devices. Further verification of the kernel step is split depending on whether there is at least one user process which has to be resumed. In case there is one, the verification of the remaining part is symmetrical: correctness proof of a C call to the process-context restore `cvm_start()` is followed by reasoning in assembly semantics about its body. The last instruction of the kernel step chain is the 'rfe' (return from exception) which switches the processor mode to user, and, therefore, has effects defined only in the VAMP ISA semantics. In case there is no user processes to be resumed the kernel goes to an idle loop by calling `cvm_wait()` (cf. Fig. 5). Here we are pending interrupts and, therefore, reason on the VAMP ISA level.

User step. User processes are modeled as virtual assembly machines with an illusion of their own, large, and isolated memory. Memory virtualization is transparent to user processes: within the CVM model page faults that might occur during a user step are handled silently by the low-level kernel functionality such that user can continue its run. During a single user step up to two page faults might occur: instruction page fault (*ipf*) and data page fault (*dpf*). The former might happen on instruction fetch whereas the latter could take place if we execute a memory operation. Our page-fault handler is designed in a way that it guarantees that no more than two page faults occur while processing a single instruction. The following five situations are possible regarding page faults: (i) there are no page faults, (ii) there is only an instruction page fault, (iii) there is only a data page fault, (iv) there is an instruction page fault followed by a data page fault, and (v) there is a data page fault followed by an instruction page fault. Fig. 6 depicts verification scheme for these cases. Essentially, the proof of the user step correctness boils down to a multiple application of the page-fault handler correctness theorem [5,19].

Page-Fault Handler. The page-fault handler is one of the most involved code portions of the CVM implementation. It maintains doubly-linked lists on the heap to implement the user virtual memory management policy and at the same time calls assembly subroutines implementing the hard disk driver. We verify the

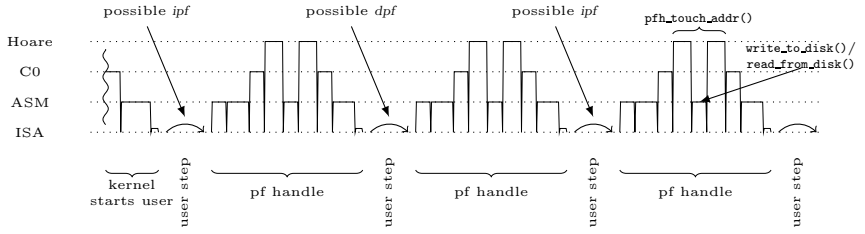

Fig. 6. Verification of a user step

page-fault handler in the Hoare logic and then transfer its functional correctness properties down to the level of VAMP ISA with devices semantics. The semantics of the assembly implemented hard disk drivers is encapsulated in XCalls. While justifying the correctness of XCalls implementation we have to deal with such peculiarities as devices steps reordering and estimating the memory consumption (cf. Sect. 4).

Besides loading the missing pages to the physical memory the page-fault handler servers as a single entry point for the software emulation of address translation in the kernel. In order to show the correctness of the latter the page-fault handler maintains a solid number of validity invariants over the page table and page management lists. One example of such invariants is that the page table entries always point outside the kernel region. This invariant turns out to be crucial for tackling the following problem.

Dealing with Self-Modifying Code. Suppose, we want to run the microkernel on a processor supporting self-modifying code: we can write at the address we have already fetched from. In system mode this peculiarity is resolved by the hardware. In user mode, however, the problem is affected by the address translation since now during the fetch we read not only at the fetch address but also the page tables. In this scenario we have to guarantee that user processes do not write page tables. By exploiting the aforementioned page-fault handler validity invariant we conclude that the translated address always lie in the user range.

6 Conclusion

We reported on the first formal and pervasive verification of a microkernel. Claiming pervasiveness means that all results can be (soundly) translated into and expressed at the low-level hardware model. We were forced to reason about many aspects and conditions of the system which are usually under the veil of 'technically but simple'. They don't show up until pervasive and formal verification is conducted. These conditions may be crucial for system correctness, as we illustrated for memory consumption. We are confident that the verification methods introduced to deal with inline assembly code, device drivers, and

memory restrictions, within the framework of a high-level programming logic as Hoare logic, can be applied to other, more complex system verification targets.

All models, theorems and proofs leading to the top-level correctness statement of the CVM have been verified in Isabelle/HOL — modulo symmetric cases (as the read case of the hard-disk driver and some of the primitives were only specified but not verified). The CVM implementation is made up of 1200 lines of C0 and 300 lines of inline assembly code. Altogether we carried out the CVM verification in almost 100k proof steps in 5k Lemmas.

Integrating the huge amount of specifications, models and proofs emerged as a highly non-trivial and time-consuming engineering task. This covers, among other things, a social process, in which the work of many researchers, located at different places, has to be combined to one uniform and formal Isabelle/HOL theory. More than 250 Isabelle theories developed by more than 10 researchers were either directly or indirectly imported to state and verify the top-level correctness of the microkernel.

Larger efforts should be undertaken to simplify and better organize the formal verification in a computer aided proof system as Isabelle/HOL. On the one hand side it would be desirable to have more proof analysis tools as e.g., proof clones detection. We think, that in projects with such a large theory corpus, 'proof-by-search' technology (as automatically finding already proven similar lemmas) may be highly promising. On the other hand, the use of automatic tools in Verisoft often failed due to a huge overhead caused by the integration of external tools. Linking results obtained by e.g. automatic first-order theorem provers or SAT solvers is often much harder than proving the requested goal by hand.

References

1. Alkassar, E.: OS Verification Extended - On the Formal Verification of Device Drivers and the Correctness of Client/Server Software. PhD thesis, Saarland University, Computer Science Dept. (2009)
2. Alkassar, E., Hillebrand, M.: Formal functional verification of device drivers. In: Shankar, N., Woodcock, J. (eds.) VSTTE 2008. LNCS, vol. 5295, pp. 225–239. Springer, Heidelberg (2008)
3. Alkassar, E., Hillebrand, M., Leinenbach, D., Schirmer, N., Starostin, A.: The verisoft approach to systems verification. In: Shankar, N., Woodcock, J. (eds.) VSTTE 2008. LNCS, vol. 5295, pp. 209–224. Springer, Heidelberg (2008)
4. Alkassar, E., Hillebrand, M., Leinenbach, D., Schirmer, N., Starostin, A., Tsyban, A.: Balancing the load: Leveraging semantics stack for systems verification. J. Autom. Reasoning 42(2-4), 389–454 (2009)
5. Alkassar, E., Schirmer, N., Starostin, A.: Formal pervasive verification of a paging mechanism. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 109–123. Springer, Heidelberg (2008)
6. Bevier, W.R., Hunt, W.A., Moore, J.S., Young, W.D.: Special issue on system verification. J. Autom. Reasoning 5(4), 409–530 (1989)
7. Beyer, S., Jacobi, C., Kröning, D., Leinenbach, D., Paul, W.: Putting it all together - formal verification of the vamp. STTT Journal, Special Issue on Recent Advances in Hardware Verification (2005)

8. Dalinger, I., Hillebrand, M., Paul, W.: On the verification of memory management mechanisms. In: Borrione, D., Paul, W. (eds.) CHARME 2005. LNCS, vol. 3725, pp. 301–316. Springer, Heidelberg (2005)
9. Daum, M., Dörrenbächer, J., Wolff, B.: Proving fairness and implementation correctness of a microkernel scheduler. J. Autom. Reasoning 42(2-4), 349–388 (2009)
10. Daum, M., Schirmer, N., Schmidt, M.: Implementation correctness of a real-time operating system. In: SEFM 2009, pp. 23–32. IEEE, Los Alamitos (2009)
11. Klein, G., et al.: seL4: Formal verification of an OS kernel. In: SOSP 2009, Big Sky, MT, USA, October 2009, pp. 207–220. ACM, New York (2009)
12. Feng, X., Shao, Z., Guo, Y., Dong, Y.: Combining domain-specific and foundational logics to verify complete software systems. In: Shankar, N., Woodcock, J. (eds.) VSTTE 2008. LNCS, vol. 5295, pp. 54–69. Springer, Heidelberg (2008)
13. Feng, X., Shao, Z., Guo, Y., Dong, Y.: Certifying low-level programs with hardware interrupts and preemptive threads. J. Autom. Reasoning 42(2-4), 301–347 (2009)
14. Gargano, M., Hillebrand, M., Leinenbach, D., Paul, W.: On the correctness of operating system kernels. In: Hurd, J., Melham, T. (eds.) TPHOLs 2005. LNCS, vol. 3603, pp. 1–16. Springer, Heidelberg (2005)
15. In der Rieden, T., Tsyban, A.: Cvm - a verified framework for microkernel programmers. In: Huuck, R., Klein, G., Schlich, B. (eds.) SSV 2008. ENTCS, vol. 217, pp. 151–168. Elsevier Science B.V., Amsterdam (2008)
16. Klein, G.: Operating system verification — an overview. Sādhanā 34(1), 27–69 (2009)
17. Leinenbach, D.: Compiler Verification in the Context of Pervasive System Verification. PhD thesis, Saarland University, Computer Science Dept. (2008)
18. Moore, J.S.: A grand challenge proposal for formal methods: A verified stack. In: Aichernig, B.K., Maibaum, T.S.E. (eds.) Formal Methods at the Crossroads. From Panacea to Foundational Support. LNCS, vol. 2757, pp. 161–172. Springer, Heidelberg (2003)
19. Starostin, A.: Formal Verification of Demand Paging. PhD thesis, Saarland University, Computer Science Dept. (2010)
20. Starostin, A., Tsyban, A.: Correct microkernel primitives. In: Huuck, R., Klein, G., Schlich, B. (eds.) SSV 2008. ENTCS, vol. 217, pp. 169–185. Elsevier Science B. V., Amsterdam (2008)
21. Starostin, A., Tsyban, A.: Verified process-context switch for C-programmed kernels. In: Shankar, N., Woodcock, J. (eds.) VSTTE 2008. LNCS, vol. 5295, pp. 240–254. Springer, Heidelberg (2008)
22. Tsyban, A.: Formal Verification of a Framework for Microkernel Programmers. PhD thesis, Saarland University, Computer Science Dept. (2009)
23. Wenzel, M., Paulson, L.C., Nipkow, T.: The isabelle framework. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 33–38. Springer, Heidelberg (2008)