



Communicating Virtual Machines

A Formal Model for the Specification
and Verification of Operating System Kernels

Dirk Leinenbach

dirkl@cs.uni-sb.de

Saarland University, Germany

joint work with: M. Gargano, M. Hillebrand,
W.J. Paul



Outline

- Verisoft project
- Hardware: VAMP processor
- C0 and Compiler-Correctness
- Communicating virtual machines \Leftrightarrow Microkernel
- Simple operating system

Project Verisoft

- funded by German federal government
- formal verification of
 - Hardware
 - Compiler
 - Operating system
 - Communication system (TCP/IP stack)
 - Application
- by a single formally proven theorem
(c.f. CLI stack)

Verisoft: partners 2004

- Saarland University (project management)
- TU Darmstadt
- TU München
- Univ. Koblenz
- DFKI (German research center for artificial intelligence) Saarbrücken
- MPI (Max Planck Institute) Saarbrücken
- BMW
- Infineon
- T-Systems
- Absint

we want to verify

- Infineon: commercial high end controller
- T-Systems: biometric identification system (parts)
- BMW: OSEK-Time OS + application
- public system:
 - VAMP processor
 - assembler, C0, C0 compiler
 - variant of L4-kernel
 - simple operating system
 - ... electronic signature of email

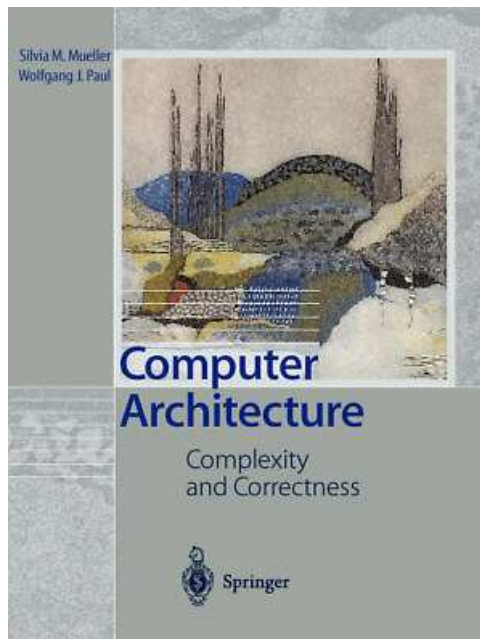
How do we verify ?

- not everything fully automatically
 - impossible (1. Gödel incompleteness theorem + related results from complexity)
- 1. paper-and-pencil proofs
 - construction plan of formal proofs
- 2. code this proof in interactive CAV-system (Isabelle/HOL)
- 3. prove as many lemmas as possible automatically (development of tools!)

Productivity (2004)

- entering 50-100 pages of mathematical text into CAV-system: 1 man year
 - verifying 1 page of typical C0-Code in Hoare-Logic: 1 man week
- ?? How big are correctness proofs ?
- ?? How long is the code ?
- ?? Where are paper and pencil proofs for real systems ??

Computer architecture (700 pages)



+ Diss. D. Kröning
(Tomasulo scheduler)

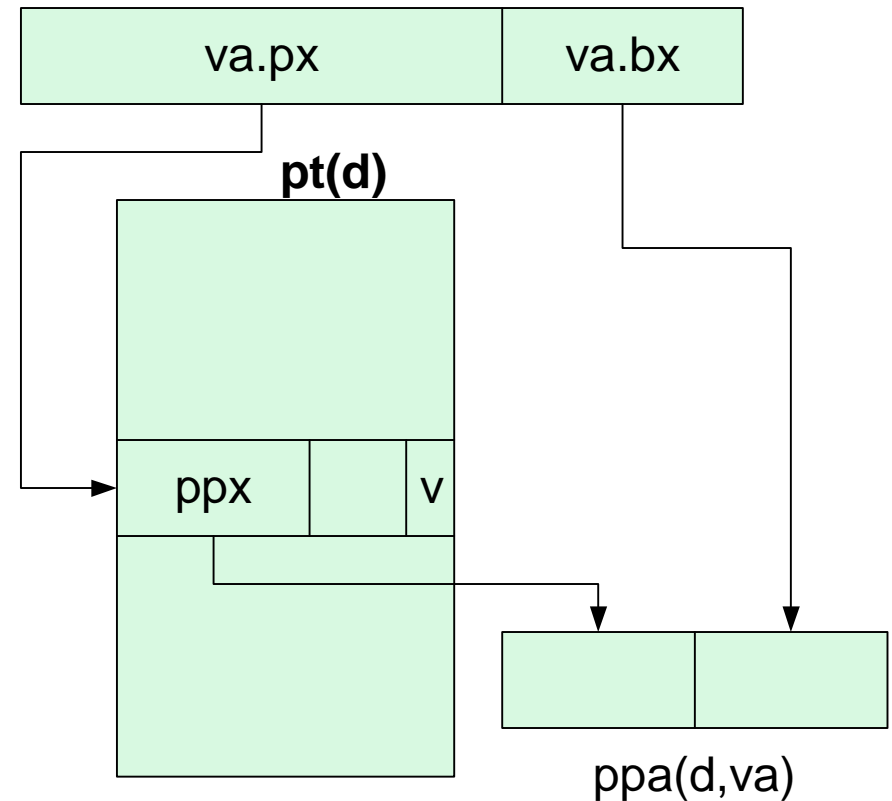
- DLX instruction set
- pipelining, forwarding, interlock
- precise, maskable nested interrupts
- split cache
- fully IEEE compatible FPU
- Tomasulo scheduler

formally verified processor

- Beyer et al.: VAMP....
- Charme 03
- CAV-System PVS
- Synthesized on FPGA
- 1.5 Mio gates
- about 8 man years
- DLX instruction set
- pipelining, forwarding, interlock
- precise, maskable nested interrupts
- split cache
- fully IEEE compatible FPU
- Tomasulo scheduler
- memory management

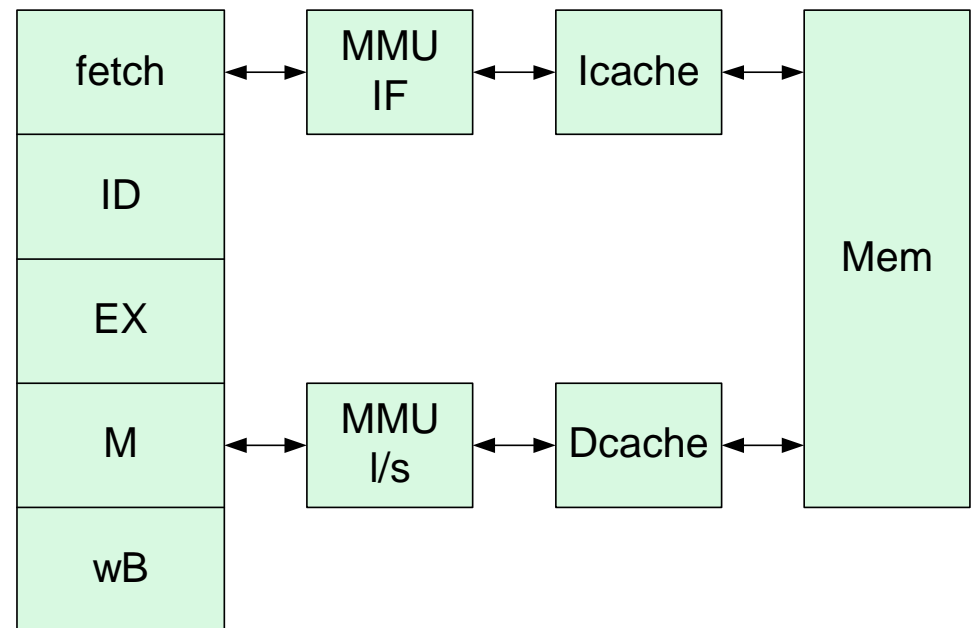
memory management (1): address translation sequentially

- Address translation of virtual addresses va
 - $va = (va.px, va.bx)$
 - px : page index,
 - bx : byte index
 - d VAMP configuration
 - pt : page table
- physical page address $ppa(d, va) = (pt(d, va.px).ppx, va.bx)$
- ppx : physical page index



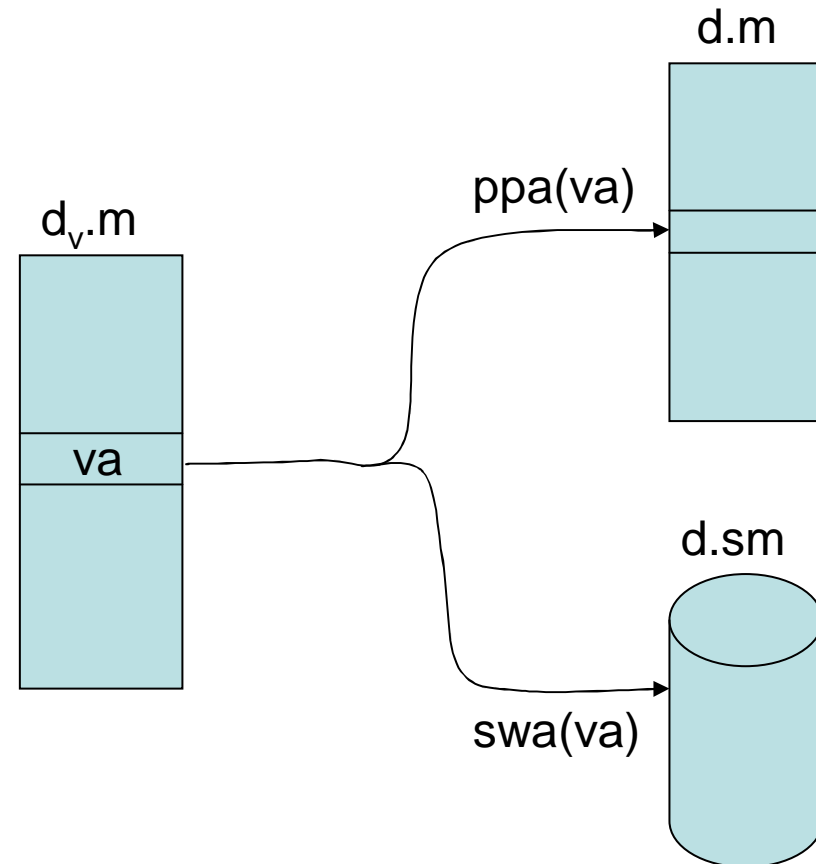
memory management (2): address translation in a pipeline

- 2 MMU's (data + instr)
- No TLB → two memory accesses (pte, memory)
- correctness: pt and memory must stay constant during translated access
- sync (pipe drain) instructions + software requirements (!)
- processor correctness theorem



memory management (3): simulation of virtual machine

- $d_v.vm(va) =$
 - $d.m(pma(va))$:
 $pt(d, va.px).v = 1$
(in cache)
 - $d.sm(sma(va))$: otherwise
- $d.m$ is **cache** für $d_v.vm$
- **theorem:**
VAMP + page fault handler simulates **virtual machine**
- no page faults visible in virtual machine



memory management (4): status

- Hardware verification formal: summer 2004 ✓
(complete in case of no interrupts)
- Simulation theorem on paper (~ 50 pages) ✓
- Simulation theorem formal: soon

C0 language

- type-safe and simplified version of C
 - fully recursive data types
 - pointers and heap
 - no pointer arithmetic
- operational **small step semantics** (c.f. Loexcks, Mehlhorn, Wilhelm 1986)
- formalized in PVS and Isabelle/HOL
- Hoare-Logics by N. Schirmer

Compiler

- delayed branch
- Filling of delay slots
- Alignment of 1, 2, 4, 8 byte data
- paper-and-pencil: construction + proof
- 50 pages (2004)
- presented in lectures on systems architecture to 1'st year students

Compiler (2): layers

- Computer systems have layers (not always....)
- Every layer corresponds to an abstract machine model
 - assembler: RAM from theoretical computer science
 - C0: abstract C0-machine
- correctness: **simulation** of C0-program p by VAMP program $\text{code}(p)$

Compiler (3): simulation relation

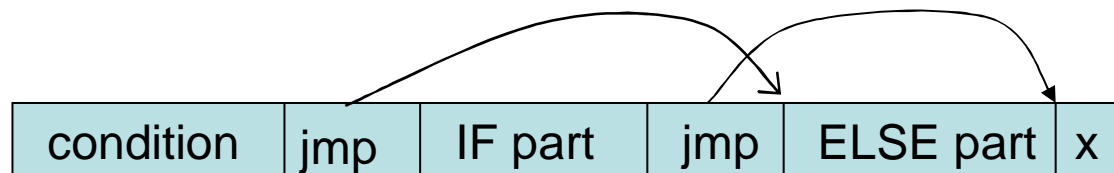
- Consis(c, alloc, d)
- C0-variable x stored at address alloc(c,x) of VAMP machine
- data consistency:
 - e-consis: elementary data have same values
 - p-consis: graph-isomorphism between heaps
 - r-consis: implementation of function stack
- code consistency:
 - c-consis: PC points to start(code(program rest))

Compiler(4) simulation theorem

- For each C0-computation c^0, c^1, \dots there is
 - assembler computation d^0, d^1, \dots
 - functions alloc^i
 - Numbers of steps $s(i)$

such that for all i (except filled delay slots)

- $\text{consis}(c^i, \text{alloc}^i, d^{s(i)})$
- Induction step for code consistency nontrivial



CVM: abstract model

- CVM: Communicating Virtual Machines
- abstract kernel: C0 program k_a
- user programs : virtual machines $VM(p)$
- in every step either kernel k_a or one user machine is active
- trap instruction of $VM(p)$ calls functions of kernel
 - formal procedure semantics for interrupt-handling (!)
- CVM primitives used to specify kernel behavior

CVM: abstract model

- CVM primitives:
 - startnext(p): set user process p active
 - memory allocation for user programs
 - definition of handlers (floating point exceptions) by user
 - copying memory regions between users VM(p) (and I/O-ports)
- scheduler is part of abstract kernel
- page faults in CVM not visible (virtual machines!)

CVM-implementation by concrete kernel K_c

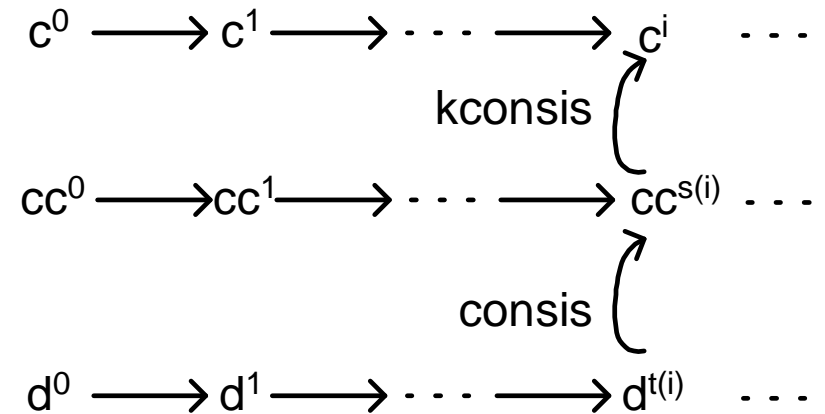
- K_c has **necessarily** in line assembler code
 - user processes and processor registers not visible as C0 variables
- $C0_A$: C0 extended by in line assembler macros
 - compilation ✓
 - **semantics**: run C0-machine and assembly machine simultaneously.
During in line code C0-variables must be held consistent (involves function **alloc** from Compiler)

handling of interrupts by concrete kernel K_c

- using inline code save registers in record `pcb(act)`
 - `pcb`: process control block
- dispatcher (C0-Code)
 - reads `pcb(act)`
 - calls function of K_c as handler
- Handler: trap, page fault (invisible in CVM), user defined,...
- interrupt is (mostly) implemented as function call

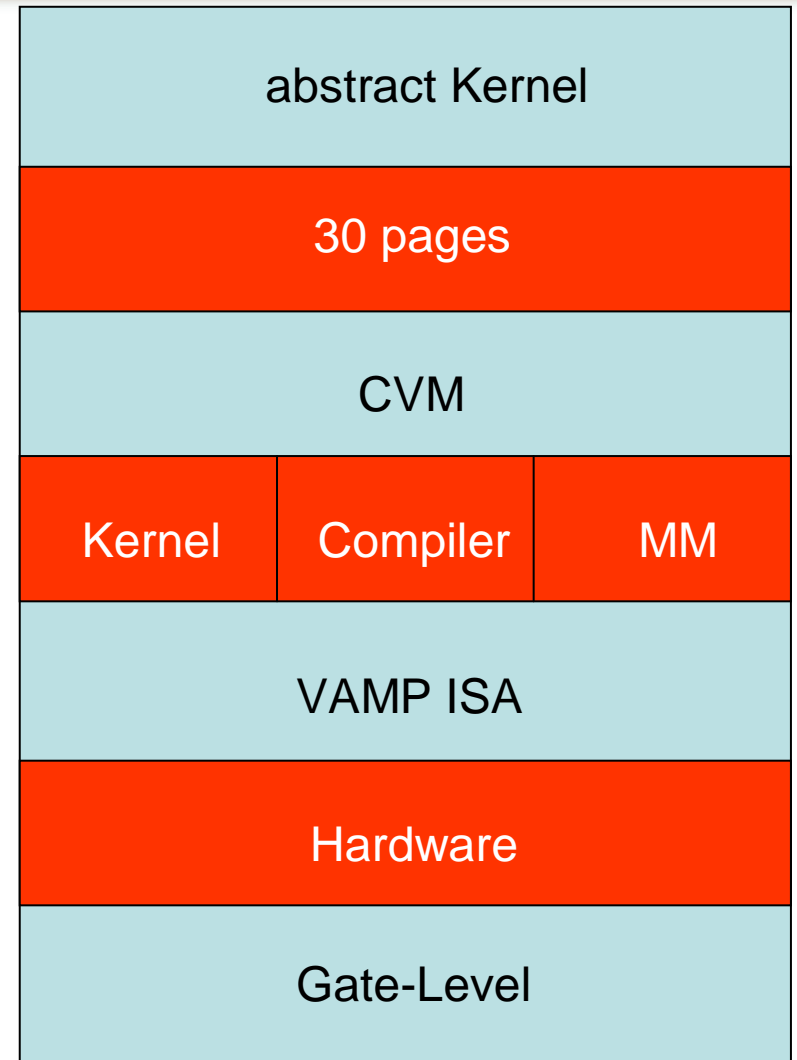
CVM: simulation theorem

- 3 computations
 - c^1, c^2, \dots : CVM (+ abstract kernel k_a)
 - cc^1, cc^2, \dots : concrete kernel K_c
 - d^1, d^2, \dots : VAMP
- 2 sequences of
 - Numbers of steps $s(i)$ and $t(j)$
 - Allocation functions $alloc^i$ and $kalloc^j$
- such that
 - $kconsis(c^i.k, kalloc^i, cc^{s(i)})$
 - $consis(cc^{s(i)}, alloc^i, d^{t(i)})$
 - $d^{t(i)}$ codes all virtual machines and the kernel from the CVM level



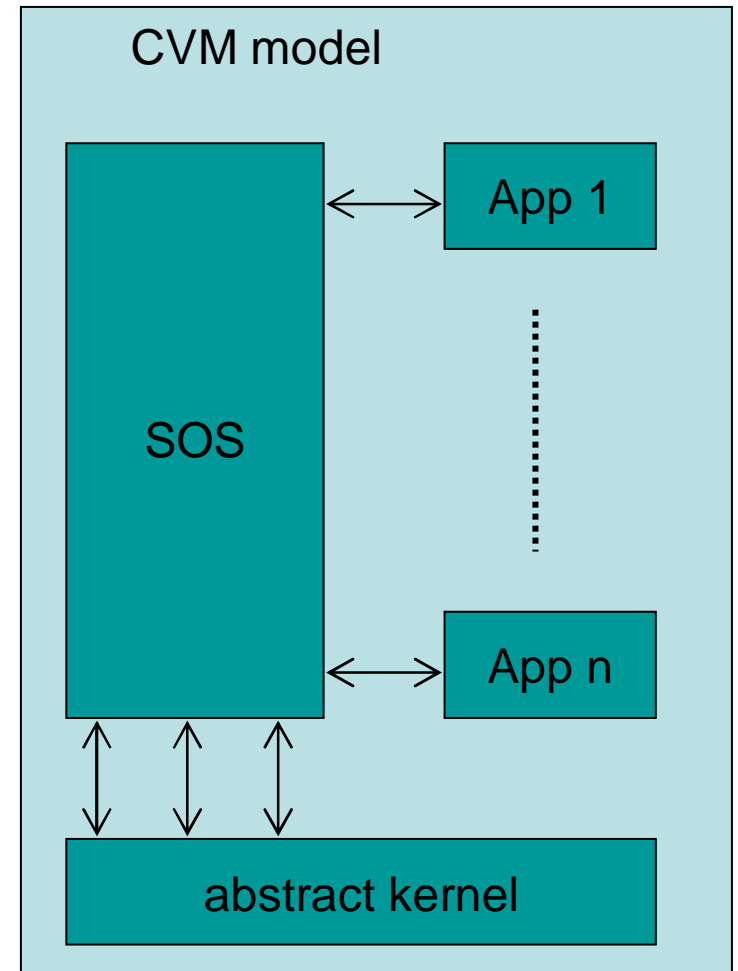
public system: correctness proof

- user mode, page fault:
uses correctness of
memory management
- kernel from $C0_A$:
uses compiler-correctness
- hardware:
assumptions on
synchronization are
proven for page fault
handlers of the concrete
kernel K_c



Simple operating system: SOS

- based on abstract kernel
- SOS is **privileged** user process
- user machines interact only with SOS (via kernel)
- functionality:
 - time, signals
 - process Management
 - files
 - sockets, screen & keyboard
 - inter process communication
 - remote procedure calls



Summary

- Verisoft:
 - persistent formal verification of computer systems
 - strong cooperation with industry
- Public demonstration system (cf. CLI stack)
 - Hardware (VAMP)
 - Assembler
 - C0, Compiler
 - Kernel
 - Simple Operating System
 - Application Software