The Semantics Stack of the Verisoft XT Project

Christoph Baumann

KTH Royal Institute of Technology
Stockholm, Sweden

January 25, ENTROPY 2018

cbaumann@kth.se
Verisoft: Pervasive OS Verification (2003-2007)

- VAMP Hardware
- DLX ISA Model
- C0+inline ASM
- CVM framework
- VAMOS / OLOS
- verification
- abstraction
- property transfer

- Kernel correctness
- OS primitive correctness
- Compiler correctness
- HW correctness

C. Baumann (KTH Stockholm)
Verisoft XT Project (2007-2010)

- Pervasive Formal Verification of Realistic Computer Systems
Pervasive Formal Verification of Realistic Computer Systems

Hardware Verification (Infineon)

Automotive Software (Audi, Bosch)

Avionics Project (SysGo, ESG, TUV)
Verisoft XT Project (2007-2010)

- Pervasive Formal Verification of Realistic Computer Systems
- Hardware Verification (Infineon)
- Automotive Software (Audi, Bosch)
- Avionics Project (Sysgo, ESG, TUV)
- Hypervisor Project (Microsoft)
Verisoft XT Project (2007-2010)

- Pervasive Formal Verification of Realistic Computer Systems
- Hardware Verification (Infineon)
- Automotive Software (Audi, Bosch)
- Avionics Project (Sysgo, ESG, TUV)
- Hypervisor Project (Microsoft)
- tool development: VCC
Results:

- large portions of code verified
- kernel and hardware specifications
- powerful verifier VCC
Results:

- large portions of code verified
- kernel and hardware specifications
- powerful verifier VCC

Main issue: missing semantical underlay
Results:
- large portions of code verified
- kernel and hardware specifications
- powerful verifier VCC

Main issue: **missing semantical underlay**
- weak memory model
- memory management units with TLBs
- mixed C & assembly code
- interruptible C
- multi-threaded C
- interleaved user and device steps
- concurrent compiler correctness
Lots of people involved in the presented work:

- Artem Alekhin
- Geng Chen
- Ernie Cohen
- Ulan Degenbaev
- Mikhail Kovalev
- Petro Lutsyk
- Jonas Oberhauser
- Hristo Pentchev
- Prof. Wolfgang J. Paul
- Norbert Schirmer
- Sabine Schmaltz
- Andrey Shadrin
“Theory of Multi Core Hypervisor Verification”
Cohen, Paul, and Schmaltz, 2013

Kernel Specification

Verification Approach

Semantics Stack

Hardware Abstraction

Gate Level Hardware
Instruction Set Architecture
Instruction Set Architecture

- system programmer’s model
- memory management unit
- cache control instructions
- pipelining artifacts
- weak memory model
- undefined behaviour
Instruction Set Architecture

- system programmer’s model
- memory management unit
- cache control instructions
- pipelining artifacts
- weak memory model
- undefined behaviour
- *software conditions*
Instruction Set Architecture

- system programmer’s model
- memory management unit
- cache control instructions
- pipelining artifacts
- weak memory model
- undefined behaviour
- software conditions

Machine readable models:

- rudimentary PowerPC model
- x64:
  
  "Formal Specification of the x86 Instruction Set Architecture"
  Ulan Degenbaev, PhD thesis, 2011
Hardware correctness: MIPS86
Hardware correctness: MIPS86

- MIPS instruction core with x86 memory system:
  - MMUs / TLBs
  - interrupts
  - devices
  - caches
  - store buffers
Hardware correctness: MIPS86

- MIPS instruction core with x86 memory system:
  - MMUs / TLBs
  - interrupts
  - devices
  - caches
  - store buffers

- correctness proof for pipelined multicore implementation with...
Hardware Abstraction

Hardware correctness: MIPS86

- MIPS instruction core with x86 memory system:
  - MMUs / TLBs
  - interrupts
  - devices
  - caches
  - store buffers

- correctness proof for pipelined multicore implementation with

- find software conditions of ISA
Recent text books / lecture notes:


Reducing caches, store buffers, and MMU
Cache abstraction

- MOESI protocol
- give parallel implementation
- correctness proof and simulation

Theorem

If every address is accessed in the same cache mode by all processors then caches are invisible.

Proof: “A Pipelined Multi-core MIPS Machine”
Kovalev, Müller, and Paul, 2014
Hardware Abstraction

Store buffer reduction

\[
\begin{align*}
\{ x \land y = 0 \} & \\
( x := 1; R_1 = y ) & || ( y := 1; R_2 = x ) \\
\{ \neg ( R_1 = 0 \land R_2 = 0 ) \}
\end{align*}
\]

need to add fences which flush the store buffer:

\[
( x := 1; \text{FENCE}; R_1 = y ) & || ( y := 1; \text{FENCE}; R_2 = x )
\]

efficient flushing policy:

1. Mark concurrent accesses to the same address as shared!
2. Between any shared write and shared read, flush the store buffer!

concurrent accesses: exists an interleaved execution schedule:

consecutive steps by different threads access the same address at least one of them is modifying its value
Store buffer reduction

- buffer writes locally before committing them to memory

\[
\begin{align*}
\{x \land y = 0 \} \\
(x := 1; R_1 = y) \mathrel{|} | (y := 1; R_2 = x) \\
\{\neg (R_1 = 0 \land R_2 = 0) \}
\end{align*}
\]

need to add fences which flush the store buffer:

\[
\begin{align*}
(x := 1; FENCE; R_1 = y) \mathrel{|} | (y := 1; FENCE; R_2 = x) \\
\end{align*}
\]

efficient flushing policy:

1. Mark concurrent accesses to the same address as shared!
2. Between any shared write and shared read, flush the store buffer!

concurrent accesses: exists an interleaved execution schedule: consecutive steps by different threads access the same address at least one of them is modifying its value
Store buffer reduction

- buffer writes locally before committing them to memory
- can ruin sequential consistency in multicore processors:

\[
\begin{align*}
\{ x = 0 \land y = 0 \} \\
(x := 1; R1 = y) \parallel (y := 1; R2 = x) \\
\{ \neg (R1 = 0 \land R2 = 0) \}
\end{align*}
\]
Store buffer reduction

- buffer writes locally before committing them to memory
- can ruin sequential consistency in multicore processors:
  \[
  \{ x = 0 \land y = 0 \}
  \]
  \[
  (x := 1; R1 = y) \parallel (y := 1; R2 = x)
  \]
  \[
  \{ \neg (R1 = 0 \land R2 = 0) \}\]

- need to add fences which flush the store buffer:
  \[
  (x := 1; FENCE; R1 = y) \parallel (y := 1; FENCE; R2 = x)
  \]
Store buffer reduction

- buffer writes locally before committing them to memory
- can ruin sequential consistency in multicore processors:
  \[
  \{ x = 0 \land y = 0 \} \\
  (x := 1; R1 = y) \ || \ (y := 1; R2 = x) \\
  \{ \neg (R1 = 0 \land R2 = 0) \}
  \]

- need to add fences which flush the store buffer:
  \[
  (x := 1; FENCE; R1 = y) \ || \ (y := 1; FENCE; R2 = x)
  \]

- efficient flushing policy:
  1. Mark concurrent accesses to the same address as *shared*!
Store buffer reduction
- buffer writes locally before committing them to memory
- can ruin sequential consistency in multicore processors:

\[
\begin{align*}
\{ x = 0 \land y = 0 \} \\
(x := 1; R1 = y) \parallel (y := 1; R2 = x) \\
\{ \neg (R1 = 0 \land R2 = 0) \}
\end{align*}
\]

- need to add fences which flush the store buffer:

\[
(x := 1; FENCE; R1 = y) \parallel (y := 1; FENCE; R2 = x)
\]

- efficient flushing policy:
  1. Mark concurrent accesses to the same address as *shared*!
  2. Between any shared write and shared read, flush the store buffer!
Store buffer reduction

- buffer writes locally before committing them to memory
- can ruin sequential consistency in multicore processors:

\[
\begin{align*}
\{ x = 0 \land y = 0 \} \\
(x := 1; R1 = y) \parallel (y := 1; R2 = x) \\
\{ \neg (R1 = 0 \land R2 = 0) \}
\end{align*}
\]

- need to add fences which flush the store buffer:

\[
(x := 1; FENCE; R1 = y) \parallel (y := 1; FENCE; R2 = x)
\]

- efficient flushing policy:
  
  1. Mark concurrent accesses to the same address as *shared*!
  2. Between any shared write and shared read, flush the store buffer!

- *concurrent accesses*: exists an interleaved execution schedule:
  - consecutive steps by different threads access the same address
  - at least one of them is modifying its value
Theorem

If the system without store buffers fulfills the flushing policy then it is a sound abstraction of the system with store buffers.

Proofs:

- “A Simpler Reduction Theorem for x86-TSO”. Jonas Oberhauser, VSTTE 2015
- “Justifying The Strong Memory Semantics of Concurrent High-Level Programming Languages for System Programming”. Jonas Oberhauser, PhD thesis, 2018
Eliminating the MMU

Proof:
“TLB Virtualization in the Context of Hypervisor Verification”
Mikhail Kovalev, PhD thesis, 2013
Hardware Abstraction

- Virtual Memory ISA
- SC Memory ISA
- Abstract Memory ISA
- ISA Model
- Gate Level Hardware

- MMU reduction
- SB reduction
- Cache reduction
- HW correctness
- Verisoft XT Semantics Stack
Sequential Compiler Correctness (Verisoft style)

CIA $= \text{consistency point (CP)}$

compiler correctness
Concurrent compiler correctness
Concurrent compiler correctness

reordered ISA

ISA
Concurrent compiler correctness

C

reordered ISA

ISA
Ownership model:

- **Shared**
- **Read-owned**
- **Write-owned**

Ownership memory access policy:
- **Local steps:** only read \textit{read-owned}, only write \textit{write-owned}
- **Shared steps:** read if not write-owned by other thread, write if not owned by other thread
- Ownership transfer via annotations at shared steps

C. Baumann (KTH Stockholm)
Ownership model:

- **shared**
- **read-owned**
- **write-owned**

Ownership memory access policy:

- **local steps:** only read *read-owned*, only write *write-owned*
- **shared steps:** read if *not write-owned by other thread*, write if *not owned by other thread*
- **ownership transfer via annotations at shared steps**
- **write-ownership = exclusive ownership = local addresses**
Commutativity of Local Steps

For two steps $\alpha$, $\beta$ by different processors were $\alpha$ is local we have:

$$\text{safe}(C, \beta \alpha) \land C \xrightarrow{\beta \alpha} C' \iff \text{safe}(C, \alpha \beta) \land C \xrightarrow{\alpha \beta} C'$$
Commutativity of Local Steps

For two steps $\alpha, \beta$ by different processors were $\alpha$ is local we have:

\[
\text{safe}(C, \beta\alpha) \land C \xrightarrow{\beta\alpha} C' \iff \text{safe}(C, \alpha\beta) \land C \xrightarrow{\alpha\beta} C'
\]
\textit{safety}(C, P):

- All computations starting in $C$ are ownership-safe and fulfill safety property $P$. 

\textit{safety}(C, P): 
- All computations starting in \( C \) are ownership-safe and fulfill safety property \( P \).

\textit{safety}_{CP}(C, P): 
- All \( CP \) block computations starting in \( C \) are ownership-safe and fulfill safety property \( P \).
\textit{safety}(C, P):
\begin{itemize}
  \item All computations starting in \( C \) are ownership-safe and fulfill safety property \( P \).
\end{itemize}

\textit{safety}_{CP}(C, P):
\begin{itemize}
  \item All \( CP \) block computations starting in \( C \) are ownership-safe and fulfill safety property \( P \).
\end{itemize}

\textit{at\_most\_one\_shared}_{CP}(C):
\begin{itemize}
  \item On all \( CP \) block computations starting in \( C \) there is at least one \( CP \) between two shared steps of the same processor
\end{itemize}
safety\((C, P)\):
- All computations starting in \(C\) are ownership-safe and fulfill safety property \(P\).

safety_{\mathcal{CP}}\((C, P)\):
- All \(\mathcal{CP}\) block computations starting in \(C\) are ownership-safe and fulfill safety property \(P\).

at\_\text{most\_one\_shared}_{\mathcal{CP}}\((C)\):
- On all \(\mathcal{CP}\) block computations starting in \(C\) there is at least one \(\mathcal{CP}\) between two shared steps of the same processor.

**Order Reduction Theorem**

\[
\text{safety}_{\mathcal{CP}}\((C, P)\) \land \text{at\_\text{most\_one\_shared}}_{\mathcal{CP}}\((C)\) \implies \text{safety}(C, P)
\]

Proof: “Ownership-Based Order Reduction and Simulation in Shared-Memory Concurrent Computer Systems”
Christoph Baumann, PhD thesis, 2014
Safety Transfer

$safety_{CP}$

$safety$
Safety Transfer

$safety_C$

$safety_{CP}$

$safety$

Must preserve ownership-safety and atomicity of shared accesses!
C Intermediate Language (C-IL)

- C with low-level control flow (gotos & function calls)
- pointer arithmetics
- function pointers
- compiler intrinsics
- operational semantics
- compiler consistency relation for MIPS86

Formalization:

“Towards the Pervasive Formal Verification of Multi-Core Operating Systems and Hypervisors Implemented in C”
Sabine Schmaltz, PhD thesis, 2013
Only C?
Only C?

- OS kernels written in C + Assembler
- C functions call assembly procedures
- Assembly procedures call C functions
Only C?

- OS kernels written in C + Assembler
- C functions call assembly procedures
- Assembly procedures call C functions
- MASM: stack and control flow abstractions
Only C?
- OS kernels written in C + Assembler
- C functions call assembly procedures
- Assembly procedures call C functions
- MASM: stack and control flow abstractions
- Mixed semantics: C-IL+MASM

Formalization:
“Integrated Semantics of Intermediate-Language C and Macro-Assembler for Pervasive Formal Verification of Operating Systems and Hypervisors from VerisoftXT”
Schmaltz and Shadrin, VSTTE 2012

“Mixed Low- and High Level Programming Language Semantics and Automated Verification of a Small Hypervisor”
Andrey Shadrin, 2012
C + Interrupts
C + Interrupts

reordered Interrupt

reordered ISA
C + Interrupts

C+MASM
+Interrupts

reordered
Interrupt

reordered
ISA
Necessary conditions:

- **transparency**: handlers restore the interrupted thread correctly and do not modify its local data.
- **independency**: handlers do not use the register content of the interrupted thread.

Necessary conditions:

- *transparency*: handlers restore the interrupted thread correctly and do not modify its local data.
- *independency*: handlers do not use the register content of the interrupted thread.

Formalization:

Multi-threading

C+MASM

ISA

write to SP
Multi-threading

C+MASM +ASM

ISA
Multi-threading

MT
C+MASM
primitive C+MASM
MT
C+MASM
+ASM
ISA
Semantics Stack

MT C+MASM

Thread switch correctness

C+MASM+ASM

Compiler correctness

IRQ@CP ISA

IRQ reordering

CP Block ISA

Order reduction

Virtual Memory ISA
Summing up:

- theory for pervasive multicore OS verification
- hardware abstraction
- semantics stack
- many general theorems
Summing up:

- theory for pervasive multicore OS verification
- hardware abstraction
- semantics stack
- many general theorems

Open issues:

- MIPS86: out-of-order execution + MMU + SB
- semantic stack without reduced MMU?
- thread migration
- machine-checked theories & proofs
- soundness proof for verification with VCC
Thank You!

Questions?