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)

SOS
Apps
User mode verification
VAMOS / OLOS
Kernel correctness
CVM framework
OS primitive correctness
C0+inline ASM
Compiler correctness
DLX ISA Model
HW correctness
VAMP Hardware
property transfer
abstraction

Verisoft XT Project (2007-2010)

- Pervasive Formal Verification of Realistic Computer Systems

Verisoft XT Project (2007-2010)

- Pervasive Formal Verification of Realistic Computer Systems
- Hardware Verfication (Infineon)
- Automotive Software (Audi, BOSCH)
- Avionics Project (SYSGO EMBEDDING INNOVATIONS, ESG , TÜV SÜD)

Verisoft XT Project (2007-2010)

- Pervasive Formal Verification of Realistic Computer Systems
- Hardware Verfication (Infineon)
- Automotive Software (Audi, BOSCH)
- Avionics Project (SYSGO EMBEDDING INNOVATIONS, ESG, TÜV SÜD)
- *Hypervisor Project (Microsoft)*

Verisoft XT Project (2007-2010)

- Pervasive Formal Verification of Realistic Computer Systems
- Hardware Verfication (Infineon)
- Automotive Software (Audi, BOSCH)
- Avionics Project (SYSGO EMBEDDING INNOVATIONS, ESG, TÜV SÜD)
- *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

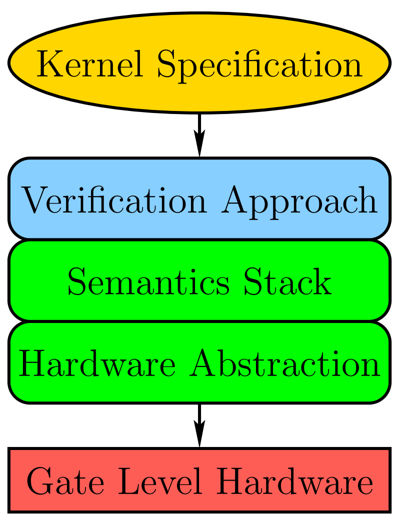
- weak memory model
- memory management units with TLBs
- mixed C & assembly code
- interruptible C
- multi-threaded C
- interleaved user and device steps
- concurrent compiler correctness

DISCLAIMER

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



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*

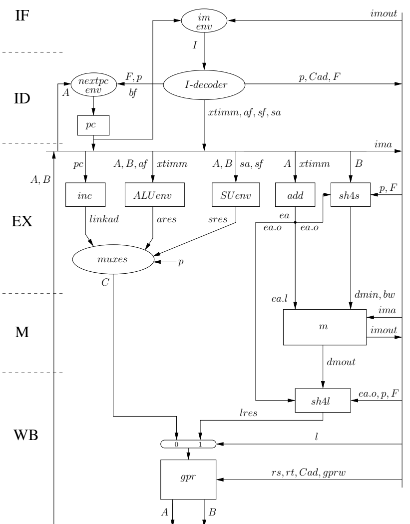
Instruction Set Architecture

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

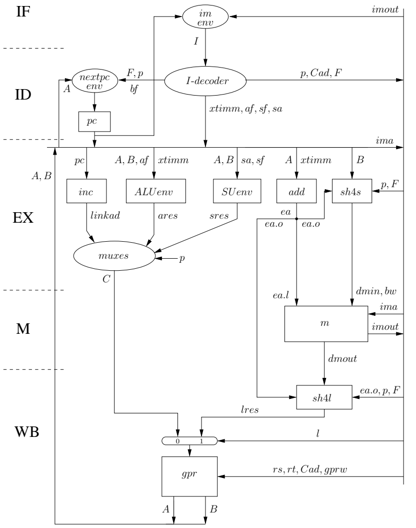
Machine readable models:

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

Hardware correctness: MIPS86

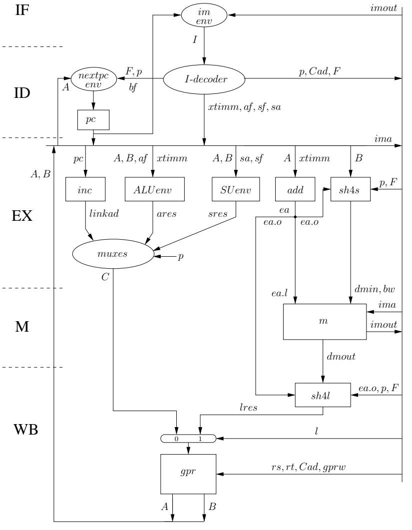


Hardware correctness: MIPS86



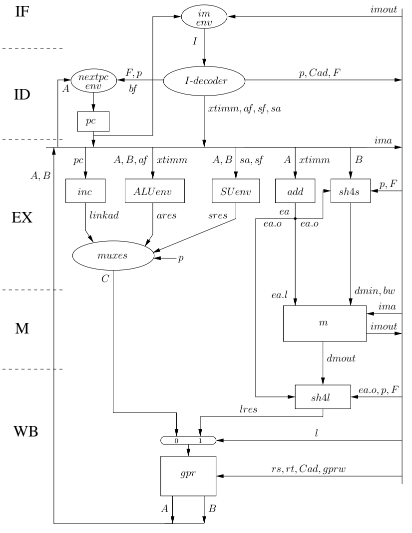
- MIPS instruction core with x86 memory system:
  - MMUs / TLBs
  - interrupts
  - devices
  - caches
  - store buffers

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 correctness: MIPS86

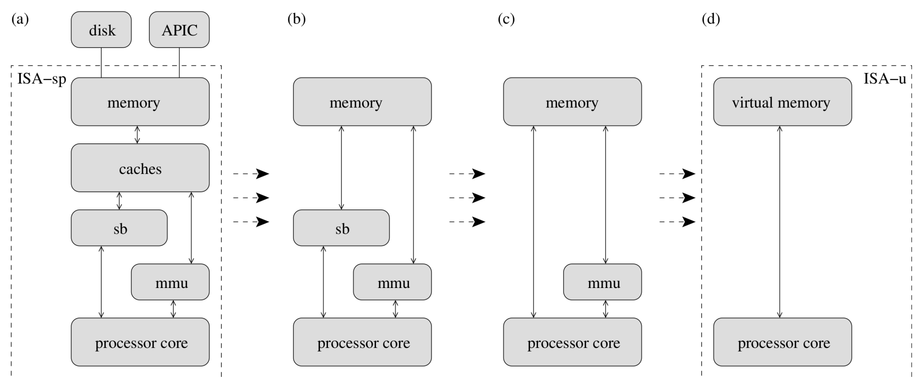


- MIPS instruction core with x86 memory system:
  - MMUs / TLBs
  - interrupts
  - devices
  - caches
  - store buffers
- correctness proof for pipelined multicore implementation with
- find software conditions of ISA

Recent text books / lecture notes:

- *"A Pipelined Multi-core MIPS Machine"*.
  M. Kovalev, S. M. Müller, and W. J. Paul,
  Springer, 2014
- *"System Architecture: An Ordinary Engineering Discipline"*.
  W. J. Paul, C. Baumann, P. Lutsyk, and S. Schmaltz,
  Springer, 2016
- *"Multicore System Architecture"*.
  W. J. Paul, P. Lutsyk, and J. Oberhauser,
  Springer, to be published

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

Store buffer reduction

Store buffer reduction

- buffer writes locally before commiting them to memory

Store buffer reduction

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

$$\{x = 0 \land y = 0\}$$
$$(x := 1; R1 = y) \mid\mid (y := 1; R2 = x)$$
$$\{\neg(R1 = 0 \land R2 = 0)\}$$

Store buffer reduction

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

$$\{x = 0 \wedge y = 0\}$$
$$(x := 1; R1 = y) \parallel (y := 1; R2 = x)$$
$$\{\neg(R1 = 0 \wedge R2 = 0)\}$$

- 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 commiting them to memory
- can ruin sequential consistency in multicore processors:

$$\{x = 0 \wedge y = 0\}$$
$$(x := 1; R1 = y) \mid\mid (y := 1; R2 = x)$$
$$\{\neg(R1 = 0 \wedge R2 = 0)\}$$

- need to add fences which flush the store buffer:

$$(x := 1; FENCE; R1 = y) \mid\mid (y := 1; FENCE; R2 = x)$$

- efficient flushing policy:
  1. Mark concurrent accesses to the same address as *shared*!

Store buffer reduction

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

$$\{x = 0 \wedge y = 0\}$$
$$(x := 1; R1 = y) \mid\mid (y := 1; R2 = x)$$
$$\{\neg(R1 = 0 \wedge R2 = 0)\}$$

- need to add fences which flush the store buffer:

$$(x := 1; FENCE; R1 = y) \mid\mid (y := 1; FENCE; R2 = x)$$

- 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 commiting them to memory
- can ruin sequential consistency in multicore processors:

$$\{x = 0 \wedge y = 0\}$$
$$(x := 1; R1 = y) \,||\, (y := 1; R2 = x)$$
$$\{\neg(R1 = 0 \wedge R2 = 0)\}$$

- 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*!
  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

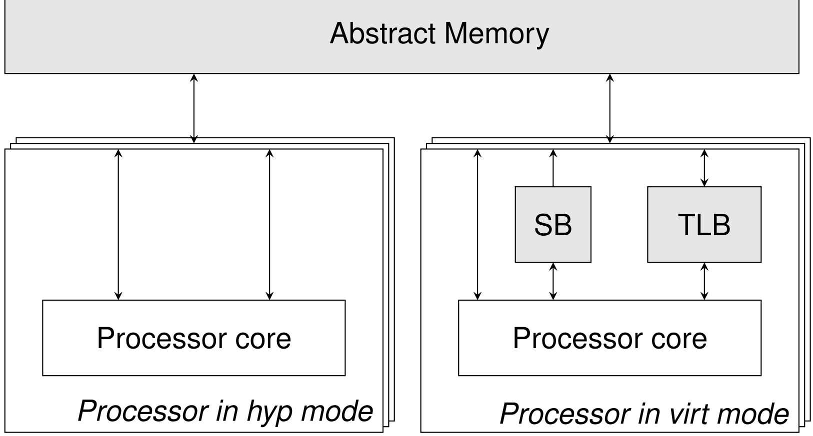
Theorem

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

Proofs:

- *"A Better Reduction Theorem for Store Buffers"*. Cohen and Schirmer, tech report 2009 / ITP 2010
- *"Store Buffer Reduction with MMUs: Complete Paper-and-pencil Proof"*. Chen, Cohen, and Kovalev, tech report 2013 / VSTTE 2014
- *"Store Buffer Reduction Theorem and Application"*. Geng Chen, PhD thesis, 2016
- *"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

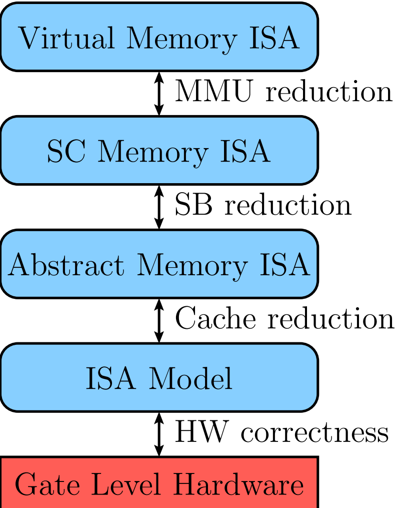
Eliminating the MMU



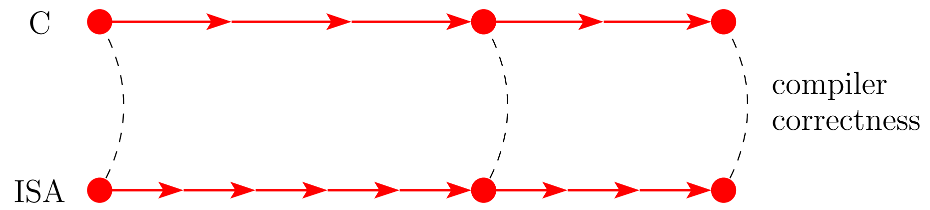
Proof:

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

Hardware Abstraction

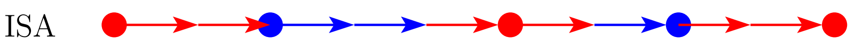


Sequential Compiler Correctness (Verisoft style)

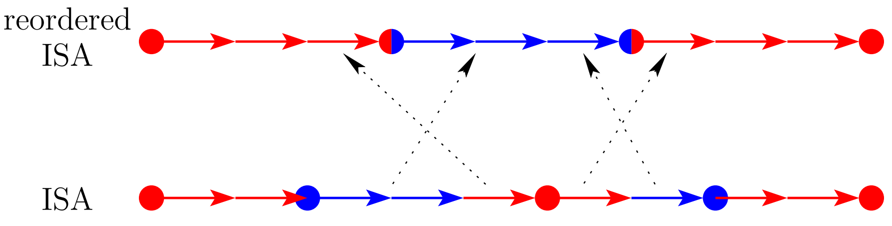


= consistency point ($\mathcal{CP}$)

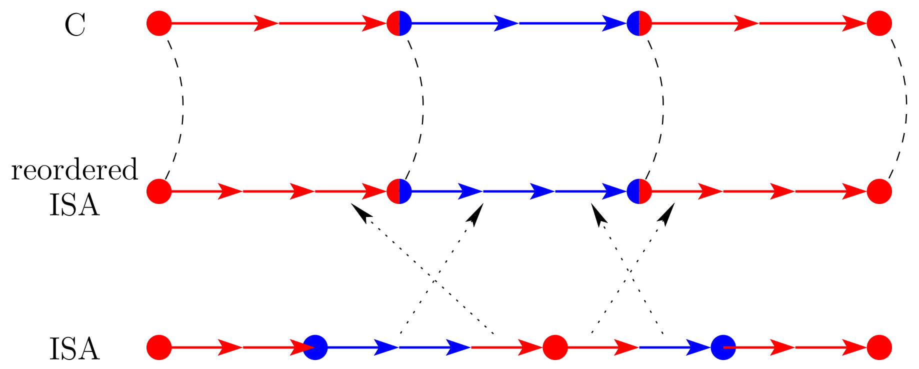
Concurrent compiler correctness



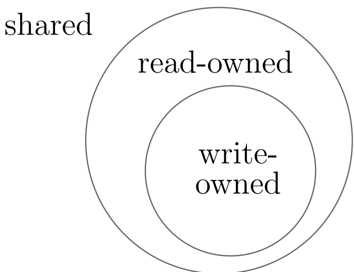
Concurrent compiler correctness



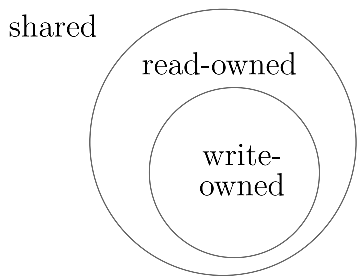
Concurrent compiler correctness



Ownership model:



Ownership model:



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

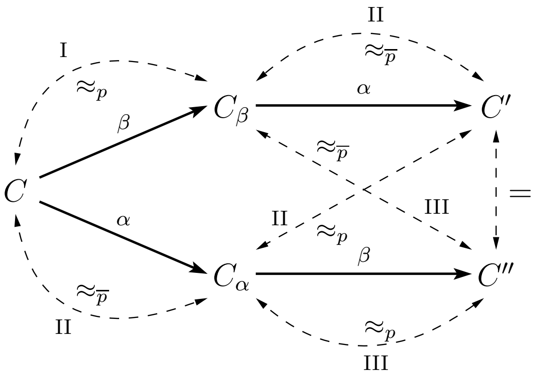
For two steps α, β by different processors were α is local we have:

$$\mathit{safe}(C, \beta\alpha) \wedge C \xmapsto{\beta\alpha} C' \iff \mathit{safe}(C, \alpha\beta) \wedge C \xmapsto{\alpha\beta} C'$$

Commutativity of Local Steps

For two steps α, β by different processors were α is local we have:

$$safe(C, \beta\alpha) \wedge C \overset{\beta\alpha}{\longmapsto} C' \iff safe(C, \alpha\beta) \wedge C \overset{\alpha\beta}{\longmapsto} C'$$



$safety(C, P)$:

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

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.

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_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

$\mathit{safety}(C, P)$:

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

$\mathit{safety}_{\mathcal{CP}}(C, P)$:

- All $\mathcal{CP}$ block computations starting in C are ownership-safe and fulfill safety property P.

$\mathit{at_most_one_shared}_{\mathcal{CP}}(C)$:

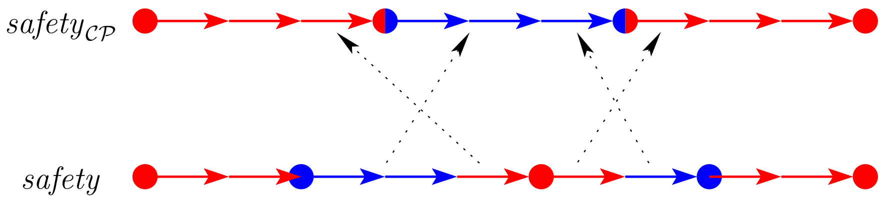
- On all $\mathcal{CP}$ block computations starting in C there is at least one $\mathcal{CP}$ between two shared steps of the same processor

Order Reduction Theorem

$$\mathit{safety}_{\mathcal{CP}}(C, P) \wedge \mathit{at_most_one_shared}_{\mathcal{CP}}(C) \implies \mathit{safety}(C, P)$$

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

Safety Transfer



Safety Transfer

$safety_C$

$safety_{C\mathcal{P}}$

$safety$

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?

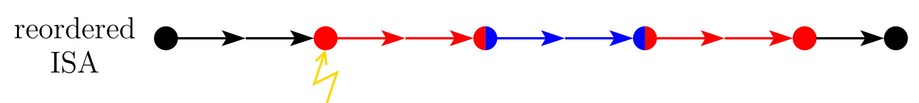
- OS kernels written in C + Assembler
- C functions call assembly procedures
- Assembly procedures call C functions
- MASM: stack and control flow abstractions
- Mixed semantics: C-IL+MASM

Formalization:

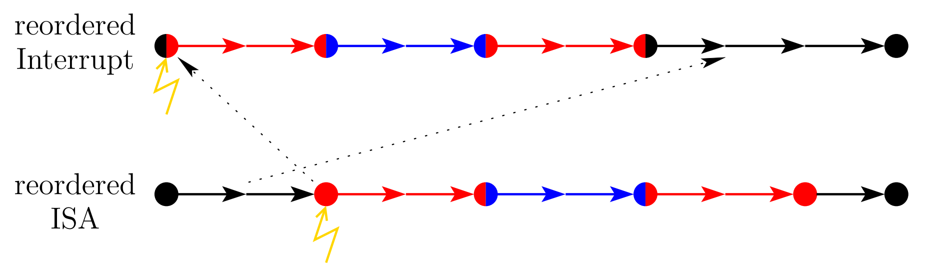
"Integrated Semantics of Intermediate-Language C and Macro-Assembler for Pervasive Formal Verification of Operating Systems and Hypervisors from VerisoftXT"
Schmaltz and Shadrin, VSTTE 2012

"Mixed Low- and High Level Programming Language Semantics and Automated Verification of a Small Hypervisor"
Andrey Shadrin, 2012

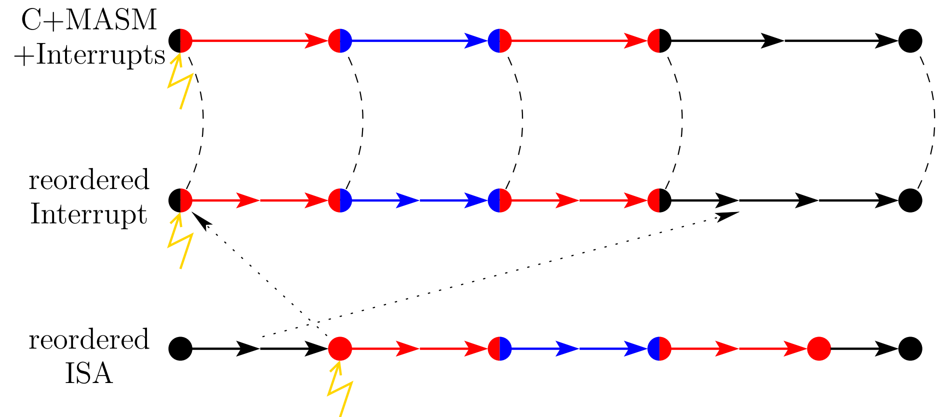
C + Interrupts



C + Interrupts



C + Interrupts



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.

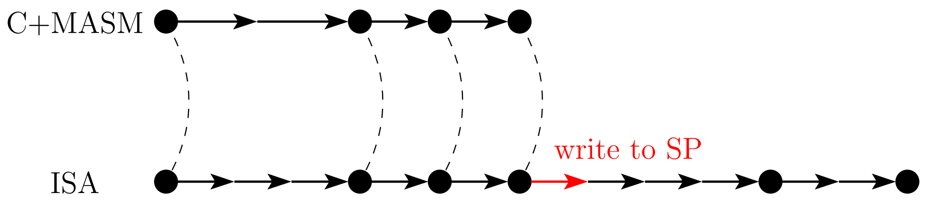
Necessary conditions:

- *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.

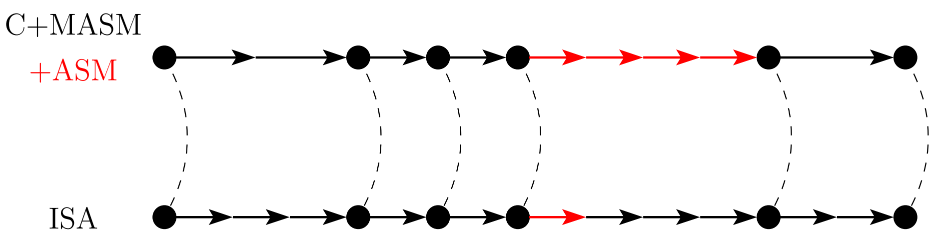
Formalization:

- *"Sound semantics of a high-level language with interprocessor interrupts"*. Hristo Pentchev, PhD thesis, 2016
- *"Order Reduction for Multi-core Interruptible Operating Systems"*. Jonas Oberhauser, VSTTE, 2016

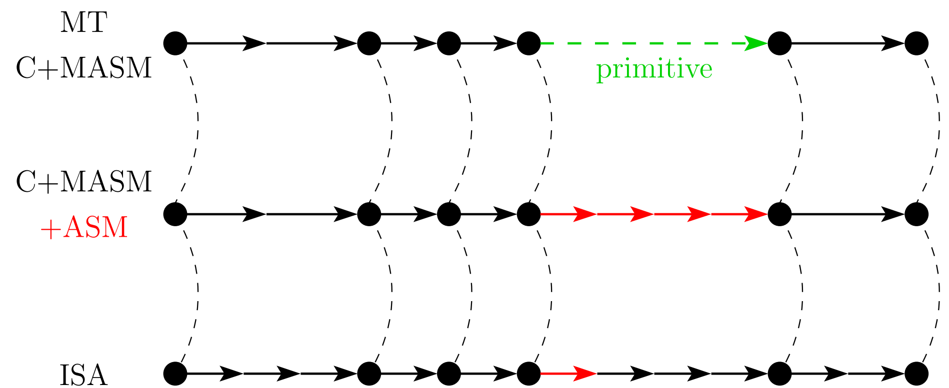
Multi-threading



Multi-threading



Multi-threading



Multi-threading

MT
C+MASM

primitive

C+MASM
+ASM

ISA

"Provably sound semantics stack for multi-core system programming with kernel threads". Artem Alekhin, PhD thesis, 2017

Semantics Stack

MT C+MASM

↕ Thread switch correctness

C+MASM+ASM

↕ Compiler correctness

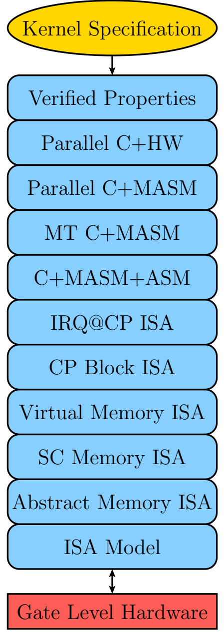
IRQ@CP ISA

↕ IRQ reordering

CP Block ISA

↕ Order reduction

Virtual Memory ISA


Kernel Specification
Verified Properties
Parallel C+HW
Parallel C+MASM
MT C+MASM
C+MASM+ASM
IRQ@CP ISA
CP Block ISA
Virtual Memory ISA
SC Memory ISA
Abstract Memory ISA
ISA Model
Gate Level Hardware

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

The End

Thank You!

Questions?