Relaxed Systems Architecture: Instruction Fetching

Ben Simner

University of Cambridge

In collaboration with Shaked Flur, Christopher Pulte, Alasdair Armstrong, Jean Pichon, Luc Maranget\(^1\) and Peter Sewell

\(^1\)INRIA Paris
Motivation

Why?

Want to understand: TLBs, Instruction Caches, Interrupts
Want to prove: Operating Systems, JITs, Hypervisors
But first…

Computers are fast…

… but terrible!
Intel (Skylake) die

Source: https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(client)

\(^2\text{Source: https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(client)}\)
Intel (Skylake) die
Intel (Skylake) die
Intel (Skylake) die
Intel (Skylake) die
Dekker’s/Peterson’s mutual exclusion algorithm (extract)

Thread A

flagA ← 1;
while flagB
{};
print(“A”)

Thread B

flagB ← 1;
while flagA
{};
print(“B”)

x86 hardware can execute both prints!
x86: TSO Architecture

Source Code

Thread A
flagA ← 1;
print(flagB)

Thread B
flagB ← 1;
print(flagA)
State of the Art

Models:
- Abstract Hardware Operational
- Axiomatic-Style
x86-TSO: Operational Semantics

▶ State = Abstracted Machine State

\[ m : \langle M : \text{addr} \rightarrow \text{value}; \]
\[ B : \text{tid} \rightarrow (\text{addr} \times \text{value}) \text{ list}; \rangle \]

▶ Structural Operational Semantics

\[ m' = \langle m \text{ with } B := m.B \oplus (t \mapsto ((x, v) : m.B t)) \rangle \]

\[ m \xrightarrow{\text{WB}} t : \text{Wx} = v \]

\[ m \xrightarrow{t : \text{Wx} = v} m' \]
x86-TSO: Axiomatic-Style

Source Code

\[
\begin{align*}
x & \leftarrow 1; \\
\text{print}(y) & \\
y & \leftarrow 1; \\
\text{print}(x)
\end{align*}
\]

Potential Execution #1

W x=1 \quad W y=1
\quad ↓ \quad \quad ↓
R y=0 \quad R x=1

Potential Execution #2

W x=1 \quad W y=1
\quad ↓ \quad \quad ↓
R y=1 \quad R x=0

\ldots
A Candidate Execution

Pre-execution = Set of Events + Induced Binary Relations (po/data/addr)
Candidate = Pre-execution + Existentially Quantified Relations (co/rf)

Allowed Execution

\[
\begin{align*}
W & \ x=1 \\
& \downarrow \ \ po \\
R & \ y=0 \\
& \downarrow \ \ rf
\end{align*}
\]

\[
\begin{align*}
W & \ y=1 \\
& \downarrow \ \ po \\
R & \ x=1
\end{align*}
\]

Definition of a valid Candidate ("Axiomatic Model"):

\[
poWR = po \cap (W \times R)
\]

\[
uniproc = po-loc \cup (po \setminus poWR)
\]

\[
fr = rf^{-1} \cup co
\]

\[
tso = rf \cup fr \cup co
\]

axiom: acyclic (uniproc \cup tso)

**po** = Program-Order

**rf** = Reads-From
**TSO: Forbidden Execution**

Forbidden Execution

\[ W \times 1 \quad R \times 0 \]

\[ \text{po} \rightarrow \text{fr} \rightarrow \text{rf} \]

- **po** = Program-Order
- **rf** = Reads-From

Axiomatic Model:

\[ \text{poWR} = \text{po} \cap (W \times R) \]

\[ \text{uniproc} = \text{po-loc} \cup (\text{po} \setminus \text{poWR}) \]

\[ \text{fr} = \text{rf}^{-1} ; \text{co} \]

\[ \text{tso} = \text{rf} \cup \text{fr} \cup \text{co} \]

\[ \text{axiom} : \text{acyclic (uniproc} \cup \text{tso}) \]
TSO: Allowed Execution

Allowed Execution

\[
\begin{align*}
W \times x &= 1 \\
R \times y &= 0 \\
po &= \text{Program-Order} \\
rf &= \text{Reads-From} \\
fr &= \text{From-Reads}
\end{align*}
\]

Axiomatic Model:

\[
\begin{align*}
poWR &= po \cap (W \times R) \\
uniproc &= po-loc \cup (po \setminus poWR) \\
fr &= rf^{-1} ; co \\
tso &= rf \cup fr \cup co \\
\text{axiom: acyclic (uniproc} \cup tso)\end{align*}
\]
“user-mode” concurrency

Much work not covered here:

- Fences
- Atomics
- Mixed-size
- Multi-copy atomicity
- Other Architectures: IBM Power, Arm, RISC-V
Instruction Fetch

ESOP2020

Exceptions and Interrupts

with Ohad Kammar

Pagetables and TLBs

Devices and NVME

Future Work . . .
JITs

Just-In-Time Compilation

Source Code

CALL f
CALL g
CALL f

Jump Table

Jump 0x1000
Jump 0x2000

f:

 Compiled Code

CALL g

Optimized code now unsound, have to re-compile!
JITs

JIT: de-opt after executing g

Source Code

Jump Table

Optimized code now unsound, have to re-compile!

Compiled Code
JITs

JIT: re-compile

Source Code

Jump Table

Compiled Code

Optimized code now unsound, have to re-compile!
ARMv8: How to safely modify code?

The write is complete for the shareability domain.

If software requires coherency between instruction execution and memory, it must manage this coherency using Context synchronization events and cache maintenance instructions. The following code sequence can be used to allow a PE to execute code that the same PE has written.

; Coherency example for data and instruction accesses within the same Inner Shareable domain.
; Enter this code with <Wt> containing a new 32-bit instruction,
; to be held in Cacheable space at a location pointed to by Xn.
  STR Wt, [Xn]
  DC CVAU, Xn    ; Clean data cache by VA to point of unification (PoU)
  DSB ISH        ; Ensure visibility of the data cleaned from cache
  IC IVAU, Xn    ; Invalidate instruction cache by VA to PoU
  DSB ISH        ; Ensure completion of the invalidations
  ISB            ; Synchronize the fetched instruction stream
RISC-V/x86/Power: How to?

Similar for IBM Power
Much easier on x86
RISC-V not decided yet . . .
Focus on ARMv8-A for rest of talk . . .
An Instruction Fetching Test

Write \( f = \text{"print(2)"} \)

\[ \text{CALL } f \]

\[ \vdots \]

\[ \text{print(1)} \]

\[ \text{RETURN} \]

\[ \vdots \]

Overwrite code of function \( f \)

Then, Call \( f \)

Memory
Real A64 Assembly

<table>
<thead>
<tr>
<th>Initial state:</th>
<th>0:W0='B l1', 0:X1=f</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 0</td>
<td>f</td>
</tr>
<tr>
<td>STR W0,[X1]</td>
<td>f: B l0</td>
</tr>
<tr>
<td>BL f</td>
<td>l1: MOV X0,#2</td>
</tr>
<tr>
<td></td>
<td>RET</td>
</tr>
<tr>
<td></td>
<td>l0: MOV X0,#1</td>
</tr>
<tr>
<td></td>
<td>RET</td>
</tr>
<tr>
<td>Allowed:</td>
<td>1:X0=1</td>
</tr>
</tbody>
</table>

Relaxed Result Observed in ~99% of experimental runs on multiple devices.
An Architectural Model!

Source Code

Write \( f = \text{"print}(2)\)"

\[
\begin{align*}
\text{CALL } f \\
\vdots \\
\text{print}(1) \\
\text{RETURN} \\
\vdots
\end{align*}
\]

\( f: \)

\[
\begin{align*}
\text{CALL } f \\
\vdots \\
\text{print}(1) \\
\text{RETURN} \\
\vdots
\end{align*}
\]
An Architectural Model!

Source Code

Write \( f = \text{"print(2)"} \)

\[
\begin{align*}
\text{CALL } f \\
\vdots \\
\text{print(1)} \\
\text{RETURN}
\end{align*}
\]

**Thread**

- Fetch Queue
- Abstract icache

**Memory**

- Abstract dcache
- Prefetching
- Stale instructions
- Data buffering

**CALL**

new fetch request

decode

write data

add to icache

per-thread
Unexpected Coherence!

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f = \text{“print(2)”}$</td>
<td>CALL $f$</td>
</tr>
<tr>
<td>\vdots</td>
<td>\vdots</td>
</tr>
<tr>
<td>$f: \begin{align*} &amp; \text{print(1)} \ &amp; \text{print($f$)} \ &amp; \text{RETURN} \end{align*}$</td>
<td>\vdots</td>
</tr>
</tbody>
</table>

If $f$ executes $\text{print(2)}$
Then $\text{print(f)}$ must print the updated memory (2).
Real A64 Assembly

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>STR $W0,[X1]</td>
<td>BL $f$</td>
<td>$f$: B $l0$</td>
</tr>
<tr>
<td>LDR $X1,[X2]$</td>
<td>$l1$: MOV $X0,#2$</td>
<td>$l0$: MOV $X0,#1$ RET</td>
</tr>
<tr>
<td></td>
<td>$l0$: MOV $X0,#1$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>RET</td>
</tr>
</tbody>
</table>

Forbidden: $1: X0 = 2$, $1: X1 = "B l0"$
Other Phenomena

Not Mentioned Here:
- (In)coherence
- Multiple images in I-cache
- Multiple images in D-cache(s)
- Direct Data Intervention
- Speculating cache maintenance
- O/S Migration
- and others ...
Operational Model

Thread

Fetch Queue

Abstract icache

Memory

Abstract dcache

write data

new fetch request

decode

fetch

add to icache

read data

per-thread

global
Operational State

\[ m: \langle ts: tid \mapsto \text{instruction\_tree} \rangle \]

\[ ss: \text{storage\_subsystem} \]

\[ \text{storage\_subsystem} : \langle \text{mem: write list} \]
\[ \text{icache: tid \mapsto write set} \]
\[ \text{dcache: write list} \]
\[ ... \rangle \]
Thread State
Thread State

Sequential ISA Spec
Thread State

Explicit Speculation

Sequential ISA Spec
Operational: Transitions

Transitions:

- Step ISA Spec
- Memory Read/Write
- ... 
- Fetch Request
- Fetch Instruction (from icache)
- Decode Instruction
- ... 
- Update Instruction Cache
- Flow Writes into Memory
- Reset Instruction

*exact names may vary
Flow Writes into Memory An instruction $i$ in the state Perform\_DC($address$, $state\_cont$) can complete if all po-previous DMB ISH and DSB ISH instructions have finished. Action:

1. For the most recent writes $ws$ which are in the same data cache line of minimum size in the abstract data cache as $address$, update the memory with $ws$;
2. Remove all those writes from the abstract data cache.
3. Set the state of $i$ to Plain($state\_cont$).
let flat_propagate_dc params state _cmr addr =
  (* remove all to that cacheline from buffer *)
let (overlapping, fetch_buf) =
  List.partition
  (write_overlaps_with_addr (cache_line_fp addr))
  state.flat_ss_fetch_buf
in
  (* flow the overlapping writes into memory *)
List.foldr
  (fun write state ->
    flat_write_to_memory params state write)
  ( <| state with flat_ss_fetch_buf = fetch_buf |>)
overlapping
Axiomatic-Style Model

```plaintext
let iseq = [W];(wco&scl);[DC];
  (wco&scl);[IC]

(* Observed-by *)
let obs = rfe | fr | wco
  | irf | (ifr;iseq)

(* Fetch-ordered-before *)
let fob = [IF]; fpo; [IF]
  | [IF]; fe
  | [ISB]; fe⁻¹; fpo

(* Dependency-ordered-before *)
let dob = addr | data
  | ctrl; [W]
  | (ctrl | (addr; po)); [ISB]
  | addr; po; [W]
  | (addr | data); rfi

(* Atomic-ordered-before *)
let aob = rmw
  | [range(rmw)]; rfi; [A|Q]

(* Barrier-ordered-before *)
let bob = [R|W]; po; [dmb.sy]
  | [dmb.sy]; po; [R|W]
  | [L]; po; [A]
  | [R]; po; [dmb.ld]

(* Cache-op-ordered-before *)
let cob = [R|W]; (po&scl); [DC]
  | [DC]; (po&scl); [DC]

(* Ordered-before *)
let ob = obs|fob|dob|aob|bob|cob

(* Internal visibility requirement *)
acyclic (po-loc|fr|co|rf) as internal

(* External visibility requirement *)
acyclic ob as external

(* Atomic *)
empty rmw & (fre; coe) as atomic

(* Constrained unpredictable *)
let cff = ([W];loc;[IF]) \ ob⁻¹ \ (co;iseq;ob+)
cff_bad cff ≡ CU
```
Axiomatic-Style Model

```haskell
let iseq = [W];(wco&scl);[DC];
     (wco&scl);[IC]

(* Observed-by *)
let obs = rfe | fr | wco
     | irf | (ifr;iseq)

(* Fetch-ordered-before *)
let fob = [IF]; fpo; [IF]
     | [IF]; fe
     | [ISB]; fe⁻¹; fpo

(* Dependency-ordered-before *)
let dob = addr | data
     | ctrl; [W]
     | (ctrl | (addr; po)); [ISB]
     | addr; po; [W]
     | (addr | data); rfi

(* Atomic-ordered-before *)
let aob = rmw
     | [range(rmw)]; rfi; [A|Q]

(* Barrier-ordered-before *)
let bob = [R|W]; po; [dmb.sy]
     | [dmb.sy]; po; [R|W]
     | [L]; po; [A]
     | [R]; po; [dmb.ld]
     | [dmb.ld]; po; [R|W]
     | [A|Q]; po; [R|W]
     | [W]; po; [dmb.st]
     | [dmb.st]; po; [W]
     | [R|W]; po; [L]
     | [R|W|F|DC|IC]; po; [dsb.ish]
     | [dsb.ish]; po; [R|W|F|DC|IC]
     | [dmb.sy]; po; [DC]

(* Cache-op-ordered-before *)
let cob = [R|W]; (po&scl); [DC]
     | [DC]; (po&scl); [DC]

(* Ordered-before *)
let ob = obs|fob|dob|aob|bob|cob

(* Internal visibility requirement *)
acyclic (po-loc|fr|co|rf) as internal

(* External visibility requirement *)
acyclic ob as external

(* Atomic *)
empty rmw & (fre; coe) as atomic

(* Constrained unpredictable *)
let cff = ([W];loc;[IF]) \ 
       ob⁺⁻¹ \ (co;iseq;ob+)

cff_bad cff ≡ CU
```

35/41
Axiomatic ifetch: an example

Initial state:
W0="B l1"
X1=f

Forbidden: X0=1

STR W0,[X1] // (b)
DC CVAU,X1 // (d)
DSB ISH
IC IVAU,X1 // (h)
DSB ISH
ISB // (l)
BL f // (m)
A Forbidden Instruction Fetch

let iseq = [W]; (wco&scl);[DC];
(wco&scl);[IC]

(* Observed-by *)
let obs = irf | (ifr;iseq)

(* Fetch-ordered-before *)
let fob = [IF]; fpo; [IF]
| [IF]; fe
| [ISB]; fe⁻¹; fpo

(* Barrier-ordered-before *)
let bob = . . .
| [R|W|F|DC|IC]; po; [dsb.ish]
| [dsb.ish]; po; [R|W|F|DC|IC]

(* Ordered-before *)
let ob = obs | fob | bob

(* External visibility requirement *)
acyclic ob
A Forbidden Instruction Fetch

```
let iseq = [W];(wco&scl);[DC];
     (wco&scl);[IC]

(* Observed-by *)
let obs = irf | (ifr;iseq)

(* Fetch-ordered-before *)
let fob = [IF]; fpo; [IF]
     | [IF]; fe
     | [ISB]; fe⁻¹; fpo

(* Barrier-ordered-before *)
let bob = ...
     | [R|W|F|DC|IC]; po; [dsb.ish]
     | [dsb.ish]; po; [R|W|F|DC|IC]

(* Ordered-before *)
let ob = obs | fob | bob

(* External visibility requirement *)
acyclic ob
```
A Forbidden Instruction Fetch

let iseq = [W]; (wco&scl); [DC];
   (wco&scl); [IC]

(* Observed-by *)
let obs = irf | (ifr; iseq)

(* Fetch-ordered-before *)
let fob = [IF]; fpo; [IF]
 | [IF]; fe
 | [ISB]; fe⁻¹; fpo

(* Barrier-ordered-before *)
let bob = ...
 | [R|W|F|DC|IC]; po; [dsb.ish]
 | [dsb.ish]; po; [R|W|F|DC|IC]

(* Ordered-before *)
let ob = obs | fob | bob

(* External visibility requirement *)
acyclic ob
A Forbidden Instruction Fetch

let iseq = [W]; (wco&scl); [DC];
             (wco&scl); [IC]

(* Observed-by *)
let obs = irf | (ifr;iseq)

(* Fetch-ordered-before *)
let fob = [IF]; fpo; [IF]
        | [IF]; fe
        | [ISB]; fe⁻¹; fpo

(* Barrier-ordered-before *)
let bob = ...
        | [R|W|F|DC|IC]; po; [dsb.ish]
        | [dsb.ish]; po; [R|W|F|DC|IC]

(* Ordered-before *)
let ob = obs | fob | bob

(* External visibility requirement *)
acyclic ob
A Forbidden Instruction Fetch

```
let iseq = [W];(wco&scl);[DC]; (wco&scl);[IC]

(* Observed-by *)
let obs = irf | (ifr;iseq)

(* Fetch-ordered-before *)
let fob = [IF]; fpo; [IF]
  | [IF]; fe
  | [ISB]; fe^{-1}; fpo

(* Barrier-ordered-before *)
let bob = ...
  | [R|W|F|DC|IC]; po; [dsb.ish]
  | [dsb.ish]; po; [R|W|F|DC|IC]

(* Ordered-before *)
let ob = obs | fob | bob

(* External visibility requirement *)
acyclic ob
```
A Forbidden Instruction Fetch

let iseq = [W];(wco&scl);[DC];
         (wco&scl);[IC]

(* Observed-by *)
let obs = irf | (ifr;iseq)

(* Fetch-ordered-before *)
let fob = [IF]; fpo; [IF]
          | [IF]; fe
          | [ISB]; fe⁻¹; fpo

(* Barrier-ordered-before *)
let bob = ... |
          | [R|W|F|DC|IC]; po; [dsb.ish]
          | [dsb.ish]; po; [R|W|F|DC|IC]

(* Ordered-before *)
let ob = obs | fob | bob

(* External visibility requirement *)
acyclic ob
A Forbidden Instruction Fetch

let iseq = [W];(wco&sc);[DC];
(wco&sc);[IC]

(* Observed-by *)
let obs = irf | (ifr;iseq)

(* Fetch-ordered-before *)
let fob = [IF]; fpo; [IF]
| [IF]; fe
| [ISB]; fe⁻¹; fpo

(* Barrier-ordered-before *)
let bob = ...
| [R|W|F|DC|IC]; po; [dsb.ish]
| [dsb.ish]; po; [R|W|F|DC|IC]

(* Ordered-before *)
let ob = obs | fob | bob

(* External visibility requirement *)
acyclic ob
A Forbidden Instruction Fetch

let iseq = [W];(wco&scl);[DC];
(wco&scl);[IC]

(* Observed-by *)
let obs = irf | (ifr;iseq)

(* Fetch-ordered-before *)
let fob = [IF]; fpo; [IF]
| [IF]; fe
| [ISB]; fe⁻¹; fpo

(* Barrier-ordered-before *)
let bob = ...
| [R|W|F|DC|IC]; po; [dsb.ish]
| [dsb.ish]; po; [R|W|F|DC|IC]

(* Ordered-before *)
let ob = obs | fob | bob

(* External visibility requirement *)
acyclic ob
A Forbidden Instruction Fetch

```
let iseq = [W];(wco&scl);[DC];
  (wco&scl);[IC]

(* Observed-by *)
let obs = irf | (ifr;iseq)

(* Fetch-ordered-before *)
let fob = [IF]; fpo; [IF]
  | [IF]; fe
  | [ISB]; fe⁻¹; fpo

(* Barrier-ordered-before *)
let bob = ...
  | [R|W|F|DC|IC]; po; [dsb.ish]
  | [dsb.ish]; po; [R|W|F|DC|IC]

(* Ordered-before *)
let ob = obs | fob | bob

(* External visibility requirement *)
acyclic ob
```
A Forbidden Instruction Fetch

```
a: fetch  b: write f = B  l1

c: fetch  d: DC

e: fetch  f: DSB

g: fetch  h: IC

```

```
let iseq = [W]; (wco&scl); [DC]; (wco&scl); [IC]

(* Observed-by *)
let obs = irf | (ifr; iseq)

(* Fetch-ordered-before *)
let fob = [IF]; fpo; [IF]
| [IF]; fe
| [ISB]; fe^{-1}; fpo

(* Barrier-ordered-before *)
let bob = ...
| [R|W|F|DC|IC]; po; [dsb.ish]
| [dsb.ish]; po; [R|W|F|DC|IC]

(* Ordered-before *)
let ob = obs | fob | bob

(* External visibility requirement *)
acyclic ob
```
A Forbidden Instruction Fetch

a: fetch          b: write f = B  l1

c: fetch          d: DC

e: fetch          f: DSB

fob

g: fetch          h: IC

obs

i: fetch          j: DSB

obs

k: fetch          l: ISB

j

m: fetch f = B   l0

fob
A Forbidden Instruction Fetch

\[
\begin{align*}
a : & \text{fetch} & \quad & b : \text{write } f = B \ l 1 \\
c : & \text{fetch} & \quad & d : \text{DC} \\
e : & \text{fetch} & \quad & f : \text{DSB} \\
g : & \text{fetch} & \quad & h : \text{IC} \\
i : & \text{fetch} & \quad & j : \text{DSB} \\
k : & \text{fetch} & \quad & l : \text{ISB} \\
m : & \text{fetch } f = B \ l 0 \\
\end{align*}
\]

\[
\begin{align*}
\text{let } \ \text{iseq} &= [W]; (\text{wco\&scl}); [DC]; (\text{wco\&scl}); [IC] \\
& \quad (* \text{Observed-by} *) \\
\text{let } \ \text{obs} &= \text{ifr} \ | \ (\text{ifr}; \text{iseq}) \\
& \quad (* \text{Fetch-ordered-before} *) \\
\text{let } \ \text{fob} &= [\text{IF}]; \text{fpo}; [\text{IF}] \\
& \quad \ | \ [\text{IF}]; \text{fe} \\
& \quad \ | \ [\text{ISB}]; \text{fe}^{-1}; \text{fpo} \\
& \quad (* \text{Barrier-ordered-before} *) \\
\text{let } \ \text{bob} &= \ldots \\
& \quad \ | \ [R|W|F|DC|IC]; \text{po}; [\text{dsb.ish}] \\
& \quad \ | \ [\text{dsb.ish}]; \text{po}; [R|W|F|DC|IC] \\
& \quad (* \text{Ordered-before} *) \\
\text{let } \ \text{ob} &= \text{obs} \ | \ \text{fob} \ | \ \text{bob} \\
& \quad (* \text{External visibility requirement} *) \\
\text{acyclic } \text{ob}
\end{align*}
\]
A Forbidden Instruction Fetch

a: fetch
b: write f=B l1

c: fetch
d: DC

e: fetch
f: DSB

(* Observed-by *)
let obs = irf | (ifr;iseq)

(* Fetch-ordered-before *)
let fob = [IF]; fpo; [IF]
| [IF]; fe
| [ISB]; fe^{-1}; fpo

(* Barrier-ordered-before *)
let bob = ...
| [R|W|F|DC|IC]; po; [dsb.ish]
| [dsb.ish]; po; [R|W|F|DC|IC]

(* Ordered-before *)
let ob = obs | fob | bob

(* External visibility requirement *)
acyclic ob

Thread 0
Modelling Process

Create Model → Write Tests → Run Tests → Talk to Architects → Create Model
Validation

Validating the model:
- approx. 35 hand-written tests.
- approx. 1500 auto-generated tests.

Ran on multiple devices and compared results to our models:
- Found some hardware bugs;
- Many places hardware not as relaxed as architecture allows!
Future

- Exceptions and Interrupts
- Pagetables and TLBs
- Devices, DMA, Non-Volatile Memory
End

So far:

- Re-cap “relaxed-memory” / x86-TSO.
  - Operational & “Axiomatic” models
- JIT usage
- Arm self-modifying code
- ARMv8 Architectural Operational Model
- ARMv8 Axiomatic Model
- Modelling and validation