Promising-ARM/RISC-V

~~a simpler and faster operational concurrency model~~

an equivalent simpler and faster presentation of ARMv8/RISC-V

Christopher Pulte [1] Jean Pichon-Pharabod [1] Jeehoon Kang [2] Sung-Hwan Lee [3] Chung-Kil Hur [3]

24 June 2019

[1]University of Cambridge

[2]Korea Advanced Institute of Science and Technology

[3]Seoul National University

```
void produce(int v) {
  D = v;
  F = 1;
}

int consume() {
  while (1) {
    if (F) break;
  }
  return D;
}
```

```
produce(42) || d = consume()
```

Initially: D = F = 0

Finally: d = 42

```
void produce(int v) {
  D = v;
  F = 1;
}

int consume() {
  while (1) {
    if (F) break;
  }
  return D;
}
```

compiler re-ordering, hardware out-of-order execution

re-ordering past control dependency

Initially: D = F = 0

Finally: d = 42

or: d = 0

```
produce(42) || d = consume()
```

```
void produce(int v) {
  D = v;
  F[rel] = 1;
}

int consume() {
  while (1) {
    if (F[acq]) break;
  }
  return D;
}
```

```
produce(42) || d = consume()
```

Initially: D = F = 0

Finally: d = 42

or: d = 0

```
void produce(int v) {
  D = v;
  F[rel] = 1;
}

int consume() {
  while (1) {
    if (F[acq]) break;
  }
  return D;
}
```

```
produce(42) || d = consume()
```

Need precise, simple semantics and tool support.

Axiomatic

- official reference model
+ abstract, concise
- not incremental: global axioms

Flat operational

- proved equivalent
+ incremental
+ ISA, mixed-size support
+ closer relation to H/W
- complex

Axiomatic

- official reference model
+ abstract, concise
- not incremental: global axioms

Flat operational

- proved equivalent
+ incremental
+ ISA, mixed-size support
+ closer relation to H/W
- complex

Promising-ARM/RISC-V

inspired by Promising C11 [Kang et al]

+ simple, incremental
+ Coq equivalence proof with Axiomatic (excl. ISA model)
+ ISA support
- no mixed-size support yet
+ fast enough for checking data structure examples

Model overview

Idea 1: out-of-order read execution by reading from message history

T1

```
store D := 42
store F := 1
```

T2

```
r1 = load F        //reads 1
if (r1 == 1) then
  r2 = load D      //reads 0
else
  ..
```

Idea 1: out-of-order read execution by reading from message history

T1

```
store D := 42
store F := 1
```

T2

```
r1 = load F        //reads 1
if (r1 == 1) then
  r2 = load D      //reads 0
else
  ..
```

T1.regs: ...

T1.promises: ...

...

T2.regs: $r1 \mapsto 0@0, r2 \mapsto 0@0 \ldots$

T2.promises: ...

...

memory: $(init)@0$

Idea 1: out-of-order read execution by reading from message history

T1

```
store D := 42
store F := 1
```

T2

```
r1 = load F        //reads 1
if (r1 == 1) then
  r2 = load D      //reads 0
else
  ..
```

T1.regs: ...

T1.promises: ...

...

T2.regs: $r1 \mapsto 0@0, r2 \mapsto 0@0 \ldots$

T2.promises: ...

...

memory: (init)@0, (D = 42)@1

Idea 1: out-of-order read execution by reading from message history

T1

```
store D := 42
store F := 1
```

T2

```
r1 = load F        //reads 1
if (r1 == 1) then
  r2 = load D      //reads 0
else
  ..
```

T1.regs: ...

T1.promises: ...

...

T2.regs: $r1 \mapsto 0@0, r2 \mapsto 0@0 \ldots$

T2.promises: ...

...

memory: $(\text{init})@0, (D = 42)@1, (F = 1)@2$

Idea 1: out-of-order read execution by reading from message history

T1

```
store D := 42
store F := 1
```

T2

```
r1 = load F        //reads 1
if (r1 == 1) then
  r2 = load D      //reads 0
else
  ..
```

T1.regs: ...

T1.promises: ...

...

T2.regs: r1 ↦ 1@0, r2 ↦ 0@0 ...

T2.promises: ...

...

memory: (init)@0, (D = 42)@1, (F = 1)@2

Idea 1: out-of-order read execution by reading from message history

T1

```
store D := 42
store F := 1
```

T2

```
r1 = load F        //reads 1
if (r1 == 1) then
  r2 = load D      //reads 0
else
  ..
```

T1.regs: ...

T1.promises: ...

...

T2.regs: $r1 \mapsto 1@0, r2 \mapsto 0@0 \ldots$

T2.promises: ...

...

memory: $(\text{init})@0, (D = 42)@1, (F = 1)@2$

Idea 1: out-of-order read execution by reading from message history

T1

```
store D := 42
store F := 1
```

T2

```
r1 = load F        //reads 1
if (r1 == 1) then
  r2 = load D      //reads 0
else
  ..
```

T1.regs: ...

T1.promises: ...

...

T2.regs: $r1 \mapsto 1@0, r2 \mapsto 0@0 \ldots$

T2.promises: ...

...

memory: (init)@0, (D = 42)@1, (F = 1)@2

Idea 1: out-of-order read execution by reading from message history

T1

```
store D := 42
store F := 1
```

T2

```
r1 = load F        //reads 1
if (r1 == 1) then
  r2 = load D      //reads 0
else
  ..
```

T1.regs: ...

T1.promises: ...

...

T2.regs: $r1 \mapsto 1@0, r2 \mapsto 0@0 \ldots$

T2.promises: ...

...

memory: (init)@0, (D = 42)@1, (F = 1)@2

Idea 2: ordering reads with views.

A view is a timestamp of a "seen" write.

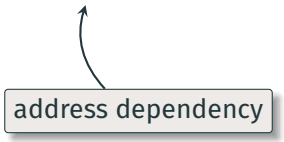
regstate: Reg $\mapsto$ Val $\times$ View

T1

```
store D := 42
store F := 1
```

T2

```
r1 = load F              //reads 1
if (r1 == 1) then
  r2 = load (D+r1-r1)    //reads 0
else
  ..
```



Idea 2: ordering reads with views.

A view is a timestamp of a "seen" write.

regstate: Reg ↦ Val × View

T1

```
store D := 42
store F := 1
```

T2

```
r1 = load F              //reads 1
if (r1 == 1) then
  r2 = load (D+r1-r1)    //reads 0
else
  ..
```

T1.regs: ...

T1.promises: ...

...

T2.regs: r1 ↦ 0@0, r2 ↦ 0@0 ...

T2.promises: ...

...

memory: (init)@0, (D = 42)@1, (F = 1)@2

Idea 2: ordering reads with views.

A view is a timestamp of a "seen" write.

regstate: $\mathtt{Reg} \mapsto \mathtt{Val} \times \mathtt{View}$

T1

```
store D := 42
store F := 1
```

T2

```
r1 = load F              //reads 1
if (r1 == 1) then
  r2 = load (D+r1-r1)    //reads 0
else
  ..
```

T1.regs: ...

T1.promises: ...

...

T2.regs: r1 ↦ 0@0, r2 ↦ 0@0 ...

T2.promises: ...

...

memory: (init)@0, (D = 42)@1, (F = 1)@2

Idea 2: ordering reads with views.

A view is a timestamp of a "seen" write.

regstate: Reg $\mapsto$ Val $\times$ View

T1

```
store D := 42
store F := 1
```

T2

```
r1 = load F              //reads 1
if (r1 == 1) then
  r2 = load (D+r1-r1)    //reads 0
else
  ..
```

T1.regs: ...

T1.promises: ...

...

T2.regs: r1 $\mapsto$ 1@2, r2 $\mapsto$ 0@0 ...

T2.promises: ...

...

memory: (init)@0, (D = 42)@1, (F = 1)@2

Idea 2: ordering reads with views.

A view is a timestamp of a "seen" write.

regstate: Reg ↦ Val × View

T1

```
store D := 42
store F := 1
```

T2

```
r1 = load F              //reads 1
if (r1 == 1) then
  r2 = load (D+r1-r1)    //reads 0
else
  ..
```

T1.regs: ...

T1.promises: ...

...

T2.regs: r1 ↦ 1@2, r2 ↦ 0@0 ...

T2.promises: ...

...

memory: (init)@0, (D = 42)@1, (F = 1)@2

Idea 2: ordering reads with views.

A view is a timestamp of a "seen" write.

regstate: Reg ↦ Val × View

T1

```
store D := 42
store F := 1
```

T2

```
r1 = load F              //reads 1
if (r1 == 1) then
  r2 = load (D+r1-r1)    //reads 0
else
  ..
```

T1.regs: ...

T1.promises: ...

...

T2.regs: r1 ↦ 1@2, r2 ↦ 0@0 ...

T2.promises: ...

...

memory: (init)@0, (D = 42)@1, (F = 1)@2

Idea 3: out-of-order writes with promises.
A thread can promise any write at any time,
if it can later fulfil the promise.

T1

```
store D := 42
store F := 1
```

T2

```
r1 = load F                 //reads 1
if (r1 == 1) then
  r2 = load (D+r1-r1)       //reads 0
else
  ..
```

Idea 3: out-of-order writes with promises.

A thread can promise any write at any time, if it can later fulfil the promise.

T1

```
store D := 42
store F := 1
```

T2

```
r1 = load F              //reads 1
if (r1 == 1) then
  r2 = load (D+r1-r1)    //reads 0
else
  ..
```

T1.regs: ...
T1.promises: ∅
...

T2.regs: r1 ↦ 0@0, r2 ↦ 0@0 ...
T2.promises: ∅
...

memory: (init)@0

Idea 3: out-of-order writes with promises.

A thread can promise any write at any time, if it can later fulfil the promise.

T1

```
store D := 42
store F := 1
```

T2

```
r1 = load F                 //reads 1
if (r1 == 1) then
  r2 = load (D+r1-r1)       //reads 0
else
  ..
```

T1.regs: ...

T1.promises: (F = 1)@1

...

T2.regs: $r1 \mapsto 0@0, r2 \mapsto 0@0 \ldots$

T2.promises: $\emptyset$

...

memory: (init)@0, (F = 1)@1

Idea 3: out-of-order writes with promises.

A thread can promise any write at any time, if it can later fulfil the promise.

T1

```
store D := 42
store F := 1
```

T2

```
r1 = load F                 //reads 1
if (r1 == 1) then
  r2 = load (D+r1-r1)       //reads 0
else
  ..
```

T1.regs: ...

T1.promises: (F = 1)@1

...

T2.regs: r1 ↦ 0@0, r2 ↦ 0@0 ...

T2.promises: ∅

...

memory: (init)@0, (F = 1)@1, (D = 42)@2

Idea 3: out-of-order writes with promises.

A thread can promise any write at any time, if it can later fulfil the promise.

T1

```
store D := 42
store F := 1
```

T2

```
r1 = load F                 //reads 1
if (r1 == 1) then
  r2 = load (D+r1-r1)       //reads 0
else
  ..
```

T1.regs: ...
T1.promises: ∅
...

T2.regs: r1 ↦ 0@0, r2 ↦ 0@0 ...
T2.promises: ∅
...

memory: (init)@0, (F = 1)@1, (D = 42)@2

Idea 3: out-of-order writes with promises.
A thread can promise any write at any time,
if it can later fulfil the promise.

T1

```
store D := 42
store F := 1
```

T2

```
r1 = load F              //reads 1
if (r1 == 1) then
  r2 = load (D+r1-r1)    //reads 0
else
  ..
```

T1.regs: ...
T1.promises: ∅
...

T2.regs: r1 ↦ 1@1, r2 ↦ 0@0 ...
T2.promises: ∅
...

memory: (init)@0, (F = 1)@1, (D = 42)@2

Idea 3: out-of-order writes with promises.

A thread can promise any write at any time, if it can later fulfil the promise.

T1

```
store D := 42
store F := 1
```

T2

```
r1 = load F                 //reads 1
if (r1 == 1) then
  r2 = load (D+r1-r1)       //reads 0
else
  ..
```

T1.regs: ...
T1.promises: $\emptyset$
...

T2.regs: $r1 \mapsto 1@1, r2 \mapsto 0@0 \ldots$
T2.promises: $\emptyset$
...

memory: $(\text{init})@0, (F = 1)@1, (D = 42)@2$

Full model details include

- **write ordering**: with views — $T.v_{\mathsf{wm}}, T.v_{\mathsf{wp}}$
- **certification**: thread-locally prevent executions with unfulfilled promises

- **coherence** — $T.\mathsf{coh}$
- **write forwarding** — $T.\mathsf{fwdb}$
- **barriers, release/acquire**: uniformly with views — $T.v_{\mathsf{rm}}, T.v_{\mathsf{rp}}, T.v_{\mathsf{wm}}, T.v_{\mathsf{wp}}, T.v_{\mathsf{rel}}$
- **load/store exclusive instructions** — $T.\mathsf{exclb}$

Executable tool

Executable tool for interactive and exhaustive execution of ARMv8 or RISC-V concurrent assembly programs.

Building on Sail, Sail ISA models, rmem

[Sarkar et al 2011/12, Gray et al 2015, Flur et al 2016/17, Pulte et al 2018, Armstrong et al 2019]

- **exhaustively:** enumerate all possible final states allowed by model, each with witnessing trace
- **interactively:** interactively replay witnessing trace for debugging

Optimisation 1: promises first (optimisation proved in Coq)

For every trace tr, there is an equivalent trace tr' s.th:

$$tr' = \underbrace{\_ \xrightarrow{t_1} \_ \xrightarrow{t_2} \_ \xrightarrow{t_3} \_ \ldots}_{promise} \underbrace{\_ \xrightarrow{t_n} \_ \xrightarrow{t_{n+1}} \_ \xrightarrow{t_{n+2}} \_ \ldots}_{non-promise}$$

Optimisation 2: explore final thread states in parallel

Executable tool

Run text book concurrent data structure/lock implementations:

1. **write data structure in C++ or Rust**
   write test code, testing operations and logging data
2. **compile with gcc/llvm -O3** to ARMv8 assembly,
   map assembly into litmus format with script
3. **run tool to enumerate final states**

Examples

```
class Deque {
  std::atomic<uint64_t> bottom;
  std::atomic<uint64_t> top;
  int64_t buffer[DequeSize];

public:
  Deque(): bottom(0), top(0) { }

  // -1 if full
  int64_t try_push(int64_t N, int64_t
  // -1 if empty, -2 if lost
  int64_t try_pop(int64_t N, int64_t&
  // -1 if empty, -2 if lost
  int64_t try_steal(int64_t N, int64_t
};


int64_t Deque::try_steal(int64_t N, in
  uint64_t t = this->top.load(std::mem

  std::atomic_thread_fence(std::memory_

  uint64_t b = this->bottom.load(std::
  int64_t len = static_cast<int64_t>(b

  if (len <= 0) {
    return -1; // empty
  }

  data = this->buffer[t % N];

  bool is_successful = this->top.compare_exchange_strong
    (t, t + 1, std::memory_order_release);
  return (is_successful ? 0 : -2); // success or lost
}
```

- spinlock variants in C++ and Rust
- single-producer-single-consumer circular queue
- single-producer-multiple-consumer circular queue
- ticket lock
- treiber stack in C++ and Rust variants
- Chase-Lev dequeue
- Michael & Scott queue

```
int64_t Deque::try_push(int64_t N, int64_t data) {
  uint64_t b = this->bottom.load(std::memory_order_relaxed);
  uint64_t t = this->top.load(std::memory_order_acquire);

  int64_t len = static_cast<int64_t>(b - t);
                                                    ull


                                                    N] = data;
                                                    e(b + 1, std::memory_order_release);



                                                    op(int64_t N, int64_t& data) {
                                                    ->bottom.load(std::memory_order_relaxed);
                                                    e(b - 1, std::memory_order_relaxed);

                                                    d_fence(std::memory_order_seq_cst);

                                                    ->top.load(std::memory_order_relaxed);
                                                    tic_cast<int64_t>(b - t);


                                                    ore(b, std::memory_order_relaxed);
                                                    mpty


                                                    er[(b - 1) % N];

  if (len > 1) {
    return 0;
  }

  bool is_successful = this->top.compare_exchange_strong
    (t, t + 1, std::memory_order_acq_rel);
  this->bottom.store(b, std::memory_order_relaxed);
  return (is_successful ? 0 : -2); // success or lost
}
```

Evaluation

Test	Promising	Flat
Linux Spinlock (asm)-7	0.61	9108.53
Linux Spinlock (cpp)-3	6.58	1472.74
Linux Spinlock (Rust)-3	4.88	52.52
Single-producer-single-consumer-3-3	1.36	249.26
Single-producer-multiple-consumers-3-3-3	71.12	ooT
Ticket lock/(opt)-3	18.08 / 20.13	ooT / ooT
Treiber stack (cpp)/(opt)-100-010-010	0.42 / 0.42	2144.52 / 5943.50
Treiber stack (cpp)/(opt)-100-100-010	8.70 / 8.70	ooT / ooT
Treiber stack (cpp)/(opt)-210-011-000	615.41 / 637.98	ooT / ooT
Treiber stack (Rust)-100-010-010	0.39	77.21
Treiber stack (Rust)-100-100-010	7.30	8940.03
Treiber stack (Rust)-210-011-000	522.19	ooT
Chase-Lev dequeue/(opt)-100-1-0	0.30 / 0.30	2.93 / 2.97
Chase-Lev dequeue/(opt)-110-1-0	0.44 / 0.44	1042.88 / 1114.39
Chase-Lev dequeue/(opt)-211-2-1	28.55 / 111.54	ooT / ooT
Chase-Lev dequeue/(opt)-100-000-000	1.34 / 2.95	2983.11 / ooT
Chase-Lev dequeue/(opt)-100-010-000	2.55 / 5.66	ooT / ooT
Chase-Lev dequeue/(opt)-110-100-010	2108.12 / ooT	ooT / ooT

ooT: out of time

Conclusion

Promising-ARM/RISC-V:

- simpler operational concurrency model
- interactive checking and debugging tool
- promising performance results

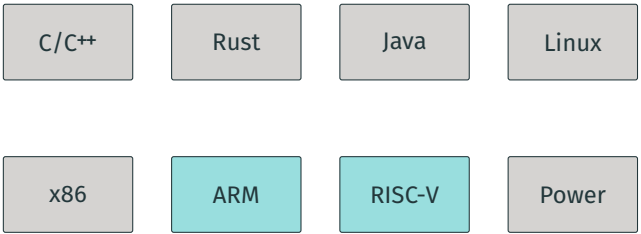
Promising and other rmem models:
`https://github.com/rems-project/rmem`

Sail and Sail models:
`https://github.com/rems-project/sail`

try model in online interface:
`https://www.cl.cam.ac.uk/~pes20/rmem`

Thank you

Why ARMv8/RISC-V?



Language concurrency models:
no accepted thin-air free semantics

Machine concurrency models:
better understood,
compiler-independent tools
that support hand-written assembly

ARMv8: widely used, subtle concurrency semantics
RISC-V: recently adopted similar model

Out-of-order writes

T1

```
store D := 42
store[rel] F := 1
```

T2

```
r1 = load F              //reads 1
if (r1 == 1) then
  r2 = load (D+r1-r1)    //reads 0
else
  ..
```

Write ordering with views

Out-of-order writes

T1

```
store D := 42
store[rel] F := 1
```

T2

```
r1 = load F              //reads 1
if (r1 == 1) then
  r2 = load (D+r1-r1)    //reads 0
else
  ..
```

T1.regs:	…
T1.promises:	∅
T1.v_{wm}:	0
…	

T2.regs:	…
T2.promises:	…
T2.v_{wm}:	…
…	

memory: (init)@0

Write ordering with views

Out-of-order writes

T1

```
store D := 42
store[rel] F := 1
```

T2

```
r1 = load F                //reads 1
if (r1 == 1) then
  r2 = load (D+r1-r1)      //reads 0
else
  ..
```

T1.regs:	…
T1.promises:	(F = 1)@1
T1.v_{wm}:	0
…	

T2.regs:	…
T2.promises:	…
T2.v_{wm}:	…
…	

memory: (init)@0, (F = 1)@1

Write ordering with views

Out-of-order writes

T1

```
store D := 42
store[rel] F := 1
```

T2

```
r1 = load F              //reads 1
if (r1 == 1) then
  r2 = load (D+r1-r1)    //reads 0
else
  ..
```

T1.regs: …
T1.promises: (F = 1)@1
T1.v_{wm}: 2
…

T2.regs: …
T2.promises: …
T2.v_{wm}: …
…

memory: (init)@0, (F = 1)@1, (D = 42)@2

Write ordering with views

Out-of-order writes

T1

```
store D := 42
store[rel] F := 1
```

T2

```
r1 = load F                 //reads 1
if (r1 == 1) then
  r2 = load (D+r1-r1)       //reads 0
else
  ..
```

T1.regs: …
T1.promises: (F = 1)@1
T1.v_{wm}: 2
…

T2.regs: …
T2.promises: …
T2.v_{wm}: …
…

memory: (init)@0, (F = 1)@1, (D = 42)@2

Prevent executions with unfulfilled promises
with certification.

For every step by thread T, do simple thread-local check: ensure
there exists trace by T executing in program-order,
alone, under current memory fulfilling all its promises.