A Full model

We continue the informal description of the model from §4 to extend the model with release/acquire instructions, weaker barriers, and load/store exclusive instructions before giving the full model definition including these.

A.1 Release/acquire accesses and weak barriers

In addition to the full `dmb.sy` barriers and dependencies, ordering can also be created in ARMv8/RISC-V using weaker barriers:

- Release and acquire are "half-barriers" that introduce ordering in one direction: a store release is ordered with respect to all program-order previous memory accesses, and a load acquire with all program-order later ones. Moreover, a strong load acquire (acq, not wacq) is ordered with respect to all program-order-earlier strong store releases (rel, not wrel). Only RISC-V features weak releases.
- `dmb.ld` orders any program-order earlier loads with any program-order later memory access.
- `dmb.st` creates ordering from any program-order earlier store to any program-order later store.
- `isb` orders any load i before any program-order later store i', if there is a conditional branch between i and i' whose condition depends on i, or if there is a memory access between i and i' whose address depends on i.
- The RISC-V equivalent of `dmb.sy`, `dmb.ld`, and `dmb.st` are $\texttt{fence}_{\texttt{RW,RW}}$, $\texttt{fence}_{\texttt{R,RW}}$, and $\texttt{fence}_{\texttt{W,W}}$, respectively. `isb` has no equivalent in RISC-V: the fence `fence.i` does not consider control and address-po dependencies. Since we do not model self-modifying code, `fence.i` would be a no-op in this model. RISC-V has some additional barriers, such as $\texttt{fence}_{\texttt{W,R}}$ and `fence.tso`, which work analogously to the ARMv8 barriers seen so far (see §A.3).

$$\begin{array}{l||l} (a)\ \texttt{store [x] 37;} & (c)\ \texttt{r}_1 := \texttt{load}_{\texttt{acq}}\ \texttt{[y];}\ \texttt{// 42} \\ (b)\ \texttt{store}_{\texttt{rel}}\ \texttt{[y] 42} & (d)\ \texttt{r}_2 := \texttt{load [x]}\ \texttt{// 0} \end{array}$$
$$r_1 = 42 \wedge r_2 = 0 \text{ forbidden}$$

Release/acquire We return to the earlier MP example. Turning b into a release write orders b after a. Making c an acquire load orders d after c. Together, this forbids the behaviour in which c reads 42 and d 0. Assume Thread 1 promises $y = 42$ before $x = 37$, at timestamp 1. Executing a places $x = 37$ at timestamp 2, and sets $\mathsf{v}_{\mathsf{wOld}} = 2$.

ρ1 A store release includes in its pre-view the view of all previous memory accesses, captured by $\mathsf{v}_{\mathsf{rOld}}$ and $\mathsf{v}_{\mathsf{wOld}}$.

Therefore, after a, the pre-view of b is 2, and it cannot fulfil the promise at timestamp 1. So, in the example, when c reads $y = 42$, memory must instead be $[1{:}\ \langle x := 37\rangle_1, 2{:}\ \langle y := 42\rangle_1]$ thereby setting c's post-view to 2.

ρ2 The load acquire, symmetrically to the store release, merges its post-view into $\mathsf{v}_{\mathsf{rNew}}$ and $\mathsf{v}_{\mathsf{wNew}}$, affecting the pre-view of all future loads and store.

Therefore, c sets $\mathsf{v}_{\mathsf{rNew}}$ and $\mathsf{v}_{\mathsf{wNew}}$ to 2. Since in this state d is constrained by timestamp $\mathsf{v}_{\mathsf{rNew}} = 2$ and the initial $x = 0$ is superseded by the write at timestamp $1 \leq 2$, the behaviour where d reads $x = 0$ is forbidden.

In addition, ARMv8/RISC-V enforces ordering from strong store releases to program-order later strong load acquires. To model this ordering:

ρ3 The thread state maintains a view $\mathsf{v}_{\mathsf{Rel}} : \mathbb{V}$ containing the maximal post-view of all strong releases executed so far.
ρ4 The pre-view of any later strong load acquire includes $\mathsf{v}_{\mathsf{Rel}}$, enforcing the memory ordering.

The rules for barriers follow the same principle:

ρ5 `dmb.st` updates $\mathsf{v}_{\mathsf{wNew}}$ to include $\mathsf{v}_{\mathsf{wOld}}$.
ρ6 `dmb.ld` updates $\mathsf{v}_{\mathsf{rNew}}$ and $\mathsf{v}_{\mathsf{wNew}}$ to include $\mathsf{v}_{\mathsf{rOld}}$.
ρ7 `isb` updates $\mathsf{v}_{\mathsf{rNew}}$ to include $\mathsf{v}_{\mathsf{CAP}}$.

A.2 Load/store exclusive instructions

The previously discussed instructions can only introduce intra-thread ordering. Exclusive instructions (called load reserve/store conditional in RISC-V) make it possible to provide inter-thread atomicity guarantees. If a load exclusive a and a store exclusive b are paired and the store exclusive b is successful, then the write w' of b is guaranteed to be the immediate coherence successor of the write w that a reads from, apart from writes by the same thread (*i.e.* there are no writes from other threads to the same address between w and w' in memory).

$$\begin{array}{l||l} (a)\ \mathtt{r_1} := \mathtt{load_{ex}}\ [\mathtt{x}];\ //\ 37 & (c)\ \mathtt{store}\ [\mathtt{x}]\ 37; \\ (b)\ \mathtt{r_2} := \mathtt{store_{ex}}\ [\mathtt{x}]\ 42 & (d)\ \mathtt{store}\ [\mathtt{x}]\ 51; \\ & (e)\ \mathtt{r_3} := \mathtt{load}\ [\mathtt{x}]\ //\ 42 \end{array}$$

$r_1 = 37 \wedge r_2 = \mathsf{v}_{\mathrm{succ}} \wedge r_3 = 42$ forbidden

In this example, if a reads $x = 37$ from c, and b succeeds, then the write $x = 51$ by d is not allowed to come between the writes of c and b, and memory is not allowed to be $[1\colon \langle x := 37\rangle_2, 2\colon \langle x := 51\rangle_2, 3\colon \langle x := 42\rangle_1]$ (although different-address writes and non-exclusive writes to x from Thread 1 are allowed in between the load exclusive and the store exclusive). A store exclusive is only allowed to be paired with the most recent program-order earlier load exclusive (whether at the same location or not), and only if there has been no interposing (successful or unsuccessful) store exclusive, independent of their locations.

To capture the pairing of load and store exclusives:

ρ8 Each thread maintains an *exclusives bank* xclb : option $\langle \mathsf{time} : \mathbb{T}; \mathsf{view} : \mathbb{V}\rangle$, initially set to *none*, containing information about the last load exclusive when there has been no other store exclusive in that thread since then. Specifically,

ρ9 xclb is set to $\langle \mathsf{time} = t; \mathsf{view} = v\rangle$ (we omit *some* for simplicity) whenever a load exclusive reads from timestamp t with post-view v.

ρ10 xclb is set to *none* whenever a store exclusive (successful or not) is executed.

Consider an execution of the previous example, where c writes $x = 37$ at timestamp 1 and a reads the write $x = 37$ and sets xclb to $\langle \mathsf{time} = 1, \mathsf{view} = 1\rangle$. Now if d writes $x = 51$ at timestamp 2, b cannot write to x exclusively and must fail by the following rule:

ρ11 A store exclusive to location z at timestamp t succeeds only if xclb is not *none* and, in case the message at xclb.time is also z (so the load exclusive was to the same location), every message to z in memory between xclb.time and t is written by this thread.

ρ12 When the store exclusive succeeds it writes to a register indicating its success. The associated view in the success case is its post-view in RISC-V, and 0 in ARMv8. This means in RISC-V if another write depends on the success of a store exclusive, this write can only be promised after that of the store exclusive. In contrast, in ARMv8 this ordering is not preserved. This is the source of the deadlocks discussed in §4.3, for which we present a solution in §C.

Specifically, in Thread 1, xclb.time is 1 and d should write $x = 42$ at timestamp 3, but then the write $x = 51$ in the middle is written by Thread 2, which violates the above rule. Thus in order for b to be successful, b should be executed before d resulting in memory $[1\colon \langle x := 37\rangle_2, 2\colon \langle x := 42\rangle_1, 3\colon \langle x := 51\rangle_2]$ after d. Then e is constrained by $\mathsf{coh}(x) = 3$, which is due to d, and thus should read 51.

In addition to this atomicity guarantee, exclusives provide some ordering guarantees: the architectures guarantee that certain loads — load acquires in ARMv8, all loads in RISC-V — cannot read from a store exclusive by thread-internal forwarding. To capture this:

ρ13 Recall, the forward bank fwdb : Loc → option $[\mathsf{time} : \mathbb{T}; \mathsf{view} : \mathbb{V}; \mathsf{xcl} : \mathbb{B}]$ records in the xcl field whether the write in the forward bank is an exclusive write. The model then prevents a load acquire on ARM, and any load in RISC-V, from a location z from obtaining the smaller forward view $\mathsf{fwdb}(z).\mathsf{view}$ if $\mathsf{fwdb}(z).\mathsf{xcl}$ is set.

RISC-V additionally guarantees ordering of the store exclusive with the paired load-exclusive even if the load and the store are to different addresses.[3] To this end:

ρ14 Recall that the exclusives bank xclb : option $\langle \mathsf{time} : \mathbb{T}; \mathsf{view} : \mathbb{V}\rangle$ records the post-view of the load exclusive in the view field. In RISC-V, this view xclb.view is included in the paired store exclusive's pre-view.

A.3 Formal model

Fig. 4 summarises the types used by the model that were introduced in §4. For simplicity, values and addresses are mathematical integers. Promising-ARM and Promising-RISC-V use the same definitions, with an architecture flag a switching between ARM and RISC-V behaviour. This only affects the treatment of store exclusive instructions, in the store and load rules. However, not all instructions exist in both architectures: RISC-V has more barriers, and a weak store release. Fig. 5 gives the formal definition of the steps of the semantics, cross-referenced with the relevant rules in §4. First, we define auxiliary definitions.

Expressions interpretation The function (second and third line) takes an expression and a register state m, and returns the expression's value and view. Constants have view 0; registers are looked up in m; the view for an arithmetic expression merges the views of the arguments (**R9**).

[3] In the case of ARMv8 the architecture specifies "constrained unpredictable" behaviour; this is still being clarified.

$$a \in \text{Arch} ::= \texttt{ARM} \mid \texttt{RISC-V} \qquad l \in \text{Loc} \stackrel{def}{=} \text{Val} \qquad v \in \text{Val} \stackrel{def}{=} \mathbb{Z} \qquad tid \in \text{TId} \stackrel{def}{=} \mathbb{N} \qquad t \in \mathbb{T} \stackrel{def}{=} \mathbb{N} \qquad \nu \in \mathbb{V} \stackrel{def}{=} \mathbb{T}$$

$$w \in \text{Msg} \stackrel{def}{=} \langle \text{loc} : \text{Loc}; \text{val} : \text{Val}; \text{tid} : \text{TId} \rangle \qquad \langle x := v \rangle_{tid} \stackrel{def}{=} \langle \text{loc} = x; \text{val} = v; \text{tid} = tid \rangle \qquad M \in \text{Memory} \stackrel{def}{=} \text{list Msg}$$

$$ts \in \text{TState} \stackrel{def}{=} \left\langle \begin{array}{l} \text{prom} : \text{set } \mathbb{T}; \quad \text{regs} : \text{Reg} \to \text{Val} \times \mathbb{V}; \\ \text{coh} : \text{Loc} \to \mathbb{V}; \quad \mathsf{v}_{\text{rOld}}, \mathsf{v}_{\text{wOld}}, \mathsf{v}_{\text{rNew}}, \mathsf{v}_{\text{wNew}}, \mathsf{v}_{\text{CAP}}, \mathsf{v}_{\text{Rel}} : \mathbb{V}; \\ \text{fwdb} : \text{Loc} \to \langle \text{time} : \mathbb{T}; \text{view} : \mathbb{V}; \text{xcl} : \mathbb{B} \rangle; \\ \text{xclb} : \text{option} \langle \text{time} : \mathbb{T}; \text{view} : \mathbb{V} \rangle \end{array} \right\rangle$$

$$T \in \text{Thread} \stackrel{def}{=} \text{St} \times \text{TState} \qquad \vec{T} \in \text{TPool} \stackrel{def}{=} \text{TId} \to \text{Thread} \qquad \langle \vec{T}, M \rangle \in \text{Machine} \stackrel{def}{=} \text{TPool} \times \text{Memory}$$

Figure 4. Types in the semantics

read read(M, l, t) gives the result of reading location l at timestamp t in memory M. For $t = 0$, this is the initial value v_{init}, here 0; otherwise either the value of the message in M at timestamp t if its location is l, otherwise *none*.

read-view read-view(a, rk, f, t) returns either the timestamp t of the read message or the forward view of the message f in the forward bank, subject to certain constraints on the architecture a and read kind rk (R13, R14, R15, R16, ρ13).

atomic atomic(M, l, tid, t_r, t_w) checks whether an exclusive write to l at timestamp t_w by thread tid can become successful, and so atomic with respect to its earlier exclusive read with read message at timestamp t_r in the current memory M (ρ8, ρ9, ρ11).

Now we define thread-local steps $T, M \xrightarrow{[t]}_{tid} T'$, which do not change the memory.

EXCLUSIVE-FAILURE A store exclusive that has not been executed is always allowed to fail. It sets r_{succ} to v_{fail} (here 1) to signal failure, with 0 timestamp, and sets xclb to *none* (ρ10).

FULFIL starts with the pre-condition (from top to bottom). First evaluate address and data expressions. Rule ρ11 explains the condition for exclusive writes. Since we assume writes always promise first and then fulfil, this step requires the write to have been promised.

Rules R10, R6, R21, ρ1, ρ14 describe the components contributing to the pre-view. The pre-view and coherence view have to be less than t (R19); the post-view is the timestamp t (R3). ρ12 explains the view v_{succ} placed on the register write indicating the success. The post-condition removes the promise (R19); writes v_{succ} (here 0) to the "success register" (ρ12); and updates the coherence view to include t (R11), certain views (R5, R22, ρ3); the forward bank (R14, ρ13); and the exclusives bank (ρ8, ρ10).

READ also starts with the pre-condition (from top to bottom). First evaluate the address l; in order to read v it must be $v = \text{read}(M, l, t)$ as described above. The pre-view calculation is described in R10, R6, ρ4. The pre-view (R2) and the coherence view (R12) constrain the read. The post-view is defined in R3, R16. The post-condition updates the register with value and post-view (R9); coherence with post-view as in rule R11; views as in R5, ρ2, R22; and exclusives bank as in ρ9.

FENCE This defines a single rule for all other ARMv8 and RISC-V fences in a format matching RISC-V's `fence` instruction. $\texttt{fence}_{K_1,K_2}$ has two arguments: K_1 indicates whether the fence creates ordering with respect to program-order preceding reads (R), writes (W), or both (RW); similarly K_2 indicates which program-order later instructions are ordered with it (R, W, or RW). It then updates v_{rNew} and/or v_{wNew} (depending on K_2), to include v_{rOld} and/or v_{wOld} (depending on K_1) according to the intuition given in R5, R6. We define ARMv8's full barrier `dmb.sy` = $\texttt{fence}_{RW,RW}$, its load barrier `dmb.ld` = $\texttt{fence}_{R,RW}$, its store barrier `dmb.st` = $\texttt{fence}_{W,W}$, and moreover RISC-V's "TSO fence" as `fence.tso` = $\texttt{fence}_{R,R}$;$\texttt{fence}_{RW,W}$. With these definitions, the behaviour of the ARM barriers is as explained with rules R5, R6 R7, ρ5, ρ6.

REGISTER A register assignment updates the register with the expressions and view from the evaluation of its expression (R9).

BRANCH The pre-condition evaluates the condition, branches as determined by this value, and updates v_{CAP} (R22).

ISB Executes an `isb` by merging v_{CAP} into v_{rNew} (ρ7).

SKIP, SEQ, and WHILE Mostly as expected. `while` is expressed using a branch.

We can then define thread steps $\langle T, M \rangle \xrightarrow{[t]}_{tid} \langle T', M' \rangle$.

EXECUTE lifts a thread-local step that does not change memory to a thread step. PROMISE allows promising any write message, appending this write to memory and recording its timestamp in prom. While thread steps allow unconstrained promises, machine steps only allow certified promises. Finally, machine steps just lift *certified* thread steps (R24).

$c\ ?\ v_1 : v_2 \stackrel{def}{=} \text{if } c \text{ then } v_1 \text{ else } v_2 \qquad c\ ?\ v \stackrel{def}{=} c\ ?\ v : 0 \qquad v_1 \sqcup v_2 \stackrel{def}{=} \max(v_1, v_2) \qquad v@v \stackrel{def}{=} \langle v, v\rangle : \text{Val} \times \mathbb{V}$

$[\![(-)_1]\!]_{(-)_2} : \mathit{Expr} \to (\text{Reg} \to \text{Val} \times \mathbb{V}) \to \text{Val} \times \mathbb{V}$

$[\![v]\!]_m \stackrel{def}{=} v@0 \qquad [\![r]\!]_m \stackrel{def}{=} m(r) \qquad [\![e_1 \text{ op } e_2]\!]_m \stackrel{def}{=} (v_1\ [\![\text{op}]\!]\ v_2)@(v_1 \sqcup v_2) \text{ with } [\![e_1]\!]_m = v_1@v_1, [\![e_2]\!]_m = v_2@v_2$

$\text{read}(M, l, t) : \text{option Val} \stackrel{def}{=} \text{if } t = 0 \text{ then } \mathsf{v}_{\text{init}} \text{ else if } M(t).\text{loc} = l \text{ then } M(t).\text{val else } \mathit{none}$

$\text{read-view}(a, rk, f, t) \stackrel{def}{=} \text{if } (f.\text{time} = t \wedge (f.\text{xcl} \Rightarrow (a = \texttt{ARM} \wedge rk \sqsubseteq \text{pln}))) \text{ then } f.\text{view else } t$

$\text{atomic}(M, l, tid, t_\text{r}, t_\text{w}) \stackrel{def}{=} M(t_\text{r}).\text{loc} = l \implies \forall t'.\ (t_\text{r} < t' < t_\text{w} \wedge M(t').\text{loc} = l) \implies M(t').\text{tid} = tid$

$\boxed{T, M \xrightarrow{[t]}_{a,tid} T'}$

(EXCLUSIVE-FAILURE)

$$\frac{xcl = \texttt{true} \quad ts' = ts[\text{regs}(r_\text{succ}) \mapsto \mathsf{v}_\text{fail}@0, \text{xclb} \mapsto \mathit{none}]}{\langle r_\text{succ} := \texttt{store}_{xcl,wk}[e_1]\ e_2, ts\rangle, M \to_{a,tid} \langle \texttt{skip}, ts'\rangle}$$

(READ)

$$\frac{\begin{array}{l} l@v_\text{addr} = [\![e]\!]_{ts.\text{regs}} \\ \text{read}(M, l, t) = v \\ v_\text{pre} = v_\text{addr} \sqcup ts.\mathsf{v}_\text{rNew} \sqcup (rk \sqsupseteq \text{acq}\ ?\ ts.\mathsf{v}_\text{Rel}) \\ \forall t'.\ t < t' \le (v_\text{pre} \sqcup ts.\text{coh}(l)) \implies M(t').\text{loc} \neq l \\ v_\text{post} = v_\text{pre} \sqcup \text{read-view}(a, rk, ts.\text{fwdb}(l), t) \\ ts' = ts\begin{bmatrix} \text{regs}(r) \mapsto v@v_\text{post}, \\ \text{coh}(l) \mapsto ts.\text{coh}(l) \sqcup v_\text{post}, \\ \mathsf{v}_\text{rOld} \mapsto ts.\mathsf{v}_\text{rOld} \sqcup v_\text{post}, \\ \mathsf{v}_\text{rNew} \mapsto ts.\mathsf{v}_\text{rNew} \sqcup (rk \sqsupseteq \text{wacq}\ ?\ v_\text{post}), \\ \mathsf{v}_\text{wNew} \mapsto ts.\mathsf{v}_\text{wNew} \sqcup (rk \sqsupseteq \text{wacq}\ ?\ v_\text{post}), \\ \mathsf{v}_\text{CAP} \mapsto ts.\mathsf{v}_\text{CAP} \sqcup v_\text{addr}, \\ \text{xclb} \mapsto xcl\ ?\ \langle \text{time} = t; \text{view} = v_\text{post}\rangle : ts.\text{xclb} \end{bmatrix} \end{array}}{\langle r := \texttt{load}_{xcl,rk}\ [e], ts\rangle, M \to_{a,tid} \langle \texttt{skip}, ts'\rangle}$$

(FULFIL)

$$\frac{\begin{array}{l} [\![e_1]\!]_{ts.\text{regs}} = l@v_\text{addr} \qquad [\![e_2]\!]_{ts.\text{regs}} = v@v_\text{data} \\ xcl \implies ts.\text{xclb} \neq \mathit{none} \wedge \text{atomic}(M, l, tid, ts.\text{xclb.time}, t) \\ t \in ts.\text{prom} \qquad M(t) = \langle l := v\rangle_{tid} \\ v_\text{pre} = v_\text{addr} \sqcup v_\text{data} \sqcup ts.\mathsf{v}_\text{wNew} \sqcup ts.\mathsf{v}_\text{CAP} \sqcup \\ \qquad (wk \sqsupseteq \text{wrel}\ ?\ (ts.\mathsf{v}_\text{rOld} \sqcup ts.\mathsf{v}_\text{wOld})) \sqcup \\ \qquad ((a = \texttt{RISC-V} \wedge xcl)\ ?\ ts.\text{xclb.view}) \\ (v_\text{pre} \sqcup ts.\text{coh}(l)) < t \\ v_\text{post} = t \qquad v_\text{succ} = (a = \texttt{RISC-V}\ ?\ v_\text{post} : \bot) \\ ts' = ts\begin{bmatrix} \text{prom} \mapsto ts.\text{prom} \setminus \{t\}, \\ \text{regs}(r_\text{succ}) \mapsto xcl\ ?\ \mathsf{v}_\text{succ}@v_\text{succ} : ts.\text{regs}(r_\text{succ}), \\ \text{coh}(l) \mapsto ts.\text{coh}(l) \sqcup v_\text{post}, \\ \mathsf{v}_\text{wOld} \mapsto ts.\mathsf{v}_\text{wOld} \sqcup v_\text{post}, \\ \mathsf{v}_\text{CAP} \mapsto ts.\mathsf{v}_\text{CAP} \sqcup v_\text{addr}, \\ \mathsf{v}_\text{Rel} \mapsto ts.\mathsf{v}_\text{Rel} \sqcup (wk \sqsupseteq \text{rel}\ ?\ v_\text{post}), \\ \text{fwdb}(l) \mapsto \langle \text{time} = t; \text{view} = v_\text{addr} \sqcup v_\text{data}; \text{xcl} = xcl\rangle \\ \text{xclb} \mapsto xcl\ ?\ \mathit{none} : ts.\text{xclb} \end{bmatrix} \end{array}}{\langle r_\text{succ} := \texttt{store}_{xcl,wk}\ [e_1]\ e_2, ts\rangle, M \xrightarrow{t}_{a,tid} \langle \texttt{skip}, ts'\rangle}$$

(FENCE)

$$\frac{\begin{array}{l} v_1 = (\mathsf{R} \sqsubseteq K_1\ ?\ ts.\mathsf{v}_\text{rOld}) \sqcup \\ \qquad (\mathsf{W} \sqsubseteq K_1\ ?\ ts.\mathsf{v}_\text{wOld}) \end{array} \qquad ts' = ts\begin{bmatrix} \mathsf{v}_\text{rNew} \mapsto ts.\mathsf{v}_\text{rNew} \sqcup (\mathsf{R} \sqsubseteq K_2\ ?\ v_1), \\ \mathsf{v}_\text{wNew} \mapsto ts.\mathsf{v}_\text{wNew} \sqcup (\mathsf{W} \sqsubseteq K_2\ ?\ v_1) \end{bmatrix}}{\langle \texttt{fence}_{K_1,K_2}, ts\rangle, M \to_{a,tid} \langle \texttt{skip}, ts'\rangle}$$

(REGISTER)

$$\frac{ts' = ts[\text{regs}(r) \mapsto [\![e]\!]_{ts.\text{regs}}]}{\langle r := e, ts\rangle, M \to_{a,tid} \langle \texttt{skip}, ts'\rangle}$$

(BRANCH)

$$\frac{[\![e]\!]_{ts.\text{regs}} = v@v \qquad ts' = ts[\mathsf{v}_\text{CAP} \mapsto ts.\mathsf{v}_\text{CAP} \sqcup v]}{\langle \texttt{if } (e)\ s_1\ s_2, ts\rangle, M \to_{a,tid} \langle v \neq 0\ ?\ s_1 : s_2, ts'\rangle}$$

(ISB)

$$\frac{ts' = ts[\mathsf{v}_\text{rNew} \mapsto ts.\mathsf{v}_\text{rNew} \sqcup ts.\mathsf{v}_\text{CAP}]}{\langle \texttt{isb}, ts\rangle, M \to_{a,tid} \langle \texttt{skip}, ts'\rangle}$$

(SKIP)

$$\frac{}{\langle \texttt{skip}; s, ts\rangle, M \to_{a,tid} \langle s, ts\rangle}$$

(SEQ)

$$\frac{\langle s_1, ts\rangle, M \to_{a,tid} \langle s_1', ts'\rangle}{\langle s_1; s_2, ts\rangle, M \to_{a,tid} \langle s_1'; s_2, ts'\rangle}$$

(WHILE)

$$\frac{s' = \texttt{if } (e)\ (s; \texttt{while } (e)\ s)\ \texttt{skip}}{\langle \texttt{while } (e)\ s, ts\rangle, M \to_{a,tid} \langle s', ts\rangle}$$

$\boxed{\langle T, M\rangle \xrightarrow{[t]}_{a,tid} \langle T', M'\rangle}$

(EXECUTE)

$$\frac{T, M \to_{a,tid} T'}{\langle T, M\rangle \to_{a,tid} \langle T', M\rangle}$$

(PROMISE)

$$\frac{w.\text{tid} = tid \quad t = |M| + 1 \quad ts' = ts[\text{prom} \mapsto ts.\text{prom} \cup \{t\}]}{\langle \langle s, ts\rangle, M\rangle \xrightarrow{t}_{a,tid} \langle \langle s, ts'\rangle, M +\!\!+ [w]\rangle}$$

$\boxed{\langle \vec{T}, M\rangle \to_a \langle \vec{T}', M'\rangle}$

(MACHINE-STEP)

$$\frac{\langle \vec{T}[tid], M\rangle \to_{a,tid} \langle T', M'\rangle \qquad \langle T', M'\rangle \text{ certified}}{\langle \vec{T}, M\rangle \to_a \langle \vec{T}[tid \mapsto T'], M'\rangle}$$

$\langle T, M\rangle \text{ certified} \stackrel{def}{=} \exists T', M'.\ \langle T, M\rangle \xrightarrow{\text{seq}}{}^*_{a,tid} \langle T', M'\rangle \wedge T'.\text{prom} = \{\}$

$\boxed{\langle T, M\rangle \xrightarrow{\text{seq}}_{a,tid} \langle T', M'\rangle}$

(SEQ-EXEC)

$$\frac{T, M \to_{a,tid} T'}{\langle T, M\rangle \xrightarrow{\text{seq}}_{a,tid} \langle T', M\rangle}$$

(SEQ-WRITE)

$$\frac{\langle T, M\rangle \xrightarrow{t}_{a,tid} \langle T', M'\rangle \qquad T', M' \xrightarrow{t}_{a,tid} T''}{\langle T, M\rangle \xrightarrow{\text{seq}}_{a,tid} \langle T'', M'\rangle}$$

Figure 5. Thread-local steps, thread steps, and machine steps

B Algorithm

In the following, we describe the *algorithm* used by the executable model of §7 to implement the certification and promise enumeration: a function that enumerates the legal next (promise or non-promise) steps of a thread. The certification algorithm has to handle two tasks. Given a thread state and the current memory state, first it has to decide which of the possible next instructions steps of the thread allow fulfilling the promises the thread has already made. Second, it has to enumerate the possible new promises the thread should be allowed to make. These promises have to correspond to feasible writes by store instructions of this thread but also be compatible with the set of promises the thread has already "committed to". The main challenge in developing the certification algorithm is in the latter, *computing* the new promise steps that should be enabled in the current thread configuration.

The model of §5 (and §A.3) allows adding arbitrary new promises *during the certification*. Doing the same in the executable model would make promise enumeration and certification computationally infeasible.

- The algorithm therefore forbids early promises during certification (*i.e.* only allows "normal writes"), using the fact that this does not change the model behaviour.

Moreover, in general, given an arbitrary program, fulfilling a promise may take arbitrarily many steps by this thread, in particular in the presence of loops whose execution is not statically bounded. For the sake of executability, the executable model necessarily bounds these, and the certification algorithm takes a *fuel* argument, limiting the number of thread steps the certification is allowed to take. The idea of the algorithm is then to enumerate all legal traces of this thread in isolation under current memory, of a length bounded by this argument, that lead to a state in which all of this thread's promises have been fulfilled. Then:

1. Any such trace's first (non-promise) step is immediately certified.
2. Moreover, any normal write done by this thread on such a trace corresponds to a legal promise step if it has the *pre-view and coherence-view (at its location)* less than or equal to the maximal timestamp of current memory (the memory before the start of the certification).

$$\begin{array}{l} (a)\ \mathtt{r_1 := load\ [w];} \\ (b)\ \mathtt{store\ [x]\ 1;} \\ (c)\ \mathtt{store_{rel}\ [y]\ 1;} \\ (d)\ \mathtt{store\ [z]\ r_1} \end{array} \Big\| \ \ldots$$

To illustrate the algorithm, consider the above (partial) program. Assume that the memory is $[1{:}\ \langle w := 1\rangle_2, 2{:}\ \langle z := 1\rangle_1]$, that the promise set of Thread 1 is prom $= \{2\}$, and that Thread 1 has not yet executed a. The certification algorithm first enumerates all traces of Thread 1 leading to states in which all its promises have been fulfilled. Here, there is only one such trace:

- a reads 1 from w: otherwise d would write $z = 0$, which cannot fulfil the outstanding promise 2: $\langle z := 1\rangle_1$.
- b writes $x = 1$ at timestamp 3 with pre-view 0 and coherence-view 0, leading to post-view 3.
- c writes $y = 1$ at timestamp 4 with pre-view 3 and coherence-view 0: as a store release c's pre-view includes b's post-view, via $\mathrm{v_{wOld}}$.
- And d fulfils 2: $\langle z := 1\rangle_1$.

Therefore:

1. The next-instruction step in which a reads 1 from w is a certified step for Thread 1. (The one where a reads 0 is not, due to the outstanding promise 2: $\langle z := 1\rangle_1$.)
2. Promising the write $x = 1$ at timestamp 3 is also certified: it is a possible write by Thread 1 on a path fulfilling its promises, and with a pre-view and coherence view both less than or equal to the current maximal timestamp 2 in memory (before the certification run).
3. Promising the write $y = 1$, however, is not a certified step, since c's pre-view in the only possible certification trace is $3 \not\leq 2$.

C Certification with ARMv8 store exclusives

In §4.3 and §5 we introduced a simple certification definition to avoid model executions in which not all promises are fulfilled. This certification is sound for RISC-V and ARMv8, and precise for RISC-V programs and for ARMv8 programs without store exclusives. In the presence of ARMv8 store exclusive instructions, however, it is imprecise: matching ARMv8's architecturally intended weak semantics of store exclusive instructions in Promising-ARM leads to executions in which the model gets

stuck due to unfulfilled promises, of a similar sort as those present in the Flat model [39]. In this section, we extend the model's machine state with locks and a certification that takes the locking into account to prevent these executions and make certification sound and precise for ARMv8 programs even with store exclusives, and makes the model deadlock-free.

C.1 The challenge of certification with ARMv8 store exclusives

One of the main simplifications of the recent revision of ARMv8 was that, where the architecture previously distinguished between notions of "true" and "false" dependencies, it now makes no such distinction. Now syntactic dependencies of the right kind induce memory ordering, with no consideration of whether the result of the register computation varies as a function of its input or not. Therefore, in the revised ARMv8, it is not always sound to replace an expression by another expression that performs the same register computation. However, ARMv8's specification of store exclusives intends to allow processors to treat a load/store exclusive pair as a single atomic operation that is guaranteed to succeed, e.g. treating $\mathtt{r}_1 := \mathtt{load}_{\mathsf{ex}}\ [\mathtt{x}]; \mathtt{r}_2 := \mathtt{store}_{\mathsf{ex}}\ [\mathtt{x}]\ (\mathtt{r}_1 + 1)$ as $\mathtt{r}_1 := \mathtt{fetch\text{-}and\text{-}add}\ [x]; \mathtt{r}_2 := 0$. Now, r_2 still has to be set to indicate success ($\mathsf{v}_{\mathsf{succ}}$), but does not syntactically depend on the store. Therefore, in ARMv8, a dependency on r_2 in the program above does not induce ordering (and Promising-ARM sets its associated view to 0). But the value of r_2 after the store exclusive still depends on the success of the store exclusive!

This leads to surprising behaviours: in the following program despite the dependency from b to c they can be reordered. In particular, Thread 1 may (1.) execute a and read the initial value 0 (so $r_1 = 0$) and, (2.) assume the success of b (so $r_2 = \mathsf{v}_{\mathsf{succ}} = 0$) and write $p = 1$ with c, *before b is in memory*: the architecture allows that d reads $p = 1$ and f reads $x = 0$ after the barrier e. (While in RISC-V the dependency from b to c means b propagates before c, forbidding this behaviour.)

$$
\begin{array}{l||l||l}
a : \mathtt{r}_1 := \mathtt{load}_{\mathsf{ex}}\ [\mathtt{x}]; & d : \mathtt{r}_3 := \mathtt{load}\ [\mathtt{p}];\ \text{// 1} & g : \mathtt{store}\ [\mathtt{x}]\ 2 \\
b : \mathtt{r}_2 := \mathtt{store}_{\mathsf{ex}}\ [\mathtt{x}]\ (\mathtt{r}_1 + 1); & e : \mathtt{dmb.sy}; & \\
c : \mathtt{store}\ [\mathtt{p}]\ (1 - \mathtt{r}_1 - \mathtt{r}_2) & f : \mathtt{r}_4 := \mathtt{load}\ [\mathtt{x}]\ \text{// 0} &
\end{array}
$$

$r_3 = 1 \wedge r_4 = 0$ allowed

This means whereas in all other cases a store may propagate to memory only when all its dependencies are "fixed", dependencies on the success of a store exclusive are special. In order to allow the above re-ordering of b and c, an operational model has to do extra work, since it has to ensure the success of b and therefore its atomicity with respect to the write a read from, until b is done. For the simple case above, mimicking the behaviour of a processor and replacing $\mathtt{r}_1 := \mathtt{load}_{\mathsf{ex}}\ [\mathtt{x}]; \mathtt{r}_2 := \mathtt{store}_{\mathsf{ex}}\ [\mathtt{x}]\ (\mathtt{r}_1 + 1)$ with $\mathtt{r}_1 := \mathtt{fetch\text{-}and\text{-}add}\ [x]; \mathtt{r}_2 := 0$ is easy. However, handling the dependency relaxation in its full generality (without deadlocks) is difficult.

This problem manifests in Promising-ARM in the following way: in the initial state Thread 1 is allowed to promise $p = 0$. Since c can produce a write $p = 0$ only if a reads $x = 0$ and b succeeds, the ability of Thread 1 to fulfil the promise now depends b's success, and so on whether b's write can enter memory as the next write to location x (after the initial write $x = 0$). If, however, g now writes to x, b will fail and the model gets stuck, with c unfulfilled. Since c's early promise has to be allowed to match ARMv8 semantics, the model must instead prevent g from writing until b's write, since g would break Thread 1's promise.

The certification definition presented in §4 and 5 works thread-locally: it takes into account only a thread's state and current memory in order to decide whether a thread step should be allowed or not. But whether g's write should be allowed cannot be determined based only on the state of Thread 3 and on the list of messages in memory: it is Thread 1's promises due to which g must not write. So a precise certification algorithm for ARMv8 needs to take into account some information about other threads. In this example, Thread 1 effectively requires "locking" location x, to constrain the behaviour of the other threads. We leverage this intuition of a thread locking a location and extend memory with a lock state, in order to still allow certifying a thread by taking into account only its own state and the (extended) memory state.

C.2 Extended certification, take 1

The example indicates a pattern: in the problematic execution a thread's ability to fulfil its promises depends on both, a read exclusive, and the success of a paired write exclusive that has been promised. More precisely, the state in which a Thread n requires a lock on a location x is the following:

- Thread n depends on a load exclusive l to location x. This is if:
  - Thread n has already executed l, or
  - Thread n has an outstanding promise whose fulfilment depends on l due to register dataflow or due to coherence or view requirements.
- And Thread n relies on the success of the write of a store exclusive s to x that is paired with l: Thread n has an outstanding promise whose fulfilment depends on s via register dataflow.

- And the write w' that l read from is already in memory, whereas the write of s is not.

If the above holds, Thread n's dependency on l "fixes" the write w' that l reads from, and due to the success dependency on s requires the write of s to succeed and be atomic with respect to w'.

The idea underlying the extended model is to precisely detect cases when this condition holds, and to then lock x for Thread n in memory for as long as the condition holds and prevent other threads from writing to locked locations. The main challenge here is in detecting the above condition. The model handles this by extending the certification and generalising the views of the thread state. During the certification the model tracks dependencies from load and store exclusive instructions to other stores, in order to detect when the fulfilment of a write promise by some store depends on such load/store exclusive pairs as described in the condition. To this end, the extended certification uses views that in addition to timestamps carry *taints* that keep track of the load/store exclusive instruction dependencies, including information about their memory location and pairing.

In the example above, in the state after Thread 1 has read $x = 0$ with a and promised $p = 1$ with c, the extended certification will work as follows:

- the only certifying execution of Thread 1 alone under current memory is one where a reads $x = 0$, and b succeeds. In this execution:
- a taints r_1 to indicate it is from a load exclusive a to location x reading from a value in memory.
- b taints r_2 to indicate it is from a successful store exclusive to location x whose write is not in memory yet and that is paired with a.
- when fulfilling $p = 0$ with c, c's pre-view includes a taint with information about both a and b: a and b are paired and to location x; a reads at timestamp 0, so from a write in memory; b is not propagated yet.
- Therefore Thread 1 requires locking x.

The information returned by the certification is then, which locations have to be locked in order to *guarantee* a thread can fulfil its promises, and a machine step is only allowed if the step is compatible with the current lock state in memory: not writing to a location locked by another thread, and not locking already-locked locations.

The taint tracking introduces complexity to the certification. Importantly however, during "normal" execution the extended model's views are simple timestamps just as before (§5), and taints are not persistent but local to the certification.

C.3 Extended certification, take 2

As the following example illustrates, unfortunately the ideas on certification above are still insufficient. In this example Thread 1's store d depends on the load exclusive b to x, and the "success register write" of the store exclusive c to location x. New here is that c has release ordering, and so c is ordered after the write a to y. Symmetrically in Thread 2 h depends on the load and store exclusive instructions f and g to y, and g is a store exclusive release ordered after a store e to x.

a : store [y] 1;	e : store [x] 1;
b : r_1 := load_{ex} [x];	f : r_3 := load_{ex} [y];
c : r_2 := $\text{store}_{\text{ex, rel}}$ [x] 1;	g : r_4 := $\text{store}_{\text{ex, rel}}$ [y] 1;
d : store [p] $(1 - r_1 - r_2)$	h : store [q] $(1 - r_3 - r_4)$

Now assume an execution in which Thread 1 promises $p = 1$. Since this depends on b reading $x = 0$ and c eventually successfully writing $x = 1$, our extended certification requires a lock on x for Thread 1 to prevent Thread 2 from breaking its promise. Now Thread 2 could analogously promise $q = 1$, after which the model also locks location y for Thread 2, which does not contradict Thread 1 locking x. But now the model is stuck again: Thread 1 cannot execute a, since y is locked by Thread 2; c's write $x = 1$ would release the lock on x, but since c is a store release, this requires first promising a. Thread 2 in turn cannot execute e due to the lock on x, and cannot promise g's $y = 1$ (and unlock y) before executing e.

In order to avoid such executions, the extended certification has to be improved to take into account some information about the thread-internal ordering requirements in order to prevent model deadlocks. To this end, the extended model's taints carry additional information about stores preceding store exclusive release instructions and the lock state captures rely-guarantee style lock information per thread; a machine step is then only allowed if the rely-guarantee lock information of all thread states is consistent. Then in the previous bad execution:

- For the promise $p = 1$ the certification returns information of the form $([y]; x)$, meaning that for this step Thread 1 requires a lock on x, and that it *relies* on being able to write to y before releasing the lock on x. Promising $p = 1$ adds this to the memory's lock state.
- Since there are no other locks, Thread 1 can promise.
- In the following state, for the promise $q = 1$ by Thread 2, symmetrically, the certification returns the information $([x], y)$.

- Now $([x], y)$ is incompatible with $([y], x)$ due to the cyclic rely-guarantee dependency. Thus Thread 2 is not allowed to promise $q = 1$.

Moreover, certain sequences of multiple such store exclusive release instructions can lead to *nesting* of these rely-guarantee locks, making the consistency checking difficult. In particular, our current algorithm for checking the consistency is exponential in the nesting depth. We believe, in practice, sequences with nesting depth greater than 1 do not occur "naturally". Hence, for the purpose of exhaustive state space exploration, the executable model approximates the lock information and consistency checking up to depth one. The model may then still get stuck in cases requiring depth more than 1, but consistency checking becomes linear in the size of the lock information. (Irrespective, the model remains sound.)

For lack of space we omit the details of the extended certification and refer the interested reader to the Coq formalisation in the supplementary material.

D Equivalence with the reference axiomatic memory model

The argument *arch* switches between ARMv8 and RISC-V. For simplicity of the formalisation, the barriers here are `dmb.rw`, `dmb.rr`, `dmb.wr`, `dmb.ww`. All others are just "macros": combinations of these. For example: ARMv8's `dmb.ld` = `dmb.rw`; `dmb.rr`. AQ is for strong read acquire, AQpc for the weak read acquire, RL for the strong write release, RLpc for the weak write release.

$$
\begin{array}{l}
\text{let } obs = rfe \mid fr \mid co \\
\text{let } dob \;= addr \mid data \\
\qquad\qquad \mid\; (addr \mid data); rfi \\
\qquad\qquad \mid\; (ctrl \mid (addr; po)); [W] \\
\qquad\qquad \mid\; (ctrl \mid (addr; po)); [\texttt{isb}]; po; [R] \\
\text{let } aob = [range(rmw)]; rfi; \\
(\textit{if } arch = \text{RISC-V } \textit{then } [R] \textit{ else } [AQ|AQpc]) \\
\text{let } bob \;= [R]; po; [\texttt{dmb.rr}]; po; [R] \\
\qquad\qquad \mid\; [R]; po; [\texttt{dmb.rw}]; po; [W] \\
\qquad\qquad \mid\; [W]; po; [\texttt{dmb.wr}]; po; [R] \\
\qquad\qquad \mid\; [W]; po; [\texttt{dmb.ww}]; po; [W] \\
\qquad\qquad \mid\; [RL]; po; [AQ] \\
\qquad\qquad \mid\; [AQ|AQpc]; po \\
\qquad\qquad \mid\; po; [RL|RLpc] \\
\qquad\qquad \mid\; \textit{if } arch = \text{RISC-V } \textit{then } rmw \\
\text{let } ob = obs \mid dob \mid aob \mid bob \\
\text{acyclic } po\text{-}loc \mid fr \mid co \mid rf \text{ as } internal \\
\text{acyclic } ob \text{ as } external \\
\text{empty } rmw \;\&\; (fre; coe) \text{ as } atomic
\end{array}
$$

Figure 6. ARMv8 and RISC-V axiomatic memory models

The revised ARMv8 has an official axiomatic concurrency model, written in herd [11], by Will Deacon [20]. RISC-V has an axiomatic model closely following ARM's, produced by the RISC-V Memory Model Task Group, chaired by Daniel Lustig. The models work in a two-step process. The models first enumerate the set of all *candidate executions*. Each candidate execution is one potential full execution of the program, specified by relations on its memory accesses $\langle po, co, rf, rmw \rangle$.

- po (program order) is a control flow unfolding of the threads of the program.
- co is the coherence order, the sequencing of writes to the same address in memory.
- rf is the reads-from relation, relating a write access w with a read access r that reads from w.
- rmw relates a read and write access of successfully paired load and store exclusive instructions.

In the second step the model checks each candidate execution for whether it satisfies its *axioms*, and only allows such *legal* executions that do. Typically the axioms require the acyclicity of certain relations of the full candidate executions.

In ARMv8 there are three axioms: a standard coherence axiom and an axiom concerning the atomicity guarantees of load/store exclusive instructions, and the "main" axiom. For the main axiom, the relation obs describes the interaction between memory accesses of different threads (using reads-from and coherence), and the relation ob describes the thread-local ordering due to dependencies and barriers every execution must preserve. The main axiom requires that the interaction between threads is compatible with this thread-internal ordering, by requiring the acyclicity of the relations. For RISC-V, the axiomatic herd

model [42] is similar. The proof currently assumes known simplifications of the axiomatic models to unify them in the Coq formalisation. We call this model Axiomatic for both cases, ARM or RISC-V (see supplementary material for the definitions).

We now define two variants of the Promising semantics. To this end, first define a *valid execution* as an execution in which the threads in the final state have no outstanding promises. Then we call Promising the model as defined in §A.3 accepting only such valid executions. Second, as an intermediate model for the proof, we define Global-Promising to be the same as Promising, except where machine-step requires no certification. These Promising model variants are equivalent. Moreover Promising for RISC-V has no deadlocks (*i.e.* every execution is valid).

The following statements all assume finite executions.

Theorem D.1. *For a program p, $\vec{R}$ is a final register state of a legal candidate execution of p in Axiomatic if and only if it is that of a valid execution of p in Global-Promising.*

Proof. Proved in Coq (for both ARM and RISC-V). □

Theorem D.2. *For a program p, $\vec{R}$ is a final register state of a valid execution of p in Global-Promising if and only if it is that of a valid execution of p in Promising.*

Proof. Proved in Coq (for both ARM and RISC-V). □

Theorem D.3 (Dead-lock freedom for RISC-V)**.** *For every certified state in Promising for RISC-V, either it is a final state with no outstanding promise, or there exists a step to another certified state.*

Proved in Coq.

E Full evaluation results

Test	*Promising*	*Flat*
SLA-1	0.27	0.41
SLA-2	0.30	3.38
SLA-3	0.33	21.57
SLA-4	0.39	110.18
SLA-5	0.44	526.76
SLA-6	0.52	2277.72
SLA-7	0.61	9108.53
SLA-8	0.73	ooT
SLA-9	0.86	ooT
SLA-10	1.01	ooT
SLC-1	3.21	8.63
SLC-2	4.69	121.98
SLC-3	6.58	1472.74
SLR-1	2.47	3.70
SLR-2	3.50	17.51
SLR-3	4.88	52.52
PCS-1-1	0.26	0.33
PCS-2-2	0.40	10.33
PCS-3-3	1.36	249.26
PCM-1-1-1	0.30	23.58
PCM-2-2-2	1.70	ooT
PCM-3-3-3	71.12	ooT
TL/(opt)-1	10.16 / 10.28	456.12 / 1180.33
TL/(opt)-2	13.72 / 14.54	2202.12 / 7115.31
TL/(opt)-3	18.08 / 20.13	ooT / ooT
STC/(opt)-100-010-000	0.36 / 0.36	35.26 / 104.57
STC/(opt)-100-010-010	0.42 / 0.42	2144.52 / 5943.50
STC/(opt)-100-100-010	8.70 / 8.70	ooT / ooT
STC/(opt)-110-011-000	7.64 / 8.13	ooT / ooT
STC/(opt)-110-100-010	21.84 / 22.48	ooT / ooT
STC/(opt)-200-020-000	7.16 / 7.12	ooT / ooT
STC/(opt)-210-011-000	615.41 / 637.98	ooT / ooT
STR-100-010-000	0.35	4.61
STR-100-010-010	0.39	77.21
STR-100-100-010	7.30	8940.03
STR-110-011-000	6.55	ooT
STR-110-100-010	18.09	ooT
STR-200-020-000	5.80	11325.87
STR-210-011-000	522.19	ooT
DQ/(opt)-100-1-0	0.30 / 0.30	2.93 / 2.97
DQ/(opt)-110-1-0	0.44 / 0.44	1042.88 / 1114.39
DQ/(opt)-110-1-1	0.66 / 0.65	ooT / ooT
DQ/(opt)-111-1-1	1.76 / 2.44	ooT / ooT
DQ/(opt)-211-1-1	9.51 / 37.10	ooT / ooT
DQ/(opt)-211-2-1	28.55 / 111.54	ooT / ooT
QU/(opt)-100-000-000	1.34 / 2.95	2983.11 / ooT
QU/(opt)-100-010-000	2.55 / 5.66	ooT / ooT
QU/(opt)-100-010-010	4.53 / 10.00	ooT / ooT
QU/(opt)-100-100-010	712.57 / 4984.94	ooT / ooT
QU/(opt)-110-011-000	589.50 / ooT	ooT / ooT
QU/(opt)-110-100-010	2108.12 / ooT	ooT / ooT
QU/(opt)-200-010-010	531.41 / ooT	ooT / ooT
QU/(opt)-200-020-000	286.99 / 10585.10	ooT / ooT

Table 3. Runtimes in seconds. ooT = more than four hours.