SC Proofs

Christopher Pulte
christopher.pulte@cl.cam.ac.uk

November 7, 2016

1 POP preserves ARM definition of single-copy atomicity

Lemma 1. *Let r be a committed read. If part of its return value was read from a write w', its return value cannot have a part read from a write w that are coherence-hidden behind w', assuming r, w and w' are not misaligned.*

Proof. Assume a trace *tr* where the return value of r was partly read from w' but where it also contains a part for footprint *fp* that was read from a write w where *fp* is coherence-hidden by w'. Let $\widetilde{s}$ be the state in *tr* where *r* is last restarted. Then in $\widetilde{s}$ the read *r* is unsatisfied.

Now there are two cases for the trace following $\widetilde{s}$: either (1) *r* first reads from *w*, or (2) *r* first reads from w'.

1. Let *s* be the state after $\widetilde{s}$ before *r* reads from *w* and s' the state afterwards. Then by definition of satisfy-read-cand there is an edge $w \rightarrow r$ in *s*.order-constraints and there is no *e* with $w \rightarrow e \rightarrow r$ in *s*.order-constraints. Now there are two cases: either (a) *w* and w' are order-constraints-related in *s* or (b) not.

   (a) Then the edge must be (w, w') because this edge will be in the (acyclic) order-constraints until the final state and this edge determines the coherence between *w* and w', which by assumption is $w \xrightarrow{co} w'$. As by assumption *r*, *w*, and w' cover the footprint *fp* and since *r* is propagated to the same threads as *w*, *r* must be related with w' as well. And as there is no *e* with $w \rightarrow e \rightarrow r$ it must be $w \rightarrow r \rightarrow w'$ in *s*.order-constraints. Now in s' after *r* reads from *w*, either *r* and *w* are swapped in order-constraints if they are not fully propagated and it is $r \rightarrow w \rightarrow w'$ in s'order-constraints, or they are fully propagated and it is $w \rightarrow r \rightarrow w'$ in s'.order-constraints. Since without restarts of *r* no transition before $r's$ satisfaction can delete the edge (r, w') and since *r* reading from w' requires an edge $w' \rightarrow r$ in the (acyclic) order-constraints this contradicts the assumption that *r* reads from w'.

   (b) Then *w* and w' have not been to propagated to any common thread yet. (Otherwise by definition of propagate-action they would be related.) Since

r and w are propagated to the same threads r is not related to w' either. Since w and r cannot be fully propagated it is $r \to w$ in s'.order-constraints and r and w', and w and w' unrelated. In order to allow r to read from w' an edge $w' \to r$ is needed, by definition of satisfy-read-cand. Let s'' be the first state with edge $w' \to r$ added. Since without restarts of r no transition can delete the edges (r, w) and $(w'w)$ in s'' and any following state it is $w' \to r \to w$ in order-constraints, so that in the final state the coherence is determined to $w' \xrightarrow{co} w$ which contradicts the assumption.

2. But then r reads from w the biggest possible footprint, which by assumption includes fp. This contradicts the assumption that r reads $f\,p$ from w.

□

2 Release/Acquire restore SC

Definition 1. An execution is SC if it corresponds to a total order eo on all events that agrees with program order and where the values of reads are determined by eo: for each bit of the read, pick the value from the eo-maximal predecessor write of the read.

Lemma 2. *A write release that is propagated to all threads is order-constraints related with all other write releases in storage.*

Proof. Let w be a write release that is propagated to all threads and w' another write release. Then w is propagated to at least one thread *tid* that w' is propagated to. Now there are two cases: 1. w was propagated to *tid* first; or 2., w' was propagated to *tid* first. 1. Since the reorder condition does not hold for w and w', at the point where w' was propagated to *tid* in some transition t', adding an edge $w' \to w$ if there was no edge already. Case 2 is symmetric. □

Corollary 1. *In a final state s all write releases are totally ordered.*

Proof. As s is a final state all write releases are fully propagated. By Lemma 2 any two write releases are now order-constraints-related. Since order-constraints is acyclic there is a total order on all write releases. □

Lemma 3. *A program with only acquire reads and release writes does not have restarts in any execution.*

Proof. Restarts are caused for two reasons: reads being issued out of order and thread-internal forwarding of writes to reads. Since reads can only be issued if all previous read-acquires are already issued, programs with only acquire reads cannot issue reads out of order. Since read-acquires cannot be forwarded to a program with only acquire reads and release writes cannot have local forwarding. Therefore there are no restarts. □

Lemma 4. *Let (e, e') in s.order-constraints and s' such that $s \xrightarrow{t} s'$ for some t. If e, and e' are write releases, then (e, e') in s'.order-constraints. If not, the edge (e, e') is in s'.order-constraints unless t is a read-satisfy transition or t causes a restart.*

Proof. By case analysis on the transition types. □

Lemma 5. *If e is fully propagated in s and $s \rightarrow^* s'$, then there cannot be (e', e) in s'.order-constraints if (e', e) is not in s.order-constraints.*

Proof. By case analysis on the transition types. □

Lemma 6. *Let r be a read and ws the writes that satisfy r. When r is fully propagated, all writes w that r reads from have to be in storage.*

Proof. By definition of satisfy-read-cand (w, r) must be in storage in order for r to read from w. If r is fully propagated (w, r) must already be in order-constraints, by Lemma 5 and thus ws must be in storage. □

Lemma 7. *If all instructions are release/acquire instructions, all events are accepted into the storage subsystem in program order.*

Proof. According to the thread semantics any read waits for po-earlier write releases to be committed; any write release for all po-earlier memory accesses to be committed. □

Theorem 1. *An ARM program whose only reads are acquire reads and whose only writes are release writes and whose memory accesses are all aligned has sequentially consistent behaviour.*

Proof. For simplicity assume the ARM program has no barriers. Let $tr = s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} \ldots \xrightarrow{t_{n-1}} s_n$ be a POP trace for a program using only release/acquire writes and reads. By Lemma 3 this trace has no restarts. Since $s := s_n$ is a final state s.order-constraints only contains writes (since all reads are satisfied). By Corolary 1 s.order-constraints is a total order.

Now let pos be a list of the list of events in program order per thread. Then define eo as follows:

eo:= s.order-constraints

Lemma 8. *For any $w \xrightarrow{po} w'$, (w, w') is now in eo.*

Proof. Now by Corollary 1 eo contains all write events. Let $w \xrightarrow{po} w'$ be two write releases from the same thread. Then (w, w') is in eo: when w' is accepted into the storage subsystem w must have already been accepted into the storage subsystem as well, by Lemma 6. When w' is accepted the edge (w, w') is added to order-constraints. By Lemma 4 (w, w') is also contained in s.order-constraints. □

```
embed' po =
  for i in [0 .. length po - 1] {
    if po[i] is read_event {
      let r = po[i] in
      eo := eo union {(w,r) | w IN eo, (w,r) IN rf}                  //A
      eo := eo union {(e,r) | e IN eo, (e,r) IN po}                  //B
      eo := eo union {(r,w) | w IN eo, w is write, (w,r) NIN eo^*} //C
```

```
      eo := eo^*;
    }
  }

embed = map embed' pos
```

First prove that before executing loop i of embed' po, eo is acyclic.

Base case: $i = 0$. Since POP's order-constraints relation is acyclic in any state and since eo is set to s.order-constraints, eo is acyclic.

Step case: $i \to i + 1$. Assume the property holds for i, show it also holds for $i + 1$. Therefore, show that the execution of loop i preserves acyclicity. There are two cases: $po[i]$ is a write event or $po[i]$ is a read event.

Case $po[i]$ is a write event. Then, since the loop does not add to eo this is true by the induction hypothesis.

Case $r = po[i]$ is a read event. By induction hypothesis eo is acyclic before running the loop body for i. Running the body for i adds r into eo. The commands A and B only add edges pointing into r. C only adds edges (r, e) if (e, r) is not already in eo's transitive closure. Therefore eo remains acyclic by construction.

Now prove that before executing the loop i the following holds (INV): Let r be a read from $po[0..i-1]$. Then:

1. eo agrees with program order:
   (a) if (e, r) in po then (e, r) in eo.
   (b) if (r, w) in po for a write w, then (r, w) in eo.
2. For any w in eo: either (w, r) or (r, w) in eo.
3. Let s' be the state when the first part of r was satisfied. The eo-prefix of r restricted to writes is exactly the prefix of r in s'.order-constraints restricted to writes.

Once we have proved the statement above, from this follows that after running embed, eo is a partial order of all events from tr that contains program order. Furthermore the read values determined by the partial order are the same as in the POP trace:

Let r be a read and let s' be the pop state where r is first satisfied. Show the prefix of r in s'.order-constraints now completely determines the read value of r. In s' the read r is fully propagated by definition of satisfy-read-cand. Therefore r is related to all writes in storage. Since for any write to be read from by r there must be an edge (w, r) in order-constraints, and since in s' the read r is already fully propagated, by Lemma 5 any write that r can still read from is in its s'.order-constraints prefix. By full propagation of r all these writes are fully propagated as well and therefore totally ordered. Now for any byte in r's footprint there is a unique maximum according to s.order-constraints-prefix. r will read exactly these maxima, by Lemma 0.

Now the only events that are not ordered in eo are pairs of reads from different threads. Any linear extension of eo will still agree with program order (since all same-thread events are already ordered in a way that agrees with program order) and with the read values from the POP trace (since any read is already ordered with all writes

in a way that is compatible with the read values in the partial order). Then follows SC according to the above definition.

Now show INV by induction on i.

Base case: $i = 0$. 1-3. are vacuously true.

Step case: $i \to i+1$. Assume INV holds for i, show it also holds for $i+1$. Therefore, show that the execution of loop i preserves INV.

There are two cases: $po[i]$ is a write event or $po[i]$ is a read event.

Case $po[i]$ is write event. As $po[i]$ is a write from, the reads from $po[0..i+1-1]$ are the same as from $po[0..i-1]$, and because eo does not change, 1.-3. hold by the induction hypothesis.

Case $r = po[i]$ is a read event.

1. (a) These edges are added by command B.

   (b) By induction hypothesis this is true for all reads in $po[0..i-1]$, only need to show it is also true for $r = po[i]$. Let w be a write such that $r \xrightarrow{po} w$. Show that (r, w) is added to eo. To do that, show that before running C in the loop for i, (w, r) is not in eo and therefore C adds (r, w). Assume (w, r) is in eo.

   Now there are two cases: (i) (w, r) is a transitive edge or (ii) not.

   (i) If it is, look at the immediate predecessor e of r in the transitive reduction of eo on the path (w, r). The path is then $(w, e), (e, r)$, where (e, r) cannot be a transitive edge. Now there are two cases: (A) (e, r) was added because e is po-before r or (B) because e is a write that r read from.

   Before running C in the loop i, embed' only adds edges pointing into r, so (w, e) must have also already been in eo before running loop i.

   (A) But then $e \xrightarrow{po} r \xrightarrow{po} w$, so (e, w) must have been in eo before the execution of the loop for i (if e is a write this is by Lemma 8, if it is a read then by induction hypothesis). But then before running loop i there was a cycle $(w, e), (e, w)$ in eo. Contradiction: eo is acyclic.

   (B) As r is program-order before w, by Lemma 7 w can only have been issued into the storage subsystem after r at which point the storage subsystem added an edge (r, w). By definition of the storage semantics this edge can only be removed by a transition satisfying r. Let s' be the state before r made its first read. In this state r by definition of POP was fully propagated. By definition of satisfy-read-cand the read of r from e is only possible if there is an edge (e, r) in order-constraints. By Lemma 5 this edge must have already been there in s'. Therefore by transitivity the edge (e, w) was in s'.order-constraints. Because all writes are totally ordered in the initial eo and since it stays acyclic, the edge (w, e) must have been in s.order-constraints already, and by Lemma 5 since r and therefore e is fully propagated in s', (w, e) must have already

been in s'.order-constraints. But then we have (w, e) and (e, w) in s'.order-constraints, a contradiction because order-constraints is acyclic.

(ii) Then by assumption that $r \xrightarrow{po} w$, (w, r) must have been added because r read from w. But w was committed into storage after r, adding an edge (r, w). Let s' be the state when r read from w. Since tr does not involve restarts: (r, w) is still in s'.order-constraints, but by definition of $satisfy - read - cand$ also (w, r) in s'.order-constraints. Contradiction to the acyclicity of order-constraints.

2. By induction hypothesis this is already true for all reads in $po[0..i-1]$. Only need to show this for $r = po[i]$. Commands A and B of loop i add certain edges pointing into r. For any event e in eo (including any read) that is not yet directly or by transitivity related to r, command C adds an edge (e, r).

3. By induction hypothesis for all r in $po[0..i-1]$ this is true before running loop i. Have to show (a) that loop i preserves it for $po[0..i-1]$ and (b) that loop i establishes it for $po[i]$.

   (a) Let r' be a read from $po[0..i-1]$. By induction hypothesis the eo-prefix of r' restricted to writes is exactly the prefix of r' in s'.order-constraints restricted to writes, where s' is the state in which r' was first satisfied. Since by induction hypothesis (2.) r' is totally related to all writes in eo before running loop i, since loop i does not add writes into eo, and since embed' preserves acyclicity of eo, $\{w | w \text{ is write}, (w, r') \in eo\}$ is the same before and after the execution of loop i. Then (a) follows.

   (b) Let s' be the state when $r = po[i]$ was first satisfied and ws be the writes satisfying r. By definition of satisfy-read-cand r has to be fully propagated, and for each of the writes w' it reads from eventually (w', r) has to be in order-constraints in the states before r reads from w'. Since r is required to be fully propagated in s', by Lemma 5 edges (w', r) have to already be in s'.order-constraints for all those w' from ws. By Lemma 2 all $w' \in ws$ are totally ordered.

   Let w be the s'.order-constraints-maximal write from ws. As w is fully propagated (by full propagation of r), by Lemma 5 the set $\{e | (e, w) \in s'.\text{order-constraints}\}$ restricted to writes is equal to the set $pref := \{e | (e, w) \in s.\text{order-constraints}\}$ restricted to writes. Since pref is a total order and included in the initial eo, since nothing is ever removed from eo, and since eo stays acyclic: $pref$ = eo-prefix of w restricted to writes before running loop i.

   Only have to show that there is no w' overlapping r in between w and r is in eo before running command C in loop i. Assume such a w' with (w, w') and (w', r) in eo.

   Now there are two cases: (i) the edge (w', r) is a direct edge, or (ii) a transitive edge.

(i) Then (w', r) must have been added either because (A) r read from w', or (B) because w' is po-before r.

(A) We assumed w was the s'.order-constraints-maximal write r read from. As we proved before, all writes w'' from ws—including w'—had edges (w'', r) and therefore must have been fully propagated in s', and therefore ordered with w. By assumption of maximality of w there would have been (w', w) in s'.order-constraints and therefore (w', w) in s.order-constraints and (w', w) in the initial eo. As nothing is ever removed from eo we have (w, w') and (w', w) which contradicts the acyclicity of eo.

(B) Then by Lemma 7 r was accepted into the storage subsystem after w', adding an edge (w', r) to order-constraints. Since tr does not involve restarts, this edge would also be in s'.order-constraints. By definition of satisfy-read-cand in s', w and w' were fully propagated and therefore by Lemma 2 the writes w and w' were related in s'.order-constraints. By the assumption that w was maximal in s'.order-constraints the relation must have been (w', w) in s'.order-constraints. By Lemma 4 (w', w) in s.order-constraints and therefore in eo before the execution of loop i. As there is also (w, w') in eo this contradicts the acyclicity of eo.

(ii) Then let (e, e') be the last edge in one of the paths from w' to r in the transitive reduction of eo so that e is a write and e' is a read. This is well-defined, since every read is satisfied. Since the path from e' to r is a path in the transitive reduction that only involves reads the edges from e' to r must have all been added for program order. Therefore $e' \xrightarrow{po} r$.

Now there are two cases: (e, e') was added because (A) e is po-before e' or (B) because e' read from e.

(A) Then by Lemma 7 r was issued into storage after e, adding an edge (e, r). Then in s' by satisfy-read-cand e would have been fully propagated. As all writes were already totally ordered in the initial eo = s.order-constraints (and by acyclicity of eo) (w', e) was contained in s.order-constraints. But by Lemma 5, since e was fully propagated in s', the edge (w', e) must have already been in s'.order-constraints, and therefore by transitivity also (w', r) in s'.order-constraints. By Lemma 6 w was in storage in s' and by Lemma 2 w' and w must have been related. Assume there would have been (w', w) in s'.order-constraints then (w', w) by Lemma 2 would have been in s.order-constraints and therefore eo, which contradicts the acyclicity of eo. Therefore it must have been (w, w') in s'.order-constraints. But then r would have read from w' so that w' in ws, contradicting maximility of w.

(B) So e is a write that e' read from and $e' \xrightarrow{po} r$. Now by Lemma 7 we know that r was accepted into the storage subsystem after e' which caused adding an edge (e', r) when r was accepted. Now

there are two cases: (a) r first read was before the first read of e' or (b) the first read of e' was before r's first read.

(a) Then (e', r) was also in s'.order-constraints and e' was fully propagated in s'. Since e' is satisfied from e in a state after s' which requires an edge (e, e') to be in order-constraints after s', the edge (e, e') by Lemma 5 must have already been in s'.order-constraints. Therefore (e, r) in s'.order-constraints and e must have been fully propagated; by Lemma 5 w was in storage in s' and by Lemma 2 e and w must have been related, and it must have been (w, e) in s'.order-constraints (otherwise (e, w) would be in eo, contradicting its acyclicity). Since (w', e) is in eo and therefore in s.order-constraints, it must have already been in s'.order-constraints by Lemma 5 as e is fully propagated in s'. By full propagation of e, the edge (w', e) requires w' to have been fully propagated in s' as well. Therefore there must be an edge between w and w', and because (w, w') in eo it must have been (w, w') in s'.order-constraints. But then we have (w, w') and (w', e) and (e, r) in s'.order-constraints which means r would have read from w' which contradicts the assumption that w was s'.order-constraints-maximal.

(b) Then let s'' be the state before the first read of e'. In s'' the edge (e', r) is still in s''.order-constraints, and by definition of satisfy-read-cand e is fully propagated and there is an edge (e, e') in s''.order-constraints. Since (w', e) in eo it must be (w', e) in s.order-constraints; by Lemma 5 $(w', e) \in s''$.order-constraints. Therefore there is an edge (w', r)in s''.order-constraints for the fully propagated w'. Then because of (w, w') in eo, it must have been (w, w') in s.order-constraints and by Lemma 5 (w, w') in s''.order-constraints.
Since w' and r by assumption overlap, and r does not read before s', no transition between s'' and s' deletes (w', r) so (w', r) is in s'.order-constraints. By Lemma 4 also (w, w') in s'.order-constraints. But then in s' the read r would have read from w' which is in contradiction to the assumption that w was the s'.order-constraints-maximal write in ws.

□

3 Barriers restore BSC+SCA

3.1 ARM

By Lemma 1 the SCA part is given in POP even without barriers. Now only need to show that fully-barriered ARM programs without misaligned accesses are BSC.

First prove the following lemma

Lemma 9. *A fully-barriered ARM or POWER program does not have restarts.*

Proof. Restarts are caused for two reasons: reads being issued out of order and thread-internal forwarding of writes to reads. Since reads can only be issued if all previous barriers are committed and therefore all other po-before reads are committed as well, fully-barriered ARM and POWER programs cannot issue reads out of order. Thread-local write forwarding can only happen if the write and a po-later read are issued but not committed yet. But since any read will wait for po-earlier barriers and therefore po-earlier writes to be committed there is no thread-local write forwarding in fully-barriered programs. Therefore there are no restarts. □

The following will assume restartless traces.

Theorem 2. *The behaviour of fully-barriered ARM programs with no misaligned memory accesses is BSC+SCA.*

Proof. Now use a similar approach as in the Release/Acquire SC proof to construct a total order on the byte-sized events that corresponds to *po*, *rf*, and *co*. Let $tr = s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} \ldots \xrightarrow{t_{n-1}} s_n$ be a POP trace with final state $s := s_n$ for a fully-barriered ARM program without misaligned accesses. Split up events from *tr* into byte-sized events. Let *po* be the program order lifted to byte-sized events. Let *rf* and *co* be the per byte reads-from and coherence relations as seen in the POP trace.

eo := *s*.order-constraints lifted to byte-sized events and with barriers removed

Now, since *s*.order-constraints is acyclic, eo is acyclic. Furthermore, if $sw \xrightarrow{po} sw'$ for two subwrites *sw* of *w* and *sw′* of *w′*, then (*sw*, *sw′*) in eo.

Proof. Acyclicity of eo follows from the acyclicity of *s*.order-constraints. As $w \xrightarrow{po} w'$ there is at least one barrier between *w* and *w′*, let *b* be the last one. Committing *w′* requires *b* to be committed, committing *b* requires *w* to be committed. When committing *b* an edge $w \to b$ will be added to order-constraints that no transition can remove. Thus the edge is still in order-constraints when *w′* is committed at which point $b \to w'$ is added to order-constraints, and by transitivity $w \to w'$. As no transition can delete $b \to w'$ from order-constraints we have $w \to w'$ in *s*.order-constraints and therefore (*sw*, *sw′*) in eo. □

Furthermore, if (*sw*, *sw′*) in *co* then (*sw*, *sw′*) in eo.

Proof. Coherence in POP is determined by the final order-constraints. Let *w* be the write of *sw*, *w′* the write of *sw′*. Since $sw \xrightarrow{co} sw'$, (*w*, *w′*) in *s*.order-constraints. Therefore (*sw*, *sw′*) in eo. □

```
embed' po =
  po := remove barriers from po;
  for i in [0 .. length po-1] {
    if po[i] is read_event {
      let r = po[i] in
      eo := eo + {(sw,sr) | sw in eo, sr is subread of r,
```

```
                          sr read from sw}              // A
      eo := eo + {(se,sr) | se in eo, sr is subread of r,
                          se is po-before sr}           // B
      eo := eo + {(sr,sw) | sr is subread of r,
                          sw is subwrite of a write w,
                          (r,w) in po,
                          (sw,sr) not in eo^*}          // C
      eo := eo + {(sr,sw) | sr is subread of r,
                          sw is subwrite of a write w,
                          sr and sw to same address,
                          (sw,sr) not in eo^*}          // D
      eo := eo^*;
    }
  }

embed = map embed' pos
```

Now prove by induction on i that before executing loop i of embed' po, eo is acyclic.

Base case: $i = 0$. This holds by acyclicity of s.order-constraints.

Step case: $i \to i + 1$. Assume the property holds for i, show it also holds for $i + 1$. Therefore, show that the execution of loop i preserves acyclicity. There are two cases: $po[i]$ is a write event or $po[i]$ is a read event.

Case $po[i]$ is a write event. Then, since the loop does not add to eo this is true by the induction hypothesis.

Case $r = po[i]$ is a read event. By induction hypothesis eo is acyclic before running the loop body for i. Running the body for i adds the subreads of r into eo. The commands A and B only add edges pointing into subreads of r. C only adds edges (sr, sw) if (sw, sr) is not already in eo's transitive closure. Neither A, B, or C add edges between the subreads sr. Therefore eo remains acyclic by construction.

Lemma 10. *Let w be a write po-before e in a fully-barriered ARM program. Whenever e is in the storage subsystem of a state s there is an edge (w, b) and (b, e) in s.order-constraints for a barrier b in between w and e.*

Proof. Let b be the last barrer between e and b. When b is in storage b must already be committed, which in turn requires w to be committed. When b is committed (w, b) is added into order-constraints which no transition can remove; when e is accepted in to storage (b, e) is added into order-constraints which can only be deleted by deleting e from the storage. □

Lemma 11. *Let sw be subwrite of a write w and sw' subwrite of a write w' and assume there is a path from sw to sw' in the transitive reduction of eo that includes no other writes but at least one read. Then (w, w') in s.order-constraints.*

Proof. By case analysis on the shapes of the shortest path. The only edges that connect writes to reads are the ones added by commands A and B; the only edges between reads

are the edges added by command B; the only edges that connect reads with writes are the edges from C and D. The possible shapes of the path are therefore:

1. $B^*; C$
2. $B^*; D$
3. $A; B^*; C$
4. $A; B^*; D$

Show that for all of the shapes $(w, w') \in s'$.order-constraints.

1. Let sr, subread of some read r, be the last subread on the path, and let s' be the state before sr is satisfied. Then it is $w \xrightarrow{po} w'$ and therefore by Lemma 10 it is $(w, w') \in s$.order-constraints.

2. Let sr, subread of some read r, be the last subread on the path, and let s' be the state before sr is satisfied. Then it is $w \xrightarrow{po} r$, and (sr, sw') is added by D and therefore sr and sw' are to the same address. Therefore in the transition after s', the subread sr reads from a subwrite $sw*$ of a write $w*$ and sw' is not before $sw*$ in eo. (Otherwise the edge $(sw', sw*)$ combined with the rf-edge $(sw*, sr)$ would have meant (sw', sr) is in eo and D would not have added (sr, sw') to eo.

   Since sw' and $sw*$ must have the same address and since coherence is a subset of eo, this means it must be $sw* \xrightarrow{co} sw'$. Now in the state s' by Lemma 10 there is (w, b) and (b, r) in order-constraints for a barrier b. By definition of satisfy-read-cand there is an edge $(w*, r)$ in s'.order-constraints.

   Now there are two cases: r is fully propagated or not.

   - If it is not, then in the state after satisfying sr with $sw*$ r and $w*$ swap and it is (w, b) and $(b, w*)$ in order-constraints and neither edge can be deleted. Thus when eventually $(w*, w')$ in order-constraints (as this will become the coherence order) it is $(w, w') \in s$.order-constraints.
   - If it is, then $w*$ is fully propagated and in order for it to become $w* \xrightarrow{co} w'$ the edge $(w*, w')$ must already be in order-constraints and the write w' must be fully propagated as well and related with b. If it were (w', b) in order-constraints then w' would be in between $w*$ and r and r would not be able to read from $w*$ as assumed. Therefore it must be $(b, w') \in s'$.order-constraints and therefore (w, w') in s'.order-constraints and (since the barrier does not allow deleting this edge) also s.order-constraints.

3. Let sr, subread of some read r, be the first subread on the path, and let s' be the state before sr is satisfied. Then sr reads from sw and it is $r \xrightarrow{po} w'$.

   In s' by definition of satisfy-read-cand w is propagated to r's thread and w' cannot be in storage yet, since there are uncommitted barriers in between r and w'. Let b be the last such barrier. When b is committed an edge (w, b) is added which no transition can delete; when w' is added (b, w') is added which cannot be deleted.

Thus in all state after the subreads on the path between sw and sw' are satisfied (all the subreads po-before w' it is (w, w') in order-constraints.

4. Let sr, subread of some read r, be the first subread on the path, and sr', subread of some read r' be the last subread on the path. Let s' be the state before sr is satisfied. Then sr' is not in storage yet, and by definition of satisfy-read-cand it w is propagated to the thread of r. Let b be the last barrier between r and r'. When b is accepted into storage there is (w, b) in order-constraints which cannot be deleted; when r' is accepted into storage it is $(b, r') \in$ order-constraints which—in the absence of restarts cannot be deleted before satisfying r'.

   Now let s'' be the state when sr' reads from some subwrite $sw*$ of a write $w*$. As (sr', sw') is added by D, sr' and sw' are to the same address. Also, sw' is not before $sw*$ in eo. (Otherwise the edge $(sw', sw*)$ combined with the rf-edge $(sw*, sr')$ would have meant (sw', sr') is in eo and D would not have added (sr', sw') to eo.

   Since sw' and $sw*$ must have the same address and since coherence is a subset of eo, this means it must be $sw* \xrightarrow{co} sw'$. Now in the state s'' there is (w, b) and (b, r') in order-constraints for barrier b. By definition of satisfy-read-cand there is an edge $(w*, r')$ in s''.order-constraints.

   Now there are two cases: r' is fully propagated or not.

   - If it is not, then in the state after satisfying sr' with $sw*$ r' and $w*$ swap and it is (w, b) and $(b, w*)$ in order-constraints and neither edge can be deleted. Thus when eventually $(w*, w')$ in order-constraints (as this will become the coherence order) it is $(w, w') \in s$.order-constraints.
   - If it is, then $w*$ is fully propagated and in order for it to become $w* \xrightarrow{co} w'$, the edge $(w*, w')$ must already be in order-constraints and the write w' must be fully propagated as well, and thus related with b. If it was $(w', b) \in$ order-constraints then w' would be in between $w*$ and r and r would not be able to read from $w*$ as assumed. Therefore it must be $(b, w') \in s''$.order-constraints and therefore (w, w') in s'.order-constraints and also s.order-constraints (since the barrier does not allow deleting this edge).

□

Corollary 2. *For any path from a subwrite sw of a write w to a subwrite sw′ of some w′ in the transitive reduction of eo it must be* (w, w') *in the final state's order-constraints.*

Proof. Divide the path into subpaths that start from a subwrite and end in a subwrite passing only subreads: whenever there is an edge $(sw*, sw*')$ for to subwrites it must have already been $(w*, w*')$ for their writes in the final order-constraints since embed' does not add direct edges between writes; whenever there is a path from one subwrite $sw*$ to another $sw*'$ with only reads according to the previous Lemma it is also $(w*, w*')$ in order-constraints for the final state for their writes. □

Now prove INV: Before executing the loop i the following holds:
Let r be a read from $po[0..i-1]$ and sr a subread of r. Then

1. eo agrees with program order:
   (a) if (se, sr) in po then (se, sr) in eo.
   (b) if (sr, sw) in po for any subwrite sw, then (sr, sw) in eo.
2. For any w with subwrite sw in eo to the same address as sr: either (sw, sr) or (sr, sw) in eo.
3. Let sw be the subwrite of a write w that satisfied sr. The same-address eo-predecessor of sr is sw.

If we can prove this, we have shown that embed returns a partial order on the byte-sized events that contains program order (we have shown acyclicity of eo before), coherence and is compatible with reads-from. (As coherence for ARM is per write rather than per byte-sized write, the coherence relation trivially is compatible with the *si* relation.) As the partial order already contains program order, coherence, and every subread is already related to all subwrites to the same address, any linear extension of this partial order is a witness that *tr* is a BSC execution. Combined with the result that ARM programs preserve single copy atomicity, we have a proof for the BSC+SCA therorem.

Thus only remains to show INV, by induction on i.

Base case: $i = 0$. 1-3. are vacuously true.

Step case: $i \to i+1$. Assume INV holds for i, show it also holds for $i+1$. Therefore, show that the execution of loop i preserves INV.

There are two cases: $po[i]$ is a write event or $po[i]$ is a read event.

Case $po[i]$ is write event. Then the reads from $po[0..i+1-1]$ are the same as from $po[0..i-1]$, and because eo does not change, 1.-3. hold by the induction hypothesis.

Case $r = po[i]$ is a read event.

1. (a) For all reads in $po[0..i-1]$ this is true by the induction hypothesis; for a subreads sr of a read $r = po[i]$ command B adds these edges.
   (b) By induction hypothesis this is true for all reads in $po[0..i-1]$, only need to show it is also true for $r = po[i]$. Let w be a write such that $r \xrightarrow{po} w$, so $sr \xrightarrow{po} sw$ for any subwrite of w. Show that (sr, sw) is added to eo. To do that, show that before loop i's command C, (sw, sr) is not in eo and therefore C adds (sr, sw). Assume (sw, sr) is in eo.

      Two cases: (sw, sr) is (i) a direct edge or (ii) a transitive edge.

      (i) (i) Then it cannot be a po-edge since it is $r \xrightarrow{po} w$. Therefore assume it is an *rf*-edge. But this cannot happen: sw can only be accepted into storage once all barriers between r and w are committed which in turn requires r to already be satisfied.

      (ii) (ii) Now let (se, sr) be the last edge in the path from sw to sr in the transitive reduction of eo. Now this was either (A) added by A as a po edge or (B) by B as an *rf* edge.

(A) Then it is $se \xrightarrow{po} sr \xrightarrow{po} sw$ and by induction hypothesis and the fact that all write pairs are related according to po it is (se, sw) in eo. But as se is on the path from sw to sr we also have (sw, se) in eo which contradicts its acyclicity.

(B) Then se is a subwrite sw' of some write w' that sr read. Let s' be the state before sr read from sw'. According to read-satisfy-cand sw' was propagated to the thread of r and by definition of the thread semantics w was not accepted into storage. When the last barrier b between r and b was accepted into storage (w', b) was added to order-constraints which cannot be deleted; when w was accepted into storage after that (it must have waited for b to be accepted by definition of the thread semantics) (b, w) was added. Therefore it is (w', w) is order-constraints in this and all future states including s, and therefore also (sw', sw) in eo. But we assumed (sw, sw') in eo which contradicts the acyclicity of eo.

2. By induction hypothesis this is true for all sr from $po[0..i-1]$. Since embed' never deletes elements of eo, only need to show this is also true for the case the sr is subread of a read $r = po[i]$. Let sw be subwrite of a write w. Show that loop i relates sw and sr. If not after running commands A to C sr and sw are unrelated then command D sw sr relates them.

3. We have to show two things: (a) for any sr, sw is in the eo-prefix of sr and (b) there is no same-address sw' inbetween sr and sw in eo.

   (a) In case sr is from $po[0..i-1]$ this holds by the induction hypothesis. In case sr is subread of $r = po[i]$ command A adds (sw, sr) to eo.

   (b) Let sr be the subread of some read r and assume there is a same-address subwrite sw' of a write w' with edges $sw \to sw' \to sr$ in eo. Now there are two cases: (i) sw' is from the same thread as sr, or (*ii*) it is not.

      (i) Let s' be the state before sr reads from sw, let b be the last barrier between w' and r. Then by Lemma 10 there is (w', b) and (b, r) in s'.order-constraints. By satisfy-read-cand w is propagated to r's thread and thus must be ordered with b. If it is (b, w) in s'.order-constraints we have (w', w) in order-constraints which due to the barrier cannot be deleted and thus (sw', sw) in eo which contradicts its acyclicity. Therefore it must be $(w, b) \in s'$.order-constraints. But then b is in between r and w so that r must be fully propagated. Since by Corollary 2 (w, w') must eventually be in order-constraints, by Lemma 5 it must already be $(w, w') \in s'$.order-constraints. But this contradicts the assumption that sr can be satisfied in this state since w' is in between w and r'.

      (ii) Then pick the last subwrite $sw*$ of some write $w*$ on the path. Since this cannot be sw the edge between $sw*$ and sr can either be (A) po or (B) $sw*$ is followed by a subread $sr*$ of a read $r*$ so that $(sw*, sr*)$ was added as an rf-edge and $r* \xrightarrow{po} r$.

(A) Let s' be the state before sr is satisfied. Then there is $(w*, b)$ and (b, r) in s'.order-constraints for the last barrier b between $w*$ and r and by satisfy-read-cand w is propagated to r's thread. Thus r must be ordered with b. If it is (b, w) the edge $(w*, w)$ is in s.order-constraints and since the edge cannot be deleted due to the barrier also $(w*, w)$ in s.order-constraints and therefore eo which contradicts its acyclicity. Therefore assume it is (w, b) but then b is in between r and w forcing r to be fully propagated. But since by Corollary 2 it will be (w, w') and $(w', w*)$ in s.order-constraints, by Lemma 5 there must already be (w, w') and $(w', w*)$ are in s'.order-constraints. But this contradicts the assumption that r can read from w since w' is in between them.

(B) Let s' be the state before sr is satisfied. Then $r*$ must have read from $w*$ and $w*$ was propagated to r's thread before the last barrier b between $r*$ and r was committed. When b was committed $(w*, b)$ was added to the order-constraints of this and all future states, when r was accepted (b, r) was added, which by assumption that tr has no restarts cannot be deleted before r is satisfied.
Now in s' the write w must be ordered with b. As before, if we assume (b, w) then it is $(w*, w)$ in s'.order-constraints and also s.order-constraints and therefore eo, contradicting the acyclicity of eo where we already have $(w, w'), (w', w*)$. Thus assume it is $(w, b) \in s'$.order-constraints. But then b is in between w and r and $w*$ must be fully propagated. By Corollary 2 it will eventually be $(w', w*)$ in s.order-constraints. Thus this edge must by Lemma 5 already be in s'.order-constraints forcing full propagation of w'. Also it will by Corollary 2 be (w, w') in s.order-constraints, which by Lemma 5 means it must already be in s'.order-constraints. But then we have (w, w') and by transitivity also the edge (w', r) is in s'.order-constraints. But this contradicts the assumption that in s' the read r can read from w because now w' is in between them.

□

3.2 POWER

The following will assume restartless traces (see Lemma 9).

Theorem 3. *The behaviour of fully-barriered POWER programs with no misaligned memory accesses is BSC+SCA.*

Proof. First: single-copy atomicity of (fully-barriered and not fully-barriered) POWER programs without misaligned accesses follows from the definitions of the write-propagation transition and the read-satisfaction transition: a read-satisfaction is an atomic transition in the POWER model where the storage subsystem returns a write value composed of the write slices that together cover the footprint of the read and that were last propagated to the thread. The order of propagation does not always match coherence, but

by definition of the write-propagation transition when a write propagation happens, a slice of a write is propagated to some thread *tid* if and only if *tid* does not already have a coherence-newer write slice for the same address.

Now use a similar approach as before to construct a total order on the byte-sized events that corresponds to *po*, *rf*, and *co*. Let $tr = s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} \ldots \xrightarrow{t_{n-1}} s_n$ be a trace of the POWER model with final state $s := s_n$ for a fully-barriered POWER program without misaligned accesses. Split up events from *tr* into byte-sized events. Let *po* be the program order lifted to byte-sized events. Let *rf* and *co* be the per byte reads-from and coherence relations as seen in the POWER trace. Define for any state s' in the trace

s'.order = s'.coherence $\cup$ $\{(w_1, w_2)|$ let *tid* be w_2's thread. w_1 and w_2 are separated by a barrier in *tid*'s events-propagated-to and where not $(w_2, w_1) \in s'$.coherence$\}$

By definition of the POWER model order is always acyclic. Now define eo := s.order lifted to byte-sized events and with barriers removed. Since the POWER model maintains the acyclicity of this set, eo is acyclic. Furthermore, if $sw \xrightarrow{po} sw'$ for two subwrites sw of w and sw' of w', then (sw, sw') in eo.

Proof. As $w \xrightarrow{po} w'$ there is at least one barrier between w and w', let b be the last one. Committing w' requires b to be committed, committing b requires w to be committed. If w and w' are to overlapping addresses it cannot be $w' \xrightarrow{co} w$. We have w before b before w' in the events-propagated-to list for the thread of w', separated by b. Thus by construction they are contained in eo. □

Furthermore, if (sw, sw') in *co* then (sw, sw') in eo, by construction of eo.

```
embed' po =
  po := remove barriers from po;
  for i in [0 .. length po-1] {
    if po[i] is read_event {
      let r = po[i] in
      eo := eo + {(sw,sr) | sw in eo, sr is subread of r,
                            sr read from sw}              // A
      eo := eo + {(se,sr) | se in eo, sr is subread of r,
                            se is po-before sr}           // B
      eo := eo + {(sr,sw) | sr is subread of r,
                            sw is subwrite of a write w,
                            (r,w) in po,
                            (sw,sr) not in eo^*}          // C
      eo := eo + {(sr,sw) | sr is subread of r,
                            sw is subwrite of a write w,
                            sr and sw to same address,
                            (sw,sr) not in eo^*}          // D
      eo := eo^*;
    }
  }

embed = map embed' pos
```

Now prove by induction on i that before executing loop i of embed' po, eo is acyclic.

Base case: $i = 0$. This holds by acyclicity of s.order-constraints.

Step case: $i \to i + 1$. Assume the property holds for i, show it also holds for $i + 1$. Therefore, show that the execution of loop i preserves acyclicity. There are two cases: $po[i]$ is a write event or $po[i]$ is a read event.

Case $po[i]$ is a write event. Then, since the loop does not add to eo this is true by the induction hypothesis.

Case $r = po[i]$ is a read event. By induction hypothesis eo is acyclic before running the loop body for i. Running the body for i adds the subreads of r into eo. The commands A and B only add edges pointing into subreads of r. C only adds edges (sr, sw) if (sw, sr) is not already in eo's transitive closure. Neither A, B, or C add edges between the subreads sr. Therefore eo remains acyclic by construction.

Now prove INV: Before executing the loop i the following holds: Let r be a read from $po[0..i-1]$ and sr a subread of r. Then

1. eo agrees with program order:
   (a) if (se, sr) in po then (se, sr) in eo.
   (b) if (sr, sw) in po for any subwrite sw, then (sr, sw) in eo.
2. For any w with subwrite sw in eo to the same address as sr: either (sw, sr) or (sr, sw) in eo.
3. Let sw be the subwrite of a write w that satisfied sr. The same-address eo-predecessor of sr is sw.

If we can prove this, we have shown that embed returns a partial order on the byte-sized events (we have shown acyclicity of eo before) that contains program order, coherence and is compatible with reads-from. (As coherence for POWER is per write rather than per byte-sized write, the coherence relation trivially is compatible with the si relation.) As the partial order already contains program order, coherence, and every subread is already related to all subwrites to the same address, any linear extension of this partial order is a witness that tr is a BSC execution. Combined with the result that POWER programs preserve single-copy atomicity, we have a proof for the BSC+SCA therorem.

Before showing INV, prove the following lemma.

Lemma 12. *In the following, edges will be called*

- *rf edge if added by A*
- *po edge if added by B or C*
- *fr edge if added by D*

Let se, se′ and se″ be subevents of e, e′, and e″ respectively, let p be a path from se to se′ in eo with no po edges (the ones added by command B and C), let (se', se'') in eo be a po-edge and let s″ be the state before e″ is accepted into the storage subsystem. Then in s″ all reads belonging to the subreads on the path are satisfied and for all subwrites sw on the path either sw or a coherence-successor of sw is propagated to all threads.

Proof. Let b be the last barrier between se' and se''. Since there are no *po* edges all edges between se and se' are *rf*, *fr*, or *co*, thus relating same byte-address events. Now prove by induction on the path length n from se to se'.

Base case: $n = 0$. Then when e'' is committed b must be committed and thus $e = e'$ committed. Thus if e is a read it is satisfied, if it is a write it is committed and b has forced the propagation of se' to all threads that do not already have a coherence-newer subwrite to its address.

Step case: $n \to n + 1$. Let $p*$ be a path of length n from some subevent $se*$ of $e*$ to se'. Show that extending $p*$ to p by adding an edge $(se, se*)$ at the start preserves the property. Then by induction hypothesis all reads except for e (if se is a subread) on the path are satisfied in s'' and for all subwrites except se (if it is a subwrite) either they or a coherence successor are propagated to all threads in s''.

Case e is a write and e' is a write. Only need to show that se or a coherence successor is propagated to all threads in s''. Since se and se' are both subwrites they must be coherence related; as coherence is subset of the acyclic eo it must be $se \xrightarrow{co} se'$. In s'' the barrier b is committed which required committing e'. Therefore a coherence successor of se is propagated to all threads in s''.

Case e is a write and e' is a read. Only need to show that se or a coherence successor is propagated to all threads in s''. As e' is a read the edge at the end of the path is an *rf* edge going from some subwrite sw' of a write w' to se'. Since sw' and se are same byte-address writes they must be coherence related and the coherence must be as in eo: $se \xrightarrow{co} sw'$. When e'' is committed, b is committed which requires e' to be satisfied, by assumption with sw'. In order for this to be possible when e' is satisfied sw'—a coherence successor of se or se itself—is propagated to *tid* and b being committed forces it or a coherence successor to be propagated to all threads that have no coherence-newer subwrite already in s''.

Case e is a read and e' is a write. Only need to show that e is satisfied in s''. As e is a read the edge $(se, se*)$ must be an *fr* edge and $se*$ is a subwrite. As shown before it must be $se* \xrightarrow{co} se'$. In s'' the barrier b is committed which requires committing e'. The barrier can only be committed when all its group A events are propagated—this includes the subwrite se'. Thus if se is to read from something coherence-before $se*$ and thus coherence-before se', e has to be satisfied before se' or a coherence-successor is propagated to the thread of e and therefore before b is committed, before s''.

Case e is a read and e' is a read. Only need to show that e is satisfied in s''. Since e and e' are both reads and there are no *po*-edges there must be a write in between them and $(se, se*)$ is an *fr* edge to a subwrite $se*$, whereas there is some subwrite sw' of a write w' such that (sw', se') is the last edge of the path, an *rf*-edge. Then as shown before $se*$ and sw' must be coherence related as $se* \xrightarrow{co} sw'$. Now in s'' the barrier b is committed which requires satisfying se' with sw'. To make this happen sw' must be propagated to *tid*. The barrier can only be committed when all its group A events or coherence-successors thereof are propagated—this includes the write sw'. Thus if se is to read from something coherence-before $se*$ and thus coherence-before sw', e has to be satisfied before sw' or a coherence-successor is propagated to the thread of e and therefore before b is committed, before s''. □

From this we derive the following:

Corollary 3. *Let p be a path from subevent se of e to subevent se' of e' in eo that ends in a po-edge and let s' be the state before e' is accepted into the storage subsystem. Then in s'' all reads for the subreads in p are satisfied and for all subwrites sw in p either sw or a coherence-successor of sw is propagated to all threads.*

Proof. This follows from a simple inductive proof where the induction is on the number of *po* edges in p where for every subpath of p the previous lemma applies. □

Thus only remains to show INV, by induction on i.

Base case: $i = 0$. 1-3. are vacuously true.

Step case: $i \to i+1$. Assume INV holds for i, show it also holds for $i+1$. Therefore, show that the execution of loop i preserves INV.

There are two cases: $po[i]$ is a write event or $po[i]$ is a read event. In case $po[i]$ is write event the reads from $po[0..i]$ are the same as from $po[0..i-1]$, and because eo does not change, 1.-3. hold by the induction hypothesis.

Case $r = po[i]$ is a read event.

1. (a) For all reads in $po[0..i-1]$ this is true by the induction hypothesis; for a subread sr of a read $r = po[i]$ command B adds these edges.

   (b) By induction hypothesis this is true for all reads in $po[0..i-1]$, only need to show it is also true for $r = po[i]$. Let w be a write such that $r \xrightarrow{po} w$, so $sr \xrightarrow{po} sw$ for any subwrite of w. Show that (sr, sw) is added to eo. To do that, show that before loop i's command C, (sw, sr) is not in eo and therefore C adds (sr, sw). Assume (sw, sr) is in eo.

   Two cases: (sw, sr) is (i) a direct edge or (ii) a transitive edge.

   (i) Then it cannot be a *po*-edge since it is $r \xrightarrow{po} w$. Therefore assume it is an *rf*-edge. But this cannot happen: sw can only be committed into storage once all barriers between r and w are committed which in turn requires r to already be satisfied.

   (ii) Now let (se, sr) be the last edge in the path from sw to sr in the transitive reduction of eo. Now this was either (A) added by A as a *po* edge or (B) by B as an *rf* edge.

   (A) Then it is $se \xrightarrow{po} sr \xrightarrow{po} sw$ and by induction hypothesis and the fact that all write pairs are related according to *po* it is (se, sw) in eo. But as se is on the path from sw to sr we also have (sw, se) in eo which contradicts its acyclicity.

   (B) Then se is a subwrite sw' of some write w' that sr read. Let s' be the state before sr read from sw', let *tid* be r's thread, and let b be the last barrier between r and w in program order. In s' some slices of w' including sw' were propagated to *tid* and thus appended to *tid*'s events-propagated-to list, and b and therefore w are uncommitted. When b is committed (this must be before w is committed) b is appended to events-propagated-to *tid*. When w is committed

in some state s'' it is appended to events-propagated-to *tid* and made coherence-after any writes already propagated to *tid*. Now if w overlaps with w' it will be $w' \xrightarrow{co} w$ and thus (sw', sw) in eo. If not then they are unrelated by coherence but we have the slices of w' including sw' in s''.events-propagated-to *tid* before b before w and thus (w', w) in order. Since no writes or barriers are ever deleted from events-propagated-to, it will therefore be $(w', w) \in s$.order in both cases and therefore (sw', sw) in eo. So in both cases it is (sw', sw) in eo, contradicting the assumption that also (sw, sw') is in the acyclic eo.

2. By induction hypothesis this is true for all sr from $po[0..i-1]$. Since embed' never deletes elements of eo, only need to show this is also true for the case that $r = po[i]$ is a read. Let sr be a subread of r and let sw be subwrite of a write w. If after running commands A to C sr and sw are unrelated then command D sw sr relates them.

3. We have to show two things: (a) for any sr, sw is in the eo-prefix of sr and (b) there is no same-address sw' inbetween sr and sw in eo.

   (a) In case sr is from $po[0..i-1]$ this holds by the induction hypothesis. In case sr is subread of $r = po[i]$ command A adds (sw, sr) to eo.

   (b) Let sr be the subread of some read r and assume there is a same-address subwrite sw' of a write w' with edges $sw \to sw' \to sr$ in eo. Since coherence is included in eo and eo is acyclic it must be $w \xrightarrow{co} w'$.

   Now there are two cases: (i) sw' is from the same thread as sr, or (*ii*) it is not.

      (i) Let s' be the state before sr reads from sw, *tid* r's thread and b the last barrier between w' and r. The read r can only be issued when b is committed which in turn requires w' to be committed. Now there are two cases: sw was propagated to *tid* before or after w' was committed. If sw was propagated to *tid* before w' was committed, when w' was committed it was made coherence after w. But since sw and sw' have the same address w' would have overwritten sw with sw' and r would not have read from sw. So assume w' was committed before the slices of w were propagated to *tid*. Then before the propagation of the slices of w to *tid* by write-propagate-cand w and w' were coherence related, and by assumption the coherence was established to $w \xrightarrow{co} w'$. But then since the slice sw' of w' is to the same address as sw of w, sw would not have been propagated to *tid* and r cannot have read from it, contradicting the assumption.

      (ii) Then look at the last edge on the path from sw via sw' to sr. Since sr reads from sw this edge cannot be an *rf*-edge, therefore it must be a *po*-edge (the other edges do not have reads in their second component). As we have an eo-path from sw' to sr, by Corollary 3 in the state before

sr's read is accepted into the storage subsystem sw' or a coherence-successor is propagated to all threads, including *sr*'s thread. But since $w \xrightarrow{co} w'$ it is $sw \xrightarrow{co} sw'$ this contradicts the assumption that *sr* reads *sw* because in the state of its satisfaction the coherence-newer subwrite sw' or a coherence-successor is already propagated to *sr*'s thread.

□