Caveats:

- `≅` is equality up to renaming of IDs (of events and instructions) and the ID state of the system, and the value of the `next_read_order` field of the thread state. The `next_read_order` value orders read events but as long as the relation between values is preserved the exact value does not matter.
- the proof assumes:
  - apply_tree_context does a correct update of the tree
  - the pending_read cleanup for `T_only transitions` will only be done for reads from forwarded writes
  - instructions returned from `list_old_instructions` can be removed from the instruction tree without affecting the possible transitions.

('Non-memory instruction' means no memory read or write and no barrier.)

Define:

```
p1 = function
| T_only_trans _ _ _ (T_internal _)
| T_only_trans _ _ _ (T_finish _)
| T_only_trans _ _ _ (T_register_read _)
| T_only_trans _ _ _ (T_potential_mem_write _)
| T_only_trans _ _ _ (T_register_write _)
    -> true
| _ -> false

p1' = function
| TSS_fetch _ ioid _ _ _ _ -> (ioid is a branch register instruction)
| _ -> false
```

```
p2 = function
| TSS_fetch _ ioid _ _ fdo _ ->
        (ioid is not a branch register instruction
          ∧ fdo is of the form FDO_success)
| _ -> false
```

Theorem 1

Assume the model is POP.

Let `t, t' in enumerate_transitions_of_system s_0` such that `t ≠ t'`, the condition `p1 t` holds and `p1' t'` does not hold, and let `system_state_after_transition s_0 t = s` and `system_state_after_transition s_0 t' = s'`. Then:

```
( t in enumerate_transitions_of_system s' ∧
  t' in enumerate_transitions_of_system s ∧
  system_state_after_transition s' t ≅ system_state_after_transition s t' )

∨

( t' in enumerate_transitions_of_system s ∧
  system_state_after_transition s t' ≅ s' )
```

Theorem 2

Assume the model is POP.

Let `t, t' in enumerate_transitions_of_system s_0` such that `t ≠ t'` and `p2 t` holds, and `let system_state_after_transition s_0 t = s` and `system_state_after_transition s_0 t' = s'`. Then:

```
( t in enumerate_transitions_of_system s' ∧
  t' in enumerate_transitions_of_system s ∧
  system_state_after_transition s' t ≅ system_state_after_transition s t' )

∨

( t' in enumerate_transitions_of_system s ∧
  system_state_after_transition s t' ≅ s' )
```

Proof of Theorem 1

Let `t = T_only_trans tid ioid ids tt` and `inst` the instruction instance with `inst.ioid = ioid`.

1. Case t' = SS_only_trans t st

Then by definition of `system_state_after_transition`
`s = s_0 with thread_states tid = thread_state_after_transition tt`.

Because of `t' in enumerate_transitions_of_system s_0`, by definition of `enumerate_transitions_of_system`:

```
t' = SS_only st
    in enumerate_transitions_of_storage_subsystem s_0.storage_subsystem =
       enumerate_transitions_of_storage_subsystem s'.storage_subsystem

s' = s_0 with storage_subsystem = st.
```

Because of `t in enumerate_transitions_of_storage_subsystem s_0`, by definition of `enumerate_transitions_of_system`:

```
(ioid, (T_only, idstate'))
    in enumerate_transitions_of_thread (s_0.thread_states tid)
     = enumerate_transitions_of_thread (s'.thread_states tid)
⟹ t in enumerate_transitions_of_system s'
```

```
system_state_after_transition s' t =
system_state_after_transition (s_0 with storage_subsystem st) t ≅
s_0 (with storage_subsystem = st
            thread_states tid = thread_state_after_transition tt) ≅
system_state_after_transition (s_0 with thread_states tid =
                                                thread_state_after_transition tt) t
' =
system_state_after_transition s t'
```

2. Case t' = SS_lazy_trans (SS_POP_read_response read source st)

---

then by definition of `enumerate_transitions_of_system` :

```
SS_interact_lazy in pop_ss_enumerate_transitions (s_0.storage) =
                    pop_ss_enumerate_transitions (s.storage)

⟹ t' in enumerate_transitions_of_system s
```

Let `inst'` be the instruction instance with `inst'.ioid = read.r_ioid` .

By definition of `pop_satisfy_read_action t'` , for any instruction `inst''` : either `inst''` in `s'` is unchanged from `inst''` in `s_0` ; or `inst'' = inst'` and is updated according to `pop_satisfy_read_action` ; or `inst''` is restarted by `t'` .

Then `inst ≠ inst'` has to hold: Two cases of `inst.micro_op_state` that can enable a `T_only` transition:

1. `MOS_plain _`
2. `MOS_pending_mem_read sr c _`

Assume `inst = inst'`

1. `pop_satisfy_read_action_trans` requires `inst'.micro_op_state` to be of the form `MOS_pending_read` . Contradiction to `inst = inst'` .
2. The only `T_only` transition enabled in `inst.micro_op_state` is `actually_satisfy_transitions` and requires `sr.sr_not_yet_requested = []` and `sr.sr_requested = []` . Because `t'` is enabled, as an invariant of the system state `sr.sr_not_yet_requested ≠ []` or `sr.sr_requested ≠ []` holds. Contradiction to `inst = inst'` . Therefore `inst ≠ inst'` .

Two cases: Taking `t'` in state `s` restarts `inst` or not.

2.1. Case inst restarted by t'

Then `tt` cannot be of form `T_commit_simple` or `T_finish` because by definition of `system_state_after_transition` and `enumerate_transitions_of_instruction` for `t in {T_only tid ioid ids (T_commit_simple _), T_only tid ioid ids (T_finish _)}` and `inst.committed = true` in `s` and committed instructions are not restarted.

By definition of `system_state_after_transition`, `enumerate_transitions_of_thread`, `enumerate_transitions_of_instruction`,

```
s = s_0 with (thread_state tid).instruction_tree with inst updated
             next_read_order updated in the case of tt = T_potential_mem_write _
```

By definition of `system_state_after_transition`, `pop_satisfy_read_action_trans`, and `pop_satisfy_read_action`, and by assumption that `inst` is restarted:

```
s' = s_0 with [ (thread_states tid).instruction_tree
                      with inst' updated and
                           dependent instructions restarted ]
                storage_subsystem cleaned up from old read_requests

≅ (s_0 with (thread_state tid).instruction_tree with inst updated
          with next_read_order updated in the case of tt = T_potential_mem_write _)
               with [ (thread_states tid).instruction_tree
                           with inst' updated and
                                dependent instructions restarted ]
                    storage_subsystem cleaned up from old read_requests

≅ (s_0 with next_read_order updated in the case of tt = T_potential_mem_write _)
               with [ (thread_states tid).instruction_tree
                           with inst' updated and
                                dependent instructions restarted ]
                    storage_subsystem cleaned up from old read_requests

= system_state_after_transition s t' (up to next_read_order value)
```

2.2. Case inst not restarted by t'

By definition of `system_state_after_transition`, `pop_satisfy_read_action_trans`, and `pop_satisfy_read_action`, by `inst ≠ inst'`, and the assumption that `inst` is not restarted, `inst` is not changed by `t'`.

Check that `t` is enabled in `s'` by case analysis on `inst` state in `s_0` that enabled `t`:

2.2.1. Case inst.micro_op_state = MOS_plain with interpreter outcome Read_mem

Then by definition of `enumerate_transitions_of_instruction` the condition `pop_memory_read_request_cand` holds in `s_0`. Then `pop_memory_read_request_cand` also holds in `s'`: The conjuncts of `pop_memory_read_request_cand` are of two forms:

1. requiring all po-before instructions of certain types to be committed
2. The following:

```
forall (prev_inst in inst_context.active_prefix).
is_load_acquire prev_inst ⟹
    (prev_inst.committed v not (Set.null prev_inst.writes_read_from) v
    match prev_inst.micro_op_state with
    | MOS_pending_mem_read sr _ -> sr.sr_not_yet_requested = []
    | _ -> false
```

All requirements of the first form also hold in `s'` because `t'` cannot restart committed instructions.

The second condition concerns load-acquire instructions `prev_inst` po-before `inst`. Assume condition 2 no longer holds. Then there is a load-acquire instruction `prev_inst` in `inst.active_prefix` that has been restarted by `t'`. But then by definition of `pop_satisfy_read_action`, `restart_dependent_subtrees`, `dependent_suffix_to_restart`, and `dependent_suffix_to_restart_helper` the clause `load_after_load_acquire_dependent` would have caused `inst` to be restarted as well, which contradicts the assumption.

Because `inst` is unchanged by `t'` and `pop_memory_read_request_cand` holds in `s'` transition `t` is enabled in `s'`.

2.2.2. Case inst.micro_op_state = MOS_plain with interpreter outcome Write_mem

No conditions to check: `inst` is unchanged, so `t` is still enabled.

2.2.3. Case inst.micro_op_state = MOS_plain with interpreter outcome Write_ea

Doesn't produce transitions `t` for which `p1 t` holds -- nothing to check.

2.2.4. Case inst.micro_op_state = MOS_plain with interpreter outcome Write_memv

No conditions to check: `inst` is unchanged, so `t` is still enabled.

2.2.5. Case inst.micro_op_state = MOS_plain with interpreter outcome Barrier

Doesn't produce transitions `t` for which `p1 t` holds -- nothing to check.

2.2.6 Case inst.micro_op_state = MOS_plain for interpreter outcome Read_reg r

Need to check that `find_reg_read r` in `s'` still finds the same value:

For `find_reg_read` to return a different value, by definition of `enumerate_transitions_of_instruction`, `find_reg_read`, and `reg_writes_to_this_register`, the `reg_writes` field of instructions in `inst.active_prefix ++ inst.old_prefix` has to have been changed by `t'`. The instructions

that have been changed by `t'` are `inst'` and the instructions `insts` that have been restarted. By definition of `pop_satisfy_read_action` `inst'.reg_writes` in `s'` is the same as in `s_0`. Assume some instruction `inst'` in `insts` has an updated `reg_writes` field, so it cannot be `inst'` and thus must have been restarted by `t'`. As in `s_0` instruction `inst` reads from `inst''` (by assumption that `find_reg_read` yields a different result), there must be a register in `inst.regs_in` that is contained in `inst'.regs_out`, but then by definition of `dependent_suffix_to_restart_helper` instruction `inst` must have been restarted as well, which contradicts the assumption. Therefore `find_reg_read r` in `s_0` and `find_reg_read r` in `s` return the same value, and `t` is also enabled in `s'`.

2.2.7. Case inst.micro_op_state = MOS_plain with interpreter outcome Write_reg

No conditions to check: `inst` is unchanged, so `t` is still enabled.

2.2.8. Case inst.micro_op_state = MOS_plain with interpreter outcome Internal

No conditions to check: `inst` is unchanged, so t is still enabled.

2.2.9. Case inst.micro_op_state = MOS_plain with interpreter outcome Footprint

Doesn't produce transitions `t` for which `p1 t` holds -- nothing to check.

2.2.10. Case inst.micro_op_state = MOS_plain with interpreter outcome Done

If `inst` is committed, there is nothing to check and `t` is enabled in `s'`. So assume `inst` is not committed. Then by assumption `inst` is not a memory instruction.

Need to check if `pop_commit_cand` still holds in `s'`. For non-memory instructions `pop_commit_cand` requires `commitDataflow` and `commitControlflow` to hold. `commitDataflow` requires instructions directly feeding into `inst` 's registers to be committed, more formally: Let `iprevs` be the set of po-previous instructions `iprev` for which there exists a register `r in iprev.regs_out` such that `r` is in `inst.regs_i` and there is no po-between instruction `ibetween` with `r in ibetween.regs_out`. Then all instructions in `iprevs` have to be committed.

Let `iprevs` be this set for `s` and `iprevs'` for `s'`.

Assume `iprev'` is an instruction in `iprevs'` which is not committed, and `rs'` all registers in `iprev'.regs_out` and `inst.regs_in` that witness the membership of `iprev'` in `iprevs'`. Since `t'` does not change the `regs_in` or `regs_out` fields of `r_inst` and only sets `regs_in` and `regs_out` of restarted instructions to the empty set, for any `r'` in `rs'` the instruction `iprev'` must be po-before instructions `iprev` from `iprevs` that have `r'` in `iprev.regs_out` but are restarted by `t'` and thus have an empty `regs_out` field in `s'` so that

`iprev'` becomes the "closest" instruction with `regs_out` determined to contain `r'`. But then, the restart of any such instruction `iprev` by definition of `dependent_suffix_to_restart_helper` would cause the restart of `inst`, which contradicts the assumption. Therefore `iprevs = iprevs'`, which have all been committed in `s_0` and thus also in `s'`. Thus `commitDataflow` still holds in `s'`.

`commitControlFlow` requires all previous branch instructions to be committed. Since this is true in `s_0` it still holds in `s'`.

Therefore `t` is also enabled in `s'`.

2.2.11. Case inst.micro_op_state = MOS_pending_mem_read

The only kind of transition `t` for which `p1 t` holds that is enabled in this state is described in `actually_satisfy_transitions`. Since `inst` is unchanged by `t'` the conditions in `actually_satisfy_transitions` still hold and `t` is still enabled in `s'`.

In all cases `t` is still enabled in `s'`.

Now `show system_state_after_transition s t' ≅ system_state_after_transition s' t`. since `t` and `t'` obviously update separate parts of the system state.

3. Case t' = TSS_Flowing_POP_commit_mem_write_exclusive_successful

---

Let `tid'` be the thread that enables `t'` and `inst'` the instruction with `inst'.ioid = ioid'`

Then

```
t' = TSS_Flowing_POP_commit_mem_write_exclusive_successful
               writes prev_bare_write (thread_cont true)
      in enumerate_transitions_of_system s_0
```

from which follows that

```
tie = (ioid', (T_interact_eager T_POP_commit_mem_write_exclusive
                        writes prev_writes thread_cont, ist'))
        in enumerate_transitions_of_thread tid' in s_0
```

for `ioid'.micro_op_state = MOS_potential_mem_write wk ws c`; that `wk` is of the form `Write_exclusive`, `Write_exclusive_release` or `Write_conditional`; that `pop_commit_cand` holds for `tid'`; and that `pop_ss_accept_write_exclusive_success_cand` holds for `s_0'.storage_subsystem`.

From the definition of `enumerate_transitions_of_instruction` follows `inst ≠ inst'` because for `t` to be enabled `inst.micro_op_state` cannot be `MOS_potential_mem_write`.

Show t' in enumerate_transitions_of_system s

If `t` is po-before `t'`, then `t` cannot be of the form `T_only T_commit_simple` for an uncommitted branch: Assume `t` commits an uncommitted branch. Then by definition of `commitControlflow` condition `pop_commit_cand` cannot hold, because `inst'`'s prefix contains an uncommitted branch. Thus when a branch that is inconsistent with the `inst`'s `NIA` is discarded after taking transition `t` no instructions are removed from the instruction tree that are po-before `inst'`. `t` only changes instruction `inst`.

Therefore, since `inst ≠ inst'` is the only instruction po-before `inst'` changed by `t` is `inst` itself, in case it is in `inst'`'s prefix.

As `inst'` is unchanged by `t`, if `pop_commit_cand` also holds in `s`, then `tie` is in `enumerate_transitions_of_thread s`.

By definition of `enumerate_transitions_of_instruction`, the folllowing kinds of conditions are checked by `pop_commit_cand` and hold in `s_0`:

1. `commitDataflow`
2. certain kinds of instructions po-before `inst'` are committed
3. there is a load-exclusive po-before `inst'`
4. all previous memory access addresses have been fully determined and for po-earlier reads to overlapping addresses it is determined which writes they read from (and they cannot be restarted any more)

1 to 4 still hold in s:

1. All instructions that directly feed into `inst'`'s input registers have been committed. Since the `regs_in` and `regs_out` fields of instruction po-before `inst'` are not changed by `t` and committed instructions remain committed this condition is preserved by `t`.
2. As 2 holds in `s_0` this also holds in `s` since committed instructions are not restarted (or "uncommitted").
3. Since `t` does not remove or restart instructions po-before `inst'` and it holds in `s_0` 3 also holds in `s`.
4. As 4 holds in `s_0` and `t` does not restart or remove any po-before `inst'` instructions this also holds in `s'`: Taking `t` only progresses the instruction state of `inst` and does not change the state of other instructions po-before `inst'`.

As 1 to 4 still hold in `s` transition `tie` is still in `enumerate_transitions_of_thread`.

Since `pop_ss_accept_write_exclusive_success_cand` holds in `s_0` and `s.storage_subsystem = s_0 storage_subsystem` the condition `pop_ss_accept_write_exclusive_success_cand` also holds in `s` so that `t' in enumerate_transitions_of_system s`.

Show t in enumerate_transitions_of_system s' or progress by t overwritten by t'

By definition of `pop_commit_mem_store_action`,

```
s'.thread_states tid' =
    s_0.thread_state tid' with
        (inst' updated to include writes in committed_mem_writes,
                with committed = true
                      micro_op_state updated)
        dependent instructions restarted
```

Two cases: `inst` restarted or `inst` not restarted.

3.1. Case inst restarted

Then `system_state_after_transition s t' ≅ s'` (up to IDs and `next_read_order`). The proof is the same as in 2.1.

3.2. Case inst not restarted.

By definition of `pop_commit_mem_store_action` any instruction changed by transition `t'` is either `inst'` or is restarted by `t'`.

Then `t in enumerate_transitions_of_system s'` and `system_state_after_transition s t' ≅ system_state_after_transition s' t`.

The proof is the same as in Case 2.2.

4. Case t' = TSS_fetch tid' ioid' ids' addr' fdo' tc'

Then by definition of `enumerate_transitions_of_system s_0`

```
tfetch = (ioid',(T_interact_eager T_fetch addr tc,ids'))
    in enumerate_transitions_of_thread tid'
```

and `fdo = s_0.program_memory addr`. Then by definition of `enumerate_transitions_of_thread tfetch` is either in `enumerate_fetch_transitions_of_instruction iic'` for some `iic'` with `iic'.iic_instance.instance_ioid = ioid'` or `enumerate_initial_fetch_transitions_of_thread tid'`.

Case tfetch in enumerate_initial_fetch_transitions

If `tfetch` is in `enumerate_initial_fetch_transitions_of_thread tid'`, then from the definition of `enumerate_initial_fetch_transitions_of_thread` follows `tid ≠ tid'`, because the function requires an empty instruction tree. Thus `t ≠ t'` cannot be enabled in `tid'` and `tid ≠ tid'` follows. In Case `tfetch` is enabled by

`enumerate_initial_fetch_transitions_of_thread` . Thus by assumption `t' in enumerate_transitions_of_system s` as well, because by `tid ≠ tid'` : `s.thread_states.tid' = s_0.thread_states.tid'` and `s.storage_subsystem = s_0.storage_subsystem` .

Now there are two cases: `fdo` of the form `FDO_success address opcode inst init_instruction_state` or otherwise.

In case fdo is of the form `FDO_success` state `s'` is `s_0` with `thread_states tid'` updated to include `inst` in initial instruction state in its instruction tree. From the fact that thread `tid` is not changed by `t'` follows that `t in enumerate_transitions_of_system s'` and because `t` and `t'` update separate parts of the system state `system_state_after_transition s t' ≅ system_state_after_transitions s' t`

In Case `fdo` has the form of a fetch decode outcome error, `s'` is an error state. Since this error state does not have any information that depends on the threads state of `tid` ,

`system_state_after_transition s t' ≅ s'` .

Case tfetch in enumerate_fetch_transitions_of_instruction

Now assume `tfetch` is not in `enumerate_initial_fetch_transitions_of_thread tid'` but in `enumerate_fetch_transitions_of_instruction iic'` for some `iic'` . Let `inst' = iic'.iic_instance` .

Now there are two cases: `fdo` of the form `FDO_success address opcode inst init_instruction_state` or otherwise.

4.1. fdo is fetch decode outcome error

In Case `fdo` has the form of a fetch decode outcome error, `s'` is an error state. Since this error state does not have any information that depends on the threads state of `tid` it follows that `system_state_after_transition s t' ≅ s'` .

4.2. fdo is of the form FDO_success.

Show t in enumerate_transitions_of_system s'

By definition of `enumerate_fetch_transitions_of_instruction` and `system_state_after_transition` transition `t'` does not affect `inst` or any instruction po-before `inst` .

Check that `t` is enabled in `s'` by case analysis on `inst` state in `s_0` that enabled t:

4.2.1. Case inst.micro_op_state = MOS_plain with interpreter outcome Read_mem in s_0

Then by definition of `enumerate_transitions_of_instruction` the condition `pop_memory_read_request_cand` holds in `s_0`. Then `pop_memory_read_request_cand` also holds in `s'`: The conjuncts of `pop_memory_read_request_cand` are of two forms:

1. requiring all po-before instructions of certain types to be committed
2. The following:

```
(forall (prev_inst in inst_context.active_prefix).
    is_load_acquire prev_inst ⟹
        prev_inst.committed ∨ not (Set.null prev_inst.writes_read_from) ∨
        match prev_inst.micro_op_state with
        | MOS_pending_mem_read sr _ -> sr.sr_not_yet_requested = []
        | _ -> false)
```

Both requirements still hold in `s'` as `t'` does not change any instruction po-before `inst`.

4.2.2. Case inst.micro_op_state = MOS_plain with interpreter outcome Write_mem

No conditions to check: inst is unchanged, so `t` is still enabled.

4.2.3. Case inst.micro_op_state = MOS_plain with interpreter outcome Write_ea

Doesn't produce transitions `t` for which `p1 t` holds -- nothing to check.

4.2.4. Case inst.micro_op_state = MOS_plain with interpreter outcome Write_memv

No conditions to check: `inst` is unchanged, so `t` is still enabled.

4.2.5. Case inst.micro_op_state = MOS_plain with interpreter outcome Barrier

Doesn't produce transitions `t` for which `p1 t` holds -- nothing to check.

4.2.6. Case inst.micro_op_state = MOS_plain with interpreter outcome Read_reg r

Need to check that `find_reg_read r` in `s'` still finds the same value:

For `find_reg_read` to return a different value, by definition of `enumerate_transitions_of_instruction`, `find_reg_read`, `reg_writes_to_this_register`, the `reg_writes` field of instructions in `inst.active_prefix ++ inst.old_prefix` has to have been changed by `t'`. `t'` does not change any instruction in `inst`'s prefix, so this condition still holds in `s'`.

4.2.7. Case inst.micro_op_state = MOS_plain with interpreter outcome Write_reg

No conditions to check: `inst` is unchanged, so `t` is still enabled.

4.2.8. Case inst.micro_op_state = MOS_plain with interpreter outcome Internal

No conditions to check: `inst` is unchanged, so `t` is still enabled.

4.2.9. Case inst.micro_op_state = MOS_plain with interpreter outcome Footprint

Doesn't produce transitions `t` for which `p1 t` holds -- nothing to check.

4.2.10. Case inst.micro_op_state = MOS_plain with interpreter outcome Done

If `inst` is committed, there is nothing to check and `t` is enabled in `s'`. So assume `inst` is not committed. Nothing to check either, since this case does not produce any transitions `t` for which `p1 t` holds.

4.2.11. Case inst.micro_op_state = MOS_pending_mem_read

The only kind of transition `t` for which `p1 t` holds that is enabled in this state is described in `actually_satisfy_transitions`. Since `inst` is unchanged by `t'` the conditions in `actually_satisfy_transitions` still hold and `t` is still enabled in `s'`.

In all cases `t` is still enabled in `s'`.

Show t' in enumerate_transitions_of_system s

Need to show that `iic'` is still in `unold_instructions`, and tfetch is still in `enumerate_fetch_transitions_of_instruction iic'` in state `s`.

Assume `iic'` is not in `unold_instructions` anymore. Then `t` moved `inst'` from the instruction tree to `old_instructions`, which by definition of `list_old_instruction` means that `inst'` must have been committed after taking transition `t` and its successor instruction must have been fetched. By definition of `p1` and `enumerate_transitions_of_instruction` transition `t` does not commit any instructions and does not fetch new instructions, so that `inst'`'s unique successor must already have been fetched in `s_0`, which by `enumerate_fetch_transitions_of_instruction` contradicts the assumption that `t'` was enabled in `s_0`. Assume therefore, that `iic'` is still in `unold_instructions`.

Remains to show that `tfetch` is in `enumerate_fetch_transitions_of_instruction iic'` in state `s`, which by definition reduces to showing that

1. the value of `potential_fetch_addresses` remains the same,
2. `already_fetched_addresses` in state `s` does not contain any elements that `already_fetched_addresses` in state `s_0` does not contain, and
3. that `fetch_transition_of_address` returns the same values.

1. As `t'` is enabled in `s_0` the condition `is_stop_fetch_instruction` cannot hold for `iic'`.

    1. Case `iic'` is committed. By definition of `p1'` transition `t` does not commit any instruction. Therefore `iic'` must have been committed in `s_0` and cannot have written to the PC, so that `next_address_of_committed_instruction` returns the same value in `s'` as in `s_0`.
    2. Case `iic'` is not committed. As `inst'` is not the direct po-successor of a branch instruction the function returns the same `successor_fetch_address` value as in `s_0`.

2. This condition holds because `t` does not add new elements to the instruction tree or change any instruction's `program_loc` field.

3. This reduces to showing that `ioids_feeding_address` and therefore `starting_inst_instance` returns the same value in state `s` as in `s_0`. An instruction `inst''`'s `reg_writes` field might have been changed by `t` when doing a register write, but that does not change `inst'`'s `ioids_feeding_address` list, as the exhaustive interpreter already finds the the register write already when initially analysing `inst''`.

Therefore `t' in enumerate_transitions_of_system s`.

Show system_state_after_transition s t' ≅ system_state_after_transition s' t

This simply follows from the fact that `t` and `t'` update separate parts of the system state.

5. Case t' = T_lazy_trans tid' ioid' ist' tt'

Then by definition of `enumerate_transitions_of_system`,

```
tlazy = (ioid',(T_interact_lazy tt',ids')) in enumerate_transitions_of_thread tid'
```

in state `s_0`. Let `inst'` be the instruction instance with `inst'.instance_ioid = ioid'`.

5.1. Case tt' = T_mem_read_request rr rr_slices t

Then by definition of `enumerate_transitions_of_system`,
`pop_ss_accept_event_cand s_0.storage_subsytem (FRead rr rr_slices [])` holds and
`(ioid', (T_interact_lazy (T_mem_read_request rr rr_slices t'), ist'))` is in
`enumerate_transitions_of_instruction inst'` for
`micro_op_state = MOS_pending_mem_read sr c`.

Because `inst'`'s `micro_op_state` is `MOS_pending_mem_read` it follows that `inst ≠ inst'` must hold, since the only transition `t` for which `p1 t` holds that is enabled for `inst.micro_op_state = MOS_pending_mem_read` requires `sr.sr_not_yet_requested` to be empty whereas for `tt'` to be enabled `sr.not_yet_requested` must be non-empty. So assume `inst ≠ inst'`.

Show t' in enumerate_transitions_of_system s

Since `pop_ss_accept_event_cand` holds in `s_0`, it must also hold in `s` because `s.storage_subystem = s_0.storage_subsystem`. Remains to check that `tt' in enumerate_transitions_of_instruction` in `s`.

By definition of `enumerate_transitions_of_instruction` and the fact `inst ≠ inst'` transition `t` does not change `inst'` so that `tt'` is still enabled in `s`. Therefore `t' in enumerate_transitions_of_system`.

Show t in enumerate_transitions_of_system s'

By definition of `enumerate_transitions_of_system`, `pop_ss_accept_event_action`, and `enumerate_transitions_of_instruction` transition `t'` only updates `inst'` and the storage subsystem state.

Check that `t in enumerate_transitions_of_instruction in s'` by case analysis on the `micro_op_state` that enabled `t` in `s_0`.

5.1.1. Case inst.micro_op_state = MOS_plain for interpreter outcome Read_mem

Then `pop_memory_read_request_cand` holds in `s_0`. `pop_memory_read_request_cand` requires

1. instruction of certain type po-before `inst` to be comitted
2. that all load-require instructions `inst''` are either committed; or if `inst'' = MOS_pending_mem_read sr' _` then `sr.sr_not_yet_requested = []` must hold; or `inst''.writes_read_from` not empty.

The only instruction updated by `t'` is `inst'`. As 1 and 2 hold in `s_0` 1 is still true in `s` and 2 is still true for all instructions `inst'' ≠ inst'`. Check that 2 holds for `inst'`.

By definition of `enumerate_transitions_of_instruction` instruction `inst'` cannot be committed in `s_0` and cannot have `sr.sr_not_yet_request = []`, otherwise `t'` wouldn't be enabled in `s_0`.Therefore if condition 2 was true for `inst'` in `s_0` the field `inst'.writes_read_from` must have been non-empty. Since `t'` does not change `inst'`'s `writes_read_from` field, `pop_memory_read_request_cand` must still hold in `s'`.

Therefore `t in enumerate_transitions_of_instruction s'`.

5.1.2. Case inst.micro_op_state = MOS_plain for interpreter outcome Write_mem

No condition to check. From the fact that `inst` is not changed by `t'` follows `t in enumerate_transitions_of s'`.

5.1.3. Case inst.micro_op_state = MOS_plain for interpreter outcome Write_memv

No condition to check. Since inst is not changed by `t'` it follows `t in enumerate_transitions_of s'`.

5.1.4. Case inst.micro_op_state = MOS_plain for interpreter outcome Barrier

Then `pop_commit_cand` holds in `s_0`. Check that `pop_commit_cand` still holds in `s'`:

`pop_commit_cand` checks:

1. `commitDataflow inst_context`
2. `commitControlflow inst_context`
3. `pop_commit_barrier_cand`

1. still holds, since `t'` does not change any instruction's `regs_in` or `regs_out` fields and does not "uncommit" instructions.
2. requires conditional branches po-before `inst` to be committed. `inst'` does not affect those instructions and 2 still holds in `s'`.
3. requires instructions of certain type po-before `inst` to be committed and that the memory access of all po-before memory_access are determined. If this was true in `s_0`, this still holds in `s'` since `t'` only progresses `inst'`, by which its memory accesses don't become undetermined.

Therefore `pop_commit_cand` still holds in `s'` and `t in enumerate_transitions_of_instruction` in state `s'`.

5.1.5. Case inst.micro_op_state = MOS_plain for interpreter outcome Read_reg

Only need to check that `find_reg_read` returns the same value.

For `find_reg_read` to return a different value, by definition of `enumerate_transitions_of_instruction`, `find_reg_read`, `reg_writes_to_this_register`, the `reg_writes` field of instructions in `inst.active_prefix ++ inst.old_prefix` has to have been changed by `t'`. Since `t'` does not change any instruction's `reg_writes` field, `find_reg_read` will return the same value and `t` is enabled in `enumerate_transitions_of_instruction` in `s'`.

5.1.6. Case inst.micro_op_state = MOS_plain for interpreter outcome Write_reg

No condition to check. Since `inst` is not changed by `t'` it follows `t in enumerate_transitions_of s'`.

5.1.7. Case inst.micro_op_state = MOS_plain for interpreter outcome Internal

No condition to check. Since `inst` is not changed by `t'` it follows `t in enumerate_transitions_of s'`.

5.1.8. Case inst.micro_op_state = MOS_plain for interpreter outcome Done

No condition to check.

5.1.9. Case inst.micro_op_state = MOS_pending_mem_read sr c

Only need to check that `sr.sr_not_yet_request = []`. Since this condition was true in `s_0` transition `t'` only changes `inst'`, and `inst' ≠ inst`, this must still be true in `s'`.

In all cases `t in enumerate_transitions_of_instruction` in state `s'` and therefore in `enumerate_transitions_of_system s'`.

Show system_state_after_transition s t' ≅ system_state_after_transitions s' t

Since `inst ≠ inst'`, by definition of `enumerate_transitions_of_instruction` and `enumerate_transitions_of_system` transitions `t` and `t'` update separate parts of the system state. Therefore `system_state_after_transition s t' ≅ system_state_after_transition s' t`.

5.2. Case tt' = T_commit_mem_write ws t

Then by definition of `enumerate_transitions_of_system` the condition `pop_ss_accept_event_cand (FWrite write)` holds in `s_0`,

```
tlazy = (ioid',(T_interact_lazy (T_commit_mem_write ws t'), ist'))
    in enumerate_transitions_of_instruction inst'
```

and in state `s_0` the conditions `inst'.micro_op_state = MOS_potential_mem_write wk ws` and `pop_commit_cand` hold.

From the definition of `enumerate_transitions_of_instruction` follows `inst ≠ inst'` since for `t` to be enabled `inst` cannot have `micro_op_state` of form `MOS_potential_mem_write`.

Show t' in enumerate_transitions_of_system s

Since `s.storage_subsystem = s_0.storage_subsystem` the condition `pop_ss_accept_event_cand` also holds in `s`. Therefore still need to check that `tlazy` is in `enumerate_transitions_of_thread` in state `s`.

Check that `find_committed_writes` returns the same value. This follows from the fact that by definition of `enumerate_transitions_of_instruction` transition `t` does not change the `committed_mem_writes` field of any instruction.

Check that `pop_commit_cand` still holds: check the following requirements:

1. `commitDataflow inst_context`
2. `commitControlflow inst_context`
3. `pop_commit_mem_access_cand`

1. Still holds in state `s` because `t` does not change the `regs_in` or `regs_out` fields of instructions and does not "uncommit" instructions.
2. Still holds because comitted instructions are not uncommitted by `t`.
3. checks that instructions of certain kinds po-before `inst'` are committed, that all previous memory accesses are fully determined, and the condition `aarch64_write_commitPrevMightSameAddress_helper`. Since `t` does not uncommit committed instructions the first condition is preserved by `t`.

   `t` only progresses `inst` so that `inst`'s memory accesses remain determined and the second condition still holds.

   Since `t` only in the case of the `actually_satisfy_transitions` transition changes the `micro_op_state` from `MOS_pending_mem_read` to a different `micro_op_state`, only need to check if this transition preserves condition 3. The `actually_satisfy_transitions` transition is only possible for `micro_op_state = MOS_pending_mem_read sr c` with `sr.sr_not_yet_requested = []` which in turn requires read-satisfy transition, but by definition of `pop_satisfy_read_action` any read-satisfy transition updates `writes_read_from` to include the source of the read.

   Therefore when `t` changes `micro_op_state` from `MOS_pending_mem_read` to a different `micro_op_state` the `writes_read_from` field has to be non-empty and the third condition holds.

From `pop_commit_cand` also holds in `s` follows `tlazy in enumerate_transitions_of_thread in s`.

Two cases: `t'` restarts inst or `t'` doesn't restart `inst`.

5.2.1. Case t' restarts inst

Since `s ≅ s_0 with inst updated` (up to `old_instructions` and `next_read_order`)

```
system_state_after_transition s t' ≅
    system_state_after_transition (s_0 with inst updated) t'
```

As by assumption and by definition of `pop_commit_mem_store_action` transition `t'` overwrites `inst` 's state with `restart_inst_instance` it follows that `system_state_after_transition s t' ≅ s'` .

5.2.2. Case t' does not restart inst

By definition of `enumerate_transitions_of_instruction` and `pop_commit_mem_store_action` transition `t'` only changes instructions that are restarted and `inst'` . As by assumption `inst` is not restarted and because of `inst ≠ inst'` instruction `inst` is unchanged.

Show t in enumerate_transitions_of_system s'

This requires showing `t in enumerate_transitions_of_instruction` in state `s'` by case analysis on the `micro_op_state` that enabled `t` in `s_0` .

5.2.2.1. Case inst.micro_op_state = MOS_plain with interpreter outcome Read_mem

Then by definition of `enumerate_transitions_of_instruction` the condition `pop_memory_read_request_cand` holds in `s_0` . Then `pop_memory_read_request_cand` also holds in `s'` : The conditions of `pop_memory_read_request_cand` are of two forms:

1. requiring all po-before instructions of certain types to be committed
2. The following:

```
forall (prev_inst in inst_context.active_prefix).
 is_load_acquire prev_inst ⟹
 (prev_inst.committed ∨ not (Set.null prev_inst.writes_read_from) ∨
 match prev_inst.micro_op_state with
 | MOS_pending_mem_read sr _ -> sr.sr_not_yet_requested = []
 | _ -> false)
```

All requirements of the first form also hold in `s'` because `t'` cannot restart committed instructions.

The second condition concerns load-acquire instructions `prev_inst` po-before `inst` . Assume condition 2 no longer holds. Then there is a load-acquire instruction `prev_inst` in `inst.active_prefix` that has been restarted by `t'` . (Committed instructions are not changed, and other than by restarting `t'` does not change the `writes_read_from` or `micro_op_state` fields.) But then by definition of `pop_commit_mem_store_action` , `restart_dependent_subtrees` , `dependent_suffix_to_restart` , and `dependent_suffix_to_restart_helper` the clause `load_after_load_acquire_dependent` would have caused `inst` to be restarted as well, which contradicts the assumption. Because `inst` is unchanged by `t'` and `pop_memory_read_request_cand` holds in `s'` transition `t` is enabled in `s'` .

5.2.2.2. Case inst.micro_op_state = MOS_plain with interpreter outcome Write_mem

No conditions to check: `inst` is unchanged, so `t` is still enabled.

5.2.2.3. Case inst.micro_op_state = MOS_plain with interpreter outcome Write_ea

Doesn't produce transitions `t` for which `p1 t` holds -- nothing to check.

5.2.3.4 Case inst.micro_op_state = MOS_plain with interpreter outcome Write_memv

No conditions to check: inst is unchanged, so `t` is still enabled.

5.2.3.5. Case inst.micro_op_state = MOS_plain with interpreter outcome Barrier

Doesn't produce transitions `t` for which `p1 t` holds -- nothing to check.

5.2.3.6. Case inst.micro_op_state = Read_reg r

Need to check that `find_reg_read r` in `s'` still finds the same value:

For `find_reg_read` to return a different value, by definition of `enumerate_transitions_of_instruction`, `find_reg_read`, `reg_writes_to_this_register`, the `reg_writes` field of instructions in `inst.active_prefix ++ inst.old_prefix` has to have been changed by `t'`. The instructions that have been changed by `t'` are `inst'` and the instructions `insts` that have been restarted. By definition of `pop_commit_mem_store_action` the field `inst'.reg_writes` in `s'` is the same as in `s_0`. Assume some instruction `inst''` in `insts` has an updated `reg_writes` field, so it cannot be `inst'` and thus must have been restarted by `t'`. As in `s_0` instruction `inst` reads from `inst''` (by assumption that `find_reg_read` returns a different result), there must be a register in `inst.regs_in` that is contained in `inst''.regs_out`, but then by definition of `dependent_suffix_to_restart_helper` instruction `inst` must have been restarted as well, which contradicts the assumption. Therefore `find_reg_read r` in `s'` and `find_reg_read r` in `s` return the same value, and `t` is also enabled in `s'`.

5.2.3.7. Case inst.micro_op_state = MOS_plain with interpreter outcome Write_reg

No conditions to check: `inst` is unchanged, so `t` is still enabled.

5.2.3.8. Case inst.micro_op_state = MOS_plain with interpreter outcome Internal

No conditions to check: `inst` is unchanged, so `t` is still enabled.

5.2.3.9. Case inst.micro_op_state = MOS_plain with interpreter outcome Footprint

Doesn't produce transitions `t` for which `p1 t` holds -- nothing to check.

5.2.3.10. Case inst.micro_op_state = MOS_plain` with interpreter outcome Done

If `inst` is committed, there is nothing to check and `t` is enabled in `s'`. If `inst` is not committed, it doesn't produce transitions `t` for which `p1 t` holds -- nothing to check.

5.2.3.11 Case inst.micro_op_state = MOS_pending_mem_read

The only kind of transition `t` for which `p1 t` holds that is enabled in this state is described in `actually_satisfy_transitions`. Since `inst` is unchanged by `t'` the conditions in `actually_satisfy_transitions` still hold and `t` is still enabled in `s'`.

In all cases `t` is still enabled in `s'`.

Show system_state_after_transition s t' ≅ system_state_after_transition s' t

This follows from the fact that `t` and `t'` update separate parts of the system state: by definition of `enumerate_transitions_of_instructions`, `pop_commit_mem_store_action`, and `system_state_after_transitions`, `t'` changes the storage subsytem state, updates `inst'` and restarts dependent instructions.

By definition of `enumerate_transitions_of_instruction` transition `t` only updates `inst`, `old_instructions` and `tid`'s `next_read_order` field. Because of `inst ≠ inst'` and by assumption that `inst` is not restarted `t` and `t'` update separate parts of the system state, so that `system_state_after_transitions s t' ≅ system_state_after_transitions s' t`.

5.3. Case tt' = T_commit_barrier b t

Then by definition of `enumerate_transitions_of_system` and `enumerate_transitions_of_instruction` condition `pop_ss_accept_event_cand (FBarrier b)` holds in `s_0`,

```
tlazy = (ioid', (T_interact_lazy (T_commit_barrier b t),ist'))
    in enumerate_transitions_of_instruction
```

in `s_0` and `inst'`'s `micro_op_state` is `MOS_plain` with interpreter outcome `Barrier bk is'` with `bk ≠ ISB`, and `pop_commit_cand` holds in `s_0`.

Then `inst ≠ inst'` as to enable `t` instruction `inst`'s `micro_op_state` cannot enable interpreter outcome `Barrier bk is'` for a non-ISB barrier.

Show t' in enumerate_transitions_of_sytem s

As `s.storage_subsystem = s_0.storage_subsystem` condition `pop_ss_accept_event_cand (Fbarrier b)` also holds in `s`. Remains to show `tlazy in enumerate_transitions_of_instruction` in state `s`, which because of `inst ≠ inst'` reduces to showing that `pop_commit_cand` still holds in `s`. The conditions required by `pop_commit_cand` are

1. `commitDataflow`
2. `commitControlflow`
3. `pop_commit_barrier_cand`

The proof that 1 and 2 still hold is the same as in Case 5.2. Remains 3: for non-ISB barriers condition 3 only requires instructions of particular kinds be committed. Since `t` does not "uncommit" any instructions this still holds in `s`.

Therefore `tlazy in enumerate_transitions_of_instruction` in state `s` and thus `t' in enumerate_transitions_of_system s`.

Show t in enumerate_transitions_of_sytem s'

This reduces to showing `t in enumerate_transitions_of_instruction` in state `s'`.

By definition of `system_state_after_transition`, `enumerate_transitions_of_instruction`, and `pop_commit_barrier_action`

```
s' = s_0 with storage_subsytem updated
             inst' updated.
```

Proof by case analysis on the `micro_op_state` in `s_0` that enabled `t`.

5.2.3.1. Case inst.micro_op_state = MOS_plain with interpreter outcome Read_mem

Then by definition of `enumerate_transitions_of_instruction` the condition `pop_memory_read_request_cand` holds in `s_0`. Then `pop_memory_read_request_cand` also holds in `s'`: The conditions of `pop_memory_read_request_cand` are of two forms:

1. requiring all po-before instructions of certain types to be committed
2. The following:

```
forall (prev_inst in inst_context.active_prefix).
is_load_acquire prev_inst ⟹ prev_inst.committed v
(Set.null prev_inst.writes_read_from) v match prev_inst.micro_op_state with
| MOS_pending_mem_read sr _ -> sr.sr_not_yet_requested = []
| _ -> false)
```

Both requirements also hold in `s'` because `t'` cannot restart committed instructions, and because `t'` does not "uncommit" any instructions, changes the `writes_read_from` or `sr.sr_not_yet_requested` fields or the form of the `micro_op_state`. Therefore `t` is enabled in `s'`.

5.2.3.2. Case inst.micro_op_state = MOS_plain with interpreter outcome Write_mem

No conditions to check: `inst` is unchanged, so `t` is still enabled.

5.2.3.3. Case inst.micro_op_state = MOS_plain with interpreter outcome Write_ea

Doesn't produce transitions `t` for which `p1 t` holds -- nothing to check.

5.2.3.4. Case inst.micro_op_state = MOS_plain with interpreter outcome Write_memv

No conditions to check: `inst` is unchanged, so `t` is still enabled.

5.2.3.5. Case inst.micro_op_state = MOS_plain with interpreter outcome Barrier

Doesn't produce transitions `t` for which `p1 t` holds -- nothing to check.

5.2.3.6 Case inst.micro_op_state = MOS_plain with interpreter outcom Read_reg r

Need to check that `find_reg_read r` in `s'` still finds the same value:

For `find_reg_read` to return a different value, by definition of `enumerate_transitions_of_instruction`, `find_reg_read`, `reg_writes_to_this_register`, the `reg_writes` field of instructions in `inst.active_prefix ++ inst.old_prefix` has to have been changed by `t'`. As `t` does not change any instruction's `reg_writes` fields, `t` is still enabled in `s'`.

5.2.3.7. Case inst.micro_op_state = MOS_plain with interpreter outcome Write_reg

No conditions to check: `inst` is unchanged, so `t` is still enabled.

5.2.3.8. Case inst.micro_op_state = MOS_plain with interpreter outcome Internal

No conditions to check: `inst` is unchanged, so `t` is still enabled.

5.2.3.9. Case inst.micro_op_state = MOS_plain with interpreter outcome Footprint

Doesn't produce transitions `t` for which `p1 t` holds -- nothing to check.

5.2.3.10. Case inst.micro_op_state = MOS_plain with interpreter outcome Done

If `inst` is committed, there is nothing to check and `t` is enabled in `s'`. If `inst` is not committed, it doesn't produce transitions `t` for which `p1 t` holds -- nothing to check.

5.2.3.11. Case inst.micro_op_state = MOS_pending_mem_read

The only kind of transition `t` for which `p1 t` holds that is enabled in this state is described in `actually_satisfy_transitions`. Since `inst` is unchanged by `t'` the conditions in `actually_satisfy_transitions` still hold and `t` is still enabled in `s'`.

In all cases `t` is still enabled in `s'`.

Show system_state_after_transition s t' ≅ system_state_after_transition s' t

This follows from the fact that `t` and `t'` update separate parts of the system state: by definition of `enumerate_transitions_of_instruction`, `pop_commit_barrier_action`, and `system_state_after_transitions` transition `t'` changes the storage subsytem state and updates `inst'`. By definition of `enumerate_transitions_of_instruction` transition `t` only updates `inst`, `old_instructions` and `tid`'s `next_read_order` field. Because of `inst ≠ inst'` and by assumption that `inst` is not restarted `t` and `t'` update separate parts of the system state, so that `system_state_after_transitions s t' ≅ system_state_after_transitions s' t`.

6. Case t' = T_only_trans tid' ioid' ids' tt'

---

then either (Case 6.1)

```
t' = T_only_trans tid' ioid' ist' tt'
    for tt' = (T_POP_commit_mem_write_exclusive_fail writes' (thread_cont' false))
```

and `pop_ss_accept_write_exclusive_success_cand'` does not hold and

```
tie = T_interact_eager (T_POP_commit_mem_write_exclusive writes' prev_writes')
        in enumerate_transitions_of_thread
```

and `pop_commit_cand` holds in `s_0`.

or

(Case 6.2) `(ioid',(T_only tt',ids')) in enumerate_transitions_of_thread`

6.1.

Show t' in enumerate_transitions_of_system s

Since `s.storage_subsystem = s_0.storage_subsystem` the condition `pop_ss_accept_write_exclusive_success_cand` still holds in `s`. From the definition of `p1` follows `ioid ≠ ioid'` because for `t` to be enabled `inst`'s `micro_op_state` cannot be of the form `MOS_potential_mem_write`. Therefore `inst'` is unchanged by `t` and it remains to show that `pop_commit_cand` still holds.

Check that `pop_commit_cand` still holds: check the following requirements:

1. `commitDataflow inst_context`
2. `commitControlflow inst_context`
3. `pop_commit_mem_access_cand`

1. Still holds in state `s` because `t` does not change the `regs_in` or `regs_out` fields of instructions and does not "uncommit" instructions.
2. Still holds because comitted instructions are not uncommitted by `t`.
3. checks that instructions of certain kinds po-before `inst'` are committed, that all previous memory accesses are fully determined, and `aarch64_write_commitPrevMightSameAddress_helper` holds. Since `t` does not uncommit committed instructions the first condition is preserved by `t`.

   `t` only progresses `inst` so that `inst`'s memory accesses remain determined, so the second condition still holds.

   Since `t` only in the case of the `actually_satisfy_transitions` transition changes the `micro_op_state` from `MOS_pending_mem_read` to a different `micro_op_state`, only need to check if this transition preserves condition 3. The `actually_satisfy_transitions` transition is only possible for a `micro_op_state` of the form `MOS_pending_mem_read sr c` with empty `sr.sr_not_yet_requested = []`. This in turn requires a read-satisfy transition which by definition of `pop_satisfy_read_action` updates `writes_read_from` to include the source of the read.

   Therefore when `t` changes the `micro_op_state` from `MOS_pending_mem_read` to a different `micro_op_state` the `writes_read_from` field has to be non-empty. Therefore also the third condition holds.

Since `pop_commit_cand` also holds in `s` transition `tie` in `enumerate_transitions_of_thread` in `s` and therefore `t'` in `enumerate_transitions_of_system s`.

Show t in enumerate_transitions_of_system s' or progress made by t overwritten by t'

According to the definition of `pop_commit_mem_store_action` with parameter `maybe_successful` set to `(Just false)` transition `t'` updates `inst'` and restarts instructions that have read from the `writes'` set.

Now there are two cases: `inst` is restarted by `t'` or not. For the case where `inst` is restarted `t` 's progress is overwritten by `t'` and `system_state_after_transition s t' ≅ s'` . The proof is the same as for Case 2.1.

For the case where `inst` is not restarted, `t` in `enumerate_transitions_of_system s'` holds and `system_state_after_transition s t' ≅ system_state_after_transition s' t` . The proof is the same as for Case 2.2.

6.2.

Cases of `inst'.micro_op_state` and interpreter outcome that enable `t'` in `s_0` :

6.2.1. Case of inst'.micro_op_state= MOS_plain and interpreter outcome Read_mem

Then `inst ≠ inst'` , since by definition of `enumerate_transitions_of_instruction` this `micro_op_state` enables only one transition.

Show t' in enumerate_transitions_of_system s

Since `inst ≠ inst'` instruction `inst'` is not changed by `t` , so the proof reduces to showing that `pop_memory_read_request_cand` still holds in `s` .

This requires

1. That certain kinds of instructions po-before inst' are committed and
2. The following:

```
forall (prev_inst in inst_context.active_prefix).
is_load_acquire prev_inst ⟹
    (prev_inst.committed ∨ not (Set.null prev_inst.writes_read_from) ∨
     match prev_inst.micro_op_state with
     | MOS_pending_mem_read sr _ -> sr.sr_not_yet_requested = []
     | _ -> false)
```

1. Still holds in `s` because `t` does not "uncommit" instructions.
2. `t` does not change any instruction's `committed` , or `writes_read_from` field. Therefore, if either of the first clauses of the disjunction was true in `s_0` it is still true in `s` . If both were false, then the third one must have been true. But then `inst` cannot be po-before `inst'` since for `t` to be enabled the `micro_op_state` must be `MOS_plain` . Therefore `t` does not affect the third clause.

From the fact that `pop_memory_read_request_cand` holds in `s` follows `t' in enumerate_transitions_of_system`.

Show t in enumerate_transitions_of_system s'

Since `inst ≠ inst'` instruction `inst` is not changed by `t'`.

Check that `t` is enabled in `s'` by case analysis on `inst` state in `s_0` that enabled `t`:

6.2.1.1. Case inst.micro_op_state = MOS_plain with interpreter outcome Read_mem

The proof is symmetrical to what was just proved.

6.2.1.2. Case inst.micro_op_state = MOS_plain with interpreter outcome Write_mem

No conditions to check: `inst` is unchanged, so t is still enabled.

6.2.1.3. Case inst.micro_op_state = MOS_plain with interpreter outcome Write_ea

Doesn't produce transitions `t` for which `p1 t` holds -- nothing to check.

6.2.1.4. Case inst.micro_op_state = MOS_plain with interpreter outcome Write_memv

No conditions to check: `inst` is unchanged, so `t` is still enabled.

6.2.1.5. Case inst.micro_op_state = MOS_plain with interpreter outcome Barrier

Doesn't produce transitions `t` for which `p1 t` holds -- nothing to check.

6.2.1.6. Case inst.micro_op_state = MOS_plain with interpreter outcome Read_reg r

Need to check that `find_reg_read r` in `s'` still finds the same value:

For `find_reg_read` to return a different value, by definition of `enumerate_transitions_of_instruction`, `find_reg_read`, `reg_writes_to_this_register`, the `reg_writes` field of instructions in `inst.active_prefix ++ inst.old_prefix` has to have been changed by `t'`. Because `t'` does not change any instruction's `reg_writes` field, this `find_reg_read` returns the same value in `s'`.

6.2.1.7. Case inst.micro_op_state = MOS_plain with interpreter outcome Write_reg

No conditions to check: `inst` is unchanged, so `t` is still enabled.

6.2.1.8. Case inst.micro_op_state = MOS_plain with interpreter outcome Internal

No conditions to check: `inst` is unchanged, so `t` is still enabled.

6.2.1.9. Case inst.micro_op_state = MOS_plain with interpreter outcome Footprint

Doesn't produce transitions `t` for which `p1 t` holds -- nothing to check.

6.2.1.10. Case inst.micro_op_state = MOS_plain with interpreter outcome Done

If `inst` is committed, there is nothing to check and `t` is enabled in `s'`. So assume `inst` is not committed. Then by assumption `inst` is not a memory instruction.

For `inst.committed = false` there is nothing to do, since this does not produce a transition `t` for which `p1 t` holds.

6.2.1.11. Case inst.micro_op_state = MOS_pending_mem_read

The only kind of transition `t` for which `p1 t` holds that is enabled in this state is described in `actually_satisfy_transitions`. Since `inst` is unchanged by `t'` the conditions in `actually_satisfy_transitions` still hold and `t` is still enabled in `s'`.

In all cases `t` is still enabled in `s'`.

Now `system_state_after_transition s t' ≅ system_state_after_transition s' t` follows from the fact that `t` and `t'` update separate parts of the system state

6.2.2. Case of inst'.micro_op_state = MOS_plain and interpreter outcome Write_mem

Now there are two cases:

`inst = inst'` or `inst ≠ inst'`

6.2.2.1. Case inst = inst'

Then by definition of `enumerate_transitions_of_instruction` and `p1`,

```
t  = (ioid,  (T_only (T_potential_mem_write ws t_potential), ist))
t' = (ioid', (T_only (T_POP_commit_mem_write_exclusive_fail ws t_fail), ist'))
```

for

```
   tfail = pop_commit_mem_store_action m t iic ws (c false) (Just false)
```

According to `enumerate_transitions_of_instruction` transition `t` updates `inst`'s `micro_op_state` to `MOS_potential_mem_write wk ws c`.

Then, in `s` either `pop_commit_cand` holds for

```
(ioid, (T_interact_eager (T_POP_commit_mem_write_exclusive ws
            load.writes_read_from thread_continuation), ist)))
```

or not. If it does not hold, then

```
[ (T_only (T_POP_commit_mem_write_exclusive_fail ws t_fail), ist))
  for tfail = pop_commit_mem_store_action m t iic ws
        (c false) (Just false) ]
in enumerate_transitions_of_instruction
```

and therefore `t' in enumerate_transitions_of_system`.

If it does hold, then

```
T_only_trans tid ioid ist (T_POP_commit_mem_write_exclusive_fail ws
    (thread_continuation false)) =
T_only_trans tid ioid ist (T_POP_commit_mem_write_exclusive_fail ws t_fail
    for tfail = pop_commit_mem_store_action m t iic ws (c false) (Just false) ]

in enumerate_transitions_of_instruction
```

and therefore `t' in enumerate_transitions_of_system`.

From the definition of `enumerate_transitions_of_instruction` follows `s = s_0` with `inst.micro_op_state = MOS_potential_meM_write wk ws c`.

By definition of `pop_commit_mem_store_action`

```
s' = s_0 with inst.micro_op_state of form MOS_plain (c false)
              inst.committed_mem_writes = [] and
              inst.committed = true
              depedent instructions restarted
```

Therefore

```
system_state_after_transition s t' =
system_state_after_transition
    (s_0 with inst.micro_op_state = MOS_potential_meM_write wk ws c.)
    with inst.micro_op_state of form MOS_plain (c false)
         inst.committed_mem_writes = [] and
         inst.committed = true
         depedent instructions restarted ≅

system_state_after_transition s_0
    with inst.micro_op_state of form MOS_plain (c false)
    inst.committed_mem_writes = [] and
    inst.committed = true
    depedent instructions restarted
```

Since by definition of `dependent_suffix_to_restart_helper` taking `t'` in `s` will restart the same instructions as in `s_0` and `t'` overwrites the changes `t` made to `inst` 's `micro_op_state` : `system_state_after_transition s t' ≅ s'` .

6.2.2.2. Case inst ≠ inst'

6.2.2.2.1. Case t' = T_only T_potential_mem_write

Since `inst ≠ inst'` transition `t` does not change `inst'` , and from the fact that there are no other preconditions to enabling `t'` follows `t' in enumerate_transitions_of_system s` .

Show t in enumerate_transitions_of_system s'

By case analysis on the state of `inst` in `s_0` that enables `t` .

6.2.2.2.1.1. Case inst.micro_op_state = MOS_plain with interpreter outcome Read_mem

This requires

1. That certain kinds of instructions po-before `inst'` are committed and
2. The following:

```
forall (prev_inst in inst_context.active_prefix).
is_load_acquire prev_inst ⟹
    (prev_inst.committed ∨ not (Set.null prev_inst.writes_read_from) ∨
    match prev_inst.micro_op_state with
    | MOS_pending_mem_read sr _ -> sr.sr_not_yet_requested = []
    | _ -> false)
```

1. still holds in `s'` because `t'` does not "uncommit" instructions.
2. `t'` does not "uncommitted" any instruction, does not change any instruction's `writes_read_from` or `sr_not_yet_requested` field, and does not change any instruction's `micro_op_state` from `MOS_pending_mem_read` to a different one. Therefore 2 still holds in `s'` .

6.2.2.2.1.2. Case inst.micro_op_state = MOS_plain with interpreter outcome Write_mem

No conditions to check: `inst` is unchanged, so `t` is still enabled.

6.2.2.2.1.3. Case inst.micro_op_state = MOS_plain with interpreter outcome Write_ea

Doesn't produce transitions `t` for which `p1 t` holds -- nothing to check.

6.2.2.2.1.4. Case inst.micro_op_state = MOS_plain with interpreter outcome Write_memv

No conditions to check: `inst` is unchanged, so `t` is still enabled.

6.2.2.2.1.5. Case inst.micro_op_state = MOS_plain with interpreter outcome Barrier

Doesn't produce transitions `t` for which `p1 t` holds -- nothing to check.

6.2.2.2.1.6. Case inst.micro_op_state = MOS_plain with interpreter outcome Read_reg r

Need to check that `find_reg_read r` in `s'` still finds the same value:

For `find_reg_read` to return a different value, by definition of `enumerate_transitions_of_instruction`, `find_reg_read`, `reg_writes_to_this_register`, the `reg_writes` field of instructions in `inst.active_prefix ++ inst.old_prefix` has to have been changed by `t`. Because `t` does not change any instruction's `reg_writes` field, `find_reg_read` returns the same value in `s'`.

6.2.2.2.1.7. Case inst.micro_op_state = MOS_plain with interpreter outcome Write_reg

No conditions to check: `inst` is unchanged, so `t` is still enabled.

6.2.2.2.1.8. Case inst.micro_op_state = MOS_plain with interpreter outcome Internal

No conditions to check: `inst` is unchanged, so `t` is still enabled.

6.2.2.2.1.9. Case inst.micro_op_state = MOS_plain with interpreter outcome Footprint

Doesn't produce transitions `t` for which `p1 t` holds--nothing to check.

6.2.2.2.1.10. Case inst.micro_op_state = MOS_plain with interpreter outcome Done

If `inst` is committed, there is nothing to check and `t` is enabled in `s'`. So assume `inst` is not committed. Then by assumption `inst` is not a memory instruction.

For `inst.committed = false` there is nothing to do, since this does not produce a transition `t` for which `p1 t` holds.

6.2.2.2.1.11. Case inst.micro_op_state = MOS_pending_mem_read

The only kind of transition `t` for which `p1 t` holds that is enabled in this state is described in `actually_satisfy_transitions`. Since `inst` is unchanged by `t'` the conditions in `actually_satisfy_transitions` still hold and `t` is still enabled in `s'`.

In all cases `t` is still enabled in `s'`.

Now `system_state_after_transition s t' ≅ system_state_after_transition s' t` follows from the fact that `t` and `t'` update separate parts of the system state.

6.2.2.2.2. Case t' = T_only T_POP_commit_mem_write_exclusive_fail

Two cases: Taking `t'` in state `s` restarts `inst` or not.

6.2.2.2.2.1. Case inst restarted by t'

Then `t'` overwrites `t's` progress and `system_state_after_transition s t' ≅ s'`. The proof is the same as for Case 2.1.

6.2.2.2.2.2. Case inst not restarted by t'

Then `t in enumerate_transitions_of_system s'` and `system_state_after_transition s t' ≅ system_state_after_transition s' t`. The proof is the same as for Case 2.2.

6.2.3. Case of inst'.micro_op_state = MOS_plain and interpreter outcome Write_ea

From the definition of `enumerate_transitions_of_instruction` follows `inst ≠ inst'`. Since `t'` does not have other preconditions and because `inst'` is unchanged by `t`:

`t' in enumerate_transitions_of_system s`.

And:

`t in enumerate_transitions_of_system s'`. Proof by case analysis on the state of `inst` in `s_0` that enables `t`. The proof is the same as for case 6.2.2.2.1.

Because `t` and `t'` update separate parts of the system state:
`system_state_after_transition s t' ≅ system_state_after_transition s' t`.

6.2.4. Case of inst'.micro_op_state = MOS_plain and interpreter outcome Write_memv

The proof here is the same as for Case 6.2.

6.2.5. Case of inst'.micro_op_state = MOS_plain and interpreter outcome Barrier

By definition of `enumerate_transitions_of_system` and `enumerate_transitions_of_instruction` transition `t'` has to be of the form `T_only_trans _ _ (T_commit_simple Nothing Nothing tt')` for `tt' = pop_commit_barrier_action m t iic' b is'` and `pop_commit_cand m t iic'` holds for `inst'`.

By definition of `enumerate_transitions_of_instruction inst ≠ inst'`.

Show t' in enumerate_transitions_of_system s

Since `inst ≠ inst'` transition `t` does not change `inst'` and the proof reduces to showing `pop_commit_cand` still holds in `s`.

`pop_commit_cand` checks:

1. `commitDataflow inst_context`
2. `commitControlflow inst_context`
3. `pop_commit_barrier_cand`

1. Still holds, since `t'` does not change any instruction's `regs_in` or `regs_out` fields and does not "uncommit" instructions, this is still true in `s`.
2. requires conditional branches po-before `inst` to be committed. `inst'` does not affect those instructions and 2 still holds in `s'`.
3. requires instruction of certain type po-before `inst` to be committed and that all po-before memory accesses are be determined. If this was true in `s_0`, this still holds in `s` since `t` only progresses `inst'`, by which its memory accesses don't become undetermined.

Therefore `t' in enumerate_transitions_of_system s`.

The proof of `t in enumerate_transitions_of_system s'` and `system_state_after_transition s t' ≅ system_state_after_transition s' t` is the same as for 6.2.2.2.1.

6.2.6) Case of inst'.micro_op_state = MOS_plain and interpreter outcome Read_reg

From the definition of `enumerate_transitions_of_instruction` follows `inst ≠ inst'` and the proof of `t' in enumerate_transitions_of_system` only requires showing that `find_reg_read` returns the same value in `s_0` and in `s`.

Assume `find_reg_read` returns a different value. Then by definition of `find_reg_read`, `reg_writes_to_this_register` must return a different value for an instruction in `inst'`'s prefix, which requires `t` to have changed the `reg_writes` or `reg_outs` field of an instruction. The only transition `t` for which `p1 t` holds that changes either of these is of the form `T_only T_register_write` and only changes `inst`'s `reg_writes` field. So assume `t` is of that

form and writes to the register `r` that `t'` reads from. Since all instructions only write to every register once `inst` cannot have written to `r` before, but `analyse_instruction` must have included `r` in `inst`'s `regs_out` field. Therefore `find_reg_read` must have returned `FRRO_blocked` in `s_0`. But then by definition of `enumerate_transitions_of_instruction` transition `t'` would not have been enabled in `s_0`.

Therefore `t' in enumerate_transitions_of_system s'`.

The proof of `t in enumerate_transitions_of_system s'` and `system_state_after_transition s t' ≅ system_state_after_transition s' t` is the same as for 6.2.2.2.1.

6.2.7. Case of inst'.micro_op_state = MOS_plain and interpreter outcome Write_reg

From the definition of `enumerate_transitions_of_instruction` follows `inst ≠ inst'`.

Since `t'` does not have any precondition and `t` does not change `inst'`, `t' in enumerate_transitions_of_instruction`.

Show `t in enumerate_transitions_of_instruction`:

Case `inst.micro_op_state = MOS_plain` with interpreter outcome `Read_mem` See proof of 6.2.2.2.1.1. Case `inst.micro_op_state = MOS_plain` with interpreter outcome `Write_mem` See 6.2.2.2.1.2. Case `inst.micro_op_state = MOS_plain` with interpreter outcome `Write_ea` See 6.2.2.2.1.3. Case `inst.micro_op_state = MOS_plain` with interpreter outcome `Write_memv` See 6.2.2.2.1.4. Case `inst.micro_op_state = MOS_plain` with interpreter outcome `Barrier` See 6.2.2.2.1.5.

Case `inst.micro_op_state = MOS_plain` with interpreter outcome `Read_reg r`

Need to check that `find_reg_read r` in `s'` still finds the same value: The proof is symmetrical to the one in 6.2.6.

Case `inst.micro_op_state = MOS_plain` with interpreter outcome `Write_reg` See 6.2.2.2.1.7. Case `inst.micro_op_state = MOS_plain` with interpreter outcome `Internal` See 6.2.2.2.1.8. Case `inst.micro_op_state = MOS_plain` with interpreter outcome `Footprint` See 6.2.2.2.1.9. Case `inst.micro_op_state = MOS_plain` with interpreter outcome `Done` See 6.2.2.2.1.10. Case `inst.micro_op_state = MOS_pending_mem_read`. See 6.2.2.2.1.11.

Since `t` and `t'` update separate parts of the system state: `system_state_after_transitions s t' ≅ system_state_after_transition s' t`.

6.2.8. Case of inst'.micro_op_state = MOS_plain and interpreter outcome Internal

From the definition of `enumerate_transitions_of_instruction` follows `inst ≠ inst'`. Therefore `t` does not change `inst'` and since `t'` has no other precondition, `t'` in `enumerate_transitions_of_systems s`.

The proof of `t in enumerate_transitions_of_system s'` and `system_state_after_transition s t' ≅ system_state_after_transition s' t` is the same as for 6.2.2.2.1.

6.2.9. Case of inst'.micro_op_state = MOS_plain and interpreter outcome Footprint

This does not produce a transition `t'` for which `p1 t'` holds.

6.2.10. Case of inst'.micro_op_state = MOS_plain and interpreter outcome Done

From `enumerate_transitions_of_instruction`'s definition follows `inst ≠ inst'`. Therefore `t` does not change `inst'` and the proof of `t' in enumerate_transitions_of_system s` reduces to showing that `pop_commit_cand` still holds. The proof of this is the same as in 5.1.4.

Now there are two cases:

`t'` commits a branch instruction and discards a subtree that contains `inst` or otherwise.

6.2.10.1: t' discards inst

Since `s ≅ s_0 with inst updated` (up to `old_instructions` and `next_read_order`) and `t'` removes `inst` from the instruction tree, `system_state_after_transition s t' ≅ s'`.

6.2.10.2: t' does not discard inst

The proof of `t in enumerate_transitions_of_system s'`

and `system_state_after_transition s t' ≅ system_state_after_transition s' t` is the same as for 6.2.2.2.1.

6.2.11. Case of inst'.micro_op_state = MOS_pending_mem_read

From the definition of `enumerate_transitions_of_instruction` follows `inst ≠ inst'`: Assume `inst = inst'`. Since `p1 t` holds, t must be an `actually_satisfy_transitions` transition. There is only one transition of this type and it requires `sr.sr_not_yet_requested` to be empty whereas the `write_forward_transitions` requires `sr.sr_not_yet_requested` to be non-empty. Contradiction. Therefore assume `inst ≠ inst'`.

6.2.11.1.

In the case that `t'` is an `actually_satisfy_transitions` transition it only has the precondition that `sr.sr_not_yet_requested` is empty. Since `inst ≠ inst'` transition `t` does not change this and `t'` is still enabled in in `s`.

The proof of `t in enumerate_transitions_of_system s'` and `system_state_after_transition s t' ≅ system_state_after_transition s' t` is the same as for 6.2.2.2.1.

6.2.11.2.

If `t'` is a `write_forward_transitions` transition, then the only precondition to `t'` being enabled in `s` is that `sr.sr_not_yet_requested` in `s` is the same as in `s'`. Because of `inst ≠ inst'` transition `t` does not change `inst'` and this is still true in `s`.

Now there are two cases: `inst` is restarted by `t'` or not.

In the case that `inst` is restarted by `t'` transition `t`'s changes are overwritten by `t'` and `system_state_after_transition s' t ≅ system_state_after_transition s t'`. For the proof see Case 2.1.

Otherwise `t in enumerate_transitions_of_system s'` and `system_state_after_transition s t' ≅ system_state_after_transition s' t`. For the proof see Case 2.2.

Proof of Theorem 2

Let `t = TSS_fetch tid ioid ist' addr fdo tc` such that `p2 t`, and let `fdo = FDO_success addr opcode inst'' init_instruction_state`. By definition of `enumerate_transitions_of_system` this means

```
tfetch = (ioid,(T_interact_eager T_fetch addr tc,ids')) in
    enumerate_transitions_of_thread tid
```

for some `tid`.

Then

`s = s_0 with thread_states tid updated with inst'' added as a leaf in the instruction tre`

1. Case t' = SS_only_trans t st

The proof is the same as for Case 1 in the proof of Theorem 1.

2. Case t' = SS_lazy_trans (SS_POP_read_response read source st)

Show t' in enumerate_transitions_of_system s

then by definition of `enumerate_transitions_of_system` :

```
SS_interact_lazy in pop_ss_enumerate_transitions (s_0.storage) =
                    pop_ss_enumerate_transitions (s.storage)
```

and `t' in enumerate_transitions_of_system s` .

Show t in enumerate_transitions_of_system s'

Let `inst'` be the instruction instance with `inst'.ioid = read.r_ioid` .

By definition of `enumerate_transitions_of_thread` transition `tfetch` is either in `enumerate_fetch_transitions_of_instruction iic` for some `iic` with `iic.iic_instance.instance_ioid = ioid` or `enumerate_initial_fetch_transitions_of_thread tid` .

Case tfetch in enumerate_initial_fetch_transitions

If `tfetch` is in `enumerate_initial_fetch_transitions_of_thread tid` , then from the definition of `enumerate_initial_fetch_transitions_of_thread` follows `tid ≠ read.r_thread` , because the function requires an empty instruction tree which cannot have enabled `t'` . From the fact that thread `tid` by definition of `pop_satisfy_read_action_trans` is not changed by `t'` follows `t in enumerate_transitions_of_system s'` and because `t` and `t'` update separate parts of the system state `system_state_after_transition s t' ≅ system_state_after_transitions s' t` .

Case tfetch in enumerate_fetch_transitions_of_instruction

Now assume `tfetch` is not in `enumerate_initial_fetch_transitions_of_thread tid` but in `enumerate_fetch_transitions_of_instruction iic` for some `iic` . Let `inst = iic.iic_instance` .

By definition of `pop_satisfy_read_action` , for any instruction `inst'''` : either `inst'''` in `s'` is unchanged from `inst'''` in `s_0` , `inst''' = inst'` and is updated according to `pop_satisfy_read_action` , or `inst'''` is restarted by `t'` .

Therefore, only need to show that and `tfetch` is still in `enumerate_fetch_transitions_of_instruction iic` in state `s'`, which by definition reduces to showing that

1. the value of `potential_fetch_addresses` remains the same,
2. `already_fetched_addresses` in state `s` does not contain any elements that already_fetched_addresses in state `s'` does not contain.
3. `fetch_transition_of_address` returns the same value

1. As `t` is enabled in `s_0` the condition `is_stop_fetch_instruction` cannot hold for `iic`.
    1. Case `iic` is committed. Then it cannot write to the PC and `next_address_of_committed_instruction` returns the same value in `s'` as in `s_0`.
    2. Case `iic` is not committed. As `iic` is no branch instruction, `successor_fetch_address` returns `iic.program_loc + 4`. Since `t'` does not change any instruction's `program_loc` field this value is the same in `s` and `s_0`.
2. This condition holds because `t'` does not add new elements to the instruction tree or change any instruction's `program_loc` field.
3. This reduces to showing that `ioids_feeding_address` and therefore `starting_inst_instance` returns the same value in state `s'` as in `s_0`. An instruction `inst'''`'s `reg_writes` field might have been changed by `t` when it is restarted, but that does not change the `ioids_feeding_address` list, as the exhaustive interpreter already finds the register write already when initially analysing `inst'''`.

Therefore `t in enumerate_transitions_of_system s'`.

Now `system_state_after_transition s t' ≅ system_state_after_transition s' t` follows from the fact that `t` and `t'` update separate parts of the system state.

3. Case t' = TSS_Flowing_POP_commit_mem_write_exclusive_successful

---

Let `tid'` be the thread that enables `t'`.

Then

```
t' = TSS_Flowing_POP_commit_mem_write_exclusive_successful writes prev_bare_write
            (thread_cont true)
        in enumerate_transitions_of_system s_0
```

from which follows that

```
tie = (ioid', (T_interact_eager T_POP_commit_mem_write_exclusive writes
                    prev_writes thread_cont, ist'))
        in enumerate_transitions_of_thread tid' in s_0
                for ioid'.micro_op_state = MOS_potential_mem_write wk ws c
```

and that `wk` is of the form `Write_exclusive` , `Write_exclusive_release` or `Write_conditional` ; that `pop_commit_cand` holds for `tid'` ; and that `pop_ss_accept_write_exclusive_success_cand s.model.ss storage write prev_bare_write` holds for `s_0'.storage_subsystem` .

Show t' in enumerate_transitions_of_system s

`t` does not change the state of any instruction in `s_0` 's instruction tree, only adds a leaf to the instruction tree for the instruction that was fetched. Therefore `inst'` and any instruction po-before `inst'` is not changed by `t` .

As `inst'` is unchanged by `t` , if pop_commit_cand also holds in `s` , then `tie` is in `enumerate_transitions_of_thread s` . Show `pop_commit_cand` still holds in `s` .

By definition of `enumerate_transitions_of_instruction` , the folllowing kinds of conditions are checked by `pop_commit_cand` and hold in `s_0` :

1. instructions directly feeding into `inst'` 's input registers are committed
2. certain kinds of po-before instructions of `inst'` are committed
3. there is a po-before `inst'` load-exclusive
4. all previous memory access addresses have been fully determined and for po-earlier reads to overlapping addresses it is determined which writes they read from (and they cannot be restarted any more)

Since all conditions refer to instruction po-before `inst'` which are unchanged by `t` condition `pop_commit_cand` still holds in `s` .

Since `pop_ss_accept_write_exclusive_success_cand` holds in `s_0` and `s.storage_subsystem = s_0.storage_subsystem` , `pop_ss_accept_write_exclusive_success_cand` also holds in `s` so that `t'` in `enumerate_transitions_of_system s` .

Show t in enumerate_transitions_of_system s'

By definition of `enumerate_transitions_of_thread` transition `tfetch` is either in `enumerate_fetch_transitions_of_instruction iic` for some `iic` with `iic.iic_instance.instance_ioid = ioid` or `enumerate_initial_fetch_transitions_of_thread tid` .

Case tfetch in enumerate_initial_fetch_transitions

If `tfetch` is in `enumerate_initial_fetch_transitions_of_thread tid`, then from the definition of `enumerate_initial_fetch_transitions_of_thread` follows `tid ≠ read.r_thread`, because the function requires an empty instruction tree which cannot have enabled `t'`. From the fact that thread `tid` by definition of `pop_satisfy_read_action_trans` is not changed by `t'` follows `t in enumerate_transitions_of_system s'` and because `t` and `t'` update separate parts of the system state
`system_state_after_transition s t' ≅ system_state_after_transitions s' t`.

Case tfetch in enumerate_fetch_transitions_of_instruction

Now assume tfetch is not in `enumerate_initial_fetch_transitions_of_thread tid` but in `enumerate_fetch_transitions_of_instruction iic` for some `iic`. Let `inst = iic.iic_instance`.

By definition of `pop_commit_mem_store_action`, for any instruction `inst''`: either `inst''` in `s'` is unchanged from `inst''` in `s_0`, `inst'' = inst'` and `inst''` is updated according to `pop_commit_mem_store_action`, or `inst''` is restarted by `t'`.

Showing that `tfetch` is still in `enumerate_fetch_transitions_of_instruction iic` in state `s'`, which by definition reduces to showing that

1. the value of `potential_fetch_addresses` remains the same,
2. `already_fetched_addresses` in state `s` does not contain any elements that `already_fetched_addresses` in state `s'` does not contain.
3. `fetch_transition_of_address` returns the same value

1. As `t` is enabled in `s_0` the condition `is_stop_fetch_instruction` cannot hold for `iic`.
    1. Case `iic` is committed. Then it cannot write to the PC and `next_address_of_committed_instruction` returns the same value in `s'` as in `s_0`.
    2. Case `iic` is not committed. As `iic` is no branch instruction, `successor_fetch_address` returns `iic.program_loc + 4`. Since `t'` does not change any instruction's `program_loc` field this value is the same in `s` and `s_0`.
2. This condition holds because `t'` does not add new elements to the instruction tree or change any instruction's `program_loc` field.
3. This reduces to showing that `ioids_feeding_address` and therefore `start_inst_instance` returns the same value in state `s'` as in `s_0`. As `t'` does not change any instruction's `reg_out`, or `reg_writes` fields this still holds in `s'`.

Therefore `t in enumerate_transitions_of_system s'`.

Now `system_state_after_transition s t' ≅ system_state_after_transition s' t` follows from the fact that `t` and `t'` update separate parts of the system state.

4. Case t' = TSS_fetch tid' ioid' ist' addr' fdo' tc'

Two cases: at least one of the transitions is enabled by `enumerate_initial_fetch_transitions_of_thread` or both are enabled by `enumerate_fetch_transitions_of_instruction`.

4.1. t or t' enabled by enumerate_initial_fetch_transitions_of_thread

Then `tid ≠ tid'` as the `enumerate_initial_fetch_transitions` requires an empty instruction tree, in which case there is only one transition enabled for the thread.

Then `s.thread_state tid' = s_0.thread_state tid'`. Therefore `t'` in `enumerate_transitions_of_system s`.

Now there are two cases: `fdo` is an error fetch decode outcome or not.

If `fdo` is an error fetch decode outcome, then taking transition `t'` leads to a whole-system error state and `system_state_after_transition s t' ≅ s'`.

If `fdo` is a `FDO_success` outcome, then `s'.thread_state tid = s_0.thread_state tid` so that `t in enumerate_transitions_of_system s'`.

As `t` and `t'` update separate parts of the system state:

`system_state_after_transition s t' ≅ system_state_after_transition s' t`.

4.2. t and t' enabled by enumerate_fetch_transitions_of_instruction

By assumption `t ≠ t'`, and by definition of `enumerate_fetch_transitions_of_instruction` for `t` and `t'` to be different, `addr ≠ addr'`. (As `already_fetched_addresses`, `outstanding_fetch_addresses`, and `fetch_transition_of_address` are functions only different addresses can produce different transitions).

Since `t` does not change any instruction's state, `enumerate_fetch_transitions_of_instruction` in system state `s` the function `potential_fetch_addreses` returns the same set, including `addr'` the field `already_fetched_addresses` contains the same elements as in state `s_0`, but with addr added, which has been fetch and added to the instruction tree by `t`. Since `addr' ≠ addr` the address `addr'` cannot be in this set and therefore must be contained in `outstanding_fetch_addresses`. Again, since `t` does not change any instruction's state, `fetch_transition_of_address` produces the same result for `addr'` as in `s_0`, so that

`t' in enumerate_transitions_of_system s`.

The proof for `t in enumerate_transitions_of_system s'` is symmetrical. Since `t` and `t'` update separate parts of the system state:
`system_state_after_transition s t' ≅ system_state_state_after_transition s' t` .

5. Case t' = T_lazy_trans tid' ioid' ist' tt'

Then by definition of `enumerate_transitions_of_system` ,

```
tlazy = (ioid',(T_interact_lazy tt',ids')) in
        enumerate_transitions_of_thread tid' in state s_0
```

Let `inst'` be the instruction instance with `inst'.instance_ioid = ioid'` .

5.1. Case tt' is of form T_mem_read_request rr rr_slices t

Then by definition of `enumerate_transitions_of_system` ,
`pop_ss_accept_event_cand s_0.storage_subsytem (FRead rr rr_slices [])` holds and
`(ioid', (T_interact_lazy (T_mem_read_request rr rr_slices t'), ist'))` is in
`enumerate_transitions_of_instruction inst'` for
`micro_op_state = MOS_pending_mem_read sr c.`

Show t' in enumerate_transitions_of_system s

Since `pop_ss_accept_event_cand holds` in `s_0` , it must also hold in `s` because `s.storage_subystem = s_0.storage_subsystem` . Remains to check that `tt'` in `enumerate_transitions_of_instruction in s` .

By definition of `enumerate_fetch_transitions_of_instruction` and `enumerate_initial_fetch_transitions_of_thread` transition `t` does not change `inst'` so that `tt'` is still enabled in `s` . Therefore `t' in enumerate_transitions_of_system s` .

Show t in enumerate_transitions_of_system s'

By definition of `enumerate_transitions_of_system` , `pop_ss_accept_event_action` , and `enumerate_transitions_of_instruction` transition `t'` only updates `inst'` and the storage subsystem state.

Then `t in enumerate_transitions_of_system s'` , since `t'` does fetch any instructions, remove instructions from the instruction tree or change any instruction's `regs_out` or `reg_writes` fields.

By definition of `enumerate_transitions_of_instruction` and `enumerate_transitions_of_system` `t` and `t'` update separate parts of the system state so that `system_state_after_transition s t' ≅ system_state_after_transition s' t` .

5.2. Case tt' is of form T_commit_mem_write ws t

Then by definition of `enumerate_transitions_of_system` the condition `pop_ss_accept_event_cand (FWrite write)` holds in `s_0`,

```
tlazy = (ioid',(T_interact_lazy (T_commit_mem_write ws t'), ist'))
        in enumerate_transitions_of_instruction inst'
```

in state `s_0` and `inst'.micro_op_state = MOS_potential_mem_write wk ws`, and `pop_commit_cand` holds.

Show t' in enumerate_transitions_of_system s

Since `s.storage_subsystem = s_0.storage_subsystem` condition `pop_ss_accept_event_cand` also holds in `s`. Therefore only need to check that `tlazy` is in `enumerate_transitions_of_thread` in state `s`.

Check that `find_committed_writes` returns the same value: follows from the fact that `t` does not change the `committed_mem_writes` field of any instruction.

Check that `pop_commit_cand` still holds: check the following requirements:

1. `commitDataflow inst_context`
2. `commitControlflow inst_context`
3. `pop_commit_mem_access_cand`

1 to 3 all check conditions for instructions po-before `inst'`. Since `t` does not change `inst'`'s prefix, all of them still hold in `s`. Since `pop_commit_cand` also holds in `s` transition `tlazy in enumerate_transitions_of_thread in s`.

Show t in enumerate_transitions_of_system s'

By definition of `enumerate_transitions_of_instruction` and `pop_commit_mem_store_action` transition `t'` only changes instructions that are restarted and `inst'`. `t'` does fetch any instructions, remove instructions from the instruction tree. The proof of `t in enumerate_transitions_of_system s'` is the same as in Case 2 of the proof of Theorem 2.

Therefore `t in enumerate_transitions_of_system s'`.

Because `t` and `t'` update separate parts of the system state: `system_state_after_transitions s t' ≅ system_state_after_transitions s' t`.

5.3. Case tt' is of form T_commit_barrier b t

Then by definition of `enumerate_transitions_of_system` and `enumerate_transitions_of_instruction` condition `pop_ss_accept_event_cand (FBarrier b)` holds in `s_0`,

```
tlazy = (ioid', (T_interact_lazy (T_commit_barrier b t),ist'))
    in enumerate_transitions_of_instruction
```

in `s_0`, `inst'`'s `micro_op_state` is `MOS_plain` with interpreter outcome `Barrier bk is'` and `bk ≠ ISB`, and `pop_commit_cand` holds in `s_0`.

Show t' in enumerate_transitions_of_sytem s

As `s.storage_subsystem = s_0.storage_subsystem` condition `pop_ss_accept_event_cand (Fbarrier b)` also holds in `s`. Remains to show `tlazy in enumerate_transitions_of_instruction` in state `s`, which because `inst'` is not changed by `t` reduces to showing that `pop_commit_cand` still holds in `s`. The conditions require `pop_commit_cand` are

1. `commitDataflow`
2. `commitControlflow`
3. `pop_commit_barrier_cand`

The proof that 1 and 2 still hold is the same as in Case 5.2. Remains 3: 3 for non-ISB barriers only requires instructions of particular kinds be committed. Since `t` does not "uncommit" any instructions this still holds in `s`.

Therefore `tlazy in enumerate_transitions_of_instruction` in state `s` and thus `t'` in `enumerate_transitions_of_system s`.

Show t in enumerate_transitions_of_sytem s'

By definition of `enumerate_transitions_of_instruction` and `pop_commit_barrier_action` transition `t'` only updates `inst'`'s `micro_op_state`, and changes the `committed_barriers` and `committed` fields, does not fetch new instructions or remove instructions from the instruction tree. The proof of `t in enumerate_transitions_of_system s'` now goes as in Case 2 of the proof of Theorem 2.

Because `t` and `t'` update separate parts of the system state: `system_state_after_transitions s t' ≅ system_state_after_transitions s' t`.

6. Case t' = T_only_trans tid' ioid' ids' tt'

For the cases for which `p1 t'` holds the proof is the same as that of Case 4.2, since we always proved `t in enumerate_transitions_of_system`, `t' in enumerate_transitions_of_system` and `system_state_after_transition s t' ≅ system_state_after_transition s' t`.

Remains the proof for those cases of `t'` for which `p1 t'` doesn't hold:
`T_only_trans ioid' tid' _ T_register_write for regissters LR or CTR`,
`T_only_trans ioid' tid' _ T_POP_commit_mem_write_exclusive_fail`
`T_only_trans ioid' tid' _ T_POP_mem_write_footprint`
`T_only_trans ioid' tid' _ T_commit_simple`

Two cases:

1. `t` enabled by `enumerate_initial_fetch_transitions_of_thread` or
2. `t` enabled by `enumerate_fetch_transitions_of_instruction`

If `t` is enabled by `enumerate_initial_fetch_transitions_of_thread`, then by definition of `enumerate_initial_fetch_transitions_of_thread : tid ≠ tid'`, since otherwise `t` would not be enabled.

Then, from `s.storage_subsystem = s_0.storage_subsystem`
and `s.thread_states tid' = s_0.thread_states tid'` follows
`t' in enumerate_transitions_of_system s`; from `s'.program_memory = s_0.program_memory`
and `s'.thread_states tid = s_0.thread_states tid` follows
`t in enumerate_transitions_of_system s'`.

Because `t` and `t'` update separate parts of the system state:

`system_state_after_transition s t' ≅ system_state_after_transition s' t`.

Now assume `t` is enabled by `enumerate_fetch_transitions_of_thread`. Then `t` in `enumerate_fetch_transitions_of_instruction iic for some iic`. Let `inst = iic.iic_instance`.

Proof by case analysis on `t'`:

6.2. T_POP_commit_mem_write_exclusive_fail

The proof of `t' in enumerate_transitions_of_system s` is the same as in Cases 6.1 and 6.2.2 of the proof of Theorem 1, the proof of `t in enumerate_transitions_of_system s'` and `system_state_after_transitions s t' ≅ system_state_after_transition s' t` is the same as in Case 2 of the proof of Theorem 2.

6.3. T_only T_POP_mem_write_footprint

The proof of `t' in enumerate_transitions_of_system s` is the same as in Case 6.2.3 of the Proof of Theorem 1, the proof of `t in enumerate_transitions_of_system s'` and of `system_state_after_transition s t' ≅ system_state_after_transition s' t` is the same as in Case 2 of the proof of Theorem 2.

6.4. T_only (T_commit_simple Nothing Nothing t'),ist')

There are two cases:

`t'` is enabled by `inst'`'s `micro_op_state Barrier` or `Done`. In the case of `inst'.micro_op_state = Barrier` the proof for `t' in enumerate_transitions_of_system s` e is the same as in Case 6.2.5 of the proof of Theorem 1, the proof of `t in enumerate_transitions_of_system s'` and `system_state_after_transition s t' ≅ system_state_after_transition s' t` is the same as in Case 2 of the proof of Theorem 2.

Therefore, assume `t'` enabled by `inst'.micro_op_state = Done`.

Show t' in enumerate_transitions_of_system s

Since `t` does not change `inst'`, only need to show that `pop_commit_cand` still holds in `s`. But since `pop_commit_cand` only refers to instructions in `inst'`'s prefix and `t` does not change `t'`'s prefix, `t` preserves `pop_commit_cand` so that `t' in enumerate_transitions_of_system s`.

Now there are two cases: `potential_fetch_addresses` in `s'` returns the same value or a different value than in `s_0`.

6.4.1. potential_fetch_addresses in s' returns the same value

Since `t'` does not fetch any instruction `out_standing_fetch_addresses` remains the same, and since `t'` does not change any instruction's `regs_out` or `reg_writes` fields `fetch_transitions_of_address` returns the same value as well. Therefore `t in enumerate_transitions_of_system s'` and as `t` and `t'` update separate parts of the system state: `system_state_after_transition s t' ≅ system_state_after_transitions s' t`.

6.4.2. potential_fetch_addresses in s' return a different value than in s.

Because `t'` does not change the instruction kind of `inst'`, does not change its `reg_writes` or `nias` fields, `is_stop_fetch_instruction` the `find_reg_read` and `inst.nias` fields must have the same values in `s'` and in `s_0`. Therefore `t'` must have committed `inst'` and

`potential_fetch_address` returns `{ next_address_of_committed_instruction iic }.` If `addr` is the member of this set, the proof goes as in 6.4.1, so assume `addr` is not in this set.

Then `t'` is the commit of a branch instruction with a now-determined successor address that is different from `adddr` : By definition of `commit_simple_action` transition `t'` deletes any subtrees with address different from `next_address_of_committed_instruction` . Therefore taking `t'` in state `s` removes `inst` from the instruction tree and undoes the progress made by transition `t` , so that `system_state_after_transition s t' ≅ s'` .