An Axiomatic Semantics for Instruction Fetching

Ben Simner[1], Shaked Flur[1], Christopher Pulte[1], Alasdair Armstrong[1], Jean Pichon-Pharabod[1], Luc Maranget[2], and Peter Sewell[1]

[1] University of Cambridge, UK
[2] INRIA Paris, France

Based on the operational instruction fetch model, we develop an axiomatic semantics for instruction fetching, as an extension of the ARMv8 axiomatic reference model [1, 2]. Arm's axiomatic reference model currently does not have mixed size support yet; hence we here do not model the aspect of the concurrent modification of conditional branches allowing the result of a fetch to be some combination of old or new condition with old or new target address, which in essence requires mixed-size machinery – beyond the scope of this paper. We make two semantics-preserving rewrites of the existing model before we add instruction fetches:

- The existing axiomatic model expresses the semantics of various barriers with composite edges: e.g. `[R|W];po;[dmb.sy];po;[R|W]` ("a `DMB SY` barrier creates order from program-order-preceding read or write events to program-order-succeeding read or write events"). We replace such edges with more "fine grained" edges, e.g. splitting the previous edge into the two edges `[R|W];po;[dmb.sy]` and `[dmb.sy];po;[R|W]`, which informally require a barrier to "wait" for preceding reads and writes, and succeeding reads and writes to wait for the barrier. Ignoring instruction fetches, this rewrite does not change the semantics. In the extended model, however, it allows capturing the interaction between the barrier ordering and instruction fetches: e.g. assume some edge (e, i) imposing an ordering constraint on the fetch i of a `dmb.sy` event b, and some read event r program-order-after the `DMB SY`. Then we want r to also be ordered after e. Splitting the edges as above means there is an edge (b, r), and we can specify the rules so (e, i) composes with (b, r).
- The existing ARMv8-A model includes the relations `fre` and `coe` into the definition of the *observed-by* relation, but for presentation not the relation `fri` and `coi`. Including these is equivalent and allows simplifying the definitions: it allows deleting the edges such as "`(ctrl|data);coi`" which are now subsumed by the combination of existing edges with `coi` or `fri` in `ob`.

We now make the following changes and additions to the model. The full model is shown in Figure 1, with comments referring to the items in the following explanation.

1. We define the relation `iseq`, relating some write w to x to an `IC` completing a cache synchronisation sequence (not necessarily on a single thread): w is `wco`-followed by a same-cache-line `DC`, in turn `wco`-followed by the same-cache-line `IC`. In operational model terms, this captures traces that propagated w to memory, subsequently performed a same-cache-line `DC`, and then a began an `IC` (and eagerly propagated the `IC` to all threads). In any state after this sequence it is guaranteed that w, or a coherence-newer same-address write, is in the instruction cache of all threads: performing the `DC` has cleared the abstract data cache of writes to x, and the subsequent `IC` has removed old instructions for location x from the instruction caches, so that any subsequent updates to the instruction caches have been with w, or `co`-newer writes.
2. After the aforementioned two rewrites, the model includes `co` in `obs`; we instead include the relation `wco`. Including `wco` in *ordered-before* corresponds to the intuition that `wco` records the ordering of the Write Propagate, Perform data cache maintenance, and Perform instruction cache maintenance transitions in a matching trace.
3. We also include `irf` in `obs`: informally, for an instruction to be fetched from a write, the write has to have been done before. Correspondingly, in the operational model, a write has to have been propagated before it can satisfy fetches in the storage subsystem.
4. We add to the *observed-by* relation the edge `ifr;iseq`, relating an instruction fetch i to x to an `IC` ic if: i fetched from a write w to x, some write w' to x is coherence-after w, and ic completes a cache synchronisation sequence (`iseq`) starting from w'. Then i must be ordered-before ic, because if it happened "after" ic, the cache synchronisation sequence would force i to read from w' or a coherence-newer write.

```
let iseq = [W];(wco&scl);[DC];  (*1*)
               (wco&scl);[IC]

(* Observed-by *)
let obs = rfe | fr | wco        (*2*)
        | irf | (ifr;iseq)      (*3,4*)

(* Fetch-ordered-before *)
let fob = [IF]; fpo; [IF]       (*5*)
  | [IF]; fe                    (*6*)
  | [ISB]; fe^-1; fpo           (*7*)

(* Dependency-ordered-before *)
let dob = addr | data
  | ctrl; [W]
  | (ctrl | (addr; po)); [ISB]
(*| [ISB]; po; [R] *)          (*8*)
  | addr; po; [W]
  | (addr | data); rfi

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

(* Barrier-ordered-before *)
let bob = [R|W]; po; [dmb.sy]
  | [dmb.sy]; po; [R|W]
  | [L]; po; [A]
  | [R]; po; [dmb.ld]
  | [dmb.ld]; po; [R|W]
  | [A|Q]; po; [R|W]
  | [W]; po; [dmb.st]
  | [dmb.st]; po; [W]
  | [R|W]; po; [L]
  | [R|W|F|DC|IC]; po; [dsb.ish]    (*9*)
  | [dsb.ish]; po; [R|W|F|DC|IC]    (*10*)
  | [dmb.sy]; po; [DC]              (*11*)

(* Cache-op-ordered-before *)
let cob = [R|W]; (po&scl); [DC]     (*12*)
  | [DC]; (po&scl); [DC]            (*13*)

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

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

(* External visibility requirement *)
irreflexive ob as external

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

(* Constrained unpredictable *)
let cff = ([W];loc;[IF]) \          (*14*)
            ob^-1 \ (co;iseq;ob)
cff_bad cff ≡ CU                    (*15*)
```

Fig. 1. Axiomatic model

This corresponds to the operational model in the following way: assume a trace where w was propagated, before w' was propagated, and before the model took a sequence of transitions containing a cache-synchronisation sequence on x's cache-line. If the fetch transition i were to satisfy its fetch in a subsequent state, it would be guaranteed that w' (or a coherence-newer write) would be in the instruction cache, and i would not be able to fetch from w. Hence, i must have happened before the `IC` completing the cache synchronisation sequence.

5. We add a relation *fetch-ordered-before* (`fob`), which is included in *ordered-before*. The relation `fob` includes `fpo`, informally requiring fetches to be ordered according to their order in the control-flow unfolding of the execution. Or correspondingly in the operational model: fetch requests for instructions within the same thread appear to be satisfied in program order.
6. `fob` also includes the `fe` *fetch-to-execute* relation, capturing the idea that an instruction must be fetched before it can execute. In the operational model, a read can only satisfy/a write can only propagate/a barrier can only commit/etc. after its instruction's fetch is satisfied.
7. More interestingly, `fob` includes the edge `[ISB];fe`$^{-1}$`;fpo`, ordering the fetch of any instruction program-order-succeeding an `ISB` instruction after the `ISB` event. In the operational model, a decoded `ISB` instruction prevents any program-order-later instructions from being removed from the fetch queue and decoded, and when an `ISB` is executed, it returns all entries in this thread's fetch queue (so any program-order-later instructions) to "un-fetched state".
8. The rule `[ISB];po;[R]` in `dob` is no longer needed as the combination of rules `[ISB];fe`$^{-1}$`;fpo`, and `[IF];fe` subsume it.
9. For `DSB ISH` instructions the edge `[R|W|F|DC|IC];po;[dsb.ish]` is included: `DSB ISH`s are ordered with all program-order-preceding non-fetch events. Correspondingly, in the operational model a `DSB ISH` instruction can only commit once all preceding loads/stores/barriers/`DC` or `IC` instructions are finished.
10. Symmetrically, all non-`IF` events are ordered after program-order-preceding `dsb.ish` events. In the operational model, instructions can only execute after all preceding `DSB ISH` instructions are finished.
11. `bob` also includes the edge `[dmb.sy];po;[DC]`, ordering `DC` events after program-order-preceding `DMB SY`s. Correspondingly, in the operational model, a `DC` can only be performed when all preceding `DMB SY` are finished.
12. We include the relation *cache-op-ordered-before* (`cob`) in `ob`. This relation contains the edge `[R|W];(po&scl);[DC]`, ordering `DC` events after program-order-preceding same-cache-line read and write events.

Operationally, a DC will be restarted by a program-order-preceding same-cache-line load if it was performed before the load was satisfied, and by a program-order-preceding same-cache-line store if it was performed before the store propagated its write.

13. Moreover, cob contains the edge [DC];(po&scl);[DC], ordering two same-cache-line, same-thread DC events in program-order. In the operational model, a DC can only be performed when program-order-preceding same-cache-line DC instructions have been performed.
14. We define the relation *could-fetch-from* (cff), capturing, for each fetch i, the writes it could have fetched from (including the one it did fetch from), as the set of same-address writes that are not ordered-after i, and which are not overwritten by coherence-newer writes that were followed by a cachesync sequence ordered-before i. Operationally, this captures writes that could have been in the instruction cache of i's thread: writes that did not happen *after* i in the trace, and excluding writes cleared by earlier cache synchronisation sequences.
15. Then finally, the *constrained unpredictable* axiom requires that the candidate execution's CU boolean is 'true' if-and-only-if the predicate cff_bad indicates "a bad execution": an execution corresponding to a trace in which some instruction fetch could have fetched from multiple writes, at least one of which writes an instruction that architecturally is not "concurrently modifiable". This predicate is defined more naturally in first-order logic than herd's cat language: cff_bad cff $= \exists i \in$ IF. $|\{w|(w,i) \in \texttt{cff}\}| > 1 \wedge \exists w.(w,i) \in \texttt{cff} \wedge \neg$concurrently-modifiable (val w).

References

1. Deacon, W.: The armv8 application level memory model. https://github.com/herd/herdtools7/blob/master/herd/libdir/aarch (accessed 2019-07-01) (2016)
2. Pulte, C., Flur, S., Deacon, W., French, J., Sarkar, S., Sewell, P.: Simplifying ARM Concurrency: Multicopy-atomic Axiomatic and Operational Models for ARMv8. In: Proceedings of the 45th ACM SIGPLAN Symposium on Principles of Programming Languages (Jan 2018). https://doi.org/10.1145/3158107