**C11’s release/acquire declarative memory model**

**RA model**: C11 model where all reads are acquire, all writes are release, and all atomic updates are acquire/release.

**Good balance between performance and programmability**:
- Supports intended hardware/compiler optimizations:
  - Elimination of redundant adjacent accesses
  - Store-load reordering: \( \text{WR} \rightarrow \text{RY} \leftrightarrow \text{RY} \rightarrow \text{WR} \) ( unlike SC)
- **DRF theorem**: No data races under SC ensures no weak behaviors
- **Monotonicity**: Adding synchronization does not introduce behaviors ( unlike TSO)
- **Verified compilation schemes** for x86-TSO and Power
- **Program logics**: RSL, GPS, OGRA

**Strong release/acquire**

**Problem**: Some behaviors allowed by RA are never observed.

\[
\begin{align*}
x &= y = 0 \\
x &= 1; & y &= 1; & \text{print } y & \text{ print x} \\
\text{print } y & \text{ print x} & \text{both threads may print 0}
\end{align*}
\]

**Proposed solution**: Rule out \( hb \cup mo \) cycles.
- No additional cost:
  - Compilation schemes are not affected.
  - Same compiler optimizations are sound.
- No better deal for Power:
  - Power model restricted to RA accesses = strong RA

**SC-fences**

**Problem**: SC-fences are overly weak.

\[
\begin{align*}
x &= y = 0 \\
x &= 1; & \text{print } y; & \text{print } y; & \text{fence}(); & \text{fence}(); & y &= 1 \\
\text{print } y & \text{ print x} & \text{print } y & \text{print } y & \text{fence}(); & \text{fence}(); & y &= 1 \\
\text{fence}(); & \text{fence}(); & y &= 1
\end{align*}
\]

**Proposed solution**: Model SC-fences as atomic updates of a distinguished fence location.
- RA semantics enforces all fence events to be ordered by \( hb \).
- Compilation schemes are not affected.
- Adding fences to guarantee SC:
  - Between every two racy accesses
  - Between racy writes and racy reads for client-server programs

**Operational semantics**

- Based on point-to-point communication.
- Each processor has a local memory and an outgoing message buffer.
- Processors non-deterministically choose between performing their own commands and processing incoming messages.
- Messages are processed in the order they were issued.

```
\[
\begin{align*}
m &= x = 0 \\
m &= 42; \\
x &= 1; & \text{skip; print m}
\end{align*}
\]
```

- Processing a message updates the local memory and adds the message to the outgoing buffer.
- Global timestamps are used to ensure coherence properties:
  - Every write obtains a new timestamp, larger than all previous ones.
  - When processing a message, the local memory is updated only if the message’s timestamp is larger than the stored one.

\[
\begin{align*}
x &= 7; & \text{print x} & x &= 8; & \text{print x} \\
\text{If the first thread prints x, the second thread cannot print 7}
\end{align*}
\]