Fast RMW for TSO: Semantics and Implementation

Bhargava Rajaram, Vijay Nagarajan, Susmit Sarkar, Marco Elver

Presented by
Akshay Gopalakrishnan

July 3, 2023

Introduction

- Read-Modify-Write (RMW) semantics is too strong for TSO.
- RMWs for TSO also implies a write buffer flush beforehand, which is very expensive.
- The write buffer flush is often not required while using RMWs for concurrent algorithms.
- Examining the possible compiler mappings from C11 to TSO for Dekker's algorithm exposes the unnecessary strong semantics of RMW.
- Weakening such strong RMW semantics for TSO is beneficial for performance and can retain correctness of concurrent algorithms.

What the paper is about

- Formally specifying weaker semantics for RMW - dividing RMW usage into 3 atomicity constraints.
- Showing which of these variants of RMW work for different implementation of Dekker's Algorithm to maintain mutual exclusion and deadlock freedom.
- Discussing and proposing implementation of such RMWs in hardware that enforces TSO semantics.

Model looking at: Total Store Order (TSO)

Intuitively:

- Every CPU has a FIFO write buffer.
- Writes issued by CPU are committed to respective buffers.
- Reads issued by CPU first checks if the respective buffer has a write to the same memory. If not, it takes the value from main memory.
- Buffers can be flushed randomly to main memory or can be forced by MFENCE instruction or read-modify-writes (RMW).

Dekker's Algorithm: 3 variants using RMW

T0	T1
x=1	y=1
if(y==0)	if(x==0)
// critical	// critical

(a)

T0	T1
W(x)	W(y)
R(y)	R(x)

(b)

T0	T1
W(x)	W(y)
RMW(z_1)	RMW(z_2)
R(y)	R(x)

(c)

T0	T1
W(x)	W(y)
RMW(y)	RMW(x)

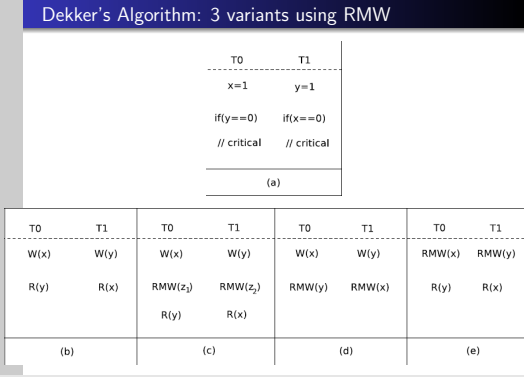
(d)

T0	T1
RMW(x)	RMW(y)
R(y)	R(x)

(e)

Fast RMW for TSO: Semantics and Implementation

Dekker's Algorithm: 3 variants using RMW



One might ask why not use MFENCE when implementing Dekker's algorithm in TSO model. The possible reasons for not doing so are the following:

- RMW itself includes fence semantics in the TSO semantics.
- (c) can definitely be avoided as it involves an RMW which is not required.
- (d) and (e) combines the fence and the memory operation, but also involves an extra memory event.
- It must be that RMW implementation allows the combination of memory operation and fence to be faster than a separate fence followed by a separate memory event.
- Hence, such an implementation makes sense.

Intuition: Original RMW - Type 1 Atomicity

No memory event can happen in between the RMW.

- First flush all pending writes to main memory.
- Then perform the RMW read part.
- Then perform the RMW write part.
- Commit this write to memory.
- We are done.

Fast RMW for TSO: Semantics and Implementation

Intuition: Original RMW - Type 1 Atomicity

Intuition: Original RMW - Type 1 Atomicity

No memory event can happen in between the RMW.

- First flush all pending writes to main memory.
- Then perform the RMW read part.
- Then perform the RMW write part.
- Commit this write to memory.
- We are done.

The default view of read-modify-write events is that no other memory access is supposed to take place or be serviced by the system until this is done. Think of it as taking a global lock and then performing a flush, followed by a read and a write committed to memory.

Intuition: Weakening RMW - Type 2 Atomicity

No conflicting memory events can happen in between the RMW.

- If write-buffer has conflicting events, only then perform a buffer flush until that write is committed.
- Then perform the RMW read part.
- In between any number of non-conflicting memory accesses can occur.
- Then perform the RMW write part.
- Commit this write to memory.

Intuition: Further weakening RMW - Type 3 Atomicity

No conflicting writes can happen in between the RMW.

- Same as Type 2 atomicity.
- Except in between any number of conflicting read accesses can also occur.

Intuition: Further weakening RMW - Type 3 Atomicity

No conflicting writes can happen in between the RMW.

- Same as Type 2 atomicity.
- Except in between any number of conflicting read accesses can also occur.

In truth, one may not require such strong guarantees from RMW, otherwise the point of MFENCE is sort of debatable: why not use RMW everywhere and ditch the MFENCE entirely?? Hence, it is rather better to have a separation and not entirely keep the fence semantics as part of RMW. Thus, these weakenings in what RMW should ensure.

Towards Axiomatic Formulation: Preliminaries

- po - Intra-thread syntactic order.
- ppo - Order preserved by TSO ($po \setminus [W]; po; [R]$)
- rf - Reads-from
- rfe - Reads-from external
- ws - Write serialization (coherence order)
- fr - From-reads (reads-before)
- bar - Barrier (MFENCE)

Axiomatic model without RMW

- $com = rfe \cup ws \cup fr$.
- $ghb = (com \cup ppo \cup bar)*$.

Axioms of TSO (without RMWs)

- ghb acyclic.
- $com \cup po_{loc} \cup bar$ acyclic (uniproc).

Fast RMW for TSO: Semantics and Implementation

Axiomatic model without RMW

Axiomatic model without RMW

- $com = rfe \cup ws \cup fr$.
- $ghb = (com \cup ppo \cup bar)*$.

Axioms of TSO (without RMWs)

- ghb acyclic.
- $com \cup po_{loc} \cup bar$ acyclic (uniproc).

To be honest, the two axioms of TSO can be potentially combined. The TSO model by Viktor and Ori in terms of irreflexivity constraints seem to be much simpler in understanding what the axioms mean. In some sense, the individual relations here do make sense. But the uniproc axiom seems unnecessary; one can instead define ppo to be po that preserves po_{loc} totally. Then the axiom would just be one acyclicity.
Perhaps for the sake of proofs it is not done so.

Adding RMW

- RMW events (R_a, W_a together) introduce additional memory ordering constraints.
- These are represented by *ato* (atomically induced orderings).
- The indicate which memory events cannot take place between the read and write part of RMW.
- Eg: Some memory operation M has to take place after an RMW gives $([W_a]; ato; [M])$.
- The above also implies $[R_a]; ato; [M]$.

Modified axiom of TSO now involves new $ghb = (com \cup ppo \cup bar \cup ato)*$.

Adding RMW
- RMW events (R_a, W_a together) introduce additional memory ordering constraints.
- These are represented by *ato* (atomically induced orderings).
- The indicate which memory events cannot take place between the read and write part of RMW.
- Eg: Some memory operation M has to take place after an RMW gives $([W_a]; ato; [M])$.
- The above also implies $[R_a]; ato; [M]$.

Modified axiom of TSO now involves new $ghb = (com \cup ppo \cup bar \cup ato)*$.

Adding RMW just involves from the axiomatic perspective, a new partial order between already existing events and RMW events. This relation is what will play a role in defining the semantics of RMW. The different variants of RMW semantics revolve around this new partial order called *ato*. What I find a bit disappointing from the paper is that this is not specified properly and hence leads to confusion.
Instead the paper uses previous *ghb* definition to define semantics of each RMW. It then uses these definitions to derive *ato* between the events, specified as lemmas. But the lemmas need to be more specific to *ato* and not the combined memory orderings.

Formal: Original RMW: Type 1

Reads/Writes to any address cannot appear in between the RMW.

$$\forall M.[M]; ghb; [R_a] \vee [W_a]; ghb; [M].$$

Lemma

A type 1 RMW placed between a write W and read R in a thread enforces:

$$[W]; ato; [R_a] \wedge [W_a]; ato; [R] \wedge [W]; ato; [R].$$

Fast RMW for TSO: Semantics and Implementation

Formal: Original RMW: Type 1

Formal: Original RMW: Type 1

Reads/Writes to any address cannot appear in between the RMW.

$\forall M.[M]; ghb; [R_a] \vee [W_a]; ghb; [M].$

Lemma

A type 1 RMW placed between a write W and read R in a thread enforces:

$[W]; ato; [R_a] \wedge [W_a]; ato; [R] \wedge [W]; ato; [R].$

Note that this is a thread-local property. The *ghb* around one of the atomic memory events imply an *ato* on the other. Simply using *ppo* and type 1 RMW is enough to prove.

Visiting Dekker's: Type 1 atomicity

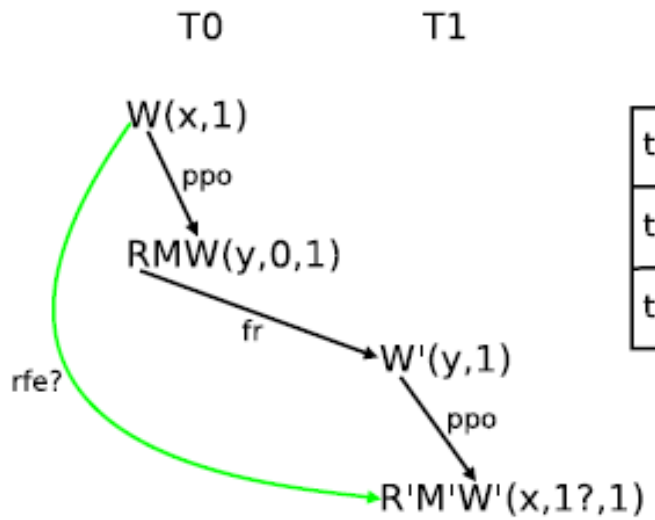


Figure 4. Dekker's with reads replaced by RMWs.

Formal: Weakening RMW: Type 2

Reads/Writes to the same address cannot appear in between the RMW.

$$\forall M(x).[M(x)]; ghb; [R(x)_a] \vee [W(x)_a]; ghb; [M(x)].$$

Lemma

A type 2 RMW placed between a write W and read R in a thread enforces:

$$\neg[R_a]; ghb; [W] \wedge \neg[R]; ghb; [W_a].$$

Fast RMW for TSO: Semantics and Implementation

Formal: Weakening RMW: Type 2

Formal: Weakening RMW: Type 2

Reads/Writes to the same address cannot appear in between the RMW.

$$\forall M(x).[M(x)]; ghb; [R(x)_a] \vee [W(x)_a]; ghb; [M(x)].$$

Lemma

A type 2 RMW placed between a write W and read R in a thread enforces:

$$\neg[R_a]; ghb; [W] \wedge \neg[R]; ghb; [W_a].$$

Note firstly that this is a thread-local property. The intuition is that it even if any read or write occur in between the type 2 RMW, it will NOT be part of the global memory order. We must note that *ghb* is actually related to write buffer flushes, which enforce certain serialization order of writes.

Visiting Dekker's: Type 2 atomicity

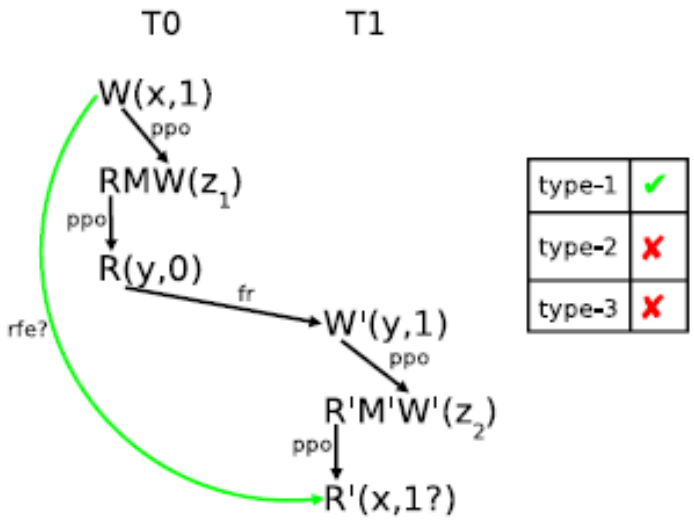


Figure 5. Dekker's with RMWs used as memory barriers. The two RMWs access different addresses z_1 and z_2.

Formal: Further Weakening RMW: Type 3

Writes to the same address cannot appear in between the RMW.

$$\forall W(x).[W(x)]; ghb; [R(x)_a] \vee [W(x)_a]; ghb; [W(x)].$$

Lemma

A type 3 RMW placed between a write W and read R in a thread enforces:

$$\neg[R_a]; ghb; [W].$$

Fast RMW for TSO: Semantics and Implementation

Formal: Further Weakening RMW: Type 3

Formal: Further Weakening RMW: Type 3

Writes to the same address cannot appear in between the RMW.

$\forall W(x).[W(x)]; ghb; [R(x)_a] \vee [W(x)_a]; ghb; [W(x)].$

Lemma

A type 3 RMW placed between a write W and read R in a thread enforces:

$\neg[R_a]; ghb; [W].$

The intuition again is that a write could take place in between, but it will not be part of a global memory order. But the read can potentially be part of it, when it occurs in between them. In between them is more to say that any read outside can also perform its read before the write is actually performed by the RMW.

Visiting Dekker's: Type 3 atomicity

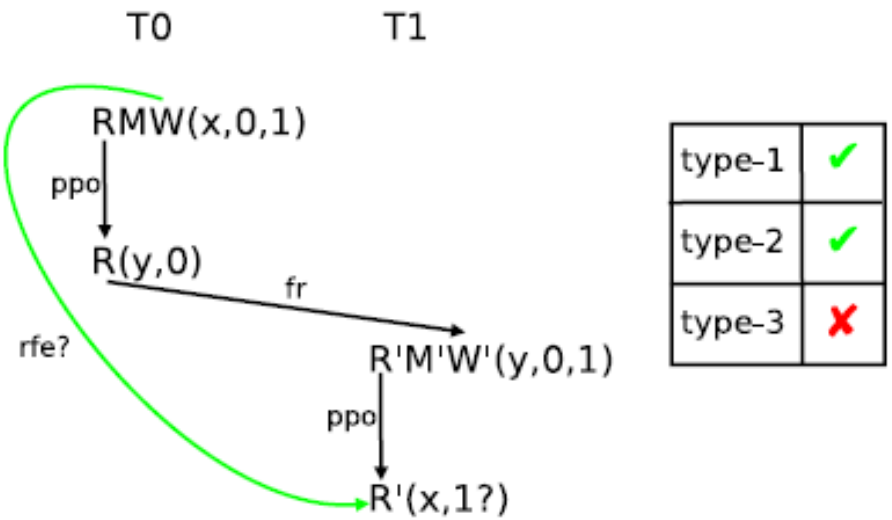


Figure 3. Dekker's with writes replaced by RMWs. In this and other examples that follow, $RMW(x, 0, 1)$ means that the RMW reads a value of 0 from location x and updates it to 1

Implementation: Original RMW: Type 1

Implementation: Weakening RMW: Type 2

Implementation: Further Weakening RMW: Type 3

Recap

Conclusion

Honestly, this paper lacks flow of intuition. The lemmas placed and stated have no purpose stated before to emphasize its need. It is my understanding that these lemmas are actually not required to be honest. Perhaps just Lemma 1 suffices, but even that can be elided as we can simply use the definition of type 1 RMW.

What the authors forget to convey, is that much of the results are actually local. Which means, lets say for type 2 RMW, there are non-conflicting read/writes that could intervene from other threads. This is not much emphasized, and Lemma 2 actually makes this worse to understand, as it is a thread-local property.

The connection of the Lemmas to the implementation is also not so clear, but my intuition says it potentially plays a role. Perhaps the authors could do a better job at that. The results surely are interesting and useful to consider for performance and as an optimization of RMW, but the presentation lacks clarity and intuition.