Making Weak Memory Models Fair

Ori Lahav, Egor Namakanov, Jonas Oberhauser, Anton Podkopaev, Viktor Vafeiadis

Presented by
Akshay Gopalakrishnan

September 3, 2024

Introduction

- Termination guarantees in a concurrent environment relies on fairness.
- In particular, thread-fairness, which ensures that each thread is eventually scheduled to run each instruction till the end.
- While this is sufficient for interleaving semantics (Sequential consistency), thread-fairness is not sufficient for weaker models of concurrency.
- The key problem lies behind the lack of write propagation guarantee from one thread to another under weaker models.
- The additional constraint required is termed as memory-fairness, which in conjunction with thread-fairness can be used to reason about program termination under weak memory models.

Paper's Contributions

- Identifies memory fairness guarantees for Operational and Declarative style models of concurrency.
- In particular, fairness constraints for SC, TSO, RA, $StrongCOH$ were identified.
- Equivalence between both operational and declarative memory fairness constraints were proven.
- Above results were extended to identify memory fairness for $RC11$, showcasing that existing results over compilation and correctness of optimizations remain unchanged.

Example Spinlock

$$x := 1 \parallel \texttt{repeat}\ \{\ a := x\ \}\ \texttt{until}\ (a \neq 0)$$

- The loop will not terminate until the corresponding loop thread in scheduled.
- The loop will not terminate until the corresponding thread modifying the value of x is scheduled.

Thread Fairness: Laymman Terms

- Each thread is scheduled at least once - Enabled.
- Each thread does keep running - Continuously enabled.
- Each thread takes further steps towards completion (forward progress) - Thread Fair.

Spinlock under TSO

$$x := 1 \,\|\, \texttt{repeat}\ \{\ a := x\ \}\ \texttt{until}\ (a \neq 0)$$

- Under thread fairness, both threads will be scheduled, thus ensuring forward progress.
- However, the loop will not terminate until the corresponding thread can observe the modified write value.
- The loop thread will not terminate until the thread modifying the value of x flushes its write buffer to main memory.

New Fairness Requirement

Models of Concurrency must also ensure *Memory Fairness*: That each write is eventually propagated to other threads in any execution.

Memory Fairness: Layman Terms

Memory propagation are named Silent Transitions

- A write memory event is eventually followed by a silent transition to propagate to other threads - Continuously Enabled Silent Transitions.
- Every silent transition step exists in any trace execution, and it is followed by the corresponding write event - Memory Fair.

The concrete details on Silent Transition is dependent on the model of concurrency considered.

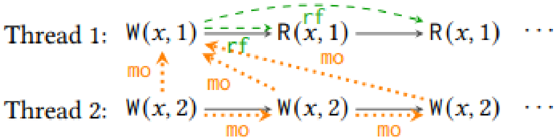
Towards Memory Fairness Constraint: Key Idea

- Each memory location will have a total order *mo* on the writes which modify them in any concurrent execution (Coherence base).
- Each thread must broadcast its write value to other threads, thus ensuring the write is part of the above mentioned total order.
- There cannot be infinite writes - *mo* finite
- A terminating thread, relying on a shared memory write value cannot perpetually read its stale contents.
- Eventually, the reads will read from the latest write in memory - $rf^{-1}; mo$ or fr finite.

Memory Fairness: Example1

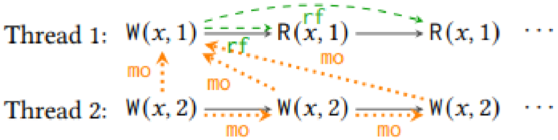
$$
\begin{array}{r|l}
\begin{array}{l} x := 1\,; \\ L_1 : a := x \text{ // only 1} \\ \textbf{goto } L1 \end{array} & \begin{array}{l} L_2 : x := 2\,; \\ \quad \textbf{goto } L_2 \end{array}
\end{array}
$$

- The value of x read by LHS thread cannot always be 1.
- This is due to the fact that the other thread writes $x = 2$ successively.
- How do we specify such a constraint?



- It cannot be possible that all writes $x = 2$ done by the other thread precede the write $x = 1$.
- Having this would imply $x = 1$ perpetually waits for every write done by the other thread to finish.
- A thread cannot wait forever to broadcast its write to other threads.

A write cannot have infinitely many *mo* predecessors to the same location.



- Lets say every write does broadcast eventually and does not perpetually wait.
- Then if there are infinitely many reads, they must eventually read the *mo* maximal write value.
- A thread must eventually be updated with the latest write value that it can read.

Every write cannot have infinitely many *fr* predecessors to the same location.

Memory Fairness: Example 2

$$
\begin{array}{ll|l}
L_1: & x := 1 & L_2: a := x \\
& x := 0 & \quad \texttt{if } a = 0 \texttt{ goto } L_2 \\
& \texttt{goto } L_1 &
\end{array}
$$

- It is possible in this case for an infinite execution to take place.
- LHS thread always finishes broadcasting its write $x = 0$ before RHS thread iteration begins.
- Such an execution is allowed with the current constraints.
- Each write does not have infinitely many predecessors in *mo*.
- And each write is eventually read, keeping *fr* predecessors finite.

Assumption of Bounded number of Threads

$L\colon i := i + 1$

$\quad \textbf{spawn} \left\{ \begin{array}{l} x_{i+1} := 1 \\ a := x_i \ \textit{// only 0} \end{array} \right\}$

$\quad \textbf{goto}\ L$

$$\begin{tikzcd}
\mathtt{W}(x_2,1) \arrow[d] & \mathtt{W}(x_3,1) \arrow[d] \arrow[dl, dotted, "fr"'] & \mathtt{W}(x_4,1) \arrow[d] \arrow[dl, dotted, "fr"'] & \cdots \arrow[dl, dotted, "fr"'] \\
\mathtt{R}(x_1,0) & \mathtt{R}(x_2,0) & \mathtt{R}(x_3,0) & \mathtt{R}(x_4,0)
\end{tikzcd}$$

- The *mo* and *fr* prefix-finiteness is sufficient fairness condition only for bounded number of threads.
- The example execution above is not fair: Every read to $x_{i>1}$ must be 1 instead.
- However, the propagation of writes to the newly spawned thread is not captured by simply pre-fix finiteness of *mo* and *fr*.

Fairness for Weaker Models

- The declarative style memory fairness conditions for TSO, RA, StrongCOH and $RC11$ remain the same.
- To recap the memory-fairness constraint is mo, fr prefix-finiteness.
- The above models have $po \cup rf$ acyclic as a common constraint in the semantic model.
- For models not obeying this constraint, a different fairness constraint is required.

The compilation correctness and optimization safety guarantees for $RC11$ remain unchanged on adding the fairness constraint.

Limitation and Future Work

- Fairness assuming bounded number of threads.
- Fairness constraint identified for models respecting $po \cup rf$ acyclicity constraint.
- Not clear the proposed fairness constraint of prefix-finiteness is for **all models respecting** $po \cup rf$ **acyclicity constraint**.

Further to Read in Paper

- The operational memory-fairness constraints for discussed models.
- Reasoning about termination of different lock implementations using declarative models and the new memory fairness constraint.
- Proof of preserving compilation and optimization safety of $RC11$ on adding memory fairness constraint.
- Preservation of Robustness Guarantees across all models considered in paper.

Thank you

Questions?