Sequential Reasoning for Designing Safe Optimisations under TSO

Akshay Gopalakrishnan, Clark Verbrugge
McGill University

November 12th 2024

Memory Models

Models of shared memory concurrency

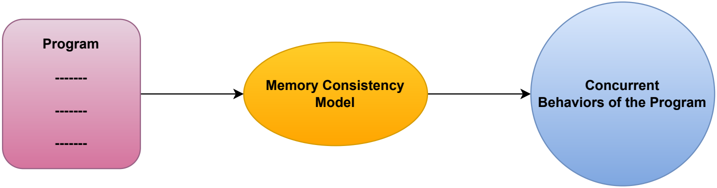


Figure: Memory model as a filter to identify the possible concurrent executions of a program.

Compilers

Respective memory model must be known to design compiler optimizations and code generation phases.

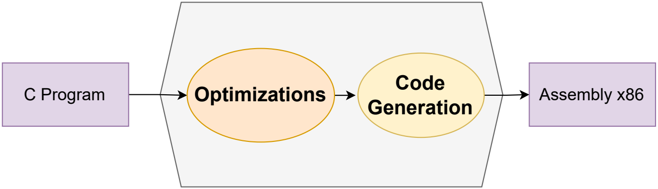
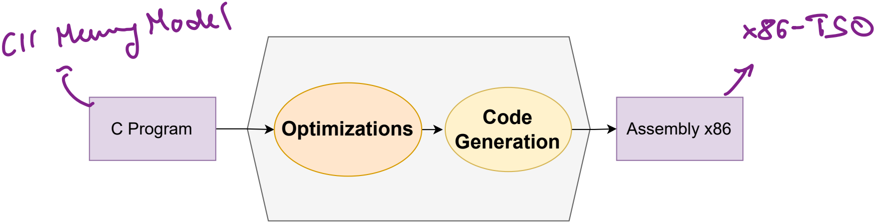


Figure: Programs are optimized (optional) and are then mapped to target hardware language.

Compilers

Respective memory model must be known to design compiler optimizations and code generation phases.



C11 Memory Model

x86-TSO

Figure: Programs are optimized (optional) and are then mapped to target hardware language.

Compilers

Respective memory model must be known to design compiler optimizations and code generation phases.

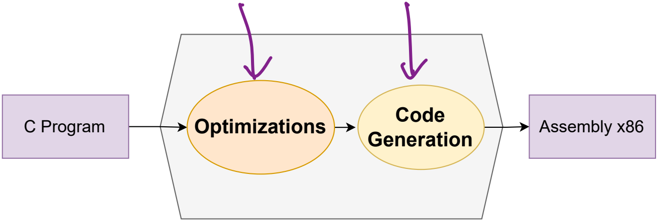


Figure: Programs are optimized (optional) and are then mapped to target hardware language.

Compilers

Respective memory model must be known to design compiler optimizations and code generation phases.

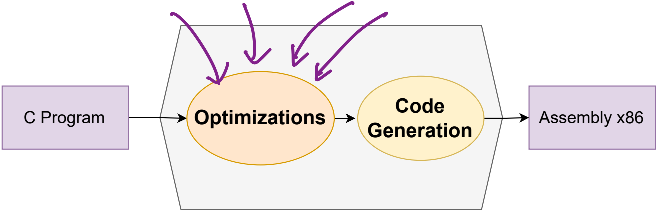


Figure: Programs are optimized (optional) and are then mapped to target hardware language.

Main Concern

- Language concurrency models are non-trivial to understand, let alone use.
- Programmers often have saving grace via guarantees of reliance on interleaving semantics.
- Similar guarantees do not exist for those designing compiler optimizations for the language model.

Main Concern

ex: DRF÷SC

- Language concurrency models are non-trivial to understand, let alone use.
- Programmers often have saving grace via guarantees of reliance on interleaving semantics.
- Similar guarantees do not exist for those designing compiler optimizations for the language model.

Main Concern

- Language concurrency models are non-trivial to understand, let alone use.
- Programmers often have saving grace via guarantees of reliance on interleaving semantics.
- Similar guarantees do not exist for those designing compiler optimizations for the language model.

General Problem

Is there anything analogous to reliance on interleaving semantics for instead, designing optimizations?

An alternative but equivalent question:

When is it implied that a safe optimization for memory model $M1$ is also for $M2$?

General Problem

Is there anything analogous to reliance on interleaving semantics for instead, designing optimizations?

An alternative but equivalent question:

When is it implied that a safe optimization for memory model $M1$ is also for $M2$?

General Problem

Is there anything analogous to reliance on interleaving semantics for instead, designing optimizations?

An alternative but equivalent question:

When is it implied that a safe optimization for memory model *M*1 is also for *M*2?

Today specific M1, M2.

Today's Talk

- $M1$ - Model desired for reasoning - Sequential Consistency (SC).
- $M2$ - Language model - Total Store Order (TSO).

Sequential Consistency and Total Store Order

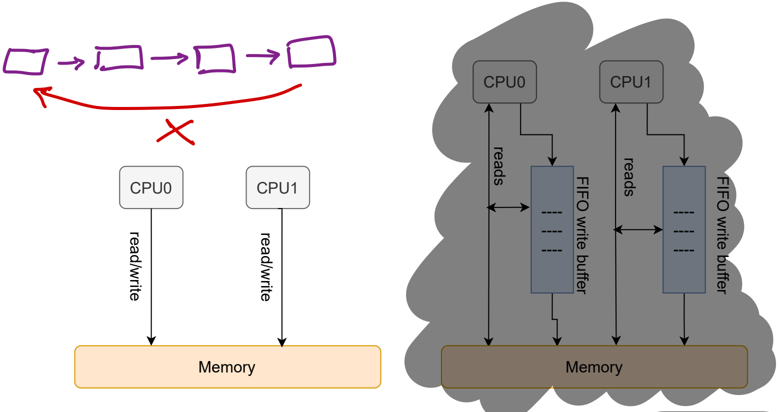


Figure: Abstract Machine for Sequential Consistency (left) and Total Store Order (right).

Sequential Consistency and Total Store Order

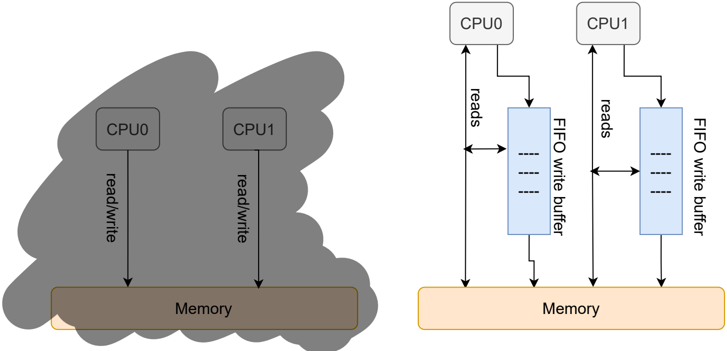


Figure: Abstract Machine for Sequential Consistency (left) and Total Store Order (right).

Example: SB under SC

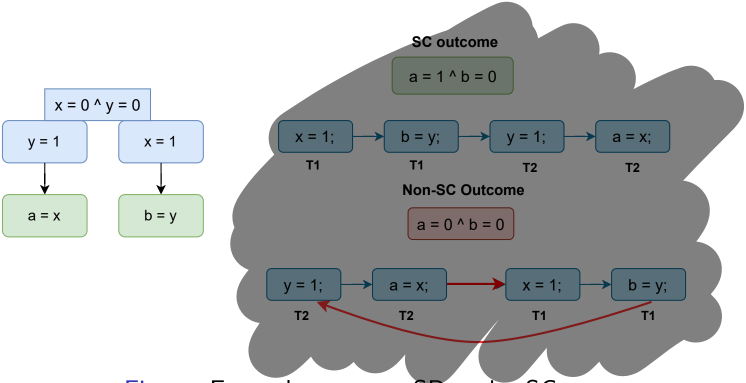


Figure: Example program SB under SC.

Example: SB under SC

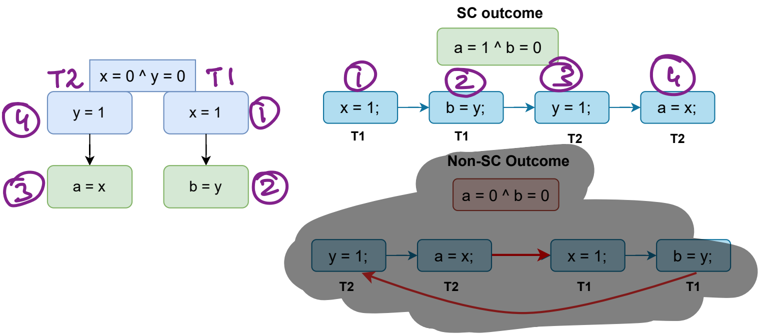


Figure: Example program SB under SC.

Example: SB under SC

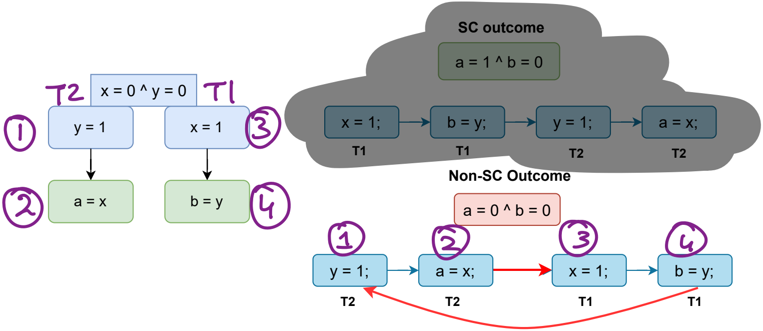


Figure: Example program SB under SC.

Example Revisit: SB under TSO

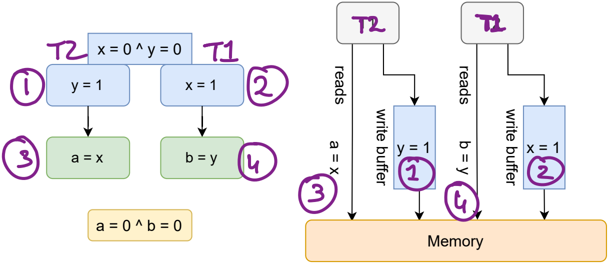


Figure: The non-SC outcome outcome possible under TSO due to store buffering (SB).

Concurrent Behaviors: SC vs TSO

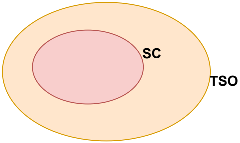


Figure: Set of concurrent behaviors permitted by *SC* and *TSO* given any program.

Concurrent Behaviors: General

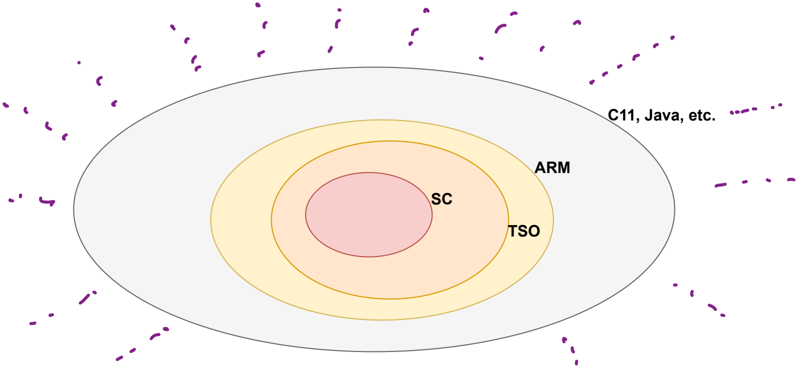


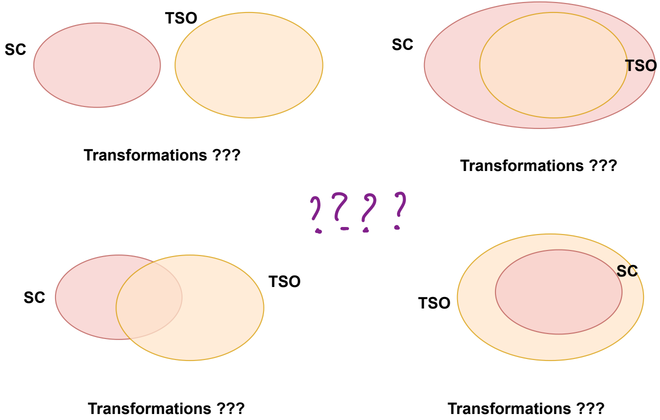
Figure: Set of concurrent behaviors permitted by memory models given any program.

Main Question to Answer

When can I rely on SC to design optimizations for TSO?

When is it implied that a optimization safe for SC is also safe for TSO?

Permitted Optimizations: SC vs TSO



Transformations ???

Transformations ???

Transformations ???

Transformations ???

Figure: Four possibilities in terms of set of optimizations permitted by TSO and SC.

Try Examples

Let us start with some simple optimizations.

Example 1: Independent Write-Read Reordering

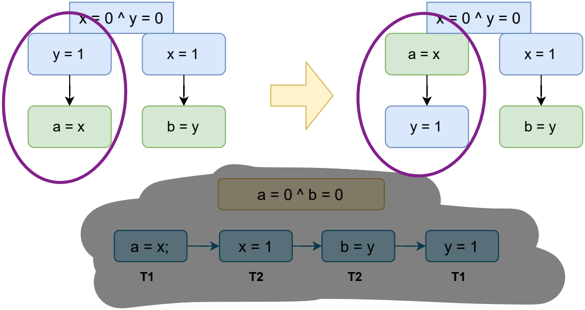


Figure: Compiler can decide to reorder accesses to independent memory.

Example 1: Independent Write-Read Reordering

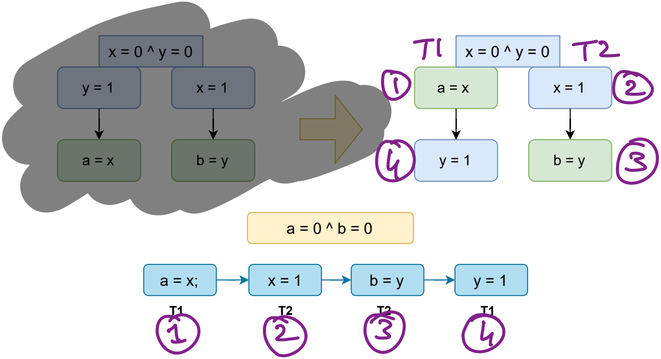


Figure: Compiler can decide to reorder accesses to independent memory.

SC ?? ✗ ✗ ✗

Example 1: Independent Write-Read Reordering

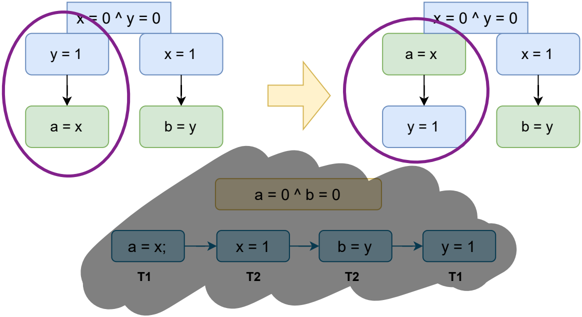


Figure: Compiler can decide to reorder accesses to independent memory.

TSO ?? ✓ ✓ ✓

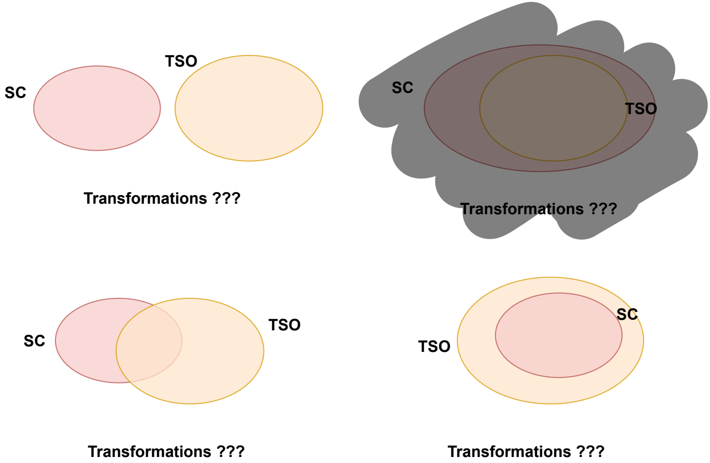


Figure: Write-read reordering removes one of the possibilities.

Example 2: Adjacent Constant Propagation

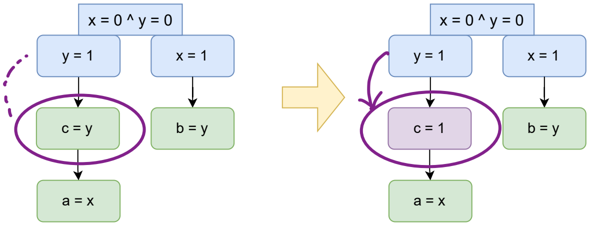


Figure: Compiler can decide to simply propagate the constant 1 to c.

Example 2: Adjacent Constant Propagation

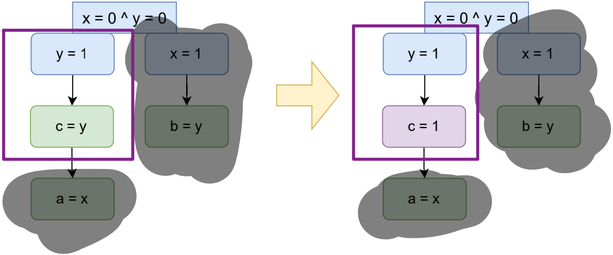


Figure: Compiler can decide to simply propagate the constant 1 to c.

SC ?? ✓ ✓ ✓

Example 2: Adjacent Constant Propagation

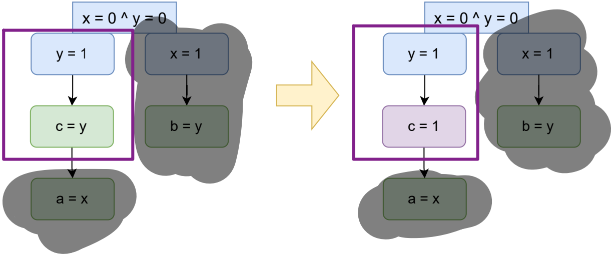


Figure: Compiler can decide to simply propagate the constant 1 to c.

SC ?? ✓ ✓ ✓

TSO ?? ✓ ✓ ✓

Example 3: Redundant Write Elimination

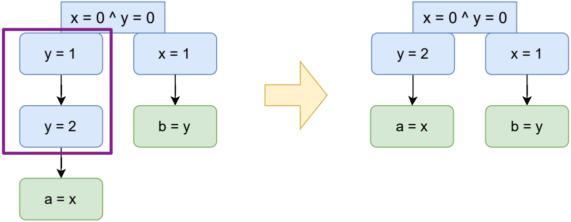


Figure: Compiler can decide to remove the write $x = 1$ as it is immediately overwritten.

SC ? ? ✓ ✓ ✓

TSO ?? ✓ ✓ ✓

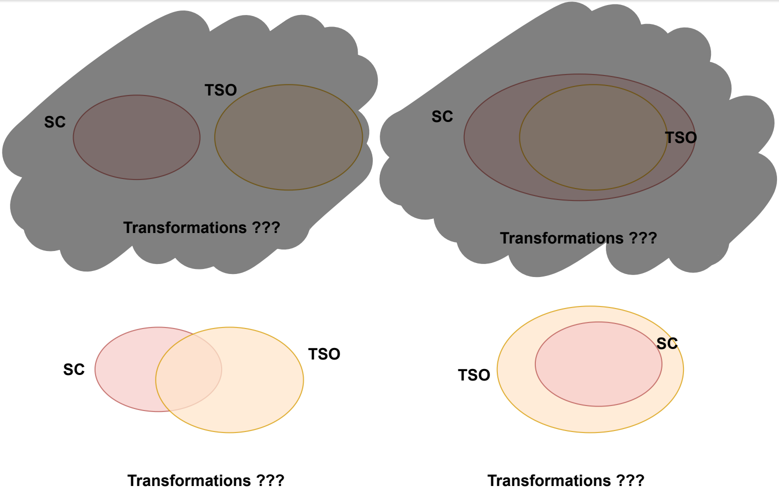


Figure: Write Elimination and Constant Propagation removes another possibility.

Example 4: Reordering CAS

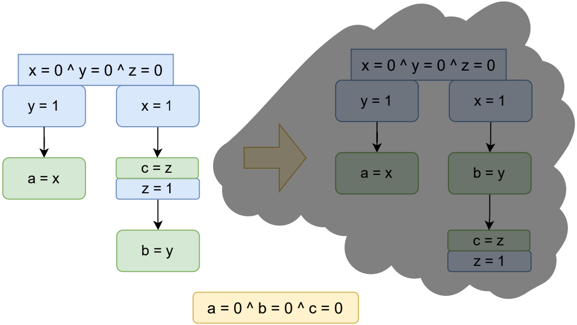


Figure: Reordering CAS okay in this program for SC but not for TSO.

Example 4: Reordering CAS

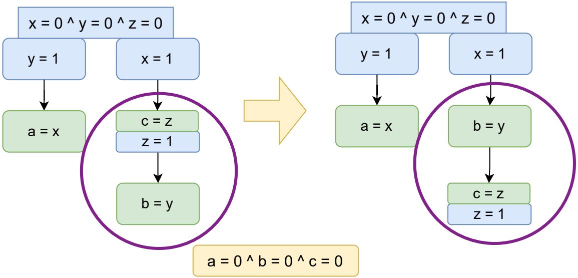


Figure: Reordering CAS okay in this program for SC but not for TSO.

SC? ?

✓

Example 4: Reordering CAS

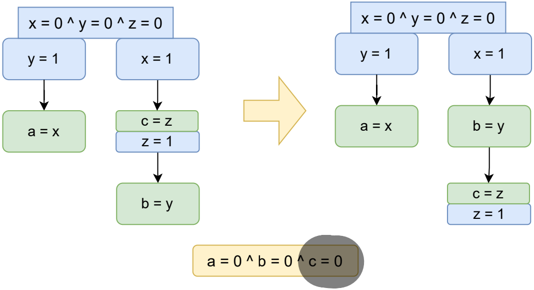


Figure: Reordering CAS okay in this program for SC but not for TSO.

TSO ??? × × ×

Example 5: Eliminating CAS

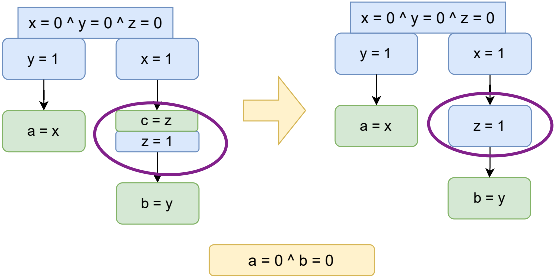


Figure: Compiler can assert the CAS atomic read plays no role, thereby removing the read entirely,

SC ?? ✓ ✓ ✓

TSO ?? ✗ ✗ ✗

Sounds like.... If we do not touch CAS, we should be okay????

Thread Merging

- We can extend the restriction of interleavings to the level of threads.
- This means we can enforce one thread to execute entirely before another.

Since restricting interleavings is safe under SC, thread merging is also a safe optimization.

Thread Merging

- We can extend the restriction of interleavings to the level of threads.
- This means we can enforce one thread to execute entirely before another.

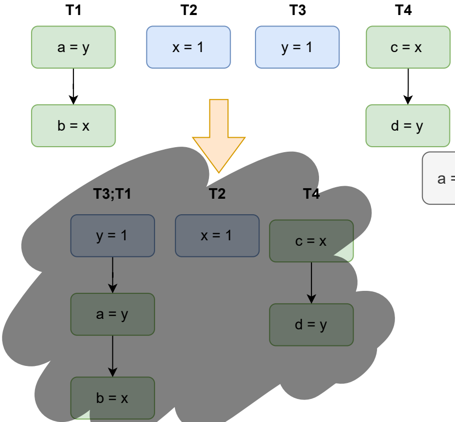
Since restricting interleavings is safe under SC, thread merging is also a safe optimization.

Thread Merging

- We can extend the restriction of interleavings to the level of threads.
- This means we can enforce one thread to execute entirely before another.

Since restricting interleavings is safe under SC, thread merging is also a safe optimization.

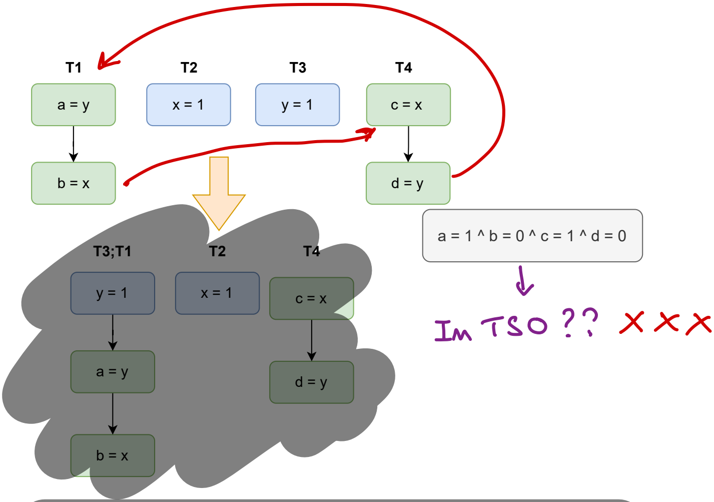
Example 6: Thread Merging



a = 1 ^ b = 0 ^ c = 1 ^ d = 0

Figure: Merging *T*3 and *T*1 is not okay under TSO.

Example 6: Thread Merging



a = 1 ^ b = 0 ^ c = 1 ^ d = 0

In TSO ?? X X X

Figure: Merging $T3$ and $T1$ is not okay under TSO.

Example 6: Thread Merging

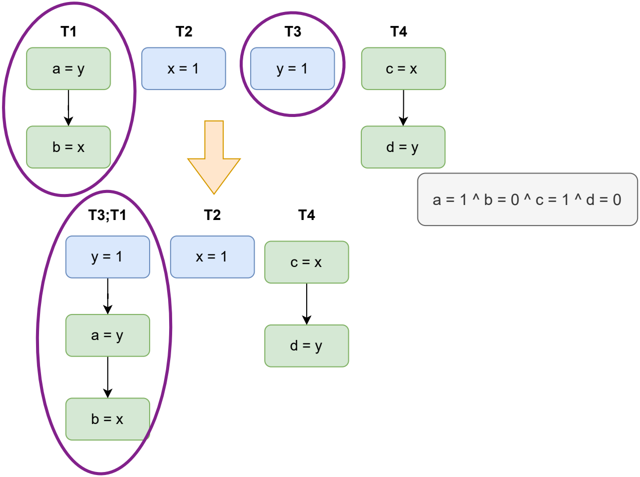


Figure: Merging $T3$ and $T1$ is not okay under TSO.

Example 6: Thread Merging

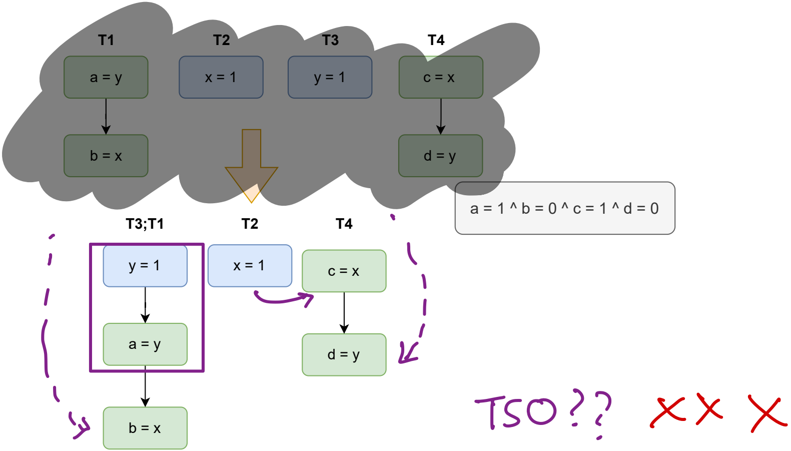


Figure: Merging $T3$ and $T1$ is not okay under TSO.

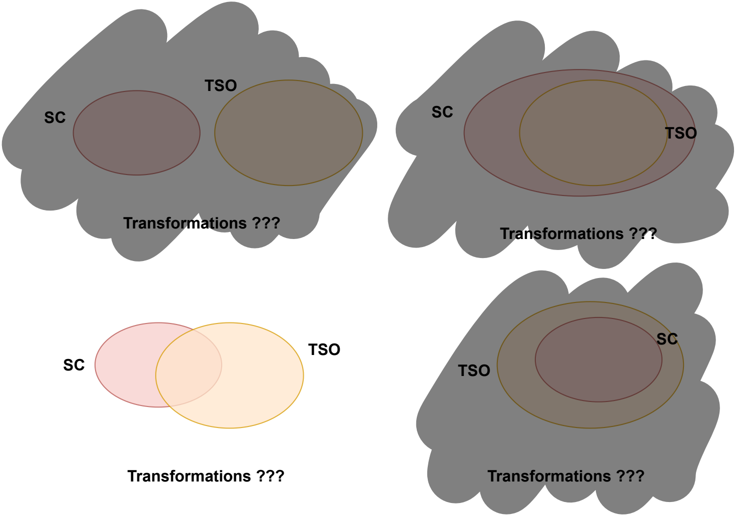


Figure: Optimizing CAS or Thread Merging removes yet another possibility.

Final Verdict

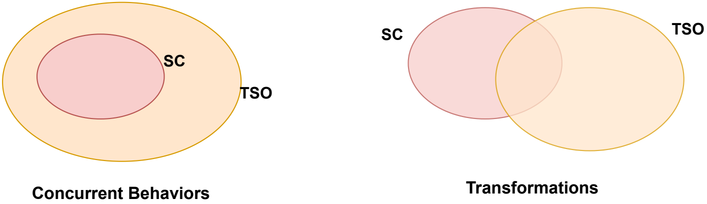


Figure: The final relation between SC and TSO w.r.t. permitted concurrent behaviors (left) and optimizations (right)

Final Verdict

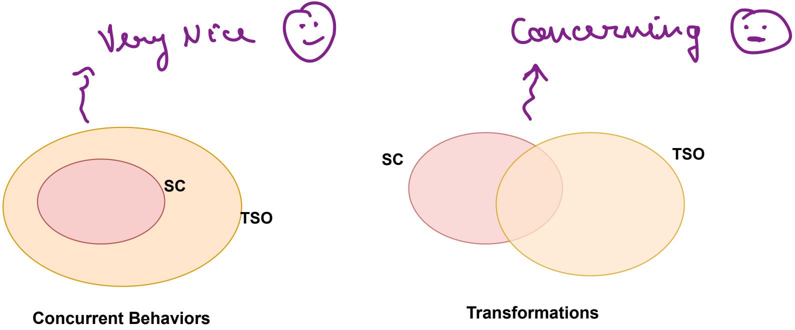


Figure: The final relation between SC and TSO w.r.t. permitted concurrent behaviors (left) and optimizations (right)

Pending Question

What is the exact common set of optimizations given memory models $M1$ and $M2$?

If there exists such a set, how do we quantify it?

Pending Question

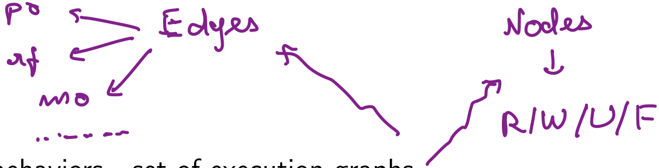
What is the exact common set of optimizations given memory models $M1$ and $M2$?

If there exists such a set, how do we quantify it?

Our Solution: Reason with Execution Traces

- Concurrent behaviors - set of execution graphs.
- *Axiomatic* description memory models - set of constraints on execution graphs.
- Decomposing optimizations into trace-level syntactic *effects* on execution graphs.

Our Solution: Reason with Execution Traces



- Concurrent behaviors - set of execution graphs.
- *Axiomatic* description memory models - set of constraints on execution graphs.
- Decomposing optimizations into trace-level syntactic *effects* on execution graphs.

Our Solution: Reason with Execution Traces

po

rf

mo

......

Edges

Nodes

R/W/U/F

- Concurrent behaviors - set of execution graphs.
- *Axiomatic* description memory models - set of constraints on execution graphs.
- Decomposing optimizations into trace-level syntactic *effects* on execution graphs.

Graph Acyclicity .....

Our Solution: Reason with Execution Traces

Edges → po, rf, mo, ...

Nodes ↓ R/W/U/F

- Concurrent behaviors - set of execution graphs.
- *Axiomatic* description memory models - set of constraints on execution graphs.
- Decomposing optimizations into trace-level syntactic *effects* on execution graphs.

Graph Acyclicity .....

eg: Change egdes

po ⇝ po'

Example: Write-Read Reordering Effect

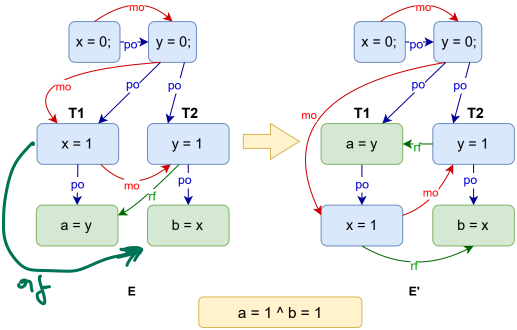


Figure: Memory model as *axioms on execution graphs*, optimizations as *effects on execution graphs*.

Example: Write-Read Reordering Effect

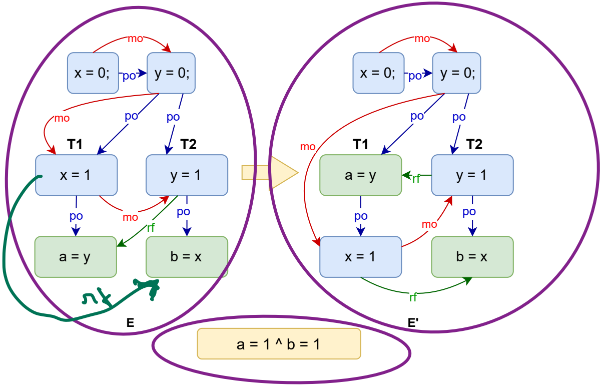


Figure: Memory model as *axioms on execution graphs*, optimizations as *effects on execution graphs*.

Example: Write-Read Reordering Effect

eg:- $(po \cup rf)$ acyclic

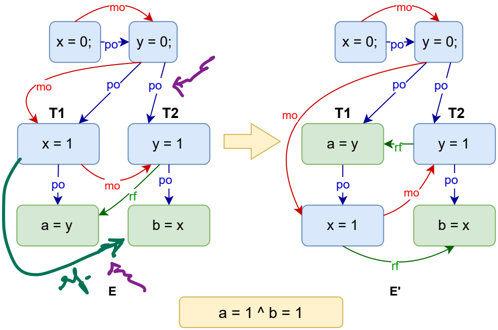
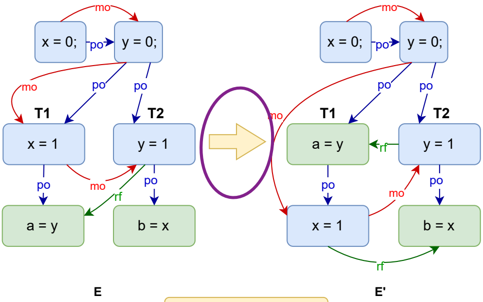


Figure: Memory model as *axioms on execution graphs*, optimizations as *effects on execution graphs*.

Example: Write-Read Reordering Effect



a = 1 ^ b = 1

Figure: Memory model as *axioms on execution graphs*, optimizations as *effects on execution graphs*.

When SC for TSO

Optimizations for TSO can be designed relying on SC which do not involve

- Introducing -
  - Independent write-read syntactic order (eg: $y = 1$ to $b = x$).
  - New writes (violate coherence).

  .
- Eliminating read-modify-write (CAS) events.
- Reordering read-modify-write with later read events in the same thread.

When SC for TSO

Optimizations for TSO can be designed relying on SC which do not involve ✗

$y=1 \parallel b=x \rightsquigarrow$

$$\begin{tikzcd}y=1 \arrow[d] \\ b=x\end{tikzcd}$$

- Introducing -
  - Independent write-read syntactic order (eg: $y = 1$ to $b = x$).
  - New writes (violate coherence).

.

- Eliminating read-modify-write (CAS) events.
- Reordering read-modify-write with later read events in the same thread.

When SC for TSO

Optimizations for TSO can be designed relying on SC which do not involve ✗

- Introducing -
  - Independent write-read syntactic order (eg: $y = 1$ to $b = x$).
  - New writes (violate coherence).

$$\begin{tikzcd} y=1 \,\|\, b=x \arrow[r, squiggly] & y=1 \arrow[d] \\ & b=x \end{tikzcd}$$

.

- Eliminating read-modify-write (CAS) events.

Don't change CAS

- Reordering read-modify-write with later read events in the same thread.

When SC for TSO

Optimizations for TSO can be designed relying on SC which do not involve ✗

- Introducing -
  - Independent write-read syntactic order (eg: $y = 1$ to $b = x$).
  - New writes (violate coherence).

$$\begin{tikzcd} y=1 \parallel b=x \arrow[r, squiggly] & y=1 \arrow[d] \\ & b=x \end{tikzcd}$$

.

- Eliminating read-modify-write (CAS) events.

Don't change CAS

- Reordering read-modify-write with later read events in the same thread.

$$\begin{tikzcd} RMW \arrow[d] & & R \arrow[d] \\ R \arrow[r, phantom, "\rightsquigarrow"] & & RMW \end{tikzcd}$$

✗

Other Results

Designing optimizations:

- Relying on SC for SC-RR (SC + independent read-read reordering).
- Relying on SC for Release Acquire (RA).
- Relying on TSO for Release Acquire (RA).
- Relying on Strong Release Acquire (SRA) for Release Acquire (RA).

Thank you

Questions?

- Akshay Gopalakrishnan *akshay.akshay@mail.mcgill.ca*
- Clark Verbrugge *clump@cs.mcgill.ca*