Overhauling SC atomics in C11 and OpenCL

John Wickerson, Mark Batty, and Alastair F. Donaldson

Imperial Concurrency Workshop
July 2015
TL;DR

- The rules for sequentially-consistent atomic operations and fences ("SC atomics") in C11 and OpenCL are too complex, too weak, and too strong.

- We suggest how to fix them.
TL;DR

• The rules for sequentially-consistent atomic operations and fences ("SC atomics") in C11 and OpenCL are 😞 too complex, 😞 too weak, and 😞 too strong.

We suggest how to fix them 😌.
TL;DR

- The rules for sequentially-consistent atomic operations and fences ("SC atomics") in C11 and OpenCL are 😒 too complex,

    😞 too weak, and
TL;DR

- The rules for sequentially-consistent atomic operations and fences ("SC atomics") in C11 and OpenCL are 😕 too complex, 🎨 too weak, and 😡 too strong.

We suggest how to fix them 😌.
TL;DR

• The rules for sequentially-consistent atomic operations and fences ("SC atomics") in C11 and OpenCL are 😔 too complex,
   😞 too weak, and
   😠 too strong.

• We suggest how to fix them 😌.
• Introduction to the C11 memory model

• Overhauling the rules for SC atomics in C11

• Introduction to the OpenCL memory model

• Overhauling the rules for SC atomics in OpenCL
C11 atomics

• A collection of indivisible operations for lock-free programming, e.g.:

```c
atomic_store_explicit(x, 1, memory_order_relaxed);
```
C11 atomics

• A collection of indivisible operations for lock-free programming, e.g.:

    atomic_store_explicit(x, 1, memory_order_release);
    memory_order_relaxed
C11 atomics

- A collection of indivisible operations for lock-free programming, e.g.:

  ```c
  atomic_store_explicit(x, 1, memory_order_acquire);
  memory_order_release
  memory_order_relaxed
  ```
C11 atomics

• A collection of indivisible operations for lock-free programming, e.g.:

    atomic_store_explicit(x, 1, memory_order_rel_acq);
    memory_order_acquire
    memory_order_release
    memory_order_relaxed
C11 atomics

• A collection of indivisible operations for lock-free programming, e.g.:

    atomic_store_explicit(x, 1, memory_order_seq_cst);
    memory_order_rel_acq
    memory_order_acquire
    memory_order_release
    memory_order_relaxed
C11 atomics

- A collection of indivisible operations for lock-free programming, e.g.:

  `atomic_store_explicit(x, 1, memory_order_seq_cst);
  memory_order_rel_acq
  memory_order_acquire
  memory_order_release
  memory_order_relaxed`

The presence of these other memory orders makes the semantics of SC atomics surprisingly complex.
C11 memory model

- Trace-based semantics ("executions").
C11 memory model

• Trace-based semantics ("executions").

• First phase: generate an overapproximation, by considering each thread in isolation.
C11 memory model

- Trace-based semantics ("executions").

- First phase: generate an overapproximation, by considering each thread in isolation.

- Second phase: remove executions that are inconsistent with the axioms of the memory model.
Example

```c
*x = 42;
atomic_store_explicit(y, 1, memory_order_release);
if (atomic_load_explicit(y, memory_order_acquire))
    print(*x);
```
Example

\[
\begin{align*}
*x &= 42; \\
\text{atomic\_store\_explicit}(y, 1, \\
\text{memory\_order\_release}); &\quad \text{if (atomic\_load\_explicit}(y, \\
\text{memory\_order\_acquire))} \\
\end{align*}
\]
\[
\begin{align*}
\end{align*}
\]
\[
\begin{align*}
\text{print(*x);} \\
\end{align*}
\]
Example

\[ *x = 42; \]
\[
\text{atomic\_store\_explicit}(y, 1, \text{memory\_order\_release});
\]

\[
\text{if (atomic\_load\_explicit}(y, \text{memory\_order\_acquire}))
\]
\[
\text{print}(*x);
\]
Example

```c
*x = 42;
atomic_store_explicit(y, 1, memory_order_release);
```

```c
if (atomic_load_explicit(y, memory_order_acquire))
    print(*x);
```
Example

*x = 42;
atomic_store_explicit(y, 1,
memory_order_release);

if (atomic_load_explicit(y,
memory_order_acquire))
print(*x);
Example

*x = 42;
atomic_store_explicit(y, 1, memory_order_release);

if (atomic_load_explicit(y, memory_order_acquire))
    print(*x);
Example

```c
*x = 42;
atomic_store_explicit(y, 1, memory_order_release);
if (atomic_load_explicit(y, memory_order_acquire))
  print(*x);
```

```
Wna(x, 42)  \downarrow
W(y, 1, REL)
```

```
R(y, 2, ACQ)
Rna(x, 23)
```
Example

```c
*x = 42;
atomic_store_explicit(y, 1, memory_order_release);

if (atomic_load_explicit(y, memory_order_acquire))
    print(*x);
```

Wna(x,42)  
W(y,1,REL)  

R(y,7,ACQ)  
Rna(x,82)
*x = 42;
atomic_store_explicit(y, 1, memory_order_release);

if (atomic_load_explicit(y, memory_order_acquire))
print(*x);
Example
Example

- \( Wna(x,42) \) → \( W(y,1,\text{REL}) \)
- \( R(y,1,\text{ACQ}) \) → \( Rna(x,42) \)
- \( Wna(x,42) \) → \( R(y,5,\text{ACQ}) \)
- \( Wna(x,42) \) → \( R(y,0,\text{ACQ}) \)

✓

- \( Wna(x,42) \) → \( W(y,1,\text{REL}) \)
- \( R(y,1,\text{ACQ}) \) → \( Rna(x,0) \)
- \( Wna(x,42) \) → \( W(y,1,\text{REL}) \)
- \( R(y,1,\text{ACQ}) \) → \( Rna(x,0) \)

✓
Example

1. Wna(x, 42) → W(y, 1, REL) → R(y, 1, ACQ) → Rna(x, 42) – Correct

2. Wna(x, 42) → W(y, 1, REL) → R(y, 0, ACQ) – Incorrect

3. Wna(x, 42) → W(y, 1, REL) → R(y, 5, ACQ) → Rna(x, 2) – Correct

4. Wna(x, 42) → W(y, 1, REL) → R(y, 1, ACQ) → Rna(x, 0) – Incorrect
Example

- Wna(x,42) → R(y,1,ACQ) → W(y,1,REL)
- Wna(x,42) → R(y,1,ACQ) → Rna(x,42)
- Wna(x,42) → R(y,0,ACQ) → W(y,1,REL)
- Wna(x,42) → R(y,1,ACQ) → W(y,1,REL)

Correct paths:
- Wna(x,42) → R(y,1,ACQ) → W(y,1,REL)
- Wna(x,42) → R(y,1,ACQ) → W(y,1,REL)

Incorrect paths:
- Wna(x,42) → R(y,5,ACQ) → W(y,1,REL)
- Wna(x,42) → R(y,1,ACQ) → W(y,1,REL)
Example

- Valid examples:
  - Wna(x,42) → R(y,1,ACQ) → W(y,1,REL)
  - Wna(x,42) → R(y,5,ACQ) → W(y,1,REL)

- Invalid examples:
  - Wna(x,42) → R(y,0,ACQ)
  - Wna(x,42) → R(y,1,ACQ) → Rna(x,0)
Consistent executions
Consistent executions

- Execution $X$ is **consistent** iff it satisfies all the **consistency axioms**.

• Execution $X$ is **consistent** iff it satisfies all the **consistency axioms**.
Consistent executions

• Execution $X$ is consistent iff

  satisfies all the consistency axioms.

• $⟦P⟧ = P$'s consistent executions*
Consistent executions

• Execution $X$ is consistent iff it satisfies all the consistency axioms.

• $⟦P⟧ = P$'s consistent executions*

*unless $P$ also admits a faulty execution, then $⟦P⟧ = $ any execution
Consistent executions

- Execution $X$ is **consistent** iff there exists $rf$, $mo$ and $S$ such that $(X,rf,mo,S)$ is well-formed and satisfies all the **consistency axioms**.

- $⟦P⟧ = P's$ consistent executions*

*unless $P$ also admits a **faulty** execution, then $⟦P⟧ = any$ execution
Candidate executions

\[
a: W_{\text{na}}(x, 0) \quad b: W_{\text{na}}(y, 0) \\
c: W(x, 1, \text{RLX}) \quad d: R(x, 1, \text{RLX}) \quad f: W(x, 2, \text{SC}) \quad h: W(y, 1, \text{SC}) \\
e: R(x, 2, \text{RLX}) \quad g: R(y, 0, \text{SC}) \quad i: R(x, 1, \text{SC})
\]
Candidate executions

\[ a: W_{na}(x, 0) \]
\[ b: W_{na}(y, 0) \]
\[ c: W(x, 1, RLX) \]
\[ d: R(x, 1, RLX) \]
\[ e: R(x, 2, RLX) \]
\[ f: W(x, 2, SC) \]
\[ g: R(y, 0, SC) \]
\[ h: W(y, 1, SC) \]
\[ i: R(x, 1, SC) \]
Some axioms
Some axioms

irreflexive($rf$)
Some axioms

irreflexive($rf$)   irreflexive($mo ; mo ; rf^{-1}$)
Some axioms

irreflexive($rf$)  irreflexive($mo ; mo ; rf^{-1}$)  irreflexive($mo ; rf$)
All consistency axioms

\[
\begin{align*}
\text{irr}(hb) & \quad \text{(Hb)} \\
\text{irr}((rf'^{-1})? ; mo ; rf? ; hb) & \quad \text{(Coh)} \\
\text{irr}(rf ; hb) & \quad \text{(Rf)} \\
\text{empty}((rf ; [nal]) \setminus \text{vis}) & \quad \text{(NaRf)} \\
\text{irr}(rf \cup (mo ; mo ; rf'^{-1}) \cup (mo ; rf)) & \quad \text{(Rmw)} \\
\text{irr}(S ; r_1) & \quad \text{where } r_1 = hb \quad \text{(S1)} \\
\text{irr}(S ; r_2) & \quad \text{where } r_2 = Fsb? ; mo ; sbF? \quad \text{(S2)} \\
\text{irr}(S ; r_3) & \quad \text{where } r_3 = rf^{-1} ; [\text{SC}] ; mo \quad \text{(S3)} \\
\text{irr}((S \setminus (mo ; S)) ; r_4) & \quad \text{where } r_4 = rf^{-1} ; hbl ; [W] \quad \text{(S4)} \\
\text{irr}(S ; r_5) & \quad \text{where } r_5 = Fsb ; rb \quad \text{(S5)} \\
\text{irr}(S ; r_6) & \quad \text{where } r_6 = rb ; sbF \quad \text{(S6)} \\
\text{irr}(S ; r_7) & \quad \text{where } r_7 = Fsb ; rb ; sbF \quad \text{(S7)}
\end{align*}
\]
Derived relations

\[
\begin{align*}
acq & \overset{\text{def}}{=} (\text{ACQ} \cup \text{AR} \cup \text{SC}) \cap (R \cup F) \\
rel & \overset{\text{def}}{=} (\text{REL} \cup \text{AR} \cup \text{SC}) \cap (W \cup F) \\
rb & \overset{\text{def}}{=} (rf^{-1} \ \text{mo}) \setminus \text{id} \\
Fsb & \overset{\text{def}}{=} [F] \ ; \ sb \\
sbF & \overset{\text{def}}{=} sb \ ; [F] \\
rs' & \overset{\text{def}}{=} \text{thd} \cup (E^2 \ ; [R \cap W]) \\
rs & \overset{\text{def}}{=} \text{mo} \cap rs' \setminus ((\text{mo} \ \text{rs'}) \ ; \text{mo}) \\
sw & \overset{\text{def}}{=} ([rel] \ ; Fsb^? \ ; [W \cap A] \ ; rs^? \ ; rf \ ; \\
& \quad [R \cap A] \ ; sbF^? \ ; [acq]) \setminus \text{thd} \\
hb & \overset{\text{def}}{=} (sb \cup (I \times \neg I) \cup sw)^+ \\
hbl & \overset{\text{def}}{=} hb \cap =_{\text{loc}} \\
\text{vis} & \overset{\text{def}}{=} (W \times R) \cap hbl \setminus (hbl \ ; [W] \ ; hb)
\end{align*}
\]
Outline

• Introduction to the C11 memory model

• Overhauling the rules for SC atomics in C11

• Introduction to the OpenCL memory model

• Overhauling the rules for SC atomics in OpenCL
SC axioms

\[
\begin{align*}
\text{irr}(S ; r_1) & \quad \text{where } r_1 = hb & (S1) \\
\text{irr}(S ; r_2) & \quad \text{where } r_2 = Fs^?b ; mo ; sbF^? & (S2) \\
\text{irr}(S ; r_3) & \quad \text{where } r_3 = rf^{-1} ; [SC] ; mo & (S3) \\
\text{irr}((S \setminus (mo ; S)) ; r_4) & \quad \text{where } r_4 = rf^{-1} ; hbl ; [W] & (S4) \\
\text{irr}(S ; r_5) & \quad \text{where } r_5 = Fs^b ; rb & (S5) \\
\text{irr}(S ; r_6) & \quad \text{where } r_6 = rb ; sbF & (S6) \\
\text{irr}(S ; r_7) & \quad \text{where } r_7 = Fs^b ; rb ; sbF & (S7)
\end{align*}
\]
### SC axioms

\[ \text{irr}(S; r_1) \quad \text{where } r_1 = hb \quad (S1) \]
\[ \text{irr}(S; r_2) \quad \text{where } r_2 = Fsb; mo; sbF? \quad (S2) \]
\[ \text{irr}(S; r_3) \quad \text{where } r_3 = rf^{-1}; [SC]; mo \quad (S3) \]
\[ \text{irr}(S; r_4) \quad \text{where } r_4 = rf^{-1}; hbl; [W] \quad (S4) \]
\[ \text{irr}(S; r_5) \quad \text{where } r_5 = Fsb; rb \quad (S5) \]
\[ \text{irr}(S; r_6) \quad \text{where } r_6 = rb; sbF \quad (S6) \]
\[ \text{irr}(S; r_7) \quad \text{where } r_7 = Fsb; rb; sbF \quad (S7) \]
SC axioms

\begin{align*}
\text{irr}(S ; r_1) &= \text{where } r_1 = \text{hb} \\
\text{irr}(S ; r_2) &= \text{where } r_2 = Fsb^? ; mo ; sbF^? \\
\text{irr}(S ; r_3) &= \text{where } r_3 = rf^{-1} ; [SC] ; mo \\
\text{irr}(S ; r_4) &= \text{where } r_4 = rf^{-1} ; \text{hb} \\
\text{irr}(S ; r_5) &= \\
\text{irr}(S ; r_6) &= \\
\text{irr}(S ; r_7) &= 
\end{align*}
Consistent executions

• Execution $X$ is **consistent** iff there exists $rf$, $mo$ and $S$ such that $(X, rf, mo, S)$ is well-formed and satisfies all the **consistency axioms**.

• $\llbracket P \rrbracket = P$'s consistent executions*

*unless $P$ also admits a **faulty** execution, then $\llbracket P \rrbracket = \text{any execution}$
We now show that it is possible to strengthen the SC semantics we establish in the following theorem. This change permits a significant simplification to the SC rules that contributes to the missing constraints so that every edge (rule the intermediate access of the the relations considered in the guarantees to the programmer. simplifies the C11 target architectures that have an established formal memory model simulators, such as. In doing so, we make use of the order extension principle: that any holds:
Theorem 2. The proposal for this simplification arises from the observation

\[
\text{acy}(\mathbf{SC}^2 \cap (r_1 \cup r_2 \cup r_3 \cup r_4 \cup r_5 \cup r_6 \cup r_7) \setminus \text{id}) \quad (S\text{\textsubscript{partial}})
\]
We now show that it is possible to strengthen the SC semantics (strict) partial order can be extended to a (strict) total order.

Theorem 2. Having replaced axioms irrefl, acy, simp, we are to prove that the relations considered in the axioms $\text{irr}(\text{SC})$, $\text{acy}(\text{SC})$, $\text{simp}(\text{SC})$ are nearly symmetrical. In particular, both $\text{axiom}$ and $\text{axiom}$ are nearly symmetrical.

We include in §A.2 a suggestion for how the wording of the axioms of C11 (Power and x86) can be simplified to

$$\text{acy}(\text{SC}^2 \cap (r_1 \cup r_2 \cup r_3 \cup r_4 \cup r_5 \cup r_6 \cup r_7) \setminus \text{id}) \quad (\text{S}_{\text{partial}})$$

This axiom is faster to simulate!
This change permits a significant simplification to the SC rules that are used in the intermediate access of the SC events – a computation that is exponential in the number of SC operations. The strengthening we propose is demonstrated in §6.2.

Proof.

Theorem 2.

Theorem 2.

SC axioms

\[ \text{acy} \left( SC^2 \cap (r_1 \cup r_2 \cup r_3 \cup r_4 \cup r_5 \cup r_6 \cup r_7) \setminus id \right) \] (S_{\text{partial}})

This axiom is faster to simulate!

Existing compilation schemes (x86 and Power) remain valid.
### SC axioms

1. **irr**($S; r_1$) where $r_1 = hb$  
   \[ S1 \]
2. **irr**($S; r_2$) where $r_2 = Fsb?; mo; sbF?$  
   \[ S2 \]
3. **irr**($S; r_3$) where $r_3 = rf^{-1}; [SC]; mo$  
   \[ S3 \]
4. **irr**($S; r_4$) where $r_4 = rf^{-1}; hbl; [W]$  
   \[ S4 \]
5. **irr**($S; r_5$) where $r_5 = Fsb; rb$  
   \[ S5 \]
6. **irr**($S; r_6$) where $r_6 = rb; sbF$  
   \[ S6 \]
7. **irr**($S; r_7$) where $r_7 = Fsb; rb; sbF$  
   \[ S7 \]
SC axioms

\[ \text{irr}(S; r_1) \quad \text{where } r_1 = hb \quad (S1) \]
\[ \text{irr}(S; r_2) \quad \text{where } r_2 = Fs_b^?; mo; sbF^? \quad (S2) \]
\[ \text{irr}(S; r_3) \quad \text{where } r_3 = rf^{-1}; mo \quad (S3) \]
\[ \text{irr}(S; r_4) \quad \text{where } r_4 = rf^{-1}; hbl; [W] \quad (S4) \]
\[ \text{irr}(S; r_5) \quad \text{where } r_5 = Fs_b; rb \quad (S5) \]
\[ \text{irr}(S; r_6) \quad \text{where } r_6 = rb; sbF \quad (S6) \]
\[ \text{irr}(S; r_7) \quad \text{where } r_7 = Fs_b; rb; sbF \quad (S7) \]
SC axioms

\[\text{acy} (SC^2 \cap (Fsb^? \cup (hb \cup rb \cup mo) \cup sbF^?) ) \]  
\hspace{1cm} (S_{simp})
SC axioms

\[ \text{acy}(\text{SC}^2 \sqcap (Fsb^? ; (hb \cup rb \cup mo) ; sbF^?)). \]  \hspace{1cm} (S_{simp})

This axiom is much simpler for programmers to understand and to use.

Existing compilation schemes (x86 and Power) remain valid.
Candidate executions

\[ a: W_{na}(x, 0) \quad b: W_{na}(y, 0) \]
\[ c: W(x, 1, RLX) \quad d: R(x, 1, RLX) \quad f: W(x, 2, SC) \quad h: W(y, 1, SC) \]
\[ e: R(x, 2, RLX) \quad g: R(y, 0, SC) \quad i: R(x, 1, SC) \]
Changing the standard

6. There shall be a single total order $S$ on all memory_order_seq_cst operations, consistent with the “happens before” order and modification orders for all affected locations, such that each memory_order_seq_cst operation $B$ that loads a value from an atomic object $M$ observes one of the following values:

- the result of the last modification $A$ of $M$ that precedes $B$ in $S$, if it exists, or
- if $A$ exists, the result of some modification of $M$ in the visible sequence of side effects with respect to $B$ that is not memory_order_seq_cst and that does not happen before $A$, or
- if $A$ does not exist, the result of some modification of $M$ in the visible sequence of side effects with respect to $B$ that is not memory_order_seq_cst.

[...]

9. For an atomic operation $B$ that reads the value of an atomic object $M$, if there is a memory_order_seq_cst fence $X$ sequenced before $B$, then $B$ observes either the last memory_order_seq_cst modification of $M$ preceding $X$ in the total order $S$ or a later modification of $M$ in its modification order.

10. For atomic operations $A$ and $B$ on an atomic object $M$, where $A$ modifies $M$ and $B$ takes its value, if there is a memory_order_seq_cst fence $X$ such that $A$ is sequenced before $X$ and $B$ follows $X$ in $S$, then $B$ observes either the effects of $A$ or a later modification of $M$ in its modification order.

11. For atomic operations $A$ and $B$ on an atomic object $M$, where $A$ modifies $M$ and $B$ takes its value, if there are memory_order_seq_cst fences $X$ and $Y$ such that $A$ is sequenced before $X$, $Y$ is sequenced before $B$, and $X$ precedes $Y$ in $S$, then $B$ observes either the effects of $A$ or a later modification of $M$ in its modification order.

[276 words; FK reading ease 41.2]

1. A value computation $A$ of an object $M$ reads before a side effect $B$ on $M$ if $A$ and $B$ are different operations and $B$ follows, in the modification order of $M$, the side effect that $A$ observes.

2. If $X$ reads before $Y$, or happens before $Y$, or precedes $Y$ in modification order, then $X$ (as well as any fences sequenced before $X$) is SC-before $Y$ (as well as any fences sequenced after $Y$).

3. If $A$ is SC-before $B$, and $A$ and $B$ are both memory_order_seq_cst, then $A$ is restricted-SC-before $B$.

4. There must be no cycles in restricted-SC-before.

[93 words; FK reading ease 73.1]
Outline

• Introduction to the C11 memory model
• Overhauling the rules for SC atomics in C11
• Introduction to the OpenCL memory model
• Overhauling the rules for SC atomics in OpenCL
OpenCL

• Execution hierarchy:
OpenCL

• Execution hierarchy:
  • Many threads form a work-group
OpenCL

- Execution hierarchy:
  - Many threads form a **work-group**
  - Many work-groups execute on a **device**
OpenCL

- Execution hierarchy:
  - Many threads form a **work-group**
  - Many work-groups execute on a **device**
  - Several devices form a **heterogeneous system**
OpenCL

- Execution hierarchy:
  - Many threads form a **work-group**
  - Many work-groups execute on a **device**
  - Several devices form a **heterogeneous system**
- Memory hierarchy:
OpenCL

• Execution hierarchy:
  • Many threads form a **work-group**
  • Many work-groups execute on a **device**
  • Several devices form a **heterogeneous system**

• Memory hierarchy:
  • **private** (accessible to one thread)
OpenCL

• Execution hierarchy:
  • Many threads form a **work-group**
  • Many work-groups execute on a **device**
  • Several devices form a **heterogeneous system**

• Memory hierarchy:
  • **private** (accessible to one thread)
  • **local** (accessible to one work-group)
OpenCL

- Execution hierarchy:
  - Many threads form a **work-group**
  - Many work-groups execute on a **device**
  - Several devices form a **heterogeneous system**

- Memory hierarchy:
  - **private** (accessible to one thread)
  - **local** (accessible to one work-group)
  - **global** (accessible to all devices)
OpenCL

- Execution hierarchy:
  - Many threads form a **work-group**
  - Many work-groups execute on a **device**
  - Several devices form a **heterogeneous system**

- Memory hierarchy:
  - **private** (accessible to one thread)
  - **local** (accessible to one work-group)
  - **global** (accessible to all devices)
  - **global_fga** (accessible to all devices, allows inter-device communication)
OpenCL memory regions
OpenCL memory regions

T0

T1
OpenCL memory regions

T0
priv

T1
priv
OpenCL memory regions

work-group

T0  T1

priv  priv
OpenCL memory regions

work-group

T0  T1

priv  priv

local
OpenCL memory regions

work-group

T0

T1

priv

priv

local

T2
OpenCL memory regions

device
work-group

T0
priv

T1
priv

T2

local
OpenCL memory regions

- device
- work-group
  - T0
  - T1
  - priv
  - priv
- local
- T2
- T3
OpenCL memory regions

device
work-group
T0 T1
priv priv
local
T2
T3

global
OpenCL memory regions

device
work-group

T0
priv

T1
priv

local

global
global_fga

T2

T3
OpenCL memory regions

device
work-group
T0
priv
local
T1
priv
T2
global
T3
global_fga
OpenCL memory regions

device

work-group

T0  T1  priv  priv

local

global  global_fga

T2

T3
OpenCL memory regions

- T0
- T1
- T2
- T3
- priv
- priv
- local
- global
- global_fga
- device
- work-group
OpenCL memory regions

device
work-group

T0
priv

T1
priv

T2

T3

local

global

global_fga
OpenCL memory regions

- device
  - work-group
    - T0
    - T1
    - priv
    - priv
  - local
    - global
    - global_fga
  - T2
  - T3
OpenCL memory scopes

- Memory consistency can be localised to one subtree of the execution hierarchy.

```c
atomic_store_explicit(x, 1,
    memory_order..., memory_scope_work_group);
```
OpenCL memory scopes

• Memory consistency can be localised to one subtree of the execution hierarchy.

```
atomic_store_explicit(x, 1,
    memory_order..., memory_scope_device )
memory_scope_work_group
```
OpenCL memory scopes

- Memory consistency can be localised to one subtree of the execution hierarchy.

```c
atomic_store_explicit(x, 1,
    memory_order..., memory_scope_all_svm_devices);
memory_scope_device
memory_scope_work_group
```
Example

Threads in the same work-group

```c
*x = 42;
atomic_store_explicit(y, 1,
memory_order_release,
memory_scope_work_group);
if (atomic_load_explicit(y,
memory_order_acquire,
memory_scope_work_group))
print(x);
```
Example

Threads in different work-groups, but same device

*x = 42;
atomc_store_explicit(y, 1,
    memory_order_release,
    memory_scope_work_group);

if (atomic_load_explicit(y,
    memory_order_acquire,
    memory_scope_work_group))
    print(x);
Example

Threads in different work-groups, but same device

```c
*x = 42;
atomic_store_explicit(y, 1,
memory_order_release,
memory_scope_device);

if (atomic_load_explicit(y,
memory_order_acquire,
memory_scope_device))
print(x);
```
Example

Threads in the same work-group

*x = 42;
atomic_store_explicit(y, 1,
memory_order_release,
memory_scope_device);

| if (atomic_load_explicit(y,
memory_order_acquire,
memory_scope_device)) |
| print(x); |
Example

Threads in the same work-group

```c
*x = 42;
atomic_store_explicit(y, 1,
memory_order_release,
memory_scope_device);
if (atomic_load_explicit(y,
memory_order_acquire,
memory_scope_work_group))
  print(x);
```
Scope inclusion

- \((e_1, e_2) \in \text{incl} \iff \text{e}_1'\text{s scope is wide enough to reach } e_2, \text{ and } e_2'\text{s scope is wide enough to reach } e_1.\)
Outline

- Introduction to the C11 memory model
- Overhauling the rules for SC atomics in C11
- Introduction to the OpenCL memory model
- Overhauling the rules for SC atomics in OpenCL
SC axioms in OpenCL
SC axioms in OpenCL

- There is a total order $S$ over [...]
SC axioms in OpenCL

- There is a total order $S$ over [...] 

- **PROVIDING** every SC operation has `memory_scope_all_svm_devices` and accesses `global_fga` memory
SC axioms in OpenCL

- There is a total order \( S \) over [...] 

- **PROVIDING** every SC operation has `memory_scope_all_svm_devices` and accesses `global_fga` memory 

- **OR** every SC operation has `memory_scope_device` and does not access `global_fga` memory
Problems
Problems

😢 Can't always tell whether a location is `global` or `global_fga`!
Problems

😢 Can't always tell whether a location is global or global_fga!

😢 The default, which is memory_scope_device, is not always enough!
Problems

😢 Can't always tell whether a location is `global` or `global_fga`!

😢 The default, which is `memory_scope_device`, is not always enough!

😢 Non-compositional!
Problems

😢 Can't always tell whether a location is global or global_fga!

😢 The default, which is memory_scope_device, is not always enough!

😢 Non-compositional!

😢 Unnecessarily restrictive!
Problems

😢 Can't always tell whether a location is global or global_fga!

😢 The default, which is memory_scope_device, is not always enough!

😢 Non-compositional!

😢 Unnecessarily restrictive!

😢 And too weak anyway!
SC axioms in OpenCL

\[ \text{acy}(\text{SC}^2 \cap (Fsb^? \ ; (ghb \cup lhb \cup rb \cup mo) \ ; sbF^?) \cap \text{incl}) \cap (O-S_{\text{scoped}}) \]
SC axioms in OpenCL

\[ \text{acy}(SC^2 \cap (F_{sb}^? ; (ghb \cup lhb \cup rb \cup mo) ; sbF^?) \cap \text{incl}) \cap (O-S_{scoped}) \]

Existing compilation scheme (for AMD GPUs) remains valid.
Changing the standard

If one of the following two conditions holds:

- All `memory_order_seq_cst` operations have the scope `memory_scope_all_svm_devices` and all affected memory locations are contained in system allocations or fine grain SVM buffers with atomics support
- All `memory_order_seq_cst` operations have the scope `memory_scope_device` and all affected memory locations are not located in system allocated regions or fine-grain SVM buffers with atomics support

then there shall exist a single total order `S` for all `memory_order_seq_cst` operations that is consistent with the modification orders for all affected locations, as well as the appropriate global-happens-before and local-happens-before orders for those locations, such that each `memory_order_seq_cst` operation `B` that loads a value from an atomic object `M` in global or local memory observes one of the following values:

- the result of the last modification `A` of `M` that precedes `B` in `S`, if it exists, or
- if `A` exists, the result of some modification of `M` in the visible sequence of side effects with respect to `B` that is not `memory_order_seq_cst` and that does not happen before `A`, or
- if `A` does not exist, the result of some modification of `M` in the visible sequence of side effects with respect to `B` that is not `memory_order_seq_cst`.

If the total order `S` exists, the following rules hold:

- For an atomic operation `B` that reads the value of an atomic object `M`, if there is a `memory_order_seq_cst` fence `X` sequenced-before `B`, then `B` observes either the last `memory_order_seq_cst` modification of `M` preceding `X` in the total order `S` or a later modification of `M` in its modification order.
- For atomic operations `A` and `B` on an atomic object `M`, where `A` modifies `M` and `B` takes its value, if there is a `memory_order_seq_cst` fence `X` such that `A` is sequenced-before `X` and `B` follows `X` in `S`, then `B` observes either the effects of `A` or a later modification of `M` in its modification order.
- For atomic operations `A` and `B` on an atomic object `M`, where `A` modifies `M` and `B` takes its value, if there are `memory_order_seq_cst` fences `X` and `Y` such that `A` is sequenced-before `X`, `Y` is sequenced-before `B`, and `X` precedes `Y` in `S`, then `B` observes either the effects of `A` or a later modification of `M` in its modification order.
- For atomic operations `A` and `B` on an atomic object `M`, if there are `memory_order_seq_cst` fences `X` and `Y` such that `A` is sequenced-before `X`, `Y` is sequenced-before `B`, and `X` precedes `Y` in `S`, then `B` occurs later than `A` in the modification order of `M`.

1. A value computation `A` of an object `M` reads before a side effect `B` on `M` if `A` and `B` are different operations and `B` follows, in the modification order of `M`, the side effect that `A` observes.
2. If `X` reads before `Y`, or global happens before `Y`, or local happens before `Y`, or precedes `Y` in modification order, then `X` (as well as any fences sequenced before `X`) is SC-before `Y` (as well as any fences sequenced after `Y`).
3. If `A` is SC-before `B`, and `A` and `B` have inclusive scopes, then `A` is restricted-SC-before `B`.
4. There must be no cycles in restricted-SC-before.

[106 words; FK reading ease 71.0]
TL;DR

• The rules for sequentially-consistent atomic operations and fences ("SC atomics") in C11 and OpenCL are 😕 too complex, 😞 too weak, and 😠 too strong.

• We suggest how to fix them 😌.