Introduction to the C/C++11 concurrency model

Graduate seminar on weakly consistent concurrency

Ori Lahav

March 14, 2018
The C11 memory model

- Introduced by the ISO C/C++ 2011 standards.
- Defines the semantics of concurrent memory accesses.
- Platform independent.
- Supported by popular compilers (GCC, LLVM, ...).

We will introduce a simplified version of (a fragment of) the C11 model as a prototypical example of a real world weak concurrency semantics.
Atomics

Two types of accesses

**Ordinary (Non-Atomic)**

Races are **errors**

**Atomic**

Welcome to the **expert mode**
Atomics

Two types of accesses

Ordinary (Non-Atomic)

Races are errors

Atomic

Welcome to the expert mode

DRF (data race freedom) guarantee

no data races only
under SC SC behaviors
A spectrum of access modes

memory_order_seq_cst
  (sc)
  full memory fence

memory_order_release
  write (rel)
  no fence (x86); lwsync (PPC)

memory_order_acquire
  read (acq)
  no fence (x86); isync (PPC)

memory_order_relaxed
  (rlx)
  no fence

| Non-atomic (na)
| no fence, races are errors

+ Explicit primitives for fences
Declarative semantics abstracts away from implementation details.

1. a program $\rightsquigarrow$ a set of directed graphs (called: execution graphs)
2. The memory model defines what executions are consistent.
3. The semantics of a program is the set of its consistent executions.

Exception: “catch-fire” semantics

- Existence of at least one “bad” consistent execution (with a forbidden data races) implies undefined behavior.
Execution graphs

Store buffering (SB)

\[ x = y = 0 \]
\[ x :=_{rlx} 1 \quad || \quad y :=_{rlx} 1 \]
\[ a := y_{rlx} \quad || \quad b := x_{rlx} \]

Relations

- Program order, \( po \)
- Reads-from, \( rf \) (we require that every read reads from some write).
### C/C++11 formal model

\[
\begin{align*}
\text{fig. 2.} & \quad \text{Semantics of closed program expressions.} \\
& \quad \forall \ell, x. \; \text{totalorder} \{ \{ a \in A \mid \text{iswrite}_\ell(a) \}, \text{mo} \} \cup h_\ell \subseteq \text{mo} \\
& \quad \forall \ell, x. \; \text{totalorder} \{ \{ a \in A \mid \text{isread}_\ell(a) \}, \text{sc} \} \cup h_{\text{seqCat}} \subseteq \text{sc} \cup \text{mo}_{\text{seqCat}} \subseteq \text{sc} \\
& \quad \forall b. \; \text{rf}(b) \neq \bot \iff \exists \ell. a. \; \text{iswrite}_\ell(a) \land \text{isread}_\ell(b) \land \text{hb}(a, b) \\
& \quad \forall a, b. \; \text{rf}(b) = a \iff \exists \ell. v. \; \text{iswrite}_\ell(v) \land \text{isread}_\ell(b) \land \neg \text{hb}(a, b) \\
& \quad \forall a, b. \; \text{rf}(b) = a \land (\text{mode}(a) = na \lor \text{mode}(b) = nb) \Rightarrow \text{hb}(a, b) \\
& \forall a, b. \; \text{rf}(b) = a \land \text{isSeqCst}(b) \Rightarrow \text{isc}(a, b) \lor \neg \text{isSeqCst}(a) \land (\forall \ell. \text{isc}(x, b) \Rightarrow \neg \text{hb}(a, x)) \\
& \quad \exists a, b. \; \text{rb}(a, b) \land \text{mo}(rf(b), a) \land \text{locs}(a) = \text{loc}(b) \\
& \quad \exists a, b. \; \text{rb}(a, b) \land \text{mo}(rf(b), a) \land \text{locs}(a) = \text{loc}(b) \\
& \quad \exists a, b. \; \text{rb}(a, b) \land \text{mo}(rf(b), a) \land \text{locs}(a) = \text{loc}(b) \\
& \quad \exists a, b. \; \text{rf}(a) \neq \bot \Rightarrow \text{mo}(rf(a), a) \land \exists c. \; \text{mo}(rf(c), a) \land \text{mo}(c, a) \\
& \quad \forall a, b, \ell. \; \text{lab}(a) = \text{lab}(b) = A(\ell) \Rightarrow a = b \\
\end{align*}
\]

where \( \text{iswrite}_\ell(v) \) is defined as \( \exists X, v_\text{old}. \; \text{lab}(a) \in \{ W_X(\ell, v), \text{RMW}_X(\ell, v_\text{old}, v) \} \), and \( \text{iswrite}(a) \) is defined as \( \exists v. \; \text{iswrite}_\ell(v)(a) \), etc.

\[\begin{align*}
\text{rsElem}(a, b) & \iff \{ a \} \cup \{ b \} \cup \text{mo}(a, b) \land (\forall c. \; \text{mo}(a, c) \land \text{mo}(c, b) \Rightarrow \text{rsElem}(a, c)) \\
\text{sw} & \iff \{ (a, b) \mid \text{mode}(a) \in \{ \text{acq}, \text{rel}, \text{acq}, \text{sc} \} \land \text{mode}(b) \in \{ \text{acq}, \text{rel}, \text{acq}, \text{sc} \} \land \text{rf}(b) \in \text{rsseq}(a) \} \\
\text{hb} & \iff (\text{sb} \cup \text{sw})^+ \\
\text{hb}_r & \iff \{ (a, b) \in \text{hb} \mid \text{iswrite}_\ell(a) \land \text{iswrite}_\ell(b) \} \\
\text{X}_{\text{seqCst}} & \iff \{ (a, b) \in X \mid \text{isSeqCst}(a) \land \text{isSeqCst}(b) \} \\
\text{isc}(a, b) & \iff \text{iswrite}_{\text{loc}}(a)(a) \land \text{sc}(a, b) \land \exists c. \; \text{sc}(c, c) \land \text{sc}(c, b) \land \text{iswrite}_{\text{loc}}(b)(c) \\
\end{align*}\]

### Figure 3. Axioms satisfied by consistent C11 executions, consistent \( A, \text{lab}, \text{sb}, \text{rf}, \text{mo}, \text{sc} \).

\[\begin{align*}
c & : W(\ell, 1) \xrightarrow{\text{rf}} a : R(\ell, 1) \\
d & : W(\ell, 2) \xrightarrow{\text{rf}} b : R(\ell, 2) \\
\end{align*}\]

violates CoherentRR.

\[\begin{align*}
c & : W(\ell, 2) \xrightarrow{\text{rf}} a : W(\ell, 1) \\
d & : W(\ell, 2) \xrightarrow{\text{rf}} b : W(\ell, 2) \\
\end{align*}\]

violates CoherentRW.

\[\begin{align*}
c & : W(\ell, 1) \xrightarrow{\text{rf}} a : R(\ell, 1) \\
d & : W(\ell, 2) \xrightarrow{\text{rf}} b : W(\ell, 2) \\
\end{align*}\]

violates CoherentRR.

\[\begin{align*}
c & : W(\ell, 1) \xrightarrow{\text{rf}} a : R(\ell, 1) \\
d & : W(\ell, 2) \xrightarrow{\text{rf}} b : W(\ell, 2) \\
\end{align*}\]

violates CoherentRW.

### Figure 4. Sample executions violating coherency conditions (Batty et al. 2011).
Sequential consistency

Before C/C++11, how would you define SC as a declarative memory model?

![Diagram](https://via.placeholder.com/150)

Memory

- **CPU 1**
- **CPU n**

READ

WRITE
Sequential consistency

Before C/C++11, how would you define SC as a declarative memory model?

![Diagram of concurrent CPUs accessing memory](image)

**Definition (Lamport ’79)**

$G$ is **SC-consistent** if there exists a relation $sc$ such that:

- $sc$ is a total order on the events of $G$.
- $po \cup rf \subseteq sc$.
- If $\langle a, b \rangle \in rf$ then there does not exist $c \in W_{loc}(a)$ such that $\langle a, c \rangle \in sc$ and $\langle c, b \rangle \in sc$. 
Sequential consistency

Before C/C++11, how would you define SC as a declarative memory model?

![Diagram of CPUs and memory](image)

**Definition (Lamport ’79)**

$G$ is **SC-consistent** if there exists a relation $\text{sc}$ such that:

- $\text{sc}$ is a total order on the events of $G$.
- $\text{po} \cup \text{rf} \subseteq \text{sc}$.
- If $\langle a, b \rangle \in \text{rf}$ then there does not exist $c \in W_{\text{loc}}(a)$ such that $\langle a, c \rangle \in \text{sc}$ and $\langle c, b \rangle \in \text{sc}$.

- We will later see alternative more “economical” definitions that do not require a total order on all events of $G$. 
Ingredients of execution graph consistency in C/C++11

1. SC-per-location (a.k.a. coherence)
2. Release/acquire synchronization
3. Global conditions on SC accesses
Ingredients of execution graph consistency in C/C++11

1. SC-per-location (a.k.a. coherence)
2. Release/acquire synchronization
3. Global conditions on SC accesses
SC-per-location

**Definition (Declarative definition of SC)**

$G$ is **$SC$-consistent** if there exists a relation $sc$ such that:

- $sc$ is a total order on the events of $G$.
- $po \cup rf \subseteq sc$.
- If $\langle a, b \rangle \in rf$ then there does not exist $c \in W_{loc(a)}$ such that $\langle a, c \rangle \in sc$ and $\langle c, b \rangle \in sc$.

**Definition (SC-per-location)**

$G$ satisfies **$SC$-per-location** if for every location $x$, there exists a relation $sc_x$ such that:

- $sc_x$ is a total order on the events of $G$ that access $x$.
- $(po \cup rf) \cap \{\langle a, b \rangle \mid loc(a) = loc(b) = x\} \subseteq sc_x$.
- If $\langle a, b \rangle \in rf$ then there does not exist $c \in W_x$ such that $\langle a, c \rangle \in sc_x$ and $\langle c, b \rangle \in sc_x$. 
SC-per-location: Example

\[
\begin{align*}
x &= 0 \\
x &\equiv_{rlx} 1 \ | \ x \equiv_{rlx} 2 \\
a &\equiv x_{rlx} \ | \ b \equiv x_{rlx}
\end{align*}
\]

Inconsistent!
SC-per-location is often too weak:

- It does not support the message passing idiom:
Implementing locks?

Can we implement a lock under SC-per-location?
Implementing locks?

Can we implement a lock under SC-per-location?

Spinlock implementation

\[
\text{lock}(l) : \quad r := 0;
\text{while } \neg r \text{ do}
\quad r := \text{CAS}(l, 0, 1)
\]

\[
\text{unlock}(l) : \quad l := 0
\]

Under SC-per-location, the spinlock implementation does not guarantee mutual exclusion.
Implementing locks?

Can we implement a lock under SC-per-location?

Spinlock implementation

\[
\text{lock}(l) : \\
\quad r := 0;
\]
\[
\text{while } \neg r \text{ do} \\
\quad r := \text{CAS}(l, 0, 1)
\]
\[
\text{unlock}(l) : \\
\quad l := 0
\]

Lock example

\[
\text{lock}(l) : \\
\quad x := 1
\]
\[
\text{unlock}(l) : \\
\quad \text{unlock}(l)
\]
\[
\text{lock}(l) : \\
\quad y := 1
\]
\[
\text{unlock}(l) : \\
\quad b := x \text{ // 0}
\]

Under SC-per-location, the spinlock implementation does not guarantee mutual exclusion.
Synchronization in C/C++11 through examples

```c
int y = 0;
int x = 0;
y = 42;
if(x == 1){
    x = 1;
    print(y);
}
```
Synchronization in C/C++11 through examples

```c
int y = 0;
int x = 0;
y = 42;
if(x == 1){
    x = 1;
    print(y);
}
```

```c
int y = 0;
atomic<int> x = 0;
y = 42;
if(x == 1){
    x = rlx;
    print(y);
}
```

```c
int y = 0;
atomic<int> x = 0;
y = 42;
if(x == 1){
    x = rel;
    print(y);
}
```

```c
int y = 0;
atomic<int> x = 0;
y = 42;
if(x == 1){
    fence rel;
    fence acq;
    x = rlx;
    print(y);
}
```
Synchronization in C/C++11 through examples

1

```c
int y = 0;
int x = 0;
y = 42;  if(x == 1){
x = 1,  race
  print(y);
}
```

race
Synchronization in C/C++11 through examples

1

```
int y = 0;
int x = 0;
y = 42;  // if(x == 1){
x = 1;    //  print(y);
}        // }
```

2

```
int y = 0;
atomic<int> x = 0;
y = 42;  // if(x_{rlx} == 1){
x =_{rlx} 1; //  print(y);
}        // }
```

race
Synchronization in C/C++11 through examples

1
int y = 0;
int x = 0;
y = 42; || if(x == 1){
x = 1; || race
} print(y);

2
int y = 0;
atomic<int> x = 0;
y = 42; || if(x_rlx == 1){
x = _rlx 1; || race
} print(y);
Synchronization in C/C++11 through examples

1. int y = 0;
   int x = 0;
y = 42;  // if(x == 1){
x = 1,  // race  print(y);
   }

2. int y = 0;
   atomic<int> x = 0;
y = 42;  // if(x_{rlx} == 1){
x =_{rlx} 1;  // race  print(y);
   }

3. int y = 0;
   atomic<int> x = 0;
y = 42;  // if(x_{acq} == 1){
x =_{rel} 1;  // race  print(y);
   }

Synchronization in C/C++11 through examples

1. ```c
   int y = 0;
   int x = 0;
   y = 42;  // if(x == 1) {
     x = 1,  // race
     print(y);
   }  // race
```  

2. ```c
   int y = 0;
   atomic<int> x = 0;
   y = 42;  // if(x_rlx == 1) {
     x = rlx 1;  // race
     print(y);
   }  // race
```  

3. ```c
   int y = 0;
   atomic<int> x = 0;
   y = 42;  // if(x_acq == 1) {
     x = rel 1;  // rf
     print(y);
   }  // rf
```
Synchronization in C/C++11 through examples

1. ```
   int y = 0;
   int x = 0;
   y = 42; // if(x == 1){
   x = 1, // race
   print(y);
   }
```  

2. ```
   int y = 0;
   atomic<int> x = 0;
   y = 42; // if(xrlx == 1){
   x =rlx 1; // race
   print(y);
   }
```  

3. ```
   int y = 0;
   atomic<int> x = 0;
   y = 42; // if(xacq == 1){
   x =rel 1; // acq
   x =rel 1; // sw
   print(y);
   }
```
Synchronization in C/C++11 through examples

1. int y = 0;
   int x = 0;
   y = 42; // if(x == 1){
   x = 1; // race
   print(y);
   }

2. int y = 0;
   atomic<int> x = 0;
   y = 42; // if(x_rlx == 1){
   x = rlx 1; // race
   print(y);
   }

3. int y = 0;
   atomic<int> x = 0;
   y = 42; // if(x_acq == 1){
   x =_rel 1; // rf
   print(y);
   }

4. int y = 0;
   atomic<int> x = 0;
   y = 42; // if(x_rlx == 1){
   fence_rel; // sw
   fence_acq;
   x = rlx 1; // race
   print(y);
   }
Synchronization in C/C++11 through examples

1. int y = 0;
   int x = 0;
   y = 42; if(x == 1){
   x = 1; print(y);
   }

2. int y = 0;
   atomic<int> x = 0;
   y = 42; if(x_rlx == 1){
   x =_rlx 1; print(y);
   }

3. int y = 0;
   atomic<int> x = 0;
   y = 42; if(x_acq == 1){
   x =_rel 1; print(y);
   }

4. int y = 0;
   atomic<int> x = 0;
   y = 42; if(x_rlx == 1){
   fence_rel; x =_rlx 1; print(y);
   }
Synchronization in C/C++11 through examples

1

```c
int y = 0;
int x = 0;
y = 42;  // if(x == 1){
    x = 1,  // race
    print(y);
}
```

2

```c
int y = 0;
atomic<int> x = 0;
y = 42;  // if(x_{rlx} == 1){
    x =_{rlx} 1;  // race
    print(y);
}
```

3

```c
int y = 0;
atomic<int> x = 0;
y = 42;  // if(x_{acq} == 1){
    x =_{rel} 1;  // rf
    print(y);
}
```

4

```c
int y = 0;
atomic<int> x = 0;
y = 42;  // if(x_{rlx} == 1){
    fence_{rel};  // rf
    fence_{acq};
    print(y);
}
```
The “happens-before” relation

**Definition (happens-before)**

\[ a \xrightarrow{p\circ} b \]

\[ a \xrightarrow{r\circ} b \]

\[ a \xrightarrow{h\circ} b \]

\[ a \xrightarrow{h\circ} b \]

\[ a \xrightarrow{h\circ} b \]

\[ b \xrightarrow{h\circ} c \]

\[ a \xrightarrow{h\circ} c \]

\[ W \times 0 \quad W y 0 \]

\[ W_{rlx} y 42 \quad R_{acq} \times 1 \]

\[ W_{rel} \times 1 \quad R_{rlx} y 0 \]
C11 coherence

» Require (for every location x) that:

\[ \text{hb} \cap \{ \langle a, b \rangle \mid \text{loc}(a) = \text{loc}(b) = x \} \subseteq \text{sc}_x \]

(instead of \((\text{po} \cup \text{rf}) \cap \{ \langle a, b \rangle \mid \text{loc}(a) = \text{loc}(b) = x \} \subseteq \text{sc}_x \)).

» Using acquire CAS’s and release writes, we can implement locks.
Using $\text{hb}$, we can formally define a race in this model.

**Definition**

Two events $a, b$ form a *race* in an execution $G$ if the following hold:

- $\text{loc}(a) = \text{loc}(b)$.
- $a \in W$ or $b \in W$ (\(W\) denotes the set of writes and RMWs).
- $\langle a, b \rangle \not\in \text{hb}$ and $\langle b, a \rangle \not\in \text{hb}$. 


SC accesses and fences

Store buffer

\[
\begin{align*}
x & := 1; & y & := 1; \\
a & := y; \quad \text{// 0} & b & := x; \quad \text{// 0}
\end{align*}
\]

How to guarantee only SC behaviors (i.e., \( a = 1 \lor b = 1 \))?
SC accesses and fences

Store buffer

\[
\begin{align*}
    x &:= 1; & y &:= 1; \\
    a &:= y; & b &:= x; \quad \parallel 0
\end{align*}
\]

How to guarantee only SC behaviors (i.e., \(a = 1 \lor b = 1\))?

\[
\begin{align*}
    x &:=_{sc} 1; & y &:=_{sc} 1; \\
    a &:= y_{sc}; & b &:= x_{sc}; \\
\Rightarrow
\end{align*}
\]

\[
\begin{align*}
    x &:=_{rlx} 1; & y &:=_{rlx} 1; \\
    a &:= y_{rlx}; & b &:= x_{rlx}; \\
\end{align*}
\]
Semantics of SC accesses

Basic idea: Require an \( \text{sc} \)-order on SC accesses:

- \( \text{sc} \) is a total order on all SC accesses in \( G \).

- \( \text{hb} \cap \{ \langle a, b \rangle \mid \text{mod}(a) = \text{mod}(b) = \text{sc} \} \subseteq \text{sc} \).

- If \( \langle a, b \rangle \in \text{rf} \) then there does not exist \( c \in \text{W}_{\text{loc}}(a) \) such that \( \langle a, c \rangle \in \text{sc} \) and \( \langle c, b \rangle \in \text{sc} \).

- Semantics of SC fences?

- Accesses to the same location in both SC and non-SC modes?

This is a complicated part of the model:

[Repairing Sequential Consistency in C/C++11 PLDI’17]
Revisiting sequential consistency

**Definition**

$G$ is *SC-consistent* if there exists a relation $\text{sc}$ such that:

- $\text{sc}$ is a total order on the events of $G$.
- $\text{po} \cup \text{rf} \subseteq \text{sc}$.
- If $\langle a, b \rangle \in \text{rf}$ then there does not exist $c \in \mathbb{W}_\text{loc}(a)$ such that $\langle a, c \rangle \in \text{sc}$ and $\langle c, b \rangle \in \text{sc}$.

Can we avoid ordering *all* events?
Example

[Sezgin '04]
**Definition (Modification order (a.k.a. coherence order))**

$\text{mo}$ is called a *modification order* for an execution graph $G$ if $\text{mo} = \bigcup_{x \in \text{Loc}} \text{mo}_x$ where each $\text{mo}_x$ is a total order on $W_x$.

**Definition (Alternative SC definition)**

An execution graph $G$ is called *SC-consistent* if the following holds:

- There exists a modification order $\text{mo}$ for $G$ such that $\text{po} \cup \text{rf} \cup \text{mo} \cup \text{fr}$ is acyclic where:
  - $\text{fr} \overset{\text{def}}{=} \text{rf}^{-1}; \text{mo} \setminus \text{id}$ (from-reads / reads-before)
Theorem

The two SC definitions are equivalent.

Proof (sketch).

Original SC $\implies$ alternative SC:

- Take $\text{mo}_x$ to be the restriction of $\text{sc}$ on $W_x$.
- Then, $\text{po} \cup \text{rf} \cup \text{mo} \cup \text{fr} \subseteq \text{sc}$.

Alternative SC $\implies$ original SC:

- Take $\text{sc}$ to be any total order extending $\text{po} \cup \text{rf} \cup \text{mo} \cup \text{fr}$. □
Alternative definition of SC-per-location

Definition (SC-per-location)

$G$ satisfies \textit{SC-per-location} if for every location $x$, there exists a relation $sc_x$ such that:

- $sc_x$ is a total order on the events of $G$ that access $x$.
- $(po \cup rf) \cap \{ \langle a, b \rangle \mid \text{loc}(a) = \text{loc}(b) = x \} \subseteq sc_x$.
- If $\langle a, b \rangle \in rf$ then there does not exist $c \in W_x$ such that $\langle a, c \rangle \in sc_x$ and $\langle c, b \rangle \in sc_x$.

Definition (Alternative SC-per-location definition)

An execution graph $G$ is satisfies \textit{SC-per-location} if the following holds:

- There exists a modification order $mo$ for $G$ such that $po|_{loc} \cup rf \cup mo \cup fr$ is acyclic where:
  - $fr \overset{\text{def}}{=} rf^{-1}; mo \setminus id$ (from-reads / reads-before)
  - $po|_{loc} = po \cap (\langle a, b \rangle \mid \text{loc}(a) = \text{loc}(b))$
Similar “optimization” is done in the full C11 model, requiring that:

\[ \text{hb}|_{\text{loc}} \cup \text{rf} \cup \text{mo} \cup \text{fr} \text{ is acyclic} \]

This acyclicity condition can be simplified to several “bad patterns”.
“Bad patterns” I

\[
\begin{align*}
R_X &\quad \text{no-future-read} \\
W_X &\quad \text{rmw-1}
\end{align*}
\]

\[
\begin{align*}
r &:= \text{CAS}(x, 1, 1) \ // \ 1 \\
a &:= x \ // \ 1 \\
x &:= 1
\end{align*}
\]

Notations:
- \(W\) is either a write or an RMW.
- \(R\) is either a read or an RMW.
“Bad patterns” II

\[
x := 1 \quad \mid \quad a := x \parallel 2
\]
\[
x := 2 \quad \mid \quad a := x \parallel 1
\]
“Bad patterns” III

rmw-2

atomicity
Alternative definition via “Bad patterns”

**Theorem**

Let $\text{mo}$ be a modification order for an execution graph $G$. $\text{hb}\mid_{\text{loc}} \cup \text{rf} \cup \text{mo} \cup \text{fr}$ is acyclic iff the following hold:

- $\text{rf}; \text{hb}$ is irreflexive. \hspace{2cm} (no-future-read)
- $\text{mo}; \text{hb}$ is irreflexive. \hspace{2cm} (coherence-ww)
- $\text{mo}; \text{rf}; \text{hb}$ is irreflexive. \hspace{2cm} (coherence-rw)
- $\text{fr}; \text{hb}$ is irreflexive. \hspace{2cm} (coherence-wr)
- $\text{fr}; \text{rf}; \text{hb}$ is irreflexive. \hspace{2cm} (coherence-rr)
- $\text{rf}$ is irreflexive. \hspace{2cm} (rmw-1)
- $\text{mo}; \text{rf}$ is irreflexive. \hspace{2cm} (rmw-2)
- $\text{fr}; \text{mo}$ is irreflexive. \hspace{2cm} (rmw-atomicity)

Side note: Java plain accesses do not guarantee (coherence-rr).
Takeaways

▶ C/C++11 concurrency semantics is specified *declaratively* using formal constrains on execution graphs.

▶ This is a *highly flexible* framework that abstracts away all execution details.

▶ Reasoning about program behaviors requires analyzing *possible cycles* in execution graphs.

▶ A declarative approach to concurrency semantics is prominent in other systems as well:

▶ *Multicore architectures:*
  Jade Alglave, Luc Maranget, Michael Tautschnig.
  Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory.

▶ *Distributed systems:*
  Sebastian Burckhardt.
  Principles of Eventual Consistency.