Relaxed Semantics of Concurrent Programs

G. Boudol, G. Petri and B. Serpette
EXPRESS-SOS 2012
The PROBLEM

- shared memory concurrency – aka multithreading = the concurrent programming model in mainstream programming languages (e.g. C/C++, JAVA), close to hardware multiprocessor architectures

- in shared memory multiprocessor ( multicore) architectures, memory accesses may be reordered (also with optimizing compilers)

⇒ there are behaviors of multithreaded programs that are not explained by the standard interleaving semantics

Formalization of such relaxed semantics?
An Example

Initially \( S(p) = \text{ff} = S(q) \)

\[
p := \text{tt}; \quad \parallel \quad q := \text{tt}; \\
r_0 := \lnot q \quad \parallel \quad r_1 := \lnot p
\]

Possible outcomes with the interleaving semantics:

\[
S(r_0) = \text{ff} \quad \& \quad S(r_1) = \text{tt} \\
S(r_0) = \text{tt} \quad \& \quad S(r_1) = \text{ff} \\
S(r_0) = \text{tt} \quad \& \quad S(r_1) = \text{tt}
\]

On most multiprocessor machines, one may also observe

\[
S(r_0) = \text{ff} \quad \& \quad S(r_1) = \text{ff}
\]
The two threads are executed on different processors

1. the write $p := tt$ is issued and put in a write buffer, but not yet performed on the shared memory

2. the read $!q$ is performed, returning the value $fff$ from the shared memory

3. the write $q := tt$ is performed, then

4. the read $!p$ is performed and returns $fff$ from the shared memory, since the second processor does not see the write buffer of the first

⇒ the write $p := tt$ and the read $!q$ appear to be reordered
Another (similar) Example

Initially $S(p) = ff = S(q)$

\[
p := tt; \quad r_0 := !q; \\
q := tt \quad r_1 := !p
\]

On some multiprocessor machines, one may observe the outcome

\[
S(r_0) = tt \quad \& \quad S(r_1) = ff
\]

Reordering of the writes $p := tt$ and $q := tt$ (conceptually: one write buffer per memory location), or of the reads $!q$ and $!p$
**Memory Barriers**

Low level instructions to *prevent* reorderings: full fence, or weaker wr, ww, rr, rw, for instance

\[
p := tt; \quad \parallel \quad q := tt;
wr;
\quad \parallel \quad wr;
\]
\[
\quad r_0 := !q \ (ff) \quad \parallel \quad r_1 := !p \ (ff)
\]

is forbidden. However

\[
p := tt; \quad r_0 := !q \ (tt)
ww; \quad rr;
\]
\[
q := tt \quad r_1 := !p \ (ff)
\]

is still possible (on some machines)
Non Atomic Writes

The IRIW (Independent Reads of Independent Writes) example:

\[ p := tt \parallel q := tt \parallel r_0 := !p \ (tt) \parallel r_2 := !q \ (tt) \]
\[ r_1 := !q \ (ff) \parallel r_3 := !p \ (ff) \]

A possible explanation: the write \( p := tt \) is issued and made visible to the third thread, but not to the fourth (the previous example is similar). Notice: no reordering
Relaxed Memory Models

According to Adve & Gharachorloo (Tutorial, 1996):

memory model = reordering, i.e. relaxation of program order, for memory accesses (write/read)
+ write visibility
+ means to maintain program order in some cases (memory barriers, acquire/release...)

Some examples:

- TSO: relaxing write/read + read-own-write-early
- PSO: TSO + write/write relaxation
- RMO: PSO + read/write + read/read relaxations
- PowerPC: RMO + read-others’-write-early
**Semantics: Approaches**

- **axiomatic**: an execution is a set of memory events, connected by various relations (po, rf, co... + various dependencies: data, addr, ctrl...), subject to a number of axioms. Given a set of events, one tentatively sets relations between them and checks if the axioms are satisfied.

- **operational**: an abstract machine with transitions from configurations (shared memory, threads, interconnection structure – write buffers, caches...) to configurations. Rules to non-deterministically choose the next step.
A Language

Syntax – a call by value, imperative λ-calculus:

\[ p, q, r \ldots \in \mathcal{P} \quad \text{pointers} \]
\[ b \in \mathcal{B} \quad \text{barriers} \]
\[ v, w \ldots ::= p \mid x \mid \lambda x e \mid \emptyset \mid \cdots \quad \text{values} \]
\[ e ::= v \mid (ve) \mid (!v) \mid (v := w) \mid b \mid \cdots \quad \text{expressions} \]

Evaluation contexts

\[ E ::= [] \mid (vE) \]

Recall:

\[ e_0 ; e_1 = \text{def} (\lambda x e_1 e_0) = (\lambda x e_1[e_0]) \]

(\text{where } x \text{ is not free in } e_1)
**Configurations**

\((S, \sigma, T)\) where

- **S**, the *shared memory*, is a mapping from a finite set of pointers to closed values

- **T**, the thread system, is a mapping from a finite set of thread identifiers \(\{t_1, \ldots, t_n\}\) to expressions, written

\[
(t_1, e_1) \parallel \cdots \parallel (t_n, e_n)
\]

- **\(\sigma\)** is the *temporary store*, a sequence \((t_{i_1}, \mu_1) \cdots (t_{i_k}, \mu_k)\) of pending memory operations \(\mu_1, \ldots, \mu_k\) issued by the threads \(t_{i_1}, \ldots, t_{i_k}\), but not yet “globally performed” (i.e. not yet touching the memory). An abstraction of the interconnection structure between processors and the shared memory
\[ \mu ::= (v := w)^W \mid !^x v \mid b \]

- **write** operations \((v := w)^W\) where \(v\) is the location to update (either a pointer or a variable, if not yet determined), \(w\) the new value, \(W\) is the **visibility** of the write, a set of thread identifiers

- **read** operations \(!^x v\) where \(v\) is the location to read and \(x\) the place-holder for the value in the thread that reads (and in subsequent memory operations)
From the threads:

\[
(\sigma, (t, e)) \rightarrow (\sigma', (t', e')) \\
\frac{(S, \sigma, (t, e) \parallel T) \rightarrow (S, \sigma', (t', e') \parallel T)}{M}
\]

where

\[
(\sigma, (t, E[(\lambda x e \, v)])) \rightarrow (\sigma, (t, E[{x \mapsto v}]e))
\]

\[
(\sigma, (t, E[(!v)])) \rightarrow (\sigma \cdot (t, !^x v), (t, E[x])) \quad \text{x fresh}
\]

\[
(\sigma, (t, E[(v := w)])) \rightarrow (\sigma \cdot (t, (v := w)^\emptyset), (t, E[\emptyset]))
\]

\[
(\sigma, (t, E[b])) \rightarrow (\sigma \cdot (t, b), (t, E[\emptyset]))
\]
**Memory Model**

\[ M = (\varpi, \mathcal{W}) \] where

- \( \varpi \), the **commutability** predicate, relates temporary stores \( \sigma \) with pending memory operations \( (t, \mu) \). If \( \sigma \varpi (t, \mu) \) then \( \mu \) may be immediately performed, overtaking the operations in \( \sigma \) – allowing reorderings/relaxations of the program order

- \( \mathcal{W} \), the **write grain**, is a set of sets of thread identifiers – the allowed visibilities

subject to some requirements, e.g. \( \varepsilon \varpi (t, \mu) \) i.e. the empty temporary store allows any memory operation to be performed, \( \emptyset \in \mathcal{W}, \ldots \)
From the temporary store:

\[(S, \sigma) \leftrightarrow (S', \sigma', \text{Sub}) \Rightarrow (S, \sigma, T) \xrightarrow{\mathcal{M}} (S', \text{Sub}(\sigma', T))\]

where

\[(S, \sigma) \leftrightarrow (S, \sigma_0 \cdot \sigma_1, \{x \mapsto v\})\] \hspace{1cm} \text{read}

\[\text{if } \sigma = \sigma_0 \cdot (t, !^x p) \cdot \sigma_1 \text{ & } \sigma_0 \uparrow (t, !^x p) \text{ & } S(p) = v\]

\[(S, \sigma) \leftrightarrow (S[p := v], \sigma_0 \cdot \sigma_1, \emptyset)\] \hspace{1cm} \text{write}

\[\text{if } \sigma = \sigma_0 \cdot (t, (p := v)^W) \cdot \sigma_1 \text{ & } \sigma_0 \uparrow (t, (p := v)^W) \text{ & } v \text{ closed}\]
Transitions

$(S, \sigma) \xrightarrow{} (S, \sigma_0 \cdot \sigma_1, \emptyset)$

if $\sigma = \sigma_0 \cdot (t, b) \cdot \sigma_1$ & $\sigma_0 \leftarrow (t, b)$

$(S, \sigma) \xrightarrow{} (S, \sigma_0 \cdot (t, (v := w)^{W'}) \cdot \sigma_1, \emptyset)$

if $\sigma = \sigma_0 \cdot (t, (v := w)^{W}) \cdot \sigma_1$ & $t \in W'$ & $W \subset W' \in \mathcal{W}$

$(S, \sigma) \xrightarrow{} (S, \sigma_0 \cdot \sigma_1, \{x \mapsto w\})$

if $\sigma = \sigma_0 \cdot (t, !^x v) \cdot \sigma_1$ & $\sigma_0 = \delta_0 \cdot (t', (v := w)^W) \cdot \delta_1$ & $t \in W$ & $\delta_1 \leftarrow (t, !^x v)$ & $\delta_0 \leftarrow^B (t, !^x v)$
Memory Models: Requirements

The commutability predicate should not be so relaxed that the semantics of sequential programs could be broken. Then $\sqsubseteq$ must satisfy

$$(t, \mu) \sqsubseteq (t', \mu) \Rightarrow \forall \sigma, \sigma'. \neg (\sigma \cdot (t, \mu) \cdot \sigma' \sqsubseteq (t', \mu'))$$

where the precedence relation $\sqsubseteq$ is inductively given by

$$v \approx v' \& t' \in \{t\} \cup W \Rightarrow \left\{ \begin{array}{l} (t, (v := w)^W) \sqsubseteq (t', !^x v') \& (t, (v := w)^W) \sqsubseteq (t', (v' := w')^W) \\ v \approx v' \Rightarrow (t, !^x v) \sqsubseteq (t, (v' := w)^W) \end{array} \right.$$
**Example 1**

\[ p := tt; \quad \parallel \quad q := tt; \]
\[ r_0 := !q (ff) \quad \parallel \quad r_1 := !p (ff) \]

The initial configuration may evolve into

\[(S, (t_0, (p := tt)\emptyset) \cdot (t_0, !^x q), (t_0, (r_0 := x)) \parallel (t_1, e_1))\]

With the relaxation of the write/read order:

\[(t_0, (p := tt)\emptyset) \leftarrow (t_0, !^x q)\]

thus \( !^x q \) can be performed, returning \( \{x \mapsto ff\} \). Then \( e_1 \) is executed: the write \((q := tt)\emptyset\) commutes with \((p := tt)\emptyset\), as well as the read \(!^y p\) (which does not see this write), and finally \((p := tt)\emptyset\) is performed.
**Memory Barriers: Semantics**

By means of the commutability predicate:

\[ (t, (v := w)^W) \triangleright (t, ww) \triangleright (t, (v' := w')^{W'}) \]
\[ (t, (v := w)^W) \triangleright (t, wr) \triangleright (t, (\!^x v')) \]

Notice: same thread

Global barrier: `sync` is a `ww`, `wr`, `rw` and `rr` (local) barrier, plus sees the writes from other threads:

\[ t' \in W \Rightarrow (t, (v := w)^W) \triangleright (t', \text{sync}) \]
Example 2

\[ p := tt; \quad r_0 := !q ; (tt) \]
\[ \text{ww}; \quad \| \quad \text{rr}; \]
\[ q := tt \quad r_1 := !p \ (ff) \]

From the temporary store

\[(t_0, (p := tt)^0) \cdot (t_0, \text{ww}) \cdot (t_0, (q := tt)^0)\]

the visibility of the last write is extended to the second thread:

\[(t_0, (p := tt)^0) \cdot (t_0, \text{ww}) \cdot (t_0, (q := tt)^{\{t_0, t_1\}})\]

Then \( t_1 \) proceeds: \( !^x q \) returns \( \{x \mapsto tt\} \) from the temporary store, the barrier \( \text{rr} \) vanishes, and \( !^y p \) returns \( \{y \mapsto ff\} \) from the shared memory. Replacing \( \text{rr} \) with \( \text{sync} \) prevents this behavior.
In the Paper

- a more refined abstract machine to deal with the (subtle) \textit{lwsync} barrier (PowerPC)

\[ \sigma \leftarrow (t, \mu) \neq \forall (t', \mu') \text{ in } \sigma. \neg ((t', \mu') \blacktriangleright (t, \mu)) \]

- extension with \textit{speculation} – branch prediction: from a conditional branching \textit{(if} \textit{v} \textit{then} \textit{e}_0 \textit{else} \textit{e}_1) one issues a prediction \textit{[v = tt]} or \textit{[v = ff]} in the temporary store. A correct prediction \textit{[v = v]} vanishes, while a prediction blocks memory updates:

\[ (t, [v = w]) \blacktriangleright (t, (v' := w')^W) \]

- a \textit{software simulator} that allows us to run a (large) number of \textit{litmus tests}, despite state explosion – trying to explore these without the simulator is highly error prone!