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 $S(p) = \mathit{ff} = S(q)$

$$\begin{array}{l} p := \mathit{tt}; \\ r_0 := !q \end{array} \;\Big\|\; \begin{array}{l} q := \mathit{tt}; \\ r_1 := !p \end{array}$$

Possible outcomes with the interleaving semantics:

$$\begin{array}{lcl} S(r_0) = \mathit{ff} & \& & S(r_1) = \mathit{tt} \\ S(r_0) = \mathit{tt} & \& & S(r_1) = \mathit{ff} \\ S(r_0) = \mathit{tt} & \& & S(r_1) = \mathit{tt} \end{array}$$

On most multiprocessor machines, one may also observe

$$S(r_0) = \mathit{ff} \quad \& \quad S(r_1) = \mathit{ff}$$

Explanation

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 ff from the shared memory
3. the write $q := tt$ is performed, then
4. the read $!p$ is performed and returns ff 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 $S(p) = \mathit{ff} = S(q)$

$$\begin{array}{l} p := tt; \\ q := tt \end{array} \;\Big\|\; \begin{array}{l} r_0 := !q; \\ r_1 := !p \end{array}$$

On some multiprocessor machines, one may observe the outcome

$$S(r_0) = tt \quad \& \quad S(r_1) = \mathit{ff}$$

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

$$\begin{array}{l} p := tt; \\ \textsf{wr}; \\ r_0 := !q\ (\mathit{ff}\,) \end{array} \quad \Bigg\| \quad \begin{array}{l} q := tt; \\ \textsf{wr}; \\ r_1 := !p\ (\mathit{ff}\,) \end{array}$$

is forbidden. However

$$\begin{array}{l} p := tt; \\ \textsf{ww}; \\ q := tt \end{array} \quad \Bigg\| \quad \begin{array}{l} r_0 := !q\ ;\ \ (tt) \\ \textsf{rr}; \\ r_1 := !p\ (\mathit{ff}\,) \end{array}$$

is still possible (on some machines)

Non Atomic Writes

The IRIW (Independent Reads of Independent Writes) example:

$$p := tt \quad \| \quad q := tt \quad \left\| \begin{array}{l} r_0 := \,!p\,;\ \ (tt) \\ \mathsf{rr}; \\ r_1 := \,!q\ (\mathit{ff}) \end{array} \right. \quad \left\| \begin{array}{l} r_2 := \,!q\,;\ \ (tt) \\ \mathsf{rr}; \\ r_3 := \,!p\ (\mathit{ff}) \end{array} \right.$$

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:

$$\begin{array}{rcll}
p,\ q,\ r\ldots & \in & \mathcal{P} & \textit{pointers} \\
b & \in & \mathcal{B} & \textit{barriers} \\
v,\ w\ldots & ::= & p \mid x \mid \lambda x e \mid () \mid \cdots & \textit{values} \\
e & ::= & v \mid (ve) & \textit{expressions} \\
 & \mid & (!v) \mid (v := w) \mid b \mid \cdots &
\end{array}$$

Evaluation contexts

$$\mathbf{E} ::= [] \mid (v\mathbf{E})$$

Recall:

$$e_0\,;e_1 =_{\mathrm{def}} (\lambda x e_1\, e_0) = (\lambda x e_1[e_0])$$

(where x is not free in e_1)

Configurations

(S, σ, 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, witten

$$(t_1, e_1) \parallel \cdots \parallel (t_n, e_n)$$

- σ 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 abtraction of the interconnection structure between processors and the shared memory

Memory Operations

$$\mu \ ::= \ (v := w)^W \ \mid \ !^x v \ \mid \ b$$

- 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)

Transitions (1/3)

From the threads:

$$\frac{(\sigma, (t, e)) \to (\sigma', (t', e'))}{(S, \sigma, (t, e) \parallel T) \xrightarrow[\mathcal{M}]{} (S, \sigma', (t', e') \parallel T)}$$

where

$$\begin{array}{rcll}
(\sigma, (t, \mathbf{E}[(\lambda x e\, v)])) & \to & (\sigma, (t, \mathbf{E}[\{x \mapsto v\} e])) & \\
(\sigma, (t, \mathbf{E}[(!v)])) & \to & (\sigma \cdot (t, !^{x} v), (t, \mathbf{E}[x])) & x \text{ fresh} \\
(\sigma, (t, \mathbf{E}[(v := w)])) & \to & (\sigma \cdot (t, (v := w)^{\emptyset}), (t, \mathbf{E}[()])) & \\
(\sigma, (t, \mathbf{E}[b])])) & \to & (\sigma \cdot (t, b), (t, \mathbf{E}[()])) &
\end{array}$$

Memory Model

$\mathcal{M} = (\Lsh, \mathcal{W})$ where

- $\Lsh$, the commutability predicate, relates temporary stores σ with pending memory operations (t, μ). If $\sigma \Lsh (t, \mu)$ then μ may be immediately performed, overtaking the operations in σ – 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. $\varepsilon \Lsh (t, \mu)$ i.e. the empty temporary store allows any memory operation to be performed, $\emptyset \in \mathcal{W}, \ldots$

Transitions (2/3)

From the temporary store:

$$(S,\sigma) \hookrightarrow (S',\sigma',\mathsf{Sub}) \;\Rightarrow\; (S,\sigma,T) \xrightarrow[\mathcal{M}]{} (S',\mathsf{Sub}(\sigma',T))$$

where

$$\begin{array}{lcll}
(S,\sigma) & \hookrightarrow & (S,\sigma_0\cdot\sigma_1,\{x\mapsto v\}) & \textit{read} \\
& & \textit{if}\quad \sigma=\sigma_0\cdot(t,!^x p)\cdot\sigma_1\ \& & \\
& & \qquad \sigma_0 \Lsh (t,!^x p)\ \&\ S(p)=v & \\
(S,\sigma) & \hookrightarrow & (S[p:=v],\sigma_0\cdot\sigma_1,\emptyset) & \textit{write} \\
& & \textit{if}\quad \sigma=\sigma_0\cdot(t,(p:=v)^W)\cdot\sigma_1\ \& & \\
& & \qquad \sigma_0 \Lsh (t,(p:=v)^W)\ \&\ v\ \text{closed} &
\end{array}$$

Transitions (3/3)

$$(S,\sigma) \hookrightarrow (S, \sigma_0 \cdot \sigma_1, \emptyset) \qquad \textit{barrier}$$

$$\textit{if} \quad \sigma = \sigma_0 \cdot (t,b) \cdot \sigma_1 \ \& \ \sigma_0 \Lsh (t,b)$$

$$(S,\sigma) \hookrightarrow (S, \sigma_0 \cdot (t, (v := w)^{W'}) \cdot \sigma_1, \emptyset) \qquad \textit{write early}$$

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

$$(S,\sigma) \hookrightarrow (S, \sigma_0 \cdot \sigma_1, \{x \mapsto w\}) \qquad \textit{read early}$$

$$\textit{if} \quad \sigma = \sigma_0 \cdot (t, !^x v) \cdot \sigma_1 \ \& \\ \sigma_0 = \delta_0 \cdot (t', (v := w)^{W}) \cdot \delta_1 \ \& \\ t \in W \ \& \ \delta_1 \Lsh (t, !^x v) \ \& \ \delta_0 \Lsh^{\mathcal{B}} (t, !^x v)$$

Memory Models: Requirements

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

$$(t,\mu) \blacktriangleleft (t',\mu) \;\Rightarrow\; \forall\sigma,\sigma'.\ \neg\big(\sigma\cdot(t,\mu)\cdot\sigma' \Lsh (t',\mu')\big)$$

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

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

$$v \approx v' \;\Rightarrow\; (t,!^x v) \blacktriangleleft (t,(v' := w)^W)$$

where

$$v \approx v' \;\Leftrightarrow_{\mathrm{def}}\; v = v' \text{ or } v \in \mathcal{V}ar \text{ or } v' \in \mathcal{V}ar$$

Example 1

$$\begin{array}{l} p := tt; \\ r_0 := !q\ (ff) \end{array} \quad \Big\| \quad \begin{array}{l} q := tt; \\ r_1 := !p\ (ff) \end{array}$$

The initial configuration may evolve into

$$(S, \quad (t_0, (p := tt)^{\emptyset}) \cdot (t_0, !^x q), \quad (t_0, (r_0 := x)) \,\|(t_1, e_1))$$

With the relaxation of the write/read order:

$$(t_0, (p := tt)^{\emptyset}) \;\Lsh\; (t_0, !^x q)$$

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:

$$\begin{array}{ccccc} (t, (v := w)^W) & \blacktriangleleft & (t, \mathsf{ww}) & \blacktriangleleft & (t, (v' := w')^{W'}) \\ (t, (v := w)^W) & \blacktriangleleft & (t, \mathsf{wr}) & \blacktriangleleft & (t, (!^x v')) \\ & & \vdots & & \end{array}$$

Notice: same thread

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

$$t' \in W \;\Rightarrow\; (t, (v := w)^W) \;\blacktriangleleft\; (t', \mathsf{sync})$$

Example 2

$$\begin{array}{l} p := tt; \\ \mathsf{ww}; \\ q := tt \end{array} \quad \Big\| \quad \begin{array}{l} r_0 := \,!q\,; \;\; (tt) \\ \mathsf{rr}; \\ r_1 := \,!p \; (\mathit{ff}) \end{array}$$

From the temporary store

$$(t_0, (p := tt)^{\emptyset}) \cdot (t_0, \mathsf{ww}) \cdot (t_0, (q := tt)^{\emptyset})$$

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

$$(t_0, (p := tt)^{\emptyset}) \cdot (t_0, \mathsf{ww}) \cdot (t_0, (q := tt)^{\{t_0, t_1\}})$$

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

In the Paper

- a more refined abstract machine to deal with the (subtle) **lwsync** barrier (PowerPC)

$$\sigma \upharpoonleft (t,\mu) \;\neq\; \forall (t',\mu') \text{ in } \sigma.\ \neg\big((t',\mu') \blacktriangleleft (t,\mu)\big)$$

- extension with speculation – branch prediction: from a conditional branching (**if** v **then** e_0 **else** e_1) one issues a prediction $[v = tt]$ or $[v = f\!f]$ in the temporary store. A correct prediction $[v = v]$ vanishes, while a prediction blocks memory updates:

$$(t,[v = w]) \;\blacktriangleleft\; (t,(v' := w')^W)$$

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