An overview of Jaroslav Ševčík's trace-set transformation semantics from "Safe Optimisations for Shared-Memory Concurrent Programs" (PLDI'11)

Jean Pichon

24th of September 2014

Goal

Replacement memory model for Java,
for which common subexpression elimination is sound
(and other typical optimisations too).

N.B. This is not the case for the current Java Memory Model.

Idea

Starting from the "naive" trace-set, the semantics is given by the closure of the trace-set under reorderings and optimisations.

Reordering and elimination of memory actions *one by one* on *thread-local* trace-sets.

Trace-sets

Thread

$$\mathtt{y} =_{\mathtt{NA}} \mathtt{x}_{\mathtt{NA}}$$

has naive trace-set (the prefix-closure of)

$$\{\mathtt{R}_{\mathtt{NA}}\,\mathtt{x}\,\mathtt{0}\;\;\mathtt{W}_{\mathtt{NA}}\,\mathtt{y}\,\mathtt{0},\;\;\mathtt{R}_{\mathtt{NA}}\,\mathtt{x}\,\mathtt{1}\;\;\mathtt{W}_{\mathtt{NA}}\,\mathtt{y}\,\mathtt{1},\;\;\ldots\}$$

(one for each value in the domain of $\mathtt{x}$).

Reordering

Reorder two adjacent memory actions in a (thread-local) trace.

Example: thread $\mathtt{x} =_{\mathtt{NA}} 1; \mathtt{y} =_{\mathtt{NA}} 2$
has naive trace-set (the prefix closure of) $\{\mathtt{W}_{\mathtt{NA}}\,\mathtt{x}\,1 \;\; \mathtt{W}_{\mathtt{NA}}\,\mathtt{y}\,2\}$.

Now, because the two writes in the trace $\mathtt{W}_{\mathtt{NA}}\,\mathtt{x}\,1 \;\; \mathtt{W}_{\mathtt{NA}}\,\mathtt{y}\,2$ are
1) at two different locations,
2) not volatile,
3) non-dependent (here, because they are writes)[1],
they can be reordered:

$$\mathtt{W}_{\mathtt{NA}}\,\mathtt{x}\,1 \;\; \mathtt{W}_{\mathtt{NA}}\,\mathtt{y}\,2 \quad \rightarrow \quad \mathtt{W}_{\mathtt{NA}}\,\mathtt{y}\,2 \;\; \mathtt{W}_{\mathtt{NA}}\,\mathtt{x}\,1$$

Note: therefore, $\mathtt{y} =_{\mathtt{NA}} 2; \mathtt{x} =_{\mathtt{NA}} 1$ is a valid "optimisation".

[1] see later

Roach motel reordering

Roach motel reordering is a design goal of Java[2].

Non-atomic actions can be moved after a lock or volatile read:

$$t_1 \;\; \mathtt{W}_{\mathtt{NA}}\,\mathtt{x}\,\mathtt{1} \;\; \mathtt{L}\,\ell \;\; t_2 \rightarrow t_1 \;\; \mathtt{L}\,\ell \;\; \mathtt{W}_{\mathtt{NA}}\,\mathtt{x}\,\mathtt{1} \;\; t_2$$

but *not* the other way around!

Symmetrically, non-atomic actions can be moved before an unlock or volatile write:

$$t_1 \;\; \mathtt{U}\,\ell \;\; \mathtt{W}_{\mathtt{NA}}\,\mathtt{x}\,\mathtt{1} \;\; t_2 \rightarrow t_1 \;\; \mathtt{W}_{\mathtt{NA}}\,\mathtt{x}\,\mathtt{1} \;\; \mathtt{U}\,\ell \;\; t_2$$

[2] to the best of my knowledge, it is not done by any compiler, at least for C — please tell me if I'm wrong, I would really like to know!

Optimisations: release/acquire pairs

Definition: A *release/acquire pair*[3] is the pair (in that order) of a volatile write and a volatile read (not necessarily to the same location), or of an unlock and lock (not necessarily of the same lock).

Motivation: this is enough to enforce synchronisation.

```
d =_NA 0
x =_V 1
                    while !x_V {};
                    d =_NA 42
                    y =_V 1
while !y_V {};
d_NA // 42
```

[3] nothing to do with C11's release/acquire

Peephole optimisations: RaR, WaR, RaW, OWE

We can define elimination of redundant read after read, redundant write after read, redundant read after write, and overwritten write.

For example, read after read:

$$t_1 \;\; \mathtt{R}_{\mathtt{NA}}\,\mathrm{x}\,1 \;\; t_2 \;\; \mathtt{R}_{\mathtt{NA}}\,\mathrm{x}\,1 \;\; t_3 \rightarrow t_1 \;\; \mathtt{R}_{\mathtt{NA}}\,\mathrm{x}\,1 \;\; t_2 \;\; t_3$$

if t_2 does not contain any release/acquire pair.

...and overwritten write:

$$t_1 \;\; \mathtt{W}_{\mathtt{NA}}\,\mathrm{x}\,1 \;\; t_2 \;\; \mathtt{W}_{\mathtt{NA}}\,\mathrm{x}\,2 \;\; t_3 \rightarrow t_1 \;\; t_2 \;\; \mathtt{W}_{\mathtt{NA}}\,\mathrm{x}\,2 \;\; t_3$$

if t_2 does not contain any release/acquire pair.

Irrelevant read elimination

If the trace-set contains all the traces of the form $t_1 \;\; \mathtt{R}_{\mathtt{NA}}\,\mathtt{x}\,\mathtt{v} \;\; t_2$, for all values $\mathtt{v}$ in the domain of $\mathtt{x}$, then that read is *irrelevant*, and can be removed, to yield trace $t_1 \;\; t_2$.

For example, thread

$$\mathtt{z} =_{\mathtt{NA}} 1; \mathtt{y} =_{\mathtt{NA}} \mathtt{x}_{\mathtt{NA}} * 0$$

has (naive) trace-set

$$\{\mathtt{W}_{\mathtt{NA}}\,\mathtt{z}\,1 \;\; \mathtt{R}_{\mathtt{NA}}\,\mathtt{x}\,0 \;\; \mathtt{W}_{\mathtt{NA}}\,\mathtt{y}\,0, \;\; \mathtt{W}_{\mathtt{NA}}\,\mathtt{z}\,1 \;\; \mathtt{R}_{\mathtt{NA}}\,\mathtt{x}\,1 \;\; \mathtt{W}_{\mathtt{NA}}\,\mathtt{y}\,0, \;\; \ldots\}$$

so it also has optimised trace

$$\mathtt{W}_{\mathtt{NA}}\,\mathtt{z}\,1 \;\; \mathtt{W}_{\mathtt{NA}}\,\mathtt{y}\,0$$

Dependencies: LB vs. LB+datas

How to check for dependency: the inverse image (by the transformation) of the prefixes of the transformed trace have to be in the trace-set.

LB is allowed to return 42/42 for non-volatile:
$\mathtt{x}_{\mathtt{NA}}; \mathtt{y} =_{\mathtt{NA}} 42$ has naive trace-set (the prefix-closure of)

$$\{\mathtt{R}_{\mathtt{NA}}\,\mathtt{x}\,0 \;\; \mathtt{W}_{\mathtt{NA}}\,\mathtt{y}\,42, \;\; \mathtt{R}_{\mathtt{NA}}\,\mathtt{x}\,42 \;\; \mathtt{W}_{\mathtt{NA}}\,\mathtt{y}\,42\}$$

so the closure also contains, by IRE, $\mathtt{W}_{\mathtt{NA}}\,\mathtt{y}\,42$,
so it also contains $\mathtt{W}_{\mathtt{NA}}\,\mathtt{y}\,42 \;\; \mathtt{R}_{\mathtt{NA}}\,\mathtt{x}\,0$ and $\mathtt{W}_{\mathtt{NA}}\,\mathtt{y}\,42 \;\; \mathtt{R}_{\mathtt{NA}}\,\mathtt{x}\,42$.

LB+datas is not allowed to return 42/42:

$$\{\mathtt{R}_{\mathtt{NA}}\,\mathtt{x}\,0 \;\; \mathtt{W}_{\mathtt{NA}}\,\mathtt{y}\,0, \;\; \mathtt{R}_{\mathtt{NA}}\,\mathtt{x}\,42 \;\; \mathtt{W}_{\mathtt{NA}}\,\mathtt{y}\,42\}$$

Conclusion

Advantages:

- Clean (I didn't explain the subtleties of the technical setup, but the ideas were there)
- DRF-SC theorem
- Validates common subexpression elimination and other expected optimisations
- Validates C++'s 29.3p9 weak no-OOTA (I think)
- Proof of soundness for typical optimisations on the source
- Cheap to implement on TSO: non-atomic reads and writes can be mapped to plain reads and writes, because all TSO behaviour is accepted by the model

Disadvantages:

- Only considers thread-local optimisations
- Unknown implementation cost on POWER/ARM

This page intentionally left blank.