Verification of Concurrent Software in the Context of Weak Memory

Jade Alglave
with Daniel Kroening, Vincent Nimal and Michael Tautschnig

May 9, 2013

Sequential Consistency

A comfortable model for concurrent programming would be Sequential Consistency (SC), as defined by Leslie Lamport in 1979:

> *The result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.*

Example

Consider the following example, where initially $x = y = 0$:

sb

P_0	P_1
(a) `x` $\leftarrow$ `1`	(c) `y` $\leftarrow$ `1`
(b) `r1` $\leftarrow$ `y`	(d) `r2` $\leftarrow$ `x`

`r1=?; r2=?;`

Following SC, we expect three possible outcomes:

$(a)(b)(c)(d)$	$\text{r1} = 0 \wedge \text{r2} = 1$
$(c)(d)(a)(b)$	$\text{r1} = 1 \wedge \text{r2} = 0$
$(a)(c)(b)(d)$ $(a)(c)(d)(b)$ $(c)(a)(b)(d)$ $(c)(a)(d)(b)$	$\text{r1} = 1 \wedge \text{r2} = 1$

Example

Consider the following example, where initially $x = y = 0$:

sb

P_0	P_1
(a) x ← 1	(c) y ← 1
(b) r1 ← y	(d) r2 ← x

r1=?; r2=?;

Following SC, we expect three possible outcomes:

$(a)(b)(c)(d)$	$\text{r1} = 0 \wedge \text{r2} = 1$
$(c)(d)(a)(b)$	$\text{r1} = 1 \wedge \text{r2} = 0$
$(a)(c)(b)(d)$ $(a)(c)(d)(b)$ $(c)(a)(b)(d)$ $(c)(a)(d)(b)$	$\text{r1} = 1 \wedge \text{r2} = 1$

Example

Consider the following example, where initially $x = y = 0$:

sb

P_0	P_1
$(a)\ \mathtt{x} \leftarrow 1$	$(c)\ \mathtt{y} \leftarrow 1$
$(b)\ \mathtt{r1} \leftarrow \mathtt{y}$	$(d)\ \mathtt{r2} \leftarrow \mathtt{x}$
r1=0; r2=?;	

Following SC, we expect three possible outcomes:

$(a)(b)(c)(d)$	$\mathrm{r1} = 0 \wedge \mathrm{r2} = 1$
$(c)(d)(a)(b)$	$\mathrm{r1} = 1 \wedge \mathrm{r2} = 0$
$(a)(c)(b)(d)$ $(a)(c)(d)(b)$ $(c)(a)(b)(d)$ $(c)(a)(d)(b)$	$\mathrm{r1} = 1 \wedge \mathrm{r2} = 1$

Example

Consider the following example, where initially $x = y = 0$:

sb

P_0	P_1
$(a)\ \texttt{x} \leftarrow 1$	$(c)\ \texttt{y} \leftarrow 1$
$(b)\ \texttt{r1} \leftarrow \texttt{y}$	$(d)\ \texttt{r2} \leftarrow \texttt{x}$

`r1=0; r2=?;`

Following SC, we expect three possible outcomes:

$(a)(b)(c)(d)$	$\text{r1} = 0 \wedge \text{r2} = 1$
$(c)(d)(a)(b)$	$\text{r1} = 1 \wedge \text{r2} = 0$
$(a)(c)(b)(d)$ $(a)(c)(d)(b)$ $(c)(a)(b)(d)$ $(c)(a)(d)(b)$	$\text{r1} = 1 \wedge \text{r2} = 1$

Example

Consider the following example, where initially $x = y = 0$:

sb

P_0	P_1
$(a)\,\texttt{x} \leftarrow 1$	$(c)\,\texttt{y} \leftarrow 1$
$(b)\,\texttt{r1} \leftarrow \texttt{y}$	$(d)\,\texttt{r2} \leftarrow \texttt{x}$

r1=0; r2=1;

Following SC, we expect three possible outcomes:

Order	Outcome
$(a)(b)(c)(d)$	$\text{r1} = 0 \wedge \text{r2} = 1$
$(c)(d)(a)(b)$	$\text{r1} = 1 \wedge \text{r2} = 0$
$(a)(c)(b)(d)$ $(a)(c)(d)(b)$ $(c)(a)(b)(d)$ $(c)(a)(d)(b)$	$\text{r1} = 1 \wedge \text{r2} = 1$

Experiment

```
{x=0; y=0;}

 P0           | P1           ;
 MOV [y],$1   | MOV [x],$1   ;
 MOV EAX,[x] | MOV EAX,[y] ;

exists (0:EAX=0 /\ 1:EAX=0)
```

Let us check that on my machine.

Weak memory models

- We just observed `r1=r2=0` on my laptop
- Modern architectures allow more executions than SC
  - x86, Power or ARM
- They provide weak memory models

Porte ouverte à deux battants

We propose two ways of verifying concurrent software running on weak memory:

- we instrument the program to embed the weak memory semantics inside it, then feed the transformed program to an SC verification tool;
- we explicitly builds partial order models representing the possible executions of the program on weak memory.

Instrumentation

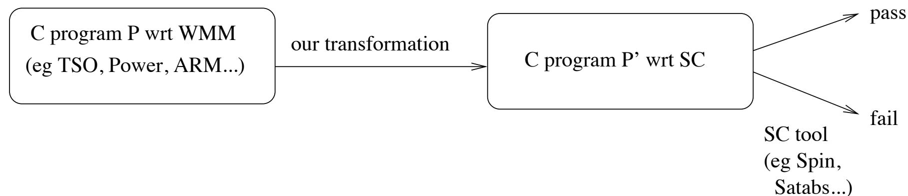
Bringing verification tools up to speed

Most verification tools assume SC: ESBMC, Poirot, SatAbs, Threader, . . .

- How can we verify concurrent programs for weak memory?
- Without having to rewrite all of these tools?
- For every architecture?

Rather than modifying a tool, we modify its input.

Instrumentation stategy



Instrumentation

Instrumenting writes

Consider the following program on SC:

sb

P_0	P_1
(a) `x` $\leftarrow$ `1`	(c) `y` $\leftarrow$ `1`
(b) `r1` $\leftarrow$ `y`	(d) `r2` $\leftarrow$ `x`

`r1=?; r2=?;`

not observable:

`r1=0; r2=0`

Instrumenting writes

Consider the following program on SC:

sb

P_0	P_1
(da) $b(x) \leftarrow 1$	(dc) $b(y) \leftarrow 1$
(b) $r1 \leftarrow y$	(d) $r2 \leftarrow x$
(fa) $x \leftarrow b(x)$	(fc) $y \leftarrow b(y)$

r1=?; r2=?;

Writes access *fifo buffers, one per memory location*.

observable:

r1=0; r2=0

Instrumenting reads

Consider the following program on SC:

iriw

P_0	P_1	P_2	P_3
(a) `r1` $\leftarrow$ `x` (b) `r2` $\leftarrow$ `y`	(c) `r3` $\leftarrow$ `y` (d) `r4` $\leftarrow$ `x`	(e) `x` $\leftarrow$ `1`	(f) `y` $\leftarrow$ `2`

`r1=1; r2=0; r3=2; r4=0;`

not observable:

`r1=1; r2=0; r3=2; r4=0;`

Instrumenting reads

Consider the following program on SC:

iriw

P_0	P_1	P_2	P_3
(a) `r1` $\leftarrow$ `x`	(c) `r3` $\leftarrow$ `y`	(de) $\mathrm{b}(x) \leftarrow 1$	(df) $\mathrm{b}(y) \leftarrow 2$
(b) `r2` $\leftarrow$ `y`	(d) `r4` $\leftarrow$ `x`	(fe) `x` $\leftarrow \mathrm{b}(x)$	(ff) `y` $\leftarrow \mathrm{b}(y)$

r1=?; r2=?; r3=?; r4=?;

not observable:

r1=1; r2=0; r3=2; r4=0;

Instrumenting reads

Consider the following program on SC:

iriw

P_0	P_1	P_2	P_3
(a) $\texttt{r1} \leftarrow \mathsf{b}(x)$	(c) $\texttt{r3} \leftarrow \mathsf{b}(y)$	(de) $\mathsf{b}(x) \leftarrow 1$	(df) $\mathsf{b}(y) \leftarrow 2$
(b) $\texttt{r2} \leftarrow \texttt{y}$	(d) $\texttt{r4} \leftarrow \texttt{x}$	(fe) $\texttt{x} \leftarrow \mathsf{b}(x)$	(ff) $\texttt{y} \leftarrow \mathsf{b}(y)$

r1=?; r2=?; r3=?; r4=?;

Reads read from the buffers.

observable:

r1=1; r2=0; r3=2; r4=0;

What about a demo?

Partial-order models

Rolling up our sleeves

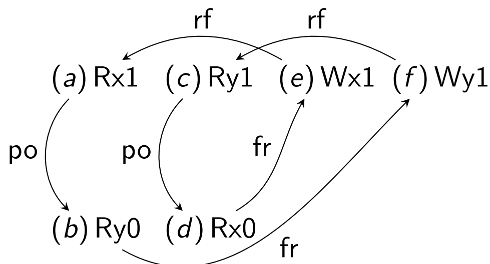
- Here we chose to build a tool that is weak memory aware by design
- We adapted CBMC (a bounded model-checking tool for C code)

Independent Reads of Independent Writes

iriw

P_0	P_1	P_2	P_3
(a) `r1` $\leftarrow$ `x`	(c) `r3` $\leftarrow$ `y`	(e) `x` $\leftarrow$ `1`	(f) `y` $\leftarrow$ `2`
(b) `r2` $\leftarrow$ `y`	(d) `r4` $\leftarrow$ `x`		

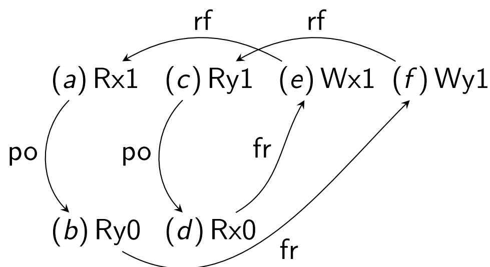
`r1`=1; `r2`=0; `r3`=2; `r4`=0;



Validity of an execution

- An execution is valid on an architecture if it does not show certain cycles.
- So we assign a clock to each event
- Then see if we can order these clocks w.r.t. less-than over $\mathbb{N}$

On iriw



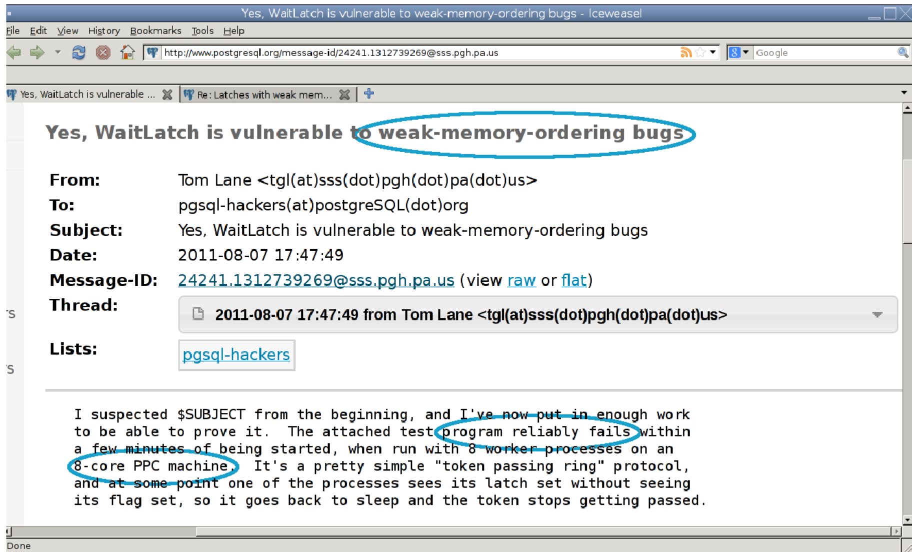
$$
\begin{array}{llll}
(\text{po } P_0) & c_{ab} & (\text{po } P_1) & c_{cd} \\
(\text{rf } x) & s_{ea} & (\text{rf } y) & s_{fc} \\
(\text{fr } x) & (s_{i_0 d} \wedge c_{i_0 e}) \Rightarrow c_{de} & (\text{fr } y) & (s_{i_1 b} \wedge c_{i_1 f}) \Rightarrow c_{bf}
\end{array}
$$

- On SC: unsatisfiable
- On Power: satisfiabe as we remove (rf x) and (rf y)

What about a demo?

A real-world example

PostgreSQL developers' discussions



Synchronisation in PostgreSQL

```
1  void worker(int i)
2   { while(!latch[i]);
3     for (;;)
4     { assert (!latch[i] || flag[i]);
5       latch[i] = 0;
6       if (flag[i])
7       { flag[i] = 0;
8         flag[(i+1)%WORKERS] = 1;
9         latch[(i+1)%WORKERS] = 1;
10      }
11      while(!latch[i]);
12    }
13  }
```

Each element of the array `latch` is a shared boolean variable dedicated to interprocess communication.

A process waits to have its latch set then should have work to do, namely passing around a token *via* the array `flag` (line 8).

Once the process is done, it sets the latch of the process the token was passed to (line 9).

Synchronisation in PostgreSQL

```
1  void worker(int i)
2   { while(!latch[i]);
3     for (;;)
4     { assert (!latch[i] || flag[i]);
5       latch[i] = 0;
6       if (flag[i])
7       { flag[i] = 0;
8         flag[(i+1)%WORKERS] = 1;
9         latch[(i+1)%WORKERS] = 1;
10      }
11      while(!latch[i]);
12    }
13  }
```

Starvation seemingly cannot occur: when a process is woken up, it has work to do.

Yet, the developers observed that the wait in line 11 would time out, i.e. starvation of the ring of processes.

The processor can delay the write in line 8 until after the latch had been set in line 9.

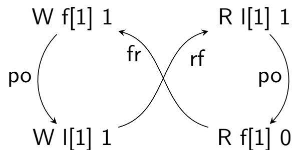
Message passing idiom in PostgreSQL

This corresponds to the message passing idiom

pgsql (mp)

Worker 0	Worker 1
(8) `f[1]=1;`	(2) `while(!l[1]);`
(9) `l[1]=1;`	(6) `if(f[1])`

Observed: `l[1]=1; f[1]=0`



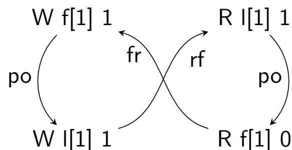
Message passing idiom in PostgreSQL

This corresponds to the message passing idiom
which requires synchronisation to behave as on SC

pgsql (mp)

Worker 0	Worker 1
(8) `f[1]=1;`	(2) `while(!l[1]);`
`lwsync`	`dependency`
(9) `l[1]=1;`	(6) `if(f[1])`

Forbidden: `l[1]=1; f[1]=0`



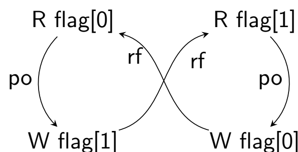
Load buffering idiom in PostgreSQL

We also found a load buffering idiom

pgsql (lb)

Worker 0	Worker 1
(6) `if(flag[0])`	(6) `if(flag[1])`
(8) `flag[1]=1;`	(8) `flag[0]=1;`

Allowed: `flag[0]=1; flag[1]=1`



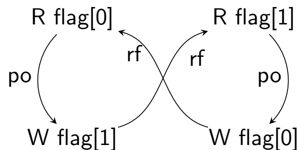
Load buffering idiom in PostgreSQL

We also found a load buffering idiom which is only a potential bug for now, since not yet implemented on Power machines

pgsql (lb)

Worker 0	Worker 1
(6) `if(flag[0])` dependency (8) `flag[1]=1;`	(6) `if(flag[1])` dependency (8) `flag[0]=1;`

Forbidden: `flag[0]=1; flag[1]=1`



Thanks!

- Instrumentation: paper at ESOP 13
  http://cprover.org/wmm
- Partial orders for BMC: paper at CAV 13
  http://cprover.org/wpo

Formally

Architectures

An architecture $A \triangleq (\text{ppo}, \text{grf}, \text{ab})$ gives us:

- the preserved program order ppo;
- the global read-from grf determines if
  - store buffering is allowed (as on x86);
  - if the stores are atomic (unlike on Power or ARM);
- the barrier semantics ab.

Machine state

A state $s \triangleq (\mathrm{m}, \mathrm{b}, \mathrm{rs})$ contains:

- the memory m: a map from addresses to writes to this address;
- the buffer b: a total order over writes per address;
- the read set rs: a set of reads.

Instrumenting writes

$$\frac{\textsc{Write to buffer}\quad \top}{s \xrightarrow{\mathsf{d}(\mathsf{w}(w))} (\mathsf{m}, \mathsf{updb}(\mathsf{b}, w), \mathsf{rs})}$$

$$\frac{\begin{array}{c}\textsc{Write from buffer to memory}\\ \mathsf{rr}(\mathsf{b}, \{e \mid (e, w) \in \mathsf{ppo} \cup \mathsf{ab}\}) = \emptyset \wedge \\ \mathsf{rs} \cap \{e \mid (e, w) \in \mathsf{ppo} \cup \mathsf{ab}\} = \emptyset \wedge \\ \mathsf{rs} \cap \{r \mid (r, w) \in \mathsf{po\text{-}loc}\} = \emptyset \wedge \\ \mathsf{last}(\mathsf{rr}(\mathsf{b}, \{e \mid \mathsf{addr}(e) = \ell\}), w)\end{array}}{s \xrightarrow{\mathsf{f}(\mathsf{w}(w))} (\mathsf{updm}(\mathsf{m}, w), \mathsf{delb}(\mathsf{b}, w), \mathsf{rs})}$$

Instrumenting reads

Delay read

$$\frac{\top}{s \xrightarrow{\mathsf{d}(\mathsf{r}(w,r))} (\mathsf{m}, \mathsf{b}, \mathsf{updrs}(\mathsf{rs}, r))}$$

Read from set

$$\frac{\begin{array}{c} r \in \mathsf{rs} \wedge \\ \mathsf{rs} \cap \{r \mid (r, w) \in \mathsf{dp}\} = \emptyset \wedge \\ \mathsf{rr}(\mathsf{b}, \{e \mid (e, r) \in \mathsf{ppo} \cup \mathsf{ab}\}) = \emptyset \wedge \\ \mathsf{rs} \cap \{e \mid (e, w) \in \mathsf{ppo} \cup \mathsf{ab}\} = \emptyset \wedge \\ \big[(w = \mathsf{m}(\mathsf{addr}(r)) \wedge \mathsf{rr}(\mathsf{b}, \{w \mid (w, r) \in \mathsf{po\text{-}loc}\}) = \emptyset) \vee \\ (w \neq \mathsf{m}(\mathsf{addr}(r)) \wedge w \in \mathsf{b} \wedge \mathsf{visible}(w, r))\big] \end{array}}{s \xrightarrow{\mathsf{f}(\mathsf{r}(w,r))} (\mathsf{m}, \mathsf{b}, \mathsf{delrs}(\mathsf{rs}, r))}$$

Visibility

A write w is *visible* to a read r, when:

- w and r share the same address ℓ;
- w is in the part of the buffer visible to r, namely if:
  - store buffering is not allowed, w cannot be on the same thread as r;
  - stores are atomic, w cannot be on a different thread from r;
- w is b-before the first write w_a to ℓ that is po-after r;
- w is equal to, or b-after, the last write w_b to ℓ that is po-before r.