Relaxed memory concurrency and verified compilation

Viktor Vafeiadis

Max Planck Institute for Software Systems (MPI-SWS)
Full functional verification

Method:

— Come up with a complete specification of the program
— Prove the program adheres to its spec

As a researcher, do functional verification when:

   Correctness important
∧ Specification possible
∧ Proof interesting

Aim: Develop “the right tools” for doing the proofs
   (program logics, abstract domains, lemmas, tactics, ...)
Compilers are *ideal* for verification

Compilers are:

- Basic computing infrastructure
- Generally reliable, but nevertheless contain many bugs
  
  e.g., Yang et al. [PLDI 2011] found 79 *gcc* & 202 *llvm* bugs
- “Specifiable”: compiler correctness = preservation of behaviours
- Interesting: naturally higher-order, involve clever algorithms
- Big, but modular
Sequential consistency (SC)

MOV \([x]\) ← 1
MOV EAX ← [y]
MOV [y] ← 1
MOV EBX ← [x]

Thread
... Thread

Shared Memory

— Thread actions are interleaved
— Does not correspond to modern hardware
x86 concurrency

MOV \([x]\) ← 1
MOV EAX ← \([y]\)
MOV \([y]\) ← 1
MOV EBX ← \([x]\)

— Can return \(EAX = 0\) and \(EBX = 0\)

— Interleaving insufficient: “store buffering” (TSO memory model)
Store buffering

\[
\begin{align*}
\text{MOV } [x] & \leftarrow 1 \\
\text{MOV } \text{EAX} & \leftarrow [y] \\
\text{MOV } \text{EBX} & \leftarrow [x]
\end{align*}
\]

x : 0   y : 0

EAX : 32

... EBX : 47
Store buffering

\[
\begin{align*}
\text{MOV} [x] & \leftarrow 1 \\
\text{MOV} \ EAX & \leftarrow [y] \\
\text{MOV} [y] & \leftarrow 1 \\
\text{MOV} \ EBX & \leftarrow [x]
\end{align*}
\]

EAX : 32

\[
\begin{align*}
x : 1
\end{align*}
\]

Thread

Write Buffer

Shared Memory

\[
\begin{align*}
x : 0 \\
y : 0
\end{align*}
\]

EBX : 47
Store buffering

MOV [x] ← 1
MOV EAX ← [y]
MOV [y] ← 1
MOV EBX ← [x]

EAX : 32
Thread
Write Buffer
x : 1

... Thread
Write Buffer
y : 1

Shared Memory
x : 0  y : 0

EBX : 47
Store buffering

MOV [x] ← 1
MOV EAX ← [y]

MOV [y] ← 1
MOV EBX ← [x]

Thread
Write Buffer
x:1

Thread
Write Buffer
y:1

Shared Memory

EAX : 0
EBX : 47

x : 0  y : 0
Store buffering

MOV [x] ← 1
MOV EAX ← [y]
MOV [y] ← 1
MOV EBX ← [x]

EAX : 0

Thread

Write Buffer

x:1

Thread

Write Buffer

y:1

Shared Memory

x : 0  y : 0
Store buffering

MOV [x] ← 1
MOV EAX ← [y]

MOV [y] ← 1
MOV EBX ← [x]

Thread
Write Buffer

EAX : 0

Thread
Write Buffer

y : 1

x : 1 y : 0

Shared Memory
Store buffering

MOV [x] ← 1
MOV EAX ← [y]
MOV [y] ← 1
MOV EBX ← [x]

Thread

Write Buffer

EAX : 0

Thread

Write Buffer

EBX : 0

Shared Memory

x : 1

y : 1
An alternative explanation: Load prefetching

MOV [x] ← 1
MOV EAX ← [y]

MOV [y] ← 1
MOV EBX ← [x]

EAX : 32
... EBX : 47

x : 0  y : 0
An alternative explanation: Load prefetching

MOV [x] ← 1
MOV EAX ← [y]

MOV [y] ← 1
MOV EBX ← [x]

EAX : 32

Thread

Prefetch Buffer

y : 0

Thread

Prefetch Buffer

EBX : 47

x : 0  y : 0
An alternative explanation: Load prefetching

```
MOV [x] ← 1
MOV EAX ← [y]
MOV [y] ← 1
MOV EBX ← [x]
```

EAX : 32

Thread

y:0
Prefetch Buffer

Thread

x:0
Prefetch Buffer

x : 0   y : 0

Shared Memory

EBX : 47
An alternative explanation: Load prefetching

MOV [x] ← 1
MOV EAX ← [y]
MOV [y] ← 1
MOV EBX ← [x]

Thread
y:0
Prefetch Buffer

Thread
x:0
Prefetch Buffer

Shared Memory

EAX : 32
EBX : 47

x : 1   y : 0
An alternative explanation: Load prefetching

```
MOV [x] ← 1
MOV EAX ← [y]
MOV [y] ← 1
MOV EBX ← [x]
```

Thread

EAX : 0

Prefetch Buffer

Shared Memory

x : 1  y : 0

Thread

EBX : 47

Prefetch Buffer
An alternative explanation: Load prefetching

MOV [x] ← 1
MOV EAX ← [y]

MOV [y] ← 1
MOV EBX ← [x]

EAX : 0

Thread

Prefetch Buffer

Shared Memory

x : 1
y : 1

EBX : 47

Thread

Prefetch Buffer

x : 0
An alternative explanation: Load prefetching

MOV [x] ← 1
MOV EAX ← [y]

MOV [y] ← 1
MOV EBX ← [x]

EAX : 0

Thread

Prefetch Buffer

Shared Memory

x : 1  y : 1

Thread

Prefetch Buffer

EBX : 0
Fence instructions

\[
\begin{align*}
\text{MOV } [x] & \leftarrow 1 \\
\text{MFENCE} & \\
\text{MOV } EAX & \leftarrow [y] \\
\end{align*}
\]

In the store buffer model,
“block until the \textit{local} buffer is empty”

In the prefetch model,
“block if the \textit{local} prefetch buffer is non-empty”
or “clear the local prefetch buffer”
Store buffering + fences

MOV [x] ← 1
MFENCE
MOV EAX ← [y]

MOV [y] ← 1
MFENCE
MOV EBX ← [x]

EAX : 32
Thread
Write Buffer
Shared Memory
x : 0

... 
Thread
Write Buffer
EBX : 47
Store buffering + fences

```
MOV [x] ← 1
MFENCE
MOV EAX ← [y]

MOV [y] ← 1
MFENCE
MOV EBX ← [x]
```
Store buffering + fences

MOV \[x\] ← 1
MFENCE
MOV EAX ← \[y\]

MOV \[y\] ← 1
MFENCE
MOV EBX ← \[x\]

EAX : 32
Thread
Write Buffer
x : 1

... Thread
Write Buffer

y : 1

EBX : 47
Shared Memory

x : 0  y : 0
Store buffering + fences

**Shared Memory**

**Thread**

MOV [x] ← 1

MFENCE

MOV EAX ← [y]

**Write Buffer**

y:1

**Thread**

MOV [y] ← 1

MFENCE

MOV EBX ← [x]

**Write Buffer**

x:1 y:0

**MFENCE** blocks until the thread buffer is empty
C++11 concurrency

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

Semantics depends on the type of x, y.

- ordinary int* => undefined semantics
- atomic_int* => SC semantics

(There are also weaker kinds of atomics.)

The compiler is responsible for adding the necessary FENCEs.
Compiling C++11 ordinary accesses

To compile ordinary int* accesses, no fences are needed on x86:

\[
\begin{align*}
*x &= 1; \\
a &= *y;
\end{align*}
\]

Reordering of ordinary memory accesses permitted.

Why is this sound?
Compiling C++11 atomic accesses

Recipe for compiling `atomic_int*` accesses on x86:

Load: `MFENCE; MOV`  
Store: `MOV; MFENCE`

In our example:

- \*x = 1;  
  Compile na"ively
  
- a = \*y;  
  Optimize

```c
MOV [x] ← 1  
MFENCE  
MOV EAX ← [y]
```
Compiler correctness

What does it mean for a compiler to be correct?

source program (e.g., C) \(\approx\) target program (e.g., x86)

What properties should “\(\approx\)” have?

Should it be reflexive? Symmetric? Transitive? Anything else?
Reflexivity & symmetry

— Sensible only if compiling to the same language

— If so, Reflexivity (doing nothing is a valid optimisation)

Symmetry To see why:

fail ⇒ print “hello”

print “hello” ⇒ fail
Example 1: Compiling C++11 ordinary accesses

Compilation of ordinary memory accesses:

*\*x = 1;*
*\*y = 2;*

This is sound because:

— Either \( C \) does not access \( *x \) and \( *y \)
  \( \Rightarrow \) same behaviour

— Or \( C \) accesses \( *x \) or \( *y \)
  \( \Rightarrow \) race condition
  \( \Rightarrow \) LHS has undefined semantics

  [NB: RHS semantics are well-defined \( \neq \) LHS semantics]
Example 2: Reordering C++11 ordinary accesses

Recall that for ordinary accesses may be reordered:

```
x = 1;
y = 2;
```

This is sound because:

- Either \( C \) does not access \(*x\) and \(*y\)
  => same behaviour

- Or \( C \) accesses \(*x\) or \(*y\)
  => race condition
  => LHS has undefined semantics

```cpp
*y = 2;
x = 1;
```
Correctness notion *should be* transitive

- Compiler = sequence of program transformations

- Want to verify each phase independently.
Correctness notion *should be* compositional (ideally)

— Separate compilation & linking:

<table>
<thead>
<tr>
<th></th>
<th>CompilerA</th>
<th></th>
<th>CompilerB</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>module_a.c</td>
<td>module_a.o</td>
<td></td>
<td>module_b.c</td>
<td>module_b.o</td>
</tr>
<tr>
<td>module_b.c</td>
<td>module_b.o</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

— We want the correctness notion to reflect this picture (Difficult!) [Ongoing work with Dreyer, Hur, Neis]

— Here, we’ll ignore the issue.
Compiler correctness as trace inclusion

\[
\text{traces(source\_program)} \supseteq \text{traces(target\_program)}
\]

- print “a” || print “b”  →  print “a” ; print “b”
- print “a” ; print “b”  →  print “a” || print “b”
- fail  →  print “hello”
- print “hello”  →  fail
Basic proof technique: simulations

Goal to prove:

By coinduction: find a “simulation” relation \( \sim \) such that:

\[ \forall t' \implies \exists s' \]

and

\[ \forall t' \]\n
compile \subseteq
CompCertTSO

— Take Leroy’s CompCert
— Generate x86 instead of PowerPC/ARM
— Add concurrency (TSO relaxed memory model)
— Remove unsound compiler optimisations (restrict CSE)
— Prove the compiler correct w.r.t. TSO semantics (reusing Leroy’s proofs as much as possible)
— Implement & verify TSO-specific optimisations
CompCertTSO

ClightTSO
  simplify
  local vars
  simplify
  instruction_selection
  CFG generation

C#minor

Cstacked

Cminor

CminorSel

RTL
  const prop.

RTL
  CSE

RTL

LTL
  branch tunnelling

LTL
  linearize
  register allocation

LTLin
  reload/spill
  act.records

Linear

Machabstr

Machconc

x86

[POPL 2011]
CompCertTSO + fence optimisations

- ClightTSO
- ClightTSO
- const prop.
- simplify
- C#minor
- C#minor
- CSE
- local vars
- Cstacked
- Cstacked
- FE1
- simplify
- Cminor
- Cminor
- PRE
- instruction selection
- CminorSel
- CminorSel
- FE2
- CFG generation
- register allocation
- RTL
- RTL
- Machabstr
- Machabstr
- branch tunnelling
- reload/spill
- LTL
- LTL
- Linear
- Linear
- act.records
- Linear
- Machconc
- Machconc
- x86
- LTLin
- LTLin
- linearize
Redundant fences (1)

If we have two consecutive fence instructions, we can remove the *latter*:

```
MFENCE
MFENCE
```

The *buffer is already empty* when the second fence is executed.

**Generalisation:**

```
MFENCE
NON-WRITE INSTR ...
NON-WRITE INSTR MFENCE
```

```
MFENCE
NON-WRITE INSTR ...
NON-WRITE INSTR NOP
```
If we have two consecutive fence instructions, we can remove the former:

\[
\text{MFENCE} \quad \text{MFENCE} \quad \text{NOP} \quad \text{MFENCE}
\]

*Intuition:* the visible effects initially published by the former fence, are now published by the latter, and nobody can tell the difference.

*Generalisation:*

\[
\text{MFENCE} \\
\text{INSTRUCTION 1} \\
\text{...} \\
\text{INSTRUCTION } n \\
\text{MFENCE} \\
\]

\[
\text{NOP} \\
\text{INSTRUCTION 1} \\
\text{...} \\
\text{INSTRUCTION } n \\
\text{MFENCE}
\]
Redundant fences (2)

If there are reads in between the fences...

<table>
<thead>
<tr>
<th>[x]=[y]=0</th>
<th>Thread 0</th>
<th>Thread 1</th>
<th>EAX = EBX = 0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MOV [x] ← 1</td>
<td>MOV [y] ← 1</td>
<td>forbidden</td>
</tr>
<tr>
<td></td>
<td>MFENCE</td>
<td>MFENCE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MOV EAX ← [y]</td>
<td>MOV EBX ← [x]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MFENCE</td>
<td>MFENCE</td>
<td></td>
</tr>
</tbody>
</table>

but

<table>
<thead>
<tr>
<th>[x]=[y]=0</th>
<th>Thread 0</th>
<th>Thread 1</th>
<th>EAX = EBX = 0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MOV [x] ← 1</td>
<td>MOV [y] ← 1</td>
<td>allowed</td>
</tr>
<tr>
<td></td>
<td>NOP</td>
<td>MFENCE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MOV EAX ← [y]</td>
<td>MOV EBX ← [x]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MFENCE</td>
<td>MFENCE</td>
<td></td>
</tr>
</tbody>
</table>
Redundant fences (2)

If there are reads in between the fences...

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV [x] ← 1</td>
<td>MOV [y] ← 1</td>
</tr>
<tr>
<td>MFENCE</td>
<td>MFENCE</td>
</tr>
<tr>
<td>MOV EAX ← [y]</td>
<td>MOV EBX ← [x]</td>
</tr>
<tr>
<td>MFENCE</td>
<td>MFENCE</td>
</tr>
</tbody>
</table>

If there are reads in between, the optimisation is unsound.

[x] = [y] = 0

EAX = EBX = 0

but

[x] = [y] = 0

EAX = EBX = 0

allowed
Redundant fences (2)

Swapping a `STORE` and a `MFENCE` is sound:

```
MFENCE; STORE  →  STORE; MFENCE
```

1. transformed program’s behaviours $\subseteq$ source program’s behaviours

   (source program might leave pending write in its buffer)

2. There is the new intermediate state if the buffer was initially non-empty, but this intermediate state *is not observable.*

   (a local read is needed to access the local buffer)

*Intuition:* Iterate this swapping...
Informal correctness argument

**Intuition:** FE2 can be thought as iterating

\[
\begin{align*}
\text{MFENCE; STORE} & \quad \rightarrow \quad \text{STORE; MFENCE} \\
\text{MFENCE; non-mem} & \quad \rightarrow \quad \text{non-mem; MFENCE}
\end{align*}
\]

and then applying

\[
\text{MFENCE; MFENCE} \quad \rightarrow \quad \text{NOP; MFENCE}
\]

This argument works for *finite traces*, but not for *infinite traces* as the later fence might never be executed:

\[
\begin{align*}
\text{MFENCE;} & \quad \rightarrow \quad \text{NOP;} \\
\text{STORE;} & \quad \rightarrow \quad \text{STORE;} \\
\text{WHILE(1);} & \quad \rightarrow \quad \text{WHILE(1);} \\
\text{MFENCE} & \quad \rightarrow \quad \text{MFENCE}
\end{align*}
\]
A closer look at the RTL

Patterns like that on the left are common.
FE1 and FE2 do not optimise these patterns.
It would be nice to hoist those fences out of the loop.
A closer look at the RTL

Patterns like that on the left are common.

FE1 and FE2 do not optimise these patterns.

It would be nice to hoist those fences out of the loop.

Do you perform PRE?
A closer look at the RTL

Patterns like that on the left are common.

FE1 and FE2 do not optimise these patterns.

It would be nice to hoist those fences out of the loop.

Do you perform PRE?

...adding a fence is always safe...
Partial redundancy elimination

PRE

FE2
Towards a verified compiler from C++11 to x86

Two options:

— Add a new front-end phase: Clight++11 to ClightTSO

  “Easy, but useless”

  (straightforward to implement, but cannot perform optimisations allowed under C++11 but not TSO)

— Propagate the C++ memory model throughout.

  Convert to TSO at the final phase.

  “Done right, but more (short-term) work”
How much more work?

CompCertTSO phases affect memory behaviour in rather simple ways:

1. Reduce non-determinism of values written to memory

2. Merge allocation blocks
   (i.e. allocate one big chunk instead of many smaller ones)

3. Insert/remove thread-local memory accesses (with SC semantics)

4. Remove unused reads

5. Insert/remove redundant fences

NB: Except for 5(b), the transformations are memory model agnostic. Exploit this!
Summary

- What are relaxed memory models
- Compiling from C++11 to x86-TSO
- What does it mean for a compiler to be correct
- Fence elimination optimisations for TSO
- Plan for a C++11 to x86-TSO verified compiler