Blowing up the (C++11) atomic barrier

Optimizing C++11 atomics in LLVM

Robin Morisset, Intern at Google
Background: C++11 atomics

Optimizing around atomics

Fence elimination

Miscellaneous optimizations

Further work: Problems with atomics
Background: C++11 atomics

Optimizing around atomics

Fence elimination

Miscellaneous optimizations

Further work: Problems with atomics
Can this possibly print 0-0 ?

Thread 1

\[
x \leftarrow 1;
\]
\[
\text{print } y;
\]

Thread 2

\[
y \leftarrow 1;
\]
\[
\text{print } x;
\]
Can this possibly print 0-0?

Yes if your compiler reorder accesses

Thread 1

```print y;
x <- 1;```

Thread 2

```print x;
y <- 1;```
Can this possibly print 0-0?

Yes on x86: needs a fence

Flush your (FIFO) store buffer

```
x <- 1;
mfence;
print y;
y <- 1;
mfence;
print x;
```
Can this possibly print 0?

```r
x <- 42;
ready <- 1;
if (ready)
  print x;
```
Can this possibly print 0?

Yes on ARM

Flush your (non-FIFO) store buffer.

```r
x <- 42;
dmb ish;
ready <- 1;
if (ready)
  print(x);
```
Can this possibly print 0?
Yes on ARM: needs 2 fences to prevent

Flush your (non-FIFO) store buffer

\[ x \leftarrow 42; \]
\[ \text{dmb ish;} \]
\[ \text{ready} \leftarrow 1; \]

Don’t speculate reads across

\[ \text{if (ready)} \]
\[ \quad \text{dmb ish;} \]
\[ \quad \text{print } x; \]
Doing it portably

C11/C++11 memory model

- data race (dynamic) = undefined
- no data race (using mutexes) = intuitive behavior (“Sequentially consistent”)
- for lock-free code: atomic accesses
Sequentially consistent

```python
x.store(1, seq_cst);
print(y.load(seq_cst));
```

```python
y.store(1, seq_cst);
print(x.load(seq_cst));
```
Release/acquire

\[ x = 42; \]
\[ \text{ready}.\text{store}(1, \text{release}); \]

\[ \text{if (ready}.\text{load(\text{acquire})}) \]
\[ \text{print}(x); \]
Release/acquire

```python
x = 42;
ready.store(1, release);
if (ready.load(acquire))
    print(x);
```
\[ x = 42; \]
\[ \text{ready}.\text{store}(1, \text{release}); \]

\[ \text{if (ready.load(} \text{acquire}) \text{)} \]
\[ \text{print}(x); \]
Background: C++11 atomics

Optimizing around atomics

Fence elimination

Miscellaneous optimizations

Further work: Problems with atomics
Compiler optimizations?

void foo(int *x, int n) {
    for(int i=0; i<n; ++i){
        *x *= 42;
    }
}

LICM

void foo(int *x, int n) {
    int tmp = *x;
    for(int i=0; i < n; ++i){
        tmp *= 42;
    }
    *x = tmp;
}
Compiler optimizations?

```c
void foo(int *x, int n) {
    int tmp = *x;
    *x = tmp;
}
```

LICM
Compiler optimizations?

void foo(int *x, int n) {
    int tmp = *x;
    *x = tmp;
}

++(*x); // in another thread...
Never introduce a store where there was none
Dead store elimination?

\[ x = 42; \]

\[ \ldots \]

\[ x = 43; \]
Dead store elimination?

\begin{verbatim}
x = 42;
flag1.store(true, release);
while (!flag2.load(acquire))
    continue;
x = 43;
\end{verbatim}
Dead store elimination?

```c
x = 42;
flag1.store(true, release);
while (!flag2.load(acquire))
    continue;
x = 43;
```

```c
while (!flag1.load(acquire))
    continue;
print(x);
flag2.store(true, release);
```
Dead store elimination?

```c
x = 42;
while (!flag2.load(acquire))
    continue;
x = 43;

Race!

print(x);
flag2.store(true, release);
```
Dead store elimination?

```c
x = 42;
flag1.store(true, release);

while (!flag1.load(acquire))
    continue;
print(x);

x = 43;
Race!
```
Anything can happen to memory between a release and an acquire.
Background: C++11 atomics

Optimizing around atomics

Fence elimination

Miscellaneous optimizations

Further work: Problems with atomics
Fence elimination

int t = y.load(acquire);
...
x.store(1, release);

ldr r0, [r0]
dmb ish
...
dmb ish
str r2, [r1]
ldr ...
dmb ish

str ...

dmb ish
str ...

2 fences on main path
1 fence on main path
1 fence on main path
Build graph from CFG
Build graph from CFG
Identify sources/sinks
Build graph from CFG
Identify sources/sinks
Build graph from CFG
Identify sources/sinks
Annotate with frequency

Source

Sink

ldr ...
str ...
str ...

2
∞
2
∞

5
∞
∞
5
∞
Build graph from CFG
Identify sources/sinks
Annotate with frequency
Find min-cut

2 + 5 = 7 is minimum

Source

Sink
Build graph from CFG
Identify sources/sinks
Annotate with frequency
Find min-cut
Move fences
while(flag.load(acquire))
{
}

.loop:
    ldr r0, [r1]
    dmb ish
    bnz .loop
while(flag.load(acquire))
{
}

.loop:
  ldr r0, [r1]
  bnz .loop
  dmb ish
.loop:
    ldr r0, [r1]
    dmb ish
    bnz .loop

memory access
.loop:
    ldr r0, [r1]
    bnz .loop

...  
    dmb ish
    memory access
Background: C++11 atomics
Optimizing around atomics
Fence elimination
Miscellaneous optimizations
Further work: Problems with atomics
x.load(release) ?
x.load(\texttt{release}) ?

\texttt{x.fetch\_add(0, release)}
x.load(release) ?

x.fetch_add(0, release)

x86

mov %eax, $0
lock
xadd (%ebx), %eax
x.load(release) ?

x.fetch_add(0, release)

x86

mov %eax, $0
lock
xadd (%ebx), %eax

mfence
mov %eax, (%ebx)

7200% speedup for a seqlock*
```
x.store(0, \textit{release})
```

```
x.load(\textit{acquire})
```

```
\begin{array}{l}
\text{Power} \\
hw sync \\
\text{str} ...
\end{array}
```

```
\begin{array}{l}
\text{ARM} \\
dmb sy \\
\text{str} ...
\end{array}
```

```
\begin{array}{l}
\text{Power} \\
lwz ...
\end{array}
```

```
\begin{array}{l}
\text{ARM} \\
ldr ...
\end{array}
```

```
\begin{array}{l}
\text{ARM} \\
dmb sy
\end{array}
```
x.store(0, release)

Power

lwsync
stw ...

ARM

dmb ish
str ...

x.load(acquire)

lwsync
lwz ...

ldr ...

lwsync
dmb ish
Power

lwsync
stw ...

ARM (Swift)

dmb ishst
str ...

x.store(0, release)

x.load(acquire)

lwsync
lwz ...
lwz sync

ldr ...
dmb ish
Power

x.store(2, relaxed)

rlwinm r2, r3, 3, 27, 28
li r4, 2
xori r5, r2, 24
rlwinm r2, r3, 0, 0, 29
li r3, 255
slw r4, r4, r5
slw r3, r3, r5
and r4, r4, r3

LBB4_1:
  lwarx r5, 0, r2
  andc r5, r5, r3
  or r5, r4, r5
  stwcx. r5, 0, r2
  bne cr0, LBB4_1
Power

Shuffling

\[ x.\text{store}(2, \text{relaxed}) \]

```
rlwinm r2, r3, 3, 27, 28
li r4, 2
xori r5, r2, 24
rlwinm r2, r3, 0, 0, 29
li r3, 255
slw r4, r4, r5
slw r3, r3, r5
and r4, r4, r3
LBB4_1:
    lwarx r5, 0, r2
    andc r5, r5, r3
    or r5, r4, r5
    stwcx. r5, 0, r2
    bne cr0, LBB4_1
```
Power

Shuffling

x.store(2, relaxed)

Loop

rlwinm r2, r3, 3, 27, 28
li r4, 2
xori r5, r2, 24
rlwinm r2, r3, 0, 0, 29
li r3, 255
slw r4, r4, r5
slw r3, r3, r5
and r4, r4, r3
LBB4_1:
lwarx r5, 0, r2
andc r5, r5, r3
or r5, r4, r5
stwcx. r5, 0, r2
bne cr0, LBB4_1
```assembly
rlwinm r2, r3, 3, 27, 28
li r4, 2
xor r5, r2, 24
rlwinm r2, r3, 0, 0, 29
li r3, 255
slw r4, r4, r5
slw r3, r3, r5
and r4, r4, r3

LBB4_1:
  lwarx r5, 0, r2
  andc r5, r5, r3
  or r5, r4, r5
  stwcx. r5, 0, r2
  bne cr0, LBB4_1
```
\texttt{x.store(2, relaxed)}

\texttt{li r2, 2}
\texttt{stb r2, 0(r3)}
x.store(2, relaxed)

x86

mov %eax, $2
mov (%ebx), %eax
x.store(2, relaxed)

mov (%ebx), $2
Background: C++11 atomics

Optimizing around atomics

Fence elimination

Miscellaneous optimizations

Further work: Problems with atomics
Relaxed attribute

print(y.load(\texttt{relaxed}));
x.store(1, \texttt{relaxed});

print(x.load(\texttt{relaxed}));
y.store(1, \texttt{relaxed});
Relaxed attribute

Can print 1-1
Relaxed attribute

\[
t_y = y.\text{load(}\text{relaxed}\text{)};
\]
\[
x.\text{store}(t_y, \text{relaxed});
\]
\[
t_x = x.\text{load(}\text{relaxed}\text{)};
\]
\[
y.\text{store}(t_x, \text{relaxed});
\]

\[x = y = ???\]
Relaxed attribute

```plaintext
if(y.load(relaxed))
    x.store(1, relaxed);
    print("foo");

if(x.load(relaxed))
    y.store(1, relaxed);
    print("bar");

Can print foobar!
```
Consume attribute

\[
\begin{align*}
  *x &= 42; \\
  x &.\text{store}(1, \text{release}); \\
  t &= x.\text{load(\text{acquire})}; \\
  \text{print}(*t);
\end{align*}
\]
Consume attribute

*x* = 42;

\[
x.\text{store}(1, \text{release});
\]

\[
t = x.\text{load(consume)};
\]

\[
\text{print(}*t*);
\]
Consume attribute

```plaintext
*x = 42;
\text{\texttt{x.store(1, release);}}

\text{\texttt{t = x.load(consume);}}
\text{\texttt{print(*y);}}
```

Unordered !
Consume attribute

\[
x = 42;
x.\text{store}(1, \text{release});
\]

\[
t = x.\text{load(consume)};
\]

\[
\text{print}(*\(y + t - t\));
\]
Conclusion

- Atomics = portable lock-free code in C11/C++11
- Tricky to compile, but can be done
- Lots of open questions
Questions ?