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 <- 1;
print y;
```

Thread 2

```
y <- 1;
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

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

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

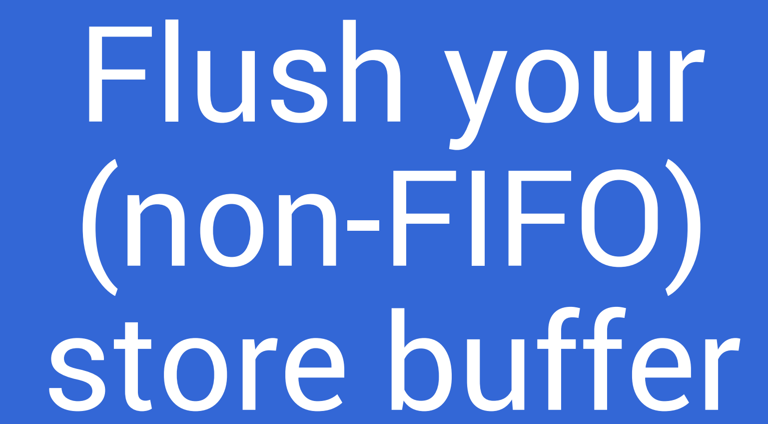
Can this possibly print 0 ?

```
x <- 42;
ready <- 1;
```

```
if (ready)
    print x;
```

Can this possibly print 0 ?

Yes on ARM




```
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 <- 42;
dmb ish;
ready <- 1;
```

```
if (ready)
    dmb ish;
    print x;
```

Don’t speculate reads across

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

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

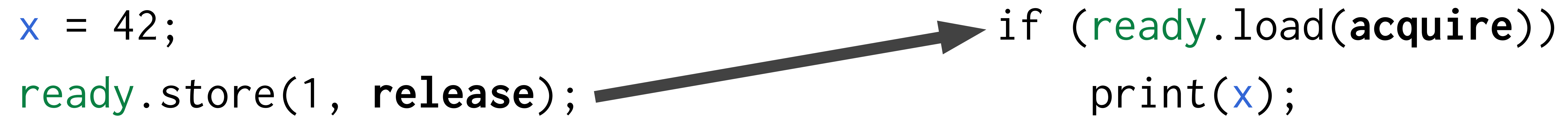
```
y.store(1, seq_cst);
print(x.load(seq_cst));
```

Release/acquire

```
x = 42;
ready.store(1, release);
```

```
if (ready.load(acquire))
    print(x);
```

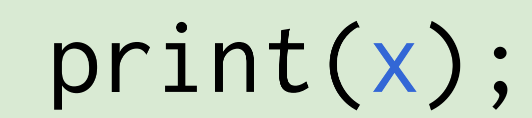
Release/acquire



Release/acquire

```
x = 42;
ready.store(1, release);
```

```
if (ready.load(acquire))
    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 ?

```
void foo(int *x, int n) {







}
```

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




  *x = tmp;
}
```

Compiler optimizations ?

```
void foo(int *x, int n) {




}
```

LICM

```
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 ?

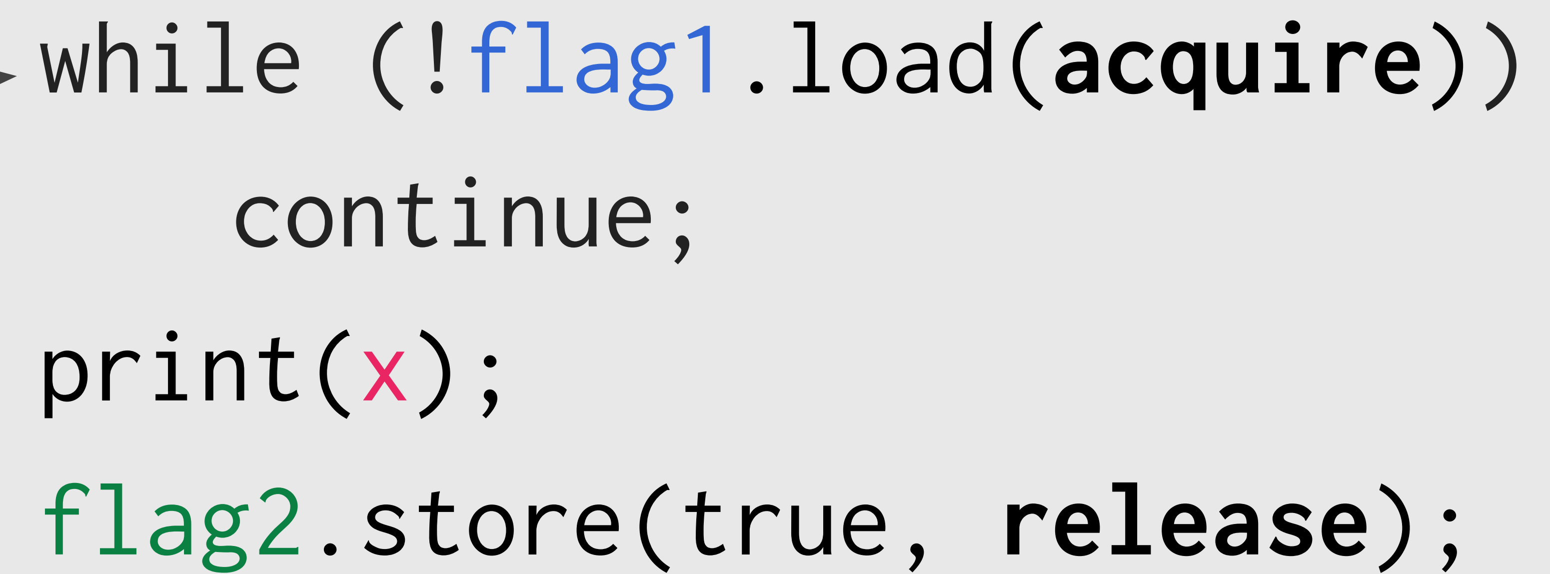
```
x = 42;

…

x = 43;
```

Dead store elimination ?

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

Dead store elimination ?

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



Dead store elimination ?

```
x = 42;

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

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

Race !

Dead store elimination ?

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


x = 43;
```

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

Race !

Anything can happen to memory between a release and an acquire

Background: C++11 atomics

Optimizing around atomics

Fence elimination

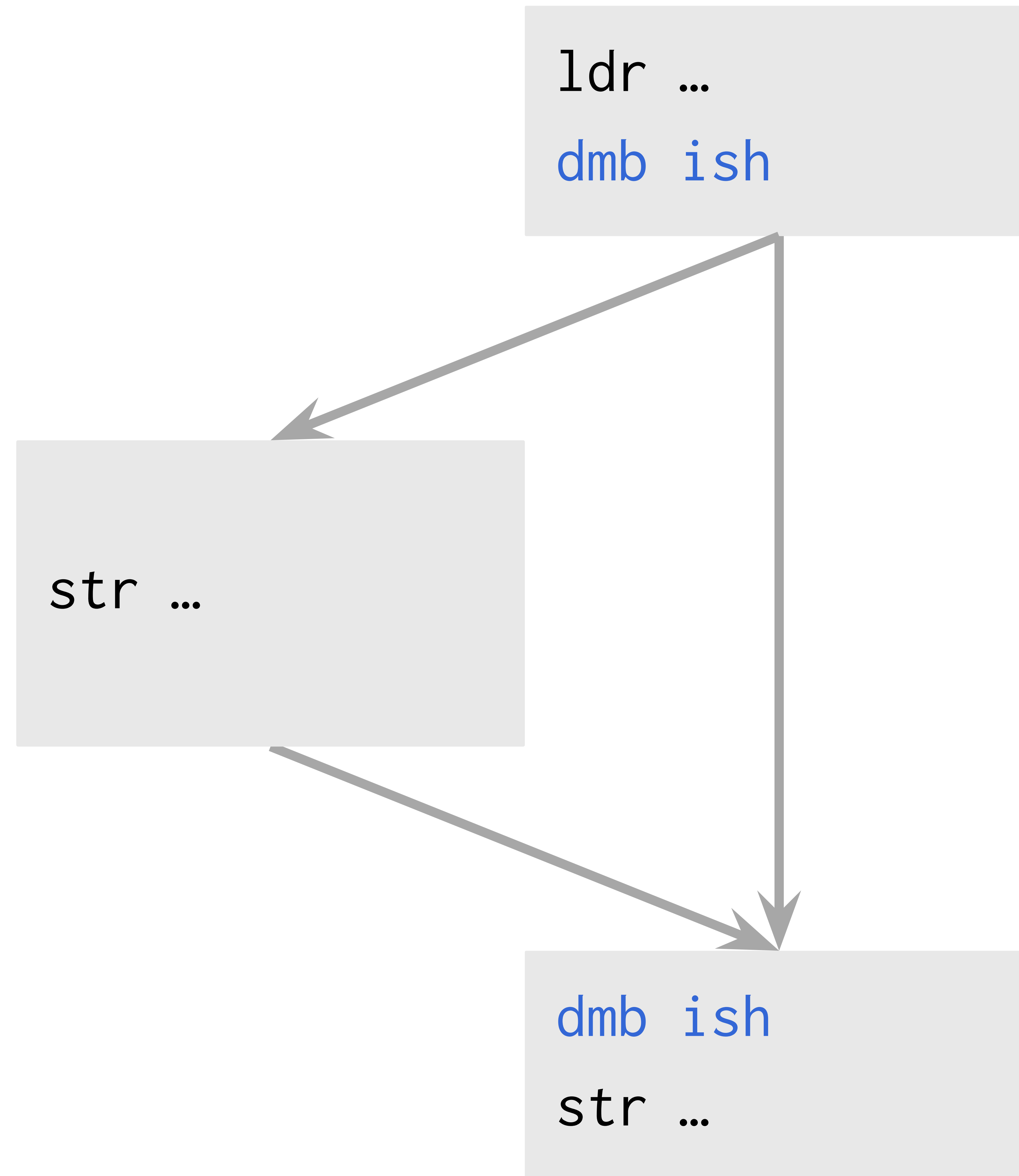
Miscellaneous optimizations

Further work: Problems with atomics

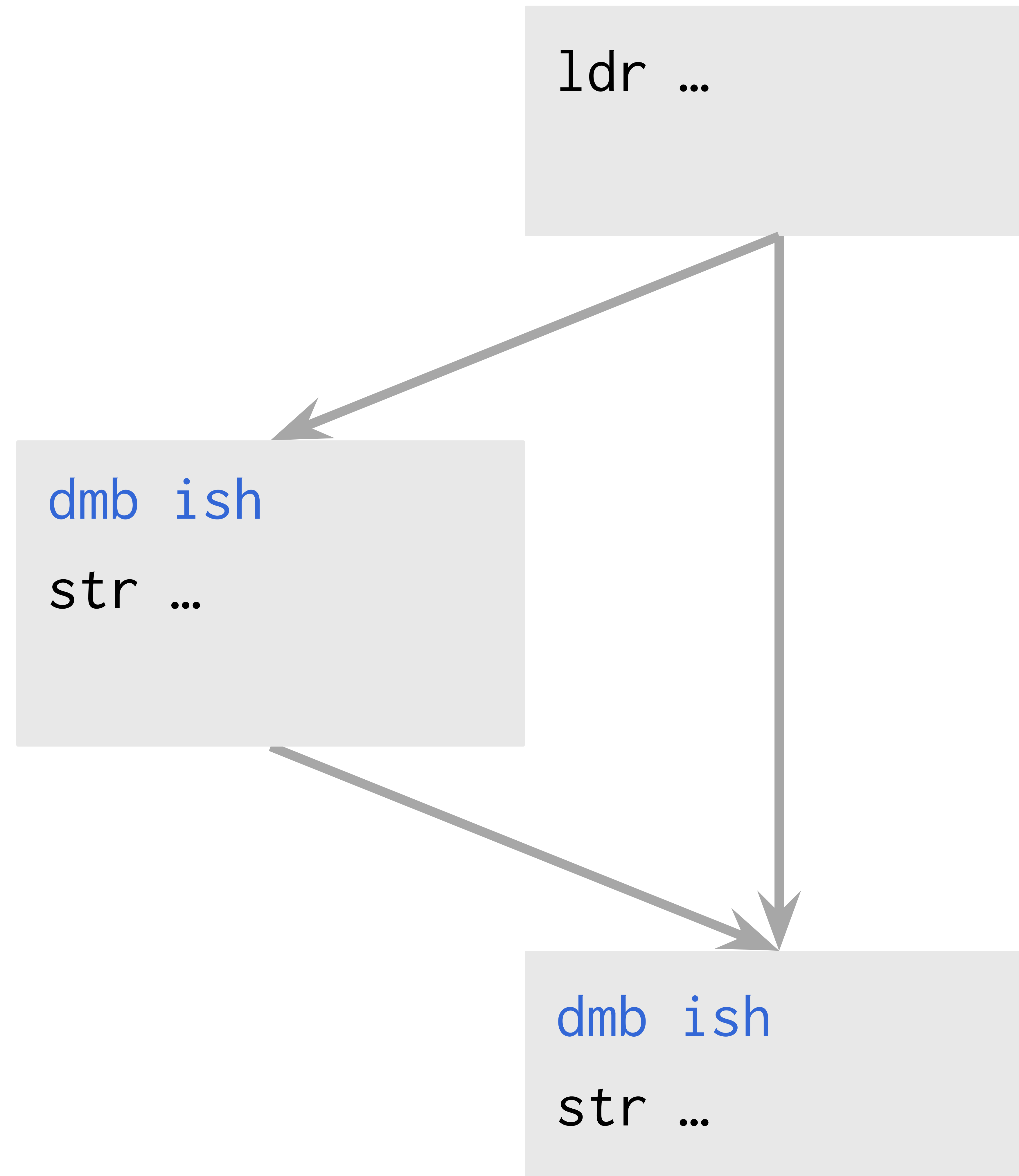
Fence elimination

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

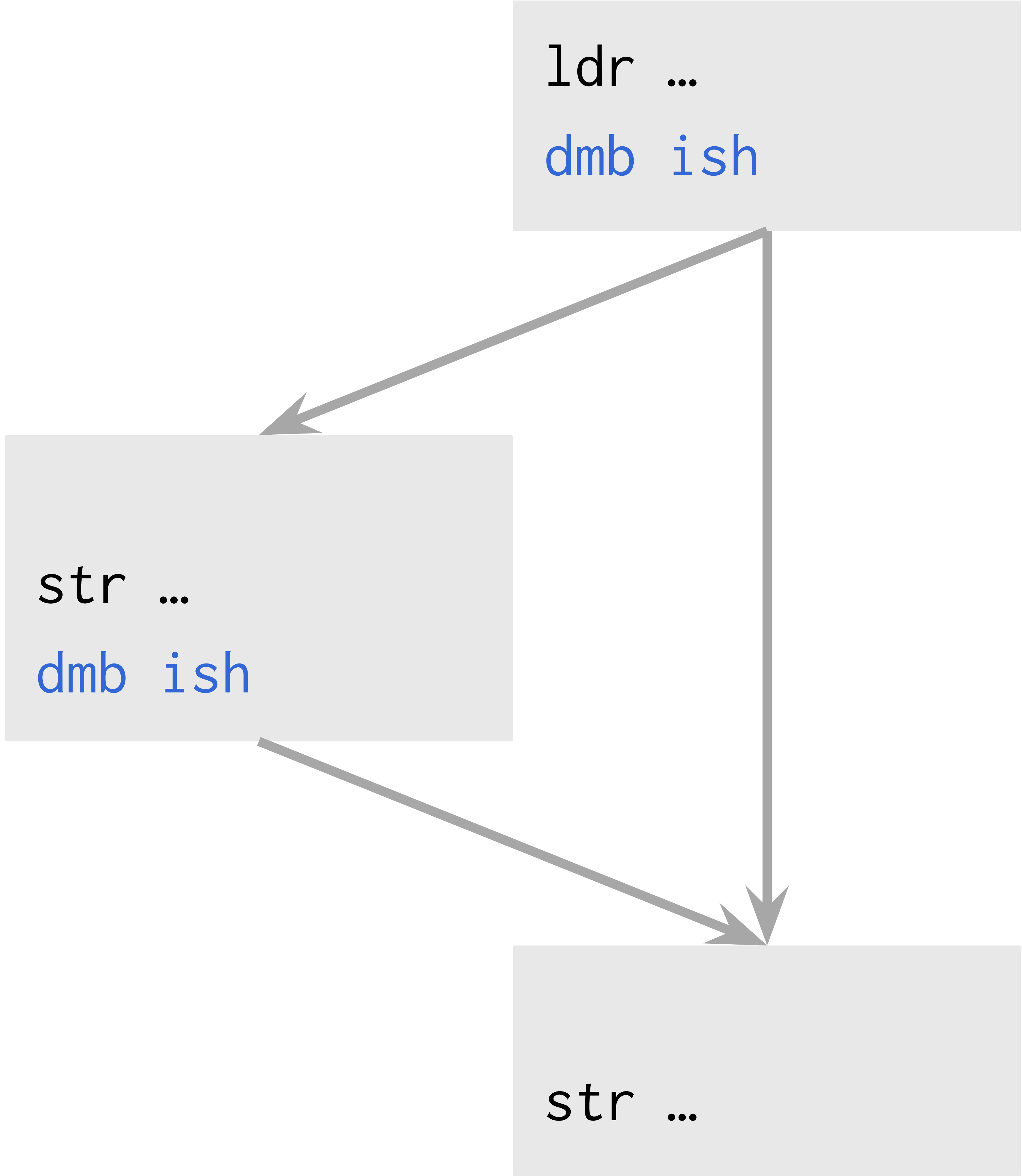
```
ldr r0, [r0]
dmb ish
…
dmb ish
str r2, [r1]
```



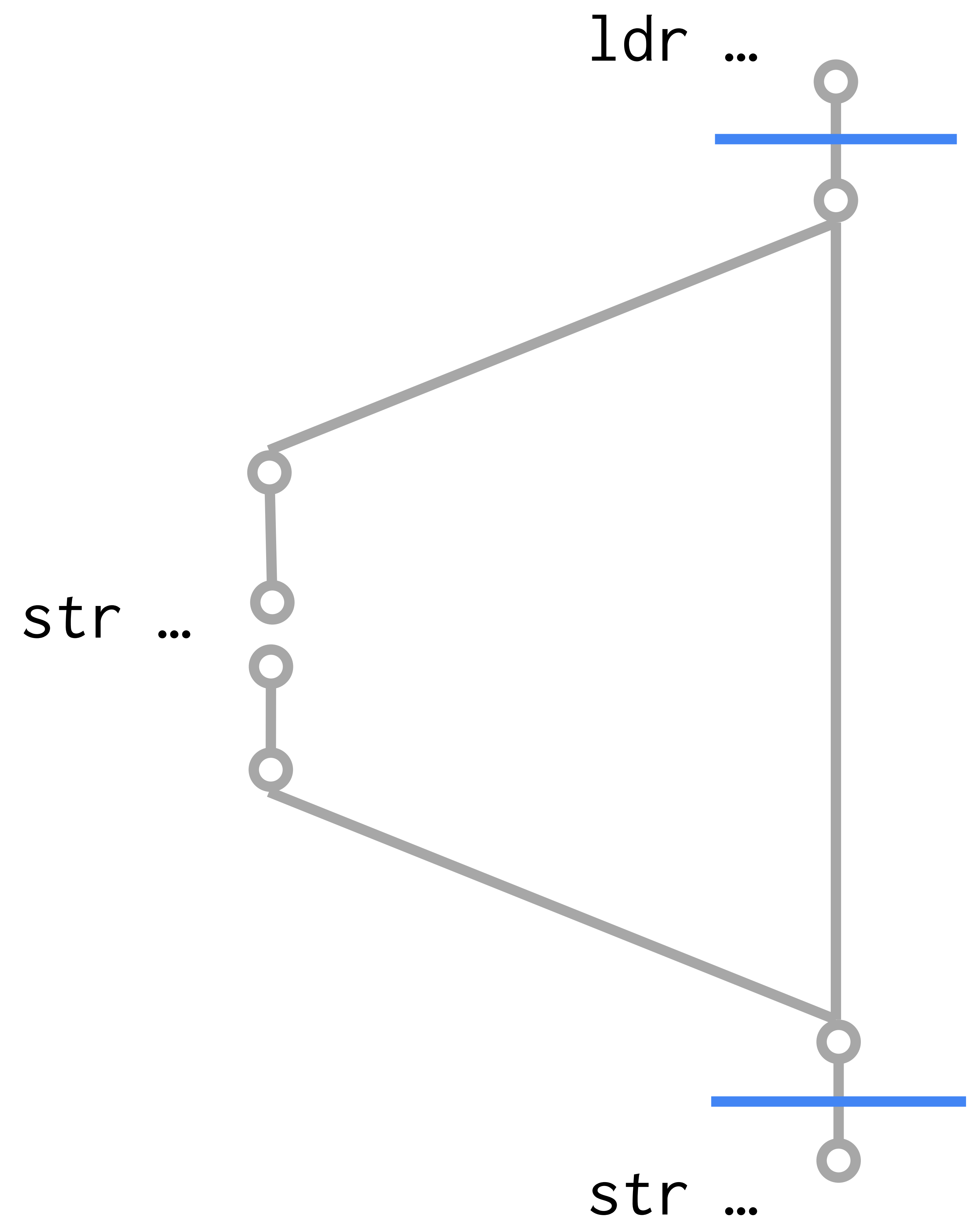
2 fences on main path



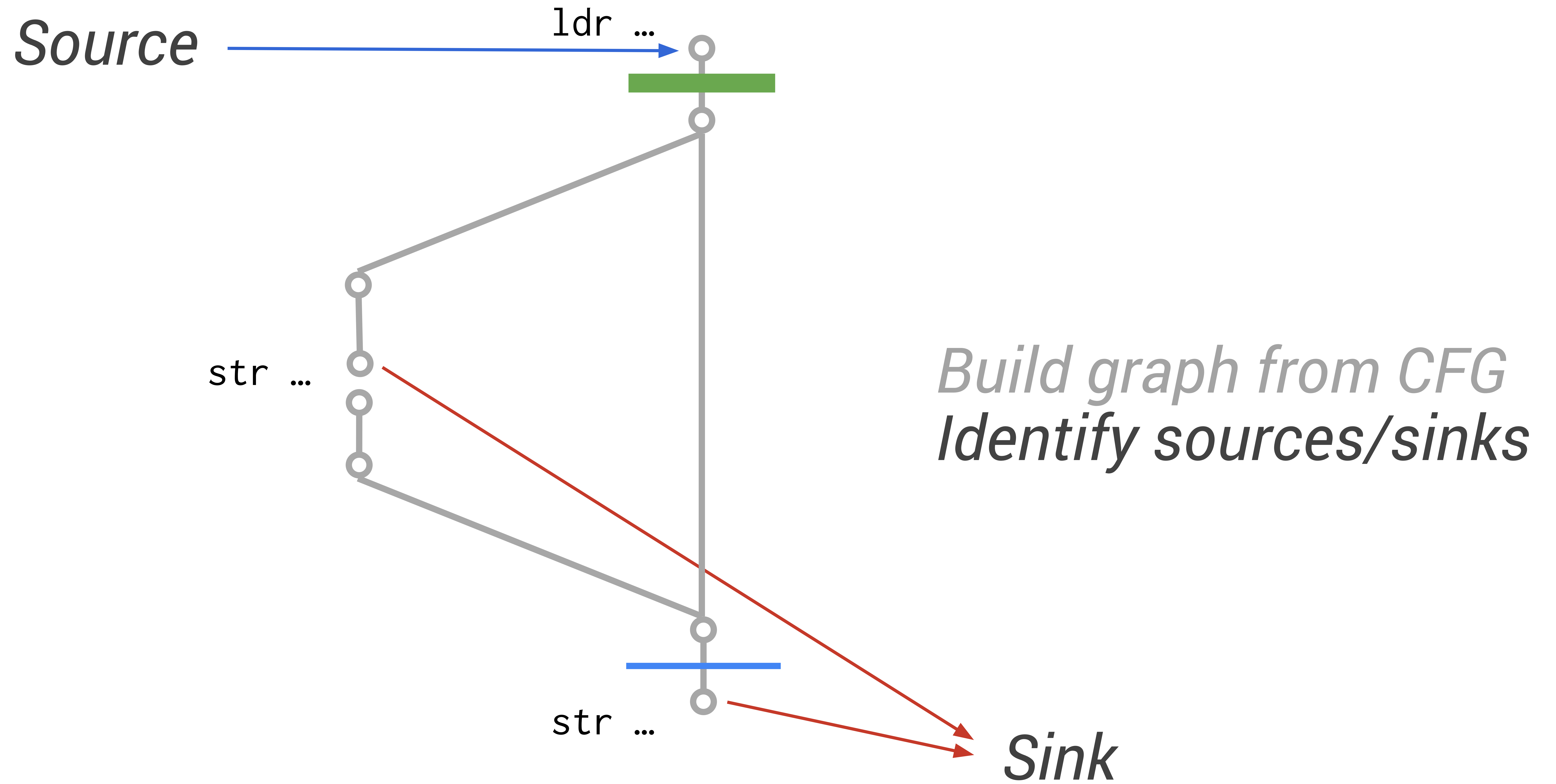
1 fence on main path

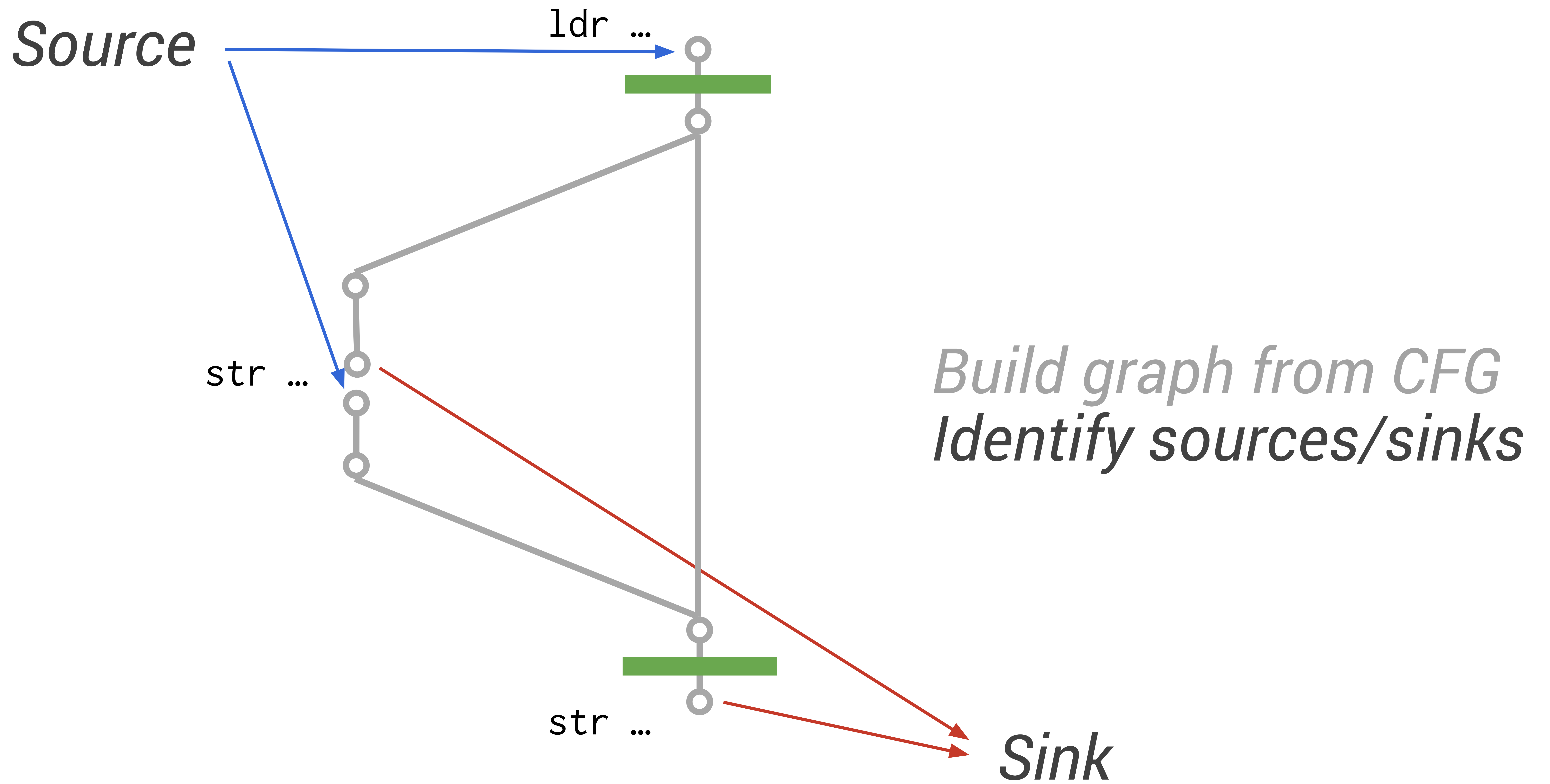


1 fence on main path

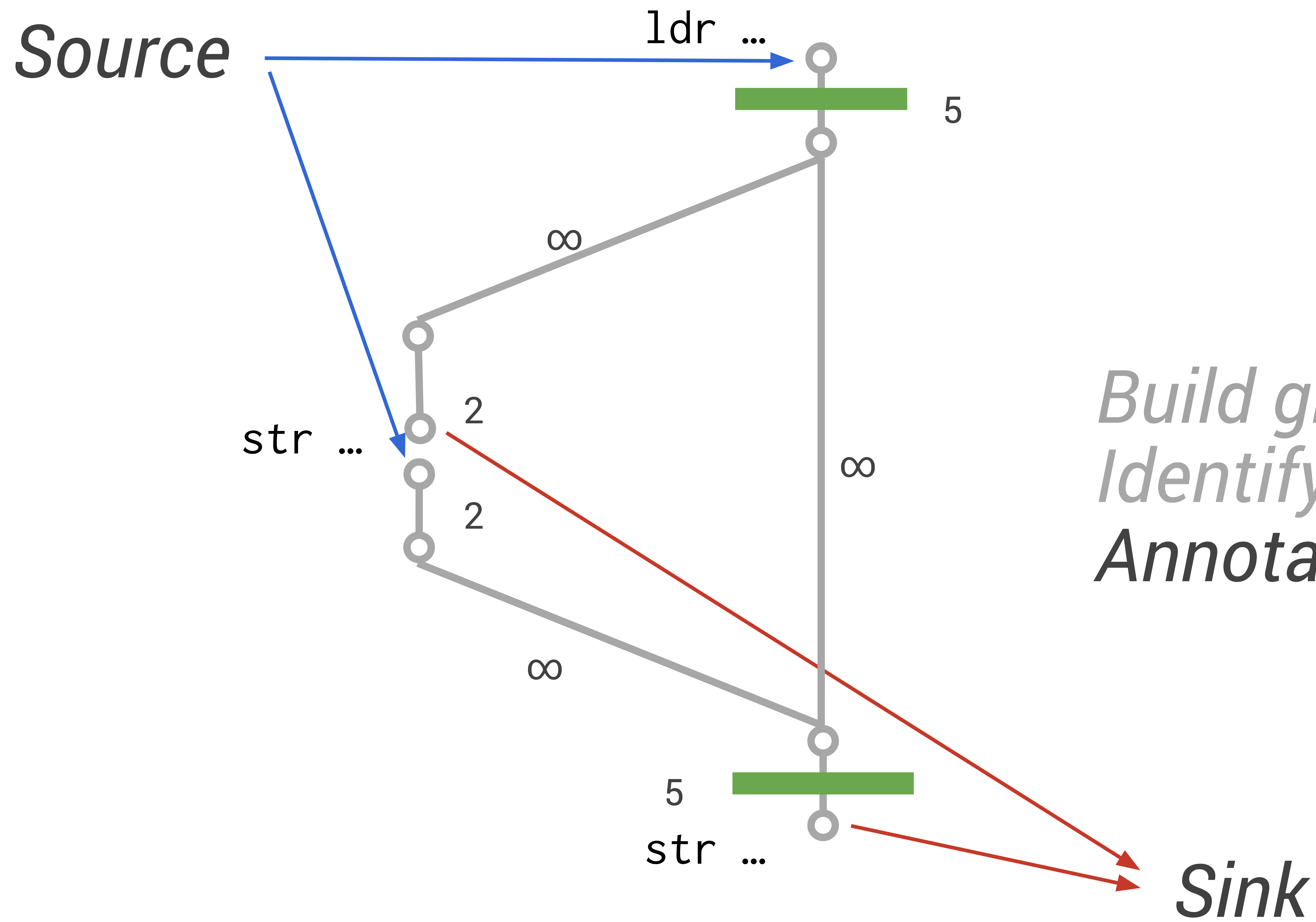


Build graph from CFG

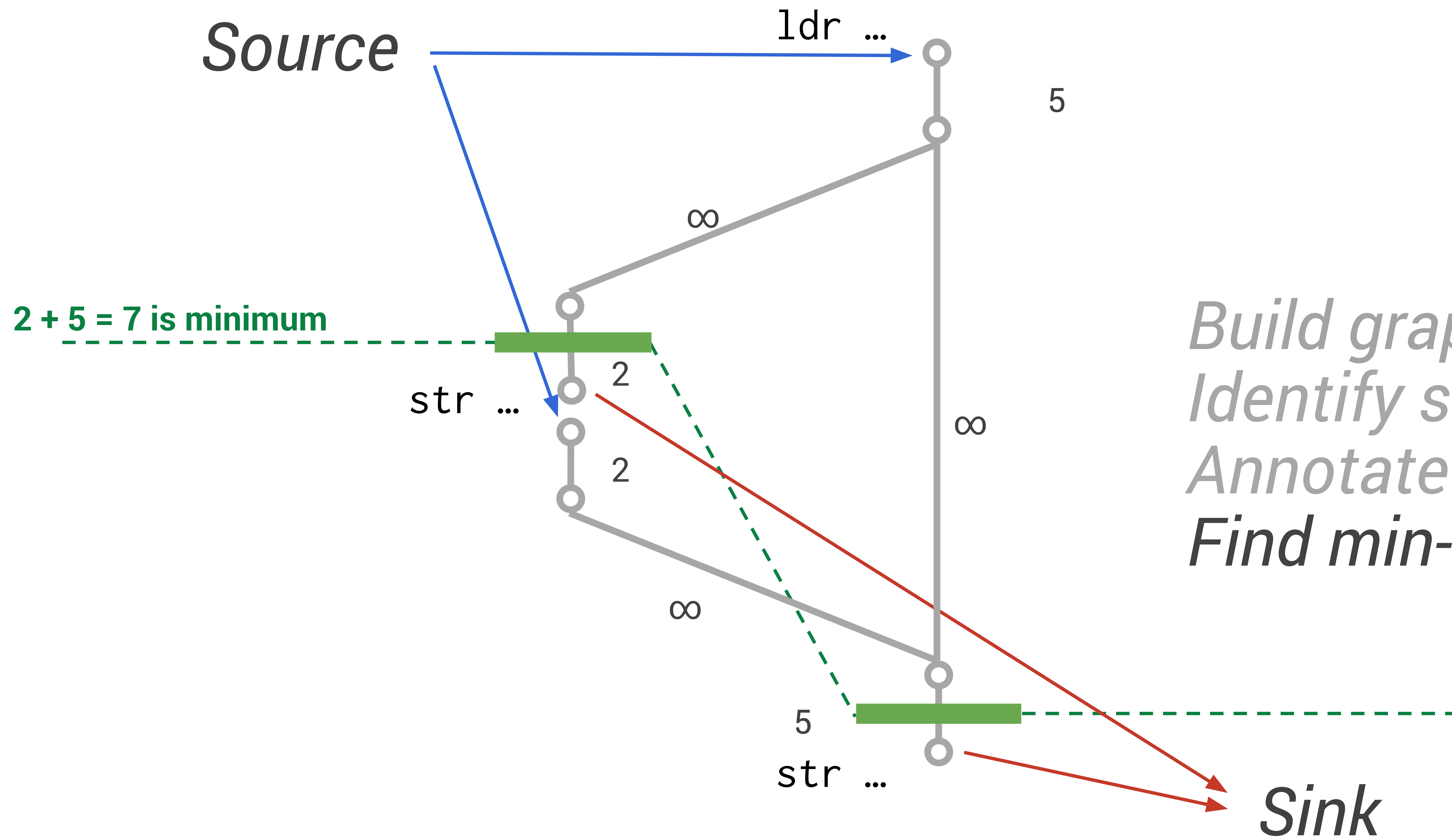

Source
ldr …
str …
str …
Build graph from CFG
Identify sources/sinks
Sink



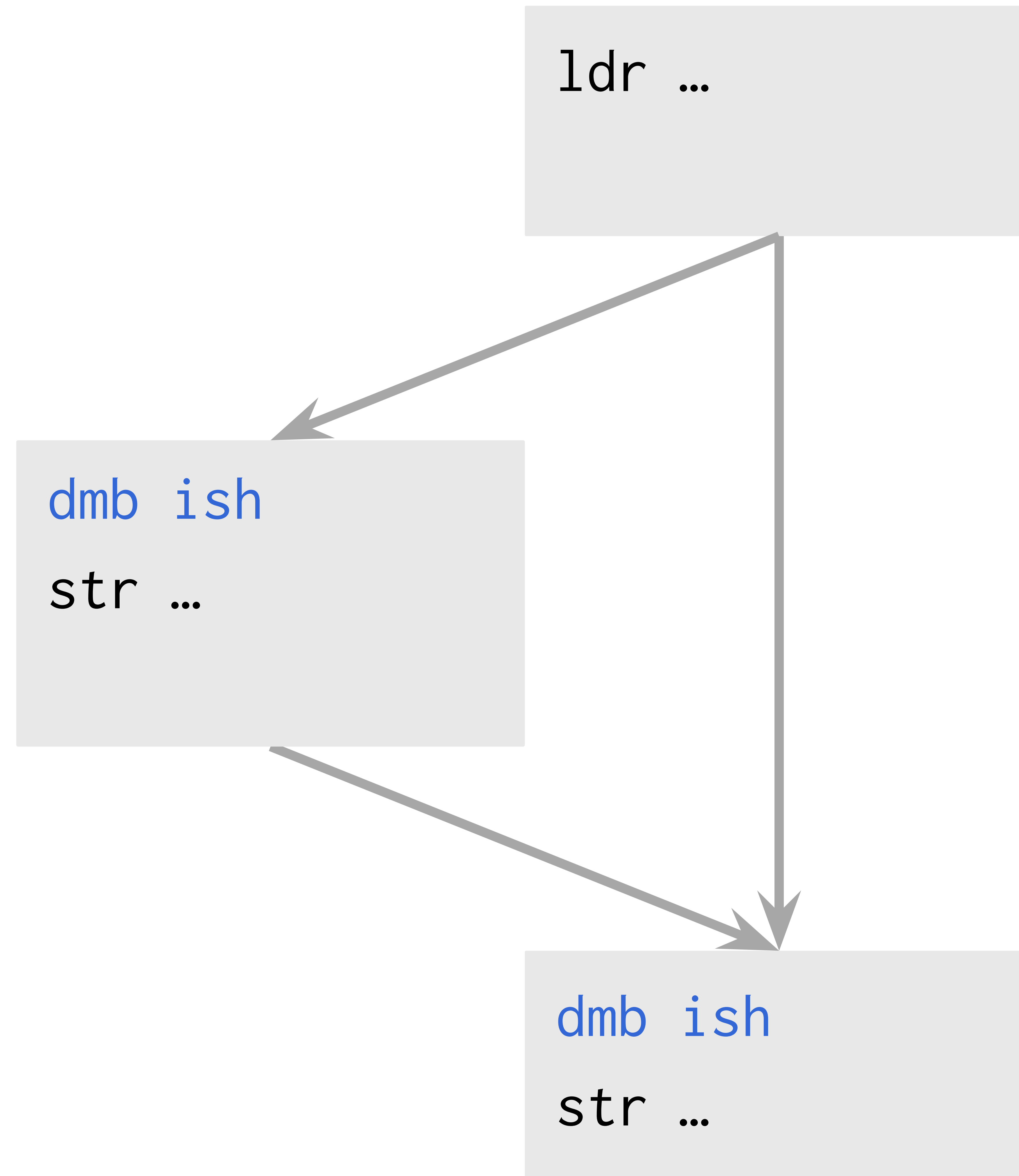
Build graph from CFG
Identify sources/sinks



Build graph from CFG
Identify sources/sinks
Annotate with frequency



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



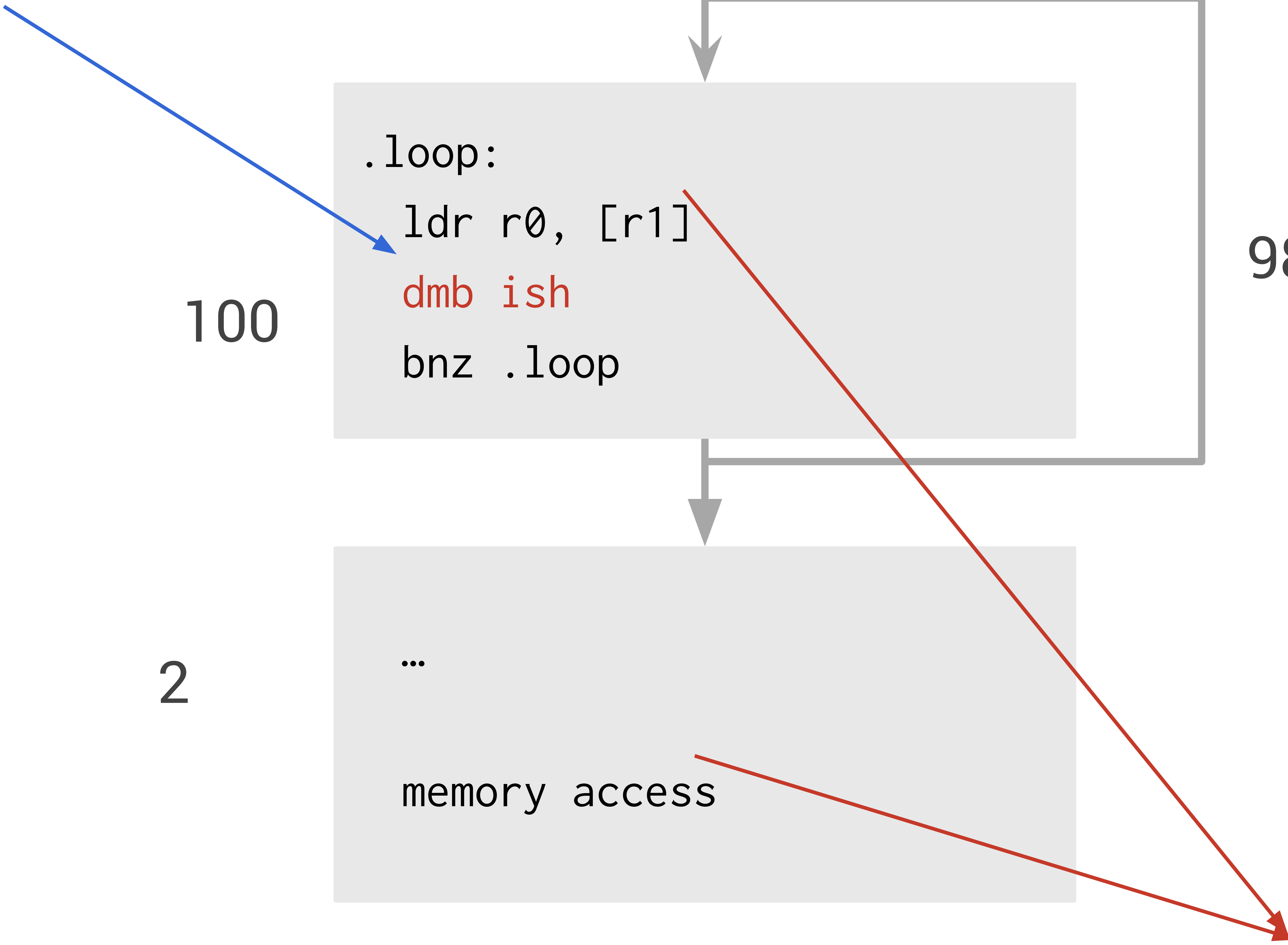
Build graph from CFG
Identify sources/sinks
Annotate with frequency
Find min-cut
Move fences

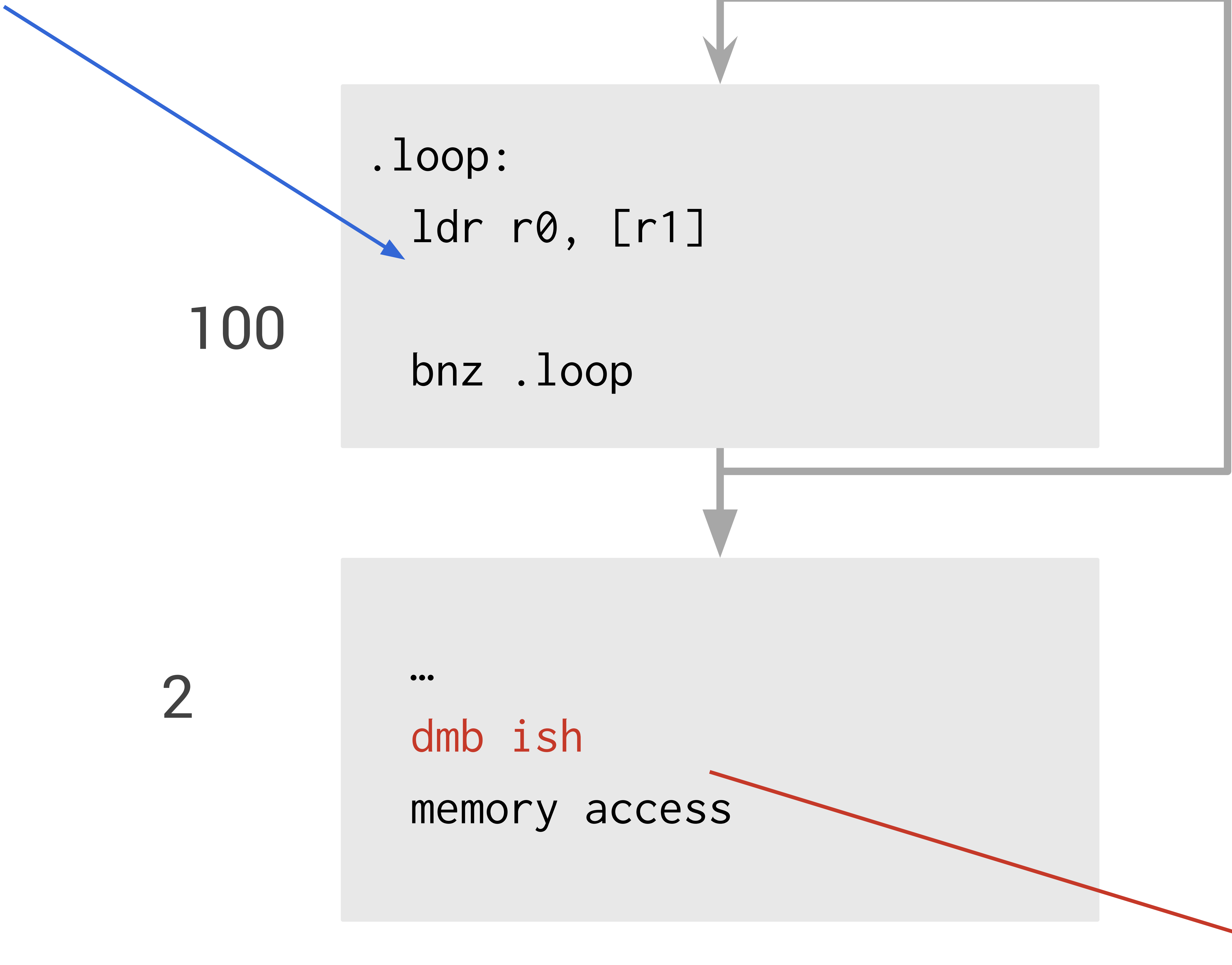
```
while(flag.load(acquire))
  {}
```

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

```
while(flag.load(acquire))
  {}
```

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


Source
.loop:
ldr r0, [r1]
dmb ish
bnz .loop
98
100
2
…
memory access
Sink


Source
.loop:
ldr r0, [r1]
bnz .loop
100
98
…
dmb ish
memory access
2
Sink

Background: C++11 atomics

Optimizing around atomics

Fence elimination

Miscellaneous optimizations

Further work: Problems with atomics

```
x.load(release) ?
```

x.load(**release**) ?

```
x.fetch_add(0, release)
```

`x.load(release) ?`

x86

```
x.fetch_add(0, release)
```

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

`x.load(release) ?`

x86

```
x.fetch_add(0, release)
```

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

```
mfence
mov %eax, (%ebx)
```

7200% speedup for a seqlock*

	Power	ARM
`x.store(0, release)`	`hwsync` `stw …`	`dmb sy` `str …`
`x.load(acquire)`	`lwz …` `hwsync`	`ldr …` `dmb sy`

	Power	ARM
`x.store(0, release)`	`lwsync` `stw …`	`dmb ish` `str …`
`x.load(acquire)`	`lwz …` `lwsync`	`ldr …` `dmb ish`

	Power	ARM (Swift)
`x.store(0, release)`	`lwsync` `stw …`	`dmb ishst` `str …`
`x.load(acquire)`	`lwz …` `lwsync`	`ldr …` `dmb ish`

```
x.store(2, relaxed)
```

Power

```
    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)
```

```
    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
```

Power

Shuffling

```
x.store(2, relaxed)
```

Load linked
Store conditional

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
```

```
x.store(2, relaxed)
```

Power

```
li r2, 2
stb r2, 0(r3)
```

```
x.store(2, relaxed)
```

x86

```
mov %eax, $2
mov (%ebx), %eax
```

```
x.store(2, relaxed)
```

x86

```
mov (%ebx), $2
```

Background: C++11 atomics

Optimizing around atomics

Fence elimination

Miscellaneous optimizations

Further work: Problems with atomics

Relaxed attribute

```
print(y.load(relaxed));

x.store(1, relaxed);
```

```
print(x.load(relaxed));

y.store(1, relaxed);
```

Relaxed attribute

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

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

Can print 1-1

Relaxed attribute

```
t_y = y.load(relaxed);
x.store(t_y, relaxed);
```

```
t_x = x.load(relaxed);
y.store(t_x, relaxed);
```

x = y = ???

Relaxed attribute

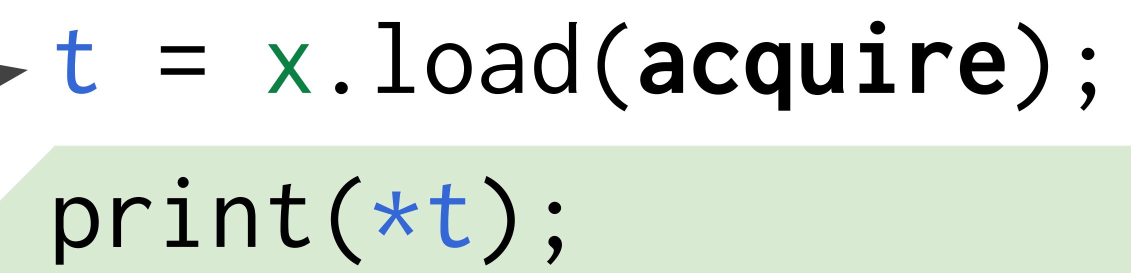
```
if(y.load(relaxed))
  x.store(1, relaxed);
  print(“foo”);
```

```
if(x.load(relaxed))
  y.store(1, relaxed);
  print(“bar”);
```

Can print foobar !

Consume attribute

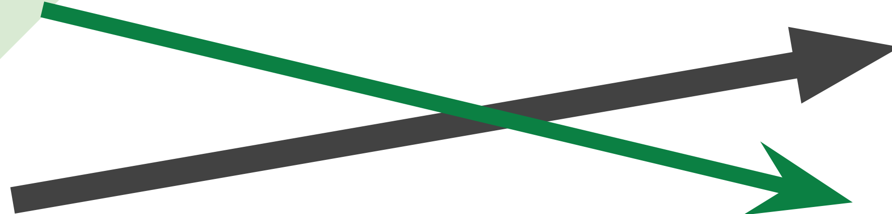
```
*x = 42;
x.store(1, release);
```



```
t = x.load(acquire);
print(*t);
```

Consume attribute

```
*x = 42;
x.store(1, release);
```



```
t = x.load(consume);
print(*t);
```

Ordered

Consume attribute

```
*x = 42;
x.store(1, release);
```

```
t = x.load(consume);
print(*y);
```

Unordered !

Consume attribute

```
*x = 42;
x.store(1, release);
```

```
t = x.load(consume);
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 ?