An Update on Optimizing Multiple Exit Loops

Philip Reames

LLVM Developers' Meeting 2020
October 6-8, 2020

Parts of a loop

```
preheader:
  br label %header

header:                           ;; <-- exiting block
  ...
  br i1 %c1, label %latch, label %exit_block_1
latch:                            ;; <-- exiting block
  ...
  br i1 %c2, label %header, label %exit_block_2

exit_block_1:
  ...
exit_block_2:
  ...
```

See https://llvm.org/docs/LoopTerminology.html

Exiting branches

Invariant Exit

```
br i1 %invariant, label %in_loop, label %out_of_loop
```

Computable Exit

```
%iv = <0,+,1>
%cmp = icmp ult i64, %iv, 64
br i1 %cmp, label %in_loop, label %out_of_loop
```

Variant (Data Dependent) Exit

```
%cmp = load i1, i1* %loop_varying_addr
br i1 %cmp, label %in_loop, label %out_of_loop
```

If this reminds you of SCEV's LoopDisposition, there's a reason!

Describes a point in time.

```
%cmp = load i1, i1* %loop_varying_addr
br i1 %cmp, label %in_loop, label %out_of_loop
```

What if %loop_varying_addr happens to point to a constant array?

There's an analogous extension for non-exiting branches.

Our starting point

LICM + Unswitch => Single Exit Loops
(For invariant exits)

IndVarSimplify
(for computable exits)

```
BasicBlock *ExitingBB = L.getExitingBlock();
if (!ExitingBB)
  return false;
```

A first attempt

Inductive Range Check Elimination

Eliminates computable exits by introduce pre/main/post loop structure. Requires duplicating loop 3x.

Loop Predication

Classic technique from JIT world. Converts computable exits into loop invariant exits via speculative optimization.

Both techniques attempt to produce “canonical” single exit loops.

Lesson learned

Results are (performance) fragile.
Pass ordering a major problem in practice.

Called for a change in approach.
Big idea: existing passes should natively handle multiple exits

OrderedInstructions in LICM

Can we hoist length?

```
header:
  %iv = phi i64 [0, %entry], [%iv.next, %header]
  ... no throw instructions ...
  %length = load i64, i64* %addr, !invariant.load !{}
  call void @may_throw()
  %iv.next = add i64 %iv, 1
  %cmp = icmp ult %iv, %length
  br i1 %cmp, label %header, label %out_of_loop
```

Nov '18 & April '20. Work by Max Kazantsev and Nikita Popov.

Linear Function Test Replace (LFTR)

Before:

```
%iv = <0,+,1>
%cmp = icmp ult %iv, %N
br i1 %cmp, label %in_loop, label %out_of_loop
```

After:

```
%iv = <0,+,1>
%cmp = icmp ne %iv, %N
br i1 %cmp, label %in_loop, label %out_of_loop
```

Long standing canonicalization for single exit loops.

Linear Function Test Replace (LFTR)

Had serious correctness problems (even for single exit loops).

`(i < N) != ((i + 1) < (N + 1))`
when i + 1 or N +1 is potentially poison.

May-July '19. Work by Nikita Popov and Philip Reames.

Rewrite Loop Exit Values

With loop:

```
%cmp = icmp ne %iv, %N
br i1 %cmp, label %in_loop, label %out_of_loop
...
br i1 %uncomputable, label %in_loop2, label %out_of_loop2
```

Before:

```
%lcssa = phi i64 [%iv, %loop_block]
```

After:

```
%lcssa = phi i64 [%N, %loop_block]
```

(Where N is loop invariant)

Rewrite Loop Exit Values

Extension to mix of computable and uncomputable exits.

May have further exposed cost modeling problems.

Sept '19. Work by Philip Reames.

Loop Predication w/o Speculation

Before:

```
%cmp = icmp ult %iv, %N
br i1 %cmp, label %in_loop, label %out_of_loop
...
%cmp2 = icmp ult %iv, %M
br i1 %cmp2, label %in_loop2, label %out_of_loop2
```

After:

```
%cmp = icmp ult %M, %N
br i1 %cmp, label %in_loop, label %out_of_loop
...
%cmp2 = icmp ult %iv, %M
br i1 %cmp2, label %in_loop2, label %out_of_loop2
```

Loop Predication w/o Speculation

Legality:

- Read only loops
- No value defined in loop used along exiting edge
- All exits must be computable and must dominate the latch

Effect:

- Replace a computable branch w/an invariant one
- Makes unswitch and peeling more powerful
- May allow induction variable to become dead

Oct-Nov '19. Work by Philip Reames. Inspired by work by Maxim Kazantsev, and Sanjoy Das

SCEV gaps addressed

Avoiding exponential compile times w/huge SCEVs.

Missing simplifications around `{u,s}{min,max}`.

Handle non-canonical and/xor/or IR
(produced by SimpleLoopUnswitch).

Handle non-canonical sdiv/srem IR
(before instcombine).

'18-'20. Work by Max Kazantsev, Philip Reames, Florian Hahn, & Roman Lebedev

SCEV -analyze

```
$ opt -analyze -scalar-evolution \
      -scalar-evolution-classify-expressions=0 input.ll

Determining loop execution counts for: @foo
Loop %do.body: <multiple exits> Unpredictable
                                backedge-taken count.
   exit count for do.body: ((-1 * %n) + %x)
   exit count for if.end: ((-1 * %n) + ((2 + %n) umax %n))
   exit count for latch: <unknown>
Loop %do.body: max backedge-taken count is 4096
```

Peeling w/multiple exits

Added mechanics for peeling multiple exit loops.

Current profitability is very limited: normal latch + exits ending in deoptimize only. Cost modeling is a challenge when needs discussion!

Also fixed a bug related to profile updates when peeling less than estimated trip counts for all loops.

July-Aug, '19. Work by Serguei Katkov.

Unrolling w/multiple exits

Runtime unrolling. Late '17. Work by Anna Thomas.
(Disabled by default.)

Full & partial unrolling. July '20. Work by Whitney Tsang.
Previous prep work by Florian Hahn in '19.

Current directions

- Canonicalization vs optimizations (LFTR, RLEV)
- Code size costs for peel, unroll, and unswitch
- Missing passes: fusion, distribution, interchange, etc..
- Pass ordering interactions - specifically vectorizer

Vectorizer Robustness

Uniform Stores

```
for (int i = 0; i < 16; i++) {
   g_var = i;
}
```

Speculated loads

```
@G = external global [16 x i64]

for (int i = 0; i < 16; i++) {
  if (i % 2 == 0)
    sum += G[i]
}
```

Oct '18, Sept '19. Work by Anna Thomas & Philip Reames.

Questions?