

Tail Call Optimization

Tail Call Optimization (TCO) is an optimization that targets tail calls, those being function calls that are the final computation of a function. Rather than generating a new stack frame for that function call, the compiler can reuse the current stack frame.

For recursive functions, this is a simple process. If f is the name of a recursive function, and in it we have a `call f` instruction followed immediately by a `ret` instruction, then we can replace the two instructions with a `goto f`.

TCO is also possible with non-recursive functions, although this is harder since it's possible that registers may be overwritten or spilled local variables would not be available on the stack frame.

This motivates the use of Continuation Passing Style (CPS). As a reminder from 15-150, a function is written in CPS if:

- The function takes in continuations
- It calls functions with continuations (including itself) as tail calls
- It calls its continuations and does so in tail calls

Note that some compilers (such as SMLNJ) do CPS conversion, which converts all functions to CPS. This minimizes the usage of the stack for function calls via TCO.

Checkpoint 0

Write x86 for the following function, applying TCO. Assume that the first two argument registers are `%edi` and `%esi`, respectively.

```
int sumacc (int n, int acc) {
    if (n == 0) {
        return 0;
    }
    return sumacc (n - 1, acc + n);
}
```

Solution:

```
sumacc:
    cmp $0, %edi
    je .base_case

.base_case:
    mov $0, %eax
    ret

.recursive_case:
    addl %edi, %esi
```

```
dec %edi
jmp sumacc
```

Partial Redundancy Elimination

Partial Redundancy Elimination (PRE) is a very powerful optimization that eliminates **redundancies** and **partial redundancies**. Since it eliminates redundancies and moves loop invariant code outside of loops (given the correct conditions), it can be thought of as a more powerful version of CSE/GVN and LICM.

The idea behind PRE is that we want to identify any redundant calculations and bring it up as far as it can go without introducing any useless calculations (i.e. performing the calculation when the result is not used). After bringing it up as far as it can go (it should appear only once now), it should then be lowered as far as it can, such that it is still never calculated more than once along a given control flow path. There will now be many calculations **statically**, but not **dynamically**.

Checkpoint 1

Identify any redundant calculations in the following code, and the highest place it should go, along with the lowest place(s) it should go.

```
int f(int x);

int calculation (int n, int p) {
    int x = 0;
    int y = n - 1;
    if (p < 50) {
        f(y + 5);
    } else {
        for (int i = 0; i <= y; i++) {
            if (i + y + n > 25) {
                x++;
                f(x);
                f(y + n);
            }
        }
        f(y + 5);
    }
    return 1;
}
```

Solution:

Redundant calculations: $y + 5$, $y + n$

Highest place for $y + 5$: before the if statement

Lowest place for $y + 5$: where it is

Highest place for $y + n$: before the loop (only if transformation is done!). Otherwise, above the if statement inside of the loop

```

{
    int i = 0;
    if (i <= y) {
        do {
            if (i + y + n > 25) {
                x++;
                f(x);
                f(y + n);
            }
            i++;
        } while (i <= y);
    }
}

```

Lowest place for $y + n$: same as highest

Loop Unrolling

Loop unrolling is a classic optimization with lots of benefits. It reduces branching cost, enables better instruction scheduling, and opens doors to more optimizations and analyses—more effective CCP/CSE, vectorization, and better dependency analyses results.

Loop optimizations have many corner cases. To simplify their implementation, modern compilers like LLVM first transform loops into a canonical form:

- A pre-header.
- A single back edge with a single latch block (source of the back edge).
- All exits should have no predecessor from outside the loop.
- All live variable across loop boundary should be redefined at loop exits. This form is called Loop-Closed SSA (LCSSA).

Loop canonicalization can be achieved via a "loop rotation" pass. Its effect can be visualized through the following C code:

<pre> // Original loop for (int i = 0; i < n; ++i) { body(); } </pre>	<pre> // Rotated loop if (i < n) { do { body(); } while (i < n); } </pre>
--	---

Note: if you're sure this loop will be executed at least once (e.g. through loop-range analysis), you can eliminate the first if check.

A word on LCSSA: Variables that are live across loop boundaries can make things difficult. To ensure your loop transformations only affect program within the loop region, LLVM enforces Loops to be in LCSSA form by inserting single entry ϕ -nodes at loop exits to break apart those live variables' def-use chain.

Checkpoint 2

Convert the following loop into its canonical form using loop rotation (for simplicity ignore LCSSA for this question):

```

1 define void @singleloop(ptr %p, i32 %k) {
2 entry:
3   br %loop.header
4
5 loop.header: ; preds = %entry, %loop.body
6   %iter = phi i32 [ 0, %loop.header ], [ %iter.update, %loop.body ]
7   %p.addr.04 = phi ptr [ %p, %loop.header ], [ %incdec.ptr, %loop.body ]
8   %cmp3 = icmp slt i32 %iter, %k
9   br i1 %cmp3, label %loop.body, label %end
10
11 loop.body: ; preds = %loop.header
12   store i32 %iter, ptr %p.addr.04, align 4
13   %iter.update = add nsw i32 %iter, 1
14   %incdec.ptr = getelementptr inbounds i32, ptr %p.addr.04, i32 1
15   br label %loop.header !llvm.loop !1
16
17 end: ; preds = %loop.header, %entry
18   ret void
19 }

```

Solution:

```

1 define void @singleloop(ptr %p, i32 %k) {
2 entry:
3   br label %loop.preheader
4
5 loop.preheader: ; preds = %entry
6   br label %loop.body
7
8 loop.body: ; preds = %loop.preheader, %loop.body
9   %iter = phi i32 [ 0, %loop.preheader ], [ %iter.update, %loop.body ]
10  %p.addr.04 = phi ptr [ %p, %loop.preheader ], [ %incdec.ptr, %loop.body ]
11  store i32 %iter, ptr %p.addr.04, align 4
12  %iter.update = add nsw i32 %iter, 1
13  %incdec.ptr = getelementptr inbounds i32, ptr %p.addr.04, i32 1
14  %cmp = icmp slt i32 %iter.update, %k ; Loop runs for %k iterations
15  br i1 %cmp, label %loop.body, label %loop.exit, !llvm.loop !1
16
17 loop.exit: ; preds = %loop.body
18   br label %end
19
20 end: ; preds = %loop.exit, %entry
21   ret void
22 }

```

Now onto the real business. Loop unroll achieves something like this:

```

// Original loop: i = {0,+,1}
for (int i = 0; i < n; ++i) {
    body(i);
}

```

```

// Fully Unrolled (tripcount: n = 4)
// i = {0,+,1}
body(0 + 0 * 1); // clone 1
body(0 + 1 * 1); // clone 2
body(0 + 2 * 1); // clone 3
body(0 + 3 * 1); // clone 4

```

```

// Peeled (tripcount: n >= 4)
// i = {0,+,1}
body(0 + 0 * 1); // clone 1
body(0 + 1 * 1); // clone 2
body(0 + 2 * 1); // clone 3
body(0 + 3 * 1); // clone 4
for (int i = 4; i < n; ++i) {
    body(i);
}

```

```

// Partially Unrolled (tripcount: ??)
// i = {0,+,1}
int step = 1;
int tc = n / step; // tripcount (truncated)
int factor = 4
int tc_floor = (tc/factor) * factor;

int i = 0;
for (; i < tc_floor * step; i += 4 * step) {
    body(0 + (i + 0) * step); // clone 0
    body(0 + (i + 1) * step); // clone 1
    body(0 + (i + 2) * step); // clone 2
    body(0 + (i + 3) * step); // clone 3
}
// epilogue: left over
for (; i < n; i += step) {
    body(i);
}

```

Checkpoint 3

Perform loop peeling on your canonical loop from previous checkpoint by peeling off the first two iterations. Assume you've already performed SCEV analysis on induction variable %k.

Solution:

```

1 define void @basic(ptr %p, i32 %k) {
2   store i32 0, ptr %p, align 4
3   %cmp.peel = icmp sgt i32 %k, 1
4   br i1 %cmp.peel, label %loop.body.peel2, label %end
5
6 loop.body.peel2:                                ; preds = %entry
7   %incdec.ptr.peel = getelementptr inbounds nuw i8, ptr %p, i64 4
8   store i32 1, ptr %incdec.ptr.peel, align 4
9   %cmp.peel5.not = icmp eq i32 %k, 2
10  br i1 %cmp.peel5.not, label %end, label %loop.body.peel7
11
12 loop.body.peel7:                                ; preds = %loop.body.peel2
13  %incdec.ptr.peel3 = getelementptr inbounds nuw i8, ptr %p, i64 8
14  %incdec.ptr.peel8 = getelementptr inbounds nuw i8, ptr %p, i64 12
15  store i32 2, ptr %incdec.ptr.peel3, align 4
16  %cmp.peel10 = icmp sgt i32 %k, 3
17  br i1 %cmp.peel10, label %loop.body, label %end
18
19 loop.body:                                       ; preds = %loop.body.peel7, %loop.body
20  %iter = phi i32 [ %iter.update, %loop.body ], [ 3, %loop.body.peel7 ]
21  %p.addr.04 = phi ptr [ %incdec.ptr, %loop.body ], [ %incdec.ptr.peel8, %loop.body.peel7 ]
22  %incdec.ptr = getelementptr inbounds nuw i8, ptr %p.addr.04, i64 4
23  store i32 %iter, ptr %p.addr.04, align 4
24  %iter.update = add nuw nsw i32 %iter, 1
25  %cmp = icmp slt i32 %iter.update, %k
26  br i1 %cmp, label %loop.body, label %end, !llvm.loop !0
27
28 end:                                            ; preds = %entry, %loop.body.peel2, %loop.body.peel7, %
    loop.body
29 ret void
30 }

```