Function Inlining

15-411/15-611 Compiler Design

Seth Copen Goldstein

April 16, 2026

© 2019-21 Goldstein

Today

- Why code layout matters
- Basic-block ordering and fall-through
- Static heuristics
- Hot/cold splitting
- Function placement and alignment
- Using Profile Data
- Example

© 2019-21 Goldstein

What is Function Inlining

Before

```
int add1(int x) {
  return x + 1;
}

foo() {
  …
  y = add1(a);
```

After

```
foo() {
  …
  t = a + 1;
  y = t;
```

Replace a call with specialized copy of callee body

- Map actual arguments to formal parameters
- Splice the callee CFG into the caller CFG
- Replace returns with jumps to caller continuation
- Merge returned values

© 2019-21 Goldstein

Why Function Inlining

- Eliminate call and return overhead
- Exposes constants, copies, and control-flow context across the call boundary
- Enables CCP, copy propagation, dead code elimination, and CFG simplification
- Can remove dead parameters and dead branches
- Allows later optimizations to “see through” abstraction boundaries

© 2019-21 Goldstein

What to Inline

- Not everything!

- Code size and I-cache pressure
- Compile times
- Possible spills in hot code
- Recursive or mutually recursive functions

- Opportunities for optimization

© 2019-21 Goldstein

Things to Keep in Mind

- Recursion?
- Callee size
- Hot or not? (needs profiling data)
- Constant arguments?
- varargs? Exceptions?
- directives
- Debug info

© 2019-21 Goldstein

When to Inline

- AST?
- Tree-IR?
- Abstract Assembly?
- Pre- or Post-optimization?

© 2019-21 Goldstein

When to Inline

- After initial cleanup of AA
- Before any machine specific passes
- Inline
- Apply more optimizations
- Maybe inline again?

© 2019-21 Goldstein

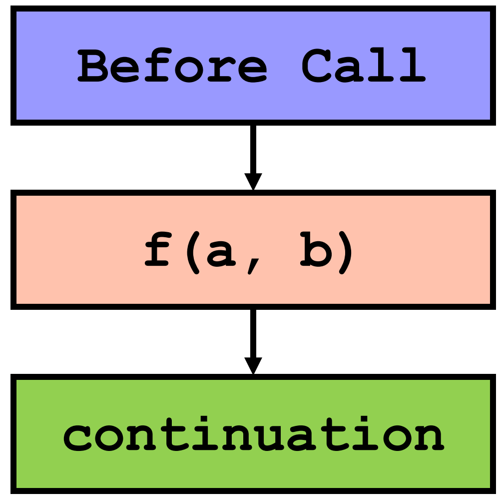
Basic Approach

- Choose call site and callee
- Clone callee CFG into fresh caller blocks
- Map formals to actuals
- Create fresh SSA-named for callee clone
- Redirect all returns to caller continuation
- Merge return values (if needed)
- (if only 1 caller) delete callee
- Delete original call

© 2019-21 Goldstein

The Layout Problem

Caller before



After Inlining

Before Call

```
f.p1 = a
f.p2 = b
…
return f.x1
…
return f.y2
```

continuation

© 2019-21 Goldstein

Maintaining SSA

- Formals to actuals
  - A map for each call site: callee names -> caller values
- Cloned variables/temps
  - Generate fresh names
  - Populate map of callee names -> in caller names
- Multiple return sites
  - Create phi at caller continuation to merge values

© 2019-21 Goldstein

Two Approaches

- Clone then repair
- Maintain SSA as you clone
  - Rename as you clone
  - Explicitly insert phi to merge return values

© 2019-21 Goldstein

Choosing

- Good Candidates:
  - Small callee
  - Hot call edge (in loop?)
  - Arguments are constants
  - One (or a few) call sites
  - Potentially dead-code/branch simplification
  - Callee is leaf function
- Bad Candidates:
  - Large callee
  - Cold call edge
  - Caller trace already hot and large
  - Potential spill increase

© 2019-21 Goldstein

Example

- Call site: 1 constant arg
- Callee:
  - Called once
  - Potential dead code
  - Small

```
int f(int p, int q) {
  int t = p + q;
  if (q < 0)
    return t;
  else
    return p - q;
}

int g(int x) {
  int y = f(x, 4);
  return y * 2;
}
```

- Track args
- Track returns

© 2019-21 Goldstein

SSA

```
int f(int p, int q) {
  int t = p + q;
  if (q < 0)
    return t;
  else
    return p - q;
}

 int g(int x) {
  int y = f(x, 4);
  return y * 2;
}
```

```
int f(int p0, int q0) {
  t0 = p0 + q0
  if (q0 < 0) F1 else F2
F1:
  return t0;
F2:
  t1 = p0 - q0
  return t1
}

 int g(int x0) {
  y0 = f(x0, 4)
  t0 = y0 * 2
  return t0
}
```


© 2019-21 Goldstein

Build Parameter Map

Map:

p0 -> x0

q0 -> 4

```
int f(int p0, int q0) {
  t0 = p0 + q0
  if (q0 < 0) F1 else F2
F1:
  return t0;
F2:
  t1 = p0 - q0
  return t1
}

 int g(int x0) {
  y0 = f(x0, 4)
  t0 = y0 * 2
  return t0
}
```


© 2019-21 Goldstein

Create Continuation

Map:

- p0 -> x0
- q0 -> 4

```
int f(int p0, int q0) {
  t0 = p0 + q0
  if (q0 < 0) F1 else F2
F1:
  return t0;
F2:
  t1 = p0 - q0
  return t1
}

 int g(int x0) {
  clone(f(x0, 4))
f.cont:
  f.ret = φ?
  y0 = f.ret
  t0 = y0 * 2
  return t0
}
```


© 2019-21 Goldstein

Clone with fresh variables

Map:

- p0 -> x0
- q0 -> 4
- t0 -> f.t0
- t1 -> f.t1

```
  f.t0 = x0 + 4
  if (4 < 0) f.F1 else f.F2
f.F1:
  f.ret0 = t0
  goto f.cont
f.F2:
  f.t1 = x0 - 4
  f.ret0 = f.t1
  goto f.cont

f.cont:
  f.ret = φ(f.ret0, f.ret1)
```

```
int f(int p0, int q0) {
  t0 = p0 + q0
  if (q0 < 0) F1 else F2
F1:
  return t0;
F2:
  t1 = p0 - q0
  return t1
}

 int g(int x0) {
  clone(f(x0, 4))
f.cont:
  f.ret = φ?
  y0 = f.ret
  t0 = y0 * 2
  return t0
}
```


© 2019-21 Goldstein

Replace call

```
int g(int x0) {
  f.t0 = x0 + 4
  if (4 < 0) f.F1 else f.F2
f.F1:
  f.ret0 = t0
  goto f.cont
f.F2:
  f.t1 = x0 - 4
  f.ret0 = f.t1
  goto f.cont

f.cont:
  f.ret = φ(f.ret0, f.ret1)
  y0 = f.ret
  t0 = y0 * 2
  return t0
}
```


© 2019-21 Goldstein

Post optimizations

```
 int g(int x0) {
  f.t0 = x0 + 4
  if (4 < 0) f.F1 else f.F2
f.F1:
  f.ret0 = t0
  goto f.cont
f.F2:
  f.t1 = x0 - 4
  f.ret0 = f.t1
  goto f.cont

f.cont:
  f.ret = φ(f.ret0, f.ret1)
  y0 = f.ret
  t0 = y0 * 2
  return t0
}
```

```
 int g(int x0) {





f.F2:
  f.t1 = x0 - 4







  t0 = f.t1 * 2
  return t0
}
```


© 2019-21 Goldstein

Summary

- At call site evaluate potential inlining
- Clone with fresh variables
- Then, perform optimizations

© 2019-21 Goldstein