Automatically inlining functions is not easy

Git revision #9c2913af[1]
Modified at 11. August 2025 16:38

Written by alex_s168[2]

Introduction

Function calls have some overhead, which can sometimes be a big issue for other optimizations. Because of that, compiler backends (should) inline function calls. There are however many issues with just greedily inlining calls…

Greedy inlining with heuristics

This is the most obvious approach. We can just inline all functions with only one call, and then inline calls where the inlined function does not have many instructions.

Example:

```
function f32 $square(f32 %x) {
@entry:
  // this is stupid, but I couldn't come up with a better example
  f32 %e = add %x, 0
  f32 %out = add %e, %x
  ret %out
}

function f32 $hypot(f32 %a, f32 %b) {
@entry:
  f32 %as = call $square(%a)
  f32 %bs = call $square(%b)
  f32 %sum = add %as, %bs
  f32 %o = sqrt %sum
  ret %o
}

function f32 $tri_hypot({f32, f32} %x) {
  f32 %a = extract %x, 0
  f32 %b = extract %x, 1
  f32 %o = call $hypot(%a, %b) // this is a "tail call"
  ret %o
}

// let's assume that $hypot is used someplace else in the code too
```

[1] https://github.com/alex-s168/website/tree/9c2913af189b62c028f6f773370f50f9e6c13307
[2] https://alex.vxcc.dev

Let’s assume our inlining treshold is 5 operations. Then we would get — Waait there are multiple options…

Issue 1: (sometimes) multiple options

If we inline the `$square` calls, then `$hypot` will have too many instructions to be inlined into `$tri_hypot`:

```
...
function f32 $hypot(f32 %a, f32 %b) {
@entry:
  // more instructions than our inlining treshold:
  f32 %ase = add %a, 0
  f32 %as = add %ase, %a
  f32 %bse = add %b, 0
  f32 %bs = add %bse, %b
  f32 %sum = add %as, %bs
  f32 %o = sqrt %sum
  ret %o
}
...
```

The second option is to inline the `$hypot` call into `$tri_hypot`. (There are also some other options)

Now in this case, it seems obvious to prefer inlining `$square` into `$hypot`.

Issue 2: ABI requirements on argument passing

If we assume the target ABI only has one f32 register for passing arguments, then we would have to generate additional instructions for passing the second argument of `$hypot`, and then it might actually be more efficient to inline `$hypot` instead of `$square`.

This example is not realistic, but this issue actually occurs when compiling lots of code.

Another related issue is that having more arguments arranged in a fixed way will require lots of moving data arround at the call site.

A solution to this is to make the heuristics not just output code size, but also make it depend on the number of arguments / outputs passed to the function.

Issue 3: (sometimes) prevents optimizations

```
function f32 $myfunc(f32 %a, f32 %b) {
@entry:
  f32 %sum = add %a, %b
  f32 %sq = sqrt %sum
  ...
}

function $callsite(f32 %a, f32 %b) {
@entry:
  f32 %as = add %a, %a
  f32 %bs = add %b, %b
  f32 %x = call $myfunc(%as, %bs)
  ...
}
```

If the target has a efficient `hypot` operation, then that operation will only be used if we inline `$myfunc` into `$callsite`.

This means that inlining is now depended on… instruction selection??

This is not the only optimization prevented by not inlining the call. If `$callsite` were to be called in a loop, then not inlining would prevent vectorization.

Function outlining

A related optimization is “outlining”. It’s the opposite to inlining. It moves duplicate code into a function, to reduce code size, and sometimes increase performance (because of instruction caching)

If we do inlining seperately from outlining, we often get unoptimal code.

A better approach

We can instead first inline **all** inlinable calls, and **then** perform more agressive outlining.

Step 1: inlining

We inline **all** function calls, except for:
- self recursion (obviously)
- functions explicitly marked as no-inline by the user

Step 2: detect duplicate code

There are many algorithms for doing this.

The goal of this step is to both:
• maximize size of outlinable section
• minimize size of code

Step 3: slightly reduce size of outlinable section

The goal is to reduce size of outlinable sections, to make the code more optimal.

This should be ABI and instruction depended, and have the goal of:
• reducing argument shuffles required at all call sites
• reducing register preassure
• not preventing good isel choices and optimizations.

this is also dependent on the targetted code size.

Step 4: perform outlining

This is obvious.

Issue 1: high compile-time memory usage

Inlining **all** function calls first will increase the memory usage during compilation by A LOT

I'm sure that there is a smarter way to implement this method, without actually performing the inlining…

Conclusion

Function inlining is much more complex than one might think.

PS: No idea how to implement this…

Subscribe to the Atom feed[3] to get notified about futre compiler-related articles.

[3] atom.xml