Lecture Notes on
Function Optimizations

15-411: Compiler Design
Jan Hoffmann

Lecture 21
April 4, 2023

1 Introduction

If implemented naively, function calls can have significant overhead. For each function call, we may copy the arguments to adhere to the calling convention, increase the call stack to store local variables of the callee, and save local variables of the caller. This can be particularly costly for recursive functions that are called many times. However, in many cases we can avoid these overheads of function calls and there are several function optimizations that are based on this observation. Such optimizations are particularly important for functional languages. In this lecture, we will focus on two of the most effective ones: inline expansion and tail-call optimization.

2 Inline Expansion

Inline expansion, also called *function inlining*, is an optimization that replaces a function call with the function body of the called function. Of course, we have to take care to replace the formal arguments in the body with concrete arguments from the call to ensure that such a transformation is correct.

Inline expansion has the following benefits.

- It can lead to improved time and space usage because we do not have to allocate, setup, and deallocate a call frame for the callee. Additionally, we do not have to save and restore the registers of the caller.
- It enables additional intra-procedural optimizations such as constant propagation and dead-code elimination.

- It can improve register allocation since the register allocator can optimize a larger portion of code and does not have to reserve registers for the function arguments.

If inline expansion has all these benefits, why don't we inline all functions? First, we cannot inline recursive calls because we (in general) cannot predict the recursion depth at compile time. We can unroll a recursive function call but we cannot completely replace the call.

But also for non-recursive functions, it is not a good idea to inline all function calls. Assume you have 21 function $f_0, \ldots, f_{20}$ in a program. If f_i calls f_{i+1} 2 times and we inline all function calls then the body of f_{20} is copied $1,084,576$ times. This will almost certainly cause performance problems if the code does not fit in the cache anymore after inline expansion.

Finally, we might not be able to inline a function because the function body is *not available*. This is the case for library functions for which we do not have access to the source code. It could also be the case that we cannot statically determine which function is called. However, this is not a problem in L4 because we do not have features such as first-class functions, function pointers, or dynamic dispatch, which are used to dynamically select functions at a call.

Inlining Heuristics In L4, we statically know which function is called at all call sides. As a result, we could theoretically inline all locally-defined functions. However, we need to balance code growth and potential performance gains from inline expansion. That is why we need to decide for which functions and call sites we should perform inline expansion.

As with many optimizations, it is not possible to exactly predict the performance impact of inlining and we have to resort to *inlining heuristics*. Good choices for inline expansion are functions that are only called once. Inlining such functions will even reduce the code size. Similarly, we always inline a function whose body is shorter than the code we generate for calling the function (saving registers, setting up arguments, ...). In this case, inlining the function will also reduce the code size of the program.

In general, it is a good idea to inline small functions since inline expansion of small functions does not increase the code size significantly. However, it can be beneficial to avoid inline expansion for calls when the caller has a small function body since expanding such calls might reduce the opportunities for inline expansion for the caller. Similarly, you might want to avoid inline expansion for calls in conditional branches and give preference to expanding calls in loops. The rational behind these stratifies is that we want to maximize the performance improvements we get from a fixed increase in code size: Loop bodies are potentially executed often and conditional branches are potentially executed rarely. When performing inline expansion, we have to avoid creating huge function bodies because it increases the stress on the register allocator. So there should be a cutoff function size at which we stop inlining functions that further increase its size.

Implementing Inline Expansion As for other optimizations, we have to select the right intermediate language to perform inline expansion. If the intermediate language is to high-level then we do not yet know the sizes of the function bodies and it is more difficult to apply heuristics. However, we want to perform inline expansion before register allocation and other optimizations that might benefit from inlining. *Abstract assembly* has the advantage that we can either implement inlining as one of the first optimizations in abstract assembly or we could alternate it with other optimizations that change the code size of function bodies.

Consider for example the function catalan(n), which computes the nth Catalan number.

```
int next (int n, int c)
{
  return 2*(2*n+1)*c/(n+2);
}

int catalan (int n)
{
  int i = 0;
  int c = 1;
  while (i<n)
  {
    c = next(i,c);
    i = i+1;
  }
  return c;
}
```

The function next is short and a good candidate for inline expansion. When we perform the inline expansion, we have to make sure to rename the arguments (and local variables) of next to avoid name clashes with the variables of catalan. In an implementation, we would pick fresh names that cannot appear anywhere else in the program. Here, we simply rename as follows.

```
int next (int n', int c')
{
  return 2*(2*n'+1)*c'/(n'+2);
}
```

Now we replace the function call, with the function body in which we replace each return with an assignment to c. Moreover, we add a preamble in which we assign the actual parameters to the formal parameters.

```
int catalan (int n)
{
  int i = 0;
  int c = 1;
  while (i<n)
  {
    int n' = i;
    int c' = n;
    c = 2*(2*n'+1)*c'/(n'+2);
    i = i+1;
  }
  return c;
}
```

3 Tail-Call Optimization

Consider the following two implementations pow and powloop of the power function b^e.

```
int powacc (int b, int e, int a)
{
  if (e == 0)
    return a;
  else
    return powacc(b,e-1,a*b);
}

int pow(int b, int e)
{
  return powacc(b,e,1)
}
```

```
int powloop(int b, int e)
{
  int acc = 1;
  while (e>0)
  {
    e = e - 1;
    acc = acc * b;
  }
  return acc;
}
```

Which implementation is more efficient? Potentially, the first version, which uses a recursive function, but it depends on the compiler. The overhead of using functions mainly comes from maintaining stack frames. But why do we have stack frames? We store local variables and registers of the caller to resume its computation after the function call. However, if there is no computation after the function call, there is no need to preserve the registers or the local variables of the caller. So we do not need to preserve the stack frame of the caller and can directly return to the caller of the caller in the callee. This is often called *tail-call* optimization.

Tail-call optimization is usually performed on the abstract assembly level. Often it is only applied to recursive calls because it is easier to implement and leads to greater performance gains. However, it can also be applied to non-recursive calls.

Consider the following translation of the function pow to high-level abstract assembly. We have two opportunities for tail call optimization: both calls to powacc are immediately followed by a return of the value that has been return by the call to powacc. We now want to replace the call to powacc with a jump to its function body. However, we need to find a way to pass the concrete arguments and to pass on the return value to the previous caller.

```
powacc($b, e, a$) :
    if ($e = 0$) goto done
    $t_0 \leftarrow e - 1$
    $t_1 \leftarrow a * b$
    $t_2 \leftarrow$ powacc($b, t_0, t_1$)
    return $t_2$

done :
    return $a$

pow($b, e$) :
    $t_0 \leftarrow$ powacc($b, e, 1$)
    return $t_0$
```

Tail Recursion As we will see, tail calls are easier to implement if we introduce argument and return registers. Consider the following translation of the code to such a lower-level abstract assembly.

```
powacc :
    $b \leftarrow$ arg$_1$
    $e \leftarrow$ arg$_2$
    $a \leftarrow$ arg$_3$
    if ($e = 0$) goto done
    $t_0 \leftarrow e - 1$
    $t_1 \leftarrow a * b$
    arg$_1 \leftarrow b$
    arg$_2 \leftarrow t_0$
    arg$_3 \leftarrow t_1$
    call powacc
    $t_2 \leftarrow$ res
    res $\leftarrow t_2$
    ret

done :
    res $\leftarrow a$
    ret

pow :
    $b \leftarrow$ arg$_1$
    $e \leftarrow$ arg$_2$
    arg$_1 \leftarrow b$
    arg$_2 \leftarrow 2$
    arg$_3 \leftarrow 1$
    call powacc
    $t_0 \leftarrow$ res
    res $\leftarrow t_0$
    ret
```

Here, we want to replace calls that are immediately followed by a return with a jump.

```
call $f$    $\rightsquigarrow$    goto $f$
ret
```

To obtain a call immediately followed by a return with first apply copy propagation and dead code elimination. In the code below we have already applied tail call optimization to the block powacc.

```
powacc :
    $b \leftarrow$ arg$_1$
    $e \leftarrow$ arg$_2$
    $a \leftarrow$ arg$_3$
    if $(e = 0)$ goto done
    $t_0 \leftarrow e - 1$
    $t_1 \leftarrow a * b$
    arg$_1 \leftarrow b$
    arg$_2 \leftarrow t_0$
    arg$_3 \leftarrow t_1$
    goto powacc

done :
    res $\leftarrow a$
    ret

pow :
    $b \leftarrow$ arg$_1$
    $e \leftarrow$ arg$_2$
    arg$_1 \leftarrow b$
    arg$_2 \leftarrow 2$
    arg$_3 \leftarrow 1$
    call powacc
    ret
```

It's that simple. We only have to be careful when we introduce code that is setting up the stack frame (and potentially saving callee-saved registers) for the function powacc. This code should be in a separate block powacc_prologue that we call the function prologue. Each call call powacc should then replaced with call powacc_prologue. However, we do not need to setup a stack frame when we perform the tail call. So the jump goto powacc remains unchanged. This works because the jump goto powacc is an internal jump between two blocks of the same function (here one block).

Non-Recursive Calls Let us now consider the same tail call optimization for the function pow.

```
pow :
    $b \leftarrow$ arg$_1$
    $e \leftarrow$ arg$_2$
    arg$_1 \leftarrow b$
    arg$_2 \leftarrow 2$
    arg$_3 \leftarrow 1$
    goto powacc
```

This change would introduce a jump from the block of one function to the block of another function. This is in general not sound because spilled local variables are not available on the stack frame of the other function and registers might be overwritten.

I the present case, however, this optimization is correct if there is no other call to the function powacc. We merge powacc and pow into a single function before we perform register allocation setup stack frames. The general case, is more complex. We still have to setup the stack frame of the callee but we can overwrite the stack frame of the caller and return directly to the caller of the caller.

Register Allocation After the tail call optimization, the register allocator could make the following assignments.

$$\begin{array}{lcl} b & \mapsto & \mathsf{arg}_1 \\ e & \mapsto & \mathsf{arg}_2 \\ a & \mapsto & \mathsf{arg}_3 \\ t_0 & \mapsto & \mathsf{arg}_2 \\ t_1 & \mapsto & \mathsf{arg}_3 \end{array}$$

After we eliminate self moves, we arrive at code that is basically identical to the abstract assembly that we produce for the function powloop.

```
powacc :
    if $(e = 0)$ goto done
    $\mathsf{arg}_2 \leftarrow \mathsf{arg}_2 - 1$
    $\mathsf{arg}_3 \leftarrow \mathsf{arg}_3 * \mathsf{arg}_1$
    goto powacc

done :
    res $\leftarrow \mathsf{arg}_3$
    ret

pow :
    $\mathsf{arg}_1 \leftarrow 1$
    goto powacc
```