Lecture Notes on Calling Conventions

15-411: Compiler Design
Frank Pfenning

Lecture 11
September 30, 2014

1 Introduction

In Lab 3 you will be adding functions to the arithmetic language with loops and conditionals. Compiling functions creates some new issues in the front end and the back end of the compiler. In the front end, we need to make sure functions are called with the right number of arguments, and arguments of the right type. In the back end, we need to create assembly code that respects the *calling conventions* of the machine architecture. Strict adherence to the calling conventions is crucial so that your code can interoperate with library routines, and the environment can call functions that you define.

Calling conventions are rather machine-specific and often quite arcane. You must carefully read the Section 3.2 of the AMD64 ABI [MHJM09][1]. Examples and additional information is provided in Section 6 of a handout on x86-64 Machine-Level Programming by Bryant & O'Hallaron.

2 IR Trees

We have already seen in Lecture 10 that function calls should take pure arguments in order to easily guarantee the left-to-right evaluation order prescribed by our language semantics. Moreover, they should be lifted to the level of commands rather than remain embedded inside expressions because functions may have side-effects.

[1] Available at `http://refspecs.linuxfoundation.org/elf/x86-64-abi-0.99.pdf`

3 Low-Level Intermediate Language

In the low level intermediate language of quads that we have used so far in this course, it is convenient to add a new form of instruction

$$d \leftarrow f(s_1, \ldots, s_n)$$

where each s_i is a source operand and d is a destination operand.

The generic $\mathsf{def}(l, x)$, $\mathsf{use}(l, x)$ and $\mathsf{succ}(l, l')$ predicates are easily defined, assuming for simplicity that source and destinations are all temps.

$$\frac{l : d \leftarrow f(s_1, \ldots, s_n)}{\begin{array}{l}\mathsf{def}(l, d) \\ \mathsf{use}(l, s_i) \quad (1 \leq i \leq n) \\ \mathsf{succ}(l, l+1)\end{array}} J_8$$

Unfortunately, this is overly simplistic, because calling conventions prescribe the use of certain fixed registers for passing arguments and receiving results, so we will have to extend the above rule further.

4 x86-64 Calling Conventions

In x86-64, the first six arguments are passed in registers, the remaining arguments are passed on the stack. The result is returned in a specific return register `%rax`. These conventions do not count floating point arguments and results, which are passed in the dedicated floating point registers `%xmm0` to `%xmm7` and on the stack only if there are more than eight floating point parameters. Fortunately, our language has only integers at the moment, so you do not have to worry about the conventions for floating point numbers.

On the x86, stack frames were required to have a frame pointer `%ebp` (base pointer) which had to be saved and restored with each function call. It provided a reliable pointer to the beginning of a stack frame for easy calculation of frame offsets to handle references to arguments and local variables. It also allowed tools such as `gdb` to print backtraces of the stack. On the x86-64 this information is maintained elsewhere and a frame pointer is no longer required.

The general organization of stack frames at the time a procedure is called, will be as follows.

Address	Contents	Frame
⋯	⋯	Caller
`16(%rsp)`	argument 8	
`8(%rsp)`	argument 7	
`(%rsp)`	return address	

Note that all arguments take 8 bytes of space on the stack, even if the type of argument would indicate that only 4 bytes need to be passed.

The function that is called, the *callee*, should set up its stack frame, reserving space for local variables, spilled temps that could not be assigned to registers, and arguments passed to functions it calls in turn. We recommend calculating the total space needed and then decrementing the stack pointer `%rsp` by the appropriate amount. By changing the stack pointer only once, at the beginning, references to parameters and local variables remain constant throughout the function's execution. The stack then looks as follows, where the size of the callee's stack frame is n.

Position	Contents	Frame
...	...	Caller
$\mathtt{n} + \mathtt{16}(\mathtt{\%rsp})$	argument 8	
$\mathtt{n} + \mathtt{8}(\mathtt{\%rsp})$	argument 7	
$\mathtt{n} + \mathtt{0}(\mathtt{\%rsp})$	return address	
	local variables	Callee
	...	
	argument build area	
	for function calls	
	...	
$(\mathtt{\%rsp})$	end of frame	
	128 bytes	red zone

Note that `%rsp` should be aligned $0 \bmod 16$ before another function is called, and may be assumed to be aligned $8 \bmod 16$ on function entry. This happens because the `call` instruction saves the 64-bit return address on the stack.

The area below the stack pointer is called the *red zone* and may be used by the callee as temporary storage for data that is not needed across function calls or even to build arguments to be used before a function call. The ABI states that the red zone "shall not be modified by signal or interrupt handlers." This can be tricky, however, because, for example, Linux kernel code may not respect the red zone and overwrite this area. We therefore suggest not using the red zone.

5 Register Convention

We extract from [MHJM09] the relevant information on register usage. In the first column is a suggested numbering for the purpose of register allocation.

Abstract form	x86-64 Register	Usage	Preserved accross function calls
res_0	`%rax`	return value*	No
arg_1	`%rdi`	argument 1	No
arg_2	`%rsi`	argument 2	No
arg_3	`%rdx`	argument 3	No
arg_4	`%rcx`	argument 4	No
arg_5	`%r8`	argument 5	No
arg_6	`%r9`	argument 6	No
ler_7	`%r10`	caller-saved	No
ler_8	`%r11`	caller-saved	No
lee_9	`%rbx`	callee-saved	Yes
lee_{10}	`%rbp`	callee-saved*	Yes
lee_{11}	`%r12`	callee-saved	Yes
lee_{12}	`%r13`	callee-saved	Yes
lee_{13}	`%r14`	callee-saved	Yes
lee_{14}	`%r15`	callee-saved	Yes
	`%rsp`	stack pointer	Yes

The starred registers have a potentially relevant alternative use. `%al` (the lower 8 bits of `%rax`) contains the number of floating point arguments on the stack in a call to varargs functions. `%rbp` is the frame pointer for the stack frame, in an x86-like calling convention (which is optional for the x86-64).

6 Typical Calling Sequence

If we have 6 or fewer arguments, a typical calling sequence for 32-bit arguments with an instruction

$$d \leftarrow f(s_1, s_2, s_3)$$

will have the following form:

$$\begin{array}{l} arg_3 \leftarrow s_3 \\ arg_2 \leftarrow s_2 \\ arg_1 \leftarrow s_1 \\ \mathsf{call}\ f \\ d \leftarrow res_0 \end{array}$$

First we move the temps into the appropriate argument registers, then we call the function f (represented by a symbolic label), and then we move the result register into the desired destination.

This organization, perhaps just before register allocation, has the advantage that the live ranges of fixed registers (called *precolored nodes* in register allocation) is minimized. This is important to avoid potential conflict. We have already applied a similar technique in the implementation of div and mod operations, which expect their arguments in fixed registers.

Let us state this as a fundamental principle of code generation that you should strive to adhere to:

The live range of precolored registers should be as short as possible!

We can now see a problem with our previous calculation of def and use information: the above sequence to actually implement the function call will overwrite the argument registers `%edx`, `%esi`, `%edi` as well as the result register `%eax` (the lower 32bits of the return register `%rax`)! In fact, any of the argument registers, the result register, as well as `%r10` (temporary register for passing static function chain pointers) and `%r11` (temporary register) may not be preserved across function calls and therefore have to be considered to be *defined* by the call. If we represent this in the low-level intermediate language, we would *add* to the rule R_8 the following rule R_8':

$$\frac{l : d \leftarrow f(s_1, \ldots, s_n) \qquad \textsf{caller-save}(r)}{\textsf{def}(l, r)}\; J_8'$$

where $\textsf{caller-save}(r)$ is true of register r among `%rax`, `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`, `%r10`, and `%r11`.

Here we assume that register aliasing is handled correctly, that is, the register allocator understands that, for example, `%eax` constitutes the lower 32 bits of `%rax`.

Note that all *argument registers* and the *result register* are *caller-save*. This is justified by the fact that we often compute a value for the purposes of passing it into a function, but we do not require that value afterwards. Of course, the result register has to be caller-save, since it will be defined by the called function before it returns.

We refer to argument registers more abstractly as $arg_1, arg_2, \ldots, arg_6$ and ler_7 and ler_8 for the other two caller-save registers (even if they are not used for passing arguments to a function). We refer to the result register `%rax` as res_0.

Now if a temp t (except for d) is live after a function call, we have to add an edge connecting t with any of the fixed registers noted above, since the value of those registers are not preserved across a function call.

The other fixed use of argument registers is of course at the beginning of a function. Again, we should be careful to generate code that keeps the live ranges of precolored registers short. We can accomplish this by moving the argument registers into temps. Under some heuristics in register allocation and coalescing, these moves can sometimes be eliminated. A function $f(x, y, z)$ might then start with

$$\begin{array}{l} f: \\ \quad x \leftarrow arg_1 \\ \quad y \leftarrow arg_2 \\ \quad z \leftarrow arg_3 \end{array}$$

One more note: if it is possible that the function f is a function accepting a variable number of arguments, some additional considerations apply. For example, the low 8 bits of `%rax`, called `%al` hold the number of floating point arguments passed to the function. One therefore sometimes sees `xorl %eax,%eax` before a function call to define zero variable arguments.

7 Callee-Save Registers

The typical calling sequence above takes care of treating caller-save registers correctly. But what about callee-save registers, namely `%rbx`, `%rbp`, `%r12`, `%r13`, `%r14` and `%r15`? In compiling a function we are required that the generated code preserves all the callee-save registers. We generically refer to these registers as lee_i where $9 \leq i \leq 14$.

The standard approach is to save those that are needed onto the stack in the function prologue and restore them from the stack in the function epilogue, just before returning. Of course, saving and restoring them all is safe, but may be overkill for small functions that do not require many registers.

Remember that callee-save registers are essentially live throughout the body of a function, since their value at the return instruction matters. This violates our general rule to keep the live ranges of precolored registers short—in fact, they are maximal!

One simple way to deal with this is by listing them last among the registers to be assigned by register allocation. If we need more than the available number of caller-save registers, we assign callee-save registers before we resort to spilling, but make sure the save them at the beginning of a function and restore them at the end. This is generally more efficient than the usual register spilling since such temps still live in a register throughout the function execution. We use this technique in the example in Section 8.

Another solution is to let register allocation together with register coalescing do the job for us. We can move the contents of all the callee-save registers into temps at the beginning of a function and then move them back at the end. If it turns out these temps are spilled, then they will be saved onto the stack. If not, they may be moved from one register to another and then back at the end. However, this only works well with the right heuristics for assigning registers or using register coalescing.[2] Register coalescing consults the interference graph to check if

[2] One technique for register coalescing is briefly described in Section 8 of Lecture 3.

we can assign the same register for variable-to-variable moves. Another optimization that can eliminate register-to-register moves is copy propagation, covered in a later lecture. However, copy propogation requires care because it might extend the live range of variables, possibly undoing the care we applied to keep precolored registers contained.

With this technique, the general shape of the code for a function f before register allocation would be

$$
\begin{array}{l}
f: \\
\quad t_1 \leftarrow lee_9 \\
\quad t_2 \leftarrow lee_{10} \\
\quad \ldots \\
\quad \textit{function body} \\
\quad \ldots \\
\quad lee_{10} \leftarrow t_2 \\
\quad lee_9 \leftarrow t_1 \\
\quad \mathsf{ret}
\end{array}
$$

One complication with this approach is that we need to be sure to spill the full 64-bit registers, while registers holding 32-bit integer values might be saved and restored (or directly used as operands) using only 32 bits. Looking ahead, we see that we will need both 32 bit and 64 bit registers and spill slots in the next lab, so we might decide to introduce this complication now. Or we can still treat callee-save registers specially and switch over to a more uniform treatment in the next lab.

With either of the techniques for using callee-save registers, the one additional rule (R_8') is not enough. We should also note that all *callee-save* registers should be considered *live* at the return instruction.

$$
\frac{l : \mathsf{return}\ s \qquad \mathsf{callee\text{-}save}(r)}{\mathsf{use}(l, r)}\ J_2'
$$

We already know, by prior rule, that s itself is live at l. The rule new rule J_2' correctly flags all callee-save registers as live throughout the function body, unless they are assigned somewhere. The code pattern above achieves exactly that, cutting their live ranges down to a minimum.

8 An Extended Example

We use the recursive version of the power function as an example to illustrate register allocation in the presence of function calls. The C0 source is on the left; the abstract assembly on the right.

```
int pow(int b, int e)            pow(b,e):
```

```
//@requires e >= 0;
{
  if (e == 0)
    return 1;
  else
    return b * pow(b, e-1);
}
```

```
  if (e == 0) goto done
  t0 <- e - 1
  t1 <- pow(b, t0)
  t2 <- b * t1
  return t2
done:
  return 1
```

First, we convert it to SSA form. Looking at the right, we see it is already in static single assignment form! Looking on the left, we see a purely functional program. Since purely functional programs do not perform assignment, they must already be in SSA form!

Next, we perform liveness analysis. We proceed backward through the program to compute the following information.

program	live-in
$\mathsf{pow}(b, e)$:	
if $(e == 0)$ goto done	b, e
$t_0 \leftarrow e - 1$	b, e
$t_1 \leftarrow \mathsf{pow}(b, t_0)$	b, t_0
$t_2 \leftarrow b * t_1$	b, t_1
return t_2	t_2
done :	
return 1	

Next, we move to a slightly lower-lever representation, making the precolored registers explicit with the code pattern in Section 6.

program	live-in
pow :	
$b \leftarrow arg_1$	arg_1, arg_2
$e \leftarrow arg_2$	b, arg_2
if $(e == 0)$ goto done	b, e
$t_0 \leftarrow e - 1$	b, e
$arg_2 \leftarrow t_0$	b, t_0
$arg_1 \leftarrow b$	b, arg_2
call pow	b, arg_1, arg_2
$t_1 \leftarrow res_0$	b, res_0
$t_2 \leftarrow b * t_1$	b, t_1
$res_0 \leftarrow t_2$	t_2
return	res_0
done :	
$res_0 \leftarrow 1$	
return	res_0

We have not made any callee-save registers explicit yet, in the hope we will not need them. After all, there are only two variables and three temps in the program, but we have eight caller-save registers.

Next, we build the interference graph. For each line l and each temp t defined at l, we create an edge between t and any variable live in the successor. The only exception is a move $t \leftarrow s$, where we don't create and edge between t and s because they could be consistently be assigned to the same register. We find that only b interferes with other temps and precolored registers:

temp	interfering with
b	$res_0, arg_1, arg_2, t_0, t_1$
e	b
t_0	b
t_1	b
t_2	

Implicitly all precolored registers interfere with each other.

However, we forgot one important piece of information, namely that the call instruction must be interpreted as *defining all caller-save registers*. Since b remains alive through the function call, it can therefore not be assigned to a caller-save register, based on the code that we have.

We proceed by admitting that we need one caller-save register lee_9 and save and restore it at the beginning and end of the function. We use the push and pop instructions for the save and restore operations.

program	live-in
pow :	
push lee_9	arg_1, arg_2, lee_9
$b \leftarrow arg_1$	arg_1, arg_2
$e \leftarrow arg_2$	b, arg_2
if $(e == 0)$ goto done	b, e
$t_0 \leftarrow e - 1$	b, e
$arg_2 \leftarrow t_0$	b, t_0
$arg_1 \leftarrow b$	b, arg_2
call pow	b, arg_1, arg_2
$t_1 \leftarrow res_0$	b, res_0
$t_2 \leftarrow b * t_1$	b, t_1
$res_0 \leftarrow t_2$	t_2
pop lee_9	res_0
return	res_0, lee_9
done :	
$res_0 \leftarrow 1$	
pop lee_9	res_0
return	res_0, lee_9

While the callee-save $lee_{10}, \ldots, lee_{14}$ are still (implicitly) live through this function, after the rewrite lee_9 no longer is. Therefore, it no longer interferes with any temps.

We can construct a *simplicial elimination ordering*, from the interference graph, such as:

$$b, e, t_0, t_1, t_2$$

We order the colors (machine registers) as

$$res_0, arg_1, \ldots, arg_6, ler_7, ler_8, lee_9$$

with the idea that caller-save registers come first (including argument registers which we will likely need anyway), followed by the only callee-save register we are currently permitted to use. If we needed more, we would first have to spill and restore them.

From this we construct the assignment

$$\begin{array}{lcl} b & \mapsto & lee_9 \\ e & \mapsto & res_0 \\ t_0 & \mapsto & res_0 \\ t_1 & \mapsto & res_0 \\ t_2 & \mapsto & res_0 \end{array}$$

Applying the substitutions:

$$\begin{array}{ll}
\textsf{pow :} & \\
\quad \textsf{push } lee_9 & \\
\quad lee_9 \leftarrow arg_1 & \\
\quad res_0 \leftarrow arg_2 & \\
\quad \textsf{if } (res_0 \texttt{ == } 0) \textsf{ goto done} & \\
\quad res_0 \leftarrow res_0 - 1 & \\
\quad arg_2 \leftarrow res_0 & \\
\quad arg_1 \leftarrow lee_9 & \text{(redundant)} \\
\quad \textsf{call pow} & \\
\quad res_0 \leftarrow res_0 & \text{(redundant)} \\
\quad res_0 \leftarrow lee_9 * res_0 & \\
\quad res_0 \leftarrow res_0 & \text{(redundant)} \\
\quad \textsf{pop } lee_9 & \\
\quad \textsf{return} & \\
\textsf{done :} & \\
\quad res_0 \leftarrow 1 & \\
\quad \textsf{pop } lee_9 & \\
\quad \textsf{return} &
\end{array}$$

There are now some redundant instructions that can be eliminated. The self-moves are obvious, and one line becomes a self-move after copy propagation. One would also typically have just one epilog for the function (which restores the callee-save registers and the stack pointer, which is not visible here). Making these last changes, we obtain:

pow :
  push lee_9
  $lee_9 \leftarrow arg_1$
  $res_0 \leftarrow arg_2$
  if $(res_0 == 0)$ goto done
  $res_0 \leftarrow res_0 - 1$
  $arg_2 \leftarrow res_0$
  call pow
  $res_0 \leftarrow lee_9 * res_0$
  goto epilogue
done :
  $res_0 \leftarrow 1$
epilogue :
  pop lee_9
  return

Using GNU assembler format for x86-64:

```
pow:
       pushq   %rbx

       movl    %edi, %ebx
       movl    %esi, %eax
       testl   %eax, %eax
       je L1
       subl    $1, %eax
       movl    %eax, %esi
       call    pow
       imull   %ebx, %eax
       goto L2
L1:
       movl $1, %eax
L2:
       popq %rbx
       ret
```

References

[MHJM09] Michael Matz, Jan Hubička, Andreas Jaeger, and Mark Mitchell. System V application binary interface, AMD64 architecture processor supplement. Available at http://refspecs.linuxfoundation.org/elf/x86-64-abi-0.99.pdf, May 2009. Draft 0.99.