The Itanium processor, part 4: The Windows calling convention, leaf functions

devblogs.microsoft.com/oldnewthing/20150730-00

July 30, 2015

Raymond Chen

Last time, we looked at the general rules for parameter passing on the Itanium. But those rules are relaxed for leaf functions (functions which call no other functions).

Before we start, I need to correct some of the explanation I had given when introducing the calling convention. I used that explanation because it makes for an easier conceptual model, but the reality is slightly different.

First of all, I said that the `alloc` function shuffles the registers around and lays out the new local region and output registers. In reality, it is the `br.call` instruction that moves the registers and the `alloc` which sets up the register frame. Since the first instruction of a function is `alloc`, it doesn't make much difference how the work is distributed between the `br.call` and the `alloc` since they come right after each other. The only time you notice the difference is if you happen to break into the debugger immediately between those two instructions.

More precisely, here's what the `br.call` instruction does:

- Copy the current register frame state (and some other stuff) to the `pfs` register.
- Rotate the registers so that the first output register is now *r32*.
- Create a new register frame as follows:
  - input registers = caller's output registers
  - no local registers
  - no output registers
  - no rotating registers
- Set the *rp* register to the return address.
- Transfer control to the target.

In other words, the register stack changes like this:

r32	r33	r34	r35	r36	r37	r38	r39	r40	r41	r42	r43

f‘s Input	*f*‘s Local	*f*‘s Output	Before *f* does a `br.c`

	r32	r33	r34
On entry to *g*	*g*‘s Input		

	r32	r33	r34	r35	r36	r37	r38	r3
After *g* does an `alloc`	*g*‘s Input			*g*‘s Local				*g*‘

A consequence of this division of labor between `br.call` and `alloc` is that leaf functions can take advantage of this default register frame: If a leaf function can do all its work with just

- its input registers
- scratch registers
- the red zone

then it doesn’t need to perform an `alloc` at all! It can use the default register allocation of “Caller’s output registers become my input registers, and I have no local registers or output registers.” When finished, the function just does a `br.ret rp` to return to the caller.

Note that this optimization is available only to leaf functions. If the function calls another function, then the `br.call` will overwrite the `pfs` and `rp` registers, which will make it hard to return to your caller when you’re done.

The red zone is officially known as the *scratch area*. The first 16 bytes on the stack are available for use by the currently executed function. If you want values to be preserved across a function call, you need to move them out of the scratch area, because they become the scratch area for the function being called! In other words, the scratch area is not preserved across function calls.

A more obscure consequence of this division of labor between `br.call` and `alloc` is that a function could in principle perform `alloc` more than once in order to change the size of its local region or the number of output registers. For example, a function might start by saying, “I have five local registers and two output registers,” and then later realize, “Oh, wait, I need to call a function with six parameters. I will issue a new `alloc` instruction that requests five local registers and *six* output registers.” Although technically legal, it doesn’t often occur in practice because it’s usually easier just to set up your register state once and stick with it for the duration of the function.

A more common case where this occurs is when a function has an early exit that can be determined using only leaf-available resources.

```
extern HANDLE LogFile;

void Log(char *message, char *file, int line)
{
 if (!LogFile) return;
 ... complicated logging code goes here ...
}
```

If profiling feedback indicates that logging is rarely enabled, then the compiler can avoid setting up all the registers and stack for the complicated logging code, at least until it knows that logging is enabled.

```
.Log:
      addl    r30, -205584, gp ;; // get address of LogFile variable
      ld8     r30, [r30] ;;       // fetch the value
      cmp.eq  p6, p0 = r30, r0    // is it zero?
(p6)  br.ret  rp                  // return if so

  // Okay, we are really logging. Set up our stack.
      alloc   r35 = ar.pfs, 3, 5, 6, 0 // set up register frame
      sub     sp = sp, 0x240      // set up stack buffers
      mov     r36 = ra            // save return address

      ... do complicated logging ...

      mov     rp = r36            // return address
      mov.i   ar.pfs = r34        // restore pfs
      br.ret.sptk.many  rp ;;     // return to caller
```

The first instruction calculates the effective address of the `LogFile` variable. We'll learn more about the *gp* register later.

The second instruction loads an 8-byte value from that address, thereby retrieving the value of `LogFile`.

The third instruction compares the value against *r0*, which is a hard-coded zero register. It asks for an equality comparison, putting the answer in the predicate variable *p6* (and putting the complement of the answer in *p0*, which effectively throws it away).

The fourth instruction conditionally returns from the function if the comparison was true. In the common case where logging is not enabled, the function returns at this point. Only if logging is enabled do the `alloc` and related instructions execute to set up the stack frame and then perform the complicated logging.

This is an example of an optimization known as *shrink-wrapping*. Shrink-wrapping occurs when a function does some work with a temporary stack frame, and then expands to a full stack frame only if it is needed. (Shrink-wrapping entails a few extra entries in the unwind exception table because the unwinding needs to take place differently depending on where in the function the exception occurred. I'll spare you the details.)

Okay, that's all for leaf functions and getting to the bottom of the whole `br.call` / `alloc` dance. Next time, we'll look at function pointers and the funky *gp* register. Here's something to whet your appetite: On ia64, a function pointer is not the address of the first instruction in the function's code. In fact, it's nowhere near the function's code.

Raymond Chen

Follow