The history of calling conventions, part 4: ia64

devblogs.microsoft.com/oldnewthing/20040113-00

January 13, 2004

Raymond Chen

The ia-64 architecture (Itanium) and the AMD64 architecture (AMD64) are comparatively new, so it is unlikely that many of you have had to deal with their calling conventions, but I include them in this series because, who knows, you may end up buying one someday.

Intel provides the Intel® Itanium® Architecture Software Developer's Manual which you can read to get extraordinarily detailed information on the instruction set and processor architecture. I'm going to describe just enough to explain the calling convention.

The Itanium has 128 integer registers, 32 of which (r0 through r31) are global and do not participate in function calls. The function declares to the processor how many registers of the remaining 96 it wants to use for purely local use ("local region"), the first few of which are used for parameter passing, and how many are used to pass parameters to other functions ("output registers").

For example, suppose a function takes two parameters, requires four registers for local variables, and calls a function that takes three parameters. (If it calls more than one function, take the maximum number of parameters used by any called function.) It would then declare at function entry that it wants six registers in its local region (numbered r32 through r37) and three output registers (numbered r38, r39 and r40). Registers r41 through r127 are off-limits.

Note to pedants: This isn't actually how it works, I know. But it's much easier to explain this way.

When the function wants to call that child function, it puts the first parameter in r38, the second in r39, the third in r40, then calls the function. The processor shifts the caller's output registers so they can act as the input registers for the called function. In this case r38 moves to r32, r39 moves to r33 and r40 moves to r34. The old registers r32 through r38 are saved in a separated register stack, different from the "stack" pointed to by the sp register. (In reality, of course, these "spills" are deferred, in the same way that SPARC register windows don't spill until needed. Actually, you can look at the whole ia64 parameter passing convention as the same as SPARC register windows, just with variable-sized windows!)

When the called function returns, the register then move back to their previous position and the original values of r32 through r38 are restored from the register stack.

This creates some surprising answers to the traditional questions about calling conventions.

What registers are preserved across calls? Everything in your local region (since it is automatically pushed and popped by the processor).

What registers contain parameters? Well, they go into the output registers of the caller, which vary depending on how many registers the caller needs in its local region, but the callee always sees them as r32, r33, etc.

Who cleans the parameters from the stack? Nobody. The parameters aren't on the stack to begin with.

What register contains the return value? Well that's kind of tricky. Since the caller's registers aren't accessible from the called function, you'd think that it would be impossible to pass a value back! That's where the 32 global registers come in. One of the global registers (r8, as I recall) is nominated as the "return value register". Since global registers don't participate in the register window magic, a value stored there stays there across the function call transition and the function return transition.

The return address is typically stored in one of the registers in the local region. This has the neat side-effect that **a buffer overflow of a stack variable cannot overwrite a return address** since the return address isn't kept on the stack in the first place. It's kept in the local region, which gets spilled onto the register stack, a chunk of memory separate from the stack.

A function is free to subtract from the sp register to create temporary stack space (for string buffers, for example), which it of course must clean up before returning.

One curious detail of the stack convention is that the first 16 bytes on the stack (the first two quadwords) are always scratch. (Peter Lund calls it a "red zone".) So if you need some memory for a short time, you can just use the memory at the top of the stack without asking for permission. But remember that if you call out to another function, then that memory becomes scratch for the function you called! So if you need the value of this "free scratchpad" preserved across a call, you need to subtract from sp to reserve it officially.

One more curious detail about the ia64: A function pointer on the ia64 does not point to the first byte of code. Instead, it points to a structure that describes the function. The first quadword in the structure is the address of the first byte of code, and the second quadword contains the value of the so-called "gp" register. We'll learn more about the gp register in a later blog entry.

(This “function pointer actually points to a structure” trick is not new to the ia64. It’s common on RISC machines. I believe the PPC used it, too.)

Okay, this was a really boring entry, I admit. But believe it or not, I’m going to come back to a few points in this entry, so it won’t have been for naught.

Raymond Chen

Follow