The Itanium processor, part 3: The Windows calling convention, how parameters are passed

devblogs.microsoft.com/oldnewthing/20150729-00

July 29, 2015

Raymond Chen

The calling convention on Itanium uses a variable-sized register window. The mechanism by which this is done is rather complicated, so I'm first going to present a conceptual version, and then I'll come back and fix up some of the implementation details. For today, I'm just going to talk about how parameters are passed. There are other aspects of the calling convention that I will cover in separate articles.

Recall that the first 32 registers *r0* through *r31* are static (do not change), and the remaining registers *r32* through *r127* are stacked. These stacked registers fall into three categories: *input registers*, *local registers*, and *output registers*.

The input registers receive the function parameters. On entry to a function, the function's parameters are received in registers starting at *r32* and increasing. For example, a function that takes two parameters receives the first parameter in *r32* and the second parameter in *r33*.

Immediately after the input registers are the registers for the function's private use. These are known as *local registers*. For example, if that function with two parameters also wants four registers for private use, those private registers would be *r34* through *r37*.

After the input registers are the registers used to call other functions, known as *output registers*.[1] For example, if the function with two parameters and four local registers wants to call a function that has three parameters, it would put those parameters in registers *r38* through *r40*. Therefore, a function needs as many output registers as the maximum number of parameters of any function it calls.

The input registers and local registers are collectively known as the *local region*. The input registers, local registers, and output registers are collectively known as the *register frame*.

Any registers higher than the last output register are off-limits to the function, and we shall henceforth pretend they do not exist. Since the registers go up to *r127*, and in practice register frames are around one or two dozen registers, there end up being a lot of registers that go unused.

The first thing a function does is notify the processor of its intended register usage. It uses the `alloc` instruction to say how many input registers, local registers, and output registers it needs.

```
alloc r35 = ar.pfs, 2, 4, 3, 0
```

This means, “Set up my register frame as follows: Two input registers, four local registers, three output registers, and no rotating registers. Put the previous register frame state (*pfs*) in register *r35*.”

The second thing a function does is save the return address, typically in one of the local registers it just created. For example, the above `alloc` might be followed by

```
mov r34 = rp
```

On entry to a function, the *rp* register contains the caller’s return address, and most of the time, the compiler will save the return address in a register. Note that this means that on the Itanium, a stack buffer overrun will never overwrite a return address, since return addresses are not kept on the stack. (Let that sink in. On Itanium, return addresses *are not kept on the stack*. This means that tricks like `_AddressOfReturnAddress` will not work!)

By convention, the *rp* and *ar.pfs* are saved in consecutive registers (here, *r34* and *r35*). This convention makes exception unwinding slightly easier.

Let’s see what happens when somebody calls this function. Suppose the caller’s register frame looks like this:

static						local region					output				
						input			local						
r0	*r1*	…	*r30*	*r31*		*r32*	*r33*	*r34*	*r35*	*r36*	*r37*	*r38*	*r39*	*r40*	*r41*

The caller places the parameters to our function in its output registers, in this case *r37* and *r38*. (Our function takes only two parameters, so *r39* and beyond are not used.)

static						local region					output				
						input			local						
r0	*r1*	…	*r30*	*r31*		*r32*	*r33*	*r34*	*r35*	*r36*	*r37*	*r38*	*r39*	*r40*	*r41*
0	A	…	F	G		H	I	J	K	L	M	N	?	?	?

The caller then invokes our function.

Our function opens by performing this `alloc`, declaring two input registers, four local registers, and three output registers.

```
alloc r35 = ar.pfs, 2, 4, 3, 0
```

That `alloc` instruction shuffles the registers like this:

- The static registers don’t change.
- The registers in the caller’s local region are saved in a magic place.
- The specified number of output registers from the caller become the new function’s input registers.
- New local and output registers are created but left uninitialized.
- The previous function state is placed in the specified register (for restoration at function exit). There are many parts of the function state, but the part we care about is the frame state, which describes how registers are assigned.

Here’s what the register frame looks like after all but the last steps above:

static					local region						output		
					input		local						
r0	*r1*	…	*r30*	*r31*	*r32*	*r33*	*r34*	*r35*	*r36*	*r37*	*r38*	*r39*	*r40*
0	A	…	F	G	M	N	?	?	?	?	?	?	?
unchanged					moved		uninitialized						

The last step (storing the previous function state in the specified register) updates the *r35* register:

static					local region						output		
					input		local						
r0	*r1*	…	*r30*	*r31*	*r32*	*r33*	*r34*	*r35*	*r36*	*r37*	*r38*	*r39*	*r40*
0	A	…	F	G	M	N	?	pfs	?	?	?	?	?

The next instruction is typically one to save the return address.

```
mov r34 = rp
```

After that `mov` instruction, the function prologue is complete, and the register state looks like this:

static					local region						output		
					input		local						
r0	*r1*	…	*r30*	*r31*	*r32*	*r33*	*r34*	*r35*	*r36*	*r37*	*r38*	*r39*	*r40*
0	A	…	F	G	M	N	ra	pfs	?	?	?	?	?

where `ra` is the function's return address.

At this point the function runs and does actual work. Once it's done, its register state might look like this:

static					local region						output		
					input		local						
r0	*r1*	…	*r30*	*r31*	*r32*	*r33*	*r34*	*r35*	*r36*	*r37*	*r38*	*r39*	*r40*
0	A′	…	F′	G′	T	U	ra	pfs	V	W	X	Y	Z

The function epilogue typically consists of three instructions:

```
mov rp = r34     // prepare to return to caller
mov ar.pfs = r35 // restore previous function state
br.ret rp        // return!
```

This sequence begins by copying the saved return address into the *rp* register so that we can jump back to it. (We could have copied *r34* into any scratch branch register, but by convention we use the *rp* register because it makes exception unwinding easier.)

Next, it restores the register state from the *pfs* it saved at function entry. Finally, it transfers control back to the caller by jumping through the *rp* register. (We cannot do a `br.ret r34` because `r34` is not a branch register; the parameter to `br.ret` must be a branch register.)

Restoring the previous function state causes the caller's register frame layout to be restored, and the values of the registers in the caller's local region are restored from that magic place.

The register state upon return back to the caller looks like this:

static	local region	output

						input			local						
r0	*r1*	…	*r30*	*r31*		*r32*	*r33*	*r34*	*r35*	*r36*	*r37*	*r38*	*r39*	*r40*	*r41*
0	A′	…	F′	G′		H	I	J	K	L	?	?	?	?	?
unchanged						restored					uninitialized				

From the point of view of the calling function, calling another function has the following effect:

- Static registers are shared with the called function. (Any changes to static registers are visible to the caller.)
- The local region is preserved across the call.
- The output registers are trashed by the call.

At most eight parameters are passed in registers. Any additional parameters are passed on the stack, and it is the caller's responsibility to clean them up. (The stack-based parameters begin *after the red zone*. We'll talk more about the red zone later.)

Thank goodness for the parameter cap, because a variadic function doesn't know how many parameters were passed, so it would otherwise not know how many input parameters to declare in its `alloc` instruction. The parameter cap means that variadic functions `alloc` eight input registers, and typically the first thing they do is spill them onto the stack so that they are contiguous with any parameters beyond 8 (if any). Note that this spilling must be done very carefully to avoid crashing if the corresponding register does not correspond to an actual parameter but happens to be a NaT left over from a failed speculative execution. (There is a special instruction for spilling without taking a NaT consumption exception.)

If any parameter is smaller than 64 bits, then the unused bits of the corresponding register are garbage and should be ignored. I didn't discuss floating point parameters or aggregates. You can read Thiago's comment for a quick version, or dig into the *Itanium Software Conventions and Runtime Architecture Guide* (Section 8.5: Parameter Passing) for gory details.

Okay, that's the conceptual model. The actual implementation is not quite as I described it, but the conceptual model is good enough for most debugging purposes. Here are some of the implementation details which will come in handy if you need to roll up your sleeves.

First of all, the processor does not actually distinguish between input registers and local registers. It only cares about the local region. In other words, the parameters to the `alloc` instruction are

- Size of local region.
- Number of output registers.

- Number of rotating registers.
- Register to receive previous function state.

When the called function established its register frame, the processor just takes all the caller's output registers (even the ones that aren't actually relevant to the function call) and slides them down to *r32*. It is the compiler's responsibility to ensure that the code passes the correct number of parameters. Therefore, our diagram of the function call process would more accurately go like this: The caller's register frame looks like this before the call:

static					local region					output				
					input			local						
r0	*r1*	…	*r30*	*r31*	*r32*	*r33*	*r34*	*r35*	*r36*	*r37*	*r38*	*r39*	*r40*	*r41*
0	A	…	F	G	H	I	J	K	L	M	N	X_1	X_2	X_3

where the X values are whatever garbage values happen to be left over from previous computations, possibly even NaT.

When the called function sets up its register frame (before storing the previous register frame), it gets this:

static					local region						output		
					input		local						
r0	*r1*	…	*r30*	*r31*	*r32*	*r33*	*r34*	*r35*	*r36*	*r37*	*r38*	*r39*	*r40*
0	A	…	F	G	M	N	X_1	X_2	X_3	?	?	?	?
unchanged					moved		uninitialized						

The processor took all the output registers from the caller and slid them down to *r32* through *r36*.

Of course, the called function shouldn't try to read from any registers beyond *r33*, if it knows what's good for it, because those registers contain nothing of value and may indeed be poisoned by a NaT.

This little implementation detail has no practical consequences because those registers were uninitialized in the conceptual model anyway. But it does mean that when you disassemble the `alloc` instruction, you'll see that the distinction between input registers and local registers has been lost, and that both sets of registers are reported as input registers. In other words, an instruction written as

```
alloc r34 = ar.pfs, 2, 4, 3, 0
```

disassembles as

```
alloc r34 = ar.pfs, 6, 0, 3, 0
```

The disassembler doesn't know how many of the six registers in the input region are input registers and how many are local, so it just treats them all as input registers.

That explains some of the undefined registers, but what about those question marks? To solve this riddle, we need to answer a different question first: "Where is this magic place that the caller's local region gets saved to and restored from?"

This is where the infamous Itanium second stack comes into play.

There are two stacks on Itanium. One is indexed by the *sp* register and is what one generally means when one says *the stack*. The other stack is indexed by the *bsp* register (*backing store pointer*), and it is the magic place where these "registers from long ago" are saved. The *bsp* register grows *upward* in memory (toward higher addresses), opposite from the *sp* which grows downward (toward lower addresses). Windows allocates the two stacks right next to each other, Here's an artistic impression by Slava Oks. Bear in mind that Slava drew the diagram upside-down (low addresses at the top, high addresses at the bottom). The *bsp* grows toward toward higher addresses, but in Slava's diagram, that direction is downward.

One curious implementation detail is that the two stacks abut each other without a gap. I'm told that the kernel team considered putting a no-access page between the two stacks, so that a runaway memory copy into the stack would encounter an access violation before it reached the backing store. For whatever reason, they didn't bother.

Now, the processor is sneaky and doesn't actually push the values onto the backing store immediately. Instead, the processor rotates them into high-numbered unused registers (all the registers beyond the last output register), and only when it runs out of space there does it spill them into the backing store. When the function returns, the rotation is undone, and the values squirreled away into the high-numbered unused registers magically reappear in the caller's local region.

Each time a function is called, the registers rotate to the left, and when a function returns, the registers rotate to the right. As a result, the local regions of functions in the call stack can be found among the off-limits registers, up until we reach the last spill point.

Suppose the call stack looks like this (most recent function at the top):

```
a() -- current function
b()
c()
d()
e()
f()
g()
```

If we zoom out, we can see all those local regions.

static		a		open	g	f	e	d	c	b
		LR	O		LR	LR	LR	LR	LR	LR
••••••		•••••	•••	••••••••••••••	•••••	•••••	••••••	••••••	••••	••••••

Why don’t we see any output registers for any functions other than the current one? You know why: Because at each function call, the caller’s output registers become the called function’s input registers. If you really wanted to draw the output registers, you could do it like this, where each function’s input registers is shared with the caller’s output registers.

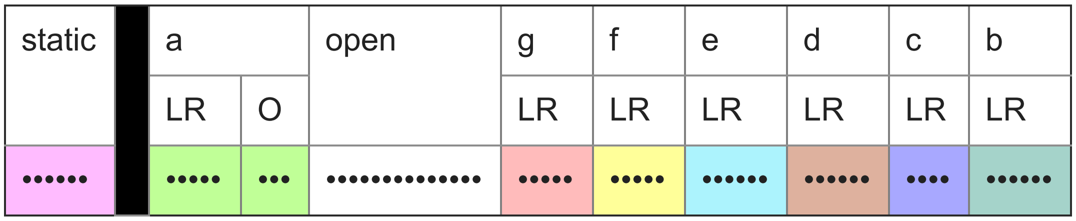
static		a			open	g				e				c			
		I	L	O		I	L	O		I	L	O		I	L	O	
••••••		••	•••	•••	••••••••••••••	••	•••	••	•••	•••	•••	•••	•••	••	••	•••	•••
		O						I	L	O		I	L	O		I	L
		b						f				d				b	

But we won’t bother drawing this exploded view any more.

Now, if the function `a` calls another function `x`, then all the registers rotate left, with `a`‘s local region wrapping around to the end of the list:

static		x		open	g	f	e	d	c	b	a
		LR	O		LR	LR	LR	LR	LR	LR	LR
••••••		•••	••••	••••••••••	•••••	•••••	••••••	••••••	••••	••••••	•••••

And when `x` returns, the registers rotate right, bringing us back to



Note that the conceptual model doesn't care about this implementation detail. In theory, future versions of the Itanium processor might have additional "bonus registers" after *r127* which are programmatically inaccessible but which are used to expand the number of register frames that can be held before needing to spill.

With this additional information, you now can see the contents of those undefined registers on entry to a function: They contain whatever garbage happened to be left over in the open registers. Similarly, the contents of those undefined output registers after the function returns to its caller are the leftover values in the called function's local region.

You can also see the contents of the uninitialized output registers on return from a function: They contain whatever garbage happened to be left over in the called function's input registers. This behavior is actually documented by the processor, so in theory somebody could invent a calling convention where information is passed from a function back to its caller through the input registers, say, for a language that supports functions with multiple return values. (In other words, the input registers are actually in/out registers.) The Windows calling convention doesn't use this feature, however.

It so happens that the debugger forces a full spill into the backing store when it gains control. This is useful, because groveling into the backing store gives you a way to see the local regions of any function on the stack.

```
kd> r
...
      r32 =      6fbffd21130 0        r33 =          1170065 0
      r34 =      6fbffd23700 0        r35 =                8 0
      r36 =      6fbffd21338 0        r37 =            20000 0
      r38 =             8000 0        r39 =             2000 0
      r40 =              800 0        r41 =              400 0
      r42 =              100 0        r43 =               80 0
      r44 =              200 0        r45 =            10000 0
      r46 =          7546fdf0 0        r47 = c000000000000693 0
      r48 =             5041 0        r49 =         75ab0000 0
      r50 =      6fbffd21130 0        r51 =          1170065 0
      r52 =      6fbfc79f770 0        r53 =         7546cbe0 0
kd> dq @bsp
000006fb`fc7a02e0  000006fb`ffd21130 00000000`01170065 // r32 and r33
000006fb`fc7a02f0  000006fb`ffd23700 00000000`00000008 // r34 and r35
000006fb`fc7a0300  000006fb`ffd21338 00000000`00020000 // r36 and r37
000006fb`fc7a0310  00000000`00008000 00000000`00002000 // r38 and r39
000006fb`fc7a0320  00000000`00000800 00000000`00000400 // r40 and r41
000006fb`fc7a0330  00000000`00000100 00000000`00000080 // r42 and r43
000006fb`fc7a0340  00000000`00000200 00000000`00010000 // r44 and r45
000006fb`fc7a0350  00000000`7546fdf0 c0000000`00000693 // r46 and r47
```

But wait, ia64 integer registers are 65 bits wide, not 64. The extra bit is the NaT bit. Where did that go?

Whenever the *bsp* hits a 512-byte boundary (*bsp* & 0x1F8 == 0x1F8, or after 63 registers have been spilled), the value spilled into the backing store is not a 64-bit register but rather the accumulated NaT bits. You are not normally interested in the NaT bits, so the only practical consequence of this is that you have to remember to skip an entry whenever you hit a 512-byte boundary.

Suppose we wanted to look at our caller's local region. Here's the start of a sample function. Don't worry about most of the instructions, just pay attention to the `alloc` and the `mov ... = rp`.

```
SAMPLE!.Sample:
       alloc    r47 = ar.pfs, 013h, 00h, 04h, 00h
       mov      r48 = pr
       addl     r31 = -2004312, gp
       adds     sp = -1072, sp ;;
       ld8.nta  r3 = [sp]
       mov      r46 = rp
       adds     r36 = 0208h, r32
       or       r49 = gp, r0 ;;
```

Suppose you hit a breakpoint partway through this function, and you want to know why the caller passed a strange value for the first input parameter *r32*.

From reading the function prologue, you see that the return address is kept in *r46*, so you can disassemble there to see how your caller set up its output parameters:

```
kd> u @r46-20
SAMPLE!.Caller+2bd0:
       ld8      r47 = [r32]
       ld4      r46 = [r33]
       or       r45 = r35, r0
       nop.b    00h
       nop.b    00h
       br.call.sptk.many  rp = SAMPLE!.Sample
```

(Notice the `nop` instructions which suggest that this is unoptimized code.)

But we don't know which of those registers are the output registers of the caller. For that, we need to know the register frame of the caller. We see from the `alloc` instruction that the previous function state (`pfs`) was saved in the *r47* register.

```
kd> ?@r47
Evaluate expression: -4611686018427386221 = c0000000`00000693
```

This value is not easy to parse. The bottom seven bits record the total size of the caller's register frame, which includes both the local region and the output registers. The size of the local region is kept in bits 7 through 13, which is a bit tricky to extract by eye. You take the third and fourth digits from the right, double the value, and add one more if the second digit from the right is 8 or higher. This is easier to do than to explain:

- The third- and fourth-to-last digits are `06` hex.
- Double that, and you get 12 (decimal).
- Since the second-to-last digit is 9, add one more.
- Result: 13.

The previous function's local region has 13 registers. Therefore, the previous function's output registers begin at 32 + 13 = 45. (You can also see that the previous function had 0x13 = 19 registers in its register frame, and you can therefore infer that it had 19 − 13 = 6 output registers.)

Applying this information to the disassembly of the caller, we see that the caller passed

- first output register *r45* = *r35*. (Recall that the *r0* register is always zero, so or'ing it with another value just copies that other value.)
- second output register *r46* = 4-byte value stored at [*r33*]
- third output register *r47* = 8-byte value stored at [*r32*]

That first output register was a copy of the *r35* register. We can grovel through the backing store to see what that value is.

```
0:000> dq @bsp-0n13*8 l4
000006fb`ffe906d8  00000000`4b1e9720 00000000`4b1ea2e8     // r32 and r33
000006fb`ffe906e8  00000000`0114a7c0 000006fb`fe728cac     // r34 and r35
```

And now we have extracted the registers from our caller's local region. Specifically, we see that the caller's *r35* is `` 000006fb`fe728cac ``.

We can extend this technique to grovel even further back in the stack. To do that, we need to obtain the *pfs* chain so we can see the structure of the register frame for each function in the call stack.

From the disassembly above, we saw that the caller was kept in *r46*. To go back another level, we need to find that caller's caller. We merely repeat the exercise, but with the caller. Sometimes it can be hard to find the start of a function (especially if you don't have symbols); it can be easier to look for the *end* of the function instead! Instead of looking for the `alloc` and `mov ... = rp` instructions which save the previous function state and return address, we look for the `mov ar.pfs = ...` and `mov rp = ...` instructions which restore them.

Here's an example of a stack trace I had to reconstruct:

```
0:000> u
00000000`4b17e9d4       mov      rp = r37              // return address
00000000`4b17e9e4       mov.i    ar.pfs = r38          // restore pfs
00000000`4b17e9e8       br.ret.sptk.many  rp ;;        // return to caller
0:000> dq @bsp
000006fb`ffe90758  000006fb`fe761cc0 000006fb`ffe8f860 // r32 and r33
000006fb`ffe90768  000006fb`ffe8fa70 00000000`00000104 // r34 and r35
000006fb`ffe90778  00000000`0114a7c0 00000000`4b1b6890 // r36 and r37
000006fb`ffe90788  c0000000`0000050e 00000000`00005001 // r38 and r39
```

Double the `05` to get 10 (decimal), and don't add one since the next digit (`0`) is less than 8. The previous function therefore has 10 registers in its local region.

The current function's return address is kept in *r37* and the *pfs* in *r38*. I've highlighted them in the *bsp* dump.

Let's disassemble at the return address and dump that function's local variables, thereby walking back one level in the call stack.

```
0:000> u 00000000`4b1b6890
...
00000000`4b1b6bd4       mov      rp = r38 ;;           // return address
00000000`4b1b6be4       mov.i    ar.pfs = r39          // restore pfs
00000000`4b1b6be8       br.ret.sptk.many  rp ;;
// we calculated that the local region of the previous function is size 0xA
0:000> dq @bsp-a*8 la
000006fb`ffe90708  000006fb`fe73bfc0 000006fb`fe73ff10     // r32 and r33
000006fb`ffe90718  00000000`00000000 000006fb`ffe8f850     // r34 and r35
000006fb`ffe90728  000006fb`ffe8f858 00000000`00000000     // r36 and r37
000006fb`ffe90738  00000000`4b1e9350 c0000000`00000308     // r38 and r39
000006fb`ffe90748  00000000`00009001 00000000`4b57e000     // r40 and r41
```

By studying the value in the caller's *r39*, we see that the caller's caller has 3 × 2 + 0 = 6 registers in its local region. And the caller's *r38* gives us the return address. Let's walk back another frame in the call stack.

```
0:000> u 4b1e9350
...
00000000`4b1e9354       mov      rp = r34              // return address
00000000`4b1e9368       mov.i    ar.pfs = r35          // restore pfs
00000000`4b1e9378       br.ret.sptk.many  rp ;;
0:000> dq @bsp-a*8-6*8 l6
000006fb`ffe906d8  00000000`0114a7c0 000006fb`fe728cac     // r32 and r33
000006fb`ffe906e8  00000000`4b1e9720 c0000000`00000389     // r34 and r35
000006fb`ffe906f8  00000000`00009001 00000000`4b57e000     // r36 and r37
```

This time, the return address is in *r34* and the previous *pfs* is in *r35*. This time, the caller's caller's caller has 3 × 2 + 1 = 7 registers in its local region.

```
0:000> u 4b1e9720
...
00000000`4b1e9784       mov      rp = r35             // return address
00000000`4b1e9788       adds     sp = 010h, sp ;;
00000000`4b1e9790       nop.m    00h
00000000`4b1e9794       mov      pr = r37, -2 ;;
00000000`4b1e9798       mov.i    ar.pfs = r36         // restore pfs
00000000`4b1e97a0       nop.m    00h
00000000`4b1e97a4       nop.f    00h
00000000`4b1e97a8       br.ret.sptk.many  rp ;;
0:000> dq @bsp-a*8-6*8-7*8 l7
000006fb`ffe906a0  00000000`0114a7c0 00000000`00000000    // r32 and r33
000006fb`ffe906b0  00000000`0114a900 00000000`4b19ba00    // r34 and r35
000006fb`ffe906c0  c0000000`0000058f 00000000`00009001    // r36 and r37
000006fb`ffe906d0  00000000`4b57e000                      // r38
```

This function also allocates 0x10 bytes from the stack, so if you want to see its stack variables, you can dump the values at *sp + 0x10* for length 0x10. The `+ 0x10` is to skip over the red zone.

Anyway, that's the way to reconstruct the call stack on an Itanium. Repeat until bored.

Maybe you can spot the fast one I pulled when discussing how the `alloc` instruction and *pfs* register work. More details next time, when we discuss leaf functions and the red zone.

Bonus chapter: How does spilling actually work?

[1] When not preparing to call another function, the output registers can be used for any purpose, with the understanding that the values will not be preserved across a function call.

Raymond Chen

Follow