ia64 – misdeclaring near and far data

devblogs.microsoft.com/oldnewthing/20040120-00

January 20, 2004

Raymond Chen

As I mentioned yesterday, the ia64 is a very demanding architecture. Today I'll discuss another way that lying to the compiler will come back and bite you.

The ia64 does not have an absolute addressing mode. Instead, you access your global variables through the r1 register, nicknamed "gp" (global pointer). This register always points to your global variables. For example, if you had three global variables, one of them might be kept at [gp+0], the second at [gp+8] and the third at [gp+16].

(I believe the Win32 MIPS calling convention also used this technique.)

On the ia64, there is a limitation in the "addl" instruction: You can only add constants up to 22 bits, which comes out to 4MB. So you can have only 4MB of global variables.

Well, it turns out that some people want more than 4MB of global variables. Fortunately, these people don't have one million DWORD variables. Rather, they have a few really big global arrays.

The ia64 compiler solves this problem by splitting global variables into two categories, "small" and "large". (The boundary between "small" and "large" can be set by a compiler flag. I believe the default is to treat anything larger than 8 bytes as "large".)

The code to access a "small" variable goes like this:

```
addl    r30 = -205584, gp;; // r30 -> global variable
ld4     r30 = [r30]         // load a DWORD from the global variable
```

(The gp register actually points into the middle of your global variables, so that both positive and negative offsets can be used. In this case, the variable happened to live at a negative offset from gp.)

By comparison, "large" global variables are accessed through a two-step process. First, the variable itself is allocated in a separate section of the file. Second, a pointer to the variable is placed into the "small" globals variables section of the module. As a result, accessing a "large" global variable requires an added level of indirection.

```
        addl    r30 = -205584, gp;; // r30 -> global variable forwarder
        ld8     r30 = [r30];;       // r30 -> global variable
        ld4     r30 = [r30]         // load a DWORD from the global variable
```

If you leave the size of an object unspecified, like

```
extern BYTE b[];
```

then the compiler plays it safe and assumes the variable is large. If it turns out that the variable is small, the forwarder pointer will still be there, and the code will do the double-indirection to fetch something that could have been accessed with a single indirection. The code is slightly less efficient, but at least it still works.

On the other hand, if you misdeclare the object as being small when it is actually large, then you end up in trouble. For example, if you write

```
    extern BYTE b;
```

in one file, and

```
    extern BYTE b[256];
```

in another, then files that include the first header will think the object is small and generate "small" code to access it, but files that include the second header will think it is large. And if the object turns out to be large after all, the code that used the first header file will fail pretty spectacularly.

So don't do that. When you declare a variable, make sure to declare it accurately. Otherwise the ia64 will catch you in a lie, and you will pay.

Raymond Chen

Follow