Although the x64 calling convention reserves spill space for parameters, you don't have to use them as such

devblogs.microsoft.com/oldnewthing/20110302-00

March 2, 2011

Raymond Chen

Although the x64 calling convention reserves space on the stack as spill locations for the first four parameters (passed in registers), there is no requirement that the spill locations actually be used for spilling. They're just 32 bytes of memory available for scratch use by the function being called.

> We have a test program that works okay when optimizations are disabled, but when compiled with full optimizations, everything appears to be wrong right off the bat. It doesn't get the correct values for `argc` and `argv`:
>
> ```
> int __cdecl
> wmain( int argc, WCHAR** argv ) { ... }
> ```
>
> With optimizations disabled, the code is generated correctly:
>
> ```
>         mov         [rsp+10h],rdx  // argv
>         mov         [rsp+8],ecx    // argc
>         sub         rsp,158h       // local variables
>         mov         [rsp+130h],0FFFFFFFFFFFFFFFEh
>         ...
> ```
>
> But when we compile with optimizations, everything is completely messed up:
>
> ```
>         mov         rax,rsp
>         push        rsi
>         push        rdi
>         push        r13
>         sub         rsp,0E0h
>         mov         qword ptr [rsp+78h],0FFFFFFFFFFFFFFFEh
>         mov         [rax+8],rbx    // ??? should be ecx (argc)
>         mov         [rax+10h],rbp  // ??? should be edx (argv)
> ```

When compiler optimizations are disabled, the Visual C++ x64 compiler will spill all register parameters into their corresponding slots. This has as a nice side effect that debugging is a little easier, but really it's just because you disabled optimizations, so the compiler generates simple, straightforward code, making no attempts to be clever.

When optimizations are enabled, then the compiler becomes more aggressive about removing redundant operations and using memory for multiple purposes when variable lifetimes don't overlap. If it finds that it doesn't need to save `argc` into memory (maybe it puts it into a register), then the spill slot for `argc` can be used for something else; in this case, it's being used to preserve the value of `rbx`.

You see the same thing even in x86 code, where the memory used to pass parameters can be re-used for other purposes once the value of the parameter is no longer needed in memory. (The compiler might load the value into a register and use the value from the register for the remainder of the function, at which point the memory used to hold the parameter becomes unused and can be redeployed for some other purpose.)

Whatever problem you're having with your test program, there is nothing obviously wrong with the code generation provided in the purported defect report. The problem lies elsewhere. (And it's probably somewhere in your program. Don't immediately assume that the reason for your problem is a compiler bug.)

Bonus chatter: In a (sadly rare) follow-up, the customer confessed that the problem was indeed in their program. They put a function call inside an `assert`, and in the nondebug build, they disabled assertions (by passing `/DNDEBUG` to the compiler), which means that in the nondebug build, the function was never called.

Extra reading: Challenges of debugging optimized x64 code. That `.frame /r` command is real time-saver.

Raymond Chen

Follow