repz ret

2012-09-05 | Ahmed Bougacha

Recently, we began work on a binary static analysis tool.
While reading some disassembly, we came across a weird-looking instruction, that was present in most everything we gave `objdump`.

```
f3 c3       repz ret
```

Now in the IA-32 (x86/x86-64) instruction set, `repz` is a prefix used to repeat the following instruction until `rcx` (or ecx, or cx) is 0.

There are two variants that add a termination condition depending on the value of the zero flag: one that stops when it becomes set (`repne/repnz`), and one that stops when it no longer is set: `repe/repz`;

The IA-32 reference also says that “the behavior of these prefixes is undefined when used with non-string instructions”.
`ret` obviously has nothing to do with strings.

The instruction is generated by gcc when optimisations are enabled, never by clang. Searching a bit in the depths of the gcc mailing list gives this cryptic message (http://gcc.gnu.org/ml/gcc-patches/2003-05/msg02117.html):

```
Subject: K8 tweak 2
Hi,
AMD recommends to avoid the penalty by adding rep prefix instead of nop
because it saves decode bandwidth.
```

Looking in the old AMD optimisation guide (http://support.amd.com/us/Processor_TechDocs/25112.PDF) for the then-current K8 processor microarchitecture (the first implementation of 64bit x86!), there is effectively mention of a “Two-Byte Near-Return `ret` Instruction”.

The text goes on to explain in advice 6.2 that “A two-byte `ret` has a `rep` instruction inserted before the `ret`, which produces the functional equivalent of the single-byte near-return `ret` instruction”.

It says that this form is preferred to the simple `ret` either when it is the target of any kind of branch, conditional (`jne/je/...`) or unconditional (`jmp/call/...`), or when it directly follows a conditional branch.

Basically, when the next instruction after a branch is a `ret`, whether the branch was taken or not, it should have a `rep` prefix.

Why? Because “The processor is unable to apply a branch prediction to the single-byte near-return form (opcode C3h) of the `ret` instruction.” Thus, “Use of a two-byte near-return can improve performance”, because it is not affected by this shortcoming.

Branch prediction is one of the most important mechanisms in modern microprocessors: by guessing which way a branch will go before it is executed, you can vastly improve the effectiveness of deep pipelines.

Pipelining is another interesting mechanism: instead of executing a single instruction at each clock cycle, modern CPUs separate its execution in several components, handled simultaneously. The most well-known components make the classic RISC pipeline, having: “Fetch” (loading the instruction from wherever it is, memory or cache), “Decode” (for the more advanced instruction sets), “Execute”, “Memory access”, and “Writeback” (writing the result). Instructions flow in the pipeline, such that when the first instruction is being executed, the next is being decoded, and the one after is being fetched. More on pipelines later.

We said that branch prediction helps pipelines. Imagine that you have a conditional branch, say a `je`.

```
...
    cmp r8, r9
    je equal

different:
    mov rax, 1
equal:
    xor rax, rax
```

How do you know which instruction to put after it in the pipeline? The `mov` at `different`? Or the `xor` at `equal`? Simple: you can’t be certain until you executed both the `cmp` and the `je`. And this is what the brilliantly named branch prediction does: predict which branch will be taken so the CPU knows which instructions to handle after the branch. Or at least, try to do so. If it is wrong, the whole pipeline has to be flushed because it was filled with the wrong instructions.

Now we know why a failure in the branch predictor would hurt performance. But we still don’t know why the simple, ubiquitous `ret` would make the branch predictor fail at its task.

The way the K8′s branch predictor works, there are 3 branch predictor entries (branch “selectors”) for every 16 bytes code block, shared by 9 (one every odd byte, plus one for byte 0) branch “indicators”. The branch predictor is linked to the cache, and the 16 bytes code blocks are grouped by 4, which is the size of a cache line (the granularity of the lowest-level cache in a CPU).
Branch indicators are two bits, encoding whether the branch is never taken (0), or which selector to use (1, 2, or 3). The branch selector knows whether the branch is a return or a call, but more importantly the branch prediction: never or always jump, or if neither, more advanced information.

Obviously, what looks like the best thing to do is to put at most 3 branch instructions per 16 bytes of code, or even better, only one. And this is what advice 6.1 of the K8 optimization guide tells us.

However, to return to our single-byte `ret`, the problem is that it is the only single-byte branch instruction. If a `ret` is at an odd offset and follows another branch, they will share a branch selector and will therefore be mispredicted (only when the branch was taken at least once, else it would not take up any branch indicator %2B selector). Otherwise, if it is the target of a branch, and if it is at an even offset but not 16-byte aligned, as all branch indicators are at odd offsets except at byte 0, it will have no branch indicator, thus no branch selector, and will be mispredicted.

Looking back at the gcc mailing list message introducing `repz ret`, we understand that previously, gcc generated:

```
nop
ret
```

But decoding two instructions is more expensive than the equivalent `repz ret`.

The optimization guide for the following AMD CPU generation, the K10, has an interesting modification in the advice 6.2: instead of the two byte `repz ret`, the three-byte `ret 0` is recommended, using what is the lesser known form of `ret`, taking a 2-byte operand. In assembly-speak, an operand that stands for itself (and not for an address where the interesting value lies) is an immediate. In the case of 2 bytes == 16 bits, it is an `imm16`.

The operand of `ret imm16` is the number of bytes to pop from the stack before after returning (thanks for the fix, 0ver!). This form is rarely used in programs using the cdecl (ubiquitous in the UNIX C world) calling convention, as the caller function is responsible for cleaning up the arguments pushed on the stack. In the 64 bit UNIX calling convention (as defined by the AMD64 SystemV ABI), most arguments are passed in registers, so this is not a problem.

At least, this instruction is fully defined. Why was it not also recommended in the K8 manual?

It turns out that among the other enhancements (http://www.anandtech.com/show/2183/6) in the K10 family, the `ret imm16` instruction is now fastpathed, instead of microcoded, which makes it faster to decode and execute than microcoded instructions (which are decoded into a series of smaller “microoperations”, or µops).

Continuing in the following generation of AMD CPUs, Bulldozer, we see that any advice regarding `ret` has disappeared from the optimization guide (http://support.amd.com/us/Processor_TechDocs/47414_15h_sw_opt_guide.pdf).

The branch predictor has been redesigned, and whereas it previously was linked to the instruction cache, it no longer is. Bulldozer has a single global branch predictor, and thus does not suffer of all the sharing issues of the 4×3 branch selectors for each 4×16 bytes cache line.

All this was a revelation.
There is so much low-level stuff that, even though almost no one ever heard of, makes our machines run so well.

We set off on a journey to read and document all these fascinating things.
This is how `repz ret` was born.

Funny bit: a while ago, someone reported a valgrind bug (http://bugs.kde.org/show_bug.cgi?id=115869) that occured when running qemu .
Julian Seward, creator of valgrind, responded:

```
> vex amd64->IR: unhandled instruction bytes: 0xF3 0xB8 0x92 0xE0

Good that someone finally tried to use Valgrind on QEMU.
[...]
0xF3 0xB8 really looks like an illegal amd64 instruction.  Is it
possible that some bug before this point made QEMU jump to an illegal
address?  Or that your custom suppressions are hiding a real problem?
```

The original poster came back with more info:

```
Last good BB seems like good code at 0x017043d5 but there is a problem decoding instruction start
ing at 0x017043d7. I asked qemu to output dyn-gen-code; qemu insists that it is a correct repz mo
v sequence:

###V's BB 10047
0x017043cf:  mov    -13631749(%rip),%eax        # 0xa042d0
0x017043d5:  jmpq   *%eax

###V's BB 10048
0x017043d7:  repz mov $0xe092,%eax
###V's choke here

0x017043dd:  mov    %eax,0x20(%rbp)
0x017043e0:  lea    -13631837(%rip),%ebx        # 0xa04289
0x017043e6:  retq
```

It turns out qemu, when handling a `repz ret`, didn't recognize it, removed the `ret` and generated the corresponding code, then added the `repz` prefix to some specific instructions that followed it. In this case, this generated `repz mov`, that was unheard of even in the AMD optimization guides.

---

repz ret - 2016

· twitter (http://twitter.com/repzret)