

Branch displacement Optimization

Source: <http://coding.derkeiler.com/Archive/Assembler/alt.lang.asm/2006-11/msg00216.html>

- *From:* "randyhyde@xxxxxxxxxxxxxx" <randyhyde@xxxxxxxxxxxxxx>
 - *Date:* 2 Nov 2006 16:00:04 -0800
-

Recently, Rene "Betov" Tournois has seen the light and thought that "small-first, grow large" is a good idea. Somehow, he thinks that this is a new and nifty idea. Of course, the explanation for it has been in this very newsgroup since Jan, 2004. See the following link and follow-ups:

http://groups.google.com/group/alt.lang.asm/browse_frm/thread/528907cad91150c4/a49af5f25dd15296?lnk=gst&q=b

However, because this seems to be of current interest, I'll repost that essay here:

=====

This is a short essay that attempts to explain the basics of "displacement optimization" in machine code such as on the x86. In posts I've made here and on various newsgroups (comp.lang.asm.x86 and alt.lang.asm) it is quite clear that many people don't understand what this is all about, and even those who do understand the basic issues can't believe the problem is as difficult to solve (optimally) as I claim that it is. In this essay I'm hoping to explain the process and the problems in such a way so to make all this clear.

What is This All About?

The first place to start is with the question "what is branch displacement optimization?" On certain CPUs (e.g., the x86), certain instructions have a limited branch range because of the way the CPU encodes the target address of the branch. For example, a typical "JE" instruction on the x86 is two bytes long – one byte for the opcode and one byte for a relative displacement. This relative displacement allows the instruction to transfer control to some instruction within +/- 128 bytes (roughly) of the JE instruction.

Branch displacement Optimization

What happens if the target address is **not** within range? Well, if the target address is within ± 4 bytes, the 386 and later devices have a special six-byte version of JE (two-byte opcode and two-byte displacement) that can transfer control to the label. If you're operating on a chip earlier than the 386, you have to emit a five-byte sequence like the following:

```
jne skipJump ;two bytes  
jmp target ;three bytes  
skipJump:
```

In early x86 assemblers (and in a few assemblers that have been created lately), it was the programmer's responsibility to manually encode the displacement sizes of these jumps. Besides being a pain in the butt to do manually, it was often the case that the programmer **missed** several optimization opportunities (hey, who wants to count bytes in a program) and so the program was not as optimal as it should have been.

It doesn't take a huge intuitive leap to realize that this kind of grunt work is **exactly** the kind of thing computers were designed to handle. As such, in the late 1980's, various assemblers started appearing that automatically handled the correct branch displacements (in most cases, anyway). I don't personally recall the name of the product, but the first time I saw this feature was in a MASM 5.1-compatible assembler (not MASM itself); TASM had the feature next. I believe Microsoft added this feature in MASM v6.0. Today, assemblers like Gas, FASM, and NASM all support this feature (and probably others too, I haven't looked that closely).

Why is this optimization even necessary? Well, it's pretty obvious that if the target location of a jump instruction is within ± 128 bytes, you're wasting four or more bytes if you're not using the smaller version of the jump instruction. Considering that most branches turn out to be within this range and there are generally quite a

Branch displacement Optimization

few branches in a program, just using the long version of each branch can make the program quite a bit larger than it needs to be.

The Problem

"So what's the big deal?" You're probably asking. "Why not just scan through the file during assembly and determine the distance to the target location and pick the appropriate size for the branch instruction?"

The problem is, the size of a given branch instruction may determine the size(s) of some other branch instruction(s). Consider the following simple code sequence:

```
jmp target
<exactly 125 bytes of code>
jmp someOtherTarget
target:
```

If "someOtherTarget" is within the +/- 128 byte range of the second jmp instruction above, you can encode it with only two bytes on the x86; otherwise you will need at least four bytes. If we do encode this second jump instruction in two bytes, then we can encode the first jump instruction above in two bytes (the actual range on the x86 is 127 bytes starting with the *next* instruction after the conditional jump, or -128 bytes before the next instruction following the conditional jump). OTOH, if the second jump requires more than two bytes, so will the first jump. Consider the following degenerate case:

```
jmp l1
<<125 bytes of code>>
jmp l2
l1:
<<125 bytes of code>>
jmp l3
l2:
<<125 bytes of code>>
jmp l4
```

Branch displacement Optimization

l3:

.
. .
etc.

The *last* jump in this sequence controls the sizes of all the other jumps in the code! This is because if the last one is a two-byte opcode, then the next-to-last one can be two bytes, then the one before that can be two bytes, then the one before that...

Unfortunately, there is no easy way (and soon, you'll see, efficient way) to determine the size of these jump instructions in a single pass over the object code the assembler generates. A typical assembler will do the following:

1. Scan over all the opcodes and determine if any branches can be reduced in size.
2. If the answer is yes, the assembler reduces the size of those branches and adjusts all other instructions whose target addresses and other values have changed as a result of reducing the size of the branch instructions.
3. If the answer was yes, repeat steps 1–3 until you make a pass with no optimizations occurring.

Note, that in the worst case, this algorithm runs in $O(n^2)$ time (that is, the amount of time it takes, worst case, is proportional to the square of the number of branches in the program; double the number of branches, it takes four times as long to process them). This isn't very good. But it isn't terrible, either.

Though the algorithm above looks easy enough, it turns out to be insufficient. Consider the following trivial code sequence:

Branch displacement Optimization

```
target2:  
jmp target1  
<< 124 bytes of object code>>  
jmp target2  
target1:
```

If we start off assuming that the branches are long (four bytes each), then a quick pass through this section of code will suggest that optimization is not possible. However, were you to arbitrarily select one of these jumps and make it a short jump, the other jump would also be short and both jumps would be within range. Therefore, an assembler that starts off using large displacements and shrinks them down can miss optimization opportunities such as this one.

Another possibility is to start with all short displacements and expand them as necessary. The reason many assemblers start with long displacements and shrink them is for reasons of robustness -- the code will work, even if it's less optimal, if a defect causes you to miss an optimization that could have been introduced into the object code. OTOH, if you start with short displacements and extend the ones that are out of range, a defect in the "correction" process results in defective code generation. As it turns out, optimization isn't guaranteed when you start small and work big, either. So those assembler authors who choose the robust aren't necessarily missing out on something.

OTOH, *most* branches are short to begin with, so starting off with short displacements and increasing the size of those that are out of range is going to be faster ($O(n^2) \gg O(m^2)$ if $n > m$).

The hard part for people to believe is that starting small and working big doesn't necessarily guarantee an optimal sized object file. In fact, and this is the part that really blows people away, sometimes the smallest object file is achieved by making certain branches larger that could have been encoded as short branches!

Branch displacement Optimization

Here's a short example of some MASM source code that demonstrates this problem:

```
.386
.model flat, syscall
.code
_HLAMain proc

jmpLbl: jmp near ptr target
jmpSize = $-jmpLbl
byte 32 - jmpSize*2 dup (0)
target:

_HLAMain endp

end
```

Here's the assembly listing:
Code:

```
.386
.model flat, syscall
00000000 .code
00000000 _HLAMain proc

00000000 EB 1C jmpLbl: jmp target
00000002 = 00000002 jmpSize = $-jmpLbl
00000002 0000001C [ byte 32 - jmpSize*2
dup
(0)
00
]
0000001E target:

0000001E _HLAMain endp

end
```

Note that the "object code" is 30 bytes long (1Eh). Also note that the jump instruction is two bytes long (that is, it's a short jump).

Branch displacement Optimization

Now look at what happens when we force the jump to be a five-byte jump:

```
.386
.model flat, syscall
.code
_HLAMain proc

jmpLbl: jmp near ptr target
jmpSize = $-jmpLbl
byte 32 - jmpSize*2 dup (0)
target:

_HLAMain endp

end
```

;;; Listing:

```
.386
.model flat, syscall
00000000 .code
00000000 _HLAMain proc

00000000 E9 00000016 jmpLbl: jmp near ptr target
00000005 = 00000005 jmpSize = $-jmpLbl
00000005 00000016 [ byte 32 - jmpSize*2
dup
(0)
00
]
0000001B target:

0000001B _HLAMain endp

end
```

Note that although the jump is now **five** bytes long (rather than two), the object module is

Branch displacement Optimization

actually *shorter* (27 bytes [1Bh] rather than 30).

Therefore, you cannot guarantee that you've got the shortest possible program by simply making all the jumps as short as they can be and letting it go at that.

Unfortunately, as this last example demonstrates, it's quite difficult to determine the optimal selection of short and long branch instructions in order to optimize your code. In fact, there have been some mathematical proofs that show that this problem is "NP-Complete", which means that the only way to guarantee you've got an optimal solution is to try all possible combinations of short and long displacements in the object file and select the (or one of the) combination(s) that is the shortest. Unfortunately, this is an intractable problem. The algorithm given earlier runs in quadratic time ($O(n^2)$), which isn't great, but still tractable. The best known algorithms for NP-Complete problems run in exponential time (i.e., $O(2^n)$ where 'n' is the number of branches in the program). This means that adding a *single* branch doubles the amount of time needed to determine the optimal output module. To put it bluntly, no computer available today and, indeed, no computer we can ever imagine building will be fast enough to handle the branches found in a reasonably-sized application program.

Yet assemblers available today *do* perform branch displacements. So how come they don't run real slow? The answer is that these assemblers *don't* guarantee that you'll get an optimal program. AFAIK, for example, none of them handle the last case I gave. Most of them simply use the "start large, work small" or "start small, work large" algorithms given earlier (for those who have had a data structures class/algorithms analysis course, you can think of this as a form the "greedy" algorithm). As such, while they produce *good* code (typically better than what someone will produce by hand), they do not necessarily produce optimal code.

Branch displacement Optimization

In fact, some assemblers limit the amount of optimization they do in order to keep assembly times low. TASM is famous for this; TASM has the /M# directive that lets you specify the maximum number of passes (this is, generally, done to actually *increase* the default number of passes that TASM performs). I don't have any information on MASM. However, the fact that FASM usually produces better code than MASM suggests that MASM stops at one point or another, before FASM does. I have no information on how NASM or Gas handles branch displacement optimization, so I cannot comment on their performance.

That's not to say that there is anything wrong with the way these assemblers do their "optimization." Most of the time they probably do produce the optimal program, and when they don't, they produce something very close to it. That's a reasonable trade-off given the intractable alternative.

Although branch displacement optimization is a tedious chore and it's difficult to add to an assembler, it is a prime example of one of those things that computers are much better suited for than human beings. A decade and a half ago it might have been reasonable to expect the programmer to manually take care of branch displacement optimizations. But given the fact that most reasonable assemblers today offer this facility, it's moved into the "must-have" category for anyone developing a serious assembler.

.