Branchless Equivalents of Simple Functions

hbfs.wordpress.com/2008/08/05/branchless-equivalents-of-simple-functions

August 6, 2008

Modern processors are equipped with sophisticated branch prediction algorithms (the Pentium family, for example, can predict a vast array of patterns of jumps taken/not taken) but if they, for some reason, mispredict the next jump, the performance can take quite a hit. Branching to an unexpected location means flushing the pipelines, prefetching new instructions, etc, leading to a stall that lasts for many tens of cycles. In order to avoid such dreadful stalls, one can use a *branchless equivalent*, that is, a code transformed to remove the if-then-elses and therefore jump prediction uncertainties.

Let us start by a simple function, the integer `abs( )` function. `abs`, for absolute value, returns… well, the absolute value of its argument. A straightforward implementation of `abs( )` in the C programming language could be

```
inline unsigned int abs(int x)
{
return (x<0) ? -x : x;
}
```

Which is simple enough but contains a hidden if-then-else. As the argument, `x`, isn't all that likely to follow a pattern that the branch prediction unit can detect, the simple function becomes potentially costly as the jump will be mispredicted quite often. How can we remove the if-then-else, then?

Let us first introduce the `sex( )` helper function—I still use the mnemonic `sex` to amuse and chock friends and coworkers, but it comes from a Motorola 6809 instruction, `s`ign `ex`tend. The `sex` function will return an integer where the sign bit of its argument have been copied in all the bits. For example, `sex(321)=0`, but `sex(-3)=0xff...ff`. This function is ideal to generate a mask based on the sign of the argument. Of course, `sex` must be branchless to be of any use to us. At the assembly language level the instruction exists on most processors (it is one of the `cbw` (convert byte to word), `cwd` (convert word to double word), etc, instructions on x86/AMD64), but what can we do at the C language level to force the compiler to use the specialized instruction, or at least an efficient replacement? One can use the right shift operator:

```
inline unsigned int sex(int x)
{
return x >> (CHAR_BIT*sizeof(int)-1);
}
```

where the (compile-time) safe expression `(CHAR_BIT*sizeof(int)-1)` evaluates to 15, 31, or 63 depending on the size of integers on the target computer (`CHAR_BIT` comes from `limits.h`, and is worth 8, most of the times). However, this one-liner relies on the underlying processor's shift instruction which, in some case, can be dreadfully slow (a few cycles for each bit shifted in micro-controllers) or very fast (one cycle simultaneously executed with other instructions in bigger processors). One can also use an `union`, which will compile to memory manipulation instructions, completely removing shifts from the function:

```
inline int sex(int x)
{
union
{
long w;
struct { int lo, hi; }
} z = { .w=x };
return z.hi;
}
```

This will basically force the compiler to use the `cbw` family of instructions. Let us rewrite `abs` using `sex`:

```
inline unsigned int abs(int x)
{
return (x ^ sex(x)) - sex(x);
}
```

Now, how does *that* work? If `x` is negative, `sex(x)` will be `0xff...ff`, what is, filled with ones. If `x` is not negative (zero or positive), `sex(x)` will be zero. So, if the number of negative, it computes its two's complement, otherwise leaves it unchanged. For example, if `x` is negative, say -3 (no point in using large, weird, numbers here), `sex(-3)` is `0xff...ff` and `-3 ^ 0xff...ff` is the same as `~(-3)`, the bitwise negation of `-3`. Then, we subtract `-1` (which is the same as *adding* `1`), computing `~(-3)+1` which is the correct two's complement. If on the other hand `x` is positive (or null), `sex(x)` evaluates to zero, and lo! `(x ^ 0)-0 = x`, which leaves the value of `x` unchanged!

Of course, when compiling the above `abs` function the compiler generates very little code, especially when one uses the `union` version of `sex`. For example, on Intel x86, it could compile down to

```
abs: cdq eax
xor eax,edx
sub eax,edx
```

assuming the value is already in (and returned by) `eax`. The `cdq` instruction sign-extends `eax` into the `edx` register: it promotes a 32 bits value to a 64 bits value held in `edx:eax`.

Now, we can use `sex` for other if-then-else type function. Take `min` and `max` for example. The pair is usually implemented as

```
inline int min(int a, int b) { return (a<b) ? a : b; }
inline int max(int a, int b) { return (a>b) ? a : b; }
```

Using `sex`, the pair becomes

```
inline int min(int a, int b)
{
return b + ((a-b) & sex(a-b));
}
inline int max(int a, int b)
{
return a + ((b-a) & ~sex(b-a));
}

```

which are now thoroughly branchless. Hurray!

Can you think of other common, simple functions, what would benefit from branch removal?