

Floating-point bit hacking: iterated nextafter() without loop?

Source: <http://coding.derkeiler.com/Archive/C/ CPP/comp.lang.c/2004-10/1401.html>

From: Daniel Vallstrom (daniel.vallstrom_at_gmail.com)

Date: 10/14/04

Date: 14 Oct 2004 03:14:35 -0700

I am having trouble with floating point addition because of the limited accuracy of floating types. Consider e.g. $x0 += f(n)$ where $x0$ and f are of some floating type. Sometimes $x0$ is much larger than $f(n)$ so that $x0 + f(n) == x0$. For example, $x0$ could be $2^{300} * (1.1234\dots)$ while $f(n)$ is $2^{100} * (1.4256\dots)$. If $x0 + f(n) == x0$ I still sometimes want the addition of $f(n)$ to $x0$ to have some effect on $x0$. A good solution seems to be to update $x0$ to the n th value greater than $x0$. Something equivalent to this:

```
for ( ; n != 0; n-- )
{
    x0 = nextafter( x0, DBL_MAX );
}
```

The problem with the above loop is that it may take too long because n could be quite large. Moreover, very many such loops would be executed.

Disappointedly gcc doesn't seem to be able to optimize over nextafter at all.

However, with only some modest bit hacking and without assuming much about the floating point implementation it is possible to calculate the n th greater value fast. At the end of this post is a program showing such a solution. Of course it is not guaranteed to work as intended but is there any non-odd platform where it will fail? If there are any places where the solution could fail, what would the consequences be? Is there a better solution to the problem than the one below?

Daniel Vallstrom

```
// Tests consecutive nextafter calls and a way to compute the same thing fast
// without iterated nextafter calls.
// Daniel Vallstrom, 041014.
// Compile with e.g: gcc -std=c99 -pedantic -Wall -O3 -lm nextafterNTest.c
```

comp.lang.c: Floating-point bit hacking: iterated nextafter() without loop?

```
/* The program should print something like:
n: 0x1000000 (16777216)
x0: 0x1.001p+8
xn: n nextafter( x0, DBL_MAX ) iterations: 0x1.0010001p+8
   The xn loop took 0.8 cpu seconds.
m: n * nextafter( 0.0, DBL_MAX ): 0x0.0000001p-1022
e: exp2( floor( log2( x0 ) ) ): 0x1p+8
z: e | m: 0x1.0000001p+8
z1: z - e: 0x1p-20
z2: x0 + z1: 0x1.0010001p+8 (==xn)
u: n unrolled nextafter( x0, DBL_MAX ) iterations: 0x1.0010001p+8 (==xn)
   The u loops took 0.7 cpu seconds.
*/
```

```
#include <stdio.h>
#include <float.h>
#include <math.h>
#include <assert.h>
#include <time.h>

#define nextafter2M( x, y ) (x) = nextafter((x),(y)); (x) = nextafter((x),(y))
#define nextafter4M( x, y ) nextafter2M( (x), (y) ); nextafter2M( (x), (y) )
#define nextafter8M( x, y ) nextafter4M( (x), (y) ); nextafter4M( (x), (y) )
#define nextafter16M( x, y ) nextafter8M( (x), (y) ); nextafter8M( (x), (y) )
#define nextafter32M( x, y ) nextafter16M( (x), (y) ); nextafter16M( (x), (y) )
#define nextafter64M( x, y ) nextafter32M( (x), (y) ); nextafter32M( (x), (y) )

int main( void )
{
    clock_t clockAtLoopStart; // Used to time the start of loops.
    clock_t clockAtLoopEnd; // Used to time the end of loops.

    // The number of nextafter iterations to use.
    unsigned int n = ( 1u << 24 );
    printf( "\nn: %#x (%u)\n", n, n );

    // The base double to apply nextafter to.
    double x0 = 0x1.001p8;
    printf( "x0: %a\n", x0 );

    // The nth double greater than x0.
    double xn = x0;
    clockAtLoopStart = clock();
    for ( unsigned int k = 0; k != n; k++ )
    {
        xn = nextafter( xn, DBL_MAX );
    }
    clockAtLoopEnd = clock();
    printf( "xn: n nextafter( x0, DBL_MAX ) iterations: %a\n", xn );
    // xn: n nextafter( x0, DBL_MAX ) iterations: 0x1.0010001p+8
    printf( " The xn loop took %.1f cpu seconds.\n",
```

comp.lang.c: Floating-point bit hacking: iterated nextafter() without loop?

```
(double) ( clockAtLoopEnd - clockAtLoopStart ) / CLOCKS_PER_SEC );
// The xn loop took 0.8 cpu seconds.

// The goal is to compute xn fast. At least gcc -O3 does *not* optimize
// iterated nextafter calls (the above loop). If the loop (n) is long (or
// many loops have to be done) it takes too long to run.

// Calculate a useful mantissa:
double m = nextafter( 0.0, DBL_MAX );
m = n*m;
printf( "m: n * nextafter( 0.0, DBL_MAX ): %a\n", m );
// m: n * nextafter( 0.0, DBL_MAX ): 0x0.0000001p-1022

// Calculate a useful exponent part.
double e = exp2( floor( log2( x0 ) ) );
printf( "e: exp2( floor( log2( x0 ) ) ): %a\n", e );
// e: exp2( floor( log2( x0 ) ) ): 0x1p+8

// Combine e and m.
double z = e;
for ( unsigned char * zPtr = &z, * zEnd = zPtr + sizeof(z), * mPtr = &m;
      zPtr != zEnd; zPtr++, mPtr++ )
{
    *zPtr |= *mPtr;
}
printf( "z: e | m: %a\n", z );
// z: e | m: 0x1.0000001p+8

// Remove the faulty first mantissa bit in z, the one coming from e.
double z1 = z - e;
printf( "z1: z - e: %a\n", z1 );
// z1: z - e: 0x1p-20

// Finally we are ready to add the value to x0.
double z2 = x0 + z1;
printf( "z2: x0 + z1: %a (==xn)\n", z2 );
// z2: x0 + z1: 0x1.0010001p+8 (==xn)

assert( z2 == xn );

// Test if unrolling the nextafter calls helps. (It doesn't (besides
// the speed-up from the unrolling itself).

double u = x0;

clockAtLoopStart = clock();
for ( unsigned int k = n/64; k != 0; k-- )
{
    nextafter64M( u, DBL_MAX );
}
```

comp.lang.c: Floating-point bit hacking: iterated nextafter() without loop?

```
for ( unsigned int k = n%64; k != 0; k-- )
{
    u = nextafter( u, DBL_MAX );
}
clockAtLoopEnd = clock();

printf( "u: n unrolled nextafter( x0, DBL_MAX ) iterations: %a (==xn)\n",
        u );
// u: n unrolled nextafter( x0, DBL_MAX ) iterations: 0x1.0010001p+8 (==xn)
assert( u == xn );
printf( " The u loops took %.1f cpu seconds.\n",
        (double) ( clockAtLoopEnd - clockAtLoopStart ) / CLOCKS_PER_SEC );
// The u loops took 0.7 cpu seconds.

return 0;
}
```