Simon Parent

January 5, 2012

The Problem – Signed Integer Overflow

Why should we care?

- Obligatory mention: the failure of Ariane 5 Flight 501, which cost more than 370 million dollars.

The Problem – Signed Integer Overflow

Why should we care?

- Obligatory mention: the failure of Ariane 5 Flight 501, which cost more than 370 million dollars.
- There are an embarrassing number of programs which purport to deal with integers, but cannot handle arbitrarily large user input. Do we really expect all users to tiptoe around the magic number $2^{31} - 1$? Why should *they* care?

The Problem – Signed Integer Overflow

Why should we care?

- Obligatory mention: the failure of Ariane 5 Flight 501, which cost more than 370 million dollars.
- There are an embarrassing number of programs which purport to deal with integers, but cannot handle arbitrarily large user input. Do we really expect all users to tiptoe around the magic number $2^{31} - 1$? Why should *they* care?
- They leave potential for security vulnerabilities, most of which abuse the fact that an overflow is often negative in order to affect the result of a comparison.

What We Want

We want an implementation of (signed) integer arithmetic that is both

- safe, in the sense that overflows are at least detected (but ideally we seamlessly transition to big integers at that point), but
- still relatively fast, so that we are not tempted to stray

Tagged Integers – The Classic Compromise

In this scheme, every integer is one of two types:

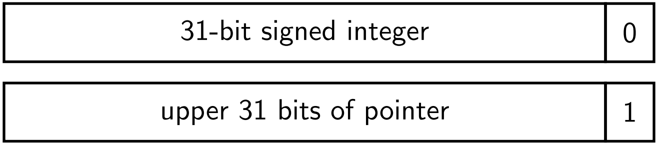
- A **fixnum**, so called because it comes from a fixed “small” range, comparable to the range of a usual 32-bit signed integer.
- A **bignum**, which has unbounded range, which forces its memory to be dynamically allocated.

Normally the bignum would be encoded as a pointer to whatever data structure the bignum library is using.

The key observation here is that these pointers are going to be word-aligned, so at the very least, the last bit is going to be zero.

Encoding Tagged Integers

We can use this last bit as the **tag**. Say that a tag of 1 means that this word represents a bignum, and a tag of 0 means that this word represents a fixnum.



This seems strange because we will have to use an offset of -1 when we want to access the data structure behind the bignum pointer.

However, this means that our encoding for a fixnum representing n (assuming two's complement representation) is exactly the usual representation for the integer $2n$. This means that we can use the usual add instruction (since $2m + 2n = 2(m + n)$), and the usual overflow check.

Operating on Tagged Integers

Now, the code to add two integers in this form will look like the following.

```
...

add eax, ebx        eax = eax + ebx

...
```

Operating on Tagged Integers

Now, the code to add two integers in this form will look like the following.

```
        ...
add eax, ebx           eax = eax + ebx
jo alternate_path      branch on overflow
        ...
```

Operating on Tagged Integers

Now, the code to add two integers in this form will look like the following.

```
       ...
mov ecx, eax         ecx = eax
or ecx, ebx          ecx = (ecx OR ebx)
test ecx, 1          temp = (ecx AND 1)
jnz alternate_path   branch if temp ≠ 0
add eax, ebx         eax = eax + ebx
jo alternate_path    branch on overflow
       ...
```

Operating on Tagged Integers

Now, the code to add two integers in this form will look like the following.

```
        ...
mov ecx, eax          ecx = eax
or ecx, ebx           ecx = (ecx OR ebx)
test ecx, 1           temp = (ecx AND 1)
jnz alternate_path    branch if temp ≠ 0
add eax, ebx          eax = eax + ebx
jo alternate_path     branch on overflow
        ...
```

We can actually use this to support a more general dynamic type system. If we tag every object which is not an integer the same as a bignum, then attempts to "add" other objects will also end up in the alternate code path, where they can be appropriately handled.

Tagged Integers – Summary

Advantages

- Fixnums do not require dynamic allocation! Arguably, this is the biggest win over a method that only recognizes bignums.
- The ordinary machine instructions can be used to add and subtract fixnums in this form, although they are surrounded by the safety checks. In this way, the overhead is fairly minimal. More on this later.

Disadvantages

- Portability.
- Storing values in strange ways makes debugging more difficult without special tools that are aware of the tagging method.

Performance

Do Object-Oriented Languages Need Special Hardware Support?
(Urs Holzle, David Ungar, 1995)

They compare the approach we just mentioned with special instructions for tagged arithmetic, which essentially perform everything we just discussed in the same time as a normal arithmetic instruction.

They estimate the difference to be around 3% in typical programs, and thus answer “No”.

Measurement

```
y = 0
for x = 0, 1, 2, . . . , 999999:
  z = (x / 999999) * 1000000000 + 1
  y = y + x / 1000 * z * z - 20
print y
```

Here “A / B” is integer division rounded towards either zero or negative infinity (it doesn’t matter since A and B will always be nonnegative).

Measurement

```
y = 0
for x = 0, 1, 2, . . . , 999999:
  z = (x / 999999) * 1000000000 + 1
  y = y + x / 1000 * z * z - 20
print y
```

Here "A / B" is integer division rounded towards either zero or negative infinity (it doesn't matter since A and B will always be nonnegative).

This program has some nice properties:

- The values of y are within the range $-2^{30} < y < 2^{30}$ for all $x < 999999$, but after the last iteration, y is greater than 2^{64}.
- Each iteration depends on all previous iterations.
- Optimizers should not be smart enough to determine the final value of y (even `gcc -O3` kept the loop structure intact).

Reality

- only fixnums (wrong)
- only bignums (slow)
- fixnums automatically promoted to bignums, but no “tag encoding”
- tagged integers

Results – C

baseline: `gcc` with no optimizations (incorrect!)

$\sim$40 ms.

Results – C

baseline: `gcc` with no optimizations (incorrect!)

$\sim$40 ms.

`gcc -O3` (incorrect!)

$\sim$11 ms. 0.28x baseline.

Results – C

baseline: `gcc` with no optimizations (incorrect!)

$\sim$40 ms.

`gcc -O3` (incorrect!)

$\sim$11 ms. 0.28x baseline.

`gcc -ftrapv`

$\sim$160 ms. 4.0x baseline.

-ftrapv

```
          ...
movl $0x1,0x4(%esp)
mov  %eax,(%esp)
call 8048590 <__addvsi3>
mov  %eax,0x18(%esp)
movl $0x3e8,0x14(%esp)
mov  0x24(%esp),%eax
mov  %eax,%edx
          ...
```

Signed Overflow in C

The C standard says that the result of signed overflow is undefined behaviour, and gcc takes great advantage of this. This is ostensibly to permit certain kinds of optimizations.

Thus, it is essentially impossible for code like

```
add eax, ebx
jo alternate_path
```

to ever be emitted by a C compiler from C code. C semantics are simply not compatible with testing for overflow after the addition has occurred.

Signed Overflow in C

The C standard says that the result of signed overflow is undefined behaviour, and gcc takes great advantage of this. This is ostensibly to permit certain kinds of optimizations.

Thus, it is essentially impossible for code like

```
add eax, ebx
jo alternate_path
```

to ever be emitted by a C compiler from C code. C semantics are simply not compatible with testing for overflow after the addition has occurred.

Recommended Reading: What Every C Programmer Should Know About Undefined Behavior <http://blog.llvm.org/2011/05/what-every-c-programmer-should-know.html>

Consequences

As a result, jump on overflow is a very underused instruction. Many programs are written in C, and many other programming languages either compile to C, or are interpreted by C programs.

Consequences

As a result, jump on overflow is a very underused instruction. Many programs are written in C, and many other programming languages either compile to C, or are interpreted by C programs.

A comment block from the Python 2 interpreter:

Integer overflow checking for * is painful: Python tried a couple ways, but they didn't work on all platforms, or failed in endcases (a product of -sys.maxint-1 has been a particular pain).

Here's another way:

. . .

Consequences

The native long product x*y is either exactly right or *way* off, being just the last n bits of the true product, where n is the number of bits in a long (the delivered product is the true product plus i*2**n for some integer i).

The native double product (double)x * (double)y is subject to three rounding errors: on a sizeof(long)==8 box, each cast to double can lose info, and even on a sizeof(long)==4 box, the multiplication can lose info. But, unlike the native long product, it's not in *range* trouble: even if sizeof(long)==32 (256-bit longs), the product easily fits in the dynamic range of a double. So the leading 50 (or so) bits of the double product are correct.

We check these two ways against each other, and declare victory if they're approximately the same. Else, because the native long product is the only one that can lose catastrophic amounts of information, it's the native long product that must have overflowed.

Results – Languages with “bignum promotion”

Python 2.7.2.

Results – Languages with “bignum promotion”

Python 2.7.2.

$\sim$2400 ms. 60x baseline.

Results – Languages with “bignum promotion”

Python 2.7.2.

$\sim$2400 ms. 60x baseline.

Python 3.2.2.

Results – Languages with “bignum promotion”

Python 2.7.2.

$\sim$2400 ms. 60x baseline.

Python 3.2.2.

$\sim$3400 ms. 85x baseline.

Results – Languages with “bignum promotion”

Python 2.7.2.

$\sim$2400 ms. 60x baseline.

Python 3.2.2.

$\sim$3400 ms. 85x baseline.

Ruby 1.8.7.

Results – Languages with “bignum promotion”

Python 2.7.2.

$\sim$2400 ms. 60x baseline.

Python 3.2.2.

$\sim$3400 ms. 85x baseline.

Ruby 1.8.7.

$\sim$5500 ms. 137x baseline.

Results – Languages with “bignum promotion”

Python 2.7.2.

$\sim$2400 ms. 60x baseline.

Python 3.2.2.

$\sim$3400 ms. 85x baseline.

Ruby 1.8.7.

$\sim$5500 ms. 137x baseline.

Racket 5.1.3 (Scheme).

Results – Languages with “bignum promotion”

Python 2.7.2.

$\sim$2400 ms. 60x baseline.

Python 3.2.2.

$\sim$3400 ms. 85x baseline.

Ruby 1.8.7.

$\sim$5500 ms. 137x baseline.

Racket 5.1.3 (Scheme).

$\sim$100 ms. 2.5x baseline.

Results – Languages with fixnum/bignum split

Java 1.6.0_22 with bignums (`java.math.BigInteger`).

Results – Languages with fixnum/bignum split

Java 1.6.0_22 with bignums (`java.math.BigInteger`).

$\sim$1100 ms. 27x baseline.

Results – Languages with fixnum/bignum split

Java 1.6.0_22 with bignums (`java.math.BigInteger`).

$\sim$1100 ms. 27x baseline.

Java 1.6.0_22 with fixnums (`int`). (incorrect!)

Results – Languages with fixnum/bignum split

Java 1.6.0_22 with bignums (`java.math.BigInteger`).

$\sim$1100 ms. 27x baseline.

Java 1.6.0_22 with fixnums (`int`). (incorrect!)

$\sim$26 ms. 0.65x baseline! (after it warms up)

Results – Languages with fixnum/bignum split

Java 1.6.0_22 with bignums (`java.math.BigInteger`).

$\sim$1100 ms. 27x baseline.

Java 1.6.0_22 with fixnums (`int`). (incorrect!)

$\sim$26 ms. 0.65x baseline! (after it warms up)

Perl 5.14.1 with bignums (`use bigint;`).

Results – Languages with fixnum/bignum split

Java 1.6.0_22 with bignums (`java.math.BigInteger`).

$\sim$1100 ms. 27x baseline.

Java 1.6.0_22 with fixnums (`int`). (incorrect!)

$\sim$26 ms. 0.65x baseline! (after it warms up)

Perl 5.14.1 with bignums (`use bigint;`).

$\sim$650 **seconds**. 16250x baseline.

Results – Languages with fixnum/bignum split

Java 1.6.0_22 with bignums (`java.math.BigInteger`).

$\sim$1100 ms. 27x baseline.

Java 1.6.0_22 with fixnums (`int`). (incorrect!)

$\sim$26 ms. 0.65x baseline! (after it warms up)

Perl 5.14.1 with bignums (`use bigint;`).

$\sim$650 **seconds**. 16250x baseline.

Perl 5.14.1 with fixnums (`use integer;`). (incorrect!)

Results – Languages with fixnum/bignum split

Java 1.6.0_22 with bignums (`java.math.BigInteger`).

$\sim$1100 ms. 27x baseline.

Java 1.6.0_22 with fixnums (`int`). (incorrect!)

$\sim$26 ms. 0.65x baseline! (after it warms up)

Perl 5.14.1 with bignums (`use bigint;`).

$\sim$650 **seconds**. 16250x baseline.

Perl 5.14.1 with fixnums (`use integer;`). (incorrect!)

$\sim$1600 ms. 40x baseline.

My Experiment with Naïve Code Generation

```
pop eax
pop ebx



add eax, ebx



push eax
```

My Experiment with Naïve Code Generation

```
    pop eax
    pop ebx
    mov ecx, eax
    or ecx, ebx
    test ecx, 1
    jnz bad
    add eax, ebx
    jno good
bad:
    sub esp, 8
    call bignum_add
    add esp, 8
good:
    push eax
```

My Experiment with Naïve Code Generation

```
push x
push 999999
divide
push 1000000000
multiply
push 1
add
duplicate
push x
push 1000
divide
multiply
multiply
push y
add
push 20
subtract
pop y
```

My Results

My Simulated Naïve Compiler

$\sim$90 ms. 2.3x baseline.

Moral

Write a compiler! Specifically, you should be emitting machine code.

People often say that the choice of algorithm dominates other concerns, such as choice of programming language. We saw that in this case, the saying is wrong. Even bad machine code is better than an interpreter.

Questions?