链滴

一个简单例子说明为什么C语言仍很重要

作者：jetlan

原文链接：https://ld246.com/article/1378041365741

来源网站：链滴

翻译自 <a href="http://jabsoft.io/2013/08/29/why-c-still-matters-in-2013
a-simple-example/">Anthony J Bonkoski

最近，我一直在开发Dynvm——一个通用的动态语言运行时。就其他任何好的语言运行时项目一样，开发是由基准测试程序驱动的。因此，我一直在用基准测试程序试各种由不同语言编写的算法，以此对其典型的运行速度有一个感觉上的认识。一个经典的测试就是代计算斐波那契数列。为简单起见，我以2^64为模，用两种语言编写实现了该算法。
用Python语言实现如下：
<pre>def fib(n):
    SZ = 2**64
    i = 0
    a, b = 1, 0
    while i < n:
        t = b
        b = (b+a) % SZ
        a = t
        i = i + 1
    return b
</pre>
用C语言实现如下：

<pre>#include <stdio.h>
#include <stdlib.h>
typedef unsigned long ulong;

int main(int argc, char *argv[])
{
ulong n = atoi(argv[1]);
ulong a = 1;
ulong b = 0;
ulong t;

for(ulong i = 0; i < n; i++) {
    t = b;
    b = a+b;
    a = t;
}

printf("%lu\n", b);
return 0;

}</pre>

Dynvm包含一个基准测试程序框架，该框架可以允许在不同语言之间对比运行速度。在一台Intel 7-3840QM（调频到1.2 GHz)机器上，当 n=1,000,000 时，对比结果如下：

<pre>==
 语言   时间 (秒)
==
 Java   0.133</pre>

```
 C Language                   0.006
 CPython                      0.534
 Javascript V8                0.284</pre>
<p> </p>
<p>很明显，C语言是这里的老大，但是java的结果有点误导性，因为大部分的时间是由JIT编译器启（~120ms）占用的。当n=100,000,000时，结果变得更明朗：</p>
<p> </p>
<pre>============================================================
 语言                         时间（秒）
============================================================
 Java                         0.300
 C Language                   0.172
 CPython                      47.909
 Javascript V8                24.179</pre>
<p> </p>
<p>在这里，我们探究下为什么C语言在2013年仍然很重要，以及为什么编程世界不会完全“跳槽”Python或者V8/Node。有时你需要原始性能，但是动态语言仍在这方面艰难挣扎着，即使对以上很单的例子而言。我个人相信这种情况会克服掉，通过几个项目我们能在这方面看到很大的希望：JVM V8、PyPy、LuaJIT等等，但在2013年我们还没有到达“目的地”。</p>
<p>然而，我们无法回避这样的问题：为什么差距如此之大？在C语言和Python之间有278.5倍的性差距！最不可思议的地方是，从语法角度讲，以上例子中的C语言和Python内部循环基本上一模一样</p>
<p>为了找到问题的答案，我搬出了反汇编器，发现了以下现象：</p>
<pre>0000000000400480 &lt;main&gt;:
247   400480:       48 83 ec 08             sub    $0x8,%rsp
248   400484:       48 8b 7e 08             mov    0x8(%rsi),%rdi
249   400488:       ba 0a 00 00 00          mov    $0xa,%edx
250   40048d:       31 f6                   xor    %esi,%esi
251   40048f:       e8 cc ff ff ff          callq  400460 &lt;strtol@plt&gt;
252   400494:       48 98                   cltq
253   400496:       31 d2                   xor    %edx,%edx
254   400498:       48 85 c0                test   %rax,%rax
255   40049b:       74 26                   je     4004c3 &lt;main+0x43&gt;
256   40049d:       31 c9                   xor    %ecx,%ecx
257   40049f:       31 f6                   xor    %esi,%esi
258   4004a1:       bf 01 00 00 00          mov    $0x1,%edi
259   4004a6:       eb 0e                   jmp    4004b6 &lt;main+0x36&gt;
260   4004a8:       0f 1f 84 00 00 00 00    nopl   0x0(%rax,%rax,1)
261   4004af:       00
262   4004b0:       48 89 f7                mov    %rsi,%rdi
263   4004b3:       48 89 d6                mov    %rdx,%rsi
264   4004b6:       48 83 c1 01             add    $0x1,%rcx
265   4004ba:       48 8d 14 3e             lea    (%rsi,%rdi,1),%rdx
266   4004be:       48 39 c8                cmp    %rcx,%rax
267   4004c1:       77 ed                   ja     4004b0 &lt;main+0x30&gt;
268   4004c3:       be ac 06 40 00          mov    $0x4006ac,%esi
269   4004c8:       bf 01 00 00 00          mov    $0x1,%edi
270   4004cd:       31 c0                   xor    %eax,%eax
271   4004cf:       e8 9c ff ff ff          callq  400470 &lt;__printf_chk@plt&gt;
272   4004d4:       31 c0                   xor    %eax,%eax
273   4004d6:       48 83 c4 08             add    $0x8,%rsp
274   4004da:       c3                      retq
275   4004db:       90                      nop</pre>
<p> </p>
```

原文链接：一个简单例子说明为什么C语言仍很重要

 最主要的部分是计算下一个斐波那契数值的内部循环：
<pre>262 4004b0: 48 89 f7 mov %rsi,%rdi
263 4004b3: 48 89 d6 mov %rdx,%rsi
264 4004b6: 48 83 c1 01 add $0x1,%rcx
265 4004ba: 48 8d 14 3e lea (%rsi,%rdi,1),%rdx
266 4004be: 48 39 c8 cmp %rcx,%rax
267 4004c1: 77 ed ja 4004b0 <main+0x30></pre>
变量在寄存器中的分配情况如下：
<pre>a: %rsi
b: %rdx
t: %rdi
i: %rcx
n: %rax</pre>

262和263行实现了变量交换，264行增加循环计数值，虽然看起来比较奇怪，265行实现了b=a+。然后做一个简单地比较，最后一个跳转指令跳到循环开始出继续执行。
手动反编译以上代码，代码看起来是这样的：
<pre>loop: t = a
    a = b
    i = i+1
    b = a+t
    if(n > i) goto loop

</pre>
整个内部循环仅用六条X86-64汇编指令就实现了（很可能内部微指令个数也差不多。译者注：Intel X86-64处理器会把指令进一步翻译成微指令，所以CPU执行的实际指令数要比汇编指令多）。CPython解释模块对每一条高层的指令字节码至少需要六条甚至更多的指令来解释，相比而言，C语言完。除此之外，还有一些其他更微妙的地方。
拉低现代处理器执行速度的一个主要原因是对于主存的访问。这个方面的影响十分可怕，在微处器设计时，无数个工程时（engineering hours）都花费在找到有效地技术来“掩藏”访存延时。通的策略包括：缓存、推测预取、load-store依赖性预测、乱序执行等等。这些方法确实在使机器更快面起了很大作用，但是不可能完全不产生访存操作。
在上面的汇编代码中，从没访问过内存，实际上变量完全存储在CPU寄存器中。现在考虑CPytho：所有东西都是堆上的对象，而且所有方法都是动态的。动态特性太普遍了，以至于我们没有办法知，a+b执行integer_add(a, b)、string_concat(a, b)、还是用户自己定义的函数。这也就意味着很多间花在了在运行时找出到底调用了哪个函数。动态JIT运行时会尝试在运行时获取这个信息，并动态产x86代码，但是这并不总是非常直接的（我期待V8运行时会表现的更好，但奇怪的是它的速度只是Python的0.5倍）。因为CPython是一个纯粹的翻译器，在每个循环迭代时，很多时间花在了解决动态特上，这就需要很多不必要的访存操作。
除了以上内存在搞鬼，还有其他因素。现代超标量乱序处理器核一次性可以取好几条指令到处理中，并且“在最方便时”执行这些指令，也就是说：鉴于结果仍然是正确的，指令执行顺序可以任意这些处理器也可以在同一个时钟周期并行执行多条指令，只要这些指令是不相关的。Intel Sandy Bride CPU可以同时将168条指令重排序，并可以在一个周期中发射（即开始执行指令）至多6条指令，同结束（即指令完成执行）至多4条指令！粗略地以上面斐波那契举例，Intel这个处理器可以大约把28译者注：28*6=168）个内部循环重排序，并且几乎可以在每一个时钟周期完成一个循环！这听起来霸气，但是像其他事一样，细节总是非常复杂的。
我们假定8条指令是不相关的，这样处理器就可以取得足够的指令来利用指令重排序带来的好处对于包含分支指令的指令流进行重排序是非常复杂的，也就是对if-else和循环（译者注：if-else需要断后跳转，所以必然包含分支指令）产生的汇编代码。典型的方法就是对于分支指令进行预测。CPU动态的利用以前分支执行结果来猜测将来要执行的分支指令的执行结果，并且取得那些它“认为”将执行的指令。然而，这个推测有可能是不正确的，如果确实不正确，CPU就会进入复位模式（译者注这里的复位不是指处理器reset，而是CPU流水线的复位），即丢弃已经取得的指令并且重新开始取。这种复位操作有可能对性能产生很大影响。因此，对于分支指令的正确预测是另一个需要花费很多程时的领域。
现在，不是所有分支指令都是一样的，有些可以很完美的预测，但是另一些几乎不可能进行预测前面例子中的循环中的分支指令——就像反汇编代码中267行——是最容易预测的其中一种，这个分

指令会连续向后跳转100,000,000次。
以下是一个非常难预测的分支指令实例：
<pre>void main(void)
{
    for(int i = 0; i < 1000000; i++) {
        int n = random();
        if(n >= 0) {
            printf("positive!\n");
        } else {
            printf("negative!\n");
        }
    }
}</pre>
如果random()是真正随机的（事实上在C语言中远非如此），那么对于if-else的预测还不如随便来的准确。幸运的是，大部分的分支指令没有这么顽皮，但是也有很少一部分和上面例子中的循环分指令一样变态。
回到我们的例子上：C代码实现的斐波那契数列，只产生一个非常容易预测的分支指令。相反地CPython代码就非常糟糕。首先，所有纯粹的翻译器有一个“分配”循环，就像下面的例子：
<pre>void exec_instruction(instruction_t *inst)
{
    switch(inst->opcode) {
        case ADD: // do add
        case LOAD: // do load
        case STORE: // do store
        case GOTO: // do goto
    }
}</pre>
编译器无论如何处理以上代码，至少有一个间接跳转指令是必须的，而这种间接跳转指令是比较预测的。
接下来回忆一下，动态语言必须在运行时确定如“ADD指令的意思是什么”这样基本的问题，这然会产生——你猜对了——更加变态的分支指令。
以上所有因素加起来，最后导致一个278.5倍的性能差距！现在，这当然是一个很简单的例子，是其他的只会比这更变态。这个简单例子足以凸显低级静态语言（例如C语言）在现代软件中的重要位。我当然不是2013年里C语言最大的粉丝，但是C语言仍然主导着低级控制领域及对性能要求高的用程序领域。
