C++中虚函数(virtual function)到底有多慢

虚函数为什么慢，cpu 分支预测技术，虚函数到底要调用哪些汇编，虚函数实现的简单图示，虚函数不能内联，

印象中经常看到有人批评 C++的虚函数很慢，但是虚函数为什么慢，虚函数到底有多慢呢？

一、理论分析

虚函数慢的原因主要有三个：

1. **多了几条汇编指令（运行时得到对应类的函数的地址）**
2. **影响 cpu 流水线**
3. **编译器不能内联优化（仅在用父类引用或者指针调用时，不能内联）**

先简单说下虚函数的实现，以下面**测试**代码中的 VirtualVector 类为例，VirtualVector 类的内存布局如下：

当在用父类的引用或者指针调用虚函数时，会先从该对象的头部取到虚函数的地址（C++标准规定虚函数表地址必须放最前），再从虚函数表中取到实际要调用的函数的地址，最终调用该函数，汇编代码大概如下：

[cpp] view plain copy

```
1.              sum += v.at(i);         //要调用 at 函数
2.  00CF1305  mov         eax,dword ptr [ebx]     //取到对象的虚函数表地址
3.  00CF1307  mov         edx,dword ptr [eax+4]   //取到实际 VirtualVector 类的 at 函数地址，因为 at 是第二个虚函数，所以要+4，如果是 clear 则+8，push_back 则不加
4.  00CF130A  push        esi                     //参数压栈
5.  00CF130B  mov         ecx,ebx
6.  00CF130D  call        edx                     //调用真正的 VirtualVector 类的 at 函数
```

所以，我们可以看到调用虚函数，相比普通函数，实际上多了三条汇编指令（取虚表，取函数地址，call 调用）。

至于虚函数如何影响 cpu 的流水线，则主要是因为 call 指令，具体可以看这个网址的演示：

CPU 流水线的一个演示：http://www.pictutorials.com/Instruction_Pipeline.html

第 3 点，编译器不能内联优化，则是因为在得到子类的引用或者指针之前，根本不知道要调用哪一个函数，所以无从内联。

但是，要注意的是，**对于子类直接调用虚函数，是可以内联优化的。**如以下的代码，编译器是完全可以内联展开的。

[cpp] view plain copy

```
VirtualVector v(100);
v.push_back(1);
v.at(0);
```

二、实际测试

光说不练显然不行，下面用代码来测试下虚函数到底有多慢。

下面的代码只是测试用，不考虑细节。

Vector 类包装了一个数组，提供 push_back，at，clear 函数。

VirtualVector 类继承了 IVector，同样实现了 push_back，at，clear 函数，但是都是虚函数。

[cpp] view plain copy

```
#include <iostream>
#include <time.h>
#include <vector>
using namespace std;

const int size = 100000000;

class Vector{
private:
    int *array;
    int pos;
public:
    Vector(int size):array(new int[size]),pos(0)
    {
    }
    void push_back(int val)
    {
        array[pos++] = val;
    }
    int at(int i)
    {
        return array[i];
    }
    void clear()
    {
        pos = 0;
    }
};
class IVector{
```

```
public:
    virtual void push_back(int val) = 0;
    virtual int at(int i) = 0;
    virtual void clear() = 0;
    virtual ~IVector() {};
};

class VirtualVector : public IVector{
public:
    int *array;
    int pos;
public:
    VirtualVector(int size):array(new int[size]),pos(0)
    {
    }
    void push_back(int val)
    {
        array[pos++] = val;
    }
    int at(int i)
    {
        return array[i];
    }
    void clear()
    {
        pos = 0;
    }
    ~VirtualVector()
    {
        if(array != NULL)
            delete array;
    }
};

void testVectorPush(Vector& v){
    v.clear();

    clock_t nTimeStart;      //计时开始
    clock_t nTimeStop;       //计时结束
    nTimeStart = clock();    //
    for(int i = 0; i < size; ++i)
    {
        v.push_back(i);
        //cout<<v.size()<<endl;
```

```
74.     }
75.     nTimeStop = clock();    //
76.     cout <<"耗时: "<<(double)(nTimeStop - nTimeStart)/CLOCKS_PER_SEC<<"秒
    "<< endl;
77. }
78.
79. void testVectorAt(Vector& v)
80. {
81.     clock_t nTimeStart;      //计时开始
82.     clock_t nTimeStop;       //计时结束
83.     int sum = 0;
84.     nTimeStart = clock();    //
85.     for(int j = 0; j < 1; ++j)
86.     {
87.         for(int i = 0; i < size; ++i)
88.         {
89.             sum += v.at(i);
90.         }
91.     }
92.
93.     nTimeStop = clock();    //
94.     cout <<"耗时: "<<(double)(nTimeStop - nTimeStart)/CLOCKS_PER_SEC<<"秒
    "<< endl;
95.     cout<<"sum:"<<sum<<endl;
96. }
97.
98. void testVirtualVectorPush(IVector& v)
99. {
100.     v.clear();
101.
102.     clock_t nTimeStart;      //计时开始
103.     clock_t nTimeStop;       //计时结束
104.     nTimeStart = clock();    //
105.     for(int i = 0; i < size; ++i)
106.     {
107.         v.push_back(i);
108.         //cout<<v.size()<<endl;
109.     }
110.     nTimeStop = clock();    //
111.     cout <<"耗时: "<<(double)(nTimeStop - nTimeStart)/CLOCKS_PER_SEC<<"秒
    "<< endl;
112. }
113.
114. void testVirtualVectorAt(IVector& v)
```

```
{
    clock_t nTimeStart;      //计时开始
    clock_t nTimeStop;       //计时结束
    int sum = 0;
    nTimeStart = clock();    //
    for(int j = 0; j < 1; ++j)
    {
        for(int i = 0; i < size; ++i)
        {
            sum += v.at(i);
        }
    }

    nTimeStop = clock();    //
    cout <<"耗时: "<<(double)(nTimeStop - nTimeStart)/CLOCKS_PER_SEC<<"秒
"<< endl;
    cout<<"sum:"<<sum<<endl;
}

int main()
{
    cout<<sizeof(VirtualVector)<<endl;


    Vector *v = new Vector(size);
    VirtualVector *V = new VirtualVector(size);

    cout<<"testVectorPush:"<<endl;
    testVectorPush(*v);
    testVectorPush(*v);
    testVectorPush(*v);
    testVectorPush(*v);

    cout<<"testVirtualVectorPush:"<<endl;
    testVirtualVectorPush(*V);
    testVirtualVectorPush(*V);
    testVirtualVectorPush(*V);
    testVirtualVectorPush(*V);

    cout<<"testVectorAt:"<<endl;
    testVectorAt(*v);
    testVectorAt(*v);
    testVectorAt(*v);
    testVectorAt(*v);
```

```
158.
159.     cout<<"testVirtualVectorAt:"<<endl;
160.     testVirtualVectorAt(*V);
161.     testVirtualVectorAt(*V);
162.     testVirtualVectorAt(*V);
163.     testVirtualVectorAt(*V);
164.
165.     return 0;
166. }
```

上面的是只有一层继承的情况时的结果，**尽管从虚函数的实现角度来看，多层继承和一层继承调用虚函数的效率都是一样的。**

但是为了测试结果更加可信，下面是一个 6 层继承的测试代码（为了防止编译器的优化，有很多垃圾代码）：

[cpp] view plain copy

```cpp
#include <iostream>
#include <time.h>
#include <vector>
using namespace std;

const int size = 100000000;

class Vector{
private:
    int *array;
    int pos;
public:
    Vector(int size):array(new int[size]),pos(0)
    {
    }
    void push_back(int val)
    {
        array[pos++] = val;
    }
    int at(int i)
    {
        return array[i];
    }
    void clear()
    {
        pos = 0;
```

```
    }
};
class IVector{
public:
    virtual void push_back(int val) = 0;
    virtual int at(int i) = 0;
    virtual void clear() = 0;
    virtual ~IVector() {};
};

class VirtualVector1 : public IVector{
public:
    int *array;
    int pos;
public:
    VirtualVector1(int size):array(new int[size]),pos(0)
    {
    }
    void push_back(int val)
    {
        array[1] = val;
    }
    int at(int i)
    {
        return array[1];
    }
    void clear()
    {
        pos = 0;
    }
    ~VirtualVector1()
    {
        if(array != NULL)
            delete array;
    }
};

class VirtualVector2 : public VirtualVector1{
public:
    VirtualVector2(int size):VirtualVector1(size)
    {
    }
    void push_back(int val)
    {
```

```
        array[2] = val;
    }
    int at(int i)
    {
        return array[2];
    }
    void clear()
    {
        pos = 0;
    }
};

class VirtualVector3 : public VirtualVector2{
public:
    VirtualVector3(int size):VirtualVector2(size)
    {
    }
    void push_back(int val)
    {
        array[3] = val;
    }
    int at(int i)
    {
        return array[3];
    }
    void clear()
    {
        pos = 0;
    }
};

class VirtualVector4 : public VirtualVector3{
public:
    VirtualVector4(int size):VirtualVector3(size)
    {
    }
    void push_back(int val)
    {
        array[4] = val;
    }
    int at(int i)
    {
        return array[4];
    }
```

```
115.     void clear()
116.     {
117.         pos = 0;
118.     }
119. };
120. 
121. class VirtualVector5 : public VirtualVector4{
122. public:
123.     VirtualVector5(int size):VirtualVector4(size)
124.     {
125.     }
126.     void push_back(int val)
127.     {
128.         array[5] = val;
129.     }
130.     int at(int i)
131.     {
132.         return array[5];
133.     }
134.     void clear()
135.     {
136.         pos = 0;
137.     }
138. };
139. 
140. class VirtualVector6 : public VirtualVector5{
141. public:
142.     VirtualVector6(int size):VirtualVector5(size)
143.     {
144.     }
145.     void push_back(int val)
146.     {
147.         array[6] = val;
148.     }
149.     int at(int i)
150.     {
151.         return array[6];
152.     }
153.     void clear()
154.     {
155.         pos = 0;
156.     }
157. };
158. 
```

```
159. class VirtualVector : public VirtualVector6{
160. public:
161.     VirtualVector(int size):VirtualVector6(size)
162.     {
163.     }
164.     void push_back(int val)
165.     {
166.         array[pos++] = val;
167.     }
168.     int at(int i)
169.     {
170.         return array[i];
171.     }
172.     void clear()
173.     {
174.         pos = 0;
175.     }
176. };
177.
178. void testVectorPush(Vector& v){
179.     v.clear();
180.
181.     clock_t nTimeStart;      //计时开始
182.     clock_t nTimeStop;       //计时结束
183.     nTimeStart = clock();    //
184.     for(int i = 0; i < size; ++i)
185.     {
186.         v.push_back(i);
187.         //cout<<v.size()<<endl;
188.     }
189.     nTimeStop = clock();    //
190.     cout <<"耗时: "<<(double)(nTimeStop - nTimeStart)/CLOCKS_PER_SEC<<"秒
    "<< endl;
191. }
192.
193. void testVectorAt(Vector& v)
194. {
195.     clock_t nTimeStart;      //计时开始
196.     clock_t nTimeStop;       //计时结束
197.     int sum = 0;
198.     nTimeStart = clock();    //
199.     for(int j = 0; j < 1; ++j)
200.     {
201.         for(int i = 0; i < size; ++i)
```

```
202.         {
203.             sum += v.at(i);
204.         }
205.     }
206. 
207.     nTimeStop = clock();    //
208.     cout <<"耗时: "<<(double)(nTimeStop - nTimeStart)/CLOCKS_PER_SEC<<"秒
    "<< endl;
209.     cout<<"sum:"<<sum<<endl;
210. }
211. 
212. void testVirtualVectorPush(IVector& v)
213. {
214.     v.clear();
215.     clock_t nTimeStart;      //计时开始
216.     clock_t nTimeStop;       //计时结束
217.     nTimeStart = clock();    //
218.     for(int i = 0; i < size; ++i)
219.     {
220.         v.push_back(i);
221.         //cout<<v.size()<<endl;
222.     }
223.     nTimeStop = clock();    //
224.     cout <<"耗时: "<<(double)(nTimeStop - nTimeStart)/CLOCKS_PER_SEC<<"秒
    "<< endl;
225. }
226. 
227. void testVirtualVectorAt(IVector& v)
228. {
229.     clock_t nTimeStart;      //计时开始
230.     clock_t nTimeStop;       //计时结束
231.     int sum = 0;
232.     nTimeStart = clock();    //
233.     for(int j = 0; j < 1; ++j)
234.     {
235.         for(int i = 0; i < size; ++i)
236.         {
237.             sum += v.at(i);
238.         }
239.     }
240. 
241.     nTimeStop = clock();    //
242.     cout <<"耗时: "<<(double)(nTimeStop - nTimeStart)/CLOCKS_PER_SEC<<"秒
    "<< endl;
```

```
    cout<<"sum:"<<sum<<endl;
}

int main()
{
    cout<<sizeof(VirtualVector)<<endl;
    {
        auto v = VirtualVector1(size);
        v.push_back(0);
        cout<<v.at(0)<<endl;
    }
    {
        auto v = VirtualVector2(size);
        v.push_back(0);
        cout<<v.at(0)<<endl;
    }
    {
        auto v = VirtualVector3(size);
        v.push_back(0);
        cout<<v.at(0)<<endl;
    }
    {
        auto v = VirtualVector4(size);
        v.push_back(0);
        cout<<v.at(0)<<endl;
    }
    {
        auto v = VirtualVector5(size);
        v.push_back(0);
        cout<<v.at(0)<<endl;
    }
    {
        auto v = VirtualVector6(size);
        v.push_back(0);
        cout<<v.at(0)<<endl;
    }

    auto *v = new Vector(size);
    auto *V = new VirtualVector(size);

    cout<<"testVectorPush:"<<endl;
    testVectorPush(*v);
    testVectorPush(*v);
    testVectorPush(*v);
```

```
287.     testVectorPush(*v);
288.
289.     cout<<"testVirtualVectorPush:"<<endl;
290.     testVirtualVectorPush(*V);
291.     testVirtualVectorPush(*V);
292.     testVirtualVectorPush(*V);
293.     testVirtualVectorPush(*V);
294.
295.     cout<<"testVectorAt:"<<endl;
296.     testVectorAt(*v);
297.     testVectorAt(*v);
298.     testVectorAt(*v);
299.     testVectorAt(*v);
300.
301.     cout<<"testVirtualVectorAt:"<<endl;
302.     testVirtualVectorAt(*V);
303.     testVirtualVectorAt(*V);
304.     testVirtualVectorAt(*V);
305.     testVirtualVectorAt(*V);
306.
307.     return 0;
308. }
```

测试结果：

测试结果都取最小时间
1 层继承的测试结果：

	push_back	at
Vector	0.263s	0.04s
VirtualVector	0.331s	0.222s
倍数	1.25	5.55

6 层继承的测试结果：

	push_back	at
Vector	0.262s	0.041s
VirtualVector	0.334s	0.223s
倍数	1.27	5.43

一、可以看出继承层数和虚函数调用效率无关

二、可以看出虚函数慢得有点令人发指了，对于 at 操作，虚函数花的时间竟然是普通函数的 5.5 倍！

但是，再看看，我们可以发现对于 push_back 操作，虚函数花的时间是普通函数的 1.25 倍。

why?

再分析下代码，我们可以发现 at 操作的逻辑，明显要比 push_back 的逻辑要简单。

虚函数额外消耗时间为 vt，函数本身所消耗时间为 ft，则有以下

倍数 = (vt + ft)/ft = 1 + vt/ft

显然当 ft 越大，即函数本身消耗时间越长，则倍数越小。

那么让我们在 at 操作中加了额外代码，统计下 1 到 100 之和：

[cpp] view plain copy

```cpp
int at(int i)
{
    sssForTest = 0;
    for(int j = 0; j < 100; ++j)
        sssForTest += j;
    return array[i];
}
```

测试代码：

[cpp] view plain copy

```cpp
#include <iostream>
#include <time.h>
#include <vector>
using namespace std;

const int size = 100000000;
int sssForTest = 0;

class Vector{
private:
    int *array;
    int pos;
public:
    Vector(int size):array(new int[size]),pos(0)
    {
    }
    void push_back(int val)
    {
        array[pos++] = val;
    }
    int at(int i)
```

```
    {
        sssForTest = 0;
        for(int j = 0; j < 100; ++j)
            sssForTest += j;
        return array[i];
    }
    void clear()
    {
        pos = 0;
    }
};
class IVector{
public:
    virtual void push_back(int val) = 0;
    virtual int at(int i) = 0;
    virtual void clear() = 0;
    virtual ~IVector() {};
};

class VirtualVector : public IVector{
public:
    int *array;
    int pos;
public:
    VirtualVector(int size):array(new int[size]),pos(0)
    {
    }
    void push_back(int val)
    {
        array[pos++] = val;
    }
    int at(int i)
    {
        sssForTest = 0;
        for(int j = 0; j < 100; ++j)
            sssForTest += j;
        return array[i];
    }
    void clear()
    {
        pos = 0;
    }
    ~VirtualVector()
    {
```

```
        if(array != NULL)
            delete array;
    }
};

void testVectorPush(Vector& v){
    v.clear();

    clock_t nTimeStart;      //计时开始
    clock_t nTimeStop;       //计时结束
    nTimeStart = clock();    //
    for(int i = 0; i < size; ++i)
    {
        v.push_back(i);
        //cout<<v.size()<<endl;
    }
    nTimeStop = clock();    //
    cout <<"耗时: "<<(double)(nTimeStop - nTimeStart)/CLOCKS_PER_SEC<<"秒"<< endl;
}

void testVectorAt(Vector& v)
{
    clock_t nTimeStart;      //计时开始
    clock_t nTimeStop;       //计时结束
    int sum = 0;
    nTimeStart = clock();    //
    for(int j = 0; j < 1; ++j)
    {
        for(int i = 0; i < size; ++i)
        {
            sum += v.at(i);
        }
    }

     nTimeStop = clock();    //
     cout <<"耗时: "<<(double)(nTimeStop - nTimeStart)/CLOCKS_PER_SEC<<"秒"<< endl;
     cout<<"sum:"<<sum<<endl;
}

void testVirtualVectorPush(IVector& v)
{
     v.clear();
```

```
108.
109.     clock_t nTimeStart;      //计时开始
110.     clock_t nTimeStop;       //计时结束
111.     nTimeStart = clock();    //
112.     for(int i = 0; i < size; ++i)
113.     {
114.         v.push_back(i);
115.         //cout<<v.size()<<endl;
116.     }
117.     nTimeStop = clock();     //
118.     cout <<"耗时: "<<(double)(nTimeStop - nTimeStart)/CLOCKS_PER_SEC<<"秒
    "<< endl;
119. }
120.
121. void testVirtualVectorAt(IVector& v)
122. {
123.     clock_t nTimeStart;      //计时开始
124.     clock_t nTimeStop;       //计时结束
125.     int sum = 0;
126.     nTimeStart = clock();    //
127.     for(int j = 0; j < 1; ++j)
128.     {
129.         for(int i = 0; i < size; ++i)
130.         {
131.             sum += v.at(i);
132.         }
133.     }
134.
135.     nTimeStop = clock();     //
136.     cout <<"耗时: "<<(double)(nTimeStop - nTimeStart)/CLOCKS_PER_SEC<<"秒
    "<< endl;
137.     cout<<"sum:"<<sum<<endl;
138. }
139.
140. int main()
141. {
142.     cout<<sizeof(VirtualVector)<<endl;
143.
144.
145.     Vector *v = new Vector(size);
146.     VirtualVector *V = new VirtualVector(size);
147.
148.     cout<<"testVectorPush:"<<endl;
149.     testVectorPush(*v);
```

```
150.     testVectorPush(*v);
151.     testVectorPush(*v);
152.     testVectorPush(*v);
153. 
154.     cout<<"testVirtualVectorPush:"<<endl;
155.     testVirtualVectorPush(*V);
156.     testVirtualVectorPush(*V);
157.     testVirtualVectorPush(*V);
158.     testVirtualVectorPush(*V);
159. 
160.     cout<<"testVectorAt:"<<endl;
161.     testVectorAt(*v);
162.     testVectorAt(*v);
163.     testVectorAt(*v);
164.     testVectorAt(*v);
165. 
166.     cout<<"testVirtualVectorAt:"<<endl;
167.     testVirtualVectorAt(*V);
168.     testVirtualVectorAt(*V);
169.     testVirtualVectorAt(*V);
170.     testVirtualVectorAt(*V);
171. 
172.     return 0;
173. }
```

at 操作中增加求和后的统计结果：

	push_back	at 增加求和代码
Vector	0.265s	6.893s
VirtualVector	0.328s	7.125s
倍数	1.23	1.03

只是简单增加了一个求和代码，我们可以看到，倍数变成了 1.03，也就是说虚函数的消耗基本可以忽略了。

所以说，虚函数的效率到底低不低和实际要调用的函数的耗时有关，当函数本身的的耗时越长，则虚函数的影响则越小。

再从另一个角度来看，一个虚函数调用到底额外消耗了多长时间？

从统计数据来看 100,000,000 次函数调用，虚函数总共额外消耗了 0.05~0.23 秒（VirtualVector 对应时间减去 Vector 时间），

也就是说，1 亿次调用，虚函数额外花的时间是 0.x 到 2.3 秒。

也就是说，如果你有个函数，要被调用 1 亿次，而这 1 亿次调用所花的时间是几秒，十几秒，且你不能容忍它慢一二秒，那么就干掉虚函数吧^_^。

三、总结：

1. **虚函数调用效率和继承层数无关；**
2. **其实虚函数还是挺快的。**
3. **如果真的要完全移除虚函数，那么如果要实现运行时多态，则要用到函数指针，据上面的分析，函数指针基本具有虚函数的所有缺点（要传递函数指针，同样无法内联，同样影响流水线），且函数指针会使代码混乱。**

BTW：测试 cpu 是 i5。

TODO：测试指针函数，boost::bind 的效率