C++ ABI for IA-64: Code and Implementation Examples

Revised 5 September 2000

Introduction

In this document, we provide a number of code examples as illustration or tests for the ABI specifications.

Contents

Base Class Layout

The following test programs check for correct sharing of the virtual pointer with a virtual base class.

```
/*
Test case for sharing virtual bases.
In Derived_too,
the primary base class is NewShareme,
The bases Base and Shareme share vptr's
with Derived and are allocated at the
same offset as Derived.
Should get:
60% a.out
(long)(NewShareme *)dd - (long)dd = 0
(long)(Derived *)dd - (long)dd = 8
(long)(Base *)dd - (long)dd = 8
(long)(Shareme *)dd - (long)dd = 8
*/

struct Shareme {
    virtual void foo();
};
struct Base : virtual Shareme {
        virtual void bar();
};
struct Derived : virtual Base {
        virtual void baz();
```

```
};

struct NewShareme {
        virtual void foo();
};

struct Derived_too : virtual NewShareme, virtual Derived {
        virtual void bar();
};

void Shareme::foo() { }
void Base::bar() { }
void Derived::baz() { }
void NewShareme::foo() { }
void Derived_too::bar() { }


extern "C" int printf(const char *,...);
#define EVAL(EXPR) printf( #EXPR " = %d\n", (EXPR) );
main()
{
  Derived_too *dd = new Derived_too;
  EVAL((long)(NewShareme *)dd - (long)dd);
  EVAL((long)(Derived *)dd - (long)dd);
  EVAL((long)(Base *)dd - (long)dd);
  EVAL((long)(Shareme *)dd - (long)dd);
}
```

```
/*
Test case for sharing virtual bases.
In Most_Derived,
the primary base class is Nonvirt1,
Nonvirt2 and Nonvirt3 share vptrs with
virtual base Shared_Virt.  Shared_Virt
should be at the same offset as Nonvirt2.
Should get:
67% a.out
(long)(Nonvirt1 *)dd - (long)dd = 0
(long)(Nonvirt2 *)dd - (long)dd = 8
(long)(Nonvirt3 *)dd - (long)dd = 16
(long)(Shared_Virt *)dd - (long)dd = 8
*/

struct Shared_Virt {
    virtual void foo();
};
struct Nonvirt2 : virtual Shared_Virt {
        virtual void bar();
};
struct Nonvirt3 : virtual Shared_Virt {
        virtual void baz();
};
struct Nonvirt1 {
        virtual void foo();
};

struct Most_Derived : Nonvirt1, Nonvirt2, Nonvirt3 {
        virtual void bar();
};
```

```
void Shared_Virt::foo() { }
void Nonvirt2::bar() { }
void Nonvirt3::baz() { }
void Nonvirt1::foo() { }
void Most_Derived::bar() { }

extern "C" int printf(const char *,...);
#define EVAL(EXPR) printf( #EXPR " = %d\n", (EXPR) );
main()
{
  Most_Derived *dd = new Most_Derived;
  EVAL((long)(Nonvirt1 *)dd - (long)dd);
  EVAL((long)(Nonvirt2 *)dd - (long)dd);
  EVAL((long)(Nonvirt3 *)dd - (long)dd);
  EVAL((long)(Shared_Virt *)dd - (long)dd);
}
```

```
/*
Test case for sharing virtual bases.
In Most_Derived, share the vptr with
Interface1 but not Interface3, since
Interface3 is indirectly inherited.

Should get:
(long)(Interface1 *)dd - (long)dd = 0
(long)(Interface2 *)dd - (long)dd = 8
(long)(Interface3 *)dd - (long)dd = 8
(long)(Concrete1 *)dd - (long)dd = 8
*/

struct Interface1 {
    virtual void foo();
};
struct Interface2 : virtual Interface1 {
        virtual void bar();
};
struct Interface3 : virtual Interface2 {
        virtual void baz();
};

struct Concrete1 : virtual Interface3 {
        virtual void foo();
        int i; // important.
};

struct Most_Derived : virtual Interface1,
                      virtual Interface2,
                      virtual Concrete1 {
        virtual void bar();
};

void Interface1::foo() { }
void Interface2::bar() { }
void Interface3::baz() { }
void Concrete1::foo() { }
void Most_Derived::bar() { }
```

```
extern "C" int printf(const char *,...);
#define EVAL(EXPR) printf( #EXPR " = %d\n", (EXPR) );
main()
{
  Most_Derived *dd = new Most_Derived;
  EVAL((long)(Interface1 *)dd - (long)dd);
  EVAL((long)(Interface2 *)dd - (long)dd);
  EVAL((long)(Interface3 *)dd - (long)dd);
  EVAL((long)(Concrete1 *)dd - (long)dd);
}
```

Virtual Function Calls

The following class definitions are intended to illustrate various cases relevant to the entries found in vtables to support virtual functions. We define a number of classes, many of which define a virtual function f. We are interested only in the vtable contents supporting calls of f; other contents of the classes are generally elided, except where necessary to emphasize some situation, and we do not attempt to describe other contents of the vtables, including precise layout.

The left column of this table contains the class definitions. The right column(s) describe the corresponding vtable contents as they are required to support calls to f. We use the notation **&C::f** to mean a vtable entry for the instance of f defined in class C, in the form defined by the ABI (currently a function pointer/GP pair). We use the notation **&thunk(A*,C::f)** to mean a vtable entry for a thunk which converts **this** from A* to C* and then calls C::f. We use the notation **vcall(A*,C*,f)** to mean a vcall offset that is added to an A* to produce a C* for use by an f thunk.

Class definition	Vtable 1 contents	Vtable 2 contents
struct X { virtual void u(); }	Class X is an auxiliary class to be used only to prevent interesting classes from being primary bases later.	
struct A { virtual void f(); }	*Vtable A*	
	&A::f	
struct B: public A { }	*Vtable A_in_B*	
	&A::f	
struct C: public A { virtual void f(); }	*Vtable A_in_C (primary)*	
	&C::f	
struct D: public X, public A { }	*Vtable A_in_D*	
	&A::f	
struct E: public X, public A { virtual void f(); }	*Vtable A_in_E*	*Vtable E*
	&thunk(A*,E::f)	&E::f
struct G: public virtual A	*Vtable A_in_G*	

`{ }`	&A::f	
`struct H:` `  public X,` `  public virtual A` `{` `  virtual void f();` `}`	*Vtable A_in_H*	*Vtable H*
	&thunk(A*,H::f) vcall(H::A*,H*,f)	&H::f
`struct I:` `  public H` `{ int i; }`	*Vtable A_in_I*	*Vtable H_in_I*
	&thunk(A*,H::f) vcall(I::A*,I::H*,f)	&H::f

Implementation

There are several possible implementations of the thunks given the above information. Note in the following that we assume that prior to calling any vtable entry, the **this** pointer has been adjusted to point to the subobject corresponding to the vtable from which the vptr is fetched.

A. Since the offsets are always known at compile time, even for virtual bases, each thunk could be distinct, adding the known offset to **this** and branching to the target function.

   This would result in a thunk for each overrider at a distinct offset. As a result, a branch mispredict and possibly an instruction cache miss would occur each time the actual type changed for a reference at any given point in the code.

B. In the case of virtual inheritance, the offset, although known when the overrider is declared, may differ depending on derivations from the overrider's class. H and I above are the simplest example. H is a primary base for I, but the int member of I means that A is at a different offset from H in I than it was from a standalone H. Because of this, the ABI specifies that the secondary vtable for a virtual base A contain a vcall offset to H, so that a shared thunk can load the vcall offset, adding it to **this**, and branch to the target function H::f.

   This would result in fewer thunks, since for a inheritance hierarchy where A is a virtual base of H, and H::f overrides A::f, all instances of H in a larger hierarchy can use the same thunk. As a result, these thunks will cause fewer branch mispredictions and instruction cache misses. The tradeoff is that they must do a load before the offset add. Since the offset is smaller than the code for a thunk, the load should miss in cache less frequently, so better cache miss behavior should produce better results in spite of the 2 or more cycles required for the vcall offset load.

C. In the case of non-virtual inheritance, when an overrider is declared, the entire set of associated overriding and overridden functions f is known along with their associated vcall offsets, and they are all constants. Therefore, it is possible to cascade the offset adds as follows. Suppose we have A::f, overridden by B::f at offset -16, and finally overridden by C::f at offset -32

from B, -48 from A. Assuming that **this** is in register **out0**, the IA-64 code can look like:

```
thunk-A-to-C::f :
    add out0 = -16,out0 ;;
thunk-A-to-C::f :
    add out0 = -32,out0 ;;
C::f :
    # normal entry code...
```

This code will leave us with about the same number of branch mispredictions for the thunk calls, but should eliminate a large proportion of the instruction cache misses, and costs one cycle per level in the adjustment sequence. Note that there is no particular required order of the thunk entries, so the compiler could optimize by putting the most frequent one closest to the target function and so on. Also, if the sequence gets too long, any entry can add its entire offset at once and branch directly to the

In order to make these implementations practical, the ABI must specify:

- where the vcall offsets are allocated in the secondary vtables,
- where the thunks are required to be emitted, and
- the names (mangled) of the thunks for reference elsewhere.

The emission of the thunks is the most relevant issue here -- the others are independent of this discussion. Cascading thunks as in (C) only works if they appear with the definition of the overriding function, so that seems to be the right answer.

Assuming that all of the above optimizations are implemented, we note a number of benefits of the design specified.

- In the case of single, non-virtual inheritance, calling a virtual function requires no adjustment to the `this' pointer, and no vcall offsets. This is in keeping with the guiding principle that "you shouldn't pay for features you don't use."

- If the static type of an object is the same as its dynamic type, then no adjustment to the `this' pointer is required.

- All thunks can be emitted in the same translation unit as the overriding function.

- Thunks from classes that are not morally virtual bases do not require a branch to the non-adjusting entry point. (We say that a subobject X is a "morally virtual" base of Y if X is an indirect or direct base class of Y, and if X is either a virtual base of Y, or the direct or indirect base of a virtual base of Y.)

  Suppose we have classes M1 and M2, base classes of a third class V. Suppose V is a virtual base of C, and A2 and A1 are non-virtual bases of C. Suppose all of these classes have definitions of virtual function f. We suggest the following implementation:

```
m2: /* Thunk for morally virtual base M2.  */
    this += offsetof (V, M1) - offsetof (V, M2)
```

```
m1: /* Thunk for morally virtual base M1.  */
    this -= offsetof (V, M1)
v:  /* Thunk for virtual base V.  */
    this += vcall offset stored in V vtable
    goto f;
a2: /* Thunk for non-virtual base A2.  */
    this += offsetof (B, A1) - offsetof (B, A2)
    /* Fall through.  */
a1: /* Thunk for non-virtual base A1.  */
    this += offsetof (B, C) - offsetof (B, A1)
    /* Fall through.  */
f:  /* Non-adjusting entry point.  */
```

(Here `offsetof' is a compile-time computable function that gives the offset of its second parameter in its first parameter.)

(Alternatively the `v' entry point above could be of the form:

```
v: /* Thunk for virtual base V.  */
   this += vcall offset stored in V vtable
           - offsetof (B, C) + offsetof (B, A2)
   /* Fall through.  */
```

Which alternative is better depends on how many adds follow at this point. In general, if many adds remain before the non-adjusting entry point, it may be better to suffer the consequences of the indirect branch.)

In this way, a virtual call through a base A1, A2, etc., that is not a virtual base of C (or a direct or indirect base of a virtual base of C), does not require an additional branch, and is therefore more likely to avoid icache misses. Even the thunk for V may avoid severe icache penalties since it is located near the non-adjusting entry point for f. Furthermore, if there are no non-virtual bases, then the sequence can become just:

```
v: /* Thunk for virtual base V.  */
   this += vcall offset stored in V vtable
f: /* Non-adjusting entry point.  */
```

Vtables During Object Construction

Following is a test program from Compaq, that breaks on many compilers.

```
/*
This test program should output:

V1 called
V2 called
C called
C::foo called 7
PASSED this == cp
D called
~C called
C::foo called 7
PASSED this == cp
~V2 called
~V1 called
```

```
Int caught
*/

extern "C" int printf(const char *,...);
struct V1 {
        int v;
        virtual int foo();
        V1();
        ~V1();
};
struct V2 : virtual V1 {
        int v2;
        virtual int foo();
        V2();
        ~V2();
};
struct C : virtual V1, virtual V2 {
        int c;
        virtual int foo();
        C();
        ~C();
};

struct B {
   int b; };
struct D : B, C {
        int d;
        virtual int bar();
        D();
        ~D();
};
extern "C" int printf(const char *,...);
main()
{
  try {
        D *d = new D;
        delete d;
  } catch (int) {
        printf("Int caught\n");
  }
}

int V1::foo() {
   printf("V1::foo called\n"); return 1; }
V1::V1() : v(5) {
   printf("V1 called\n"); }
V1::~V1() {
   printf("~V1 called\n"); }

int V2::foo() {
   printf("V2::foo called\n"); return 1; }
V2::V2() : v2(6) {
   printf("V2 called\n"); }
V2::~V2() {
   printf("~V2 called\n"); }

int C::foo() {
   printf("C::foo called %d\n", c); return 1; }
C::C() : c(7) {
```

```
        printf("C called\n");
        V1 *vv = this; vv->foo();
        C *cp = dynamic_cast<C *>(vv);
        if (this == cp) {
            printf("PASSED this == cp\n");
        } else {
            printf("FAILED this != cp\n");
        }
}
C::~C() {
        printf("~C called\n");
        V1 *vv = this; vv->foo();
        C *cp = dynamic_cast<C *>(vv);
        if (this == cp) {
            printf("PASSED this == cp\n");
        } else {
            printf("FAILED this != cp\n");
        }
}

int D::bar() {
   printf("D::bar called\n"); return 1; }
D::D() : d(8) {
   printf("D called\n"); throw 5; }
D::~D() {
   printf("~D called\n"); }
```

External Name Mangling

In the table below, **Ret?** in the source name indicates a function for which the return type is not part of the mangling. **Type?** indicates a data object (for which the type is never part of the mangling). Spaces appear in some of the mangled names to assist a human parser -- they are not part of the actual mangled name, and should be ignored.

Mangled name ***(ignore spaces)***	**Source name**
f	C function or variable "f" or a global namespace variable "f"
_Z1fv	Ret? f(); *or* Ret? f(void);
_Z1fi	Ret? f(int);
_Z3foo3bar	Ret? foo(bar);
Zrm1XS	Ret? operator%(X, X);
ZplR1XS0	Ret? operator+(X&, X&);
ZlsRK1XS1	Ret? operator<< (X const&, X const&); (Note: X is S_, X const is S0_)

_ZN3FooIA4_iE3barE	Type? Foo<int[4]>::bar;
_Z1fIiEvi	void f<int>(/*nondependent*/int); (Note: the return type is always explicitly encoded for template functions taking parameters.)
Z5firstI3DuoEvS0	void first<Duo>(/*nondependent*/Duo); (Note: first template is S_, Duo is S0_, first(Duo) is S1_. Since the function parameter is not dependent, don't use T_.)
Z5firstI3DuoEvT	void first<Duo>(/*T1=*/Duo);
_Z3fooIiPFidEiEvv	void foo<int,int(*)(double),int>(); (Note: return type encoded for template function.)
_ZN1N1fE	Type? N::f
_ZN6System5Sound4beepEv	Ret? System::Sound::beep();
_ZN5Arena5levelE	Type? Arena::level;
_ZN5StackIiiE5levelE	Type? Stack<int, int>::level;
_Z1fI1XE vPV N1AIT_E1TE	void f<X>(A</*T1=*/X>::T volatile*);
_ZngILi42EE v N1A I XplT_Li2EE E 1TE	void operator—</*int J=*/42>(A<J+2>::T);
_Z4makeI7FactoryiE T_IT0_E v	/*T1=*/Factory</*T2=*/int> make<Factory, int>(); (Note: T_ = factory (a template), T0_ = int)
_Z3foo 5Hello5WorldS0_S_	Type? foo(Hello,World,World,Hello) (Note: Hello is S_, World is S0_, foo(...) is S1_)
_Z3fooPM2ABi	foo(int AB::**) // M is a pointer, P adds another level
_ZlsRSoRKSs	operator<< (std::ostream&,std::string const&)
_ZTI7a_class	typeid(class a_class)

Local Scope Mangling

Following are several examples illustrating local scope mangling, all based on the following code snippet:

```
    class A {
      void foo (int) {
        class B { };
      }
    };

    void foo () {
      class C {
        class D { };
        void bar () {
          struct E {
            void baz() { }
          };
        }
      };
    }
```

First consider the mangling of A::foo::B, which is the case of a simple local name (B) in a nested function.

```
  <encoding> ::- <name>
             ::- <local-name>
             ::- Z <function encoding> E <entity name>
```

where:

```
  <function encoding> ::- <name> <bare-function-type>
                      ::- <nested-name> <bare-function-type>
                      ...
                      ::- N <unqualified-name> <unqualified-name> E <bare-function-type>
                      ::- N 1A 3foo E i
  <entity name> ::- <name>
                ::- 1B
```

So the final result is "Z N 1A 3foo E i E 1B". The next example is foo::C::D, i.e. a nested local name in a simple function:

```
  <encoding> ::- <name>
             ::- <local-name>
             ::- Z <function encoding> E <entity name>
```

where:

```
  <function encoding> ::- <name> <bare-function-type>
                      ::- 3foo v
  <entity name> ::- <nested-name>
                ...
                ::- N <unqualified-name> <unqualified-name> E
                ::- N 1C 1D E
```

So the final result is "Z 3foo v E N 1C 1D E". Finally, as an example of nested local scopes, consider foo::C::bar::E::baz:

```
  <encoding> ::- <name>
             ::- <local-name>
             ::- Z <function encoding> E <entity name>
```

where:

```
  <function encoding> ::- <name> <bare-function-type>    (bar)
                      ::- <local-name> v
                      ::- Z <function encoding> E <entity name> v
                      ::- Z <name> <bare-function-type> E <nested-name> v
                      ...
                      ::- Z 3foo v E N 1C 3bar E v
  <entity name> ::- <function name>
                ::- <function name> <bare-function-type>
                ::- <nested-name> <bare-function-type>
                ...
                ::- N <unqualified-name> <unqualified-name> E <bare-function-type>
                ::- N 1E 3baz E v
```

yielding the final result:

```
       Z Z 3foo v E N 1C 3bar E v E N 1E 3baz E v
```

Vague Linkage

COMDAT groups are a new gABI feature specified during the IA-64 ABI definition, and may not be implemented by all vendors immediately. Pending their availability, other implementations may be required for the features specified to use COMDAT in the ABI. Using weak symbols for the objects requiring weak linkage instead of putting them in COMDAT groups will normally work, even in interaction with correctly-defined COMDAT group representations, although it will not have the space efficiency of COMDAT. However, weak symbol semantics are not consistent among Unix implementations, and implementors should be careful that whenever multiple symbols are specified to reside in a single COMDAT, and might interact (e.g. thunks falling through to primary function definitions), all are generated together as weak symbols as well.

Please send corrections to Mark Mitchell.