Design of the ABI Test-Suite

By: Sunil Srivastava.

July 2, 2014

This document describes the design of a C++ IA64 ABI test-suite that verifies a compiler's compliance against the IA64 ABI specification to ensure link compatibility. The test-suite currently supports two targets, with an ability to add more. The test-suite consists of slightly over one million unique classes spread over roughly four hundred test files and the full set of tests take approximately an hour to run on a single threaded 3 GHz machine.

A C++ ABI defines how high-level C++ constructs, such as the class inheritance, virtual functions, function overloading, etc. are mapped to the lower-level. The IA64 ABI is one of the most commonly used C++ ABIs, used by both clang/llvm and g++ compilers. The current version of the specification can be found at http://mentorembedded.github.io/cxx-abi/abi.html.

Strict conformance to this definition is needed to ensure that C++ code compiled by different compilers or by different releases of the same compiler can be linked together to make a working application. In some industries, consistency is more important than correctness. Hence the need for a test-suite to guard against changes.

Overview

An ABI test-suite tests a compiler's implementation against an ABI specification, by having C++ code that exercise various parts of the ABI specification, and comparing the code generated by the compiler-under-test to the 'correct value'. However, the 'correct value' of each test is based both on the rules given in the IA64 ABI specification and on the *base-rules*[1], which leads to one of the biggest hurdles towards writing a truly multi-target test-suite.

The test-suite currently supports the following two configurations[2], with the ability of adding others:

1. 64-bit x86-64 clang 3.4, LP64
2. 32 bit x86 clang 3.4, ILP32

The object layout tests are comprised of C/C++ code, automatically generated from class definitions, that are compiled by the compiler-under-test and run on the target machine. The test verifies that the layout done by the compiler-under-test matches the expected layout.

[1] The term base-rules refers to the rules defining the size and alignments of basic types and pointers. These are usually defined by the Base Document in the IA64 ABI specification, section 1.5.
[2] We tested these on a Ubuntu 12.04, but they should work on any modern x86 platform.

A set of 'interesting' classes for the test-suite is generated by:

1. Manually, going over the ABI specification.
2. Automatically, generating 'every possible' class by varying key parameters[3].
3. A tool, capturing the 'essence' of class structures[4] of existing code and regenerating simplified classes that match the original classes in the specifics that matter to the allocation algorithm.

Some points to note are:

- Tests use the 'lit' framework of LLVM.
- Each test file is self-contained with no dependencies except for a common include file.
- Each test file is compiled twice, once as a C file and once as a C++ file[5], and then the two resulting object files are linked with a common utility to generate a runnable test.
- The selection between targets is made by the ABISELECT(…) macro. It has one argument for each target, such as ABISELECT(a,b) where 'a' and 'b' are the expected results for the two targets in a two-target implementation.
- The test files arrange the runtime code to be called automatically using the static constructor mechanism, so that an arbitrary number of files can be linked together to make an executable.

A top-level python script is provided to run the test-suite, as:

```
python lp64tools.py test -v
```

This script takes all normal lit options, for example –j<n> to enable multiple threads.

The tests

The tests verify size and alignment of classes, offsets of all base classes and fields, including bit-fields, positioning of bit-fields, structure of virtual function tables (vtables) and VTTs, including vtables during construction and destruction of base classes and name mangling of all member functions.

Generated test files contain:

- The class definition.
- Empty stubs for member functions, to satisfy vtables.
- Code to do the testing.
- A class descriptor. This is a C structure that describes the class, its vtables, VTTs, base classes etc.

An example test file is provided at the end of this document.

Testing both C and C++

While this is primarily a C++ ABI test-suite, a section of C level tests of structures without any base classes or member functions is included. These C level structures are tested in both C and C++ modes.

[3] Key parameters include: depth and breadth of derivation tree, virtual vs non-virtual bases, number/presence of virtual functions, etc.

[4] Class derivation and virtual function override pattern, and other minor aspects.

[5] The C and C++ parts are separated by `'#ifdef __cplusplus'`. The C++ part contains the class definition and tests, and the C part contains tables that describe expected contents of vtables, VTTs etc. The lit header manages the compile/link/run sequence.

Test-suite coverage

A major part of the test-suite is devoted to object layout and allocation tests and includes tests for the following:

- Size and alignments of classes, as reported by sizeof() and __alignof__() operators.
- Offsets of fields by subtracting the address of a local variable of the class type from the address of fields inside them.
- Position of bit-fields are tested by setting bit-fields to different values at runtime and looking for those values in the structure based on our understanding of where they should be. Values 0, 1 and (1<<(bitfield_size-1)) are tested.
- Offsets of all base classes, both direct and indirect, by subtracting the address of the full object from the address of that base class. In cases of virtual base classes having multiple derivation paths, offsets across every derivation path are tested. In cases of non-virtual base classes appearing multiple times[6], all paths are tested except for those that cannot be written as valid C++[7].
- Contents of vtable variables are tested by looking for variables of expected mangled names and comparing their contents to those specified in the Specification.
- Contents of vtables attached to objects of class types, and to their base classes are tested. This is done while constructors or destructors are being executed[8] and when the full object has been constructed but fore starting its destruction.
- Construction and destruction sequences are checked. This is, strictly speaking, a correctness test, but the test of ctor and dtor vtables in effect checks this as well.
- Contents of VTTs, if any, are checked by looking for variables of expected mangled names and comparing their contents to those specified in the Specification.
- Names of all member functions are checked by making references to expected mangled names. Mangled names of virtual member functions get tested by the vtables also, but this step is needed to test the mangling of non-virtual member functions.
- Thunks are pointed to from vtables, hence their presence and name mangling are tested by tests of vtables. The adjustments made by the thunk are encoded in the mangled names of thunks, therefore thunk names depend on base-rules[9].
- Name mangling.
- Empty classes.
- Initialization guard variables, including those for multithreaded implementations.
- Contents and name mangling of RTTI variables.

[6] Ambiguous base classes, but it may still be possible to reach them unambiguously by specifying a typecast chain.

[7] Given `struct abcd{int p;}; struct efgh:abcd{int e;};struct xyz:efgh,abcd{int t;}*ptr;` the direct typecast ((abcd*)ptr) is invalid, so we cannot test it, but ((abcd*)(efgh*)ptr) is valid and we do test it.

[8] These can be different from the full object vtables as per C++ rules.

[9] There are situations where multiple thunks can have the same effect, and compilers do not always match in their choice. The test-suite includes a mechanism of Alternate-Thunk-Names which generates sets of equivalent thunks, and the tests are considered passing if the thunk generated by the compiler matches any member of the equivalence set of the expected thunk.

- Pure-virtual and deleted functions.
- Layout of pointer-to-members, both data and functions.
- Array cookies, new[] and delete[] operators.

The test-suite makes some simplifying assumptions: the two important ones being that structs and classes are equivalent; that member access does not change allocation.

Test-suite non-coverage

For the sake of completeness, here is a non-exhaustive list of items that this test-suite does NOT cover:

- Functional correctness of the C++ language.
- The calling of member functions, virtual or not. The test-suite only checks that the vtables point to correctly named functions.
- The allocation of variables on the stack or in data sections, beyond testing for size and alignment of classes as reported by sizeof() and __alignof__() operators.
- Exception handling behavior.
- The basic calling convention, because they are considered part of the base-rules. (These may be added in the future).

Tests for Name Mangling

Automatically generated tests do check name mangling by simply making references to mangled names. An error in name mangling will lead to link-time errors in these cases. More complicated tests for name mangling are written by hand and these use LLVM's FileCheck utility to check for the presence of properly mangled names in generated assembly files.

The Test harness

The test harness uses the 'lit' framework from the LLVM distribution. The first few lines of every test file contain lines that need to be executed for the test to be run. The test passes if every line ends in a pass status.

```
// RUN: cxx_compiler -c %s -I "common" cxx_rtti -o %t1.o
// RUN: c_compiler -c %s -I "common" -o %t2.o
// RUN: c_compiler -c -o %t3.o -I "common" "common/testsuite.c"
// RUN: linker -o %t.exe -I "common"  %t1.o %t2.o %t3.o
// RUN: runtool %t.exe | checker "TEST PASSED"
```

The keywords in the above lines are defined as:

- `cxx_compiler` is expanded to the C++ compiler under test.
- `c_compiler` expands to the corresponding C compiler.
- `linker` is the linker, or the compiler command needed to link them
- `common` is a directory containing the utility files `testsuite.c` and `testsuite.h`
- `%t[num]` expand to a unique string (a normal lit operation).
- `runtool` is the tool needed to run the linked program on the target. For native runs it should be left as the empty string.

- `checker` defaults to grep, but users can supply their own tool which can look at any failure messages to determine whether they are 'expected'. This is meant for extensibility to handle minor variations from the layout expected by the test-suite.

A top-level python file determines values of these symbols. We use the lit harness that comes with LLVM, with no modifications, so there is no need for the test-suite to include a copy of lit.

Configuration example

The test configuration is done in Python. An example top-level file looks like:

```
import os
import sys
test_params = {
    "c_compiler" : "clang" ,
    "cxx_compiler" : "clang -x c++" ,
    "Mode" : "LP64", # possible values are "ILP32" or "LP64"
    "bindump" : "nm", # tool to dump symbol names of object files
    "runtool" : "", # in case this is cross compiler
    "Platform" : "x64",
    "cxx_rtti" : "-frtti",   # option to enable rtti
    "cxx_exceptions" : "-fexceptions",   # option to enable exceptions
    "cxx_cpp11" : "" # The option to enable C++11 mode. No need with clang
}
builtin_parameters = {
    'build_mode' : "Release",
    'llvm_site_config' : os.path.join(os.getcwd(), 'lit.site.cfg'),
    'clang_site_config': os.path.join(os.getcwd(), 'lit.site.cfg'),
    'test_params' : test_params
    }
lit.main(builtin_parameters)
```

The test_params and builtin_parameters are python dictionaries. The compilers are specified as the elements "c_compiler" and "cxx_compiler". The linker can be specified separately, but if not, "c_compiler" is used as the default.

Test file example

A typical test looks something like:

```
// RUN: c_compiler -c -o %t1.o -I "common" "common/testsuite.c"
// RUN: cxx_compiler cxx_rtti -c %s -I "common" -o %t2.o
// RUN: c_compiler -c %s -I "common" -o %t3.o
// RUN: linker -o %t2.self  %t1.o %t2.o %t3.o
// RUN: runtool %t2.self | checker "TEST PASSED"
#include "testsuite.h"
#ifdef __cplusplus
struct  efgh  {  long r;};
static void Test_efgh(){
  init_simple_test("efgh");
  efgh lv;
  check2(sizeof(lv), ABISELECT(8,4), "sizeof(efgh)");
  check2(__alignof__(lv), ABISELECT(8,4), "__alignof__(efgh)");
  check_field_offset(lv, r, 0, "efgh.r");
}
static Arrange_To_Call_Me vefgh(Test_efgh, "efgh", ABISELECT(8,4));
#else // __cplusplus
... some C data structure desribing abcd
#endif // __cplusplus

#ifdef __cplusplus
struct  abcd  : efgh {
  int pa;
  virtual void  foo(); // _ZN4abcd3fooEv
   ~abcd(); // _ZN4abcdD1Ev
   abcd(); // _ZN4abcdC1Ev
};
void  abcd ::foo(){vfunc_called(this, "_ZN4abcd3fooEv");}
abcd ::~abcd(){ note_dtor("abcd", this);}
abcd ::abcd(){ note_ctor("abcd", this);}
static void Test_abcd()
{
  extern Class_Descriptor cd_abcd;
  void *lvp;
  ABISELECT(double,int) buf[4];
  init_test(&cd_abcd, buf);
  abcd *dp, &lv = *(dp=new (buf) abcd());
  lvp = (void*)&lv;
  check2(sizeof(lv), ABISELECT(24,12), "sizeof(abcd)");
  check2(__alignof__(lv), ABISELECT(8,4), "__alignof__(abcd)");
  check_base_class_offset(lv, (efgh*), ABISELECT(8,4), "abcd");
  check_field_offset(lv, pa, ABISELECT(16,8), "abcd.pa");
  test_class_info(&lv, &cd_abcd);
  dp->~abcd();
  Check_Ctor_Dtor_Calls(lvp);
}
static Arrange_To_Call_Me vabcd(Test_abcd, "abcd", ABISELECT(24,12));
#else // __cplusplus
.. some C data structure describing abcd, its vtables etc ...
#endif // __cplusplus
```