ABI Test-Suite

Frequently asked questions v1.0.0

By: Sunil Srivastava.

Aug 12, 2014

1. Which versions of clang/llvm does the ABI test-suite work with?

   At the time of writing this document, the test-suite has been validated using the svn/trunk revision #213127. This revision is just prior to the branching of v3.5. We will verify the test-suite with the official clang v3.5 when it is released. Not all of the tests will pass using clang v3.4 as there are bugs in it, where these bugs have been fixed for v3.5.

2. Do you expect this ABI test-suite to work with future versions of clang?

   Yes. If there are changes to the ABI in future versions of clang, as there have been in past, the test-suite will find them. After all, that is the purpose of the test-suite! In such cases the ABI needs to be returned to the previous state or the test-suite needs to be updated to reflect the change.

3. There are two configurations in the ABI test-suite: x86 32-bit and x86 64-bit. How can we add more?

   Unfortunately, there is no good solution for that. The test-suite depends on both the IA64 ABI spec and the base document that defines C level layout (base-rules). Hence it is very difficult to write a test-suite that can test the IA64 ABI for an arbitrary set of base-rules. We have chosen the two most commonly used base-rules.

4. How do I add new tests for one of the currently supported configurations?

   Some tests can be added along the lines of existing tests, but be aware that the object layout tests (the majority of the test-suite) have been automatically generated by an automatic test-generator. At present we cannot release the test generator to the open-source community. We realize that is a handicap and in the future we may re-implement this based on clang technology so it can be released to the community.

5. What does the automatic test-generator do?

   The test-generator takes C/C++ source and performs standard compiler object layout but instead of generating code it emits C/C++ code to test the layout against the layout it has just created. It does this once per configuration, changing the layout rules to match their different requirements. The multiple configurations are combined using the ABISELECT macro.

6. Will you run the automatic test-generator for requested configurations?

   No. We have provided the x86 32-bit and x86 64-bit configurations to help the community maintain ABI quality on these common platforms. We cannot support all possible configurations but we do not rule out adding additional configurations in the future.

7. What versions of g++ can I use with this test-suite?

   The g++ compiler has not been tested. Theoretically, it should just work, though there may be some minor differences between g++ and clang. Clang has done an exemplary job of following g++ in layout compatibility, but clang and g++ may have diverged.

8. Will the test-suite work with other IA64-ABI compilers?

   Only if the compiler exactly matches clang's ABI layout. Chances are, you will see some failures.

9. If I have my own compiler and this gives 23 failures due to an unavoidable incompatibility, how can I handle these?

   There are multiple ways you can handle such 'small' mismatches:

   - You can make note of the failing test and just ignore them in future runs.
   - Or, you can add the failing tests to the skip_list (see point 13 below), to mark them XFAIL. Note that marking a test as XFAIL is not the same as just ignoring it. If an XFAIL tests starts passing in the future, it will be flagged as a test failure.
   - Or, run the executable of the failing tests and store their output as a 'Golden Master' (see point 14 below) and then compare all future runs against that.
   - Or, you can write your own checker. Note that the last RUN line of most tests looks like:

     ```
     runtool %t2%exeext | checker "TEST PASSED"
     ```

     The 'checker' defaults to 'grep', but you can write your own filter that either stores its input as the 'Golden Master' or it compares the incoming result to the 'Golden Master'. Error messages have the name of the file and the line number so the checker can always figure out which file it is checking. We have not written such a clever checker, but we can see the possibilities.
   - Or, you can write a checker that goes and modifies the tests to match what the compiler is doing.
   - Or, modify or remove the tests in your local copy as you see fit.

10. Why is it necessary to run the tests on a target machine?

    The test-suite could have looked at the generated assembly/object code or even the clang IL to verify the layout, however, that would have required writing an assembly language parser, or an IL dump parser for multiple platforms. We did not want to go down that path. There are things in the object layout that are hard to test statically, such as the constructor and destructor vtables. While we can statically test VTTs and vtables, which part of them get used at which stage of construction or destruction requires runtime checks or some kind of code emulator.

11. Why do you not use static asserts?

    Static asserts are even weaker than reading the assembly code. Also, while static checks can find an error more quickly, one failed static check in a file will stop the run, thus hiding potentially more

errors. There is a NO_STATIC_CHECKS macro defined in testsuite.h. If you undefine it, static checks will be used wherever possible, but that is a small set.

12. Why have you not enclosed tests in unique ifdefs, so one can selectively remove them, as some other test-suites do?

    In some cases perhaps we should have done this but it becomes complicated with complex derivation graphs, where we cannot just remove a class from the middle of the derivation tree. Still, this may be included in future improvements.

13. Why are you using the suffixes .x and .xpp for the test files?

    We wanted to have a mechanism to mark certain tests XFAIL on a per file per configuration basis. This is something that lit does not provide (though lit can ignore whole directories via lit.local.cfg). So we came up with a skip_list mechanism. You can specify the skip_list at the top level python runner. When a run starts, lit.site.cfg copies every .x and .xpp file to a corresponding .c and .cpp file, adding XFAIL lines if that file appears in the skip_list. Then the tests are run on the .c or .cpp files. This process of 'copying' is expensive (over a minute), but it happens only once per new directory. In subsequent runs, files which are not in the skip_list are not copied again, thus making the subsequent overhead trivial. (In retrospect, we should have called it xfail_list instead of skip_list.)

14. How can I generate a 'Golden Master' result of test executables?

    Use the '–v' option on executables, or '-v –v' to print more details.