Documentation of the Test Framework for Osprey

May 23, 2007

1 Introduction

This framework is designed specifically to test the Open64 compiler Osprey. However, it can deal with any C/C++/Fortran compiler.

Following matters are managed by this framework:

1. Update the source code of the compiler by SVN;
2. Rebuild the compiler;
3. Run the compiler through the test cases, and catch any compiling and linking error;
4. Execute the output of the test cases, and catch any runtime error;
5. Record the execution time for some cases;
6. Generate the report, including the check-in log, compiler building log, and the detail for cases' testing;
7. Send the report to certain mail list.

The framework can work on two kinds of test cases:

1. Test case with a single source file;
2. Test case with multi source files.

To run this framework, `DejaGnu` and `exim4` is required. You should make sure that `exim4` is well configured to send remote mails. For more information, see `http://www.gnu.org/software/dejagnu/` and `http://www.exim.org`.

2 Global Configuration

The configuration files, `test.conf` and `mails.conf`, are located in `./conf`.

2.1 test.conf

`test.conf` is a configuration file to define some global variables. Each line of the file is a definition of one variable, in the format <*variable name*> = <*value*>. Table 1 is the specification of these variables.

Table 1: variables defined in test.conf

name	description	example	default
`CC`	The C compiler.	`opencc, gcc`	—
`CXX`	The C++ compiler.	`openCC, g++`	—
`FC`	The Fortran-90 compiler.	`openf90, gfortran`	—
`CFLAGS`	The compiling flags for C test cases.	`-O2, -O3`	—
`CXXFLAGS`	The compiling flags for C++ test cases.	`-O2, -O3`	—
`FFLAGS`	The compiling flags for Fortran test cases.	`-O2, -O3`	—
`SIM`	The simulator to run a testcase, which is used to test cross compilers. To test a native compiler , set it to null string.	`spim`	—
`UPDATE_COMPILER`	Whether update the compiler automatically.	true/false	false
`COMPILER`	The directory to save the source code of the compiler.	—	—
`BUILD`	The command line to build the compiler.	`make all -f Make.native`	—
`INSTALL`	The command line to install the compiler.	`./INSTALL.native`	—
`CURRENT_PLATFORM`	The current architecture name.	IA64	IA64
`PLATFORMS`	The available platforms for the testcases.	IA64,X8664	—
`VALIDATION_ONLY`	Whether only test for validation and not concerned about its performance.	true/false	false
`WHOLE_COPY`	Whether copy the cases to the output directory wholly.	true/false	false
`EVALUATOR`	The name of the evaluator to judge the runtime correctness and measure the cost of the cases.	default, validation	default
`MAIL_SENDER`	The sender field of the auto-generated email.	— —	— —
`SUBCONF`	The names of the subsidiary configures.	—	—

2.2 Subsidiary Configuration Files

The test cases can be tested under several different configurations. For example, you can test them using the flag -O2 and -O3 separately. Each configuration is specified by a `.conf` file under the directory `./conf`. The `SUBCONF` variable in `test.conf` is used to specify the list of the subsidiary configurations. Different configurations are separated by commas.

For example, if you specify `SUBCONF=O2, O3` in `test.conf`, the subsidiary configuration files `O2.conf` and `O3.conf` will be used.

The format of a subsidiary file is the same with `test.conf`. You can define the following variables in a subsidiary files: `NAME`, `CC`, `CXX`, `FC`, `CFLAGS`, `CXXFLAGS`, `FFLAGS` and `SIM`. `NAME` is a variable to specify a readable name of that subsidiary configuration, while the other variables have the same meaning with them in `test.conf` and can override them.

2.3 mails.conf

`mails.conf` is a file to keep the list of the E-mail addresses to which the test report will be sent. Each line of the file is a E-mail address.

3 the Directory Hierarchy of the Test Cases

All of the test cases are located under the directory `./cases`. The cases are divided into several groups, and each case must belong to one group. Each group is formed by a first level directory under `./cases`.

3.1 the Configuration File for Test Groups

Under the directory of a test group, you should write a configuration file `test.conf`. The format of this file is the same with `./conf/test.conf`, and you can define the same variables with it. The definitions of that file can override them in `test.conf`. However, the override for compiler variables(`CC`, `CXX`, `FC`, `CFLAGS`, `CXXFLAGS`, `FFLAGS` and `SIM`) has a lower priority than those in the subsidiary configuration files under `./conf`.

It seems that only the override for `VALIDATION_ONLY`, `WHOLE_COPY`, `EVALUATOR`, `PLATFORMS` and `SUBCONF` is useful.

3.2 the Testing Process for a Group

The test framework will search every file and directory under the group directory recursively. If a Makefile has been found under a directory, then that directory will be treated as a whole test case with multiple source files. Otherwise, for those source files which has no Makefile in the same directory and any ancestral directory within the group directory, it will be treated as as test case with single source file.

However, for some test cases, there's some individual source files which don't form a test case, e.g. the file `timing.cpp` in CERN loop. To deal with that case, `WHOLE_COPY` should be set to true. When `WHOLE_COPY=true`, the framework will ignore the individual source files and only test the directories with a Makefile, but the individual source files will also be copied into the output directory.

3.3 Building Process for Test Cases with Single Source File

The framework identifies the language of a source file by the extension. The extensions for each language are assumed as follow:

C:	`.c`
C++:	`.cpp .cxx .cc .C`
Fortran:	`.f .f90`

Instructions to control the testing behavior can be inserted in source file. Each instruction appears in the form of single-line comment, and starts from the beginning of a line. In a C/C++ test case, the following instructions are available:

- `//CMD:`<*command line*>

  This instruction indicates a command line to build the test case. There may be multi `CMD` instructions existing in one source file, and such commands will be executed in order. Variables may appear in the command lines, and the framework will replace the variables with its values while testing. The following variables are also available:

  1. `CC`: The C compiler, defined by `test.conf` or a subsidiary configuration file.
  2. `CXX`: The C++ compiler, defined by `test.conf` or a subsidiary configuration file.
  3. `FC`: The fortran compiler, defined by `test.conf` or a subsidiary configuration file.
  4. `CFLAGS`: The compiling flags for C test cases, defined by `test.conf` or a subsidiary configuration file.
  5. `CXXFLAGS`: The compiling flags for C++ test cases, defined by `test.conf` or a subsidiary configuration file.
  6. `FFLAGS`: The compiling flags for fortran test cases, defined by `test.conf` or a subsidiary configuration file.
  7. `SOURCE`: The name of the source file;
  8. `TARGET`: The name of the target file, which may be a .o file, .s file or executable file(the file type dependends on the presence of `//OBJ` and `//ASM` instruction).

  The variables are represented by the Makefile style, e.g. `$(CC)`, `$(SOURCE)`.

- `//FLAGS:`<*flags*>

  You can override the compiling flags `$(CFLAGS)`, `$(CXXFLAGS)` or `$(FFLAGS)` by that instruction.

- `//NOEXEC`

  This insruction tells the framework not to execute the test case after building, i.e. only take the building test but not the runtime test.

- `//OBJ`

  Tell the framework to generate `.o` file only. This instruction implies `//NOEXEC`.

- `//ASM`

  Tell the framework to generate `.s` file only. This instruction implies `//NOEXEC`.

- `//PLATFORM:`<*architecture name*>

  The available architectures for the current test case. If more than one architectures are available, separate them by comma. If the CURRENT_PLATFORM value is not among those architecture names, the test case will be skipped.

In a fortran test case, the usage of instructions are all the same with C/C++ test cases, except that they begins with '!' instead of '//'.

All of the instructions above are optional. To compile/link a test case, the framework will execute the commands indicated by `CMD` consequently. If no `CMD` command is present, the framework will generate a single command line automatically.

For example, to deal with a C source file, if both `//OBJ` and `//ASM` is absent, the command line will be

```
$(CC) $(CFLAGS) $(SOURCE) -o $(TARGET)
```

To deal with a C++ source file, while `//OBJ` is present, the command line will be

```
$(CXX) -c $(CXXFLAGS) $(SOURCE) -o $(TARGET)
```

3.4 Building Process for Test Cases with Multiple Files

There's some constraints for a test case with multiple files,

- Each test case must contain a `Makefile`;
- The first target in the `Makefile` must be the target file;
- The name of the target file must be the same with the name of that directory.

The framework will execute `make` on the `Makefile` to build the test case. The variables, which can be used in single file test cases(only except `$(SOURCE)`), are also available in `Makefile`. You can also override those variables by setting the values in the beginning of `Makefile`.

Like the single source test cases, you can specify some test instructions in a Makefile. An test instruction should be written in a separated line and start with '#'. There're two kinds of available test instructions for test cases with multiple source files, i.e. NOEXEC and PLATFORM. Their usage is the same with the single file test cases.

3.5 the Runtime Evaluation for a Test Case

The framework use a evaluator for the evaluation of a test case. A evaluator can tell the correctness, maybe also the run time of a test case. There're two general evaluators in the framework, named `default` and `validation`. They all judge the correctness in the same way, i.e. a test case run correctly if and only if it exits with zero and there's no standard output. The only difference of those two evaluators is that `default` evaluator not only tells the correctness, but also measures the run time cost, while `validation` only tells the correctness.

However, for some test cases, e.g. CERN loop, the correctness cannot be evaluated in that way. Therefore, we have defined a specified evaluator for it, named `cern`.

The evaluator used for each group is specified by the `EVALUATOR` variable.

4 Run the Framework

To run the framework, just execute `runtest` under the first level directory of the test framework. The contents in the directories `./output` and `./log` are generated by the framework.

The outputs files (object files, assembled files, executable files) for the test cases are located in `./output/yy-mm-dd` (`yy-mm-dd` is the current date). The log files are located in `./log/yy-mm-dd`.