PTF CFS Plugin - User's Guide
- version 1.5 -

Anca Berariu

February 7, 2014

Contents

1 Introduction **2**

2 Quick Start **3**
2.1 Quick installation 3
2.2 Basic configuration - `config.cfg` 3
2.3 Running CFS 4
2.4 Execution results 4

3 CFS Autotuning Approach **6**
3.1 Tuning parameter 6
3.2 Search strategy 6
3.3 Tuning scenario 6
3.4 Tuning action 7

4 Configuration **8**
4.1 `config.cfg` file 8
4.2 Application settings 8
4.3 Search strategies 9
4.3.1 Exhaustive search 9
4.3.2 Individual search 9
4.4 CFS tuning parameters 9
4.4.1 "ON/OFF" compiler flags 10
4.4.2 Flags with multiple values 11
4.4.3 Combining flags 11
4.4.4 Excluding flags 11
4.4.5 Compiler default configuration 11
4.5 Improved tuning time 13
4.5.1 Selective `make` 13
4.5.2 Instrumented applications 14

5 How To Use the Tuning Advice **16**

Chapter 1

Introduction

One of the main targets in performance optimization is the minimization of the execution time of an application. Besides the choice of the implemented algorithm and the way the program is written, another important factor is represented by the *the compiler*. The compiler generates the actual executed code, the *machine code*, from the high-level source code.

Nowadays, compilers apply a large number of program transformations to generate the best code for a given architecture. Such transformations are, for example: loop interchange, data prefetching, vectorization, or software pipelining. While the compiler ensures the correctness of the transformations, it is very difficult to predict the *performance impact* and also to select the right sequence of transformations. They rather provide a long list of compiler flags (and even directives) and expect the programmer to guide the compiler in the optimization phase by choosing the right flags and combinations.

Due to the large number of flags and the required background knowledge in the compiler transformations and their interaction with the application and the hardware, it is very difficult for the programmer to select the best flags and to guide the compiler by inserting directives. It is thus often the case, that only the standard flags O2 and O3 are used to change the approach of the compiler optimization.

The CFS Plugin automatically searches for the best combination of compiler flags to be used when building a particular application. The programmer only has to provide a list of flags which should be taken into consideration. Using the Periscope Framework, the execution time of the application compiled with different configurations are being measured and tracked. The combination with best execution time is then being displayed.

Chapter 2

Quick Start

2.1 Quick installation

CFS is being installed along with the Periscope Tuning Framework. Please refer to the *PTF Installation Guide* for a complete description of the installation process.

2.2 Basic configuration - `config.cfg`

In order to use CFS, a set of configuration instructions are required. These instructions are read at execution time from the `config.cfg` configuration file.

To start with, copy the default configuration file `config.cfg.default` into the folder containing the executable of your application and rename it to `config.cfg`.

```
$PSC_ROOT/templates/config.cfg.default →
        $APP_ROOT/.../config.cfg
```

For example, for the NPB benchmarks[1], copy the configuration file into the `bin` folder:

```
>cp $PSC_ROOT/templates/config.cfg.default NPB3.3-MZ/bin/config.cfg
```

Edit `config.cfg` to reflect the current context of your application. Here is an example for the NPB BT-MZ benchmark:

[1] See `http://www.nas.nasa.gov/publications/npb.html` for downloading and documentation.

```
// ********* application related settings **********
// the path to the Makefile
makefile_path="../";
// the variable containing the build flags
makefile_flags_var="FFLAGS";
// arguments for the make command
makefile_args="BT-MZ CLASS=W TARGET=BT-MZ";
// path to the source files of the application
application_src_path="../BT-MZ";
// ************************************************

// ********* plugin related settings ***************
// the desired search algorithm:  exhaustive or individual
search_algorithm="exhaustive";
// the compiler flags to be considered in the search
tp "Opt" = "-" ["O1", "O2", "O3"];
// ************************************************
```

2.3 Running CFS

CFS runs as a plugin within the Periscope Tuning Framework. It can be started using `psc_frontend` (see also *PTF User's Guide*) by setting the `tune` flag to `compilerflags`.

```
--tune=compilerflags
```

For the NPB BT-MZ example, one would call from within the folder containing the execution file:

```
psc_frontend --apprun="./bt-MZ.W" --uninstrumented --mpinumprocs=1
--tune=compilerflags --force-localhost --cfs-config="config.cfg"
```

This will start the measurements and the CFS tuning strategy for the uninstrumented version of the BT benchmark using one process.

2.4 Execution results

Upon successful completion of the tuning measurements, the CFS plugin displays at the standard output the list of all flags combinations (*scenarios*)

that were used in the search along with the corresponding execution times (*severity*). It also outputs the scenario with the best execution time.

For example, this is the output of the above call to `psc_frontend` for the BT-MZ benchmark:

```
 AutoTune Results:
-----------------------

Optimum Scenario:  2

Compiler Flags tested:
Scenario 0 flags:  " -O1 "
Scenario 1 flags:  " -O2 "
Scenario 2 flags:  " -O3 "


All Results:
Scenario | Severity
0 | 3.82434
1 | 3.81748
2 | 3.81678

------------------------
```

Chapter 3

CFS Autotuning Approach

CFS follows the general PTF plugin approach (see also *PTF User's Guide*).

3.1 Tuning parameter

Each entry in the flag list represents a *tuning parameter*. All tuning parameters define together the tuning space.

3.2 Search strategy

In order to find the best tuning of an application, a search through the tuning space has to be performed. For the CFS plugin, the *search strategy* can be selected by the plugin user. CFS provides two search strategies:

- exhaustive search and
- individual search

See section 4.3 for more details about the search algorithms.

3.3 Tuning scenario

Based on the chosen strategy, consecutive *tuning scenarios* are then being generated at run time and the performance of the application is being evaluated for each of these scenarios.

In the CFS plugin, one scenario represents one combination of compiler flags.

3.4 Tuning action

Applying one specific scenario to the application represents in the CFS case recompiling the application using the compiler flags corresponding to that particular scenario. Thus, the *tuning action* is the recompilation of the application.

Chapter 4

Configuration

4.1 `config.cfg` file

All configuration settings for the CFS plugin are read at execution time from the configuration file. The default name of the configuration file is

`config.cfg`

Another configuration file can be specified by setting the `cfs-config` parameter when calling the `psc_frontend`:

```
psc_frontend --cfs-config="<config_file_name>"
```

The configuration file is being searched in the folder from which the `psc_frontend` was started. Hence, if the name also includes a relative path to the file, it has to be relative to that folder.

4.2 Application settings

All path settings within the configuration file are relative to the path from which `psc_frontend` was started.

The following settings are mandatory for any application:

- the path to the application Makefile (where `make` should be issued)
  `makefile_path="<pathName>";`
- the variable used inside the Makefile to store compiler flags
  `makefile_flags_var="<varName>";`
- the path to the source files of the application
  `application_src_path="<pathName>";`

Additionally, one could use the `makefile_args` parameter for passing necessary arguments to the `make` process:

```
makefile_args="<listOfArguments>";
```

4.3 Search strategies

The search algorithm to be used by the CFS plugin can be set using the `search_algorithm` parameter:

```
search_algorithm="<algorithmName>";
```

The default search algorithm is the exhaustive search.

4.3.1 Exhaustive search

Exhaustive search generates all possible combinations of the given flags (the cross-product). This means that the size of the search space grows very fast (exponentially) with the number of flags.

To select exhaustive search, one should add to the configuration file:

```
search_algorithm="exhaustive";
```

4.3.2 Individual search

The individual search starts with the scenario containing only the first given flag and then iteratively adds the next flags, always keeping for the next step only the k best scenarios from the current step.

To select individual search, one should add to the configuration file:

```
search_algorithm="individual";
    individual_keep=<k>;
```

4.4 CFS tuning parameters

The tuning parameters for the CFS plugin are defined in the `config.cfg` file as follows:

```
tp␣"<paramName>"␣"<prefix>"␣[<valuesList>]
```

where

- `<prefix>` is a (not empty) string and

- `<valuesList>` specifies the list of values of the current parameter, either as a list of strings:

  `<valuesList> = "`*value1*`","`*value2*`", ...`
  with *value1, value2, ...* string values

  or as a integer range:

  `<valuesList> =` *valStart*`,[`*step*`,]`*valEnd*
  with *valStart*, *step* and *valEnd* integer values.

  If *step* is omitted, the default step value of 1 is being used.

The `prefix` is prepended to each of the values listed for a tuning parameter.

When building the scenarios, all given values of a tuning parameter are considered one at a time when combining them with the values of the other tuning parameters.

For example, having defined:

```
tp "TP1" "-O" ["0","2","3 -opt-prefetch"]
tp "TP2" " " ["-ip"," "]
```

will generate the following scenarios:

```
-O0 -ip
-O0
-O2 -ip
-O2
-O3 -opt-prefetch -ip
-O3 -opt-prefetch
```

4.4.1 "ON/OFF" compiler flags

The most simple tuning parameter for the CFS plugin is represented by one single compiler flag which can either be enabled or disabled. Such are, for example, the `ip`, `ipo`, or `opt-prefetch` flags.

An "ON/OFF" flag has two states which have to be given as two different values. For example:

```
tp "SingleFlag1" " " ["-ip"," "]
tp "SingleFlag2" " " ["-opt-prefetch"," "]
```

4.4.2 Flags with multiple values

Some compiler flags also accept the assignment of a particular value. Such are, for example, the `unroll` flag which accepts a value for the unroll transformation factor, or the optimization flag `O` which also accepts an optimization level.

These kinds of flags can be easily defined as tuning parameters with either a range of integer values, or a list of string values:

```
tp "ParameterFlag1" "-unroll=" [1,5]
tp "ParameterFlag2" "-O" ["0","2","3"]
```

4.4.3 Combining flags

There are cases where several compiler flags are known to give best results if considered together. In this case one would like to define such a "combined" flag.

This can be achieved by simply giving the two or more flags as one single value of a tuning parameter. For example:

```
tp "CombinedFlag" " " ["-ip -ipo"," "]
```

4.4.4 Excluding flags

For conflicting compiler flags, where it is known that they actually should exclude each other in any flags combination, one could set them as different values of the same tuning parameter. For example:

```
tp "ExcludingFlags" " " ["-O3","-no-prefetch"]
```

4.4.5 Compiler default configuration

The CFS plugin comes along with a series of standard configuration files for different compilers. These can be used by setting in **the first line** in the configuration file, the name of the compiler which is going to be used:

`compiler=<`*compilerName*`>`

For example, one could set:

```
compiler=icc
```

or

```
compiler=ifort
```

As of the current version, the following compilers are provided with a standard flags selection file:

Compilers:	`ifort`

The compiler configuration files are located at

`$PSC_ROOT/templates/cfs_`*compilerName*`.cfg`

and contain a list of predefined tuning parameters and configuration options.

For example, the `cfs_ifort.cfg` has the following content:

```
tp "TP_IFORT_OPT" = "-" ["O2", "O3", "O4"];
tp "TP_IFORT_XHOST" = " " ["-xhost", " "];
tp "TP_IFORT_UNROLL" = " " ["-unroll", " "];
tp "TP_IFORT_PREFETCH" = " " ["-opt-prefetch", " "];
tp "TP_IFORT_IP" = " " ["-ip -ipo", " "];

individual_keep=1;
search_algorithm="individual";
```

The settings defined in the compiler configuration file are loaded at runtime before those defined in the user configuration file. If the name of a tuning parameter defined in the compiler file is also encountered in the user configuration file, then a duplicate tuning parameter is being created.

All other settings besides the tuning parameters are being overwritten by the settings in the user configuration file.

For example, if the following `config.cfg` file is being used:

```
compiler=ifort;
makefile_path="../";
makefile_flags_var="FFLAGS";
makefile_args="BT-MZ CLASS=W TARGET=BT-MZ";
application_src_path="../BT-MZ";

search_algorithm="exhaustive";

tp "TP_IFORT_OPT" = "-" ["O2", "O3"];
```

then, first of all, the compiler configuration file `cfs_ifort.cfg` is going to be loaded, setting the search strategy to *individual search*. Afterwards the settings in the `config.cfg` are also being parsed, thus changing the search strategy from individual to *exhaustive search*.

The optimization levels, however, are not going to be overwritten. There will be two tuning parameters called `TP_IFORT_OPT`. As result, in this particular

case, scenarios like `-O2 -O2` and `-O3 -O2` will also be created (which, of course, is not a recommended practice).

4.5 Improved tuning time

There are two means to guide the CFS plugin to speedup the tuning process.

4.5.1 Selective `make`

As described in section 3, the CFS plugin performs as a tuning action the recompilation of the test application. This means that for each test scenario the entire application will be rebuilt. Even for relatively small source codes this might already require considerable time compared to the rest of the autotuning process.

The rebuild process can be directed to recompile only a restricted list of source files, i.e. the files which contain the code with a high percentage of the execution time.

This option can be activated by setting the `make_selective` flag to `true` and the `selective_file_list` to a particular list of files.

For the previous example, the NPB BT-MZ application, one would set:

```
                make_selective="true";
selective_file_list="x_solve.f y_solve.f z_solve.f";
```

The list of files which should be included in the build process can also be determined automatically by means of the `cfs_extract_files.sh` script, which comes along with the CFS plugin. The script makes use of the profile file generated by using the `-profile-functions` flag of the Intel compiler. Proceed as follows:

1. add the `-profile-functions` flag to your build command;
2. build the application using the Intel compiler;
3. run the application (as usually). This will generate in the current folder one `*.xml` file and one `*.dump` file.
4. run `cfs_extract_files.sh` giving the generated profile file as input, e.g.:

```
:$cfs_extract_files.sh < loop_prof_funcs_1391808148.dump
```

5. copy the list of files generated in `files2touch` to your `config.cfg` configuration file.

4.5.2 Instrumented applications

Another means to reduce the tuning time is to carry out perform measurements only on a (short) interval of the execution and not on the entire application. For example, if there is a main iterative loop, one could measure performance for only one iteration step instead of the entire execution of the loop.

Such a behaviour can be achieved by instrumenting the application with appropriate phase region definition. More precisely, for the case above, the entire body of the main loop would be defined as a phase region[1].

For example, the NPB BT-MZ application can be instrumented by adding the phase region declarations to the `bt.f` file:

```
c---------------------------------------------------------------------
c       start the benchmark time step loop
c---------------------------------------------------------------------

        do  step = 1, niter

! (lines omitted here ...)

!$MON user region
          call exch_qbc(u, qbc, nx, nxmax, ny, nz)

          do zone = 1, num_zones
            call adi(rho_i(start1(zone)), us(start1(zone)),
     $               vs(start1(zone)), ws(start1(zone)),
     $               qs(start1(zone)), square(start1(zone)),
     $               rhs(start5(zone)), forcing(start5(zone)),
     $               u(start5(zone)),
     $               nx(zone), nxmax(zone), ny(zone), nz(zone))
          end do
!$MON end user region

        end do
```

By default, CFS assumes that the application is instrumented. If no phase region is given in the application, then the main program is used.

[1]Please also refer to the *PTF User's Guide* for more details regarding application instrumentation.

In order to carry out the tuning process in the uninstrumented mode, one can pass to `psc_frontend` the flag

```
--uninstrumented
```

Please note that, in the uninstrumented mode, the execution time is measured as the wall clock time of the system command which executes the application. This means that reliable results can be achieved only if the execution time of the application is not too small.

Chapter 5

How To Use the Tuning Advice

Upon successful completion, the CFS plugin outputs a list of all tested scenarios as well as the id of the best scenario. This best scenario is the tuning advice of the plugin and it consists of the compiler flag combination which provided the best execution time of the test application.

One should copy the string indicating the best scenario (best combination) and add it to the Makefile as an option for the compiler.

Given the plugin output:

```
 AutoTune Results:
-----------------------

Optimum Scenario:  2

Compiler Flags tested:
Scenario 0 flags:  " -O1 "
Scenario 1 flags:  " -O2 "
Scenario 2 flags:  " -O3 "
```

one should add in the Makefile, for example:

```
gcc -O2 myFile.c
```