CFS Plugin User's Guide

PTF Version: 1.1
CFS Plugin Version: 2.0

Anca Berariu

13.04.2015

Contents

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 *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 the 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 - *cfs_config.cfg*

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

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

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

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

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

Edit `cfs_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="cfs_config.cfg"
```

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

Please note that the application *has to be built* and the executable file passed to the `apprun` flag *must exist* when calling `psc_frontend`.

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 the following search strategies:

- exhaustive search
- individual search
- random search
- GDE3 search
- Random search based on machine learning

See section 4.4 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 `cfs_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

```
cfs_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)
- the variable used inside the Makefile to store compiler flags
- the path to the source files of the application

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

```
makefile_path="<pathName>";
makefile_flags_var="<varName>";
application_src_path="<pathName>";
makefile_args="<listOfArguments>";
```

4.3 Remote `make`

Some HPC systems do not support compilation of applications on a compute node. To allow the CFS plugin to recompile the application, the make command has to be executed on a remote system. This is called *remote make* and is activated by setting in the configuration file `remote_make` to `true`. The make command is then executed on a remote machine via `ssh` with public key authentification. The following configuration options are supported:

`identity_path` - path to the private key
`remote_make_machine_name` - name of the remote system, e.g. on SuperMUC one of the login nodes such as login05.

4.4 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.

4.4.1 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";
```

4.4.2 Individual search

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>;
```

4.4.3 Random search

Random search selects k random points in the multi-dimensional search space.

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

```
search_algorithm="random";
sample_count=<k>;
```

4.4.4 GDE3 search

GDE3 (Generalized Differential Evolution 3) is a genetic search algorithm. The algorithm starts by randomly creating scenarios within the search space. These scenarios are called parents and constitute a population. In each generation, for every parent a child scenario is created by crossover between three other parent scenarios and mutation of the parent scenario.

After all children are created and evaluated, parent scenarios are compared to their respective children for dominance. A scenario dominates another scenario if it is better with respect to all objectives. Dominated scenarios are rejected, while non-dominated scenarios constitute parents for the next generation. The search goes on until the stopping criterion is met. The GDE3 search stops if any of the following conditions is met:

1. For three consecutive generations, same scenarios stay non-dominated at the end of generation, which means there is no improvement in solutions for three consecutive generations.

2. The limit for the maximum number of generations is reached.

3. The timer, if registered, expires. The timer is registered if the user wants to put an upper limit on the tuning execution time.

4. The number of attempts at generating not yet explored children has reached the upper limit. Currently, the number of attempts in each generation is set to 10000.

To select GDE3 search, one should add to the configuration file:

```
search_algorithm="GDE3";
gde3_population_size=<k>;
minutes_to_search=<k>;
```

If the population size is not given, GDE3 will use a default value. If the time to search is not specify, no limit is enforced.

4.4.5 Machine learning

If the search algorithm is random search, the probability for values of a compiler flag can be determined based on previous tunings with the CFS plugin. This approach is called machine learning and will be switched on by:

```
machine_learning="true";
```

The plugin will first run a pre-analysis to determine the signature of the current program. The signature is a vector of hardware counter measurements taken after compiling the program with optimization `O1`.

This signature is then passed to the random search algorithm which will look up similar programs and their tuning results in a tuning database. From those previous results it computes a probability distribution for the compiler flag values. This is then applied when a number of random samples is taken.

After the evaluation of the generated scenarios, the results are inserted into the database to increase the gathered knowledge.

4.5 CFS tuning parameters

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

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

where

- `<prefix>` is a non-empty string[1] 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,valEnd[,step]`
  with *valStart*, *step* and *valEnd* integer values.

[1] If you need an empty string here, please use instead a space inside the quotation marks: "␣".

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
```

4.5.1 "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"," "]
```

4.5.2 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"]
```

4.5.3 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"," "]
```

4.5.4 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"]
```

4.5.5 Compiler default configuration

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

```
compiler="<compilerName>"
```

Please note that currently only one compiler can be set per run[2].

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

Compilers:	`ifort`
	`icc`

The complete list of the compilers supported by your installed version of CFS can be retrieved by looking into the `$PSC_ROOT/templates` directory. The compiler configuration files are named

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

Please note that only those *compilerName* which are present in the *template* directory can be used as values for the `compiler` variable in the configuration file. Custom locations for compiler template files are not supported.

[2]See section 6.2 for more details on this issue.

Compiler templates 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 `cfs_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).

4.6 Improved tuning time

There are two means to guide the CFS plugin to speedup the tuning process: the selective build of source files and the instrumentation of the application.

4.6.1 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.

In order to avoid this overhead, the rebuild process can be directed to recompile only a restricted list of source files. These source files should be, in most cases, the files which contain the code with a high percentage of the execution time.

The *selective make* can be activated by setting in the configuration file `make_selective` to `true` and `selective_file_list` to the corresponding list of source 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 source files to be rebuilt can be determined automatically using one of the two methods:

1. the Intel compiler profiling, or
2. the Periscope profiling feature.

Intel compiler profiling - can be used only for serial and MPI applications and is based on the profiling measurements generated with the help of the `-profile-functions` flag of the Intel compiler.

To use this method 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.

Periscope profiling - supports also parallel applications and is based on the profiling feature of Periscope.

Unlike the previous method, this method is fully automatic. Nevertheless, it requires that the subroutines are instrumented in the test application. This can be done by setting either `all` or `sub` as instrumentation method for PTF. Please check the Section *Automatic Instrumentation* of the *PTF User's Guide* for further details on instrumentation.

The Periscope profiler is being called if `make_selective` option is set to `true` in the configuration file, but no list of files is given through the `selective_file_list` flag and there are also no `loop_prof_funcs_*` files generated.

Periscope executes a test run of the application and measures the execution time of all routines. It then selects the routines taking more than a given threshold[3] of the total execution time and registers the corresponding files for selective rebuild. This process is transparent to the user and it is integrated within the Periscope general tuning workflow as the pre-analysis step.

Please note that, apart from speeding up the tuning process, using the Periscope profiling also has the advantage that in the end it also delivers the best compiler combinations for the given files and execution times for the most time consuming routine in each of the files, as opposed to only providing the global best scenario for the entire application.

The decision which profiling method to apply is taken based on the following criteria[4]:

1. If `selective_file_list` is given in the configuration file, then this list of files is used.
2. If there is no list given in the configuration file, then the profile file `loop_prof_funcs_*` is searched and used.
3. If the `loop_prof_funcs_*` is not given, then the Periscope profiler is being used.

For the latter, the list of selected files can be checked by the user only in the final output or in the advice file.

[3]The threshold is internally set within Periscope and cannot be changed by the user. The common value is of 70%.

[4]Note that `make_selective` has to be set to `true`.

4.6.2 Instrumented applications

Another means to reduce the tuning time is to carry out performance 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 an appropriate phase region definition. More precisely, for the case above, the entire body of the main loop would be defined as a phase region[5].

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.

Besides the phase declarations, a minimal modification of the `Makefile` (or of the compilation and linking commands for the application) is required, in

[5]Please refer to the *PTF User's Guide* for more details regarding application instrumentation and running Periscope for an instrumented application.

order for Periscope to execute with the instrumented application. Basically, the `psc_inst` script call has to be added in front of the compiler call in the `Makefile`. Please refer to the *Quick Start* section of the *PTF User's Guide* for an example and details on how to work with instrumented applications.

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[6].

4.7 Output options

There are two options in the configuration file with respect to the output of the CFS plugin:

4.7.1 CSV output

By default, the CFS plugin will generate two types of output, as described in Section 5 of this guide:

1. console output in plain text and
2. standard Periscope advice in XML format.

There is a third option which can be activated in the configuration file by means of the `results_file` flag:

```
results_file="<filePath>"
```

This is a CSV (Comma Separated Values) formatted file and it contains the same information as the console output.

4.7.2 Routines measurements

The standard tuning output of the CFS plugin provides execution times and compiler flag combinations with respect to the entire application. There is a means to retrieve more fine granular information though, if the `routine` option is used in the configuration file:

[6]See also section 6.1 of this guide for more details.

```
routine=" <routine1>, <routine2>, ...  "
```

If given, the routines are measured separately and their execution time is output along with the best scenario for the corresponding file.

Please note that the same happens with the routines detected by Periscope profiling[7], if the profiling feature is activated. Nevertheless, the two sets of routines - those detected with the Periscope profiling feature, and these declared here with the help of the `routine` flag - do not interfere with each other.

Also note that, unlike the routines from the Periscope profiling list, all routines defined in the configuration file are going to be measured on the process with rank 0 (for parallel applications, of course).

As in the case of the Periscope profiling too, in order to be able to use the `routine` option in the configuration file, it is required that the instrumentation of the subroutines is turned on for the application currently being tested[8].

[7]See Section 4.6.1.

[8]Please check the Section *Automatic Instrumentation* of the *PTF User's Guide* for further details on instrumentation.

Chapter 5

How To Use the Tuning Advice

Upon successful completion, the CFS plugin outputs a list of all tested scenarios and their corresponding global times, as well as the id of the best scenario. The best global scenario refers to the entire application and it consists of the compiler flag combination which provided the best execution time of the test application.

If routines are also measured[1], then there will be more *best scenarios* printed, namely the best scenario for each file containing one measured routine. The best execution time for the routines are also being printed.

The output is delivered in two formats:

1. plain text at console output, and
2. XML format in the standard Periscope advice file.

The XML advice file also contains the routine execution times for all scenarios, besides the best execution time.

In order to use the results of the CFS plugin tuning, one should copy the string indicating the best scenario (best combination) and add it to the Makefile as an option for the compiler.

For example, given the CFS plugin console output:

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

Optimum Scenario:  2
```

[1]See Sections 4.6.1 and 4.7.2.

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

one should add in the Makefile:

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

Chapter 6

Limitations and Known Issues

There are a couple of limitations and known issues of which the users of the CFS plugin should be aware of:

6.1 Unreliable measurements

The CFS tuning strategy highly relies on the performance measurements of the underlying framework, namely the Periscope Tuning Framework (PTF), which is responsible to deliver the execution time measurements. As with most measurement tools, there is a limit in the accuracy level that can be achieved, meaning that any values lower than the provided accuracy cannot usually be registered.

The execution time which can be measured by PTF for uninstrumented applications has such a limitation as well. While the exact threshold also depends on the machine and the system on which the measurements are conducted, one should generally consider a lower limit for the execution times of approximately 2 s.

Please be aware that this refers to the region or phase actually measured by PTF. For example, when performing tuning on instrumented applications, it is not the total execution time, but the smallest region execution time that should be considered.

6.2 Multiple compilers

As of the current version of the CFS plugin it is not possible to tune one application with different compilers in one tuning run. For example, one might like to declare in one configuration file two or more compilers and the corresponding sets of flags and then use CFS to choose the best execution time of all. This is not possible, as currently only one single compiler can be defined in the configuration file (see section 4.5.5).

Nevertheless, the above can be achieved by starting CFS several times, once for each compiler, and then manually compare the execution times of the best scenarios of each run.