Periscope Tuning Framework

Michael Firbach
Technische Universität München

Outline

- Overview of the Periscope Tuning Framework
  - Features
  - Tuning plugins
- Hands-on: Importance analysis
- Hands-on: Using the CFS plugin

Estimated time: 45 min

Overview of the Periscope Tuning Framework

Overview of the Periscope Tuning Framework

PTF is a **framework** for **automated online** analysis and tuning.

- Distributed online tool
- Based on expert knowledge
- Currently being developed in Score-E (BMBF) and READEX (EU-FP7)
- Open source
- Homepage: http://periscope.in.tum.de/

New in version 2.0

- Uses Score-P measurement infrastructure
- Score-P has been extended with tuning functionality
- **Used in this course**

Overview of the Periscope Tuning Framework

PTF is a *framework* designed to be extended:

- It provides the infrastructure to instrument the application, run it, take measurements and apply optimizations
- The actual tuning is done by *tuning plugins*
  - Plugins address one specific optimization each (e.g. compiler flags, MPI settings, parallelism-capping, energy-tuning, ...)
  - The expert knowledge about specific optimizations is in the plugins, not in the framework
  - Capabilities of PTF is determined by the available plugins

Application requirements:

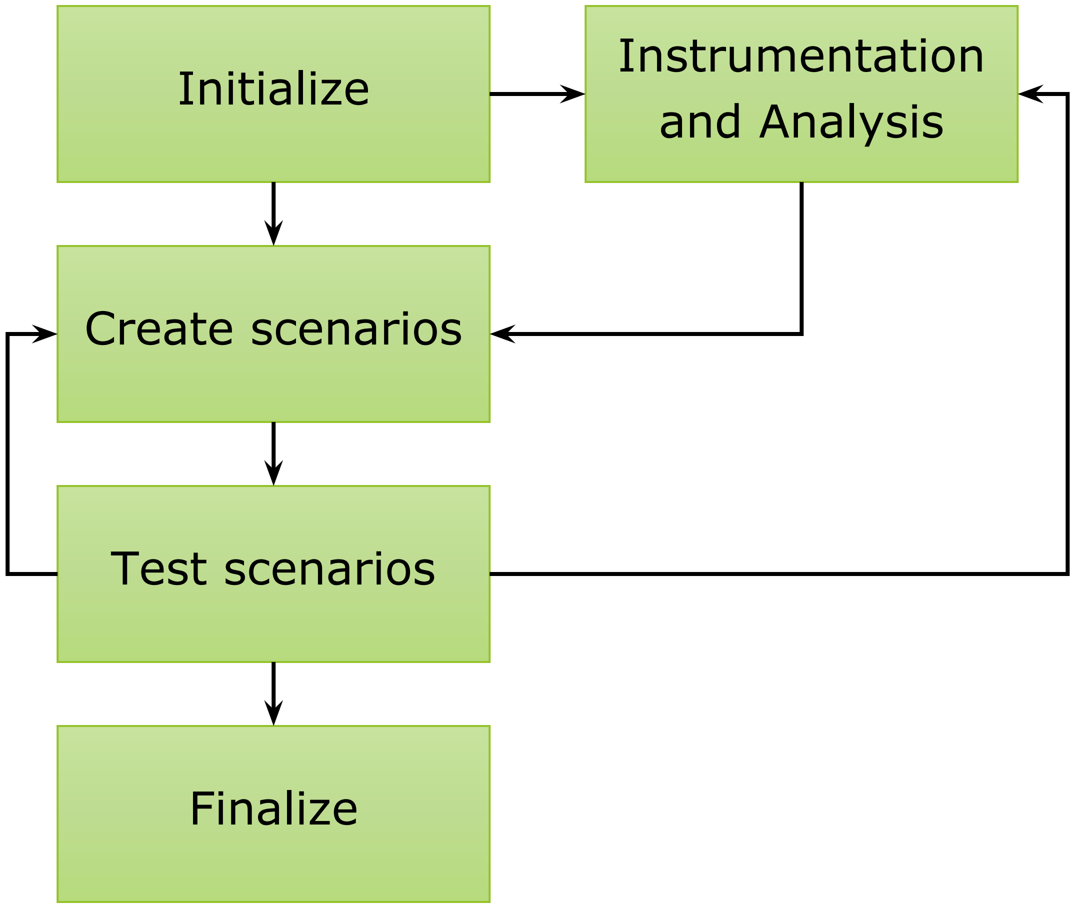
- SPMD
- Repetitive “main” loop (timesteps, refinement iterations, etc.)
- Many scientific codes qualify

Tuning plugins

How tuning plugins work

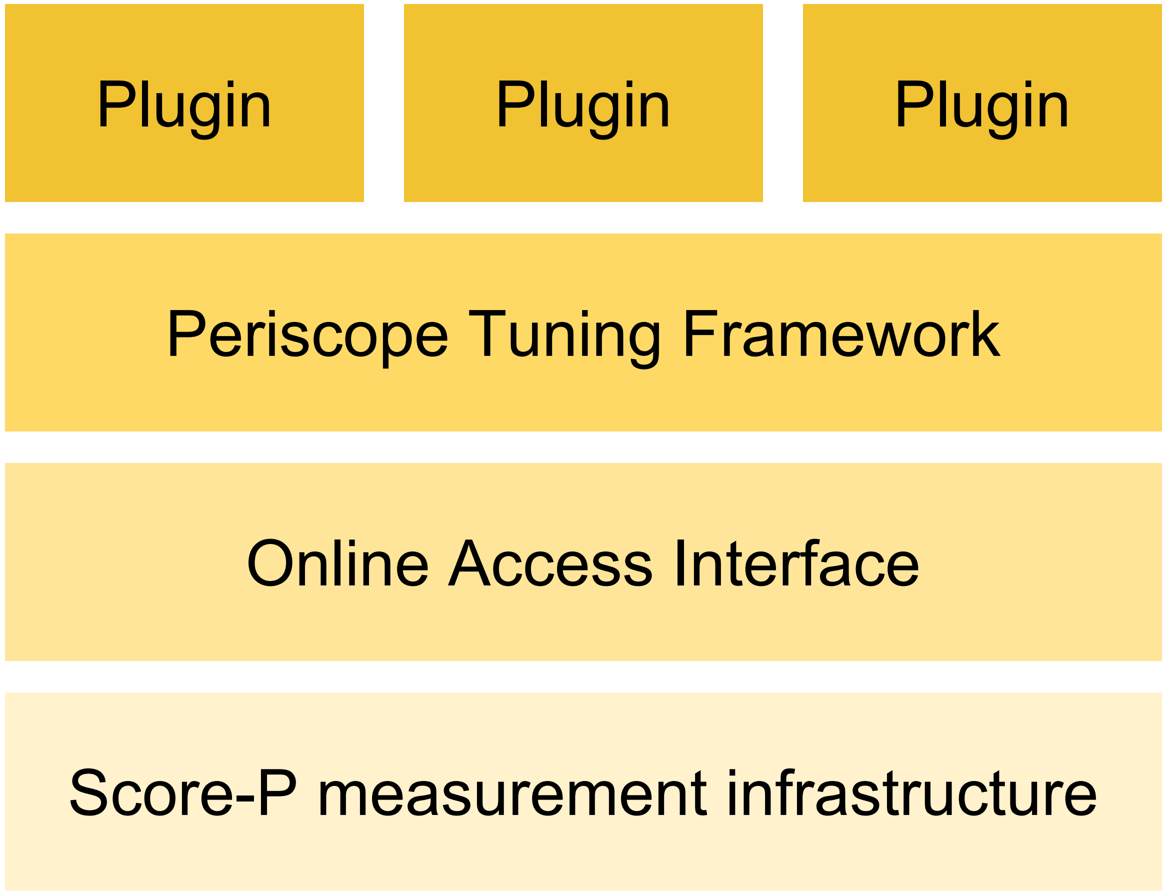
- All tuning plugins follow the lifecycle to the right
- During the lifecycle, *scenarios* will be created and executed
- For each scenario, plugins can:
  - request performance properties
  - apply tuning actions
  - re-compile or re-run the application

Please note: This is a very simplified picture!



Software stack

- Score-P gathers measurement data and applies tuning actions (one for each process)
- PTF spawns agents that connect to online access interface and evaluate properties from measurement data
- Central PTF frontend
  - Accumulation of properties
  - Runs the plugin to generate tuning decisions



Properties

- All analysis and tuning functions are based on *properties*
  - During the application run, Periscope tests various hypotheses about the performance
  - When a hypothesis is fulfilled by measurement data, a property is generated
  - Properties are generated for each relevant process and code region

- Hypothesis examples:
  - “This is an important code region for overall execution time”
  - “This region is not energy-efficient”
  - “OpenMP threads are imbalanced”
  - …

- The *severity* of the property indicates how strong the impact is on the overall performance

Examples of tuning plugins

- Compiler flag selection (CFS)
  - Determines optimal combination of compiler flags
  - Supports different compilers
  - Very portable

- Dynamic voltage and frequency scaling (DVFS)
  - Modifies CPU voltage & frequency to consume less energy
  - Weighted against increase in runtime
  - Available on selected systems only (root access / energy daemon required)

- MPI parameters
  - Optimizes MPI settings for given application
  - Some MPI implementations ignore settings

See http://periscope.in.tum.de/ for a full list of plugins.

Hands-on: Importance analysis

Finding important code regions

Hands-on: Importance analysis

In this exercise, you will:

- Perform the most basic automated performance analysis
- Use a Score-P **online access** region
  - Analysis and tuning is done on each entry of this region
  - Should be **repetitive**
  - Additional code in your application (Fortran, C and C++):

```
#include "SCOREP_User.inc"
SCOREP_USER_REGION_DEFINE( OA_Phase )

SCOREP_USER_OA_PHASE_BEGIN( OA_Phase, "foo", 0 )
// important repetitive code here
SCOREP_USER_OA_PHASE_END( OA_Phase )
```

Hands-on: Importance analysis

- BT-MZ has a **time step loop** which is suitable to be our "main loop"

- We have little time, so I have prepared an instrumented version of BT-MZ:

```
$ cp -r /home/hpc/a2c06/lu23veq/NPB3.3-MZ-MPI ~
$ cd ~/NPB3.3-MZ-MPI
```

- Open `bt.f` and identify the online access region (line 217):

```
$ vim BT-MZ/bt.f
```

Hands-on: Importance analysis

- Also note that I have modified BT-MZ's **`config/make.def`** to instrument with Score-P and online access:

```
F77 = scorep --online-access --user [...] mpif77 -fopenmp [...]
```

- Add to **`~/.bashrc`** (and log in again):

```
module load gcc/4.9
module load scorep/2.0_tuning
```

- Build the benchmark:

```
$ make bt-mz CLASS=C NPROCS=1
```

Hands-on: Importance analysis

- Run job with:

```
$ cd bin
$ sbatch jobscript_importance.slurm
```

- Check job status:

```
$ squeue --clusters=uv2 | grep $USER
```

- Check output:

```
$ cat out.txt
```

- Cancel job:

```
$ scancel --clusters=uv2 <job-id>
```

Hands-on: Importance analysis

- The result is **.psc** properties file
- Tool for tabular output:

```
$ module load ptf
$ psc_properties.py properties_Importance_XXXXXX.psc
```

```
=================================================================================================
NAME                  SEVERITY CONFIDENCE REGIONID
=================================================================================================
ExecTimeImportance         100          1 psc_file_name_none*foo*217
ExecTimeImportance       30.31          1 z_solve.f*!$omp parallel @z_solve.f:43*43
ExecTimeImportance     28.3702          1 y_solve.f*!$omp parallel @y_solve.f:43*43
ExecTimeImportance     27.6123          1 x_solve.f*!$omp parallel @x_solve.f:46*46
ExecTimeImportance     11.9437          1 rhs.f*!$omp parallel @rhs.f:28*28
ExecTimeImportance    0.438644          1 add.f*!$omp parallel @add.f:22*22
ExecTimeImportance    0.429296          1 add.f*!$omp do @add.f:22*23
[...]
```

Hands-on: Importance analysis

Other analysis strategies are available (besides Importance analysis):

- OpenMP load imbalances
- MPI load imbalances
- Energy inefficiencies
- …

- Still incomplete support in Periscope 2.0

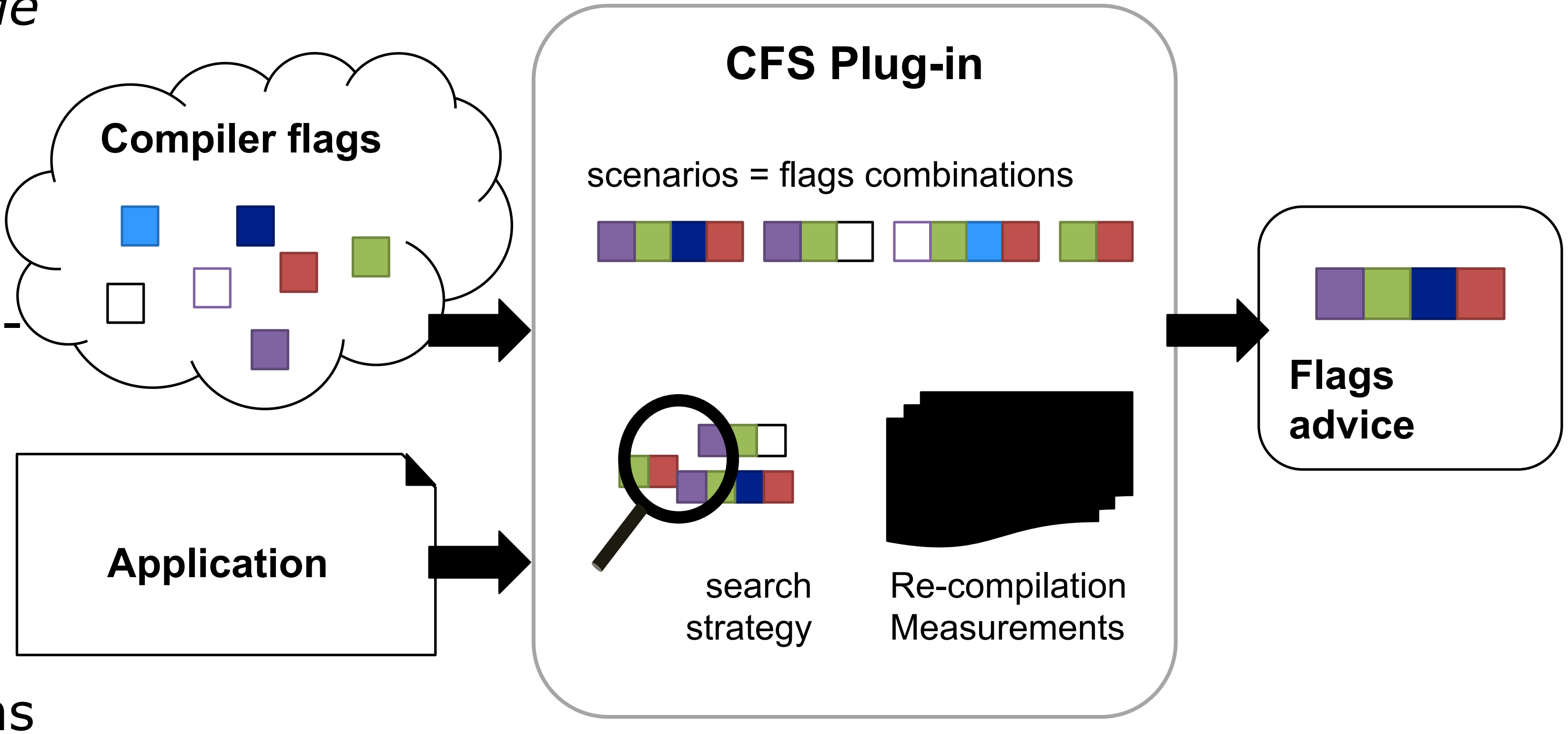
Hands-on: Using the CFS-plugin

Finding the optimal combination of compiler flags

Hands-on: Using the CFS-plugin

- Many compiler flags for *code generation*
- All possible combinations form a *search space*
- For every search step, the application is rebuilt and re-run
- Result of the search is optimal flag combination

Applicable to:
- Compute-bound applications
- Single-core optimization



Hands-on: Using the CFS plugin

Contents of the cfs_config.cfg:

```
makefile_path = "..";
makefile_flags_var = "FFLAGS";
makefile_args = "bt-mz CLASS=C NPROCS=1";
application_src_path = "../BT-MZ";
make_selective = "false";

search_algorithm = "exhaustive";

tp "OPT" = "-" ["O2", "O3", "O4"];
tp "HOST" = " " [" ", "-xhost"];
```

Build instructions

Search strategy

Flags to test (2×3 scenarios)

Hands-on: Using the CFS plugin

- Tuning process:
  - Compiler flags to be tested are inserted at `${FFLAGS}` in the **make.def** file
  - Application is recompiled and tested automatically

- Problem: We often cannot build on compute node
  - Possible on UV2, but very slow

- Solution: Use ssh to build on login node
  - Create ssh key-pair:

```
$ ssh-keygen -N "" -f ~/cfs_temp_key
$ cat ~/cfs_temp_key.pub >> ~/.ssh/authorized_keys
```

Hands-on: Using the CFS plugin

- Run job with:

```
$ cd bin
$ sbatch jobscript_cfs.slurm
```

- Check job status:

```
$ squeue --clusters=uv2 | grep $USER
```

- Check output:

```
$ cat out.txt
```

- Cancel job:

```
$ scancel --clusters=uv2 <job-id>
```

Hands-on: Using the CFS plugin

My results on UV2:

Scenario	Severity
0	5.43338
1	5.31741
2	5.46417
3	5.43542
4	5.4531
5	5.43023

- All scenarios are optimized builds, worst to best case: about 3% reduction
- Larger differences possible on other machines or with other applications (e.g. 16% during last workshop in Chile)

Hands-on: Using the CFS plugin

Advanced features for big searches (see User's Guide):

- Other search strategies, like individual search:
  - Creates scenarios with only one flag altered at a time
  - Might miss the optimal combination
  - Much faster (linear complexity)
- Selective make:
  - Periscope can determine relevant source files automatically and re-build only those
  - Or, user provides list of files
  - Selected files are touched, then the application is rebuilt
- Periscope can suggest flags to test for a specific compiler

Hands-on: Using the CFS plugin

What you can expect:

- Performance increase will be moderate in most cases (e.g. 5%)
- However, you don't invest a lot of time
  - Instrument application
  - Configure plugin
  - Plugin runs without user interaction
- Probably a good ratio of time spent and runtime improvement

Done!

Thank you for your attention.

You can tune your own applications later.