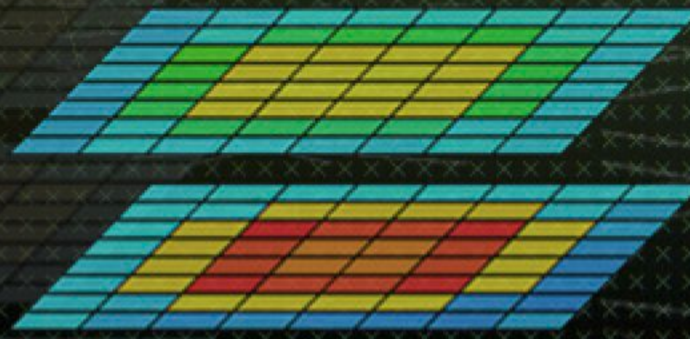


Periscope Tuning Framework

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

Overview of the Periscope Tuning Framework

Overview of the Periscope Tuning Framework

PTF is a framework for automated 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/

Version 1.1

- Current release, on download page
- Uses custom measurement infrastructure

Version 2.0

- Beta-version, future development
- Does not have all features of 1.1 yet
- Uses Score-P measurement infrastructure
- **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:

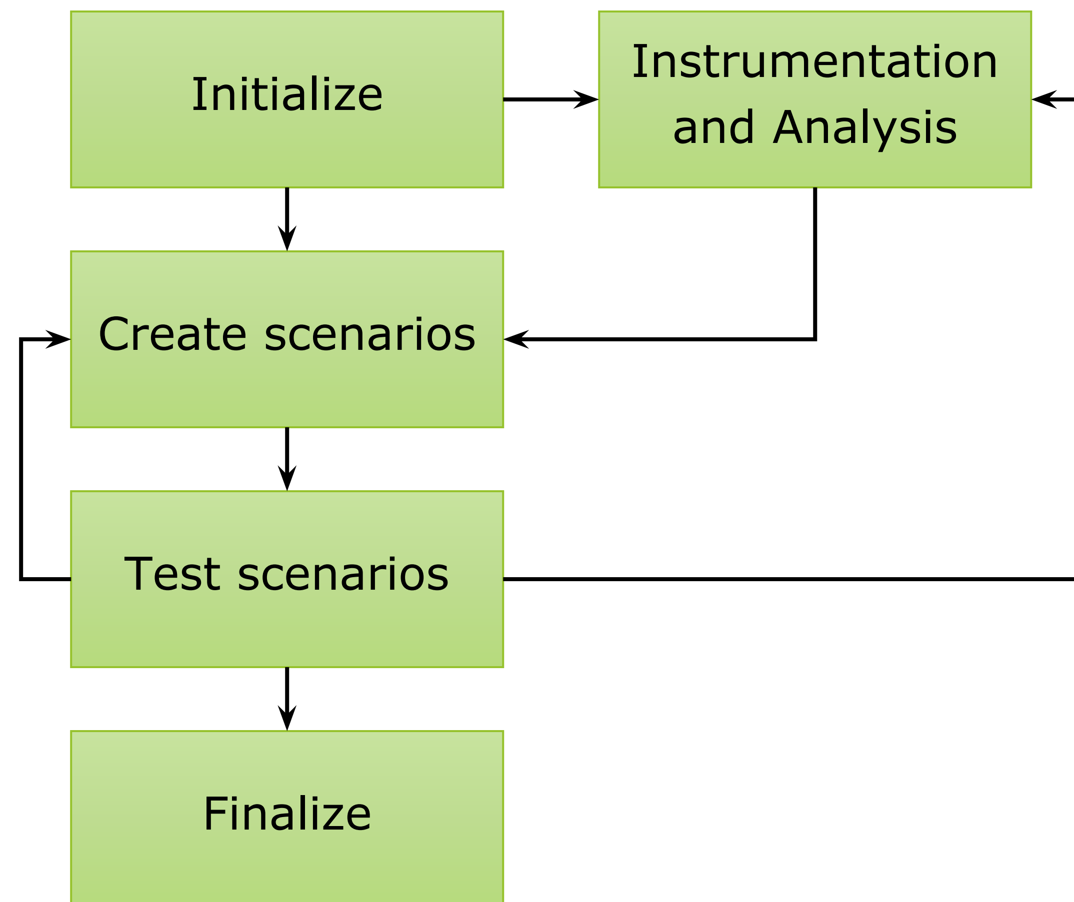
- SPMD
- Repetitive main loop (timesteps, refinement iterations, etc.)
- Many scientific codes qualify

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!



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

Software stack

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

Plugin	Plugin	Plugin
Periscope Tuning Framework		
Online Access Interface		
Score-P measurement infrastructure		

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 ony (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
- Define a Score-P **online access** region
  - Analysis and tuning is done on each entry of this region
  - Should be **repetitive**
  - Additions to your own 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 code here
SCOREP_USER_OA_PHASE_END( OA_Phase )
```

Hands-on: Importance analysis

- I have prepared an instrumented version of BT-MZ:

```
$ cp -r /home/courses/instructor06/NPB3.3-MZ-MPI_instrumented ~
$ cd ~/NPB3.3-MZ-MPI_instrumented
```

- Add to .bashrc:

```
module load scorep
module load periscope/2.0.0
```

- Copy the Periscope config file to your home:

```
$ cp /home/courses/instructor06/.periscope ~
```

Overview of the Periscope Tuning Framework

- Note that I have modified BT-MZ's **`config/make.def`** to instrument with online access:

```
F77 = scorep --online-access --user mpiifort -cpp
```

- Build the benchmark (smaller class now, since we are doing a lot of runs):

```
$ make bt-mz CLASS=A NPROCS=4
```

- Command line to run Periscope with Importance analysis:

```
psc_frontend --phase="foo" --apprun=./bt-mz.A.4 --mpinumprocs=4 --force-localhost --strategy=Importance --debug=2
```

- Run job script with:

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

Hands-on: Importance analysis

- Results:

```
$ cat out.txt
------------------------------------------------------------------------------
Procs          Region       Location      Severity      Description
------------------------------------------------------------------------------
P   0;            foo;              27;   psc_file_name_none:213;  100.000;       ExecTimeImportance
P   3;            foo;              27;   psc_file_name_none:213;  100.000;       ExecTimeImportance
[...]
P   0;          adi_;               7;        adi.f:0;     99.895;       ExecTimeImportance
P   3;          adi_;               7;        adi.f:0;     97.640;       ExecTimeImportance
[...]
P   2;      z_solve_;               7;    z_solve.f:0;     32.436;       ExecTimeImportance
P   1;      z_solve_;               7;    z_solve.f:0;     32.358;       ExecTimeImportance
[...]
```

- Note: Properties are generated for each process
  - Written to XML file for further analysis
- Line numbers not yet working with compiler-instrumented Fortran codes :-(

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

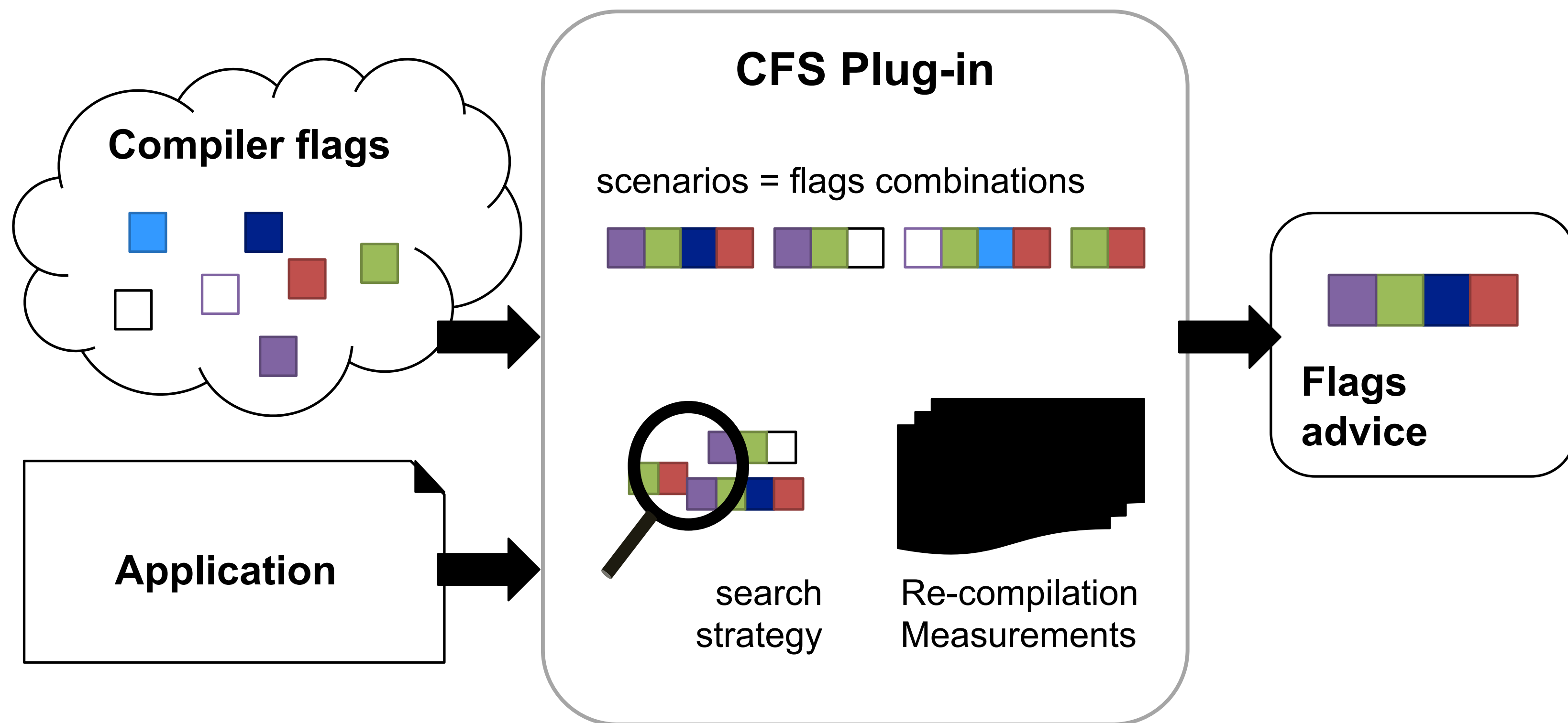
Hands-on: Using the CFS-plugin

Finding the optimal combination of compiler flags

Hands-on: Importance analysis

- 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



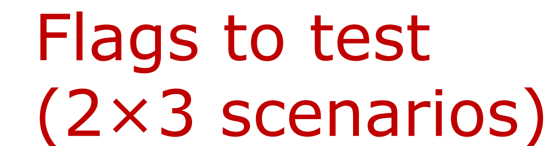
Hands-on: Using the CFS plugin

Contents of the cfs_config.cfg:

```
makefile_path = "..";
makefile_flags_var = "CFS_FLAGS";
makefile_args = "bt-mz CLASS=A NPROCS=4";
application_src_path = "../BT-MZ";
make_selective = "false";

search_algorithm = "exhaustive";

tp "OPT" = "-O" ["1", "2", "3"];
tp "FAST" = " " [" ", "-xHOST"];
```



Hands-on: Using the CFS plugin

- Modify BT-MZ's **config/make.def** to add a place for the compiler flags:

```
F77 = scorep --online-access --nocompiler --user mpiifort -cpp
[...]
FFLAGS  = ${CFS_FLAGS}
```

- Compiler flags to be tested are inserted at `${CFS_FLAGS}`
- `--nocompiler` reduces overhead (only our custom region is instrumented)

- Run job script with:

```
$ cd bin
$ sbatch jobscript_cfs.slurm
$ tail -F out.txt
```

Hands-on: Using the CFS plugin

My results on Leftraru:

Scenario	Severity
0	4.96712
1	4.88561
2	4.43144
3	4.25404
4	4.49323
5	4.16391

- Worst to best case: about 16% reduction
- Which flag has had more impact?

Hands-on: Using the CFS plugin

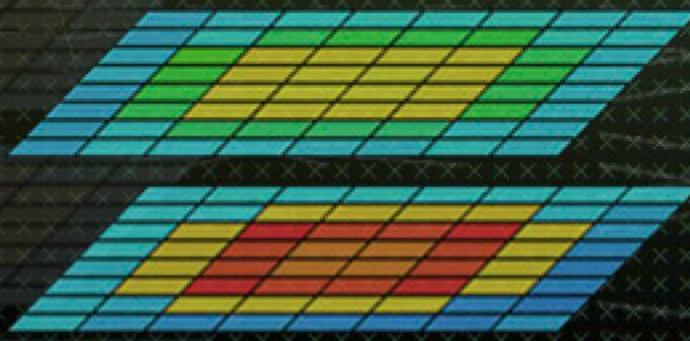
Advanced features for big searches (see User's Guide):

- 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 re-built
- 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 (maybe 5% to 10%)
- 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 now tune your own applications.