Automatic Compiler Flag Selection with Periscope

Michael Firbach
firbach@in.tum.de

February 13th, 2014

Outline

- Scope
- Motivation
- Basic workflow
  - Can we make this faster?
- Limitations
- Hands-on exercise
- Bonus track: Workflow automation with Pathway

Scope

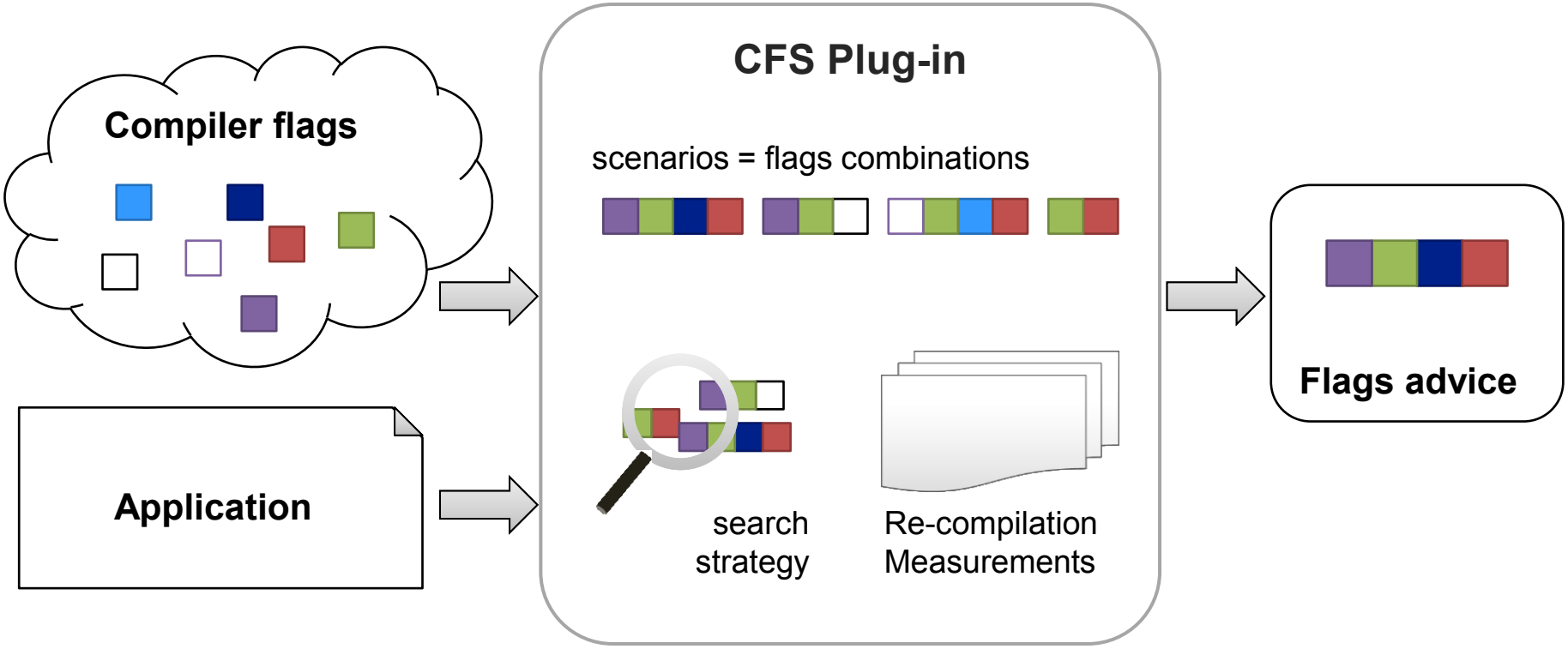
- Flags control the behavior of the compiler
  - Language version, Warning levels, Code generation
- Compiler Flag Selection is a process that systematically determines the best configuration
  - Combinations of flags → multi-dimensional search space
  - Optimization problem: Find best combination
- Applications that will benefit most:
  - Compute-bound applications
  - Single-core optimization

Motivation

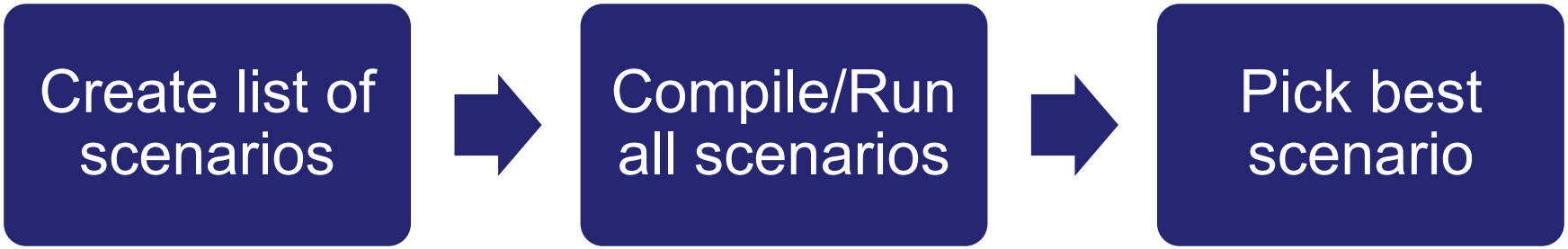
- Optimization potential left unused
  - Most flags are optional, programmers don't bother much
  - The effects of many flags is hard to predict
    - Specific to compiler, micro-architecture and application
    - Need a lot (often non-public) knowledge
  - Testing all possible combinations of flags is very cumbersome
  - Must be be re-done for a different HPC system or application

- A tool can help automate this process
  - Can automatically evaluate flag combinations
    - Re-compile, re-run, log execution time
  - Tool already knows important flags for specific compilers

Basic workflow



Basic workflow



- Basic search strategy: Exhaustive search
  - Create a scenario for every possible combination of flags
  - Guaranteed to find best combination
  - Can take a long time (exponential complexity)

- How can we speed up our workflow?

Basic workflow – can we make this faster?

- Faster search strategy: Individual search
  - Only creates scenarios for one flag at a time
    - Continues with best setting for this flag
    - E.g. test **–O[n]** first, then **–xhost** with best **O** setting
  - Might miss the optimal combination
  - The order of flags could be important!
  - Much faster (linear complexity)

- Intermediate solution
  - Keep the **k** best scenarios while going through the list of flags
  - See User's Guide

Basic workflow – can we make this faster?

- Faster build times: Selective make
  - Time spent on re-building the application over and over can be significant
  - We don't need to re-compile the whole application for each scenario
  - User provides list of files to touch
  - With the Intel Fortran compiler, we can create list automatically
    - A script does a profile run and creates list

Limitations

- It will be always too time-consuming for real-world applications
  - Must use smaller, hopefully representative data set
    - If too small, Initialization overhead dominates the run-time
  - Test the combinations of most promising flags only
  - Individual search might miss the optimal combination

- Restrictions for mixed-language applications
  - Flags should apply all compilers (e.g. Fortran, C++)

Hands-on exercises… get ready

Hands-on Exercise (1)

Load required modules

- `module use ~nct00001/gpfs_projects/UNITE/tutorial/mf`
- `module load UNITE`
- `module load periscope`

Actually only required in job script, but nice for testing.

Hands-on Exercise (2)

Copy the benchmark and periscope config file

- `cd ~`
- `cp -r ~nct00001/gpfs_projects/thursday_material/cfs .`
- `mv cfs/.periscope ~`
- `cd cfs`
- `unzip CFS_Demo.zip`

Hands-on Exercise (3)

Check your home:

- `ls -a ~`

```
.   ..   cfs   .periscope
```

Periscope config

Check your bin:

- `cd NPB3.3-MZ-MPI/bin`
- `ls`

```
config.cfg  job.lsf
```

Plug-in config

Hands-on Exercise (4)

Command line for compiler flag selection:

- **`psc_frontend --apprun=./bt-mz.W.4 --uninstrumented --mpinumprocs=4 --tune=compilerflags`**

- Hint: No need to make first

Use the job script and follow the output file:

- **`bsub < job.lsf`**
  `tail -F cfs.out`

In the meantime, let's see how CFS is configured…

- `vim config.cfg`

```
makefile_path="../";
makefile_flags_var="FFLAGS";
makefile_args="BT-MZ CLASS=W";
application_src_path="../BT-MZ";
make_selective="false";

search_algorithm="exhaustive";

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

Build instructions

Search strategy

Flags

(In the afternoon, you can add your own flags here)

See cfs_results.txt

```
➢ cat cfs_results.txt
  Optimum Scenario: 3

  Compiler Flags tested:
  Scenario 0 flags: " -O2  -xhost "
  Scenario 1 flags: " -O2  -  "
  Scenario 2 flags: " -O3  -xhost "
  Scenario 3 flags: " -O3  -  "
  Scenario 4 flags: " -O4  -xhost "
  Scenario 5 flags: " -O4  -  "
  […]
```

Tip: Runtime varies more widely with bigger problem classes

Hands-on Exercise (7)

- Now try with bigger problem class & individual search
- CFS config
  - Modify build instructions (e.g. class A)
  - Configure **"individual"** search
- Job script
  - Change executable name

- Run & verify which combinations (scenarios) have been left out

Hands-on Exercise (8)

- Now try selective make
- Only re-compile perforance-relevant files
- Useful for applications with long build times

- Config file:
- `make_selective="true";`
  `selective_file_list="x_solve.f y_solve.f z_solve.f";`
- Check output to verify which files are rebuilt
- How much time is saved by that?

Bonus track: Pathway

Bonus track

- We develop a tool that automates performance engineering workflows
  - ... like the workflow you just performed
  - Comes with graphical workflow editor
  - Makes new performance tools more accessible

- Live demo of Pathway