Periscope User's Guide

PTF Version: 1.1

Periscope Version: xx.xx

Michael Gerndt, Anca Berariu

13.04.2015

Contents

Chapter 1

Introduction

Periscope is a scalable automatic performance analysis tool currently under development at Technische Universität München and is part of the Periscope Tuning Framework (PTF), along with tools like Pathway and tuning plugins.

Periscope provides two main functionalities for Fortran and C/C++ applications: *performance analysis* and *performance tuning*.

Performance analysis is performed at runtime, using an iterative approach. There is a starting set of performance properties, which is then refined based on the measurements and the chosen search strategy. In the end, the appropriate set of performance properties is provided for the application being analysed. The search threshold, the confidence value, and the severity are defined by means of a formal specification of the properties.

Based on expert knowledge, Periscope uses several strategies to identify possible performance issues. Such strategies include exploiting parallel MPI or OpenMP regions, as well as system specific approaches, like for example for Power6 machines.

The second functionality, performance tuning, is provided through the tuning framework. Periscope offers the necessary support for measurements and search logic for a series of tuning plugins. Different application and environment setups are tested within the plugins and the best configuration is provided as an advice at the end of the tuning.

Periscope consists of four main components: the *frontend*, the *hierarchy of communication and analysis agents*, the *monitoring library* and the *GUI*.

- *The frontend* is responsible for starting both the application to be analysed, as well as all the internal components of Periscope. All settings regarding the execution of Periscope can be selected by means of command-line parameters of the frontend process.

- The *agent hierarchy* is transparent for the common users. At the bottom layer of the hierarchy there are the analysis agents. They control and configure the measurements for each application node/process. They can start, halt, or resume the execution, and they also retrieve the performance data. The strategy is communicated upon startup by the frontend and at the end of the local search, the performance properties are communicated back to the frontend via the agent hierarchy.
- The *monitoring library* is also transparent to the user and it provides the measurement and communication layer between the application being tested and the performance tool.
- The *GUI* is used to visualise and explore the performance results. It is an Eclipse plugin which can be easily used to identify the most sever performance properties, as well as the corresponding source lines responsible for the performance issues.

Periscope Tuning Framework

Alongside Periscope, the Periscope Tuning Framework (PTF) also provides PAThWay, a workflow management tool for HPC experiments, as well as a series of tuning plugins for automatic tuning of applications.

Chapter 2

Quick Start

2.1 Installation

Periscope can be installed from the source files, following the common process of configuring and building using Autotools.

Please check the *Periscope Installation Manual* for a thorough guide on how to install Periscope on your machine. The basic installation steps are:

1. check and install prerequisites: ACE, Boost, etc.
2. checkout the source files from the Periscope repository

   ```
   $git clone https://periscope.in.tum.de/git/Periscope.git
   periscope
   ```

3. configure your installation choosing appropriate options[1], for example:

   ```
   $ configure --prefix=$HOME/install/psc
   ```

4. build the files

   ```
   $ make -j 16
   ```

5. install the files

   ```
   $ make install
   ```

If you are using SuperMUC, Periscope is already installed on the system. In order to use it, you have to add to your `.bashrc` file:

```
$ module load periscope
```

[1]Please refer to the *PTF Installation Manual* for further details regarding available options.

and then issue in your home directory:

```
$ source .bashrc
```

Note: Please make sure to add the command for loading the periscope module into your .bashrc. Just issuing the command at the command line is not going to work properly.

2.1.1 .periscope configuration file

Before using Periscope, the `.periscope` setup file has to be created in your home directory. You may create a new one, or copy it from the Periscope installation directory:

```
$ cp $PSC_ROOT/templates/.periscope ~
```

The setup file contains a list of `<option>=<value>` pairs, as follows:

```
MACHINE = SuperMUC
SITE = LRZ
REGSERVICE_HOST = localhost
REGSERVICE_PORT = 50001
REGSERVICE_HOST_INIT = localhost
REGSERVICE_PORT_INIT = 50001
APPL_BASEPORT = 51000
AGENT_BASEPORT = 50002
```

If running on your local machine only, then the `MACHINE` option above should be set to `localhost`.

```
MACHINE = localhost
```

Please refer to the *PTF Periscope Installation Manual* for a detailed description on how to choose the proper option values for your particular system.

2.1.2 SSH access

In order to run Periscope, a private key based `ssh` access has to be provided on the machine running the tool. If not already configured, you can do so in few steps:

1. `$ mkdir ~/.ssh`
2. `$ cd ~/.ssh`
3. `$ ssh-keygen -t rsa -N '' -f id_rsa`
4. `$ cat id_rsa.pub >> authorized_keys`

5. `$ chmod 600 authorized_keys`

The `ssh` access is not required if running on your local machine, i.e. the `MACHINE` option is set to `localhost` in your `.periscope` file.

2.1.3 GUI

The Periscope GUI used for analysing the performance measurements is provided as an Eclipse plugin. You can install the GUI from this location

`http://www.lrr.in.tum.de/~petkovve/psc/eclipse`

following the common plugin installation process in Eclipse[2].

2.2 Basic analysis run

Having Periscope properly installed, there are only few steps required for a basic analysis of a test application:

1. specify a *phase region* by instrumenting the source code of the application;
2. modify the `Makefile` to enable instrumentation;
3. build the application;
4. start the analysis;
5. visualize and explore the performance results.

For the remainder of this section we consider as the test application the NPB-MZ BT benchmark[3].

2.2.1 Specify the phase region in NPB-MZ BT

Periscope uses an *iterative analysis* approach. It starts first with a set of performance properties which are measured for the test application throughout an *experiment* run. Based on the measurements result, it then determines new candidate properties which are going to be evaluated in the next experiment. The iteration stops when there are no new candidate properties.

[2]Please refer to the *PTF Periscope Installation Manual* for a step-by-step description of the installation process.

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

If the test application has a repetitive region, like for example the body of a main loop, then the consecutive experiments could be performed without the need of restarting the entire application. In order to do so, the repetitive region has to be marked in the source code as *a phase region*.

For the BT application, the phase region can be defined in file `bt.f`, lines 188 to 198, by inserting `!$MON user region` and `!$MON end user region` as shown below:

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

    do step = 1, niter
c------- 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
```

2.2.2 Modify the `Makefile`

In order to enable performance measurements, the test application has to be instrumented by the performance tool. To enable instrumentation, one has to substitute the compile/link commands usually defined in the `Makefile`.

For NPB-MZ BT, one should edit the `config/make.def` file and update the F77 variable as follows:

```
#---------------------------------------------------------------------
# This is the fortran compiler used for fortran programs
#---------------------------------------------------------------------
F77=psc_instrument -i -v -d -s ../bin/bt-mz.$(CLASS).$(NPROCS).sir
-t user,mpi mpif90
```

```
# This links fortran programs; usually the same as $(F77)
FLINK=$(F77)
```

2.2.3 Build the application

After the phase region was defined and the build command was adjusted, one can continue with the common build process of the test application.

For the NPB-MZ BT example, one should go to the root directory of the NPB-MZ series and issue:

```
$ make clean
$ make bt-mz CLASS=C NPROCS=16
```

2.2.4 Start Periscope analysis

Periscope can be started via its frontend `psc_frontend`. Upon calling the executable with proper parameters, both Periscope's internal components as well as the test application are being started and the performance measurements are then carried out.

For the NPB-MZ BT example, one should go to the `bin` directory and then call `psc_frontend` as follows:

```
$ psc_frontend --apprun=./bt-mz.C.16 --mpinumprocs=16
--strategy=MPI --force-localhost
```

2.2.5 Explore the results

Upon successful termination, Periscope generates a `*.psc` results file. This is a standard XML file and could be opened using any text editor. Periscope provides a Graphical User Interface (GUI) with enhanced visualisation and exploration functionalities for working with these performance result files.

Having started Periscope like described above for the NPB-MZ BT benchmark, the `properties_*.psc` should have been created into the same `bin` directory. Please follow the instructions in section 3.6 for opening this file within the GUI.

Chapter 3

Analysis Flow within Periscope

Periscope follows an **iterative analysis** approach: it determines performance properties based on measurements, decides on possible new candidate properties and then it performs again new experiments to measure the data required to check whether the candidate properties hold. See also the cycle depicted in Figure 3.1.

Figure 3.1: Periscope iterative analysis.

The number of experiments carried out in one run of Periscope depends on both the execution time of the application itself and also the performance issues it might exhibit.

The number of experiments carried out in one run of Periscope depends on the performance issues it might detect. Thus the total execution time of one Periscope analysis will depend on both the the execution time of the application itself, as well as the amount and severity of detected performance issues.

3.1 Specification of a phase region

The performance measurements carried out within one **experiment** of the iterative analysis could be applied to either the entire application or only a particular execution *phase* or code *region*. Periscope offers the possibility to define such a **phase region** by means of manual instrumentation of the source code.

Section 3.4 describes manual instrumentation in more detail. We only mention here that a phase region can be in terms of Periscope code instrumentation any regular user region.

A user region can be defined by inserting the following directives into the source code:

Fortran:
```
!$MON USER REGION
S1
S2
...
!$MON END USER REGION
```

C/C++:
```
#pragma start_user_region
S1
S2
...
#pragma end_user_region
```

Periscope allows the specification of several *user regions*, but only one such region can be defined as the **phase region**. This is done by passing the `--phase` option to the `psc_frontend` process at startup:

```
$ psc_frontend --phase=fileid:rfl
```

where:

- `fileid` is the *id* of the file containing the phase region. It is the same id used in the `psc_inst_config` file. See also Section 3.3.1.
- `rfl` is the ***r**egion **f**irst **l**ine number*. It represents the line number in the source file specified above, at which the region starts.

If several user regions are defined, but none of them is specified as the phase region, then the behaviour of Periscope is undefined.

If only one user region is specified, then this is automatically defined as the phase region.

If no phase region is specified, Periscope will automatically **restart the application** to perform new experiments, until no new candidate properties are found and the search terminates.

The use of phase regions is strongly recommended:

- it reduces the overall execution time of the Periscope performance measurements;
- it delivers more accurate results, as measurements are only performed for relevant execution fragments.

The best example for a phase region is the body of the main loop of an application. It is common that scientific applications have a main loop iterating through time steps or grid elements. If such a *repetitive region* is defined in the source code as a phase region, then the experiments can be done during the same application run. The application is suspended at the beginningof the phase region and new measurements are requested. The application is then released and the analysis is started. When the application encounters again the end of the region, it is suspended and the measured values are retrieved.

3.2 Enabling instrumentation - `psc_instrument`

Measuring performance of an application is commonly based on the ability of the performance tool to "communicate" with the application at runtime. This can be achieved through the instrumentation of the application, i.e. inserting tool specific calls inside the source code or the compiled binary of the application. See also the right hand side of figure 3.1.

In order to enable instrumentation with Periscope, one needs to *prepend the compiling and linking commands* with the call to the `psc_instrument` script. This could usually be done by editing the `Makefile` of the application.

For example, one should replace

```
mpif90 -c <args>
```

with

```
psc_instrument <psc_options> mpif90 -c <args>
```

for a Fortran code, and

```
mpicc -c <args>
```

with

```
psc_instrument <psc_options> mpicc -c <args>
```

for a C/C++ code.

Do not forget to change both the compiling **and** the linking commands.

Please note that the script recognizes the `-c` argument passed to the compiler itself and uses it to decide between the instrumentation and the linking steps. It is thus required that the respective test application is built in two distinct steps: compilation and linking.

Please check the next section for detailed information regarding the most common options in `<psc_options>`. A complete list can be found in table ??.

3.3 Automatic instrumentation

`psc_instrument` is a source code instrumenter. It parses the given source files and modifies them accordingly. Usually this means inserting library calls at the proper places in the code.

Please note that Periscope will create four additional directories to store the instrumented versions of the files:

```
prep
inst
instmod
compmod
```

To switch to verbose mode and follow all actions performed by the instrumenter, please pass the `-v` option to the `psc_instrument` script:

```
psc_instrument -v <other_psc_options> mpif90 -c <args>
```

Frequently used options are:

Option	Description
-t <regions>	List of region types to be instrumented. Some commonly used region types: **mpi**: mpi functions; **omp**: OMP constructs except atomic; **user**: user regions; **none**: no instrumentation, files are only compiled. See also Sections 3.3.1 and 5.3.

-s <SIR file>	This file name will be used for the static program information. It is recommended to name the sir file as the executable, adding the .sir extension. Default: appl.sir . See also Section 3.3.2.
-d	Provide debug information.

3.3.1 Region types

Periscope's automatic instrumentation can handle an entire set of region types. It can detect MPI and OpenMP operations, loops, subroutines and call statements. All these code entries are considered to be separate regions, alongside the user regions that can be defined manually (see Section 3.4).

By default, Periscope instruments only the main routine. There are two ways to instruct Periscope about which region types to instrument for the current application:

The first method is to pass to `psc_instrument` the option `-t` followed by a comma separated list of region types. For example:

```
psc_instrument -t user,mpi <other_psc_options> mpif90 ...
```

Please refer to Table 5.3 for the complete list of valid region types.

Passing region types via the `-t` option will enforce Periscope to apply the same region types configuration to all the files.

Setting different region types per file for instrumentation is also possible. This can be done by editing the `psc_inst_config` file. This file is generated by `psc_instrument` in the application source directory after the first build. It contains a list of all the files that are going to be instrumented along with their corresponding region types. For example:

```
#
# instrumentation control for periscope
#
# id filename [none,mod_only,all,user,sub,call,loop,omp,mpi]
# (if any)
#

1   bt.f            user,mpi
2   initialize.f    user,mpi
```

```
3   exact_solution.f  user,mpi
4   exact_rhs.f       user,mpi
5   set_constants.f   user,mpi
6   adi.f             user,mpi
7   rhs.f             user,mpi
8   zone_setup.f      user,mpi
9   x_solve.f         user,mpi
10  y_solve.f         user,mpi
11  exch_qbc.f        user,mpi
12  z_solve.f         user,mpi
13  solve_subs.f      user,mpi
14  add.f             user,mpi
15  error.f           user,mpi
16  verify.f          user,mpi
```

Editing the region type for a specific file instructs Periscope to apply that kind of instrumentation for that particular file.

For example, in the file listed above one could instruct Periscope to also instrument subroutines and call statements for the `bt.f` file and only loops for the `adi.f` and `rhs.f` files:

```
  #
  # instrumentation control for periscope
  #
  # id filename [none,mod_only,all,user,sub,call,loop,omp,mpi]
  # (if any)
  #

1  bt.f              user,mpi,sub,call
2  initialize.f      user
3  exact_solution.f  user
4  exact_rhs.f       user
5  set_constants.f   user
6  adi.f             loop
7  rhs.f             loop
8  zone_setup.f      user
9  x_solve.f         user
   ...
```

Please note that the settings in the `psc_inst_config` file only apply if the `-t` option **is not passed** when calling `psc_instrument`. Passing `-t` to `psc_instrument` will **overwrite** any changes of the `psc_inst_config` file.

Especially for the debugging phase, it might be interesting to use the `none` value as a region type. This switches off instrumentation for some files and could be useful to circumvent any issues that might occur due to the source instrumenter. Please note that files which are not instrumented cannot be analysed into detail. Thus, the selective instrumentation reduces the overhead, but it is limiting the precision of the analysis with respect to the location in the code.

Although available, the usage of the value `all` for the region type is strongly **not** recommended. If needed, please use it with care, as it frequently produces a high amount of instrumentation overhead.

3.3.2 `.sir` file

Upon successful completion, `psc_instrument` generates:

1. an instrumented *executable* of the application and
2. a *`.sir` file* storing static information about the program.

SIR stands for *Standard Intermediate Representation* and is a format specific to Periscope[1].

Periscope can only start its performance analysis, if both the executable of the application, as well as the `.sir` file is provided.

By default, `psc_instrument` stores the `.sir` file under the name `appl.sir`, in the directory where the link process is executed. You can change the name of the generated file by providing the option `-s` to the instrumenter:

```
psc_instrument -s sirfilename.sir ...
```

The same file name will then have to be passed to the Periscope executable[2] upon startup:

```
$ psc_frontend --sir=sirfilename.sir ...
```

Please note that if `--sir` is not provided, Periscope will search for a `.sir` file called `<applname>.sir`, where `applname` is the actual name of the application executable. It is thus a good practice to name the SIR file as the application itself, just adding the `.sir` extension at the end.

[1]For further information on the SIR format, please check section **??** of this Guide.

[2]More on `psc_frontend` in section **??**.

3.3.3 Fortran particularities - module instrumentation

Fortran modules require special attention in the instrumentation process. This is due to the fact that besides the common objects generated at compile time, there is also an extra module description file (`.mod`) generated for each module source.

The `.mod` files may have different formats from compiler to compiler. Periscope instrumenter uses its own format as well, which most often do not match formats used by compilers.

In this context, one should consider the following when instrumentating Fortran code containing modules:

- if a file `a.f90` refers to the module implemented in `b.f90`, e.g. it contains a statement like `USE MODULE BModule`, then the file `a.f90` can only be instrumented, if the Periscope instrumenter can also load the corresponding module file `bmodule.mod`.
- due to format differences, the Periscope instrumenter can only load `.mod` files generated by itself.
- a `.mod` file can only be generated if the corresponding source file (`.f90`, `.F90`, etc.) is available.

There are two main issues that a user should take care of:

1. The `psc_instrument` needs to know where the `.mod` files can be loaded from. See option `-M` for setting the include paths.
2. If the application uses a module for which the source code is not available, then the files referencing this module cannot be instrumented. They have to be marked in the `psc_inst_conf` with `none` for the region type.

3.3.4 Reducing the instrumentation overhead

Especially in the case of large applications, the automatically instrumented code has a high execution overhead. To overcome this issue, Periscope can be instructed to perform an analysis of the generated overhead and to re-instrument the code accordingly. This can be achieved by means of the `--inst` parameter of the `psc_frontend` executable:

```
psc_frontend --inst=<overhead|all_overhead|analysis>
```

There are three possible automatic re-instrumentation strategies: *overhead*, *all_overhead* and *analysis*.

The *overhead* and *all_overhead* strategies will first determine too fine granular regions and remove their instrumentation. The *overhead* strategy removes only the overhead concerning the single node measurements. Other overheads may still lead to an extended execution time. The *all_overhead* strategy removes all overhead so that the extra execution time produced due to the instrumentation will be negligible.

The *analysis* instrumentation strategy first determines the too fine granular regions, like the previous strategies too, but, unlike those, it will then only instrument those regions which are required in the next experiment. These regions are determined based on the analysis strategy given by the `--strategy` parameter[3].

3.4 Manual instrumentation - `user region`

Besides the regions detected and instrumented automatically by Periscope, the user also has the possibility to define own custom regions.

An user region can be defined by surrounding the corresponding piece of code with the following directives, as also shown before:

Fortran:
```
!$MON USER REGION
S1
S2
...
!$MON END USER REGION
```

C/C++:
```
#pragma start_user_region
S1
S2
...
#pragma end_user_region
```

When `psc_instrument` is called, the source file is parsed and the directives are replaced with proper calls to the Periscope library.

There is no limit on the number of user regions that can be defined in a code.

[3] See Table 5.2.

Any user region has to be defined within one scope of the source code. For example, a user region cannot pass beyond the end of a subroutine, if it starts within that subroutine.

3.5 Starting performance analysis - `psc_frontend`

The Periscope performance measurement and analysis process can be started via the `psc_frontend` executable. For example:

```
$ psc_frontend --apprun=./bt-mz_C.16 --mpinumprocs=16
--force-localhost --debug=1
```

All needed configuration options can be passed to Periscope by means of the command line parameters.

The mandatory parameters which are required for Periscope analysis to start are:

Option	Description
--apprun=<command line>	Specify the command line to start the application. It will be passed to the mpirun command. The executable specified in the command line must exist when Periscope is started.
--mpinumprocs=<np>	Number of MPI processes for the application. For serial applications, please set this value to `1`. Periscope treats serial applications as 1-process MPI applications.

Other frequently used options are:

Option	Description
--debug=<level>	Level of debug output (default: 0).
--force-localhost	Locally start the agents instead of using SSH.

--strategy=<strategy>	Specify one of the following strategies: MPI, SCA, SCABF, P6, P6BF, P6BF_Memory, SCPS_BF, scalability_OMP. Please note: Some strategies are platform dependent (default: all).
--sir=<filename>	SIR file to be used during the analysis (default: `<appl>.sir`)
--propfile=<filename>	Store the detected properties into filename (default: properties.psc)
--ompnumthreads=<threads>	Number of OpenMP threads (default: 1).

Please see table 5.2 for a complete list of options accepted by `psc_frontend`.

On startup, a hierarchy of analysis and communication agents is first created, then the application to be measured is started and the analysis agents attach to the application nodes. The performance data are gathered by means of the monitoring library and communicated to the low-level agents. There it is analysed using the strategy established at the beginning within the frontend and based on the results, the next step of the iterative analysis is established.

The final results are propagated through the agent hierarchy up to the frontend, which then stores them in the properties file.

The frontend is the control point of Periscope. Users can configure and direct the performance analysis process from here. The agent hierarchy and the monitoring library remain transparent to the common user.

3.6 Exploring the results - `GUI`

The frontend writes the found performance properties into a file called `properties_*` with the `.psc` extension. This file is in XML format and can be opened with any off-the-shelf text editor or a spreadsheet application.

Periscope also offers a Graphical User Interface (GUI) for an enhanced visualisation and exploration of the analysis results. It is an Eclipse based plugin, featuring a multi-functional table for displaying and organizing the textual data. Following functionalities are available:

- multiple criteria sorting algorithm
- complex categorization utility

- searching engine using *regular expressions*
- filtering operations
- direct navigation from the bottlenecks to their precise source location using the default IDE editor for that source file type (e.g. CDT/Photran editor).

An outline view for the instrumented code regions that were used in an experiment is also available. The information it shows is a combination of the standard intermediate representation of the analyzed application and the distribution of its bottlenecks. The main goals of the view are to assist the navigation in the source code and attract developer's attention to the most problematic code areas.

The multivariate statistical clustering is another key feature of the plug-in that enhances the scalability of the GUI and provides means of conducting Peta-scale performance analysis. It can effectively summarize the displayed information and identify a runtime behavior possibly hidden in the large amount of data.

Chapter 4

Performance Tuning with Periscope

Performance tuning using PTF (Periscope Tuning Framework) is based on the collaborative work performed by customized tuning plugins on the one side and Periscope as the host application of the plugins on the other side.

The high-level architecture of PTF can be seen in figure 4.1. Similar to using the analysis feature of PTF, users can start and configure the tuning process by calling the `psc_frontend` with appropriate parameters. The option enabling the tuning execution mode of Periscope is `--tune`:

```
$ psc_frontend --tune=<nameofplugin> ...
```

For example, the following will run compiler flags tuning (CFS) on the BT application:

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

Depending on each particular plugin, there might be also other options available for configuration. Please consult the corresponding *User's Guide* for details specific to each of the plugins.

All other components in figure 4.1 are transparent to the users of the plugins and of the PTF tuning feature.

Figure 4.1: Plugin architecture of the Periscope Tuning Framework.

4.1 Tuning plugins

For the current version, PTF provides the following tuning plugins:

CFS: the *Compiler Flags Selection* plugin tunes the application to find the combination of compiler flags with which the best execution time is achieved.

DVFS: the *Dynamic Voltage and Frequency Scaling* plugin tunes the energy consumption of an application.

Master-Worker: the *Master-Worker* plugin tunes the number of tasks and processes to be used by applications based on the master-worker paradigm.

MPI Paramenters: automatically optimizes the values of a user selected subset of MPI configuration parameters.

Patterns: the *Parallel Patterns* plugin works on applications using a *Pipeline-based* execution to determine the best combination of the pipeline stages.

4.2 Tuning advice

As a result of the tuning process, Periscope generates an XML file describing:

- the final *tuning advice* to be applied to the application
- the *tuning scenarios* which were used in searching the best advice
- other information specific to the tuning plugin, like, for example, the tuning parameters, the execution times, or the energy consumption.

4.3 The tuning flow

Being the host of the tuning plugins, Periscope provides several services to build a standard tuning flow.

Data model

The main components of the tuning data model are:

tuning parameters: represent the parameters based on which a tuning of the application can be done. These are plugin dependent and their semantics is strictly defined in each plugin. For example, the CFS plugin uses compiler flags as tuning parameters, while the MPI Parameters plugin uses MPI related switches and parameters.

For most plugins, the tuning parameters are the given by user input through a configuration file.

tuning scenario: represents a combination of tuning parameters. The application is analysed by Periscope using one scenario at a time.

Scenarios are computed internally based on a chosen search algorithm. Users can choose between different search algorithms, but cannot directly define tuning scenarios.

tuning space: the set of all valid tuning scenarios.

analysis result: the analysis result associated with one specific tuning scenario. Results are partially displayed in the final tuning advice provided by Periscope.

Operations

On the functional side, the tuning flow is supported by means two main operations:

search algorithm: the search algorithm generates the tuning space and delivers the next scenario to be evaluated. For most tuning plugins, users can choose the preferred search algorithm.

There are several search algorithms available: exhaustive search, individual search, random search and GDE3 search (one genetic algorithm).

pre-analysis: some plugins require an analysis step before the tuning process can start. The Periscope performance analysis feature is being used in this case.

Required pre-analysis is very much plugin specific. Please consult the given *User's Guide* to see whether user input is possible for each particular case.

4.4 Uninstrumented applications

The CFS Plugin an the MPI Plugin also allow tunning of uninstrumented applications, but this is strongly not encouraged. When measuring performance for uninstrumented applications, Periscope relies exclusively on the data retrieved from the system. This mostly leads to inaccuracies, especially for applications with a short execution time. If one does want to use the uninstrumented version, this can be done by passing the `--uninstrumented` option to the `psc_frontend` process at the command line.

Chapter 5

Configuration Options

5.1 Environment Variables

Option	Description
PSC_ROOT	Root directory of the Periscope installation.
PERISCOPE_DEBUG	0..2 0=quiet 1=startup, found properties in each search 2=candidate properties and found properties in each strategy step

5.2 The frontend - `psc_frontend`

The frontend starts up the application and the agent hierarchy.

Option	Description

--apprun=<appl cmdline>	This is the command line used to start the application. It should be the same as in `mpirun -np <procs> <appl cmdline>`. This value is also used to determine the name of the SIR file, when `--sir` is missing. The executable specified in the command line must exist when Periscope is started. This is true also for the cases where the tuning feature of Periscope is used in combination with plugins which by themselves re-build the application from its source files (e.g. the CFS plugin).
--bg-mode=SMP|DUAL|VN	The node mode used on the Bluegene.
--debug=level	Level of debugging. All debug output up to that level will be printed. Default: PERISCOPE_DEBUG or 0
--delay=<n>	Number of phase executions that are skipped before the search is started. This is useful for applications that have a different behaviour at the beginning.
--dontcluster	Do not use online clustering for the detected bottlenecks.
--force-localhost	Locally start the agents instead of using SSH.
--help	Help information

--inst=overhead | all_overhead | analysis	Automatic instrumentation strategy. The *overhead* and *all_overhead* strategies will first determine too fine granular regions and remove their instrumentation. It will then apply the selected analysis strategy. The overhead strategy removes the overhead that influences the single node measurements but other overheads may lead to a prolongation of the execution. The all_overhead strategy removes all overhead so that the prolongation of the execution will be negligible. The *analysis* instrumentation strategy will first determine too fine granular regions and will then instrument exactly those regions that are required in the next experiment.
--inst-folder=<relative path>	Path to the folder with the instrumented sources relative to the execution directory. This is needed to modify the instrumentation in during automatic instrumentation.
--make=<make command>	Command to be issued in order to recompile the application.
--maxcluster=<n>	Maximum number of MPI processes analyzed by a single analysisagent. It is not used on the Bluegene since the analysisagents are running on the IO nodes. All processes on the compute nodes of an IO nodes connect to its analysisagent. Default: 64
--maxfan=<n>	Determines the fan-out of the tree of high-level agents in interactive mode. Default: 4
--mpinumprocs=<n>	Number of MPI processes to be started.

--nprops=<n>	Specifies the number of properties the frontend prints to standard output. Regardless of this value, all properties are output to the properties file. Default: 50.
--ompnumthreads=<n>	Number of OMP threads to be started per MPI process. Default: 1.
--pedantic	Shows all detected properties.
--phase=<fileid:rfl>	Specifies the phase region via the fileid and the region first line number. If no phase region is specified, a user region is selected if at least one is given in the code. If multiple are given, it is undefined which is selected. If no user region is given, the main program is the user region and the program will be restarted for each strategy step. If you mark the phase region via a user region and would like to use user regions also to guide analysis, you have to give the fileid and rfl for the phase region.
--propfile=<filename>	Specify the file to use when exporting the properties. Default: properties.psc
--psc-inst-config=<relative path to inst config file>	File name relative to the execution directory.
--quiet	Turns off the debug messages.
--srcrev=<source revision>	Specify the source code revision. It will be written in the output file.
--sir=<filename>	SIR file of the application to be analyzed. Default: The file name is composed of the executable's name and the extension .sir. If `--apprun` is omitted, the default is appl.sir.

--src-folder=<relative path>	Path to the source folder relative to the execution directory. This is needed to touch the sources to trigger recompilation of the instrumented versions.
--strategy=<strategyname>	Strategy used by analysisagent. Currently one of MPI - MPI Communication analysis OMP - OpenMP analysis P6 - Power6 Analysis (only on Power6 machines) P6BF - Power6 Breadth First (only on Power6 machines) P6BF_Memory - Power6 Memory Behavior Analysis (only on Power6machines) SCPS_BF - Generic memory analysis strategy scalability_OMP - Automatic OpenMP scalability analysis
--timeout=<secs>	Timeout for startup of the agent hierarchy. Default: varying depending on the number of processes
--uninstrumented	**Autotuning only**: instructs Periscope to tune an uninstrumented application. Use with caution. See also Section 4.2.
--version	Displays the version of Periscope.
--with-deviation-control	Enables performance deviation control on POWER architectures.

5.3 The instrumenter - `psc_instrument`

`psc_instrument` prepares the application for analysis with Periscope. In the existing `Makefile`, the compilation step generating the object files has to be modified by prepending `psc_instrument` to the compiler. The script will preprocess the file, instrument it, and finally call the compiler for generating the instrumented object file. In addition, the compiler has to be augmented with `psc_instrument` in the linking step too. Here `psc_instrument` will link the monitoring library to the executable as well as generate the SIR containing the static information of the program.

The instrumentation is controlled by a file called `psc_inst_config` in which the file id and the region types to be instrumented are given for each file individually.

The calling syntax is:

```
psc_instrument [-t <regions>] [-s <sirfile>] [-f] [-n]
[-d] [-v] <compiler> [<options>] <file> [<libs>]
```

Please note that, while `psc_instrument` can process both Fortran and C/C++ files, some options are specific to only one of the two programming languages.

Option	Description
-d	Provide debug information.
-f <fixed|free>	**Fortran only:** forces a specific Fortran file format. By default, `.f90` files are in free format.
-M <path>	**Fortran only:** location where module files are placed.
-n	Dryrun: run the makefile without executing the commands.
-s <SIR file>	This file name will be used for the static program information. It is recommended to name the SIR file as the executable, adding the .sir extension. Default: appl.sir

-t <regions>	List of region types to be instrumented. This overwrites the specifications in `psc_inst_config`. Fortran and C/C++: **all**: all regions (use with care, as this option will generate a lot of instrumentation overhead); **loop**: outermost loops only; **mpi**: mpi functions; **none**: no instrumentation, files are only compiled; **omp**: OMP constructs except atomic; **par**: OMP parallel and worksharing constructs; **sub**: subroutines; **sync**: OMP synchronization statements except atomic; **user**: user regions. Fortran only: **call**: call statements; **forall**: forall statements; **io**: IO statements; **mod_only**: no instrumentation but processing by the instrumenter to generate compatible module files; **nestedloop**: non-perfectly nested loops; **vect**: vector statements. Default: **all**.
-v	Verbose.
<compiler>	Compiler for final compilation of the instrumented files, e.g., `mpif90` or `mpicc`.
<file>	Name of the file to be instrumented. **Fortran only:** file extensions .f90 and .F90 determine free source format, while .f determines fixed source format.
<libs>	Libraries for linking.
<options>	List of compiler options used in the original call to the compiler. These are passed to the compiler. Please note that if `-c` is specified in the options list, `psc_instrument` will instrument and compile the given file. Otherwise it will link the application.

Chapter 6

Advanced user information - technical details

The application and the agent network are started through the `psc_frontend` process. First the set of available processors is analysed and based on this the mapping of application and analysis agent processes are determined. Both the application and the agent hierarchy are then started and a command is propagated from the frontend down to the analysis agents to start the search. The search is performed according to a search strategy selected when the frontend is started.

Each of the analysis agents, i.e. the nodes of the agent hierarchy, searches autonomously for inefficiencies in a subset of the application processes.

The application processes are linked with a monitoring system that provides the *Monitoring Request Interface* (MRI). The agents attach to the monitor via sockets. The MRI allows the agent to configure the measurements, to start, to halt, to resume the execution, and to retrieve the performance data. The monitor currently only supports summary information.

At the end of the local search, the detected performance properties are reported back via the agent hierarchy to the frontend.

6.1 Agent hierarchy

The layout of the agent hierarchy can be controlled by the user by means of the specific parameters of the `psc_frontend` executable:

maxfan: determines the fan-out of the tree of high-level agents. By default this is set to 4.

maxcluster: gives the maximum number of MPI processes analysed by a single analysisagent. The default number is 64.

Further information on how the agents work within a specific run of PTF can be gathered by using the `--selective-debug` parameter of the same `psc_frontend` executable:

```
--selective-debug= <level1>,<level2>...
```

with the following *levels* being relevant for the agent hierarchy:

AgentApplComm: displays information regarding the communication between the agents and the application nodes.

AutotuneAgentStrategy: displays information regarding the analysis strategy used in the analysis agent for tuning. To be used only when the tuning feature of PTF is being used.

Other values for the `--selective-debug` parameter can be found in the *PTF Developer's Guide*.

Using a proper layout of the agent hierarchy is very important especially when performing analysis and tuning of applications on large systems.

Please note that, if the `--force-localhost` option of the `psc_frontend` executable is being used, then the entire agent hierarchy will be started on a single node. This is not recommended for applications using a large number of processes, as the communication between the agents and the application nodes would result in a bottleneck with a negative influence on the overall analysis time.

Chapter 7

Known Issues

- Automatic restart of the application does not work on the Bluegene. Make sure, you specify a user region that is executed repetitively.
- C instrumentation: The name of an OMP pragma should not occur again as a string in another context in this pragma, e.g., in a variable name.
- Measurements might be wrong in recursive algorithms.
- Multiple running instances of Periscope might not work on some systems.

Examples

You can find two examples with the adapted makefile in ~/psc/test/add and ~/psc/test/cx_parallel. Both directories include a file makefile.psc_instrument.

Example on SuperMUC

Periscope can be used in batch jobs.

Example batch script:

```
#!/bin/bash
#PBS -j oe
#PBS -S /bin/bash
#PBS -l select=80:ncpus=1
#PBS -l walltime=0:20:00
#PBS -N cx64
#PBS -M gerndt@in.tum.de
#PBS -m e
. /etc/profile
cd psc/test/cx_parallel/
psc_regsrv &
sleep 10
sudo /lrz/sys/lrz_perf/bin/lrz_perf_off_hlrb2

psc_frontend --apprun=cx --mpinumprocs=64 --strategy=SCA --debug=1
```

```
#!/bin/bash
#
#@ job_type = parallel
#@ class = test
#@ island_count = 1
#@ node = 1
```

```
#@ wall_clock_limit = 1:12:30
#@ job_name = add
#@ network.MPI = sn_all,not_shared,us
#@ initialdir = $(HOME)/TestingRepository/add/
#@ output = $(jobid).out
#@ error = $(jobid).err
#@ notification = never
#@ notify_user = gerndtin.tum.de
#@ queue
. /etc/profile
. /etc/profile.d/modules.sh

psc_frontend --apprun=add --mpinumprocs=4 --sir=add.sir --tune=demo --force-localhost --debug=1
```