SCORE-P
USER MANUAL
8.4 (revision 624046d2)

Score-P
Scalable performance measurement
infrastructure for parallel codes

Fri Mar 15 2024 06:50:47

Score-P License Agreement

Contents

Chapter 1

Introduction

This document provides an introduction to `Score-P`: the *Scalable Performance Measurement Infrastructure for Parallel Codes*. It is a software system that provides a measurement infrastructure for profiling and event trace recording of High Performance Computing (HPC) applications. It has been developed within the framework of the Scalable Infrastructure for the Automated Performance Analysis of Parallel Codes (`SILC`) project funded by the German Federal Ministry of Education and Research (`BMBF`) under its HPC programme and the Performance Refactoring of Instrumentation, Measurement, and Analysis Technologies for Petascale Computing (`PRIMA`) project,funded by the United States Department of Energy (`DOE`) with the goals of being highly scalable and easy to use.

The partners involved in the development of this system within the SILC and PRIMA projects were:

- `Forschungszentrum Jülich`,

- `German Research School for Simulation Sciences`,

- `Gesellschaft für numerische Simulation mbH`,

- `Gesellschaft für Wissens- und Technologietransfer der TU Dresden (GWT-↩ TUD GmbH)`,

- `Rheinisch-Westfälische Technische Hochschule (RWTH) Aachen`,

- `Technische Universität Dresden`,

- `Technische Universität München`,

- and `University of Oregon`

The goal of Score-P is to simplify the analysis of the behavior of high performance computing software and to allow the developers of such software to find out where and why performance problems arise, where bottlenecks may be expected and where their codes offer room for further improvements with respect to the run time. A number of tools have been around to help in this respect, but typically each of these tools has only handled a certain subset of the questions of interest. A software developer who wanted to have a complete picture of his code therefore was required to use a multitude of programs to obtain the desired information. Most of these utilities work along similar principles. The first step is usually an instrumentation of the code to be investigated. Next, the instrumented programs are executed and write out (often very large amounts of) performance data. These data are then finally graphically displayed and analyzed after the end of the program run. In certain special cases, the visualization and analysis of the program behavior is also done while the program is running.

A crucial problem in the traditional approach used to be the fact that each analysis tool had its own instrumentation system, so the user was commonly forced to repeat the instrumentation procedure if more than one tool was to be employed. In this context, Score-P offers the user a maximum of convenience by providing the Opari2 instrumenter as a common infrastructure for a number of analysis tools like `Scalasca`, `Vampir`, and `Tau` that obviates the

need for multiple repetitions of the instrumentation and thus substantially reduces the amount of work required. It is open for other tools as well. Moreover, Score-P provides the new Open Trace Format Version 2 (OTF2) for the tracing data and the new CUBE4 profiling data format which allow a better scaling of the tools with respect to both the run time of the process to be analyzed and the number of cores to be used.

Score-P supports the following parallel programming paradigms:

Multi-process paradigms:

- *MPI*
- *SHMEM*

Thread-parallel paradigms:

- *OpenMP*
- *Pthreads*

Accelerator-based paradigms:

- *CUDA*
- *HIP*
- *OpenCL*
- *OpenACC*

And possible combinations from these including simple *serial* programs.

In addition, Score-P supports event recording for the following programming interfaces:

Memory management interfaces:

- *C*, up to C11 (i.e., `malloc/free`)
- *C++*, up to C++14 and old PGI/EDG (i.e., `new/delete`)
- *High bandwidth memory API*, from the memkind library (i.e., `hbw_malloc/hbw_free`)
- *MPI*
- *SHMEM*

I/O interfaces:

- *POSIX I/O* (i.e., `open/close`)
- *POSIX asynchronous I/O* (i.e., `aio_read/aio_write`)
- *ISO C standard I/O* (i.e., `fopen/fclose`)
- *MPI I/O*

1.1   About this Document

This document consists of three parts. This chapter is devoted to a basic introduction to performance analysis in general and the components of the Score-P system in particular. Chapter 'Getting Started' is a beginner's guide to using the Score-P tool suite. It demonstrates the basic steps and commands required to initiate a performance analysis of a parallel application. In the Chapters 'Application Instrumentation', 'Application Sampling', and 'Application Measurement', the reader can find more detailed information about the components of Score-P. Chapter 'Performance Analysis Workflow Using Score-P' provides a typical workflow of performance analysis with Score-P and detailed instructions.

1.2   Getting Help and Support

The Score-P project uses various mailing lists to coordinate the development and to provide support to the user community. An overview of the available mailing lists can be found in the Table Score-P mailing lists.

You can subscribe to the `news@score-p.org` and `support@score-p.org` by ...

List Address	Subscription	Posting	Usage
`news@score-p.org`	open	core team	Important news regarding the Score-P software, e.g., announcements of new releases.
`support@score-p.↩` `org`	closed	anyone	Bug reports and general user support for the Score-P software.

Table 1.1 Score-P mailing lists

1.3   Basics of Performance Optimization

Performance optimization is a process that is usually executed in a work cycle consisting of a number of individual steps as indicated in Figure 1.1.
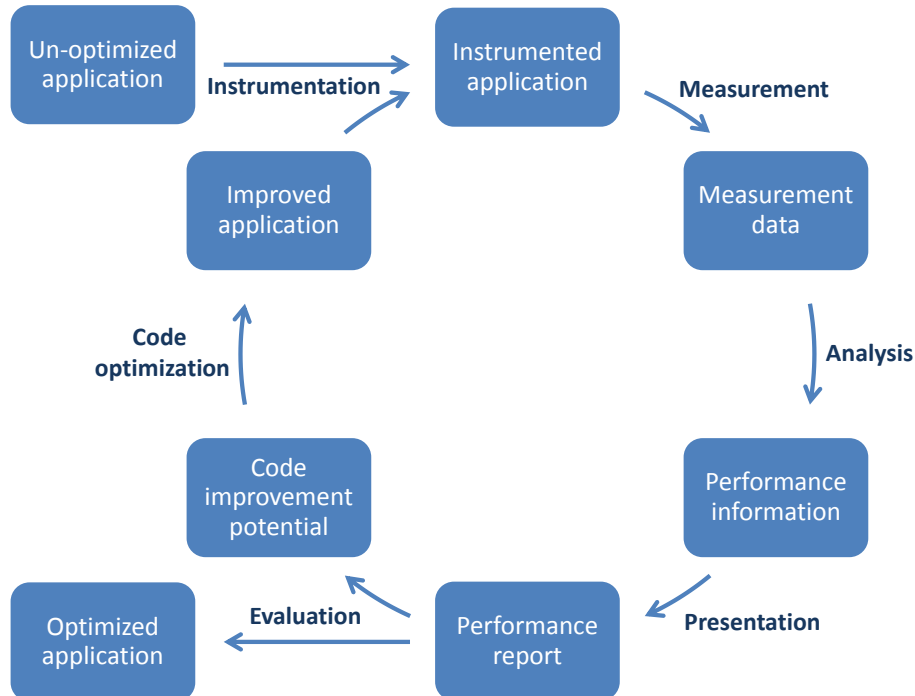
Figure 1.1 The performance optimization cycle

Thus, the process always begins with the original application in its unoptimized state. This application needs to be instrumented, i. e. it must be prepared in order to enable the measurement of the performance properties to take place. There are different ways to do this, including manual instrumentation of the source code by the

user, automatic instrumentation by the compiler, or linking against pre-instrumented libraries. All these options are available in Score-P.

When the instrumented application obtained in this way is executed, the additional commands introduced during the instrumentation phase collect the data required to evaluate the performance properties of the code. Depending on the user's requirements, Score-P allows to store these data either as a profile or as event traces. The user must keep in mind here that the execution of the additional instructions of course requires some run time and storage space. Thus the measurement itself has a certain influence of the performance of the instrumented code. Whether the perturbations introduced in this way have a significant effect on the behavior depends on the specific structure of the code to be investigated. In many cases the perturbations will be rather small so that the overall results can be considered to be a realistic approximation of the corresponding properties of the uninstrumented code. However, certain constructions like regions with very small temporal extent that are executed frequently are likely to suffer from significant perturbations. It is therefore advisable not to measure such regions.

The next step is the analysis of the data obtained in the measurement phase. Traditionally this is done post mortem, i. e. after the execution of the instrumented application has ended. If the collected data are event traces then a more detailed investigation is possible than in the case of profiles. In particular, one can then also look at more sophisticated dependencies between events happening on different processes.

The optimization cycle then continues with the presentation of the analysis results in a report. Here it is important to eliminate the part of the information that is irrelevant for the code optimization from the measured data. The reduction of the complexity achieved in this way will simplify the evaluation of the data for the user. However, care must be taken in order not to present the results in a too abstract fashion which would hide important facts from the user.

The performance report then allows the user to evaluate the performance of the code. One can then either conclude that the application behaves sufficiently well and exit the optimization cycle with the optimized version of the software being chosen as the final state, or one can proceed to identify weaknesses that need to be addressed and the potential for improvements of the code.

In the latter case, one then continues by changing the source code according to the outcome of the previous step and thus obtains an improved application that then can again be instrumented to become ready for a re-entry into the optimization cycle.

1.4 Score-P Software Architecture Overview

In order to allow the user to perform such an optimization of his code (typically written in Fortran, C, or C++ and implemented in a serial way or using a parallelization via an multi-process, thread-parallel, accelerator-based paradigm, or a combination thereof), the Score-P system provides a number of components that interact with each other and with external tools. A graphical overview of this structure is given in Fig. 1.2. We shall now briefly introduce the elements of this structure; more details will be given in the later chapters of this document.

In order to instrument an application, the user needs to recompile the application using the Score-P instrumentation command, which is added as a prefix to the original compile and link command lines. It automatically detects the programming paradigm by parsing the original build instructions and utilizes appropriate and configurable methods of instrumentation. These are currently:

- compiler instrumentation,

- MPI and SHMEM library interposition,

- source code instrumentation via the TAU instrumenter,

- OpenMP source code instrumentation using Opari2,

- Pthread and OpenCL instrumentation via GNU ld library wrapping,
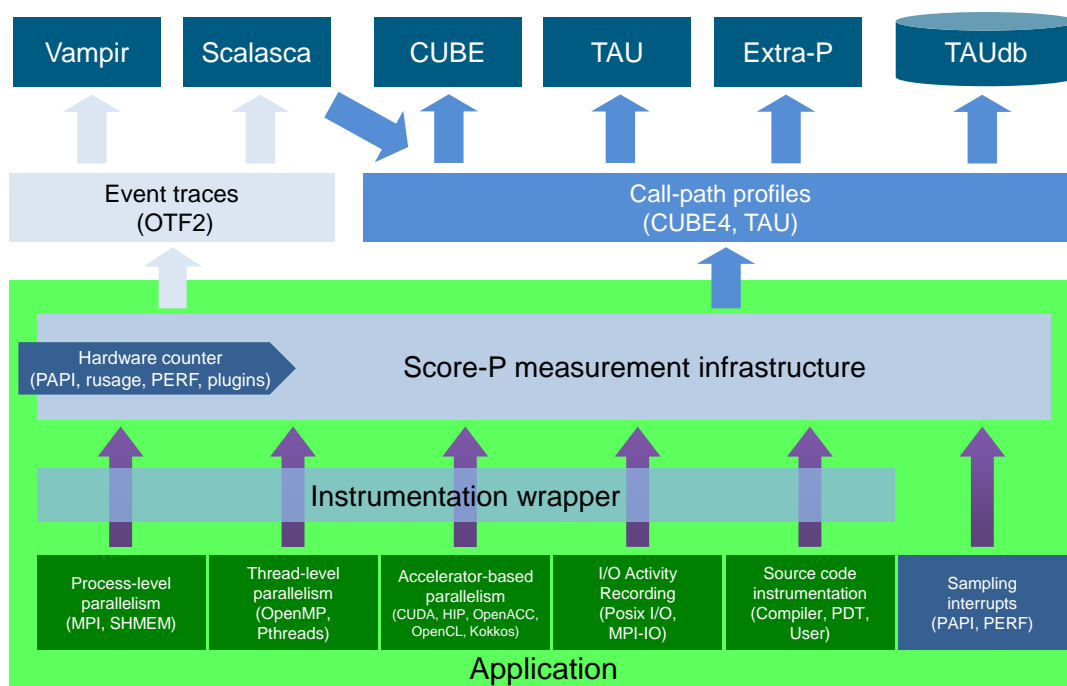
Figure 1.2 Overview of the Score-P measurement system architecture and the tools interface.

- CUDA instrumentation via the NVIDIA CUDA Profiling Tools Interface (CUPTI),

- HIP instrumentation via the AMD ROCm interface

- OpenACC instrumentation using the OpenACC Profiling Interface,

- I/O library interposition.

While the first three of these methods are based on using tools provided externally, the Opari2 instrumenter for OpenMP programs is a part of the Score-P infrastructure itself. It is an extension of the well known and frequently used `OpenMP Pragma And Region Instrumenter` system (Opari) that has been successfully used in the past in combination with tools like Scalasca, VampirTrace and ompP. The fundamental concept of such a system is a source-to-source translation that automatically adds all necessary calls to a runtime measurement library allowing to collect runtime performance data of Fortran, C, or C++ OpenMP applications. This translation is based on the idea of OpenMP pragma/directive rewriting. The key innovation in Opari2, as compared to its predecessor, is the capability to support features introduced in version 3.0 of the OpenMP standard, in particular its new tasking functionality and OpenMP nesting. Opari used to work by automatically wrapping OpenMP constructs like parallel regions with calls to the portable OpenMP monitoring interface `POMP`. In order to reflect the above-mentioned extensions, this interface also had to be replaced by an enhanced version, POMP2.

Additionally, the user may instrument the code manually with convenient macros provided by Score-P. Score-P also supports sampling functionality that provides an alternative to direct instrumentation.

During measurement, the system records several performance metrics including execution time, communication metrics, and optionally hardware counters. Performance data is stored in appropriately sized chunks of a preallocated memory buffer that are assigned to threads on demand, efficiently utilizing the available memory and avoiding measurement perturbation by flushing the data to disk prematurely.

Without recompilation, measurement runs can switch between *tracing* and *profiling* mode. In tracing mode, the performance events are passed to the tracing back-end of Score-P and are written to files for subsequent post

mortem analysis using Scalasca or Vampir. This backend uses the newly developed Open Trace Format 2 (OTF2), the joint successor of the `Open Trace Format` used by Vampir and the Epilog format used by Scalasca. The Score-P system contains a new library with reading and writing routines for OTF2. Basically, OTF2 is a full merge of its two predecessors that retains all their features, and it is planned to become the default data source for future versions of both Vampir and Scalasca. In this way, the user is free to choose between these two complementary tools to investigate the trace files and may select the one that is more appropriate for the specific question at hand.

In profiling mode, the performance events are summarized at runtime separately for each call-path like in Scalasca. Additionally, support for phases, dynamic regions and parameter-based profiling has been integrated. The collected data is passed to the Score-P's profiling back-end CUBE4 for post mortem analysis using Scalasca or TAU. Also in profiling mode, Score-P supports the automatic detection of MPI wait states. Usually such inefficiencies are important bottlenecks and are thoroughly investigated by means of automatic trace analysis and subsequent visual analysis using a time-line representation. In the case of Score-P wait time profiling, inefficiencies are detected immediately when the respective MPI call is completed and stored as an additional metric in the call-path profile. In comparison to earlier versions of CUBE, this new one features a more powerful data model, more flexibility in the specification of system resource hierarchies and display parameters, and various techniques to enhance the efficiency that result in a much better scaling behavior of the analysis tool even in a range of tens of thousands of processes.

As a rough guideline for users who are uncertain which of these two modes to employ, we provide a brief comparison of their main advantages and disadvantages. Specifically, tracing mode allows to retain temporal and spatial connections, and it can reflect the dynamical behavior to an arbitrary precision. Moreover, statistical information and profiles may be derived from the program traces. On the other hand, the amount of data that is produced in the tracing mode can become prohibitively large; profiles tend to require much less storage space. In addition, the additional load that is imposed on the process, and hence the perturbations of the behavior of the code to be analyzed, are much smaller in profiling mode than in tracing mode. And finally we mention that the accurate synchronization of the clocks is an important aspect in tracing mode that may cause difficulties.

1.5 Overview article and citing

Please refer to the *Score-P measurement infrastructure* by citing the overview article `Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir` (this link provides citations in various formats) (`PDF`), or use the DOI `10.↩ 1007/978-3-642-31476-6_7` directly.

Version-specific DOIs of the software can be found on `Score-P's Zenodo page`.

See also the file `CITATION.cff` in `<scorep_prefix>/share/doc/scorep/` for machine-readable citation information.

1.6 Acknowledgment

- European High-Performance Computing Joint Undertaking (EuroHPC JU)

- German Aerospace Center (DLR)

- German Research Foundation (DFG)

- Helmholtz Association

- Information Technology for European Advancement (ITEA2)

- Intel

- NVIDIA

- Oak Ridge National Laboratory (ORNL)

- Siemens AG

Chapter 2

Getting Started

In order to quickly introduce the user to the Score-P system, we explain how to build and install the tool and look at a simple example. We go through the example in full detail.

As mentioned above, the three core steps of a typical work cycle in the investigation of the behavior of a software package can be described as follows:

- *Instrumentation of user code*: Calls to the measurement system are inserted into the application. This can be done either fully automatically or with a certain amount of control handed to the software developer.

- *Measurement and analysis*: The instrumented application is executed under the control of the measurement system and the information gathered during the run time of this process is stored and analyzed.

- *Examination of results*: The information about the behavior of the code at run time is visualized and the user gets the opportunity to examine the reported results.

After building and installing the tool, we shall go through these three steps one after the other in the next sections. This will be followed by a full workflow example. For getting detailed presentations of available features, see Section 'Application Instrumentation' for the instrumentation step and Section 'Application Measurement' for the measurement.

2.1   Score-P Quick Installation

The Score-P performance analysis tool uses the GNU Autotools (Autoconf, Automake, Libtool and M4) build system. The use of Autotools allows Score-P to be build in many different systems with varying combinations of compilers, libraries and MPI implementations.

Autotools based projects are build as follows:

1. The available compilers and tools available are detected from the environment by the configure script.

2. Makefiles are generated based on the detected compilers and tools.

3. The generated Makefile project is then built and installed.

Score-P will have features enabled or disabled, based on the detection made by the Autotools generated configure script. The following 2 sub-sections cover mandatory prerequisites as well as optional features that are enabled based on what is available in the configured platform.

2.1.1 Prerequisites

To build Score-P, C, C++ and Fortran compilers and related tools are required. These can be available as modules (typically on super-computer environments) or as packages (on most Linux or BSD distributions).

For Debian based Linux systems using the APT package manager, the following command (as root) is sufficient to build Score-P with minimal features enabled:

```
apt-get install gcc g++ gfortran mpich2
```

On Red-Hat and derivative Linux systems running the YUM package manager, in a similar way:

```
yum install gcc g++ gfortran mpich2
```

For users of the SuperMUC, it is recommended to load the following modules:

```
module load ccomp/intel/12.1 fortran/intel/12.1 \
          mpi.ibm/5.2_PMR-fixes papi/4.9
```

2.1.2 General Autotools Build Options

System administrators can build Score-P with the familiar:

```
mkdir _build
cd _build
../configure && make && make install
```

The previous sequence of commands will detect compilers, libraries and headers, and then build and install Score-P in the following system directories:

```
/opt/scorep/bin
/opt/scorep/lib
/opt/scorep/include
/opt/scorep/share
```

Users that are not administrators on the target machine may need to install the tool in an different location (due to permissions). The prefix flag should be specified with the target directory:

```
../configure --prefix=<installation directory>
```

For example, in the `install/scorep` directory on his/her home folder:

```
../configure --prefix=$HOME/install/scorep
```

In this case, the user's `PATH` variable needs to be updated to include the `bin` directory of Score-P, and the appropriate library and include folders specified (with `-L` and `-I`) when instrumenting and building applications.

Users of the SuperMUC (after loading the required modules mentioned previously), can issue the following command to configure Score-P:

```
../configure --prefix=$HOME/install/scorep --enable-static    \
    --disable-shared --with-nocross-compiler-suite=intel      \
    --with-mpi=openmpi --with-papi-header=$PAPI_BASE/include \
    --with-papi-lib=$PAPI_BASE/lib
```

2.1.3 Score-P Specific Build Options

In addition to general options available in all Autotools based build systems, there are Score-P configuration flags. These can be printed out by passing the `--help` flag to the configure script.

They are usually self explanatory. Here is a list of them with a short explanation:

- `--with-nocross-compiler-suite=(gcc|ibm|intel|pgi|clang)`

  Specifies the compiler suite to use when not cross-compiling. Selecting one of the options sets all relevant variables to their expected names. These are CC, FC, F77, as well as the linker, preprocessor, etc.

- `--with-frontend-compiler-suite=(gcc|ibm|intel|pgi|clang)`

  Similar to the previous configuration flag, but for cross-compiling environments.

- `--with-mpi=(mpich2|impi|openmpi)`

  The MPI compiler and runtime suite to use. Currently there are entries for MPICH2, Intel MPI and Open MPI.

- `--with-shmem=(openshmem|openmpi|sgimpt)`

  The SHMEM compiler suite to build this package in non cross-compiling mode. Usually autodetected. Needs to be in $PATH.

- `--with-otf2=(yes|<otf2-bindir>)`

  An already install OTF2 can be specified with this flag. This is usually not necessary since OTF2 is built together with Score-P. Specify yes if the tool is in your $PATH, otherwise specify the full path.

- `--with-opari2=(yes|<opari2-bindir>)`

  Similar to the previous configuration flag, but for OPARI2.

- `--with-cube=(yes|<cube-bindir>)`

  Similar to the previous two configuration flags, but for CUBE.

2.2 Instrumentation

Various analysis tools are supported by the Score-P infrastructure. Most of these tools are focused on certain special aspects that are significant in the code optimization process, but none of them provides the full picture. In the traditional workflow, each tool used to have its own measurement system, and hence its own instrumenter, so the user was forced to instrument his code more than once if more than one class of features of the application was to be investigated. One of the key advantages of Score-P is that it provides an instrumentation system that can be used for all the performance measurement and analysis tools, so that the instrumentation work only needs to be done once.

Internally, the instrumentation itself will insert special measurement calls into the application code at specific important points (events). This can be done in an almost automatic way using corresponding features of typical compilers, but also semi-automatically or in a fully manual way, thus giving the user complete control of the process. In general, an automatic instrumentation is most convenient for the user. However, this approach may lead to too many and/or too disruptive measurements, and for such cases it is then advisable to use selective manual instrumentation and measurement instead. For the moment, we shall however start the procedure in an automatic way to keep things simple for novice users.

To this end, we need to ask the Score-P instrumenter to take care of all the necessary instrumentation of user and MPI functions. This is done by using the `scorep` command that needs to be prefixed to all the compile and link commands usually employed to build the application. Thus, an application executable `app` that is normally generated from the two source files `app1.f90` and `app2.f90` via the command:

```
mpif90 app1.f90 app2.f90 -o app
```

will now be built by:

```
scorep  mpif90 app1.f90 app2.f90 -o app
```

using the Score-P instrumenter.

In practice one will usually perform compilation and linking in separate steps, and it is not necessary to compile all source files at the same time (e.g., if makefiles are used). It is possible to use the Score-P instrumenter in such a case too, and this actually gives more flexibility to the user. Specifically, it is often sufficient to use the instrumenter not in all compilations but only in those that deal with source files containing MPI code. However, when invoking the linker, the instrumenter must *always* be used.

When makefiles are employed to build the application, it is convenient to define a placeholder variable to indicate whether a "preparation" step like an instrumentation is desired or only the pure compilation and linking. For example, if this variable is called PREP then the lines defining the C compiler in the makefile can be changed from:

```
MPICC = mpicc
```

to

```
MPICC = $(PREP) mpicc
```

(and analogously for linkers and other compilers). One can then use the same makefile to either build an instrumented version with the

```
make PREP="scorep"
```

command or a fully optimized and not instrumented default build by simply using:

```
make
```

in the standard way, i.e. without specifying PREP on the command line. Of course it is also possible to define the same compiler twice in the makefile, once with and once without the PREP variable, as in:

```
MPICC           = $(PREP) mpicc
MPICC_NO_INSTR =         mpicc
```

and to assign the former to those source files that must be instrumented and the latter to those files that do not need this.

2.3   Measurement and Analysis

Once the code has been instrumented, the user can initiate a measurement run using this executable. To this end, it is sufficient to simply execute the target application in the usual way, i.e.:

```
mpiexec $MPIFLAGS app [app_args]
```

in the case of an MPI or hybrid code, or simply:

```
app [app_args]
```

for a serial or pure OpenMP program. Depending on the details of the local MPI installation, in the former case the `mpiexec` command may have to be substituted by an appropriate replacement.

When running the instrumented executable, the measurement system will create a directory called `scorep-Y`↩ `YYYMMDD_HHMM_XXXXXXXX` where its measurement data will be stored. Here `YYYYMMDD` and `HHMM` are the date (in year-month-day format) and time, respectively, when the measurement run was started, whereas `XX`↩ `XXXXXX` is an additional identification number. Thus, repeated measurements, as required by the optimization work cycle, can easily be performed without the danger of accidentally overwriting results of earlier measurements. The environment variables `SCOREP_ENABLE_TRACING` and `SCOREP_ENABLE_PROFILING` control whether event trace data or profiles are stored in this directory. By setting either variable to `true`, the corresponding data will be written to the directory. The default values are `true` for `SCOREP_ENABLE_PROFILING` and `false` for `SCOREP_ENABLE_TRACING`.

2.4   Report Examination

After the completion of the execution of the instrumented code, the requested data (traces or profiles) is available in the indicated locations. Appropriate tools can then be used to visualize this information and to generate reports, and thus to identify weaknesses of the code that need to be modified in order to obtain programs with a better performance. A number of tools are already available for this purpose. This includes, in particular, the `CUBE4 performance report explorer` for viewing and analyzing profile data, `Vampir` for the investigation of trace information, and the corresponding components of the `TAU` toolsuite.

2.5   Simple Example

As a specific example, we look at a short C code for the solution of a Poisson equation in a hybrid (MPI and OpenMP) environment. The corresponding source code comes as part of the Score-P distribution under the *scorep/test/jacobi/* folder. Various other versions are also available - not only hybrid but also for a pure MPI parallelization, a pure OpenMP approach, and in a non-parallel way; and, in each case, not only in C but also in C++ and Fortran.

As indicated above, the standard call sequence:

```
mpicc -std=c99 -g -O2 -fopenmp -c jacobi.c
mpicc -std=c99 -g -O2 -fopenmp -c main.c
mpicc -std=c99 -g -O2 -fopenmp -o jacobi jacobi.o main.o -lm
```

that would first compile the two C source files and then link everything to form the final executable needs to be modified by prepending `scorep` to each of the three commands, i.e. we now have to write:

```
scorep mpicc -std=c99 -g -O2 -fopenmp -c jacobi.c
scorep mpicc -std=c99 -g -O2 -fopenmp -c main.c
scorep mpicc -std=c99 -g -O2 -fopenmp -o jacobi jacobi.o \
            main.o -lm
```

This call sequence will create a number of auxiliary C source files containing the original source code and a number of commands introduced by the measurement system in order to enable the latter to create the required measurements when the code is actually run. These modified source files are then compiled and linked, thus producing the desired executable named `jacobi`.

The actual measurement process is then initiated, e.g., by the call:

```
mpiexec -n 2 ./jacobi
```

The output data of this process will be stored in a newly created experiment directory `scorep-YYYYMMDD_↩ HHMM_XXXXXXXX` whose name is built up from the date and time when the measurement was started and an identification number.

As we had not explicitly set any Score-P related environment variables, the profiling mode was active by default. We obtain a file called `profile.cubex` containing profiling data in the experiment directory as the result of the measurement run. This file can be visually analyzed with the help of CUBE.

If we had set the variable `SCOREP_ENABLE_TRACING` to `true`, we would additionally have obtained trace data, namely the so called anchor file `traces.otf2` and the global definitions `traces.def` as well as a subdirectory `traces` that contains the actual trace data. This trace data is written in Open Trace Format 2 (OTF2) format. OTF2 is the joint successor of the classical formats OTF (used, e. g., by Vampir) and Epilog (used by Scalasca). A tool like Vampir can then be used to give a visual representation of the information contained in these files.

Chapter 3

Application Instrumentation

Score-P provides several possibilities to instrument user application code. Besides the automatic compiler-based instrumentation (Section 'Automatic Compiler Instrumentation'), it provides manual instrumentation using the Score-P User API (Section 'Manual Region Instrumentation'), semi-automatic instrumentation using POMP2 directives (Section 'Semi-Automatic Instrumentation of POMP2 User Regions') and, if configured, automatic source-code instrumentation using the PDToolkit-based instrumenter (Section 'Source-Code Instrumentation Using PDT').

As well as user routines and specified source regions, Score-P currently supports the following kinds of events:

MPI library calls: Instrumentation is accomplished using the standard MPI profiling interface PMPI. To enable it, the application program has to be linked against the Score-P MPI (or hybrid) measurement library plus MPI-specific libraries. Note that the Score-P libraries must be linked *before* the MPI library to ensure interposition will be effective.

SHMEM library calls: Instrumentation is accomplished using the SHMEM profiling interface or the GNU linker for library wrapping. To enable it, the application program has to be linked against the Score-P SHMEM (or hybrid) measurement library plus SHMEM-specific libraries. Note that the Score-P libraries must be linked *before* the SHMEM library to ensure interposition will be effective.

OpenMP directives & API calls: The Score-P measurement system uses the OPARI2 tool for instrumentation of OpenMP constructs. See the OPARI2 documentation on how to instrument OpenMP source code. In addition, the application must be linked with the Score-P OpenMP (or hybrid) measurement library.

Pthread library calls: The Score-P measurement system uses GNU linker for instrumentation of Pthreads library calls. At the moment only a few library calls are supported.

I/O library calls: The Score-P measurement system supports instrumentation of calls to POSIX I/O and MPI I/O routines.

The Score-P instrumenter command `scorep` automatically takes care of compilation and linking to produce an instrumented executable. For manual builds or build-systems based on plain Makefiles it is usually sufficient to prefix the compiler or compiler definitions like `CC` or `MPICC` (and equivalents) with the `scorep` instrumenter command, see below.

More complex build-systems that consist of a configuration-step and a make-step, like **autotools** and **CMake**, need to be handled differently. We need to take care that `scorep` is invoked during the make-step only to not confuse the configuration-step by additional Score-P output. To address this issue we provide so called Score-P compiler wrappers. Section 'Score-P Compiler Wrapper Usage' explains their usage in detail and provides CMake and autotools examples.

Usually the Score-P instrumenter `scorep` is able to automatically detect the programming paradigm from the set of compile and link options given to the compiler. In some cases however, when the compiler or compiler wrapper

enables specific programming paradigm by default (e.g., Pthreads on Cray and Blue Gene/Q systems) or using non-standard ways of activating features (e.g., providing the OpenMP runtime via `-lgomp` instead of `-fopenmp` as seen with CMake), `scorep` needs to be made aware of the programming paradigm in order to do the correct instrumentation. Please see `scorep --help` for the available options.

When using Makefiles, it is often convenient to define a "preparation preposition" placeholder (e.g., `PREP`) which can be prefixed to (selected) compile and link commands:

```
MPICC  = $(PREP) mpicc
MPICXX = $(PREP) mpicxx
MPIF90 = $(PREP) mpif90
```

These can make it easier to prepare an instrumented version of the program with

```
make PREP="scorep"
```

while default builds (without specifying `PREP` on the command line) remain fully optimized and without instrumentation.

In order to instrument applications which employ GNU Autotools for building, following instrumentation procedure has to be used:

1. Configure application as usual, but provide additional argument:

   ```
   --disable-dependency-tracking
   ```

2. Build application using `make` command with compiler specification variables set as follows:

   ```
   make CC="scorep <your-cc-compiler>"  \\
        CXX="scorep <your-cxx-compiler>" \\
        FC="scorep <your-fc-compiler>" ...
   ```

When compiling without the Score-P instrumenter, the `scorep-config` command can be used to simplify determining the appropriate linker flags and libraries, or include paths:

```
scorep-config [--mpp=none|--mpp=mpi|--mpp=shmem] \
[--thread=none|--thread=omp|--thread=pthread] --libs
```

The `--mpp=<paradigm>` switch selects which message passing paradigm is used. Currently, Score-↩
P supports applications using MPI (`--mpp=mpi`) or SHMEM (`--mpp=shmem`) and applications without any message passing paradigm. It is not possible to specify two message passing systems for the same application. The `--thread=<paradigm>` switch selects which threading system is used in Score-P. You may use OpenMP (`--thread=omp`), no threading system (`--thread=none`) or POSIX threading system (`--thread=pthread`). It is not possible to specify two threading systems for the same application. However, you may combine a message passing system with a threading system.

A particular installation of Score-P may not offer all measurement configurations!

The `scorep-config` command can also be used to determine the right compiler flags for specifying the include directory of the scorep/SCOREP_User.h or `scorep/SCOREP_User.inc` header files. When compiling without using the Score-P instrumenter, necessary defines and compiler instrumentation flags can be obtained by calling one of the following, depending on the language:

```
scorep-config --cflags [<options>]
scorep-config --cxxflags [<options>]
scorep-config --fflags [<options>]
```

If you compile a C file, you should use `--cflags`. If you use a C++ program, you should use `--cxxflags`. And if you compile a Fortran source file, you should use `--flags`.

With the additional options it is possible to select the used adapter, the threading system and the message passing system. For each adapter, we provides a pair of flags of the form `--adapter`, and `--noadapter` (please replace `adapter` by the name of the adapter). This allows to get options for non-default instrumentation possibilities. E.g., `--user` enables the manual instrumentation with the Score-P user API, the `--nocompiler` option disables compiler instrumentation.

Note

Disabling OpenMP measurements with the `--noopenmp` flag, disables all except parallel regions. Internally Score-P needs to track events on a per-thread basis and thus needs to be aware of the creation and destruction of OpenMP threads. Accordingly these regions will also show up in the measurements.

Score-P supports a variety of instrumentation types for user-level source routines and arbitrary regions, in addition to fully-automatic MPI and OpenMP instrumentation, as summarized in Table Score-P instrumenter option overview.

When the instrumenter determines that MPI or OpenMP are being used, it automatically enables MPI library instrumentation or OPARI2-based OpenMP instrumentation, respectively. The default set of instrumented MPI library functions is specified when Score-P is installed. All OpenMP parallel constructs and API calls are instrumented by default.

Note

To fine-tune instrumentation of OpenMP regions, use the `--opari=<parameter-list>` option. For available parameters please refer to the OPARI2 manual.

Note

Since Score-P version 1.3 there were two variants of internal OpenMP data handling, namely `--thread=omp:pomp_tpd` and `--thread=omp:ancestry`, depending on the functionality available on the target system. From Score-P version 4 on, due to internal refactorings, we replace the two OpenMP threading variants by only one: `--thread=omp`. The possible options are detected at configure time. If both are available, the ancestry mechanism will be used by default.

By default, automatic instrumentation of user-level source routines by the compiler is enabled (equivalent to specifying `--compiler`). The compiler instrumentation can be disabled with `--nocompiler` when desired, such as when using PDToolkit, or POMP2 or Score-P user API manual source annotations, are enabled with `--pdt`, `--pomp` and `--user`, respectively. Compiler, PDToolkit, POMP2 and Score-P user API instrumentation can all be used simultaneously, or in arbitrary combinations, however, it is generally desirable to avoid instrumentation duplication (which would result if all are used to instrument the same routines). Note that enabling PDToolkit instrumentation automatically enables Score-P user instrumentation, because it inserts Score-P user macros into the source code.

Sometimes it is desirable to explicitly direct the Score-P instrumenter to do nothing except execute the associated compile/link command. For such cases it is possible to disable default instrumentation with `--nocompiler`, `--thread=none`, and/or `--mpp=none`. Although no instrumentation is performed, this can help verify that the Score-P instrumenter correctly handles the compile/link commands.

Note

Disabling OpenMP in the instrumenter for OpenMP applications will cause errors during program execution if any event occurs inside of a parallel region.

3.1 Automatic Compiler Instrumentation

Most current compilers support automatic insertion of instrumentation calls at routine entry and exit(s), and Score-P can use this capability to determine which routines are included in an instrumented measurement.

Compiler instrumentation of all routines in the specified source file(s) is enabled by default by Score-P, or can be explicitly requested with `--compiler`. Compiler instrumentation is disabled with `--nocompiler`.

Note

> Depending on the compiler, and how it performs instrumentation, insertion of instrumentation may disable in-lining and other significant optimizations, or in-lined routines may not be instrumented at all (and therefore "invisible").

Automatic compiler-based instrumentation has been tested with a number of different compilers:

- GCC (UNIX-like operating systems, not tested with Windows)

- Clang/LLVM (UNIX-like operating systems, not tested with Windows)

- IBM xlc, xlC (version 7 or later, IBM Blue Gene)

- IBM xlf (version 9.1 or later, IBM Blue Gene)

- PGI (on Linux)

- Intel compilers (version 10 or later, Linux)

In all cases, Score-P supports automatic instrumentation of C, C++ and, Fortran codes, except for the Clang compiler which only provides support for instrumentation of C/C++ codes.

Note

> The automatic compiler instrumentation might create a significant relative measurement overhead on short function calls. This can impact the overall application performance during measurement. C++ applications are especially prone to suffer from this, depending on application design and whether C++ STL functions are also instrumented by the compiler. Currently, it is not possible to prevent the instrumentation of specific functions on all platforms when using automatic compiler instrumentation.

Note

> As an exception, instrumentation via the GCC plug-in or Intel compilers support all filtering features when using the Score-P option `--instrument-filter=<file>`. This option may require an absolute file path. Note that the GCC plug-in instrumentation by default does not instrument functions declared as `inline`. By providing a filter that matches `inline` functions in an explicit `INCLUDE` rule, the default behavior can be overridden. This `INCLUDE` is not allowed to be the *match-all* rule `INCLUDE *` (see Sec. 'Filtering').

Note

> CMake marks headers from imported targets as *system headers*, which consequently marks functions contained in these headers uninstrumentable by the GCC compiler instrumentation. The CMake option `-DCMAKE_NO_SYSTEM_FROM_IMPORTED` can be used to circumvent this marking, thus ensuring full instrumentation coverage of these targets, if this is desired (see previous note).

Names provided for instrumented routines depend on the compiler, which may add underscores and other decorations to Fortran and C++ routine names, and whether name "demangling" has been enabled when Score-P was installed and could be applied successfully.

3.2 Manual Region Instrumentation

In addition to the automatic compiler-based instrumentation (see Section 'Automatic Compiler Instrumentation'), instrumentation can be done manually. Manual instrumentation can also be used to augment automatic instrumentation with region or phase annotations, which can improve the structure of analysis reports. Furthermore, it offers the possibility to record additional, user defined metrics. Generally, the main program routine should be instrumented, so that the entire execution is measured and included in the analysis.

Instrumentation can be performed in the following ways, depending on the programming language used.

Fortran:

```
#include "scorep/SCOREP_User.inc"

subroutine foo
  SCOREP_USER_REGION_DEFINE( my_region_handle )
  ! more declarations

  SCOREP_USER_REGION_BEGIN( my_region_handle, "foo",
      SCOREP_USER_REGION_TYPE_COMMON )
  ! do something
  SCOREP_USER_REGION_END( my_region_handle )

end subroutine foo
```

C/C++:

```
#include <scorep/SCOREP_User.h>

void foo()
{
  SCOREP_USER_REGION_DEFINE( my_region_handle )

  // more declarations

  SCOREP_USER_REGION_BEGIN( my_region_handle, "foo",
      SCOREP_USER_REGION_TYPE_COMMON )

  // do something

  SCOREP_USER_REGION_END( my_region_handle )
}
```

C++ only:

```
#include <scorep/SCOREP_User.h>

void foo()
{
  SCOREP_USER_REGION( "foo", SCOREP_USER_REGION_TYPE_FUNCTION
      )

  // do something
}
```

Note

> When using Fortran, make sure the C preprocessor expands the macros. In most cases, the fortran compiler invoke the C preprocessor if the source file suffix is in capital letters. However, some compilers provide extra flags to tell the compiler to use a C preprocessor. Furthermore, it is important to use the C-like `#include` with the leading '#'-character to include the `SCOREP_User.inc` header file. Otherwise, the inclusion may happen after the C preprocessor ran. As result the fortran compiler complains about unknown preprocessing directives.

Region handles (`my_region_handle`) should be registered in each annotated function/subroutine prologue before use within the associated body, and should not already be declared in the same program scope.

For every region, the region type can be indicated via the region type flag. Possible region types are:

SCOREP_USER_REGION_TYPE_COMMON Indicates regions without a special region type.

SCOREP_USER_REGION_TYPE_FUNCTION Indicates that the region is a function or subroutine

SCOREP_USER_REGION_TYPE_LOOP Indicates that the region is the body of a loop, with the same number of iterations in all locations.

SCOREP_USER_REGION_TYPE_DYNAMIC Set this type to create a separate branch in the call-tree for every execution of the region. See Section 'Dynamic Region Profiling'.

SCOREP_USER_REGION_TYPE_PHASE Indicates that this region belongs to a special phase. See Section 'Phase Profiling'.

To create a region of combined region types you can connect two or more types with the binary OR-operator, e.g.:

```
SCOREP_USER_REGION_BEGIN( handle, "foo",
                          SCOREP_USER_REGION_TYPE_LOOP |
                          SCOREP_USER_REGION_TYPE_PHASE |
                          SCOREP_USER_REGION_TYPE_DYNAMIC )
```

For function instrumentation in C and C++, Score-P provides macros, which automatically pass the name and function type to Score-P measurement system. The `SCOREP_USER_FUNC_BEGIN` macro contains a variable definition. Thus, compilers that require strict separation of declaration and execution part, may not work with this macro.

C/C++:

```
#include <scorep/SCOREP_User.h>

void foo()
{
    SCOREP_USER_FUNC_BEGIN()
    // do something
    SCOREP_USER_FUNC_END()
}
```

In some cases, it might be useful to have the possibility to define region handles with a global scope. In C/C++, a region handle can be defined at a global scope with `SCOREP_USER_GLOBAL_REGION_DEFINE`. In this case, the `SCOREP_USER_REGION_DEFINE` must be omitted. The `SCOREP_USER_GLOBAL_REGION_DEFINE` must only appear in one file. To use the same global variable in other files, too, declare the global region in other files with `SCOREP_USER_GLOBAL_REGION_EXTERNAL`.

File 1:

```
SCOREP_USER_GLOBAL_REGION_DEFINE( global_handle )

foo()
{
    SCOREP_USER_REGION_BEGIN( global_handle, "phase 1",
                              SCOREP_USER_REGION_TYPE_PHASE)
    // do something
    SCOREP_USER_REGION_END( global_handle )
}
```

File 2:

```
SCOREP_USER_GLOBAL_REGION_EXTERNAL( global_handle )

bar()
{
    SCOREP_USER_REGION_BEGIN( global_handle, "phase 1",
                              SCOREP_USER_REGION_TYPE_PHASE)
    // do something
    SCOREP_USER_REGION_END( global_handle )
}
```

3.2 Manual Region Instrumentation

Note

These macros are not available in Fortran.

In addition, the macros `SCOREP_USER_REGION_BY_NAME_BEGIN( name, type )` and `SCOREP_US`↩
`ER_REGION_BY_NAME_END( name )` are available. These macros might introduce more overhead than the standard macros but can annotate user regions without the need to take care about the handle struct. This might be useful for automatically generating instrumented code or to avoid global declaration of this variable.

C/C++:

```
#include <scorep/SCOREP_User.h>

/* Application functions are already instrumented with these two calls. */
void instrument_begin(const char* regionname)
{
  /* code added for Score-P instrumentation */
  SCOREP_USER_REGION_BY_NAME_BEGIN( regionname, SCOREP_USER_REGION_TYPE_COMMON
      )
}

void instrument_end(const char* regionname)
{
  SCOREP_USER_REGION_BY_NAME_END( regionname )
}
```

Fortran:

```
#include "scorep/SCOREP_User.inc"

subroutine instrument_begin(regionname)
  character(len=*) :: regionname
  SCOREP_USER_REGION_BY_NAME_BEGIN( regionname, SCOREP_USER_REGION_TYPE_COMMON
      )
end subroutine instrument_begin

subroutine instrument_end(regionname)
  character(len=*) :: regionname
  SCOREP_USER_REGION_BY_NAME_END( regionname )
end subroutine instrument_end
```

Note

When using the "BY_NAME" macros in Fortran, be aware of section 12.4.1.1 of the F90/95/2003 standard. If you pass *name* through a dummy argument of a subroutine the length *len* of the character array *name* must be exactly the size of the actual string passed. In the Fortran examples above this is assured by `len=*`.

Note

To ensure correct nesting, avoid automatic compiler instrumentation for these helper functions.

The source files instrumented with Score-P user macros have to be compiled with `-DSCOREP_USER_ENAB`↩
`LE` otherwise `SCOREP_*` calls expand to nothing and are ignored. If the Score-P instrumenter `--user` flag is used, the `SCOREP_USER_ENABLE` symbol will be defined automatically. Also note, that Fortran source files instrumented this way have to be preprocessed with the C preprocessor (CPP).

Manual routine instrumentation in combination with automatic source-code instrumentation by the compiler or PDT leads to double instrumentation of user routines, i.e., usually only user region instrumentation is desired in this case.

3.3 Instrumentation for Parameter-Based Profiling

The Score-P user API provides also macros for parameter-based profiling. In parameter-based profiling, the parameters of a function are used to split up the call-path for executions of different parameter values. In Score-P parameter-based profiling is supported for integer and string parameters. To associate a parameter value to a region entry, insert a call to SCOREP_USER_PARAMETER_INT64 for signed integer parameters, SCOREP_↩ USER_PARAMETER_UINT64 for unsigned integer parameters, or SCOREP_USER_PARAMETER_STRING for string parameters after the region entry (e.g., after SCOREP_USER_REGION_BEGIN or SCOREP_USER_FU↩ NC_BEGIN).

Fortran:

```fortran
#include "scorep/SCOREP_User.inc"

subroutine foo(i, s)
    integer :: i
    character (*) :: s

    SCOREP_USER_REGION_DEFINE( my_region_handle )
    SCOREP_USER_PARAMETER_DEFINE( int_param )
    SCOREP_USER_PARAMETER_DEFINE( string_param )
    SCOREP_USER_REGION_BEGIN( my_region_handle, "my_region",
      SCOREP_USER_REGION_TYPE_COMMON )
    SCOREP_USER_PARAMETER_INT64(int_param, "myint",i)
    SCOREP_USER_PARAMETER_UINT64(uint_param, "myuint",i)
    SCOREP_USER_PARAMETER_STRING(string_param, "mystring",s)

    // do something

    SCOREP_USER_REGION_END( my_region_handle )
 end subroutine foo
```

C/C++:

```c
#include <scorep/SCOREP_User.h>

void foo(int64_t myint, uint64_t myuint, char *mystring)
{
  SCOREP_USER_REGION_DEFINE( my_region_handle )
  SCOREP_USER_REGION_BEGIN( my_region_handle, "foo",
      SCOREP_USER_REGION_TYPE_COMMON )
  SCOREP_USER_PARAMETER_INT64("myint",myint)
  SCOREP_USER_PARAMETER_UINT64("myuint",myuint)
  SCOREP_USER_PARAMETER_STRING("mystring",mystring)

  // do something

  SCOREP_USER_REGION_END( my_region_handle )
}
```

In C/C++, only a name for the parameter and the value needs to be provided. In Fortran, the handle must be defined first with SCOREP_USER_PARAMETER_DEFINE. The defined handle name must be unique in the current scope. The macro SCOREP_USER_PARAMETER_INT64 as well as the macro SCOREP_USER_PARAMETER_STR↩ ING need the handle as the first argument, followed by the name and the value.

Note

> If a region that has parameters triggers a user metric, the metric must be triggered after the parameters are provided.

Note

> The order of the parameters is important. If the region will have different orders for the same parameters, than only one will survive in the measurement data. Also in Cube, parameters are grouped into numeric and string types, thus the order will may be different than the instrumentation.

3.4   Measurement Control Instrumentation

The Score-P user API also provides several macros for measurement control that can be incorporated in source files and activated during instrumentation. The macro SCOREP_RECORDING_OFF can be used to (temporarily) pause recording until a subsequent SCOREP_RECORDING_ON. Just like the already covered user-defined annotated regions, SCOREP_RECORDING_ON and the corresponding SCOREP_RECORDING_OFF must be correctly nested with other enter/exit events. Please beware that if program start is recorded, i.e., main or MAIN_ are instrumented and not filtered, the recording needs to be switched on before program end in order to get valid measurements. Finally, with SCOREP_RECORDING_IS_ON you can test whether recording is switched on.

Events are not recorded when recording is switched off (though associated definitions are), resulting in smaller measurement overhead. In particular, traces can be much smaller and can target specific application phases (e.g., excluding initialization and/or finalization) or specific iterations. Since the recording switch is process-local, and effects all threads on the process, it can only be initiated outside of OpenMP parallel regions. Switching recording on/off is done independently on each MPI process without synchronization.

Note

> Switching recording on/off may result in inconsistent traces or profiles, if not applied with care. In particular, if communication is recorded incomplete (e.g., if the send is missing but the corresponding receive event is recorded) it may result in errors during execution or analysis. Furthermore, it is not possible to switch recording on/off from within parallel OpenMP regions. We recommend to use the selective recording interface, instead of the manual on/off switch whenever possible. Special care is required in combination with selective recording (see Section 'Selective Recording', which also switches recording on/off.

3.5   Semi-Automatic Instrumentation of POMP2 User Regions

Note

> Since Score-P version 1.4, OpenMP instrumentation using OPARI2 no longer activates POMP2 instrumentation implicitly. You need to explicitly add the --pomp option to the Score-P instrumenter.

If you manually instrument the desired user functions and regions of your application source files using the POMP2 INST directives described below, the Score-P instrumenter --pomp flag will generate instrumentation for them. POMP2 instrumentation directives are supported for Fortran and C/C++. The main advantages are that

- being directives, the instrumentation is ignored during "normal" compilation and

- this semi-automatic instrumentation procedure can be used when fully automatic compiler instrumentation is not supported.

The INST BEGIN/END directives can be used to mark any user-defined sequence of statements. If this block has several exit points (as is often the case for functions), all but the last have to be instrumented by INST ALTEND.

Fortran:

```
subroutine foo(...)
    !declarations
    !POMP$ INST BEGIN(foo)
    ...
    if (<condition>) then
        !POMP$ INST ALTEND(foo)
        return
    end if
    ...
    !POMP$ INST END(foo)
end subroutine foo
```

C/C++:

```
void foo(...)
{
    /* declarations */
    #pragma pomp inst begin(foo)
    ...
    if (<condition>)
    {
        #pragma pomp inst altend(foo)
        return;
    }
    ...
    #pragma pomp inst end(foo)
}
```

At least the main program function has to be instrumented in this way, and additionally, one of the following should be inserted as the first executable statement of the main program:

Fortran:

```
program main
    ! declarations
    !POMP$ INST INIT
    ...
end program main
```

C/C++:

```
int main(int argc, char** argv)
{
    /* declarations */
    #pragma pomp inst init
    ...
}
```

By default, the source code is preprocessed before POMP2 instrumentation happens. For more information on the preprocessing, see Section 'Preprocessing before POMP2 and OpenMP instrumentation'.

3.6 Preprocessing before POMP2 and OpenMP instrumentation

By default, source files are preprocessed before the semi-automatic POMP2 instrumentation or the OpenMP construct instrumentation with OPARI2 happens. This ensures, that all constructs and regions that might be contained in header files, templates, or macros are properly instrumented. Furthermore, conditional compilation directives take effect, too. The necessary steps are performed by the Score-P instrumenter tool.

Some Fortran compilers do not regard information about the original source location that the preprocessing leaves in the preprocessed code. This causes wrong source code information for regions from compiler instrumentation, and manual source code instrumentation. However, these compilers also disregard the source code information left by OPARI2. Thus, for these compilers the source location information is incorrect anyway.

If the preprocessing is not desired, you can disable it with the `--nopreprocess` flag. In this case the instrumentation is performed before the preprocessing happens. In this case constructs and regions in header files, macros, or templates are not instrumented. Conditional compilation directives around constructs may also lead to broken instrumentation.

Note

> If a parallel region is not instrumented, the application will crash during runtime.

The preprocessing does not work in combination with PDT source code instrumentation. Thus, if PDT instrumentation is enabled, it changes the default to not preprocess a source file. If you manually specify preprocessing and PDT source code instrumentation, the instrumenter will abort with an error.

3.7   Source-Code Instrumentation Using PDT

If Score-P has been configured with PDToolkit support, automatic source-code instrumentation can be used as an alternative instrumentation method. In this case, the source code of the target application is pre-processed before compilation, and appropriate Score-P user API calls will be inserted automatically. However, please note that this feature is still somewhat experimental and has a number of limitations (see Section 'Limitations').

To enable PDT-based source-code instrumentation, call `scorep` with the `--pdt` option, e.g.,

```
scorep --pdt  mpicc -c foo.c
```

This will by default instrument all routines found in `foo.c`. (To avoid double instrumentation, automatic compiler instrumentation is disabled when using Source-Code Instrumentation with PDT. However, if you you can enforce additional compiler instrumentation with `--compiler`.) The underlying PDT instrumentor supports a set a instrumentation options, which can be set like

```
scorep --pdt="-f <inclusion/exclusion file>"  mpicc -c foo.c
```

This particular option for example can be used to manually include/exclude specific functions from the instrumentation process. The respective file format is described here. Please check the documentation about the `tau_↩instrumentor` for more valid options.

3.7.1   Limitations

Currently the support for the PDT-based source-code instrumenter still has a number of limitations:

- When instrumenting Fortran 77 applications, the inserted instrumentation code snippets do not yet adhere to the Fortran 77 line length limit. Typically, it is possible to work around this issue by supplying extra command line flags (e.g., `-ffixed-line-length-132` or `-qfixed=132`) to the compiler.

- Code in C/C++ header files as well as included code in Fortran (either using the C preprocessor or the `include` keyword) will currently not be instrumented.

- Support for C++ templates and classes is currently only partially implemented.

- Advanced TAU instrumentation features such as static/dynamic timers, loop, I/O and memory instrumentation are not yet supported. Respective entries in the selective instrumentation file will be ignored.

3.8   User Library Wrapping

User library wrapping enables you to install library wrappers for any C/C++ library you want.

Without this mechanism, in order to intercept calls to a library, you would need to either build this library with Score-P or add manual instrumentation to the application using the library. Another advantage of user library wrapping is you don't need access to the source code of the to-be-wrapped library. Headers and library files suffice.

This feature requires Score-P to be configured with libclang. You can find out whether user library wrapping is enabled via `scorep-info config-summary` in Section "Score-P (libwrap)".

This section covers how to use already installed wrappers. Appendix 'Score-P User Library Wrapping' explains how to create user library wrappers, and provides additional details.

To find out which user library wrappers are installed call

```
$ scorep-info libwrap-summary
```

It lists all found wrappers, either installed into Score-P's installation directory or found via the SCOREP_LIBWRA↩
P_PATH environment variable. Optionally you can run

```
$ scorep-info libwrap-summary <wrappername>
```

to show the configuration of a specific wrapper.

You can then use Score-P to link your application in the usual way and additionally provide --libwrap=[<wrapmode>↩
:]<wrappername> to enable library wrapping for the target library.

Example with only relinking the application:

```
$ scorep --libwrap=fftw3 gcc -o main main.o -L$FFTW_LIB -lfftw3
```

Example with both recompiling and linking the application:

```
$ scorep --libwrap=fftw3 --nocompiler gcc -o main -I$FFTW_INC main.c \
    -L$FFTW_LIB -lfftw3
```

3.8.1 Runtime vs Linktime Wrapping

There exist two ways to wrap calls to the to-be-wrapped (or target-) library. The main difference lies in when the actual wrapping takes place - at *linktime* or *runtime*. While they are in essence the same, they differ in which function calls can be intercepted. Specifically:

linktime: Wraps calls originating from object files that are part of the linker command line. In addition, calls originating from static libraries are wrapped as well. The actual technique used is the -wrap linker flag.

runtime: Wraps all calls that linktime wrapping would, plus those which originate from already linked shared objects. The actual technique used is replacing the original function of the target library and using dlopen and dlsym in the wrapper to open the target library plus finding and calling the original function.

You can choose linktime or runtime wrapping manually via the --libwrap flag by prefixing the wrapper name with either linktime: or runtime:.

3.9   Recording of I/O activities

Score-P checks for availability of several I/O paradigm at configure time. Please have a look at the configure summary to get an overview of the detected I/O routines. You can provide multiple options to the scorep instrumenter to record I/O routines of multiple paradigms at a time. In the following example Score-P will instrument calls to POSIX I/O routines.

```
    scorep --io=posix mpicc foo.c -o foo
```

In addition calls to MPI I/O routines are automatically instrumented if the MPI programming paradigm is detected by the scorep instrumenter.

3.10 Enforce Linking of Static/Shared Score-P Libraries

If the Score-P was build with shared libraries and with static libraries, the instrumenter uses the compiler defaults for linking. E.g., if the compiler chooses shared libraries by default, the instrumenter will link your application with the shared Score-P libraries. Furthermore, the linking is affected by parameters in the original link command. E.g., if your link command contains a `-Bstatic` flag, afterwards appended Score-P libraries are also linked statically.

If you want to override the default and enforce linking of static or dynamic Score-P libraries, you can add the flag `--static` or `--dynamic` for the instrumenter. E.g., a command to enforce static linking can look like:

```
scorep --static mpicc foo.c -o foo
```

In this case, the linking against the static version of the Score-P libraries is enforced.

If enforcing static or dynamic linking is not possible on your system, e.g., because no static/dynamic Score-P libraries are installed, the instrumenter will abort with an error. You can determine whether `--static` or `--dynamic` is available from the output of `scorep --help`. If the `--static` or `--dynamic` flags are not shown, then they are not available.

Type of instrumentation	Instrumenter switch	Default value	Instrumented routines	Runtime measurement control
MPI	--mpp=mpi/ --mpp=none	(auto)	configured by install	'Selection of MPI Groups'
SHMEM	--mpp=shmem/ --mpp=none	(auto)	configured by install	—
CUDA (incl. NVTX)	--[no]cuda	enabled	all	'CUDA Performance Measurement' ('CUDA NVTX Measurement')
HIP	--[no]hip	enabled	all	'HIP Performance Measurement'
OpenCL	--[no]opencl	enabled	configured by install	'OpenCL Performance Measurement'
OpenACC	--[no]openacc	enabled	configured by install	'OpenACC Performance Measurement'
OpenMP	--thread=omp / --[no]openmp	(auto)	all parallel constructs, see Note below	—
Pthread	--thread=pthread	(auto)	Basic Pthread library calls	—
'Automatic Compiler Instrumentation'	--[no]compiler	enabled	all	'Filtering'
'Recording of I/O activities'	--[no]io[=...]	disabled	configured by install	'Recording of I/O activities'
'Source-Code Instrumentation Using PDT'	--[no]pdt	disabled	all	'Filtering'
'Semi-Automatic Instrumentation of POMP2 User Regions'	--[no]pomp	disabled	manually annotated	'Filtering'
'Manual Region Instrumentation'	--[no]user	disabled	manually annotated	'Filtering' and 'Selective Recording'
'Score-P User Library Wrapping'	--libwrap=[...]	disabled	all by library wrapper	'Filtering'

Table 3.1 Score-P instrumenter option overview

Chapter 4

Application Sampling

This document describes how to use the sampling options within Score-P.

4.1 Introduction

Score-P supports sampling that can be used concurrently to instrumentation to generate profiles and traces. In the following, we will describe how sampling differs from instrumentation. Reading this text will help you to interpret resulting performance data. However, if you are aware of how sampling works, you can skip the preface.

In our context, we understand sampling as a technique to capture the behavior and performance of programs. We interrupt the running programs at a specified interval (the sampling period) and capture the current state of the program (i.e., the current stack) and performance metrics (e.g., PAPI). The obtained data is than further stored as a trace or a profile and can be used to analyze the behavior of the sampled program.

Before version 2.0 of Score-P, only instrumentation-based performance analysis had been possible. Such an instrumentation relies on callbacks to the measurement environment (`instrumentation points`), e.g., a function enter or exit. The resulting trace or profile presented the exact runtimes of the functions, augmented with performance data and communication information. However, instrumentation introduces a constant overhead for each of the instrumentation points. For small instrumented functions, this constant overhead can be overwhelming.

Sampling provides the opportunity to prevent this overwhelming overhead, and even more, the overhead introduced by sampling is controllable by setting the sampling rate. However, the resulting performance data is more "fuzzy". Not every function call is captured and thus the resulting data should be analyzed carefully. Based on the duration of a function and the sampling period, a function call might or might not be included in the gathered performance data. However, statistically, the profile information is correct. Additionally, the sampling rate allows to regulate the trade-off between overhead and correctness, which is not possible for instrumentation.

In Score-P we support both instrumentation and sampling. This allows you for example to get a statistical overview of your program as well as analyzing the communication behavior. If a sample hits a function that is known to the measurement environment via instrumentation (e.g., by OPARI2), the sample will show the same function in the trace and the profile.

4.2 Prerequisites

This version of Score-P provides support for sampling. To enable sampling, several prerequisites have to be met.

- **libunwind**:
  Additionally to the usual configuration process of Score-P, `libunwind` is needed. `libunwind` can be installed using a standard package manager or by downloading the latest version from

  http://download.savannah.gnu.org/releases/libunwind/

  This library must be available at your system to enable sampling. In our tests, we used the most current stable version (1.1) as previous versions might result in segmentation faults.

- **Sampling Sources**:
  Sampling sources generate interrupts that trigger a sample. We interface three different interrupt generators, which can be chosen at runtime.

  1. **Interval timer**:
     Interval timers are POSIX compliant but provide a major drawback: They cannot be used for multi-threaded programs, but only for single-threaded ones. We check for `setitimer` that is provided by `sys/time.h`.

  2. **PAPI**:
     We interface the PAPI library, if it is found in the configure phase. The PAPI interrupt source uses overflowing performance counters to interrupt the program. This source can be used in multi-threaded programs. Due to limitations from the PAPI library, PAPI counters will not be available if PAPI sampling is enabled. However, you can use perf metrics, e.g.,

     ```
     export SCOREP_METRIC_PERF=instructions:page-faults
     ```

  3. **perf**:
     `perf` is comparable to PAPI but much more low-level. We directly use the system call. This source can be used in multi-threaded programs. PAPI counters are available if `perf` is used as an interrupt source. Currently we only provide a cycle based overflow counter via `perf`.

We recommend using `PAPI` or `perf` as interrupt sources. However, these also pose a specific disadvantage when power saving techniques such as DVFS or idle states are active on a system. In this case, a constant sampling interval cannot be guaranteed. If, for example, an application calls a sleep routine, then the cycle counter might not increase as the CPU might switch to an idle state. This can also influence the result data. Such idling times can also be introduced by OpenMP runtimes and can be avoided by setting the block times accordingly or setting the environment variable `OMP_WAIT_POLICY` to `ACTIVE`.

4.3 Configure Options

4.3.1 libunwind

If libunwind is not installed in a standard directory, you can provide the following flags in the configure step:

```
--with-libunwind=(yes|no|<Path to libunwind installation>)
                      If you want to build scorep with libunwind but do
                      not have a libunwind in a standard location, you
                      need to explicitly specify the directory where it is
                      installed. On non-cross-compile systems we search
                      the system include and lib paths per default [yes];
                      on cross-compile systems, however, you have to
                      specify a path [no]. --with-libunwind is a shorthand
                      for --with-libunwind-include=<Path/include> and
                      --with-libunwind-lib=<Path/lib>. If these shorthand
                      assumptions are not correct, you can use the
                      explicit include and lib options directly.
--with-libunwind-include=<Path to libunwind headers>
--with-libunwind-lib=<Path to libunwind libraries>
```

4.4 Sampling Related Score-P Measurement Configuration Variables

The following lists the Score-P measurement configuration variables which are related to sampling. Please refer to the individual variables for a more detailed description.

- SCOREP_ENABLE_UNWINDING
- SCOREP_SAMPLING_EVENTS
- SCOREP_SAMPLING_SEP
- SCOREP_TRACING_CONVERT_CALLING_CONTEXT_EVENTS

4.5 Use Cases

4.5.1 Enable unwinding in instrumented programs

Additionally to the instrumentation, you now see where the instrumented region has been called. A pure MP←
I instrumentation for example does not tell you which functions have been issuing communications. With unwinding enabled, this is revealed and stored in the trace or profile.

Instrument your program, e.g., with MPI instrumentation enabled.

```
scorep mpicc my_mpi_code.c -o my_mpi_application
```

Set the following environment variables:

```
export SCOREP_ENABLE_UNWINDING=true
export SCOREP_SAMPLING_EVENTS=
```

Run your program

```
mpirun -np 16 ./my_mpi_application
```

4.5.2 Instrument a hybrid parallel program and enable sampling

In this example you get rid of a possible enormous compiler instrumentation overhead but you are still able to see statistical occurrences of small code regions. The NAS Parallel Benchmark BT-MZ for example uses small sub functions within OpenMP parallel functions that increase the measurement overhead significantly when compiler instrumentation is enabled.

Instrument your program, e.g., with MPI and OpenMP instrumentation enabled.

```
scorep mpicc -fopenmp my_hybrid_code.c -o my_hybrid_application
```

Note: If you use the GNU compiler and shared libraries of Score-P you might get errors due to undefined references depending on your gcc version. Please add `--no-as-needed` to your scorep command line. This flag will add a GNU ld linker flag to fix undefined references when using shared Score-P libraries. This happens on systems using `--as-needed` as linker default. It will be handled transparently in future releases of Score-P.

Set the following environment variables:

```
export SCOREP_ENABLE_UNWINDING=true
```

If you want to use a sampling event and period differing from the default settings you additionally set:

```
export SCOREP_SAMPLING_EVENTS=PAPI_TOT_CYC@1000000
```

Run your program

```
mpirun -np 16 ./my_mpi_application
```

4.6   Test Environment

Example

4.6.1   Instrument NAS BT-MZ code

```
cd <NAS_BT_MZ_SRC_DIR>
vim config/make.def
```

Set add the Score-P wrapper to your MPI Fortran compiler.

```
MPIF77 = scorep mpif77
```

Recompile the NAS BT-MZ code.

```
make clean
make bt-mz  CLASS=C  NPROCS=128
```

4.6.2   Run instrumented binary

```
cd bin
sbatch run.slurm
```

Batch script example:

```
#!/bin/bash
#SBATCH -J NAS_BT_C_128x2
#SBATCH --nodes=32
#SBATCH --tasks-per-node=4
#SBATCH --cpus-per-task=2
#SBATCH --time=00:30:00

export OMP_NUM_THREADS=2

export NPB_MZ_BLOAD=0

export SCOREP_ENABLE_TRACING=true
export SCOREP_ENABLE_PROFILING=false
export SCOREP_ENABLE_UNWINDING=true
export SCOREP_TOTAL_MEMORY=200M
export SCOREP_SAMPLING_EVENTS=perf_cycles@2000000
export SCOREP_EXPERIMENT_DIRECTORY='bt-mz_C.128x2_trace_unwinding'

srun ./bt-mz_C.128
```

Chapter 5

Application Measurement

If an application was instrumented with Score-P, you will get an executable, which you can execute like the uninstrumented application. After the application run, you will find an experiment directory as child of the current working directory, which contains all recorded data (note that there will be no directory if no substrate requested one). The experiment directory has the format `scorep-YYYYMMDD_HHMM_XXXXXXXX`, where `YYYYMMDD` and `H↩HMM` encodes the date followed by a series of random numbers. You may specify the name of the experiment directory by setting the environment variable `SCOREP_EXPERIMENT_DIRECTORY` to the desired name of the directory. Note that this directory will also be created directly under the current working directory. No parent directories will be created. If the directory already exists, the existing directory will be renamed by appending a date like above by default. You can let Score-P abort the measurement immediately by setting `SCOREP_OVERWRI↩TE_EXPERIMENT_DIRECTORY` to `false` if the experiment directory already exists. This has only an effect if `SCOREP_EXPERIMENT_DIRECTORY` was set too.

In general, you can record a profile and/or a event trace. Whether a profile and/or a trace is recorded, is specified by the environment variables `SCOREP_ENABLE_PROFILING` and `SCOREP_ENABLE_TRACING`. If the value of this variables is zero or `false`, profiling/tracing is disabled. Otherwise Score-P will record a profile and/or trace. By default, profiling is enabled and tracing is disabled.

You may start with a profiling run, because of its lower space requirements. According to profiling results, you may configure the trace buffer limits, filtering or selective recording for recording traces.

Score-P allows to configure several parameters via environment variables. See Appendix 'Score-P Measurement Configuration' for a detailed description of how to configure the measurement.

5.1 Profiling

Score-P implements a call-tree based profiling system. Every node in the call tree represent a recorded region. The edges of the tree represent the caller-callee relationship: The children of a node are those regions, that are entered/exited within a region. The path from the root to an arbitrary node, represents a call-path. Thus, every node in the tree identifies also the call-path from the root to itself.

Together with a node, the statistics for the call-path are stored. By default, the runtime and the number of visits are recorded. Additionally, hardware counters can be configured and are stored for every call-path. User defined metrics are only stored in those nodes, where the metric was triggered.

For enabling profiling, set the `SCOREP_ENABLE_PROFILING` environment variable to 1 or `true`. After the execution of your application you will then find a file, named `profile.cubex` in your measurement directory, which you can display with the CUBE4 with `cube profile.cubex`. The name of the profile can be changed

through the environment variable `SCOREP_PROFILING_BASE_NAME`. The extension `.cubex` will be appended to the base name you specify in the environment variable `SCOREP_PROFILING_BASE_NAME`.

By default, Score-P writes the profile in CUBE4 base format. Hereby, for every metric contains one value, usually only the sum. However, Score-P allows to store the profile in other formats. To change the default format, set the environment variable `SCOREP_PROFILING_FORMAT`. Please refer to the description of this variable for possible values.

Score-P records a call tree profile. The maximum call-path depth that is recorded is limited to 30, by default. This avoids extremely large profiles for recursive calls. However, this limit can be changed with the environment variable `SCOREP_PROFILING_MAX_CALLPATH_DEPTH`.

5.1.1 Parameter-Based Profiling

Parameter-based profiling allows to separate the recoded statistics for a region, depending on the values of one or multiple parameters. In the resulting call-tree, each occurred parameter-value will create a sub-node of the region. Every parameter has a parameter name. Thus, if multiple parameters are used, they can be distinguished and split the call-tree in the order of the parameter events. In the final call-tree it looks like every parameter-name/parameter-value pair is a separate region.

Currently, the only source for parameter events is manual instrumentation (see Section 'Instrumentation for Parameter-Based Profiling').

5.1.2 Phase Profiling

Phase-profiling allows, to group the execution of the application into logical phases. Score-P records a separate call-tree for every phase in the application. A phase starts when a region of type `SCOREP_USER_REGION_T↩YPE_PHASE` (see Section 'Manual Region Instrumentation') is entered. If the region is exited, the phase is left. If two phases are nested, then the outer phase is left, when the inner phase is entered. If the inner phase is exited, the outer phase is re-entered. Figure 5.1 shows the difference in the call-tree if the regions with the names phase1 and phase2 are not of type `SCOREP_USER_REGION_TYPE_PHASE` on the left side and the forest if they are of type `SCOREP_USER_REGION_TYPE_PHASE` on the right side.

If the phase consists of multiple partitions, and thus cannot be enclosed by a single code region, all code-regions that form the phase must have the same region handle. The possibility to define global region handles in C/C++ might be useful for the definition of phases that are have multiple partitions (see Section 'Manual Region Instrumentation').

5.1.3 Dynamic Region Profiling

When profiling, multiple visits of a call-path are summarized. However, e.g, for investigations in time-dependent behavior of an application, each iteration of a main loop (or some other region) should create a separate profile sub-tree. For such cases, Score-P allows to define regions to by of type dynamic. For dynamic regions, each entry of the region will create a separate path. For this cause, the Score-P profiling system creates an extra parameter, named `instance`. On each visit to a dynamic region, the instance parameter for this call-path is increased and triggered automatically. Thus, the every visit to a dynamic region generates a separate subtree in the profile.

As an example, let us assume that an application contains the regions `foo` and `main`, where `main` calls `foo` three times. A regular profile would show two call-pathes:

- `main`

- `main/foo`
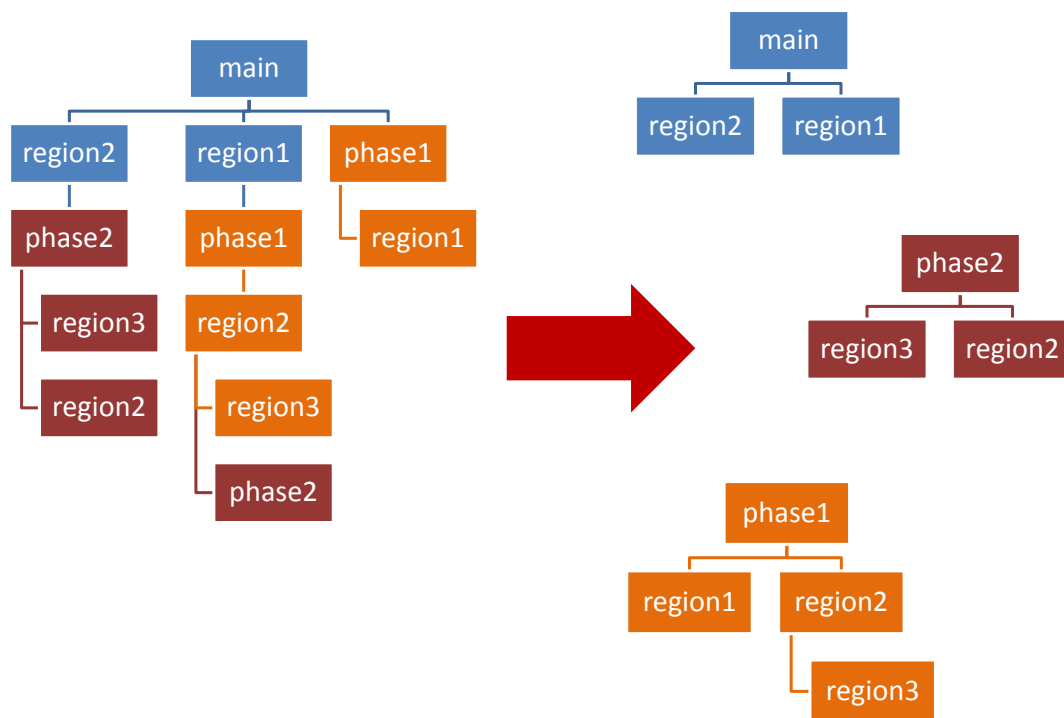
Figure 5.1 Call-tree changes when using phases. The left side shows the calltree if no region is of type phase. The right side shows the call-tree forest with phases.

If `foo` is a dynamic region, the profile would contain additional sub-nodes for each visit of `foo`. The resulting profile would contain the following call-paths:

- `main`

- `main/foo`

- `main/foo/instance=0`

- `main/foo/instance=1`

- `main/foo/instance=2`

In this case `main/foo` contains the summarized statistics for all 3 visits, while `main/foo/instance=0` contains the statistics for the first visits of the call-path.

Note

> The enumeration of the instance is per call-path and not per dynamic region. In particular, if a dynamic region foo appears in 2 call-paths, it has 2 instance number 0, one in both call-paths. It is not a global enumeration of the visits to foo but enumerates the visits of foo in a particular call-path from 0 to N.

Currently, the only possibility to define dynamic regions is via the manual region instrumentation, described in Section 'Manual Region Instrumentation'.

Note

> Using dynamic regions can easily create very large profiles. Thus, use this feature with care. If you are only interested in some parts of the application, selective recording (see Section 'Selective Recording') might be a memory space save alternative. Furthermore, you can use clustering (see Section 'Clustering') to reduce the memory requirements.

5.1.4 Clustering

Clustering allows to reduce the memory requirements of a dynamic region, by clustering similar sub-trees into one cluster. A visualization tool (like CUBE 4) might expand the clusters back to single iterations transparently. You can enable/disable clustering via the environment variable SCOREP_PROFILING_ENABLE_CLUSTERING. By default, clustering is enabled.

Currently, clustering is limited to the instances of one node in the call-tree. If a dynamic region appears on several call-paths, Score-P will only cluster one, and generate separate sub-trees for every iterations in all other call-paths. By default, Score-P will cluster the instances of that dynamic region call-path that it enters first. If you have only one call-path where a dynamic region occurs (e.g., if the body of the main loop is the only dynamic region), this region will be clustered automatically. Otherwise, we recommend to specify the region you want to cluster in the environment variable SCOREP_PROFILING_CLUSTERED_REGION.

Note

> If the selected region appears on multiple call-paths, only one of them is clustered. Score-P chooses the call-path of that regions that it enters first. In particular, if the selected dynamic region is nested into itself, the outermost occurrence is clustered.

Furthermore, the clustered region must not be inside of a parallel region, but must be at a sequential part of the program. However, the clustered region may contain parallel regions.

Clustering is a lossy compression mechanism. The accuracy increases if more clusters are available. On the downside, more clusters require more memory. You can specify the number of clusters you want by setting the environment variable SCOREP_PROFILING_CLUSTER_COUNT to the number of cluster you want to have. The default cluster number is 64.

Furthermore, you can enforce a minimal structural similarity of instances of a cluster. Clusters that fit the minimal structural similarity requirements belong to the same equivalence class. Only instances of the same equivalence class will be clustered together. If you have more equivalence classes than the number of clusters you specified in SCOREP_PROFILING_CLUSTER_COUNT, the maximal number of clusters is increased. Thus, you might get more clusters than you specified.

The minimal structural similarity is defined by the clustering mode which can be set via the environment variable SCOREP_PROFILING_CLUSTERING_MODE. Please refer to the description of this variable for possible values.

5.1.5 Enabling additional debug output on inconsistent profiles

If the Score-P profiling system detects inconsistencies during measurement, it stops recording the profile and prints an error message. Examples for reasons of an inconsistent profile are, if the nesting order of function entries and exits is broken, or events appear for an uninitialized thread. This might indicate an bug of the profile, but typically the cause is an erroneous instrumentation. E.g., if manual instrumentation is applied, but not all possible exit points of a function are instrumented.

In order to support debugging of manual instrumentation, or during the development of own automatic instrumentation techniques, the profile can write additional information about its current state in a textual form into a file. This output may contain the following information:

- The current call stack of the failing thread

- The profile structure of the failing thread

- The complete profile structure

None of the three entries is guaranteed to appear in the output, it depends on the current state of the profile. It might not be possible to provide any output at all. Furthermore, the online representation of the profile structure may differ from the final profile structure.

You can enable this additional output by setting the environment variable SCOREP_PROFILING_ENABLE_CO←RE_FILES to true. Then, if the profile detects an inconsistency, it will write a core file into your measurement directory. If an inconstant profiles is detected on multiple locations, every location where an inconsistency is detected will write a core file. Thus, it is not recommended, to enable this feature for large scale runs.

5.2   Tracing

Score-P can write events to OTF2 traces. By setting the environment variable `SCOREP_ENABLE_TRACING`, you can control whether a trace is recorded. If the value is `0` or `false` no trace is recorded, if the value is non-zero or `true`, a trace is recorded. If the variable is not specified, Score-P does not record traces. After trace recording you will find the OTF2 anchor file, named `traces.otf2` in the experiment directory, along with the trace data.

5.3   Filtering

When automatic compiler instrumentation or automated source code instrumentation with PDT has been used to instrument user-level source-program routines, there are cases where measurement and associated analysis are degraded, e.g., by frequently-executed, small and/or generally uninteresting functions, methods and subroutines.

A measurement filtering capability is therefore only supported for compiler instrumented regions, regions instrumented with the user API from Score-P (see section 'Manual Region Instrumentation'), regions instrumented with the user API from OPARI2 (see section 'Semi-Automatic Instrumentation of POMP2 User Regions'), CUDA device and host activities (see Section 'CUDA Performance Measurement'), HIP device and host activities (see Section 'HIP Performance Measurement'), and Kokkos activities (see Section 'Kokkos Performance Measurement'). See section 'Selection of MPI Groups' to restrict the recording of MPI features and the OPARI2 documentation of `--disable` to restrict instrumentation of OpenMP directives. This `--disable` flag can than be passed on to the OPARI2 invocation with the `--opari=<parameter-list>` flag of the Score-P instrumenter. Because PDT instrumentation (Section 'Source-Code Instrumentation Using PDT') inserts Score-P user API instrumentation those regions can be filtered, too. Regions can be filtered based on their region name (e.g., their function name) or based on the source file, in which they are defined.

A file that contains the filter definition can be specified via the environment variable `SCOREP_FILTERING_FILE`. If no filter definition file is specified, all instrumented regions are recorded. For filtered regions, the enter/exit events are not recorded in trace and profile.

The filter definition file can contain two blocks:

- One block defines filter rules for filtering regions based on the source files they are defined in.

- One filter block defined rules for region names.

When the filter rules are applied, the source file name filter is evaluated first. If a region is filtered because it appears in a filtered source file, it cannot be included by the function name filter. If a region was defined in a not-filtered source file, the region name filter is evaluated. This means, events for a region are not recorded if they are filtered by the source file filter or the region name filter. Events for a region are recorded if the region is neither filtered by the source file filter nor by the region name filter. If one of the both filter blocks is not specified, it is equivalent to an empty filter block.

Beside the two filter blocks, you may use comments in the filter definition file. Comments start with the character '#' and is terminated by a new line. You may use comments also inside the filter blocks. If a region name or source file name contains '#', you must escape it with a backslash.

5.3.1 Source File Name Filter Block

The filter block for source file names, must be enclosed by SCOREP_FILE_NAMES_BEGIN and SCOREP_FIL↩
E_NAMES_END. In between you can specify an arbitrary number of include and exclude rules which are evaluated
in sequential order. At the beginning all source files are included. Source files that are excluded after all rules are
evaluated, are filtered.

An exclude rule starts with the keyword EXCLUDE followed by one or multiple white-space separated source file
names. Respectively, include rules start with INCLUDE followed by one or multiple white-space separated file
names. For the specification of file names, bash-like wild-cards are supported. In particular, the '∗' wild-card
matches an string of arbitrary length, the '?' matches exactly one arbitrary character, or within [] you may specify
multiple options.

Note

> Unlike bash, a '∗' may match a string that contains slashes. E.g, you may use the '∗' wild-card for path prefixes.

An example source file filter block could look like this:

```
SCOREP_FILE_NAMES_BEGIN # This is a comment
  EXCLUDE */filtering/filter*
  INCLUDE */filter_test.c
SCOREP_FILE_NAMES_END
```

Note

> The keywords (SCOREP_FILE_NAMES_BEGIN, SCOREP_FILE_NAMES_END, EXCLUDE, and INCL↩
> UDE) are case-sensitive.

The filtering is based on the filenames as seen by the measurement system. Depending on instrumentation method
and compiler the actual filename may contain the absolute path, a relative path or no path at all. The instrumentation
tool tries to create as much absolute paths as possible. Paths are simplified before comparison to a rule. E.g., it
removes path/../, ./ and multiple slashes. You may look up the actual filename in the resulting output of the
measurement.

5.3.2 Region Name Filter Block

The filter block for the region names, must be enclosed by SCOREP_REGION_NAMES_BEGIN and SCOREP↩
_REGION_NAMES_END. In between you can specify an arbitrary number of include and exclude rules which are
evaluated in sequential order. At the beginning, all regions are included. Regions that are excluded after all rules
are evaluated, are filtered.

Note

> Regions that are defined in source files that are filtered, are excluded due to the source file filter. They cannot
> be included anymore by an include rule in the region filter block.

An exclude rule starts with the keyword EXCLUDE followed by one or multiple white-space separated region names.
Respectively, include rules start with INCLUDE followed by one or multiple white-space separated expressions. For
the specification of region names, bash-like wild-cards are supported. In particular, the '∗' wild card matches an
string of arbitrary length, the '?' matches exactly one arbitrary character, or within [] you may specify multiple
options.

An example region filter block could look like this:

```
SCOREP_REGION_NAMES_BEGIN
  EXCLUDE *
  INCLUDE bar foo
          baz
          main
SCOREP_REGION_NAMES_END
```

In this example, all but the functions bar, foo, baz and main are filtered.

The filtering is based on the region names as seen by the measurement system. Depending on instrumentation method and compiler the actual region name may be mangled, or decorated. Thus, you may want to inspect the profile to determine the name of a region inside the measurement system.

In some cases, the instrumentation provides mangled names, which are demangled by Score-P. In this cases, Score-P uses the demangled form for display in profile and trace definitions, and thus, the demangled form should be used in the filter file. However, The MANGLED keyword marks a filter rule to be applied on the mangled name, if a different mangled name is available. If no mangled name is available, the rule is applied on the displayed name instead. The MANGLED keyword must appear inside of an include rule or exclude rule. All patterns of the rule that follow the MANGLED keyword, are applied to the mangled name, if the mangled name is available. As a counterpart to the MANGLED keyword there is also a DEMANGLED keyword, which switches the pattern matching back to demangled region names. Both switches are limited to the scope of their respective include and exclude rule, with each rule starting in the demangled mode.

In the following example, foo and baz are applied to the mangled name, while bar and main are applied on the displayed name.

```
SCOREP_REGION_NAMES_BEGIN
  EXCLUDE *
  INCLUDE bar MANGLED foo
          baz
  INCLUDE main
SCOREP_REGION_NAMES_END
```

The displayed name may also be mangled if no demangled form is available. It is not necessary to prepend rules with the MANGLED keyword if the displayed name is mangled, but only if a mangled name is available that differs from the displayed name.

Note

> The keywords (e.g., EXCLUDE, INCLUDE, SCOREP_REGION_NAMES_BEGIN, SCOREP_REGION_N↩
> AMES_END, and MANGLED are case-sensitive.
> Instrumentation via the GCC plug-in or Intel compilers support all above mentioned filtering features when using the --instrument-filter=<file> flag to the Score-P instrumenter (see Sec. 'Automatic Compiler Instrumentation'). The filter file is then used during compilation time but the use of such a filter file during runtime is still possible. The usage of the filter during compilation time removes the overhead of runtime filtering.

5.4   Selective Recording

Score-P experiments record by default all events during the whole execution run. If tracing is enabled the event data will be collected in buffers on each process that must be adequately sized to store events from the entire execution.

Instrumented routines which are executed frequently, while only performing a small amount of work each time they are called, have an undesirable impact on measurement. The measurement overhead for such routines is large in comparison to the execution time of the uninstrumented routine, resulting in measurement dilation. Recording these events requires significant space and analysis takes longer with relatively little improvement in quality. Filtering can

be employed during measurement (described in Section 'Filtering') to ignore events from compiler-instrumented routines or user-instrumented routines.

Another possibility is not to record the whole application run. In many cases, only parts of the application are of interest for analysis (e.g., a frequently performed calculation) while other parts are of less interest (e.g., initialization and finalization) for performance analysis. Or the calculation itself shows iterative behavior, where recording of one iteration would be sufficient for analysis. Restricting recording to one or multiple time intervals during measurement would reduce the required space and overhead. This approach is called selective recording.

Score-P provides two possibilities for selective recording.

- A configuration file can specify recorded regions. The entry and exit of those regions define an interval during which events are recorded.

- With user instrumentation, the recording can be manually switched on /off. (See Section 'Manual Region Instrumentation').

Switching recording on or off, can result in inconsistent traces or profiles, if not applied with care. Especially, switching recording on/off manually via `SCOREP_RECORDING_ON` and `SCOREP_RECORDING_OFF` from the Score-P user instrumentation macros is not recommended. Inconsistent traces may result in errors or deadlocks during analysis, or show unusable data. The consistency is endangered if:

- OpenMP events are missing in one thread while other threads have them. Furthermore, the OpenMP parallel region events are required if any event inside a parallel region is recorded. To prevent inconsistencies from incomplete recording of OpenMP events, it is not possible to switch recording on/off from inside a parallel region

- MPI a communication is only recorded partially, e.g., if a send is missing, but the corresponding receive on another process is recorded. To ensure recording of complete communication is the responsibility of the user.

- enter/exit events are not correctly nested.

How recording can be controlled through Score-P macros which are inserted in the application's source code, is explained in Section 'Manual Region Instrumentation'. Thus, this section focuses on first possibility, where the user specify recorded regions via a configuration file. Selective recording affects tracing and profiling.

For selective recording, you can specify one or multiple traced regions. The recording is enabled when a recorded region is entered. If the region is exited, recording of events is switched off again. If a recorded region is called inside another recorded region, thus, the recording is already enabled, it will not disable recording of it exits, but recording will be switched off, if all recorded regions are exited.

For recorded regions only regions from Score-P user instrumentation can be selected. If regions from other instrumentation methods are specified in the configuration file for selective recording, they are ignored.

For a recorded region, the recording can be restricted to certain executions of that region. Therefor, the enters for a recorded region are counted, and a particular execution can be specified by the number of its enter. If a recorded region is called recursively, the recording is only switched off, if the exit is reached, that corresponds to the enter that enabled recording.

The configuration file is a simple text file, where every line contains the name of exactly one region. Optionally, a comma-separated list of execution numbers or intervals of execution numbers can be specified. A configuration file could look like follows:

```
foo
bar 23:25, 50, 60:62
baz 1
```

This configuration file would record all executions of foo, the executions 23, 24, 25, 50, 60, 61, and 62 of bar, and the second (numbering starts with 0) execution of baz.

To apply the selective recording configuration file to a measurement run of your application, set the environment variable `SCOREP_SELECTIVE_CONFIG_FILE` to the configuration file and run your instrumented application. If `SCOREP_SELECTIVE_CONFIG_FILE` is empty, or the given file cannot be opened, the whole application run will be recorded (no selective recording will apply).

5.5   Trace Buffer Rewind

Introducing a long-term event-trace recording mode, the trace buffer rewind feature allows to discard the preceding section of the event trace at certain control points or phase markers. The live decision whether to keep or discard a section can depend on the presence or absence of certain behaviour patterns as well as on similarity or difference with other sections.

Based on user regions (see 'Manual Region Instrumentation'), three macros are given which control the rewind. These are:

```
  // to define a local region handle based on the function
  // SCOREP_USER_REGION_DEFINE( ... )
SCOREP_USER_REWIND_DEFINE( regionHandle )
  // similar to SCOREP_USER_REGION_BEGIN( ... )
SCOREP_USER_REWIND_POINT( regionHandle, "name" )
  // similar to SCOREP_USER_REGION_END( ... )
  // w/ additional parameter to control the rewind (yes or no)
SCOREP_USER_REWIND_CHECK( regionHandle, boolean )
```

The user has to specify whether or not a rewind is requested with a boolean variable in the SCOREP_USER_R↩EWIND_CHECK function. There are two different approaches what to do with the rewind region in the trace based on the boolean variable. If the boolean variable is true, the trace buffer will be reset to an old snapshot and after that rewind region enter and leave events will be written into the trace buffer to mark the presence of the trace buffer rewind. This rewind region then looks like a normal user-defined region in the trace. If the variable is false, than no events of the rewind region are written into the trace, so that the trace buffer looks like the user never instrumented the code w/ rewind regions. Trace buffer flushes have an impact on the rewind regions, i.e. if a flush occurs all previous stored rewind points (which are not "checked", i.e. the flush is in between the region) will be deleted and the SCOREP_USER_REWIND_CHECK function won't write the enter/leave events into the trace independently from the boolean variable. Wrong nested rewind regions are handled as follows:

```
SCOREP_USER_REWIND_POINT( point 1, ...);
... do stuff ...
SCOREP_USER_REWIND_POINT( point 2, ...);
... do stuff ...
SCOREP_USER_REWIND_CHECK( point 1, true );
... do stuff ...
SCOREP_USER_REWIND_CHECK( point 2, true );
```

The check for point 2 would corrupt the trace buffer, so point 2 would be deleted and ignored in the second check.

5.6   Recording Performance Metrics

If Score-P has been built with performance metric support it is capable of recording performance counter information. To request the measurement of certain counters, the user is required to set individual environment variables. The user can leave these environment variables unset to indicate that no counters are requested. Requested counters will be recorded with every enter/exit event.

5.6.1   PAPI Hardware Performance Counters

Score-P provides the possibility to query hardware performance counters and include these metrics into the trace and/or profile. Score-P uses the Performance Application Programming Interface (PAPI) to access hardware performance counters. Recording of PAPI performance counters is enabled by setting the environment variable SCOREP_METRIC_PAPI to a comma-separated list of counter names. Counter names can be any PAPI preset names or PAPI native counter names.

Example:

```
SCOREP_METRIC_PAPI=PAPI_FP_OPS,PAPI_L2_TCM
```

This will record the number of floating point instructions and level 2 cache misses. If any of the requested counters is not recognized, program execution will be aborted with an error message. The PAPI utility programs `papi_avail` and `papi_native_avail` report information about the counters available on the current platform.

If you want to change the separator used in the list of PAPI counter names, set the environment variable SCORE↩P_METRIC_PAPI_SEP to the desired character.

Note

> In addition it is possible to specify metrics that will be recorded only by the initial thread of a process. Please use SCOREP_METRIC_PAPI_PER_PROCESS for that reason.

5.6.2 Resource Usage Counters

Besides PAPI, Resource Usage Counters can be recorded. These metrics use the Unix system call `getrusage` to provide information about consumed resources and operating system events such as user/system time, received signals, and number of page faults. The manual page of `getrusage` provides a list of resource usage counters. Please note that the availability of specific counters depends on the operating system.

You can enable recording of resource usage counters by setting the SCOREP_METRIC_RUSAGE environment variable. The variable should contain a comma-separated list of counter names.

Example:

```
SCOREP_METRIC_RUSAGE=ru_utime,ru_stime
```

This will record the consumed user time and system time. If any of the requested counters is not recognized, program execution will be aborted with an error message.

Note

> Please be aware of the scope of displayed resource usage statistics. Score-P records resource usage statistics for each individual thread, if the output while configuring your Score-P installation contains something like
>
> ```
> RUSAGE_THREAD support: yes
> ```
>
> Otherwise, the information displayed is valid for the whole process. That means, for multi-threaded programs the information is the sum of resources used by all threads in the process.

A shorthand to record all resource usage counters is

```
SCOREP_METRIC_RUSAGE=all
```

However, this is not recommended as most operating systems does not support all metrics.

If you want to change the separator used in the list of resource usage metrics, set the environment variable SCO↩REP_METRIC_RUSAGE_SEP to the desired character.

Example:

```
SCOREP_METRIC_RUSAGE_SEP=:
```

This indicates that counter names in the list are separated by colons.

Note

> In addition it is possible to specify metrics that will be recorded only by the initial thread of a process. Please use SCOREP_METRIC_RUSAGE_PER_PROCESS for that reason.

5.6.3   Recording Linux Perf Metrics

This metric source uses the Linux Perf Interface to access hardware performance counters. First it is explained how to specify PERF metrics that will be recorded by every location.

You can enable the recording of PERF performance metrics by setting the environment variable SCOREP_MET↩RIC_PERF to a comma-separated list of metric names. Metric names can be any PERF preset names or PAPI native counter names.

Example:

```
SCOREP_METRIC_PERF=cycles,page-faults,LLC-load-misses
```

In this example the number of CPU cycles, the number of page faults, and Last Level Cache Load Misses will be recorded. If any of the requested metrics is not recognized program execution will be aborted with an error message. The user can leave the environment variable unset to indicate that no metrics are requested. Use the tool `perf list` to get a list of available PERF events.

If you want to change the separator used in the list of PERF metrics, set the environment variable SCOREP_ME↩TRIC_PERF_SEP to the desired character.

Example:

```
SCOREP_METRIC_PERF_SEP=:
```

This indicates that counter names in the list are separated by colons.

Note

> In addition it is possible to specify metrics that will be recorded per-process. Please use SCOREP_METRI↩C_PERF_PER_PROCESS for that reason.

5.6.4   Metric Plugins

Metric plugins extend the functionality of Score-P by providing additional counters as external libraries. The libraries are loaded when tracing or profiling your application. So there is no need to recompile your application or instrument it manually.

A simple example of a synchronous metric plugin can be found in Appendix 'Score-P Metric Plugin Example'. Every plugin needs to include SCOREP_MetricPlugins.h. The commands to build the corresponding library of this plugin might look like:

```
gcc -c -fPIC hello_world.c   \
    -o libHelloWorld_plugin.so.o 'scorep-config --cppflags'
gcc -shared -Wl,-soname,libHelloWorld_plugin.so \
    -o libHelloWorld.so libHelloWorld_plugin.so.o
```

To enable a metric plugin, add the plugin <PLUGINNAME> to the environment variable SCOREP_METRIC_↩PLUGINS and configure the used metrics through the environment variable SCOREP_METRIC_PLUGINNAME. In the following example we want to use the above `HelloWorld` plugin. We select two counters `metric1` and `metric2` from the plugin. Make sure that the metric plugin library is placed in a directory which is part of `LD_LIBRARY_PATH`.

```
SCOREP_METRIC_PLUGINS=HelloWorld_plugin
SCOREP_METRIC_HELLOWORLD_PLUGIN=metric1,metric2
```

Note

> Plugins are not supposed to trigger events (e.g., via MPI, OpenMP, Pthreads or user instrumentation) during initialization and finalization of the plugin.
> A set of open source metric plugins is available at `GitHub`.

Token	Module
ALL	Activate all available modules
DEFAULT	Activate the configured default modules of CG, COLL, ENV, IO, P2P, RMA, TOPO, XNONBLOCK. This can be used to easily activate additional modules.
CG	Communicators and groups
COLL	Collective communication
ENV	Environmental management
ERR	Error handlers
EXT	External interfaces
IO	I/O
MISC	Miscellaneous
P2P	Point-to-point communication
RMA	One-sided communication
SPAWN	Process management interface (aka Spawn)
TOPO	Topology communicators
TYPE	MPI Datatypes
XNONBLOCK	Extended non-blocking communication events
XREQTEST	Test events for tests of uncompleted requests

5.7 MPI Performance Measurement

The Message Passing Interface (MPI) adapter of Score-P supports the tracing of most of MPI's 300+ function calls. MPI defines a so-called 'profiling interface' that supports the provision of wrapper libraries that can easily interposed between the user application and the MPI library calls.

5.7.1 Selection of MPI Groups

The general Score-P filtering mechanism is not applied to MPI functions. Instead, the user can decide whether event generation is turned on or off for a group of MPI functions, at start time of the application. These groups are the listed sub-modules of this adapter. Each module has a short string token that identifies this group. To activate event generation for a specific group, the user can specify a comma-separated list of tokens in the configuration variable SCOREP_MPI_ENABLE_GROUPS. Additionally, special tokens exist to ease the handling by the user. A complete list of available tokens that can be specified in the runtime configuration is listed below.

Note

> Event generation in this context only relates to flow and transfer events. Tracking of communicators, groups, and other internal data is unaffected and always turned on.

Example:

```
SCOREP_MPI_ENABLE_GROUPS=ENV,P2P
```

This will enable event generation for environmental management, including `MPI_Init` and `MPI_Finalize`, as well as point-to-point communication, but will disable it for all other functions groups.

A shorthand to get event generation for all supported function calls is

```
SCOREP_MPI_ENABLE_GROUPS=ALL
```

A shorthand to add a single group, e.g., `TYPE`, to the configured default is

```
SCOREP_MPI_ENABLE_GROUPS=DEFAULT,TYPE
```

A detailed overview of the MPI functions associated with each group can be found in Appendix 'MPI wrapper affiliation'.

A somewhat special role plays the `XNONBLOCK` flag. This flag determines what kind of events are generated by non-blocking peer-to-peer MPI function calls. If `XNONBLOCK` is not set, an OTF2_MPI_Send event is created at the non-blocking send call and an OTF2_MPI_Recv event is recorded when a non-blocking receive request has completed. **Measurements without the `XNONBLOCK` flag are deprecated**: Set it either explicitly, or by enabling one of the presets `DEFAULT` or `ALL`. If `XNONBLOCK` is set, an OTF2_Isend event is recorded at the non-blocking send and an OTF2_IsendComplete event when the event was completed. Furthermore, on a non-blocking receive, it records an OTF2_IRecvRequest event. On request completion an OTF2_IRecv event is recorded. In both cases the group `P2P` must be enabled. Otherwise Score-P records no events for peer-to-peer communication functions.

5.7.2   Recording MPI Communicator Names

The measurement system tracks also the names of MPI communicators to easily identify them later in the analysis. This is done via the `MPI_Comm_set_name` call. But there are some restrictions. First, the name of a communicator is only recorded at the first call to `MPI_Comm_set_name` for this communicator. Later calls are ignored. Also this call is only honored when the call was made from the rank which is rank 0 in this communicator. Other calls from other ranks are ignored. And lastly the name will also be not recorded if the communicator has only one member.

5.8   CUDA Performance Measurement

If Score-P has been built with CUDA support it is capable of recording CUDA API function calls and GPU activities. The measurement is based on NVIDIA's **CU**DA **P**rofiling and **T**ool **I**nterface (CUPTI), which is an integral part of the CUDA Toolkit since version 4.1.

Score-P can wrap the NVIDIA's **nvcc** compiler like any other compiler by either prefixing (**scorep nvcc**) or by replacing (**scorep-nvcc**) it to instrument `.cu` files. **nvcc** by default uses **g++** to compile host-code. For Score-P to be able to compiler-instrument the host-code, Score-P's compiler and **nvcc**'s host-code compiler must match. If they don't match, compilation is likely to fail with an `unrecognized command-line option`. There are ways around this problem:

- Make the `nvcc` host-code compiler match the Score-P compiler by adding `-ccbin <SCOREP_COMP`↩ `ILER>` to the `nvcc` invocation. You can query `<SCOREP_COMPILER>` and the entire command using `scorep-info config.summary | grep 'nvcc works'`. E.g., if the `.cu` file does neither contain MPI nor SHMEM and Score-P uses an Intel compiler suite, the command could be `nvcc -ccbin icpc -Xcompiler=-std=c++11`. By exporting `NVCC_PREPEND_FLAGS="-ccbin icpc -`↩ `Xcompiler=-std=c++11"`, this configuration will be used by subsequent `nvcc` invocations.

- Disable the host-code instrumentation of `.cu` files by using the Score-P flag `--nocompiler`.

- Use a Score-P installation that is based on the GNU compiler tool chain.

Note

> If Score-P cannot build a minimal CUDA program using `-ccbin`, CUDA support will be disabled for this installation. Please inform the system administrator about the inconsistency in the software stack.

Setting the environment variable SCOREP_CUDA_ENABLE to **yes** enables CUDA measurement. Please refer to the description of this variable to enable a particular combination of CUDA measurement features.

CUPTI uses an extra buffer to store its activity records. If the size of this buffer is too small, Score-P will print a warning about the current buffer size and the number of dropped records. To avoid dropping of records increase the buffer size via the environment variable SCOREP_CUDA_BUFFER (default: 1M).

Since CUDA toolkit version 5.5 the chunk size for the CUPTI activity buffer can be specified via the environment variable SCOREP_CUDA_BUFFER_CHUNK (default: 8k). Buffer chunks are allocated whenever CUPTI requests a buffer (e.g., to record activities on a CUDA stream). SCOREP_CUDA_BUFFER specifies the upper bound of memory to be allocated for CUPTI activities. Therefore it should be a multiple of SCOREP_CUDA_BUFFER_CH↩
UNK.

Note

> Make sure to call `cudaDeviceReset()` or `cudaDeviceSynchronize()` before the exit of the program. Otherwise GPU activities might be missing in the trace. Previous versions of Score-P allowed users to optionally mitigate this through the **flushatexit** value in SCOREP_CUDA_ENABLE. This has been inverted; now users who perform this synchronization and would like to see if performance improves by removing a redundant synchronize in Score-P may set **dontflushatexit** in SCOREP_CUDA_ENABLE instead. **flushatexit** is deprecated and will be removed in a future version of Score-P.
> For CUDA 5.5 there is an error in CUPTI buffer handling. The last activity in a CUPTI activity buffer (SC↩
> OREP_CUDA_BUFFER_CHUNK) gets lost, when the buffer is full. To avoid this issue specify SCOREP_↩
> CUDA_BUFFER_CHUNK as large as necessary to store all CUDA device activities until the CUDA device is synchronized with the host. In CUDA 6.0 this issue is fixed and CUPTI does not request buffers for individual streams any more.
> Score-P supports CUDA monitoring since CUDA toolkit version 4.1. Make sure that the Score-P installation has configured CUDA support. The configure summary should contain the line:
>
> ```
> CUDA support: yes
> ```
>
> If not, for most systems it is sufficient to specify the CUDA toolkit directory at Score-P configuration time:
>
> ```
>   --with-libcudart=<path-to-cuda-toolkit-directory>
> ```
>
> Otherwise check the configure help output to specify the location of the CUDA toolkit and CUPTI libraries and include files:
>
> ```
>   ../configure --help=recursive | grep -E "(cuda|cupti)"
> ```

CUDA device and host activities can be filtered by name at runtime using the Score-P filter file (see Section 'Filtering'). Filtering does not remove CUDA activities inserted by Score-P or CUDA data transfers inserted as RDMA events. If a kernel is filtered, no kernel launch properties activated in SCOREP_CUDA_ENABLE using `kernel`↩
`_counter` are inserted for this kernel. GPU idle time is not affected by kernel filtering.

5.8.1   CUDA NVTX Measurement

NVTX intrumentation is enabled if instrumented with `--cuda`. The Start/Stop Range events are not supported, because they allow to span different threads and do not need to be nested. NVTX Mark events are modelled as a short sequence of Enter and Leave events. NVTX user regions are grouped by the NVTX domain. The region group name for the default domain is `"NVTX"`, for user defined domains it is `"NVTX Domain '<DOMAIN-N`↩
`AME>'"`. Payloads in the `nvtxEventAttributes_t` arguments are converted to event attributes of the Enter event. These are only available in tracing mode, as they are potential non-categorical data and profiling does not support floating point parameters. Payload names are of the form `"NVTX Payload <DATATYPE-NAME>"`, where `<DATATYPE-NAME>` is derived from the NVTX datatype enum by removing the `NVTX_PAYLOAD_TY`↩
`PE_` prefix. Categories in the `nvtxEventAttributes_t` arguments are converted to the parameter `"NVTX Category"` (see 'Parameter-Based Profiling'), hence are available in profiling and tracing.

Note

> While NVTX considers different domains as disjoint range stacks, Score-P does not support overlapping
> ranges.

The following resource naming APIs are supported:

nvtxNameOsThread{A,W} Name of CPU location

nvtxNameCuStream{A,W} Name of GPU location

nvtxNameCuContext{A,W} Name of GPU location group

5.9 HIP Performance Measurement

If Score-P has been built with HIP support it is capable of recording ROCm API function calls and GPU activities.
The measurement is based on AMD's ROCm API.

Score-P can wrap the AMD's **hipcc** compiler like any other compiler by either prefixing (**scorep hipcc**) or by
replacing it with (**scorep-hipcc**) to instrument single-source HIP files.

Setting the environment variable SCOREP_HIP_ENABLE to **yes** enables HIP measurement. Please refer to the
description of this variable to enable a particular combination of HIP measurement features.

Host API calls can be filtered by name at runtime using the Score-P filter file (see Section 'Filtering'). Filtering does
not apply to API functions which are enabled by settings in SCOREP_HIP_ENABLE other than `api`. For example,
the API `hipLaunchKernel` can be filtered if **api** is the only setting in SCOREP_HIP_ENABLE. If **kernels**
is set additionally, then it is not allowed to filter this call anymore.

ROCm uses an extra buffer to store its activity records. The size of this buffer can be controlled via the environment
variable SCOREP_HIP_ACTIVITY_BUFFER_SIZE (default: 1M).

For each HIP stream a location is created with the name **HIP[D:S]**. Where `D` denotes the HIP device and `S`
a stream identifier. The former number corresponds to the visible devices for the process. The HIP `NULL`-stream
always gets the identifier `0`. Additionally created streams get a number starting with `1` in creation order. The location
for the HIP `NULL`-stream also gets a property named `hipNullStream` with a value of `1`. The non-`NULL`-streams
get properties for their flags and priority:

hipStreamNonBlocking `1` if the streams was created with this flag

hipStreamPriority Priority for this stream

All streams of a device are grouped into a location group named **HIP Context D** with the same value for `D` as
the streams. This location group is attributed to a system tree node denoting the actual hardware device, which is
named **ROCm Device H**. Here `H` denotes the hardware identifier of the device, which might be different to `D`, if
the visible devices were restricted for the process and Score-P was built with support for AMD's SMI library.

The following values from `hipDeviceProp_t` are added as properties to the system tree node for each device
(member -> property name mapping):

name Device name

gcnArchName AMD GCN architecture name

pciDomainID PCI Domain ID

pciBusID PCI Bus ID

pciDeviceID PCI Device ID

totalGlobalMem Size of global memory region (in bytes)

sharedMemPerBlock Size of shared memory region (in bytes)

totalConstMem

regsPerBlock Registers per block

warpSize Warp size

maxThreadsPerBlock Maximum work items per work group or workgroup max size

maxThreadsDim Maximum number of threads in each dimension (XYZ) of a block

maxGridSize Maximum grid dimensions (XYZ)

clockRate Maximum clock frequency of the multi-processors (in khz)

memoryClockRate Maximum global memory clock frequency (in khz)

clockInstructionRate Timer clock frequency (in khz)

memoryBusWidth Global memory bus width (in bits)

isMultiGpuBoard Is multi-GPU board (1 if yes, 0 if not)

canMapHostMemory Device can map host memory (1 if yes, 0 if not)

concurrentKernels Device can possibly execute multiple kernels concurrently (1 if yes, 0 if not)

multiProcessorCount Number of multi-processors (compute units)

l2CacheSize L2 cache size

maxThreadsPerMultiProcessor Maximum resident threads per multi-processor

Additionally the device gets the device's UUID as a property named *UUID*.

Recording code annotations via the ROCTX API must be instrumented with the Score-P flag `--hip` and enabled by setting **user** in SCOREP_HIP_ENABLE. The Start/Stop Range events are not supported, because they allow to span different threads and do not need to be nested. The Mark events are modeled as a short sequence of Enter and Leave events. All regions are in the `"ROCTX"` group and are subject to the filtering rules (see Section 'Filtering').

5.10   OpenCL Performance Measurement

If Score-P has been built with OpenCL support it is capable of recording OpenCL API function calls.

Setting the environment variable SCOREP_OPENCL_ENABLE to **yes** enables OpenCL measurement. Please refer to the description of this variable to enable a particular combination of OpenCL measurement features.

OpenCL measurement uses an extra buffer to store its activity records. If the size of this buffer is too small, Score-P will print a warning about the current buffer size and the number of dropped records. To avoid dropping of records increase the buffer size via the environment variable SCOREP_OPENCL_BUFFER (default: 1M). Memory in bytes for the OpenCL command queue buffer can be adjusted by setting the environment variable SCOREP_OPENCL↩ _BUFFER_QUEUE (default: 8k).

5.11 OpenACC Performance Measurement

If Score-P has been built with OpenACC support it is capable of recording OpenACC regions as well as activities such as enqueuing kernels, data uploads, and data downloads. OpenACC activities that are implicitly generated by the compiler are attributed with `acc_implicit`.

To enable OpenACC measurement in Score-P the user has to:

- Build and install shared libraries of Score-P (use `--enable-shared` option when configuring Score-P).

- Set the environment variable `ACC_PROFLIB` to specify the OpenACC profiling library.
  Example:

  ```
  export ACC_PROFLIB=<path_to_scorep_installation>/lib/libscorep_adapter_openacc_event.so
  ```

- Set the environment variable `SCOREP_OPENACC_ENABLE` to **yes**. Please refer to the description of this variable to enable a particular combination of OpenACC measurement features.

Note

Score-P supports OpenACC monitoring, if the respective OpenACC compiler implements the OpenACC profiling interface that is part of the OpenACC standard since version 2.5. Make sure that the Score-P installation has configured OpenACC support. The configure summary should contain the line:

```
OpenACC support: yes
```

5.12 Kokkos Performance Measurement

Score-P is capable of recording information from the Kokkos performance portability framework's tools interface, including parallel regions, user regions, memory allocation, and memory copies. As with OpenACC, the user must perform the following steps:

- Build and install shared libraries of Score-P (use `--enable-shared` option when configuring Score-P).

- Set the environment variable `KOKKOS_PROFILE_LIBRARY` to specify the Kokkos profiling library.
  Example:

  ```
  export KOKKOS_PROFILE_LIBRARY=<path_to_scorep_installation>/lib/libscorep_adapter_kokkos_even
  ```

- Set the environment variable `SCOREP_KOKKOS_ENABLE` to **yes**. Please refer to the description of this variable to enable a particular combination of Kokkos measurement features.

Kokkos events can be filtered by name at runtime using the Score-P filter file (see Section 'Filtering'). This applies to all types of Kokkos events. Users should note that the Kokkos tools interface allows explicit naming of parallel regions, and those names, if provided, will be the names used for filtering.

5.13 Substrate Plugins

Substrate plugins extend the functionality of Score-P by providing additional backends. The libraries are loaded when your application is started and Score-P is initialized. So there is no need to recompile your application or instrument it manually.

A simple example of a backend can be found in Appendix 'Score-P Substrate Plugin Example'. Every plugin needs to include SCOREP_SubstratePlugins.h and define its name via SCOREP_SUBSTRATE_PLUGIN_EN↩TRY(<name>). The same name must be used with the prefix `scorep_substrate_` as library name. The commands to build the corresponding library of this plugin might look like:

```
gcc -c -fPIC print_regions.c   \
    -o libscorep_substrate_PrintRegions.so.o `scorep-config --cppflags`
gcc -shared -Wl,-soname,libscorep_substrate_PrintRegions.so  \
    -o libscorep_substrate_PrintRegions.so libscorep_substrate_PrintRegions.so.o
```

To enable a substrate plugin, add the plugin <PLUGINNAME> to the environment variable SCOREP_SUBSTR↩ATE_PLUGINS. In the following example we want to use the above `PrintRegions` plugin. Make sure that the substrate plugin library is placed in a directory which is part of `LD_LIBRARY_PATH`.

```
SCOREP_SUBSTRATE_PLUGINS=PrintRegions
```

Note

> Multiple substrate plugins can be loaded by listing them `SCOREP_SUBSTRATE_PLUGINS=foo,bar`, the default separator ',' can be changed by setting the environment variable SCOREP_SUBSTRATE_PLUGIN↩S_SEP.
> Developers of plugins are highly encouraged to use environment variables in their plugins with the naming scheme `SCOREP_SUBSTRATE_<substrate name>_<variable>`, e.g., `SCOREP_SUBSTRAT↩E_PRINTREGIONS_VERBOSE`

Chapter 6

Scoring a Profile Measurement

`scorep-score` is a tool that allows to estimate the size of an OTF2 trace from a CUBE4 profile. Furthermore, the effects of filters are estimated. The main goal is to define appropriate filters for a tracing run from a profile.

The general work-flow for performance analysis with Score-P is:

1. Instrument an application (see Section 'Application Instrumentation').

2. Perform a measurement run and record a profile (see Section 'Application Measurement'). The profile already gives an overview what may happen inside the application.

3. Use `scorep-score` to define an appropriate filter for an application Otherwise the trace file may become too large. This step is explained in this Chapter.

4. Perform a measurement run with tracing enabled and the filter applied (see Section 'Tracing' and Section 'Filtering').

5. Perform in-depth analysis on the trace data.

6.1   Basic usage

To invoke `scorep-score` you must provide the filename of a CUBE4 profile as argument. Thus, the basic command looks like this:

```
scorep-score profile.cubex
```

The output of the command may look like this (taking an MPI/OpenMP hybrid application as an example):

```
Estimated aggregate size of event trace:                20MB
Estimated requirements for largest trace buffer (max_buf): 20MB
Estimated memory requirements (SCOREP_TOTAL_MEMORY):      24MB
(hint: When tracing set SCOREP_TOTAL_MEMORY=24MB to avoid intermediate flushes
 or reduce requirements using USR regions filters.)

flt type       max_buf[B]         visits       time[s]  time[%]  time/visit[us]  region
     ALL      19,377,048        786,577         27.48    100.0           34.93  ALL
     USR      16,039,680        668,320          0.36      1.3            0.53  USR
     OMP       3,328,344        117,881         26.92     98.0          228.37  OMP
     COM           9,024            376          0.20      0.7          532.17  COM
  SCOREP            41              1          0.00      0.0           13.82  SCOREP
```

The first line of the output gives an estimation of the total size of the trace, aggregated over all processes. This information is useful for estimating the space required on disk. In the given example, the estimated total size of the event trace is 20MB.

The second line prints an estimation of the memory space required by a single process for the trace. The memory space that Score-P reserves on each process at application start must be large enough to hold the process' trace in memory in order to avoid flushes during runtime, because flushes heavily disturb measurements. In addition to the trace, Score-P requires some additional memory to maintain internal data structures. Thus, it provides also an estimation for the total amount of required memory on each process. The memory size per process that Score-P reserves is set via the environment variable SCOREP_TOTAL_MEMORY. In the given example the per process memory should be larger than 24MB.

Beginning with the 6th line, `scorep-score` prints a table that show how the trace memory requirements and the runtime is distributed among certain function groups. The column `max_tbc` shows how much trace buffer is needed on a single process. The column `time(s)` shows how much execution time was spend in regions of that group in seconds, the column `%` shows the fraction of the overall runtime that was used by this group, and the column `time/visit(us)` shows the average time per visit in microseconds.

The following groups exist:

ALL: Includes all functions of the application

OMP: This group contains all regions that represent an OpenMP construct

MPI: This group contains all MPI functions

SHMEM: This group contains all SHMEM functions

PTHREAD: This group contains all Pthread functions

CUDA: This group contains all CUDA API functions and kernels

OPENCL: This group contains all OpenCL API functions and kernels

OPENACC: This group contains all OpenACC API functions and kernels

MEMORY: This group contains all libc and C++ memory (de)allocation functions

IO: This group contains all I/O functions

KOKKOS: This group contains all Kokkos regions

HIP: This group contains all HIP functions

LIB: This group contains all user wrapped library functions (See 'Score-P User Library Wrapping')

COM: This group contains all functions, implemented by the user that appear on a call-path to any functions from the above groups, except **ALL**

USR: This group contains all user functions except those in the **COM** group

SCOREP: This group aggregates activities within the measurment system

6.2 Additional per-region information

For a more detailed output, which shows the data for every region, you can use the `-r` option. The command could look like this.

```
scorep-score profile.cubex -r
```

This command adds information about the used buffer sizes and execution time of every region to the table. The additional lines of the output may look like this:

```
flt type      max_buf[B]      visits     time[s]  time[%]  time/visit[us]  region

     COM              24           4        0.00      0.0           67.78  Init
     COM              24           4        0.00      0.0           81.20  main
     USR              24           4        0.12      2.0        30931.14  InitializeMatrix
     COM              24           4        0.05      0.8        12604.78  CheckError
     USR              24           4        0.00      0.0           23.76  PrintResults
     COM              24           4        0.01      0.2         3441.83  Finish
     COM              24           4        0.48      7.7       120338.17  Jacobi
```

The region name is displayed in the column named `region`. The column `type` shows to which group this region belongs. In the example above the function main belongs to group `COM` required 24 bytes per process and used 0 s execution time. By default, the regions are sorted by their buffer requirements. With the option `-s <choice>` a different sorting mode can be chosen. Available options are `totaltime`, `timepervisit`, `maxbuffer`, `visits` and `name`.

By default `scorep-score` uses demangled function names. However, if you want to map data to tools which use mangled names you might want to display mangled names. Furthermore, if you have trouble with function signatures that contain characters that also have a wildcard meaning, defining filters on mangled names might be easier. To display mangled names instead of demangled names, you can use the -m flag, e.g.,

```
scorep-score profile.cubex -r -m
```

Note

> The -m flag takes only effect if you display region names. In particular it means that the -m flag is only effective if also the -r is specified.
> In some cases, the same name is shown for the mangled and the demangled name. Some instrumentation methods, e.g., user instrumentation, provide only a demangled name. For C-compilers mangled and demangled names are usually identical. Or the demangling might have failed and only a mangled name is available. In these cases we show always the one name that is available.

6.3 Defining and testing a filter

For defining a filter, it is recommended to exclude short frequently called functions from measurement, because they require a lot of buffer space (represented by a high value under `max_tbc`) but incur a high measurement overhead. Furthermore, for communication analysis, functions that appear on a call-path to MPI functions and OpenMP constructs (regions of type `COM`) are usually of more interest than user functions of type `USR` which do not appear on call-path to communications. MPI functions and OpenMP constructs cannot be filtered. Thus, it is usually a good approach to exclude regions of type `USR` starting at the top of the list until you reduced the trace to your needs. Section 'Filtering' describes the format of a filter specification file.

If you have a filter file, you can test the effect of your filter on the trace file. Therefor, you need to pass a `-f` followed by the file name of your filter. E.g., if your filter file name is `myfilter`, the command looks like this:

```
scorep-score profile.cubex -f myfilter
```

An example output is:

```
Estimated aggregate size of event trace:                 7kB
Estimated requirements for largest trace buffer (max_buf): 1806 bytes
Estimated memory requirements (SCOREP_TOTAL_MEMORY):      5MB
(hint: When tracing set SCOREP_TOTAL_MEMORY=5MB to avoid intermediate flushes
 or reduce requirements using USR regions filters.)

flt   type     max_buf[B]        visits       time[s]  time[%]  time/visit[us]  region
 -     ALL        2,093           172           5.17    100.0      30066.64  ALL
 -     MPI        1,805           124           4.20     81.3      33910.31  MPI
 -     COM          240            40           0.84     16.3      21092.44  COM
 -     USR           48             8           0.12      2.4      15360.71  USR
 -   SCOREP          41             4           0.00      0.0         13.82  SCOREP

 *     ALL        1,805           124           4.20     81.3      33910.31  ALL-FLT
 -     MPI        1,805           124           4.20     81.3      33910.31  MPI-FLT
 -   SCOREP          41             4           0.00      0.0         13.82  SCOREP-FLT
 +     FLT          288            48           0.97     18.7      20137.15  FLT
```

Now, the output estimates the total trace size an the required memory per process, if you would apply the provided filter for the measurement run which records the trace. A new group `FLT` appears, which contains all regions that are filtered. Under `max_tbc` the group `FLT` displays how the memory requirements per process are reduced. Furthermore, the groups that end on `-FLT`, like `ALL-FLT` contain only the unfiltered regions of the original group. E.g., `USR-FLT` contains all regions of group `USR` that are not filtered.

Furthermore, the column `flt` is no longer empty but contain a symbol that indicates how this group is affected by the filter. A '−' means 'not filtered', a '+' means 'filtered' and a '*' appears in front of groups that potentially can be affected by the filter.

You may combine the `-f` option with a `-r` option. In this case, for each function a '+' or '−' indicates whether the function is filtered.

6.4   Calculating the effects of recording hardware counters

Recording additional metrics, e.g., hardware counters may significantly increase the trace size, because for many events additional metric values are stored. In order to estimate the effects of these metrics, you may add a `-c` followed by the number of metrics you want to record, e.g.,

```
scorep-score profile.cubex -c 3
```

would mean that `scorep-score` estimates the disk and memory requirements for the case that you record 3 additional metrics.

6.5   Generating initial filter files

With the `-g` option `scorep-score` can generate an initial filter file in Score-P filter file format. It provides a starting point for the user to adapt and change the filter file to his requirements.

The user can provide an optional list of parameters to control the inclusion heuristic of the filter file generation. A valid parameter list has the form `KEY=VALUE[,KEY=VALUE]*`. By default, the following control parameters are used:

`bufferpercent=1,timepervisit=1`

A region is included in the filter file (i.e., excluded from measurement) if it matches all of the given conditions, with the following keys:

1. `bufferpercent`: The estimated memory requirements exceed the given threshold in percent of the total estimated trace buffer requirements. This limits the inclusion to relevant regions based on total memory impact.

2. `bufferabsolute`: The estimated memory requirements exceed the given absolute threshold in MB.

3. `visits`: The number of visits exceeds the given threshold.

4. `timepervisit`: The time per visit value is below the given threshold in microseconds. This limits the inclusion to relevant regions based on per visit time overhead.

5. `type`: The region type matches the given value. Allowed values are 'usr', 'com', and 'both'.

Note

> If the file name already exists, the previous file will be renamed instead overwritten to safeguard against unintentional deletion and to allow being used in an iterative use of the the filter file generation. To that end, filtered regions from a file specified via `-f` will be included at the end of the newly generated file. Those regions are marked and added 'as is' without being checked against the inclusion heuristics.

Chapter 7

Performance Analysis Workflow Using Score-P

This chapter demonstrates a typical performance analysis workflow using Score-P. It consist of the following steps:

1. Program instrumentation (Section 'Program Instrumentation')

2. Summary measurement collection (Section 'Summary Measurement Collection')

3. Summary report examination (Section 'Summary report examination')

4. Summary experiment scoring (Section 'Summary experiment scoring')

5. Advanced summary measurement collection (Section 'Advanced summary measurement collection')

6. Advanced summary report examination (Section 'Advanced summary report examination')

7. Event trace collection and examination (Section 'Event trace collection and examination')

The workflow is demonstrated using NPB BT-MZ benchmark as an example. BT-MZ solves a discretized version of unsteady, compressible Navier-Stokes equations in three spatial dimensions. It performs 200 time-steps on a regular 3-dimensional grid using ADI and verifies solution error within acceptable limit. It uses intra-zone computation with OpenMP and inter-zone communication with MPI. The benchmark can be build with a predefined data class (S,W,A,B,C,D,E,F) and any number of MPI processes and OpenMP threads.

NPB BT-MZ distribution already prepared for this example could be obtained from `here`.

7.1   Program Instrumentation

In order to collect performance measurements, BT-MZ has to be instrumented. There are various types of instrumentation supported by Score-P which cover a broad spectrum of performance analysis use cases (see Chapter 'Application Instrumentation' for more details).

In this example we start with automatic compiler instrumentation by prepending compiler/linker specification variable `MPIF77` found in `config/make.def` with `scorep`:

```
#              SITE- AND/OR PLATFORM-SPECIFIC DEFINITIONS
#---------------------------------------------------------------
# Items in this file may need to be changed for each platform.
#---------------------------------------------------------------
...
#---------------------------------------------------------------
# The Fortran compiler used for MPI programs
#---------------------------------------------------------------
#MPIF77 = mpif77
# Alternative variants to perform instrumentation
...
MPIF77 = scorep mpif77
# This links MPI Fortran programs; usually the same as ${MPIF77}
FLINK  = $(MPIF77)
...
```

After the makefile is modified and saved, it is recommended to return to the root folder of the application and clean-up previously build files:

```
% make clean
```

Now the application is ready to be instrumented by simply issuing the standard build command:

```
% make bt-mz CLASS=W NPROCS=4
```

After the command is issued, the make command should produce the output similar to the one below:

```
cd BT-MZ; make CLASS=W NPROCS=4 VERSION=
make: Entering directory 'BT-MZ'
cd ../sys; cc  -o setparams setparams.c -lm
../sys/setparams bt-mz 4 W
scorep mpif77 -c  -O3 -fopenmp bt.f
[...]
cd ../common;  scorep --user mpif77 -c  -O3 -fopenmp timers.f
scorep mpif77 -O3 -fopenmp -o ../bin.scorep/bt-mz_W.4 \
bt.o initialize.o exact_solution.o exact_rhs.o set_constants.o \
adi.o rhs.o zone_setup.o x_solve.o y_solve.o exch_qbc.o \
solve_subs.o z_solve.o add.o error.o verify.o mpi_setup.o \
../common/print_results.o ../common/timers.o
Built executable ../bin.scorep/bt-mz_W.4
make: Leaving directory 'BT-MZ'
```

When make finishes, the built and instrumented application could be found under `bin.scorep/bt-mz_W.4`.

7.2 Summary Measurement Collection

Now instrumented BT-MZ is ready to be executed and to be profiled by Score-P at the same time. However before doing so, one has an opportunity to configure Score-P measurement by setting Score-P environment variables. For the complete list of variables please refer to Appendix 'Score-P Measurement Configuration'.

The typical Score-P performance analysis workflow implies collecting summary performance information first and then in detail performance exploration using execution traces. Therefore Score-P has to be configured to perform profiling and tracing has to be disabled. This is done by setting corresponding environment variables:

```
% export SCOREP_ENABLE_PROFILING=1
% export SCOREP_ENABLE_TRACING=0
```

Performance data collected by Score-P will be stored in an experiment directory. In order to efficiently manage multiple experiments, one can specify a meaningful name for the experiment directory by setting

```
% export SCOREP_EXPERIMENT_DIRECTORY=scorep_bt-mz_W_4x4_sum
```

After Score-P is prepared for summary collection, the instrumented application can be started as usual:

```
% cd bin.scorep
% export OM_NUM_THREADS=4
% mpiexec -np 4 ./bt-mz_W.4
```

The BT-MZ output should look similar to the listing below:

```
NAS Parallel Benchmarks (NPB3.3-MZ-MPI) BT-MZ MPI+OpenMP Benchmark

Number of zones:   4 x   4
Iterations: 200    dt:   0.000800
Number of active processes:     4

Use the default load factors with threads
Total number of threads:    16  ( 4.0 threads/process)

Calculated speedup =    15.78

Time step    1
[... More application output ...]
```

After application execution is finished, the summary performance data collected by Score-P is stored in the experiment directory `bin.scorep/scorep_bt-mz_W_4x4_sum`. The directory contains the following files:

- `scorep.cfg` - a record of the measurement configuration used in the run

- `profile.cubex` - the analysis report that was collated after measurement

7.3   Summary report examination

After BT-MZ finishes execution, the summary performance data measured by Score-P can be investigated with CUBE or ParaProf interactive report exploration tools.

CUBE:

```
% cube scorep_bt-mz_W_4x4_sum/profile.cubex
```

ParaProf:

```
% paraprof scorep_bt-mz_W_4x4_sum/profile.cubex
```

Both tools will reveal the call-path profile of BT-MZ annotated with metrics: *Time*, *Visits count*, MPI message statistics (bytes sent/received). For more information on using the tool please refer to the corresponding documentation (CUBE, ParaProf).

7.4   Summary experiment scoring

Though we were able to collect the profile data, one can mention that the execution took longer in comparison to un-instrumented run, even when the time spent for measurement start-up/finalization is disregarded. Longer execution times of the instrumented application is a sign of high instrumentation/measurement overhead. Furthermore, this overhead might result in large trace files and buffer flushes in the later tracing experiment if Score-P is not properly configured.

In order to investigate sources of the overhead and to tune measurement configuration for consequent trace collection with Score-P, `scorep-score` tool (see Section 'Scoring a Profile Measurement' for more details about `scorep-score`) can be used:

```
% scorep-score scorep_bt-mz_W_4x4_sum/profile.cubex
Estimated aggregate size of event trace (total_tbc):
                              990247448 bytes
Estimated requirements for largest trace buffer (max_tbc):
                              256229936 bytes
(hint: When tracing set SCOREP_TOTAL_MEMORY > max_tbc to avoid
intermediate flushes
or reduce requirements using file listing names of USR regions
to be filtered.)
flt type         max_tbc        time       % region
     ALL        256229936     5549.78   100.0 ALL
     USR        253654608     1758.27    31.7 USR
     OMP          5853120     3508.57    63.2 OMP
     COM           343344      183.09     3.3 COM
     MPI            93776       99.86     1.8 MPI
```

The textual output of the tool generates an estimation of the size of an OTF2 trace produced, should Score-P be run using the current configuration. Here the trace size estimation could be also seen as a measure of overhead, since both are proportional to the number of recorded events. Additionally, the tool shows the distribution of the required trace size over call-path classes. From the report above one can see that the estimated trace size needed is equal to 1 GB in total or 256 MB per MPI rank, which is significant. From the breakdown per call-path class one can see that most of the overhead is due to user-level computations. In order to further localize the source of the overhead, `scorep-score` can print the breakdown of the buffer size on per-region basis:

```
% scorep-score -r scorep_bt-mz_W_4x4_sum/profile.cubex
  [...]
flt  type    max_tbc       time       % region
     ALL   256229936    5549.78   100.0 ALL
     USR   253654608    1758.27    31.7 USR
     OMP     5853120    3508.57    63.2 OMP
     COM      343344     183.09     3.3 COM
     MPI       93776      99.86     1.8 MPI
     USR    79176312     559.15    31.8 binvcrhs_
     USR    79176312     532.73    30.3 matvec_sub_
     USR    79176312     532.18    30.3 matmul_sub_
     USR     7361424      50.51     2.9 binvrhs_
     USR     7361424      56.35     3.2 lhsinit_
     USR     3206688      27.32     1.6 exact_solution_
     OMP     1550400    1752.20    99.7 !$omp implicit barrier
     OMP      257280       0.44     0.0 !$omp parallel @exch_qbc.f
     OMP      257280       0.61     0.0 !$omp parallel @exch_qbc.f
     OMP      257280       0.48     0.0 !$omp parallel @exch_qbc.f
```

The regions marked as USR type contribute to around 32% of the total time, however, much of that is very likely to be measurement overhead due to frequently-executed small routines. Therefore, it is highly recommended to remove these routines from measurements. This can be achieved by placing them into a filter file (please refer to Section 'Defining and testing a filter' for more details about filter file specification) as regions to be excluded from measurements. There is already a filter file prepared for BT-MZ which can be used:

```
% cat ../config/scorep.filt
SCOREP_REGION_NAMES_BEGIN EXCLUDE
binvcrhs*
matmul_sub*
matvec_sub*
exact_solution*
binvrhs*
lhs*init*
timer_*
```

One can use `scorep-score` once again to verify the effect of the filter file :

```
% scorep-score -f ../config/scorep.filt scorep_bt-mz_W_4x4_sum
Estimated aggregate size of event trace (total_tbc):
                                  20210360 bytes
Estimated requirements for largest trace buffer (max_tbc):
                                  6290888 bytes
(hint: When tracing set SCOREP_TOTAL_MEMORY > max_tbc to avoid
intermediate flushes
or reduce requirements using file listing names of USR regions
to be filtered.)
```

Now one can see that the trace size is reduced to just 20MB in total or 6MB per MPI rank. The regions filtered out will be marked with "+" in the left-most column of the per-region report.

7.5 Advanced summary measurement collection

After the filtering file is prepared to exclude the sources of the overhead, it is recommended to repeat summary collection, in order to obtain more precise measurements.

In order to specify the filter file to be used during measurements, the corresponding environment variable has to be set:

```
% export SCOREP_FILTERING_FILE=../config/scorep.filt
```

It is also recommended to adjust the experiment directory name for the new run:

```
% export SCOREP_EXPERIMENT_DIRECTORY=\
        scorep_bt-mz_W_4x4_sum_with_filter
```

Score-P also has a possibility to record hardware counters (see Section 'PAPI Hardware Performance Counters') and operating system resource usage (see Section 'Resource Usage Counters') in addition to default time and number of visits metrics. Hardware counters could be configured by setting Score-P environment variable SCO↩REP_METRIC_PAPI to the comma-separated list of PAPI events (other separator could be specified by setting SCOREP_METRIC_PAPI_SEP):

```
% export SCOREP_METRIC_PAPI=PAPI_TOT_INS,PAPI_FP_INS
```

Note

> The specified combination of the hardware events has to be valid, otherwise Score-P will abort execution. Please run `papi_avail` and `papi_native_avail` in order to get the list of the available PAPI generic and native events.

Operating system resource usage metrics are configured by setting the following variable:

```
% export SCOREP_METRIC_RUSAGE=ru_maxrss,ru_stime
```

Additionally Score-P can be configured to record only a subset of the mpi functions. This is achieved by setting SCOREP_MPI_ENABLE_GROUPS variable with a comma-separated list of sub-groups of MPI functions to be recorded (see Appendix 'MPI wrapper affiliation'):

```
% export SCOREP_MPI_ENABLE_GROUPS=cg,coll,p2p,xnonblock
```

In case performance of the CUDA code is of interest, Score-P can be configured to measure CUPTI metrics as follows (see Section 'CUDA Performance Measurement'):

```
% export SCOREP_CUDA_ENABLE=gpu,kernel,idle
```

In case performance of the OpenCL code is of interest, Score-P can be configured to measure OpenCL events as follows (see Section 'OpenCL Performance Measurement'):

```
% export SCOREP_OPENCL_ENABLE=api,kernel,memcpy
```

When the granularity offered by the automatic compiler instrumentation is not sufficient, one can use Score-↩P manual user instrumentation API (see Section 'Manual Region Instrumentation') for more fine-grained annotation of particular code segments. For example initialization code, solver or any other code segment of interest can be annotated.

In order to enable user instrumentation, an `--user` argument has to be passed to Score-P command prepending compiler and linker specification:

```
% MPIF77 = scorep --user mpif77
```

Below, the loop found on line ... in file ... is annotated as a user region:

```
#include "scorep/SCOREP_User.inc"
subroutine foo(...)
  ! Declarations
  SCOREP_USER_REGION_DEFINE( solve )
  ! Some code...
  SCOREP_USER_REGION_BEGIN( solve, "<solver>", \
                            SCOREP_USER_REGION_TYPE_LOOP )
  do i=1,100
    [...]
  end do
  SCOREP_USER_REGION_END( solve )
  ! Some more code...
end subroutine
```

This will appear as an additional region in the report.

BT-MZ has to be recompiled and relinked in order to complete instrumentation.

```
% make clean
% make bt-mz CLASS=W NPROCS=4
```

After applying advanced configurations described above, summary collection with Score-P can be started as usual:

```
% mpiexec -np 4 ./bt-mz_W.4
```

7.6   Advanced summary report examination

After execution is finished, one can use `scorep-score` tool to verify the effect of filtering:

```
% scorep-score scorep_bt-mz_W_4x4_sum_with_filter/profile.cubex
Estimated aggregate size of event trace (total_tbc):
                                      20210360 bytes
Estimated requirements for largest trace buffer (max_tbc):
                                       6290888 bytes
(hint: When tracing set SCOREP_TOTAL_MEMORY > max_tbc to avoid
intermediate flushes
or reduce requirements using file listing names of USR regions
to be filtered.)
flt type        max_tbc         time      % region
    ALL         6290888       241.77  100.0 ALL
    OMP         5853120       168.94   69.9 OMP
    COM          343344        35.57   14.7 COM
    MPI           93776        37.25   15.4 MPI
    USR             672         0.01    0.0 USR
```

The report above shows significant reduction in runtime (due to elimination of the overhead) not only in USR regions but also in MPI/OMP regions as well.

Now, the extended summary report can be interactively explored using CUBE:

```
% cube scorep_bt-mz_W_4x4_sum_with_filter/profile.cubex
```

or ParaProf:

```
% paraprof scorep_bt-mz_W_4x4_sum_with_filter/profile.cubex
```

7.7   Event trace collection and examination

After exploring extended summary report, additional insight into performance of BT-MZ can be gained by performing trace collection. In order to do so, Score-P has to be configured to perform tracing by setting corresponding variable to `true` and disabling profile generation:

```
% export SCOREP_ENABLE_TRACING=true
% export SCOREP_ENABLE_PROFILING=false
```

Additionally it is recommended to set a meaningful experiment directory name:

```
% export SCOREP_EXPERIMENT_DIRECTORY=scorep_bt-mz_W_4x4_trace
```

After BT-MZ execution is finished, a separate trace file per thread is written into the new experiment directory. In order to explore it, `Vampir` tool can be used:

```
% vampir scorep_bt-mz_W_4x4_trace/traces.otf2
```

Please consider that traces can become extremely large and unwieldy, because the size of the trace is proportional to number of processes/threads (width), duration (length) and detail (depth) of measurement. When the trace is too large to hold in the memory allocated by Score-P, flushes can happen. Unfortunately the resulting traces are of little value, since uncoordinated flushes result in cascades of distortion.

Traces should be written to a parallel file system, e.g., to `/work` or `/scratch` which are typically provided for this purpose.

Appendices

Appendix A

Score-P INSTALL

```
Score-P INSTALL GUIDE
=====================
```

This file describes how to configure, compile, and install the Score-P
measurement infrastructure. If you are not familiar with using the
configure scripts generated by GNU autoconf, read the "Generic
Installation Instructions" section below; then return here. Also,
make sure to carefully read and follow the platform-specific
installation notes (especially when building for the Intel Xeon Phi
platform).

Quick start
===========

In a nutshell, configuring, building, and installing Score-P can be as
simple as executing the shell commands

```
    mkdir _build
    cd _build
    ../configure --prefix=<installdir>
    make
    make install
```

If you don't specify --prefix, /opt/scorep will be used.

Depending on your system configuration and specific needs, the build
process can be customized as described below.

Configuration
=============

The configure script in this package tries to automatically determine
the platform for which Score-P will be compiled in order to provide
reasonable defaults for backend (i.e., compute node) compilers,
MPI compilers, and, in case of cross-compiling environments, frontend
(i.e., login node) compilers.

Depending on the environment it is possible to override the platform
defaults by using the following configure options:

```
  --with-machine-name=<default machine name>
```
                The default machine name used in profile and trace
                output. We suggest using a unique name, e.g., the
                fully qualified domain name. If not set, a name
                based on the detected platform is used. Can be
                overridden at measurement time by setting the
                environment variable SCOREP_MACHINE_NAME.

Score-P requires a full compiler suite with language support for C99,
C++11 and optionally Fortran 77 and Fortran 90. The following section

describes how to select supported compiler suits.

In non-cross-compiling environments, the compiler suite used to build
the backend parts can be specified explicitly if desired. On Linux
clusters it is currently recommended to use this option to select a
compiler suite other than GCC.

```
--with-nocross-compiler-suite=(gcc|ibm|intel|oneapi|nvhpc|pgi|clang| \
                               aocc|amdclang|cray)
          The compiler suite used to build this package in
          non-cross-compiling environments. Needs to be in $PATH.
          [Default: gcc]
```

Note: if you select 'pgi', CXX will be set to 'pgc++', which is
PGI's default C++ compiler. If you have a PGI compiler installation
prior to 16.1, you might want to use 'pgCC' instead if your MPI and
SHMEM compiler wrappers use this one. To select pgCC, please add
'CXX=pgCC' to your configure line.

Note that selecting 'cray' results in using the Cray compiler wrappers
cc, CC, ftn for compiling code, independently of the selected PrgEnv.

In cross-compiling environments, the compiler suite used to build the
frontend parts can be specified explicitly if desired.

```
--with-frontend-compiler-suite=(gcc|ibm|intel|oneapi|nvhpc|pgi|clang| \
                                aocc|amdclang|cray)
          The compiler suite used to build the frontend parts of
          this package in cross-compiling environments.  Needs to
          be in $PATH.
          [Default: gcc]
```

Note that selecting 'cray' results in using the Cray compiler wrappers
cc, CC, ftn for compiling code, independently of the selected PrgEnv.

The MPI compiler, if in $PATH, is usually autodetected. Users should
normally not need to specify which compiler will be invoked by the MPI
compiler wrapper; Score-P handles this automatically using the
appropriate command line arguments for the correct version of
MPI. While many MPI distributions also allow selecting the compiler
that the wrapper will invoke via environment variables, this mechanism
should not be used with Score-P.

Note that this also applies to instrumentation time. Here, users
should use the command line arguments along with the Score-P wrappers
for the MPI compiler wrappers: scorep-mpicc -cc=/path/to/specific/compiler ...

If there are several MPI compilers in $PATH the user should
select one using the configure option:

```
--with-mpi=(bullxmpi|cray|hp|ibmpoe|intel|intel2|intel3|intelpoe|lam| \
            mpibull2|mpich|mpich2|mpich3|mpich4|openmpi|openmpi3| \
            platform|scali|sgimpt|sgimptwrapper|spectrum|sun)
           The MPI compiler suite to build this package in non
           cross-compiling mode. Usually autodetected. Needs to be
           in $PATH.
```

Note that there is currently no consistency check if backend and MPI
compiler are from the same vendor. If they are not, linking problems
(undefined references) might occur.

Note that selecting 'cray' results in using the Cray compiler wrappers
cc, CC, ftn for compiling MPI code, independently of the selected PrgEnv.

The SHMEM compiler, if in $PATH, is usually autodetected. If there are
several SHMEM compilers in $PATH the user is requested to select one
using the configure option:

```
--with-shmem=(cray|openshmem|openmpi|openmpi3|sgimpt|sgimptwrapper|spectrum)
           The SHMEM compiler suite to build this package in
           non cross-compiling mode. Usually autodetected.
           Needs to be in $PATH.
```

Note that selecting 'cray' results in using the Cray compiler wrappers
cc, CC, ftn for compiling SHMEM code, independently of the selected PrgEnv.

If a particular system requires to use compilers different to those
Score-P currently supports, please edit the three files
build-config/common/platforms/platform-*-user-provided to your
needs and use the following configure option:

  --with-custom-compilers
                Customize compiler settings by 1. copying the three
                files
                <srcdir>/build-config/common/platforms/platform-*-user-provided
                to the directory where you run configure <builddir>,
                2. editing those files to your needs, and 3. running
                configure. Alternatively, edit the files under <srcdir>
                directly. Files in <builddir> take precedence. You are
                entering unsupported terrain. Namaste, and good luck!

On cross-compile systems the default frontend compiler is IBM XL for
the Blue Gene series and GCC on all other platforms. The backend
compilers will either be automatically selected by the platform
detection (IBM Blue Gene series) or by the currently loaded
environment modules (Cray X series). If you want to customize these
settings please use the configure option '--with-custom-compilers' as
described above.

Although this package comes with recent versions of the OTF2 and Cube
libraries as well as the OPARI2 instrumenter included, it is possible
to use existing installations instead. Here, the --without option
means 'without external installation', i.e., the component provided
with the tarball will be used:

  --with-otf2[=<otf2-bindir>]
                Use an already installed and compatible OTF2 library
                (v3.0 or newer). Provide path to otf2-config.
                Auto-detected if already in $PATH.
  --with-cubew[=<cubew-bindir>]
                Use an already installed and compatible cubew
                library (v4.8 or newer). Provide path to cubew-config.
                Auto-detected if already in $PATH.
  --with-cubelib[=<cubelib-bindir>]
                Use an already installed and compatible cubelib
                library (v4.8 or newer). Provide path to cubelib-config.
                Auto-detected if already in $PATH.
  --with-opari2[=<opari2-bindir>]
                Use an already installed and compatible OPARI2 (v2.0
                or newer). Provide path to opari2-config.
                Auto-detected if already in $PATH.

For the components otf2, cubew, cubelib, and opari2, the corresponding
--without-<component> or --with-<component>=no options will ignore the
<component>-config in $PATH but use the Score-P internal components.

To enable support for CUDA measurement via the CUPTI interface we
provide three configure options: the path to the CUDA runtime, the
path to the CUPTI library, and the path to the CUDA library. Usually
you just need to provide the path to the runtime via --with-libcudart;
CUPTI will be detected automatically in the extras/CUPTI subdirectory
and the system libcuda will be used. This usually works for GNU and
Intel compilers:

  --with-libcudart=<Path to libcudart installation>
                If you want to build scorep with libcudart but do not
                have a libcudart in a standard location then you need
                to explicitly specify the directory where it is
                installed. On non-cross-compile systems we search the
                system include and lib paths per default [yes], on
                cross-compile systems however,you have to specify a
                path [no]. --with-libcudart is a shorthand for
                --with-libcudart-include=<Path/include> and
                --with-libcudart-lib=<Path/lib>. If these shorthand
                assumptions are not correct, you can use the explicit
                include and lib options directly.

```
   --with-libcudart-include=<Path to libcudart headers: cuda.h cuda_runtime_api.h>
   --with-libcudart-lib=<Path to libcudart libraries>
```

For PGI compilers since version 16.10 you need at least two options,
--with-libcudart pointing to the <PGI CUDA runtime> and
--with-libcupti pointing to <NVIDIA CUDA runtime>/extras/CUPTI. Note
that CUDA/CUPTI support in Score-P with PGI compilers prior to 16.10
does not work (unless you modify the NVIDIA CUDA runtime headers).

```
   --with-libcupti=(yes|no|<Path to libcupti installation>)
               If you want to build with libcupti support but do
               not have a libcupti in a standard location, you need
               to explicitly specify the directory where it is
               installed. On non-cross-compile systems we search
               the system include and lib paths per default [yes];
               on cross-compile systems, however, you have to
               specify a path [no]. --with-libcupti is a shorthand
               for --with-libcupti-include=<Path/include> and
               --with-libcupti-lib=<Path/lib>. If these shorthand
               assumptions are not correct, you can use the
               explicit include and lib options directly.
   --with-libcupti-include=<Path to libcupti headers: cupti.h>
   --with-libcupti-lib=<Path to libcupti libraries>
```

If you want to use a CUDA library different from the system one you
need to specify its location via:

```
   --with-libcuda=<Path to libcuda installation>
               Usually not needed, specifying --with-libcudart should
               be fine!
               If you want to build scorep with libcuda but do not
               have a libcuda in a standard location then you need to
               explicitly specify the directory where it is
               installed. On non-cross-compile systems we search the
               system include and lib paths per default [yes], on
               cross-compile systems however,you have to specify a
               path [no]. --with-libcuda is a shorthand for
               --with-libcuda-include=<Path/include> and
               --with-libcuda-lib=<Path/lib>. If these shorthand
               assumptions are not correct, you can use the explicit
               include and lib options directly.
   --with-libcuda-include=<Path to libcuda headers>
   --with-libcuda-lib=<Path to libcuda libraries>
```

If libnvidia-ml, which is used to determine correct device-IDs in a
multi-GPU/multi-process environment, isn't detected automatically, you
can specify its location via:

```
   --with-libnvidia-ml=(yes|no|<Path to libnvidia-ml installation>)
               If you want to build with libnvidia-ml support but
               do not have a libnvidia-ml in a standard location,
               you need to explicitly specify the directory where
               it is installed. On non-cross-compile systems we
               search the system include and lib paths per default
               [yes]; on cross-compile systems, however, you have
               to specify a path [no]. --with-libnvidia-ml is a
               shorthand for
               --with-libnvidia-ml-include=<Path/include> and
               --with-libnvidia-ml-lib=<Path/lib>. If these
               shorthand assumptions are not correct, you can use
               the explicit include and lib options directly.
   --with-libnvidia-ml-include=<Path to libnvidia-ml headers: nvml.h>
   --with-libnvidia-ml-lib=<Path to libnvidia-ml libraries>
```

To enable support for HIP measurement via ROCm configure with
--with-rocm=<path-to-ROCm-installation>. If an argument is missing the value
from $ROCM_PATH is used.

```
   --with-rocm=<path-to-ROCm-installation>
               Location of the ROCm installation (e.g., /opt/rocm)
```

If your installation differs from the default layout, you can provide paths via
the following configure arguments:

```
--with-libamdhip64=(yes|no|<Path to libamdhip64 installation>)
              If you want to build with libamdhip64 support but do
              not have a libamdhip64 in a standard location, you
              need to explicitly specify the directory where it is
              installed. On non-cross-compile systems we search
              the system include and lib paths per default [yes];
              on cross-compile systems, however, you have to
              specify a path [no]. --with-libamdhip64 is a
              shorthand for
              --with-libamdhip64-include=<Path/include> and
              --with-libamdhip64-lib=<Path/lib>. If these
              shorthand assumptions are not correct, you can use
              the explicit include and lib options directly.
--with-libamdhip64-include=<Path to libamdhip64 headers: hip/hip_runtime.h>
--with-libamdhip64-lib=<Path to libamdhip64 libraries>
--with-libroctracer64=(yes|no|<Path to libroctracer64 installation>)
              If you want to build with libroctracer64 support but
              do not have a libroctracer64 in a standard location,
              you need to explicitly specify the directory where
              it is installed. On non-cross-compile systems we
              search the system include and lib paths per default
              [yes]; on cross-compile systems, however, you have
              to specify a path [no]. --with-libroctracer64 is a
              shorthand for
              --with-libroctracer64-include=<Path/include> and
              --with-libroctracer64-lib=<Path/lib>. If these
              shorthand assumptions are not correct, you can use
              the explicit include and lib options directly.
--with-libroctracer64-include=<Path to libroctracer64 headers: roctracer_hip.h>
--with-libroctracer64-lib=<Path to libroctracer64 libraries>
--with-librocm_smi64=(yes|no|<Path to librocm_smi64 installation>)
              If you want to build with librocm_smi64 support but
              do not have a librocm_smi64 in a standard location,
              you need to explicitly specify the directory where
              it is installed. On non-cross-compile systems we
              search the system include and lib paths per default
              [yes]; on cross-compile systems, however, you have
              to specify a path [no]. --with-librocm_smi64 is a
              shorthand for
              --with-librocm_smi64-include=<Path/include> and
              --with-librocm_smi64-lib=<Path/lib>. If these
              shorthand assumptions are not correct, you can use
              the explicit include and lib options directly.
--with-librocm_smi64-include=<Path to librocm_smi64 headers: rocm_smi/rocm_smi.h>
--with-librocm_smi64-lib=<Path to librocm_smi64 libraries>

Options to further specify which features and external packages should
be used to build Score-P are as follows:

  --enable-platform-mic   Force build for Intel Xeon Phi co-processors
                          [no].  This option is only needed for Xeon
                          Phi co-processors, like the Knights Corner
                          (KNC). It is not needed for self-hosted Xeon
                          Phis, like the Knights Landing (KNL); for these
                          chips no special treatment is required.
  --enable-experimental-platform
                          Enables builds on platforms that are not officially
                          supported (currently macOS and MinGW).
  --enable-debug          activate internal debug output [no]
  --enable-shared[=PKGS]  build shared libraries [default=no]
  --enable-static[=PKGS]  build static libraries [default=yes]
  --enable-backend-test-runs
              Enable execution of tests during 'make check' [no]
              (does not affect building of tests, though). If
              disabled, the files 'check-file-*' and/or
              'skipped_tests' listing the tests are generated in the
              corresponding build directory.
  --enable-cuda
              Enable or disable support for CUDA. Fails if support
              cannot be satisfied but was requested.
  --enable-openacc
              Enable or disable support for OpenACC. (defaults to yes)
```

```
--enable-default=<comma-separated list>
              Make experimental feature(s) the default (if established
              alternative exists).
              Valid value: ompt
              List of valid values may change without deprecation
              notice.
--disable-gcc-plugin
              Disable support for the GCC plug-in
              instrumentation. Default is to determine support
              automatically. This disables it by request and fails
              if support cannot be satisfied but was requested.
--with-pdt=<path-to-binaries>
              Specifies the path to the program database toolkit
              (PDT) binaries, e.g., cparse.
--with-sionlib[=<sionlib-bindir>]
              Use an already installed sionlib. Provide path to
              sionconfig. Auto-detected if already in $PATH. This
              option is not used by Score-P itself but passed to an
              internal OTF2.
--with-papi-header=<path-to-papi.h>
              If papi.h is not installed in the default location,
              specify the dirname where it can be found.
--with-papi-lib=<path-to-libpapi.*>
              If libpapi.* is not installed in the default location,
              specify the dirname where it can be found.
--with-libunwind=(yes|no|<Path to libunwind installation>)
              If you want to build with libunwind support but do
              not have a libunwind in a standard location, you
              need to explicitly specify the directory where it is
              installed. On non-cross-compile systems we search
              the system include and lib paths per default [yes];
              on cross-compile systems, however, you have to
              specify a path [no]. --with-libunwind is a shorthand
              for --with-libunwind-include=<Path/include> and
              --with-libunwind-lib=<Path/lib>. If these shorthand
              assumptions are not correct, you can use the
              explicit include and lib options directly.
--with-libunwind-include=<Path to libunwind headers: libunwind.h>
--with-libunwind-lib=<Path to libunwind libraries>
--with-openacc-include=<path-to-openacc.h>
              If openacc.h is not installed in the default
              location, specify the directory where it can be
              found.
--with-openacc-prof-include=<path-to-acc_prof.h>
              If acc_prof.h is not installed in the default
              location, specify the directory where it can be
              found.
--with-libOpenCL=(yes|no|<Path to libOpenCL installation>)
              If you want to build with libOpenCL support but do
              not have a libOpenCL in a standard location, you
              need to explicitly specify the directory where it is
              installed. On non-cross-compile systems we search
              the system include and lib paths per default [yes];
              on cross-compile systems, however, you have to
              specify a path [no]. --with-libOpenCL is a shorthand
              for --with-libOpenCL-include=<Path/include> and
              --with-libOpenCL-lib=<Path/lib>. If these shorthand
              assumptions are not correct, you can use the
              explicit include and lib options directly.
--with-libOpenCL-include=<Path to libOpenCL headers: CL/cl.h OpenCL/opencl.h>
--with-libOpenCL-lib=<Path to libOpenCL libraries>
--with-libpmi=(yes|no|<Path to libpmi installation>)
              If you want to build with libpmi support but do not
              have a libpmi in a standard location, you need to
              explicitly specify the directory where it is
              installed. On non-cross-compile systems we search
              the system include and lib paths per default [yes];
              on cross-compile systems, however, you have to
              specify a path [no]. --with-libpmi is a shorthand
              for --with-libpmi-include=<Path/include> and
              --with-libpmi-lib=<Path/lib>. If these shorthand
              assumptions are not correct, you can use the
              explicit include and lib options directly.
```

```
   --with-libpmi-include=<Path to libpmi headers: pmi.h>
   --with-libpmi-lib=<Path to libpmi libraries>
   --with-librca=(yes|no|<Path to librca installation>)
                 If you want to build with librca support but do not
                 have a librca in a standard location, you need to
                 explicitly specify the directory where it is
                 installed. On non-cross-compile systems we search
                 the system include and lib paths per default [yes];
                 on cross-compile systems, however, you have to
                 specify a path [no]. --with-librca is a shorthand
                 for --with-librca-include=<Path/include> and
                 --with-librca-lib=<Path/lib>. If these shorthand
                 assumptions are not correct, you can use the
                 explicit include and lib options directly.
   --with-librca-include=<Path to librca headers>
   --with-librca-lib=<Path to librca libraries>
   --with-liblustreapi[=yes|no|<path to liblustreapi installation>]
                 Try to use liblustreapi. Defaults to [yes] on
                 non-cross-compile systems and expects library and
                 headers to be found in system locations. Defaults to
                 [no] on cross-compile systems. Provide
                 liblustreapi's installation prefix [path] to
                 override the defaults. --with-liblustreapi=<path> is
                 a shorthand for
                 --with-liblustreapi-include=<path/include> and
                 --with-liblustreapi-lib=<path/(lib64|lib)>. If this
                 is not the case, use the explicit options directly
                 or provide paths via LIBLUSTREAPI_LIB and
                 LIBLUSTREAPI_INCLUDE.
   --with-liblustreapi-include=<Path to liblustreapi headers: lustre/lustreapi.h>
   --with-liblustreapi-lib=<Path to liblustreapi libraries>
   --with-libbfd[=yes|download|<path to libbfd installation>]
                 A shared or PIC libbfd is required. Option defaults
                 to [yes] on non-cross-compile systems and expects
                 library and headers to be found in system locations.
                 Provide libbfd's installation prefix [path] to
                 override this default. --with-libbfd=<path> is a
                 shorthand for --with-libbfd-include=<path/include>
                 and --with-libbfd-lib=<path/(lib64|lib)>. If this
                 not the case, use the explicit options directly or
                 provide paths via LIBBFD_LIB and LIBBFD_INCLUDE. Use
                 [download] to automatically obtain and use libbfd
                 via external tarball.
   --with-libbfd-include=<Path to libbfd headers: bfd.h>
   --with-libbfd-lib=<Path to libbfd libraries>
   Please note that if a Score-P installation uses libbfd, the Score-P
   instrumented binaries may only be distributed under GPLv3 licensing.
   --with-llvm[=<llvm-bindir>]
                 Use an already installed LLVM, including libclang as we
                 need libclang for library-wrapping. Provide path to
                 llvm-config. Auto-detected if already in $PATH.

Instead of passing command-line options to the 'configure' script, the package
configuration can also be influenced by setting the following environment
variables:

  CC          C compiler command
  CFLAGS      C compiler flags
  LDFLAGS     linker flags, e.g. -L<lib dir> if you have libraries in a
              nonstandard directory <lib dir>
  LIBS        libraries to pass to the linker, e.g. -l<library>
  CPPFLAGS    (Objective) C/C++ preprocessor flags, e.g. -I<include dir> if
              you have headers in a nonstandard directory <include dir>
  LT_SYS_LIBRARY_PATH
              User-defined run-time library search path.
  CPP         C preprocessor
  CXX         C++ compiler command
  CXXFLAGS    C++ compiler flags
  CXXCPP      C++ preprocessor
  CCAS        assembler compiler command (defaults to CC)
  CCASFLAGS   assembler compiler flags (defaults to CFLAGS)
  CXXCPP      C++ preprocessor
  F77         Fortran 77 compiler command
```

```
FFLAGS      Fortran 77 compiler flags
FC          Fortran compiler command
FCFLAGS     Fortran compiler flags
CC_FOR_BUILD
            C compiler command for the frontend build
CXX_FOR_BUILD
            C++ compiler command for the frontend build
F77_FOR_BUILD
            Fortran 77 compiler command for the frontend build
FC_FOR_BUILD
            Fortran compiler command for the frontend build
CPPFLAGS_FOR_BUILD
            (Objective) C/C++ preprocessor flags for the frontend build,
            e.g. -I<include dir> if you have headers in a nonstandard
            directory <include dir>
CFLAGS_FOR_BUILD
            C compiler flags for the frontend build
CXXFLAGS_FOR_BUILD
            C++ compiler flags for the frontend build
FFLAGS_FOR_BUILD
            Fortran 77 compiler flags for the frontend build
FCFLAGS_FOR_BUILD
            Fortran compiler flags for the frontend build
LDFLAGS_FOR_BUILD
            linker flags for the frontend build, e.g. -L<lib dir> if you
            have libraries in a nonstandard directory <lib dir>
LIBS_FOR_BUILD
            libraries to pass to the linker for the frontend build, e.g.
            -l<library>
CC_FOR_BUILD_LIBWRAP
            C compiler used for building library wrapper generator. Defaults
            to CC from llvm-config or gcc.
CXX_FOR_BUILD_LIBWRAP
            C++ compiler used for building library wrapper generator.
            Defaults to CXX from llvm-config or g++.
CPPFLAGS_FOR_BUILD_LIBWRAP
            C preprocesser flags for building library wrapper generator.
            Will be amended with 'llvm-config --cppflags'.
CFLAGS_FOR_BUILD_LIBWRAP
            C compiler flags for building library wrapper generator.
CXXFLAGS_FOR_BUILD_LIBWRAP
            C++ compiler flags for building library wrapper generator.
LDFLAGS_FOR_BUILD_LIBWRAP
            Linker flags for building library wrapper generator.
LIBS_FOR_BUILD_LIBWRAP
            Libraries for building library wrapper generator. Will be
            amended with '-lclang'.
MPICC       MPI C compiler command
MPICXX      MPI C++ compiler command
MPIF77      MPI Fortran 77 compiler command
MPIFC       MPI Fortran compiler command
MPI_CPPFLAGS
            MPI (Objective) C/C++ preprocessor flags, e.g. -I<include dir>
            if you have headers in a nonstandard directory <include dir>
MPI_CFLAGS  MPI C compiler flags
MPI_CXXFLAGS
            MPI C++ compiler flags
MPI_FFLAGS  MPI Fortran 77 compiler flags
MPI_FCFLAGS MPI Fortran compiler flags
MPI_LDFLAGS
            MPI linker flags, e.g. -L<lib dir> if you have libraries in a
            nonstandard directory <lib dir>
MPI_LIBS    MPI libraries to pass to the linker, e.g. -l<library>
SHMEMCC     SHMEM C compiler command
SHMEMCXX    SHMEM C++ compiler command
SHMEMF77    SHMEM Fortran 77 compiler command
SHMEMFC     SHMEM Fortran compiler command
SHMEM_CPPFLAGS
            SHMEM (Objective) C/C++ preprocessor flags, e.g. -I<include dir>
            if you have headers in a nonstandard directory <include dir>
SHMEM_CFLAGS
            SHMEM C compiler flags
SHMEM_CXXFLAGS
```

```
                SHMEM C++ compiler flags
SHMEM_FFLAGS
                SHMEM Fortran 77 compiler flags
SHMEM_FCFLAGS
                SHMEM Fortran compiler flags
SHMEM_LDFLAGS
                SHMEM linker flags, e.g. -L<lib dir> if you have libraries in a
                nonstandard directory <lib dir>
SHMEM_LIBS  SHMEM libraries to pass to the linker, e.g. -l<library>
SHMEM_LIB_NAME
                name of the SHMEM library
SHMEM_NAME  name of the implemented SHMEM specification
CFLAGS_FOR_BUILD_SCORE
                C compiler flags for building scorep-score. Please note
                that CC comes from cubelib.
CXXFLAGS_FOR_BUILD_SCORE
                C++ compiler flags for building scorep-score. Please
                note that CXX comes from cubelib.
LDFLAGS_FOR_BUILD_SCORE
                Linker flags for building scorep-score.
PTHREAD_CFLAGS
                CFLAGS used to compile Pthread programs
PTHREAD_LIBS
                LIBS used to link Pthread programs
RUNTIME_MANAGEMENT_TIMINGS
                Whether to activate time measurements for Score-P's
                SCOREP_InitMeasurement() and scorep_finalize() functions.
                Activation values are '1', 'yes', and 'true'. For developer use.
PAPI_INC    Include path to the papi.h header.
PAPI_LIB    Library path to the papi library.
LIBUNWIND_INCLUDE
                Path to libunwind headers: libunwind.h
LIBUNWIND_LIB
                Path to libunwind libraries.
LIBCUDART_INCLUDE
                Path to libcudart headers: cuda.h cuda_runtime_api.h
LIBCUDART_LIB
                Path to libcudart libraries.
LIBCUDA_LIB Path to libcuda libraries.
LIBCUPTI_INCLUDE
                Path to libcupti headers: cupti.h
LIBCUPTI_LIB
                Path to libcupti libraries.
LIBNVIDIA_ML_INCLUDE
                Path to libnvidia-ml headers: nvml.h
LIBNVIDIA_ML_LIB
                Path to libnvidia-ml libraries.
LIBOPENCL_INCLUDE
                Path to libOpenCL headers: CL/cl.h OpenCL/opencl.h
LIBOPENCL_LIB
                Path to libOpenCL libraries.
LIBLUSTREAPI_INCLUDE
                Path to liblustreapi headers: lustre/lustreapi.h
LIBLUSTREAPI_LIB
                Path to liblustreapi libraries.
LIBBFD_INCLUDE
                Path to libbfd headers: bfd.h. Superseded by --with-libbfd
                variants.
LIBBFD_LIB  Path to libbfd libraries. Superseded by --with-libbfd variants.
OPENACC_INCLUDE
                Path to openacc.h header.
OPENACC_PROFILING_INCLUDE
                Path to acc_prof.h header.
ROCM_PATH   Base directory of the AMD ROCm installation
LIBAMDHIP64_INCLUDE
                Path to libamdhip64 headers: hip/hip_runtime.h
LIBAMDHIP64_LIB
                Path to libamdhip64 libraries.
LIBROCTRACER64_INCLUDE
                Path to libroctracer64 headers: roctracer_hip.h
LIBROCTRACER64_LIB
                Path to libroctracer64 libraries.
LIBROCM_SMI64_INCLUDE
```

```
                   Path to librocm_smi64 headers: rocm_smi/rocm_smi.h
  LIBROCM_SMI64_LIB
                   Path to librocm_smi64 libraries.
  LIBPMI_INCLUDE
                   Path to libpmi headers: pmi.h
  LIBPMI_LIB   Path to libpmi libraries.
  LIBRCA_LIB   Path to librca libraries.
  BACKEND_TEST_RUNS_NON_RECURSIVE
                   Whether to prevent passing --enable-backend-test-runs to
                   sub-packages. Activation values are '1', 'yes', and 'true'.
```

```
Building & Installing
=====================
```

Before building Score-P, carefully check whether the configuration summary
printed by the configure script matches your expectations (i.e., whether MPI
and/or OpenMP support is correctly enabled/disabled, external libraries are
used, etc). If everything is OK, Score-P can be built and installed using

```
        make
        make install
```

Note that parallel builds (i.e., using 'make -j <n>') are fully supported.

```
                   Platform-specific Instructions
                   ==============================
```

```
GNU Compiler Plug-In
====================
```

   On some system the necessary header files, for compiling with support for
the GNU Compiler plug-in instrumentation, are not installed by default. Therefore
an extra package needs to be installed.

On Debian and it's derivatives the package is called:

```
    gcc-<version>-plugin-dev
```

On Fedora and it's derivatives the package is called:

```
    gcc-plugin-devel
```

```
Intel Xeon Phi (aka. MIC) co-processors
=======================================
```

   [Note: The following instructions only apply to Intel Xeon Phi
 co-processors, like the Knights Corner (KNC). They do not apply to
 self-hosted Xeon Phis, like the Knights Landing (KNL); for these
 chips no special treatment is required.]

   Building Score-P for Intel Xeon Phi co-processors requires some
extra care, and in some cases two installations into the same
location. Therefore, we strongly recommend to strictly follow the
procedure as described below.

  1. Ensure that Intel compilers and Intel MPI (if desired) are
     installed and available in $PATH, and that the Intel Manycore
     Platform Software Stack (MPSS) is installed.

  2. Configure Score-P to use the MIC platform:

```
     mkdir _build-mic
     cd _build-mic
     ../configure --enable-platform-mic [other options, e.g., '--prefix']
```

  3. Build and install:

```
     make; make install
```

   In case a native MIC-only installation serves your needs, that's
it. However, if the installation should also support instrumentation

and measurement of host code, a second installation *on top* of the
just installed one is required:

  4. Create a new build directory for the host build:

        cd ..
        mkdir _build-host
        cd _build-host

  5. Reconfigure for the host using *identical directory options* (e.g.,
     '--prefix' or '--bindir') as in step 2:

        ../configure [other options as used in step 2]

     This will automatically detect the already existing native MIC
     build and enable the required support in the host tools. On
     non-cross-compile systems (e.g., typical Linux clusters), make
     sure to explicitly select Intel compiler support by passing
     '--with-nocross-compiler-suite=intel' to the configure script.

  6. Build and install:

        make; make install

   Note that this approach also works with VPATH builds (even with two
separate build directories) as long as the same options defining directory
locations are passed in steps 2 and 5.

User Library Wrapping
=====================

   To enable the user library wrapping feature, Score-P needs the LLVM compiler
infrastructure. For one the 'llvm-config' program is needed, and also `libclang`
and it's developer packages. The clang/clang++ compilers are not strictly
needed, but will be used when available.

On Debian and it's derivatives the packages are called:

    llvm-dev
    libclang-dev

On Fedora and it's derivatives the packages are called:

    llvm-devel
    clang-devel

Or the pre-built packages at:

    http://llvm.org/releases/download.html

                    Generic Installation Instructions
                    =================================

Basic Installation
==================

   Briefly, the shell commands './configure; make; make install' should
configure, build, and install this package. The following more-detailed
instructions are generic; see the section above for instructions
specific to this package. Some packages provide this 'INSTALL' file but
do not implement all of the features documented below. The lack of an
optional feature in a given package is not necessarily a bug.

   The 'configure' shell script attempts to guess correct values for
various system-dependent variables used during compilation. It uses

those values to create a 'Makefile' in each directory of the package.
It may also create one or more '.h' files containing system-dependent
definitions. Finally, it creates a shell script 'config.status' that
you can run in the future to recreate the current configuration, and a
file 'config.log' containing compiler output (useful mainly for
debugging 'configure').

   It can also use an optional file (typically called 'config.cache'
and enabled with '--cache-file=config.cache' or simply '-C') that saves
the results of its tests to speed up reconfiguring. Caching is
disabled by default to prevent problems with accidental use of stale
cache files.

   If you need to do unusual things to compile the package, please try
to figure out how 'configure' could check whether to do them, and mail
diffs or instructions to support@score-p.org so they can be considered
for the next release. If you are using the cache, and at some point
'config.cache' contains results you don't want to keep, you may remove
or edit it.

   The file 'configure.ac' (or 'configure.in') is used to create
'configure' by a program called 'autoconf'. You need 'configure.ac' if
you want to change it or regenerate 'configure' using a newer version
of 'autoconf'.

   The simplest way to compile this package is:

  1. 'cd' to the directory containing the package's source code and type
     './configure' to configure the package for your system.

     Running 'configure' might take a while. While running, it prints
     some messages telling which features it is checking for.

  2. Type 'make' to compile the package.

  3. Optionally, type 'make check' to run any self-tests that come with
     the package, generally using the just-built uninstalled binaries.

  4. Type 'make install' to install the programs and any data files and
     documentation. When installing into a prefix owned by root, it is
     recommended that the package be configured and built as a regular
     user, and only the 'make install' phase executed with root
     privileges.

  5. Optionally, type 'make installcheck' to repeat any self-tests, but
     this time using the binaries in their final installed location.
     This target does not install anything. Running this target as a
     regular user, particularly if the prior 'make install' required
     root privileges, verifies that the installation completed
     correctly.

  6. You can remove the program binaries and object files from the
     source code directory by typing 'make clean'. To also remove the
     files that 'configure' created (so you can compile the package for
     a different kind of computer), type 'make distclean'. There is
     also a 'make maintainer-clean' target, but that is intended mainly
     for the package's developers. If you use it, you may have to get
     all sorts of other programs in order to regenerate files that came
     with the distribution.

  7. Often, you can also type 'make uninstall' to remove the installed
     files again. In practice, not all packages have tested that
     uninstallation works correctly, even though it is required by the
     GNU Coding Standards.

  8. Some packages, particularly those that use Automake, provide 'make
     distcheck', which can by used by developers to test that all other
     targets like 'make install' and 'make uninstall' work correctly.
     This target is generally not run by end users.

Compilers and Options
=====================

Some systems require unusual options for compilation or linking that the 'configure' script does not know about. Run './configure --help' for details on some of the pertinent environment variables.

   You can give 'configure' initial values for configuration parameters by setting variables in the command line or in the environment. Here is an example:

      ./configure CC=c99 CFLAGS=-g LIBS=-lposix

   *Note Defining Variables::, for more details.

Compiling For Multiple Architectures
====================================

   You can compile the package for more than one kind of computer at the same time, by placing the object files for each architecture in their own directory. To do this, you can use GNU 'make'. 'cd' to the directory where you want the object files and executables to go and run the 'configure' script. 'configure' automatically checks for the source code in the directory that 'configure' is in and in '..'. This is known as a "VPATH" build.

   With a non-GNU 'make', it is safer to compile the package for one architecture at a time in the source code directory. After you have installed the package for one architecture, use 'make distclean' before reconfiguring for another architecture.

   On MacOS X 10.5 and later systems, you can create libraries and executables that work on multiple system types--known as "fat" or "universal" binaries--by specifying multiple '-arch' options to the compiler but only a single '-arch' option to the preprocessor. Like this:

      ./configure CC="gcc -arch i386 -arch x86_64 -arch ppc -arch ppc64" \
                  CXX="g++ -arch i386 -arch x86_64 -arch ppc -arch ppc64" \
                  CPP="gcc -E" CXXCPP="g++ -E"

   This is not guaranteed to produce working output in all cases, you may have to build one architecture at a time and combine the results using the 'lipo' tool if you have problems.

Installation Names
==================

   By default, 'make install' installs the package's commands under '/usr/local/bin', include files under '/usr/local/include', etc. You can specify an installation prefix other than '/usr/local' by giving 'configure' the option '--prefix=PREFIX', where PREFIX must be an absolute file name.

   You can specify separate installation prefixes for architecture-specific files and architecture-independent files. If you pass the option '--exec-prefix=PREFIX' to 'configure', the package uses PREFIX as the prefix for installing programs and libraries. Documentation and other data files still use the regular prefix.

   In addition, if you use an unusual directory layout you can give options like '--bindir=DIR' to specify different values for particular kinds of files. Run 'configure --help' for a list of the directories you can set and what kinds of files go in them. In general, the default for these options is expressed in terms of '${prefix}', so that specifying just '--prefix' will affect all of the other directory specifications that were not explicitly provided.

   The most portable way to affect installation locations is to pass the correct locations to 'configure'; however, many packages provide one or both of the following shortcuts of passing variable assignments to the 'make install' command line to change installation locations without having to reconfigure or recompile.

   The first method involves providing an override variable for each affected directory. For example, 'make install

prefix=/alternate/directory' will choose an alternate location for all
directory configuration variables that were expressed in terms of
`${prefix}'.  Any directories that were specified during `configure',
but not in terms of `${prefix}', must each be overridden at install
time for the entire installation to be relocated. The approach of
makefile variable overrides for each directory variable is required by
the GNU Coding Standards, and ideally causes no recompilation.
However, some platforms have known limitations with the semantics of
shared libraries that end up requiring recompilation when using this
method, particularly noticeable in packages that use GNU Libtool.

   The second method involves providing the `DESTDIR' variable. For
example, `make install DESTDIR=/alternate/directory' will prepend
`/alternate/directory' before all installation names. The approach of
`DESTDIR' overrides is not required by the GNU Coding Standards, and
does not work on platforms that have drive letters. On the other hand,
it does better at avoiding recompilation issues, and works well even
when some directory options were not specified in terms of `${prefix}'
at `configure' time.

Optional Features
=================

   If the package supports it, you can cause programs to be installed
with an extra prefix or suffix on their names by giving `configure' the
option `--program-prefix=PREFIX' or `--program-suffix=SUFFIX'.

   Some packages pay attention to `--enable-FEATURE' options to
`configure', where FEATURE indicates an optional part of the package.
They may also pay attention to `--with-PACKAGE' options, where PACKAGE
is something like `gnu-as' or `x' (for the X Window System).

   For packages that use the X Window System, `configure' can usually
find the X include and library files automatically, but if it doesn't,
you can use the `configure' options `--x-includes=DIR' and
`--x-libraries=DIR' to specify their locations.

   Some packages offer the ability to configure how verbose the
execution of `make' will be.  For these packages, running `./configure
--enable-silent-rules' sets the default to minimal output, which can be
overridden with `make V=1'; while running `./configure
--disable-silent-rules' sets the default to verbose, which can be
overridden with `make V=0'.

Particular systems
==================

   On HP-UX, the default C compiler is not ANSI C compatible. If GNU
CC is not installed, it is recommended to use the following options in
order to use an ANSI C compiler:

     ./configure CC="cc -Ae -D_XOPEN_SOURCE=500"

and if that doesn't work, install pre-built binaries of GCC for HP-UX.

   On OSF/1 a.k.a. Tru64, some versions of the default C compiler cannot
parse its `<wchar.h>' header file.  The option `-nodtk' can be used as
a workaround. If GNU CC is not installed, it is therefore recommended
to try

     ./configure CC="cc"

and if that doesn't work, try

     ./configure CC="cc -nodtk"

   On Solaris, don't put `/usr/ucb' early in your `PATH'. This
directory contains several dysfunctional programs; working variants of
these programs are available in `/usr/bin'.  So, if you need `/usr/ucb'
in your `PATH', put it _after_ `/usr/bin'.

   On Haiku, software installed for all users goes in `/boot/common',
not `/usr/local'. It is recommended to use the following options:

```
    ./configure --prefix=/boot/common
```

Specifying the System Type
==========================

   There may be some features 'configure' cannot figure out
automatically, but needs to determine by the type of machine the package
will run on. Usually, assuming the package is built to be run on the
same architectures, 'configure' can figure that out, but if it prints
a message saying it cannot guess the machine type, give it the
'--build=TYPE' option. TYPE can either be a short name for the system
type, such as 'sun4', or a canonical name which has the form:

     CPU-COMPANY-SYSTEM

where SYSTEM can have one of these forms:

     OS
     KERNEL-OS

   See the file 'config.sub' for the possible values of each field. If
'config.sub' isn't included in this package, then this package doesn't
need to know the machine type.

   If you are _building_ compiler tools for cross-compiling, you should
use the option '--target=TYPE' to select the type of system they will
produce code for.

   If you want to _use_ a cross compiler, that generates code for a
platform different from the build platform, you should specify the
"host" platform (i.e., that on which the generated programs will
eventually be run) with '--host=TYPE'.

Sharing Defaults
================

   If you want to set default values for 'configure' scripts to share,
you can create a site shell script called 'config.site' that gives
default values for variables like 'CC', 'cache_file', and 'prefix'.
'configure' looks for 'PREFIX/share/config.site' if it exists, then
'PREFIX/etc/config.site' if it exists. Or, you can set the
'CONFIG_SITE' environment variable to the location of the site script.
A warning: not all 'configure' scripts look for a site script.

Defining Variables
==================

   Variables not defined in a site shell script can be set in the
environment passed to 'configure'. However, some packages may run
configure again during the build, and the customized values of these
variables may be lost. In order to avoid this problem, you should set
them in the 'configure' command line, using 'VAR=value'. For example:

     ./configure CC=/usr/local2/bin/gcc

causes the specified 'gcc' to be used as the C compiler (unless it is
overridden in the site shell script).

Unfortunately, this technique does not work for 'CONFIG_SHELL' due to
an Autoconf bug. Until the bug is fixed you can use this workaround:

     CONFIG_SHELL=/bin/bash /bin/bash ./configure CONFIG_SHELL=/bin/bash

'configure' Invocation
======================

   'configure' recognizes the following options to control how it
operates.

'--help'
'-h'
     Print a summary of all of the options to 'configure', and exit.
```

```
'--help=short'
'--help=recursive'
     Print a summary of the options unique to this package's
     'configure', and exit.  The 'short' variant lists options used
     only in the top level, while the 'recursive' variant lists options
     also present in any nested packages.

'--version'
'-V'
     Print the version of Autoconf used to generate the 'configure'
     script, and exit.

'--cache-file=FILE'
     Enable the cache: use and save the results of the tests in FILE,
     traditionally 'config.cache'.  FILE defaults to '/dev/null' to
     disable caching.

'--config-cache'
'-C'
     Alias for '--cache-file=config.cache'.

'--quiet'
'--silent'
'-q'
     Do not print messages saying which checks are being made.  To
     suppress all normal output, redirect it to '/dev/null' (any error
     messages will still be shown).

'--srcdir=DIR'
     Look for the package's source code in directory DIR.  Usually
     'configure' can determine that directory automatically.

'--prefix=DIR'
     Use DIR as the installation prefix.  *note Installation Names::
     for more details, including other options available for fine-tuning
     the installation locations.

'--no-create'
'-n'
     Run the configure checks, but stop before creating any output
     files.
```

'configure' also accepts some other, not widely useful, options. Run
'configure --help' for more details.

Appendix B

MPI wrapper affiliation

Some wrapper functions are affiliated with a function group that has not been described for direct user access in section 'Selection of MPI Groups'. These groups are subgroups that contain function calls that are only enabled when both main groups are enabled. The reason for this is to control the amount of events generated during measurement, a user might want to turn off the measurement of non-critical function calls before the measurement of the complete main group is turned off. Subgroups can either be related to `MISC` (miscellaneous functions, e.g., handle conversion), `EXT` (external interfaces, e.g., handle attributes), or `ERR` (error handlers).

For example, the functions in group `CG_MISC` will only generate events if both groups `CG` and `MISC` are enabled at runtime.

Furthermore, the group `REQUEST` cannot be enabled directly by the user. The group is automatically enabled if any of the groups `CG`, `COLL`, `EXT`, `IO`, `P2P`, or `RMA` is enabled. It is disabled only if all of these groups are disabled as well.

B.1   Function to group

Function	Group
MPI_Abort	EXT
MPI_Accumulate	RMA
MPI_Add_error_class	ERR
MPI_Add_error_code	ERR
MPI_Add_error_string	ERR
MPI_Address	MISC
MPI_Allgather	COLL
MPI_Allgather_init	COLL
MPI_Allgatherv	COLL
MPI_Allgatherv_init	COLL
MPI_Alloc_mem	MISC
MPI_Allreduce	COLL
MPI_Allreduce_init	COLL
MPI_Alltoall	COLL
MPI_Alltoall_init	COLL

MPI_Alltoallv	COLL
MPI_Alltoallv_init	COLL
MPI_Alltoallw	COLL
MPI_Alltoallw_init	COLL
MPI_Attr_delete	CG_EXT
MPI_Attr_get	CG_EXT
MPI_Attr_put	CG_EXT
MPI_Barrier	COLL
MPI_Barrier_init	COLL
MPI_Bcast	COLL
MPI_Bcast_init	COLL
MPI_Bsend	P2P
MPI_Bsend_init	P2P
MPI_Buffer_attach	P2P
MPI_Buffer_detach	P2P
MPI_Cancel	REQUEST
MPI_Cart_coords	TOPO
MPI_Cart_create	TOPO
MPI_Cart_get	TOPO
MPI_Cart_map	TOPO
MPI_Cart_rank	TOPO
MPI_Cart_shift	TOPO
MPI_Cart_sub	TOPO
MPI_Cartdim_get	TOPO
MPI_Close_port	SPAWN
MPI_Comm_accept	SPAWN
MPI_Comm_c2f	CG_MISC
MPI_Comm_call_errhandler	CG_ERR
MPI_Comm_compare	CG
MPI_Comm_connect	SPAWN
MPI_Comm_create	CG
MPI_Comm_create_errhandler	CG_ERR
MPI_Comm_create_from_group	CG
MPI_Comm_create_group	CG
MPI_Comm_create_keyval	CG_EXT
MPI_Comm_delete_attr	CG_EXT
MPI_Comm_disconnect	SPAWN
MPI_Comm_dup	CG
MPI_Comm_dup_with_info	CG
MPI_Comm_f2c	CG_MISC
MPI_Comm_free	CG
MPI_Comm_free_keyval	CG_EXT
MPI_Comm_get_attr	CG_EXT
MPI_Comm_get_errhandler	CG_ERR
MPI_Comm_get_info	CG_EXT
MPI_Comm_get_name	CG_EXT
MPI_Comm_get_parent	SPAWN

MPI_Comm_group	CG
MPI_Comm_idup	CG
MPI_Comm_idup_with_info	CG
MPI_Comm_join	SPAWN
MPI_Comm_rank	CG
MPI_Comm_remote_group	CG
MPI_Comm_remote_size	CG
MPI_Comm_set_attr	CG_EXT
MPI_Comm_set_errhandler	CG_ERR
MPI_Comm_set_info	CG_EXT
MPI_Comm_set_name	CG_EXT
MPI_Comm_size	CG
MPI_Comm_spawn	SPAWN
MPI_Comm_spawn_multiple	SPAWN
MPI_Comm_split	CG
MPI_Comm_split_type	CG
MPI_Comm_test_inter	CG
MPI_Compare_and_swap	RMA
MPI_Dims_create	TOPO
MPI_Dist_graph_create	TOPO
MPI_Dist_graph_create_adjacent	TOPO
MPI_Dist_graph_neighbors	TOPO
MPI_Dist_graph_neighbors_count	TOPO
MPI_Errhandler_create	ERR
MPI_Errhandler_free	ERR
MPI_Errhandler_get	ERR
MPI_Errhandler_set	ERR
MPI_Error_class	ERR
MPI_Error_string	ERR
MPI_Exscan	COLL
MPI_Exscan_init	COLL
MPI_Fetch_and_op	RMA
MPI_File_c2f	IO_MISC
MPI_File_call_errhandler	IO_ERR
MPI_File_close	IO
MPI_File_create_errhandler	IO_ERR
MPI_File_delete	IO
MPI_File_f2c	IO_MISC
MPI_File_get_amode	IO
MPI_File_get_atomicity	IO
MPI_File_get_byte_offset	IO
MPI_File_get_errhandler	IO_ERR
MPI_File_get_group	IO
MPI_File_get_info	IO
MPI_File_get_position	IO
MPI_File_get_position_shared	IO
MPI_File_get_size	IO

MPI_File_get_type_extent	IO
MPI_File_get_view	IO
MPI_File_iread	IO
MPI_File_iread_all	IO
MPI_File_iread_at	IO
MPI_File_iread_at_all	IO
MPI_File_iread_shared	IO
MPI_File_iwrite	IO
MPI_File_iwrite_all	IO
MPI_File_iwrite_at	IO
MPI_File_iwrite_at_all	IO
MPI_File_iwrite_shared	IO
MPI_File_open	IO
MPI_File_preallocate	IO
MPI_File_read	IO
MPI_File_read_all	IO
MPI_File_read_all_begin	IO
MPI_File_read_all_end	IO
MPI_File_read_at	IO
MPI_File_read_at_all	IO
MPI_File_read_at_all_begin	IO
MPI_File_read_at_all_end	IO
MPI_File_read_ordered	IO
MPI_File_read_ordered_begin	IO
MPI_File_read_ordered_end	IO
MPI_File_read_shared	IO
MPI_File_seek	IO
MPI_File_seek_shared	IO
MPI_File_set_atomicity	IO
MPI_File_set_errhandler	IO_ERR
MPI_File_set_info	IO
MPI_File_set_size	IO
MPI_File_set_view	IO
MPI_File_sync	IO
MPI_File_write	IO
MPI_File_write_all	IO
MPI_File_write_all_begin	IO
MPI_File_write_all_end	IO
MPI_File_write_at	IO
MPI_File_write_at_all	IO
MPI_File_write_at_all_begin	IO
MPI_File_write_at_all_end	IO
MPI_File_write_ordered	IO
MPI_File_write_ordered_begin	IO
MPI_File_write_ordered_end	IO
MPI_File_write_shared	IO
MPI_Finalize	ENV

MPI_Finalized	ENV
MPI_Free_mem	MISC
MPI_Gather	COLL
MPI_Gather_init	COLL
MPI_Gatherv	COLL
MPI_Gatherv_init	COLL
MPI_Get	RMA
MPI_Get_accumulate	RMA
MPI_Get_address	MISC
MPI_Get_count	EXT
MPI_Get_elements	EXT
MPI_Get_elements_x	EXT
MPI_Get_library_version	ENV
MPI_Get_processor_name	EXT
MPI_Get_version	MISC
MPI_Graph_create	TOPO
MPI_Graph_get	TOPO
MPI_Graph_map	TOPO
MPI_Graph_neighbors	TOPO
MPI_Graph_neighbors_count	TOPO
MPI_Graphdims_get	TOPO
MPI_Grequest_complete	EXT
MPI_Grequest_start	EXT
MPI_Group_c2f	CG_MISC
MPI_Group_compare	CG
MPI_Group_difference	CG
MPI_Group_excl	CG
MPI_Group_f2c	CG_MISC
MPI_Group_free	CG
MPI_Group_from_session_pset	CG
MPI_Group_incl	CG
MPI_Group_intersection	CG
MPI_Group_range_excl	CG
MPI_Group_range_incl	CG
MPI_Group_rank	CG
MPI_Group_size	CG
MPI_Group_translate_ranks	CG
MPI_Group_union	CG
MPI_Iallgather	COLL
MPI_Iallgatherv	COLL
MPI_Iallreduce	COLL
MPI_Ialltoall	COLL
MPI_Ialltoallv	COLL
MPI_Ialltoallw	COLL
MPI_Ibarrier	COLL
MPI_Ibcast	COLL
MPI_Ibsend	P2P

MPI_Iexscan	COLL
MPI_Igather	COLL
MPI_Igatherv	COLL
MPI_Improbe	P2P
MPI_Imrecv	P2P
MPI_Ineighbor_allgather	TOPO
MPI_Ineighbor_allgatherv	TOPO
MPI_Ineighbor_alltoall	TOPO
MPI_Ineighbor_alltoallv	TOPO
MPI_Ineighbor_alltoallw	TOPO
MPI_Info_c2f	MISC
MPI_Info_create	MISC
MPI_Info_create_env	MISC
MPI_Info_delete	MISC
MPI_Info_dup	MISC
MPI_Info_f2c	MISC
MPI_Info_free	MISC
MPI_Info_get	MISC
MPI_Info_get_nkeys	MISC
MPI_Info_get_nthkey	MISC
MPI_Info_get_string	MISC
MPI_Info_get_valuelen	MISC
MPI_Info_set	MISC
MPI_Init	ENV
MPI_Init_thread	ENV
MPI_Initialized	ENV
MPI_Intercomm_create	CG
MPI_Intercomm_create_from_groups	CG
MPI_Intercomm_merge	CG
MPI_Iprobe	P2P
MPI_Irecv	P2P
MPI_Ireduce	COLL
MPI_Ireduce_scatter	COLL
MPI_Ireduce_scatter_block	COLL
MPI_Irsend	P2P
MPI_Is_thread_main	ENV
MPI_Iscan	COLL
MPI_Iscatter	COLL
MPI_Iscatterv	COLL
MPI_Isend	P2P
MPI_Isendrecv	P2P
MPI_Isendrecv_replace	P2P
MPI_Issend	P2P
MPI_Keyval_create	CG_EXT
MPI_Keyval_free	CG_EXT
MPI_Lookup_name	SPAWN
MPI_Mprobe	P2P

MPI_Mrecv	P2P
MPI_Neighbor_allgather	TOPO
MPI_Neighbor_allgather_init	TOPO
MPI_Neighbor_allgatherv	TOPO
MPI_Neighbor_allgatherv_init	TOPO
MPI_Neighbor_alltoall	TOPO
MPI_Neighbor_alltoall_init	TOPO
MPI_Neighbor_alltoallv	TOPO
MPI_Neighbor_alltoallv_init	TOPO
MPI_Neighbor_alltoallw	TOPO
MPI_Neighbor_alltoallw_init	TOPO
MPI_Op_c2f	MISC
MPI_Op_commutative	MISC
MPI_Op_create	MISC
MPI_Op_f2c	MISC
MPI_Op_free	MISC
MPI_Open_port	SPAWN
MPI_Pack	TYPE
MPI_Pack_external	TYPE
MPI_Pack_external_size	TYPE
MPI_Pack_size	TYPE
MPI_Parrived	REQUEST
MPI_Pcontrol	PERF
MPI_Pready	REQUEST
MPI_Pready_list	REQUEST
MPI_Pready_range	REQUEST
MPI_Precv_init	P2P
MPI_Probe	P2P
MPI_Psend_init	P2P
MPI_Publish_name	SPAWN
MPI_Put	RMA
MPI_Query_thread	ENV
MPI_Raccumulate	RMA
MPI_Recv	P2P
MPI_Recv_init	P2P
MPI_Reduce	COLL
MPI_Reduce_init	COLL
MPI_Reduce_local	COLL
MPI_Reduce_scatter	COLL
MPI_Reduce_scatter_block	COLL
MPI_Reduce_scatter_block_init	COLL
MPI_Reduce_scatter_init	COLL
MPI_Register_datarep	IO
MPI_Request_c2f	MISC
MPI_Request_f2c	MISC
MPI_Request_free	REQUEST
MPI_Request_get_status	REQUEST

MPI_Rget	RMA
MPI_Rget_accumulate	RMA
MPI_Rput	RMA
MPI_Rsend	P2P
MPI_Rsend_init	P2P
MPI_Scan	COLL
MPI_Scan_init	COLL
MPI_Scatter	COLL
MPI_Scatter_init	COLL
MPI_Scatterv	COLL
MPI_Scatterv_init	COLL
MPI_Send	P2P
MPI_Send_init	P2P
MPI_Sendrecv	P2P
MPI_Sendrecv_replace	P2P
MPI_Session_call_errhandler	ENV
MPI_Session_create_errhandler	ENV
MPI_Session_finalize	ENV
MPI_Session_get_errhandler	ENV
MPI_Session_get_info	ENV
MPI_Session_get_nth_pset	ENV
MPI_Session_get_num_psets	ENV
MPI_Session_get_pset_info	ENV
MPI_Session_init	ENV
MPI_Session_set_errhandler	ENV
MPI_Ssend	P2P
MPI_Ssend_init	P2P
MPI_Start	REQUEST
MPI_Startall	REQUEST
MPI_Status_c2f	MISC
MPI_Status_f2c	MISC
MPI_Status_set_cancelled	EXT
MPI_Status_set_elements	EXT
MPI_Status_set_elements_x	EXT
MPI_Test	REQUEST
MPI_Test_cancelled	REQUEST
MPI_Testall	REQUEST
MPI_Testany	REQUEST
MPI_Testsome	REQUEST
MPI_Topo_test	TOPO
MPI_Type_c2f	TYPE_MISC
MPI_Type_commit	TYPE
MPI_Type_contiguous	TYPE
MPI_Type_create_darray	TYPE
MPI_Type_create_f90_complex	TYPE
MPI_Type_create_f90_integer	TYPE
MPI_Type_create_f90_real	TYPE

MPI_Type_create_hindexed	TYPE
MPI_Type_create_hindexed_block	TYPE
MPI_Type_create_hvector	TYPE
MPI_Type_create_indexed_block	TYPE
MPI_Type_create_keyval	TYPE_EXT
MPI_Type_create_resized	TYPE
MPI_Type_create_struct	TYPE
MPI_Type_create_subarray	TYPE
MPI_Type_delete_attr	TYPE_EXT
MPI_Type_dup	TYPE
MPI_Type_extent	TYPE
MPI_Type_f2c	TYPE_MISC
MPI_Type_free	TYPE
MPI_Type_free_keyval	TYPE_EXT
MPI_Type_get_attr	TYPE_EXT
MPI_Type_get_contents	TYPE
MPI_Type_get_envelope	TYPE
MPI_Type_get_extent	TYPE
MPI_Type_get_extent_x	TYPE
MPI_Type_get_name	TYPE_EXT
MPI_Type_get_true_extent	TYPE
MPI_Type_get_true_extent_x	TYPE
MPI_Type_hindexed	TYPE
MPI_Type_hvector	TYPE
MPI_Type_indexed	TYPE
MPI_Type_lb	TYPE
MPI_Type_match_size	TYPE
MPI_Type_set_attr	TYPE_EXT
MPI_Type_set_name	TYPE_EXT
MPI_Type_size	TYPE
MPI_Type_size_x	TYPE
MPI_Type_struct	TYPE
MPI_Type_ub	TYPE
MPI_Type_vector	TYPE
MPI_Unpack	TYPE
MPI_Unpack_external	TYPE
MPI_Unpublish_name	SPAWN
MPI_Wait	REQUEST
MPI_Waitall	REQUEST
MPI_Waitany	REQUEST
MPI_Waitsome	REQUEST
MPI_Win_allocate	RMA
MPI_Win_allocate_shared	RMA
MPI_Win_attach	RMA
MPI_Win_c2f	RMA_MISC
MPI_Win_call_errhandler	RMA_ERR
MPI_Win_complete	RMA

MPI_Win_create	RMA
MPI_Win_create_dynamic	RMA
MPI_Win_create_errhandler	RMA_ERR
MPI_Win_create_keyval	RMA_EXT
MPI_Win_delete_attr	RMA_EXT
MPI_Win_detach	RMA
MPI_Win_f2c	RMA_MISC
MPI_Win_fence	RMA
MPI_Win_flush	RMA
MPI_Win_flush_all	RMA
MPI_Win_flush_local	RMA
MPI_Win_flush_local_all	RMA
MPI_Win_free	RMA
MPI_Win_free_keyval	RMA_EXT
MPI_Win_get_attr	RMA_EXT
MPI_Win_get_errhandler	RMA_ERR
MPI_Win_get_group	RMA
MPI_Win_get_info	RMA_EXT
MPI_Win_get_name	RMA_EXT
MPI_Win_lock	RMA
MPI_Win_lock_all	RMA
MPI_Win_post	RMA
MPI_Win_set_attr	RMA_EXT
MPI_Win_set_errhandler	RMA_ERR
MPI_Win_set_info	RMA_EXT
MPI_Win_set_name	RMA_EXT
MPI_Win_shared_query	RMA
MPI_Win_start	RMA
MPI_Win_sync	RMA
MPI_Win_test	RMA
MPI_Win_unlock	RMA
MPI_Win_unlock_all	RMA
MPI_Win_wait	RMA
MPI_Wtick	EXT
MPI_Wtime	EXT

B.2   Group to function

CG - **Communicators and Groups**
MPI_Comm_compare, MPI_Comm_create, MPI_Comm_create_from_group, MPI_Comm_create_group, MPI_Comm_dup, MPI_Comm_dup_with_info, MPI_Comm_free, MPI_Comm_group, MPI_Comm_idup, MPI_Comm_idup_with_info, MPI_Comm_rank, MPI_Comm_remote_group, MPI_Comm_remote_size, MPI_Comm_size, MPI_Comm_split, MPI_Comm_split_type, MPI_Comm_test_inter, MPI_Group_compare, MPI_Group_difference, MPI_Group_excl, MPI_Group_free, MPI_Group_from_session_pset, MPI_Group_incl, MPI_Group_intersection, MPI_Group_range_excl, MPI_Group_range_incl, MPI_Group_rank, MPI_Group_size, MPI_Group_translate_ranks, MPI_Group_union, MPI_Intercomm_create, MPI_Intercomm_create_from_groups, MPI_Intercomm_merge,

CG_ERR - **Error handlers for Communicators and Groups**
MPI_Comm_call_errhandler, MPI_Comm_create_errhandler, MPI_Comm_get_errhandler, MPI_Comm_set_errhandler,

CG_EXT - **External interfaces for Communicators and Groups**

MPI_Attr_delete, MPI_Attr_get, MPI_Attr_put, MPI_Comm_create_keyval, MPI_Comm_delete_attr, MPI_Comm_free_keyval, MPI_Comm_get_attr, MPI_Comm_get_info, MPI_Comm_get_name, MPI_Comm_set_attr, MPI_Comm_set_info, MPI_Comm_set_name, MPI_Keyval_create, MPI_Keyval_free,

CG_MISC - **Miscellaneous functions for Communicators and Groups**
MPI_Comm_c2f, MPI_Comm_f2c, MPI_Group_c2f, MPI_Group_f2c,

COLL - **Collective communication**
MPI_Allgather, MPI_Allgather_init, MPI_Allgatherv, MPI_Allgatherv_init, MPI_Allreduce, MPI_Allreduce_init, MPI_Alltoall, MPI_Alltoall_init, MPI_Alltoallv, MPI_Alltoallv_init, MPI_Alltoallw, MPI_Alltoallw_init, MPI_Barrier, MPI_Barrier_init, MPI_Bcast, MPI_Bcast_init, MPI_Exscan, MPI_Exscan_init, MPI_Gather, MPI_Gather_init, MPI_Gatherv, MPI_Gatherv_init, MPI_Iallgather, MPI_Iallgatherv, MPI_Iallreduce, MPI_Ialltoall, MPI_Ialltoallv, MPI_Ialltoallw, MPI_Ibarrier, MPI_Ibcast, MPI_Iexscan, MPI_Igather, MPI_Igatherv, MPI_Ireduce, MPI_Ireduce_scatter, MPI_Ireduce_scatter_block, MPI_Iscan, MPI_Iscatter, MPI_Iscatterv, MPI_Reduce, MPI_Reduce_init, MPI_Reduce_local, MPI_Reduce_scatter, MPI_Reduce_scatter_block, MPI_Reduce_scatter_block_init, MPI_Reduce_scatter_init, MPI_Scan, MPI_Scan_init, MPI_Scatter, MPI_Scatter_init, MPI_Scatterv, MPI_Scatterv_init,

ENV - **Environmental management**
MPI_Finalize, MPI_Finalized, MPI_Get_library_version, MPI_Init, MPI_Init_thread, MPI_Initialized, MPI_Is_thread_main, MPI_Query_thread, MPI_Session_call_errhandler, MPI_Session_create_errhandler, MPI_Session_finalize, MPI_Session_get_errhandler, MPI_Session_get_info, MPI_Session_get_nth_pset, MPI_Session_get_num_psets, MPI_Session_get_pset_info, MPI_Session_init, MPI_Session_set_errhandler,

ERR - **Common error handlers**
MPI_Add_error_class, MPI_Add_error_code, MPI_Add_error_string, MPI_Errhandler_create, MPI_Errhandler_free, MPI_Errhandler_get, MPI_Errhandler_set, MPI_Error_class, MPI_Error_string,

IO - **Parallel I/O**
MPI_File_close, MPI_File_delete, MPI_File_get_amode, MPI_File_get_atomicity, MPI_File_get_byte_offset, MPI_File_get_group, MPI_File_get_info, MPI_File_get_position, MPI_File_get_position_shared, MPI_File_get_size, MPI_File_get_type_extent, MPI_File_get_view, MPI_File_iread, MPI_File_iread_all, MPI_File_iread_at, MPI_File_iread_at_all, MPI_File_iread_shared, MPI_File_iwrite, MPI_File_iwrite_all, MPI_File_iwrite_at, MPI_File_iwrite_at_all, MPI_File_iwrite_shared, MPI_File_open, MPI_File_preallocate, MPI_File_read, MPI_File_read_all, MPI_File_read_all_begin, MPI_File_read_all_end, MPI_File_read_at, MPI_File_read_at_all, MPI_File_read_at_all_begin, MPI_File_read_at_all_end, MPI_File_read_ordered, MPI_File_read_ordered_begin, MPI_File_read_ordered_end, MPI_File_read_shared, MPI_File_seek, MPI_File_seek_shared, MPI_File_set_atomicity, MPI_File_set_info, MPI_File_set_size, MPI_File_set_view, MPI_File_sync, MPI_File_write, MPI_File_write_all, MPI_File_write_all_begin, MPI_File_write_all_end, MPI_File_write_at, MPI_File_write_at_all, MPI_File_write_at_all_begin, MPI_File_write_at_all_end, MPI_File_write_ordered, MPI_File_write_ordered_begin, MPI_File_write_ordered_end, MPI_File_write_shared, MPI_Register_datarep,

IO_ERR - **Error handlers for Parallel I/O**
MPI_File_call_errhandler, MPI_File_create_errhandler, MPI_File_get_errhandler, MPI_File_set_errhandler,

IO_MISC - **Miscellaneous functions for Parallel I/O**
MPI_File_c2f, MPI_File_f2c,

EXT - **Common external interfaces**
MPI_Abort, MPI_Get_count, MPI_Get_elements, MPI_Get_elements_x, MPI_Get_processor_name, MPI_Grequest_complete, MPI_Grequest_start, MPI_Status_set_cancelled, MPI_Status_set_elements, MPI_Status_set_elements_x, MPI_Wtick, MPI_Wtime,

MISC - **Miscellaneous functions**

MPI_Address, MPI_Alloc_mem, MPI_Free_mem, MPI_Get_address, MPI_Get_version, MPI_Info_c2f,
MPI_Info_create, MPI_Info_create_env, MPI_Info_delete, MPI_Info_dup, MPI_Info_f2c, MPI_Info_free,
MPI_Info_get, MPI_Info_get_nkeys, MPI_Info_get_nthkey, MPI_Info_get_string, MPI_Info_get_valuelen,
MPI_Info_set, MPI_Op_c2f, MPI_Op_commutative, MPI_Op_create, MPI_Op_f2c, MPI_Op_free,
MPI_Request_c2f, MPI_Request_f2c, MPI_Status_c2f, MPI_Status_f2c,

P2P - **Point-to-point communication**

MPI_Bsend, MPI_Bsend_init, MPI_Buffer_attach, MPI_Buffer_detach, MPI_Ibsend, MPI_Improbe, MPI_Imrecv,
MPI_Iprobe, MPI_Irecv, MPI_Irsend, MPI_Isend, MPI_Isendrecv, MPI_Isendrecv_replace, MPI_Issend,
MPI_Mprobe, MPI_Mrecv, MPI_Precv_init, MPI_Probe, MPI_Psend_init, MPI_Recv, MPI_Recv_init,
MPI_Rsend, MPI_Rsend_init, MPI_Send, MPI_Send_init, MPI_Sendrecv, MPI_Sendrecv_replace, MPI_Ssend,
MPI_Ssend_init,

PERF - **Profiling Interface**

MPI_Pcontrol,

REQUEST - **Completion of request-based operations**

MPI_Cancel, MPI_Parrived, MPI_Pready, MPI_Pready_list, MPI_Pready_range, MPI_Request_free,
MPI_Request_get_status, MPI_Start, MPI_Startall, MPI_Test, MPI_Test_cancelled, MPI_Testall, MPI_Testany,
MPI_Testsome, MPI_Wait, MPI_Waitall, MPI_Waitany, MPI_Waitsome,

RMA - **One-sided communication (Remote Memory Access)**

MPI_Accumulate, MPI_Compare_and_swap, MPI_Fetch_and_op, MPI_Get, MPI_Get_accumulate, MPI_Put,
MPI_Raccumulate, MPI_Rget, MPI_Rget_accumulate, MPI_Rput, MPI_Win_allocate,
MPI_Win_allocate_shared, MPI_Win_attach, MPI_Win_complete, MPI_Win_create, MPI_Win_create_dynamic,
MPI_Win_detach, MPI_Win_fence, MPI_Win_flush, MPI_Win_flush_all, MPI_Win_flush_local,
MPI_Win_flush_local_all, MPI_Win_free, MPI_Win_get_group, MPI_Win_lock, MPI_Win_lock_all,
MPI_Win_post, MPI_Win_shared_query, MPI_Win_start, MPI_Win_sync, MPI_Win_test, MPI_Win_unlock,
MPI_Win_unlock_all, MPI_Win_wait,

RMA_ERR - **Error handlers for One-sided communication (Remote Memory Access)**

MPI_Win_call_errhandler, MPI_Win_create_errhandler, MPI_Win_get_errhandler, MPI_Win_set_errhandler,

RMA_EXT - **External interfaces for One-sided communication (Remote Memory Access)**

MPI_Win_create_keyval, MPI_Win_delete_attr, MPI_Win_free_keyval, MPI_Win_get_attr, MPI_Win_get_info,
MPI_Win_get_name, MPI_Win_set_attr, MPI_Win_set_info, MPI_Win_set_name,

RMA_MISC - **Miscellaneous functions for One-sided communication (Remote Memory Access)**

MPI_Win_c2f, MPI_Win_f2c,

SPAWN - **Process spawning**

MPI_Close_port, MPI_Comm_accept, MPI_Comm_connect, MPI_Comm_disconnect, MPI_Comm_get_parent,
MPI_Comm_join, MPI_Comm_spawn, MPI_Comm_spawn_multiple, MPI_Lookup_name, MPI_Open_port,
MPI_Publish_name, MPI_Unpublish_name,

TOPO - **Topology (cartesian and graph) communicators**

MPI_Cart_coords, MPI_Cart_create, MPI_Cart_get, MPI_Cart_map, MPI_Cart_rank, MPI_Cart_shift,
MPI_Cart_sub, MPI_Cartdim_get, MPI_Dims_create, MPI_Dist_graph_create,
MPI_Dist_graph_create_adjacent, MPI_Dist_graph_neighbors, MPI_Dist_graph_neighbors_count,
MPI_Graph_create, MPI_Graph_get, MPI_Graph_map, MPI_Graph_neighbors, MPI_Graph_neighbors_count,
MPI_Graphdims_get, MPI_Ineighbor_allgather, MPI_Ineighbor_allgatherv, MPI_Ineighbor_alltoall,
MPI_Ineighbor_alltoallv, MPI_Ineighbor_alltoallw, MPI_Neighbor_allgather, MPI_Neighbor_allgather_init,
MPI_Neighbor_allgatherv, MPI_Neighbor_allgatherv_init, MPI_Neighbor_alltoall, MPI_Neighbor_alltoall_init,
MPI_Neighbor_alltoallv, MPI_Neighbor_alltoallv_init, MPI_Neighbor_alltoallw, MPI_Neighbor_alltoallw_init,
MPI_Topo_test,

TYPE - **Datatypes**

MPI_Pack, MPI_Pack_external, MPI_Pack_external_size, MPI_Pack_size, MPI_Type_commit,
MPI_Type_contiguous, MPI_Type_create_darray, MPI_Type_create_f90_complex,
MPI_Type_create_f90_integer, MPI_Type_create_f90_real, MPI_Type_create_hindexed,
MPI_Type_create_hindexed_block, MPI_Type_create_hvector, MPI_Type_create_indexed_block,
MPI_Type_create_resized, MPI_Type_create_struct, MPI_Type_create_subarray, MPI_Type_dup,
MPI_Type_extent, MPI_Type_free, MPI_Type_get_contents, MPI_Type_get_envelope, MPI_Type_get_extent,
MPI_Type_get_extent_x, MPI_Type_get_true_extent, MPI_Type_get_true_extent_x, MPI_Type_hindexed,
MPI_Type_hvector, MPI_Type_indexed, MPI_Type_lb, MPI_Type_match_size, MPI_Type_size,
MPI_Type_size_x, MPI_Type_struct, MPI_Type_ub, MPI_Type_vector, MPI_Unpack, MPI_Unpack_external,

TYPE_EXT - **External interfaces for datatypes**

MPI_Type_create_keyval, MPI_Type_delete_attr, MPI_Type_free_keyval, MPI_Type_get_attr,
MPI_Type_get_name, MPI_Type_set_attr, MPI_Type_set_name,

TYPE_MISC - **Miscellaneous functions for datatypes**

MPI_Type_c2f, MPI_Type_f2c,

Appendix C

Score-P Metric Plugin Example

Simple example of a Score-P metric plugin

```c
#include <scorep/SCOREP_MetricPlugins.h>

#include <stdio.h>
#include <stdlib.h>
#include <string.h>


/* Maximum number of metrics */
#define NUMBER_METRICS 5

/* Maximum string length */
#define MAX_BUFFER_LENGTH 255

/* Number of individual metrics */
static int32_t num_metrics = 0;

int32_t
init()
{
    return 0;
}

int32_t
add_counter( char* eventName )
{
    int id = num_metrics;
    num_metrics++;

    return id;
}

SCOREP_Metric_Plugin_MetricProperties*
get_event_info( char* eventName )
{
    SCOREP_Metric_Plugin_MetricProperties* return_values;
    char                                   name_buffer[ MAX_BUFFER_LENGTH ];
    int                                    i;

    /* If wildcard is specified, add some default counters */
    if ( strcmp( eventName, "*" ) == 0 )
    {
        return_values = malloc( ( NUMBER_METRICS + 1 ) * sizeof(
    SCOREP_Metric_Plugin_MetricProperties ) );
        for ( i = 0; i < NUMBER_METRICS; i++ )
        {
            snprintf( name_buffer, MAX_BUFFER_LENGTH, "sync counter #%i", i );
            return_values[ i ].name        = strdup( name_buffer );
            return_values[ i ].description = NULL;
            return_values[ i ].unit        = NULL;
            return_values[ i ].mode        =
    SCOREP_METRIC_MODE_ABSOLUTE_LAST;
            return_values[ i ].value_type  =
    SCOREP_METRIC_VALUE_UINT64;
            return_values[ i ].base        = SCOREP_METRIC_BASE_DECIMAL;
            return_values[ i ].exponent    = 0;
        }
        return_values[ NUMBER_METRICS ].name = NULL;
```

```c
    }
    else
    {
        /* If no wildcard is given create one counter with the passed name */
        return_values = malloc( 2 * sizeof(
    SCOREP_Metric_Plugin_MetricProperties ) );
        snprintf( name_buffer, MAX_BUFFER_LENGTH, "sync counter #%s", eventName );
        return_values[ 0 ].name        = strdup( name_buffer );
        return_values[ 0 ].description = NULL;
        return_values[ 0 ].unit        = NULL;
        return_values[ 0 ].mode        = SCOREP_METRIC_MODE_ABSOLUTE_LAST
    ;
        return_values[ 0 ].value_type  = SCOREP_METRIC_VALUE_UINT64;
        return_values[ 0 ].base        = SCOREP_METRIC_BASE_DECIMAL;
        return_values[ 0 ].exponent    = 0;
        return_values[ 1 ].name        = NULL;
    }
    return return_values;
}

bool
get_value( int32_t   counterIndex,
           uint64_t* value )
{
    *value = counterIndex;

    return true;
}

void
fini()
{
    return;
}

SCOREP_METRIC_PLUGIN_ENTRY( HelloWorld )
{
    /* Initialize info data (with zero) */
    SCOREP_Metric_Plugin_Info info;
    memset( &info, 0, sizeof( SCOREP_Metric_Plugin_Info ) );

    /* Set up */
    info.plugin_version     = SCOREP_METRIC_PLUGIN_VERSION;
    info.run_per            = SCOREP_METRIC_PER_PROCESS;
    info.sync               = SCOREP_METRIC_SYNC;
    info.initialize         = init;
    info.finalize           = fini;
    info.get_event_info     = get_event_info;
    info.add_counter        = add_counter;
    info.get_optional_value = get_value;

    return info;
}
```

See also

SCOREP_MetricPlugins.h

Appendix D

Experiment Directory Contents

The contents of the experiment directory might vary depending on the settings for the measurement run. The following table shows the possible files and the environment variables they are depending on.

Files that should be present from the start, e.g., in an aborted measurement:

README *Information about this measurement, e.g., list of expected files*

scorep.cfg *Listing of used environment variables*

scorep.filter *Copy of the used filter file*
SCOREP_FILTERING_FILE

scorep.selective *Copy of the used config file for selective recording*
SCOREP_SELECTIVE_CONFIG_FILE

Files that will be created at the end of the measurement:

profile.cubex *Cube4 result file of the summary measurement or an extended set of statistics*
SCOREP_ENABLE_PROFILING, SCOREP_PROFILING_BASE_NAME, SCOREP_PROFILING_FORMAT

tau/snapshot.[rank].0.0 *TAU snapshot files*
SCOREP_ENABLE_PROFILING, SCOREP_PROFILING_BASE_NAME, SCOREP_PROFILING_FORMAT

traces/ *Sub-directory with per location trace data*
SCOREP_ENABLE_TRACING

traces.def *OTF2 global definitions file*
SCOREP_ENABLE_TRACING

traces.otf2   *OTF2 anchor file*
SCOREP_ENABLE_TRACING

Additional debug information, the files and their contents may vary depending on the measurement progress:

profile.[rank].[thread].core   *Profiling core file in case of error*
SCOREP_PROFILING_ENABLE_CORE_FILES, SCOREP_PROFILING_BASE_NAME

Appendix E

Score-P Substrate Plugin Example

Simple example of a Score-P substrate plugin

```c
#include <assert.h>
#include <inttypes.h>
#include <stdio.h>
#include <string.h>
#include <scorep/SCOREP_SubstratePlugins.h>

static const SCOREP_SubstratePluginCallbacks* callbacks;

static size_t plugin_id;

static void
print( const char*              event,
       struct SCOREP_Location*  location,
       uint64_t                 timestamp,
       SCOREP_RegionHandle      regionHandle,
       uint64_t                 stackdepth )
{
    uint64_t    i;
    const char* slocation = callbacks->SCOREP_Location_GetName( location );
    const char* sregion   = callbacks->SCOREP_RegionHandle_GetName( regionHandle
        );

    printf( "%" PRIu64 ": %s >%" PRIu64 " %s %s\n", timestamp, slocation, stackdepth, event, sregion );
}

/* An enter event has been received from Score-P */
static void
enter_region(
    struct SCOREP_Location*  location,
    uint64_t                 timestamp,
    SCOREP_RegionHandle      regionHandle,
    uint64_t*                metricValues )
{
    uint64_t* stackdepth = callbacks->SCOREP_Location_GetData( location, plugin_id )
        ;
    print( "Enter", location, timestamp, regionHandle, *stackdepth );
    ( *stackdepth )++;
}

/* An exit event has been received from Score-P */
static void
exit_region(
    struct SCOREP_Location*  location,
    uint64_t                 timestamp,
    SCOREP_RegionHandle      regionHandle,
    uint64_t*                metricValues )
{
    uint64_t* stackdepth = callbacks->SCOREP_Location_GetData( location, plugin_id )
        ;
    ( *stackdepth )--;
    print( "Exit", location, timestamp, regionHandle, *stackdepth );
}

/* is only called when process is currently not parallel */
static void
disable(
```

```c
    struct SCOREP_Location* location,
    uint64_t                timestamp,
    SCOREP_RegionHandle     regionHandle,
    uint64_t*               metricValues )
{
    const char* slocation = callbacks->SCOREP_Location_GetName( location );
    const char* sregion   = callbacks->SCOREP_RegionHandle_GetName( regionHandle
        );
    printf( "--------Disabled recording (%s - %s)------------\n",
            slocation,
            sregion );
}

/* is only called when process is currently not parallel */
static void
enable(
    struct SCOREP_Location* location,
    uint64_t                timestamp,
    SCOREP_RegionHandle     regionHandle,
    uint64_t*               metricValues )
{
    const char* slocation = callbacks->SCOREP_Location_GetName( location );
    const char* sregion   = callbacks->SCOREP_RegionHandle_GetName( regionHandle
        );
    printf( "--------Enabled recording (%s - %s)------------\n",
            slocation,
            sregion );
}

/* Register event functions */
static uint32_t
get_event_functions(
    SCOREP_Substrates_Mode       mode,
    SCOREP_Substrates_Callback** returned )
{
    SCOREP_Substrates_Callback* functions = calloc(
      SCOREP_SUBSTRATES_NUM_EVENTS,
                                                    sizeof(
      SCOREP_Substrates_Callback ) );

    /* Only print region events when recording is enabled */
    if ( mode == SCOREP_SUBSTRATES_RECORDING_ENABLED )
    {
        functions[ SCOREP_EVENT_ENTER_REGION ] = (
      SCOREP_Substrates_Callback )enter_region;
        functions[ SCOREP_EVENT_EXIT_REGION ]  = (
      SCOREP_Substrates_Callback )exit_region;
        /* function prototypes for other events can be found in scorep/SCOREP_SubstrateEvents.h */

        /* enable and disable are guaranteed to be called for SCOREP_SUBSTRATES_RECORDING_ENABLED */
        functions[ SCOREP_EVENT_ENABLE_RECORDING ]  = (
      SCOREP_Substrates_Callback )enable;
        functions[ SCOREP_EVENT_DISABLE_RECORDING ] = (
      SCOREP_Substrates_Callback )disable;
    }

    *returned = functions;
    return SCOREP_SUBSTRATES_NUM_EVENTS;
}

/* Receive callbacks from Score-P */
static void
set_callbacks( const SCOREP_SubstratePluginCallbacks* incoming_callbacks,
               size_t                                  size )
{
    assert( sizeof( SCOREP_SubstratePluginCallbacks ) <= size );
    callbacks = incoming_callbacks;
}

/* assign id */
static void
assign_id( size_t id )
{
    plugin_id = id;
}


static void
create_location( const struct SCOREP_Location* location,
                 const struct SCOREP_Location* parentLocation )
{
    /* we store the stackdepth per location and initialize it with 0 */
    uint64_t* stackdepth = calloc( sizeof( uint64_t ), 1 );
    callbacks->SCOREP_Location_SetData( location, plugin_id, ( void* )stackdepth );
}

/* Define plugins and some plugin functions */
```

```
SCOREP_SUBSTRATE_PLUGIN_ENTRY( PrintRegions )
{
    SCOREP_SubstratePluginInfo info;
    memset( &info, 0, sizeof( SCOREP_SubstratePluginInfo ) );
    info.plugin_version      = SCOREP_SUBSTRATE_PLUGIN_VERSION
      ;
    info.set_callbacks       = set_callbacks;
    info.create_location     = create_location;
    info.assign_id           = assign_id;
    info.get_event_functions = get_event_functions;
    return info;
}
```

See also

SCOREP_SubstratePlugins.h

Appendix F

Score-P Tools

F.1 scorep

A call to `scorep` has the following syntax:

```
This is the Score-P instrumentation tool. The usage is:
scorep <options> <original command>

Common options are:
  --help, -h      Show help output. Does not execute any other command.
  -v, --verbose[=<value>] Specifies the verbosity level. The following
                  levels are available:
                  0 = No output
                  1 = Executed commands are displayed (default if no
                      value is specified)
                  2 = Detailed information is displayed
  --dry-run       Only displays the executed commands. It does not
                  execute any command.
  --keep-files    Do not delete temporarily created files after successful
                  instrumentation. By default, temporary files are deleted
                  if no error occurs during instrumentation.
  --instrument-filter=<file>
                  Specifies the filter file for filtering functions during
                  compile-time. Not supported by all instrumentation methods.
                  It applies the same syntax, as the one used by Score-P during
                  run-time.
  --version       Prints the Score-P version and exits.
  --disable-preprocessing
                  Tells scorep to skip all preprocessing related steps,
                  the input files are already preprocessed.
  --static        Enforce static linking of the Score-P libraries.
  --dynamic       Enforce dynamic linking of the Score-P libraries.
  --no-as-needed  Adds a GNU ld linker flag to fix undefined references
                  when using shared Score-P libraries. This happens on
                  systems using --as-needed as linker default. It will
                  be handled transparently in future releases of Score-P.
  --thread=<paradigm>[:<variant>]
                  Possible paradigms and variants are:
              none
                  No thread support.
              omp:opari2
                  OpenMP support using thread tracking via ancestry functions
                  in OpenMP 3.0 and later.
                  It requires and, thus, automatically enables OPARI2
                  instrumentation.
              omp:ompt
                  OpenMP support using thread tracking via OMPT.
                  It conflicts and, thus, automatically disables pdt
                  instrumentation.
```

```
                pthread
                        Pthread support using thread tracking via library wrapping.
                        It requires and, thus, automatically enables linktime-wrapping
                        instrumentation.
                        It conflicts and, thus, automatically disables OPARI2
                        instrumentation.
  --mpp=<paradigm>[:<variant>]
                        Possible paradigms and variants are:
                none
                        No multi-process support.
                mpi
                        MPI support using library wrapping
                shmem
                        SHMEM support using library wrapping
  --io=[<wrap-mode>:]<paradigm>[:<variant>](,[<wrap-mode>:]<paradigm>[:<variant>])*
                        <wrap-mode> may be 'linktime' or 'runtime'.
                        The default is the first supported mode in the above order.
                        Possible paradigms and variants are:
                none
                        No I/O wrapping support.
                posix
                        POSIX I/O support using library wrapping. This includes the
                        file descriptor based POSIX API (i.e., 'open'/'close'). The
                        POSIX asynchronous I/O API (i.e., 'aio_read'/'aio_write'), if
                        available. And the ISO C 'FILE' handle based API (i.e.,
                        'fopen'/'fclose').
  --compiler      Enables compiler instrumentation.
                        By default, it disables pdt instrumentation.
  --nocompiler    Disables compiler instrumentation.
  --cuda          Enables CUDA instrumentation. Enabled by default, if the
                        nvcc compiler is in use. In this case it also conflicts and
                        thus automatically disables preprocessing.
  --nocuda        Disables CUDA instrumentation.
  --pomp          Enables OPARI2 pomp user instrumentation. By default, it also
                        enables preprocessing.
  --nopomp        Disables OPARI2 pomp user instrumentation (Default).
  --openmp        Enables OPARI2 instrumentation of OpenMP directives. By default,
                        it also enables preprocessing (Default for compile units
                        with enabled OpenMP support during the compilation).
                        Conflicts with --thread=omp:ompt.
  --noopenmp      Disables OPARI2 instrumentation of OpenMP directives.
                        Note: To ensure thread-safe execution of the measurement,
                        parallel regions still need to be tracked and will appear
                        in the results (Default for compile units without OpenMP
                        enabled compilation).
  --opari=<parameter-list>
                        Pass options to the source-to-source instrumenter OPARI2
                        to have finer control over the instrumentation process.
                        Please refer to the OPARI2 user documentation for more
                        details.
  --pdt[=<parameter-list>]
                        Enables pdt instrumentation.
                        You may add additional parameters that are passed to pdt.
                        It requires and, thus, automatically enables user
                        instrumentation.
                        It conflicts and, thus, automatically disables preprocess
                        instrumentation.
                        By default, it disables compiler instrumentation.
  --nopdt         Disables pdt instrumentation.
  --preprocess    Enables preprocess instrumentation.
                        It cannot be enabled, if not at least one of the following is
                        enabled: OPARI2 instrumentation.
                        It conflicts and, thus, automatically disables pdt
                        instrumentation.
  --nopreprocess  Disables preprocess instrumentation.
  --user          Enables user instrumentation.
  --nouser        Disables user instrumentation.
  --opencl[:<wrap-mode>]
                        Enables OpenCL instrumentation.
  --noopencl      Disables OpenCL instrumentation.
  --openacc       Enables OpenACC instrumentation.
  --noopenacc     Disables OpenACC instrumentation.
  --memory        Enables memory usage instrumentation. It is enabled by default.
```

```
  --nomemory      Disables memory usage instrumentation.
  --libwrap=[<wrap-mode>:](<wrapper-name>...|<path-to-wrapper>)
                  Enables user library wrapping for specified libraries.
                  <wrap-mode> may be 'linktime' or 'runtime'.
                  The default is the first supported mode in the above order.
                  <wrapper-name>... is a comma-separated list of library wrappers
                  which will be looked up in the paths of the colon-separated
                  'SCOREP_LIBWRAP_PATH' environment variable and in the installation
                  directory of Score-P. <path-to-wrapper> is a full path to the
                  .libwrap anchor file.
                  Use 'scorep-info libwrap-summary' for a more detailed listing
                  of available library wrappers.
  --kokkos        Enables Kokkos instrumentation.
  --nokokkos      Disables Kokkos instrumentation.
  --hip           Enables HIP instrumentation. Enabled by default, if the
                  hipcc compiler is in use. In this case it also disables by
                  default the compiler instrumentation and conflicts with the
                  preprocessing.
  --nohip         Disables HIP instrumentation.

Report bugs to <support@score-p.org>
```

F.2   scorep-config

A call to `scorep-config` has the following syntax:

```
Usage:
scorep-config <command> [<options>]
  Commands:
   --prefix   Prints the canonical installation prefix of this Score-P
              installation.
   --cflags   Prints additional compiler flags for a C compiler.
              They already contain the include flags.
   --cxxflags Prints additional compiler flags for a C++ compiler.
              They already contain the include flags.
   --fflags   Prints additional compiler flags for a Fortran compiler.
              They already contain the include flags.
   --cppflags[=language]
              Prints the include flags. They are already contained in the
              output of the --cflags, --cxxflags, and --fflags commands.
              language may be one of c (default), c++, or fortran.
   --ldflags  Prints the library path flags for the linker.
   --libs     Prints the required libraries to link against
              (combines --event-libs and --mgmt-libs).
   --event-libs
              Prints only the required libraries to link against which
              includes event entry points into the measurement.
   --mgmt-libs
              Prints only the required libraries to link against which
              includes management code from the Score-P measurement and
              their dependencies.
   --preload-libs
              Prints only the required libraries which should be listed in
              LD_PRELOAD.
   --cc       Prints the C compiler name.
   --cxx      Prints the C++ compiler name.
   --fc       Prints the Fortran compiler name.
   --mpicc    Prints the MPI C compiler name.
   --mpicxx   Prints the MPI C++ compiler name.
   --mpifc    Prints the MPI Fortran compiler name.
   --shmemcc  Prints the SHMEM C compiler name.
   --shmemcxx Prints the SHMEM C++ compiler name.
   --shmemfc  Prints the SHMEM Fortran compiler name.
   --libtool  Prints the path to the libtool script used to build Score-P
              libraries.
   --mpilibtool
              Prints the path to the libtool script used to build Score-P
              MPI libraries.
```

```
 --shmemlibtool
            Prints the path to the libtool script used to build Score-P
            SHMEM libraries.
 --help     Prints this usage information.
 --version  Prints the version number of the Score-P package.
 --scorep-revision
            Prints the revision number of the Score-P package.
 --common-revision
            Prints the revision number of the common package.
 --remap-specfile
            Prints the path to the remapper specification file.
 --adapter-init
            Prints the code for adapter initialization.
 --libwrap-support=linktime
            Prints true if link-time library wrapping is supported.
 --libwrap-support=runtime
            Prints true if run-time library wrapping is supported.
Options:
 --target   Get flags for specified target, e.g., mic.
 --nvcc     Convert flags to be suitable for the nvcc compiler.
 --static   Use only static Score-P libraries if possible.
 --dynamic  Use only dynamic Score-P libraries if possible.
 --compiler|--nocompiler
            Specifies whether compiler instrumentation is used.
            On default compiler instrumentation is enabled.
 --user|--nouser
            Specifies whether user instrumentation is used.
            On default user instrumentation is disabled.
 --pomp|--nopomp
            Specifies whether pomp instrumentation is used.
            On default pomp instrumentation is disabled.
 --cuda|--nocuda
            Specifies whether cuda instrumentation is used.
            On default cuda instrumentation is enabled.
 --openacc|--noopenacc
            Specifies whether openacc instrumentation is used.
            On default openacc instrumentation is enabled.
 --hip|--nohip
            Specifies whether hip instrumentation is used.
            On default hip instrumentation is enabled.
 --opencl[:<wrapping mode>]|--noopencl
            Specifies whether opencl instrumentation is used.
            On default opencl instrumentation is enabled.
            Available wrapping modes are:
             linktime (default)
             runtime
 --kokkos|--nokokkos
            Specifies whether kokkos instrumentation is used.
            On default kokkos instrumentation is enabled.
 --preprocess|--nopreprocess
            Specifies whether preprocess instrumentation is used.
            On default preprocess instrumentation is disabled.
 --memory=<api list>|--nomemory
            Specifies whether memory usage recording is used.
            On default memory usage recording is disabled.
            The following memory interfaces may be recorded:
             libc:
              malloc,realloc,calloc,free,memalign,posix_memalign,valloc
             libc11:
              aligned_alloc
             c++L32|c++L64:
              new,new[],delete,delete[] (IA-64 C++ ABI)
             pgCCL32|pgCCL64:
              new,new[],delete,delete[] (old PGI/EDG C++ ABI)
 --io=[<wrap-mode>:]<paradigm,...>|--noio
            Specifies whether I/O recording is used.
            On default I/O recording is disabled.
            The following I/O paradigms may be recorded:
             none
             posix
 --libwrap=<wrap-mode>:<libwrap-anchor-file>
            Uses the specified library wrapper.
 --thread=<threading system>
```

```
            Available threading systems are:
             none (default)
             omp
             pthread
  --mpp=<multi-process paradigm>
            Available multi-process paradigms are:
             none (default)
             mpi
             shmem

Report bugs to <support@score-p.org>
```

F.3   scorep-info

A call to `scorep-info` has the following syntax:

```
Usage: scorep-info <info command> <command options>
       scorep-info --help [<info command>]
This is the Score-P info tool.

Available info commands:

  config-summary:
    Shows the configure summary of the Score-P package.

  config-vars:
    Shows the list of all measurement config variables with a short description.

    Info command options:
      --help       Displays a description of the Score-P measurement
                   configuration system.
      --full       Displays a detailed description for each config variable.
      --values     Displays the current values for each config variable.
                   Warning: These values may be wrong, please consult the
                            manual of the batch system how to pass the values
                            to the measurement job.

  ldaudit:
    Shows the LD_AUDIT value needed to extend address lookup to dynamically
    loaded shared objects.

  libwrap-summary:
    Shows known library wrappers available to this Score-P installation.

    Info command options:
      --build      Shows detailed information about how the library wrapper
                   was built.

  license:
    Shows the license of the Score-P package.

  open-issues:
    Shows open and known issues of the Score-P package.

Report bugs to <support@score-p.org>
```

F.4   scorep-score

A call to `scorep-score` has the following syntax:

```
Usage: scorep-score [options] <profile>
Options:
 -r          Show all regions.
```

```
-h, --help  Show this help and exit.
-f <filter> Shows the result with the filter applied.
-c <num>    Specifies the number of hardware counters that shall be measured.
            By default, this value is 0, which means that only a timestamp
            is measured on each event. If you plan to record hardware counters
            specify the number of hardware counters. Otherwise, scorep-score
            may underestimate the required space.
-m          Prints mangled region names instead of demangled names.
-s <choice> Sorting of entries. Possible choices are totaltime, timepervisit,
             maxbuffer, visits and name (default=maxbuffer).
-g [<list>] Generation of an initial filter file with the name
            'initial_scorep.filter'. A valid parameter list has the form
            KEY=VALUE[,KEY=VALUE]*. By default, uses the following control
            parameters:

                'bufferpercent=1,timepervisit=1'

            A region is included in the filter file (i.e., excluded from
            measurement) if it matches all of the given conditions, with the
            following keys:
            - 'bufferpercent'       : estimated memory requirements exceed the
                                      given threshold in percent of the total
                                      estimated trace buffer requirements
            - 'bufferabsolute'      : estimated memory requirements exceed
                                      the given absolute threshold in MB
            - 'visits'              : number of visits exceeds the given
                                      threshold
            - 'timepervisit'        : time per visit value is below the given
                                      threshold in microseconds
            - 'type'                : region type matches the given value
                                      (allowed: 'usr', 'com', 'both')
```

F.5 scorep-backend-info

Note

> This tool is intended to run as a batch job. Please consult the manual of the batch system how to submit jobs.

A call to `scorep-backend-info` has the following syntax:

```
Usage: scorep-backend-info <info command> <command options>
       scorep-backend-info --help
This is the Score-P backend info tool.
Available info commands:

  system-tree:
    Shows the available system tree levels, starting with the root.

  config-vars:
    Shows the current values of all measurement config variables.
```

Appendix G

Score-P Measurement Configuration

Introduction

Score-P allows to configure several measurement parameters via environment variables. After the measurement run you can find a 'scorep.cfg' file in your experiment directory which contains the configuration of the measurement run. If you did not set any configuration values explicitly, this file will contain the default values. This file is safe to be used as input for a POSIX shell. For example, if you want to reuse the same configuration from a previous measurement run, do something like this:

```
$ set -a
$ . scorep.cfg
$ set +a
```

Measurement configuration variables have a specific type which accepts certain values.

Supported Types

String

An arbitrary character sequence, no white space trimming is done.

Path

Like String but a path is expected. Though no validation is performed.

Boolean

A Boolean value, no white space trimming is done. Accepted Boolean values for true are case insensitive and the following:

- 'true'
- 'yes'
- 'on'
- any positive decimal number

Everything else is interpreted as the Boolean value false.

Number

A decimal number, white space trimming is done.

Number with size suffixes

Like Number, but also accepts unit case insensitive suffixes after optional white space:

- 'B', 'Kb', 'Mb', 'Gb', 'Tb', 'Pb', 'Eb'

The 'b' suffix can be omitted.

Set

A symbolic set. Accepted members are listed in the documentation of the variable. Multiple values are allowed, are case insensitive, and are subject to white space trimming. They can be separated with one of the following characters:

- ' ' - space

- ',' - comma

- ':' - colon

- ';' - semicolon

Acceptable values can also have aliases, which are listed in the documentation and separated by '/'. Values can be negated by preceeding it with '$\sim$'. Order of evaluation is from left to right.

Option

Like Set, but only one value allowed to be selected.

List of Configuration Variables

This is the list of all known configure variables to control a Score-P measurement.

Not all variables are supported by one Score-P installation. Use the `scorep-info config-vars` command to list only those supported by the used installation.

SCOREP_ENABLE_PROFILING Enable profiling

> **Type:** Boolean
> **Default:** true

SCOREP_ENABLE_TRACING Enable tracing

> **Type:** Boolean
> **Default:** false

SCOREP_ENABLE_UNWINDING Enables recording calling context information for every event

> **Type:** Boolean
> **Default:** false
>
> The calling context is the call chain of functions to the current position in the running program. This call chain will also be annotated with source code information if possible.
> This is a prerequisite for sampling but also works with instrumented applications.
> Note that when tracing is also enabled, Score-P does not write the usual Enter/Leave records into the OTF2 trace, but new records.
> See also SCOREP_TRACING_CONVERT_CALLING_CONTEXT_EVENTS.
> Note also that this supresses events from the compiler instrumentation.

SCOREP_VERBOSE Be verbose

> **Type:** Boolean
> **Default:** false

SCOREP_TOTAL_MEMORY Total memory in bytes per process to be consumed by the measurement system

> **Type:** Number with size suffixes
> **Default:** 16000k
>
> SCOREP_TOTAL_MEMORY will be split into pages of size SCOREP_PAGE_SIZE (potentially reduced to a multiple of SCOREP_PAGE_SIZE). Maximum size is 4 GBminus one SCOREP_PAGE_SIZE.

SCOREP_PAGE_SIZE Memory page size in bytes

Type: Number with size suffixes
Default: 8k

If not a power of two, SCOREP_PAGE_SIZE will be increased to the next larger power of two. SCO↩
REP_TOTAL_MEMORY will be split up into pages of (the adjusted) SCOREP_PAGE_SIZE. Minimum size is 512 bytes.

SCOREP_EXPERIMENT_DIRECTORY   Name of the experiment directory as child of the current working directory

Type: Path
Default: ""

The experiment directory is created directly under the current working directory. No parent directories will be created. The experiment directory is only created if it is requested by at least one substrate. When no experiment name is given (the default) Score-P names the experiment directory 'scorep-measurement-tmp' and renames this after a successful measurement to a generated name based on the current time.

SCOREP_OVERWRITE_EXPERIMENT_DIRECTORY   Overwrite an existing experiment directory

Type: Boolean
Default: true

If you specified a specific experiment directory name, but this name is already given, you can force overwriting it with this flag. The previous experiment directory will be renamed.

SCOREP_MACHINE_NAME   The machine name used in profile and trace output

Type: String
Default: "Linux"

We suggest using a unique name, e.g., the fully qualified domain name. The default machine name was set at configure time (see the INSTALL file for customization options).

SCOREP_EXECUTABLE   Executable of the application

Type: Path
Default: ""

File name, preferably with full path, of the application's executable. This is a fallback if Score-P cannot determine the executable's name automatically. The name is required by some compiler adapters. They will complain if this environment variable is needed.

SCOREP_ENABLE_SYSTEM_TREE_SEQUENCE_DEFINITIONS   Use system tree sequence definitions

Type: Boolean

Default: false

Enables an internal system tree representation that specifies a sequence of system tree nodes with one record instead of creating one record per system tree node, location group or location. It is more scalable and has less memory requirements then single-node records. However, it costs individual names of nodes, but simply enumerates them based on types. Currently, system tree sequence definitions support only MPI (and trivially single-process) applications.

SCOREP_FORCE_CFG_FILES Force the creation of experiment directory and configuration files

Type: Boolean

Default: true

If this is set to 'true' (which is the default), the experiment directory will be created along with some configuration files, even if no substrate writes data (i.e., profiling and tracing are disabled and no substrate plugin registered for writing).
If this is set to 'false', the directory will only be created if any substrate actually writes data.

SCOREP_TIMER Timer used during measurement

Type: Option

Default: tsc

The following timers are available for this installation:

tsc Low overhead time stamp counter (X86_64) timer.
gettimeofday gettimeofday timer.
clock_gettime clock_gettime timer with CLOCK_MONOTONIC_RAW as clock.

SCOREP_DEBUG_UNIFY Writes the pre-unified definitions also in the local definition trace files

Type: Boolean

Default: true

SCOREP_PROFILING_TASK_EXCHANGE_NUM Number of foreign task objects that are collected before they are put into the common task object exchange buffer

Type: Number with size suffixes

Default: 1K

The profiling creates a record for every task instance that is running. To avoid locking, the required memory is taken from a preallocated memory block. Each thread has its own memory block. On task completion, the created object can be reused by other tasks. However, if tasks migrate, the data structure migrates with them. Thus, if there is an imbalance in the migration from a source thread that starts the execution of tasks towards a sink thread that completes the tasks, the source thread may continually creating new task objects while in the sink, released task objects are collected. Thus, if the sink collected a certain number of tasks it should trigger a backflow of its collected task objects. However, this requires locking which should be avoided as much as possible. Thus, we do not want the locking to happen on

every migrated task, but only if a certain imbalance occurs. This environment variable determines the number of migrated task instances that must be collected before the backflow is triggered.

SCOREP_PROFILING_MAX_CALLPATH_DEPTH Maximum depth of the calltree

Type: Number
Default: 100

SCOREP_PROFILING_BASE_NAME Base for construction of the profile filename

Type: String
Default: "profile"

String which is used as based to create the filenames for the profile files.

SCOREP_PROFILING_FORMAT Profile output format

Type: Option
Default: default

Sets the output format for the profile.

The following formats are supported:

none No profile output. This does not disable profile recording.
tau_snapshot Tau snapshot format.
cube4 Stores the sum for every metric per callpath per location in Cube4 format.
cube_tuple Stores an extended set of statistics in Cube4 format.
thread_sum Sums all locations within a location group and stores the data in Cube4 format.
thread_tuple Sums all locations within a location group and store in addition some statistical data about the distribution among the location of a location group.
key_threads Stores the initial location, the slowest location and the fastest location per process. Sums all other locations within a location group. The result is stored in Cube4 format.
cluster_threads Clusters locations within a location group if they have the same calltree structure. Sums locations within a cluster. Stores the result in Cube4 format.
default Default format. If Cube4 is supported, Cube4 is the default else the Tau snapshot format is default.

SCOREP_PROFILING_ENABLE_CLUSTERING Enable clustering

Type: Boolean
Default: true

SCOREP_PROFILING_CLUSTER_COUNT Maximum cluster count for iteration clustering

Type: Number with size suffixes
Default: 64

SCOREP_PROFILING_CLUSTERING_MODE Specifies the level of strictness when comparing call trees for equivalence

Type: Option
Default: subtree

Possible levels:

none/0 No structural similarity required.
subtree/1 The sub-trees structure must match.
subtree_visits/2 The sub-trees structure and the number of visits must match.
mpi/3 The structure of the call-path to MPI calls must match.
   Nodes that are not on an MPI call-path may differ.
mpi_visits/4 Like above, but the number of visits of the MPI calls must match, too.
mpi_visits_all/5 Like above, but the number of visits must match also match on all nodes on the call-path to an MPI function.

SCOREP_PROFILING_CLUSTERED_REGION Name of the clustered region

Type: String
Default: ""

The clustering can only cluster one dynamic region. If more than one dynamic region are defined by the user, the region is clustered which is exited first. If another region should be clustered instead you can specify the region name in this variable. If the variable is unset or empty, the first exited dynamic region is clustered.

SCOREP_PROFILING_ENABLE_CORE_FILES Write .core files if an error occurred

Type: Boolean
Default: false

If an error occurs inside the profiling system, the profiling is disabled. For debugging reasons, it might be feasible to get the state of the local stack at these points. It is not recommended to enable this feature for large scale measurements.

SCOREP_TRACING_USE_SION Whether or not to use libsion as OTF2 substrate

Type: Boolean
Default: false

SCOREP_TRACING_MAX_PROCS_PER_SION_FILE Maximum number of processes that share one sion file (must be > 0)

Type: Number with size suffixes

Default: 1K

All processes are then evenly distributed over the number of needed files to fulfill this constraint. E.g., having 4 processes and setting the maximum to 3 would result in 2 files each holding 2 processes.

SCOREP_TRACING_CONVERT_CALLING_CONTEXT_EVENTS Write calling context information as a sequence of Enter/Leave events to trace

Type: Boolean

Default: false

When recording the calling context of events (instrumented or sampled) then these could be stored in the trace either as the new CallingContext records from OTF2 or they could be converted to the legacy Enter/Leave records. This can be controlled with this variable, where the former is the false value.
This is only in effect if SCOREP_ENABLING_UNWINDING is on.
Note that enabling this will result in an increase of records per event and also of the loss of the source code locations.
This option exists only for backwards compatibility for tools, which cannot handle the new OTF2 records. This option my thus be removed in future releases.

SCOREP_FILTERING_FILE A file name which contain the filter rules

Type: Path

Default: ""

SCOREP_SUBSTRATE_PLUGINS Specify list of used plugins

Type: String

Default: ""

List of requested substrate plugin names that will be used during program run.

SCOREP_SUBSTRATE_PLUGINS_SEP Separator of substrate plugin names

Type: String

Default: ","

Character that separates plugin names in SCOREP_SUBSTRATE_PLUGINS.

SCOREP_METRIC_PAPI PAPI metric names to measure

> **Type:** String
> **Default:** ""
>
> List of requested PAPI metric names that will be collected during program run.

SCOREP_METRIC_PAPI_PER_PROCESS PAPI metric names to measure per-process

> **Type:** String
> **Default:** ""
>
> List of requested PAPI metric names that will be recorded only by first thread of a process.

SCOREP_METRIC_PAPI_SEP Separator of PAPI metric names

> **Type:** String
> **Default:** ","
>
> Character that separates metric names in SCOREP_METRIC_PAPI and SCOREP_METRIC_PAPI_↩
> PER_PROCESS.

SCOREP_METRIC_RUSAGE Resource usage metric names to measure

> **Type:** String
> **Default:** ""
>
> List of requested resource usage metric names that will be collected during program run.

SCOREP_METRIC_RUSAGE_PER_PROCESS Resource usage metric names to measure per-process

> **Type:** String
> **Default:** ""
>
> List of requested resource usage metric names that will be recorded only by first thread of a process.

SCOREP_METRIC_RUSAGE_SEP Separator of resource usage metric names

> **Type:** String
> **Default:** ","
>
> Character that separates metric names in SCOREP_METRIC_RUSAGE and SCOREP_METRIC_RU↩
> SAGE_PER_PROCESS.

SCOREP_METRIC_PLUGINS Specify list of used plugins

Type: String

Default: ""

List of requested metric plugin names that will be used during program run.

SCOREP_METRIC_PLUGINS_SEP Separator of plugin names

Type: String

Default: ","

Character that separates plugin names in SCOREP_METRIC_PLUGINS.

SCOREP_METRIC_PERF PERF metric names to measure

Type: String

Default: ""

List of requested PERF metric names that will be collected during program run.

SCOREP_METRIC_PERF_PER_PROCESS PERF metric names to measure per-process

Type: String

Default: ""

List of requested PERF metric names that will be recorded only by first thread of a process.

SCOREP_METRIC_PERF_SEP Separator of PERF metric names

Type: String

Default: ","

Character that separates metric names in SCOREP_METRIC_PERF and SCOREP_METRIC_PERF↩
_PER_PROCESS.

SCOREP_SAMPLING_EVENTS Set the sampling event and period: <event>[<period>]

Type: String

Default: "perf_cycles@10000000"

This selects the interrupt source for sampling.
This is only in effect if SCOREP_ENABLE_UNWINDING is on.

Possible values:

- perf event (perf_<event>, see "perf list")
period in number of events, default: 10000000
e.g., perf_cycles@2000000
- PAPI event (PAPI_<event>, see "papi_avail")
period in number of events, default: 10000000
e.g., PAPI_TOT_CYC@2000000
- timer (POSIX timer, invalid for multi-threaded)
period in us, default: 10000
e.g., timer@2000

SCOREP_SAMPLING_SEP Separator of sampling event names

Type: String
Default: ","

Character that separates sampling event names in SCOREP_SAMPLING_EVENTS

SCOREP_TOPOLOGY_PLATFORM Record hardware topology information for this platform, if available.

Type: Boolean
Default: true

SCOREP_TOPOLOGY_PROCESS Record the Process x Thread topology.

Type: Boolean
Default: true

SCOREP_TOPOLOGY_USER Record topologies provided by user instrumentation

Type: Boolean
Default: true

SCOREP_TOPOLOGY_MPI Record MPI cartesian topologies.

Type: Boolean
Default: true

SCOREP_SELECTIVE_CONFIG_FILE A file name which configures selective recording

Type: Path
Default: ""

SCOREP_MPI_MAX_COMMUNICATORS Determines the number of concurrently used communicators per process

Type: Number
Default: 50

SCOREP_MPI_MAX_WINDOWS Determines the number of concurrently used windows for MPI one-sided communication per process

Type: Number
Default: 50

SCOREP_MPI_MAX_EPOCHS Maximum amount of concurrently active access or exposure epochs per process

Type: Number
Default: 50

SCOREP_MPI_MAX_GROUPS Maximum number of concurrently used MPI groups per process

Type: Number
Default: 50

SCOREP_MPI_ENABLE_GROUPS The names of the function groups which are measured

Type: Set
Default: default

Other functions are not measured.

Possible groups are:

all All MPI functions
cg Communicator and group management
coll Collective functions
default Default configuration.
    Includes:
    - cg
    - coll

- env
- io
- p2p
- rma
- topo
- xnonblock

env Environmental management

err MPI Error handling

ext External interface functions

io MPI file I/O

p2p Peer-to-peer communication

misc Miscellaneous

perf PControl

rma One sided communication

spawn Process management

topo Topology

type MPI datatype functions

xnonblock Extended non-blocking events

xreqtest Test events for uncompleted requests

none/no Disable feature

`SCOREP_MPI_MEMORY_RECORDING` Enable tracking of memory allocations done by calls to MPI_AL↩
LOC_MEM and MPI_FREE_MEM

Type: Boolean

Default: false

Requires that the MISC group is also recorded.

`SCOREP_SHMEM_MEMORY_RECORDING` Enable tracking of memory allocations done by calls to the S↩
HMEM allocation API

Type: Boolean

Default: false

`SCOREP_CUDA_ENABLE` CUDA measurement features

Type: Set

Default: yes

SCOREP_CUDA_ENABLE sets the CUDA measurement mode to capture.

Notes:
- Options required by other options will be included automatically.
- idle and pure idle are mutually exclusive.
- The tag (tracing only) indicates that profiling will not yield additional
data from this option.

The following options or sets are available:

runtime   CUDA runtime API

driver   CUDA driver API

kernel   CUDA kernels

kernel_serial   Serialized kernel recording

kernel_counter   Fixed CUDA kernel metrics

kernel_callsite   Track kernel callsites between launch and execution

memcpy   CUDA memory copies

sync   Record implicit and explicit CUDA synchronization

idle   GPU compute idle time

pure_idle   GPU idle time (memory copies are not idle)

gpumemusage   Record CUDA memory (de)allocations as a counter

references   Record references between CUDA activities (tracing only)

dontflushatexit   Disable flushing CUDA activity buffer at program exit

flushatexit   [DEPRECATED] Flush CUDA activity buffer at program exit (see 'dontflushatexit')

default/yes/1   CUDA runtime API and GPU activities.
   Includes:
   - driver
   - kernel
   - kernel_counter
   - memcpy
   - idle
   - sync
   - gpumemusage
   - references

none/no   Disable feature

SCOREP_CUDA_BUFFER   Total memory in bytes for the CUDA record buffer

Type: Number with size suffixes

Default: 1M

SCOREP_CUDA_BUFFER_CHUNK   Chunk size in bytes for the CUDA record buffer (ignored for CUDA 5.5 and earlier)

Type: Number with size suffixes

Default: 8k

SCOREP_OPENCL_ENABLE   OpenCL measurement features

Type: Set

Default: no

Sets the OpenCL measurement mode to capture:

api   OpenCL runtime API

kernel   OpenCL kernels

memcpy   OpenCL buffer reads/writes

default/yes/true/1   OpenCL API and GPU activities.
Includes:
- api
- kernel
- memcpy
none/no   Disable feature

SCOREP_OPENCL_BUFFER_QUEUE   Memory in bytes for the OpenCL command queue buffer

Type: Number with size suffixes
Default: 8k

SCOREP_OPENACC_ENABLE   OpenACC measurement features

Type: Set
Default: no

Sets the OpenACC measurement mode to capture:

regions   OpenACC regions
wait   OpenACC wait operations
enqueue   OpenACC enqueue operations (kernel, upload, download)
device_alloc   OpenACC device memory allocations
kernel_properties   Record kernel properties such as the kernel name as well as the gang, worker and vector size for kernel launch operations
variable_names   Record variable names for OpenACC data allocation and enqueue upload/download
default/yes/1   OpenACC regions, enqueue and wait operations.
Includes:
- regions
- wait
- enqueue
none/no   Disable feature

SCOREP_MEMORY_RECORDING   Memory recording

Type: Boolean
Default: false

Memory (de)allocations are recorded via the libc/C++ API.

SCOREP_KOKKOS_ENABLE   Kokkos measurement features

Type: Set
Default: no

Sets the Kokkos measurement mode to capture:

regions   Kokkos parallel regions

user   Kokkos user regions

malloc   Kokkos memory allocation

memcpy   Kokkos deep copy

default/**yes**/**1**   Kokkos parallel regions, user regions, and allocations

none/**no**   Disable feature

`SCOREP_HIP_ENABLE`   HIP measurement features

Type: Set

Default: yes

Sets the HIP measurement mode to capture:

api   All HIP API calls

kernel   HIP kernels

kernel_callsite   Track kernel callsites between launch and execution. Depends on 'kernel', therefore enables that too

malloc   HIP allocations

memcpy   HIP memory copies

sync   HIP synchronization

user   User instrumentation through ROCTX API

default/**yes**/**1**/**true**   HIP tracing

none/**no**   Disable feature

`SCOREP_HIP_ACTIVITY_BUFFER_SIZE`   HIP device activity buffer size

Type: Number with size suffixes

Default: 1M

Buffer size for device activity events. Must be a power-of-2.

Appendix H

Score-P Address Lookup

Address lookup

Several components in Score-P rely on mapping addresses to source-code locations like filename and line number. Score-P uses the BFD library for this mapping and takes addresses within the instrumented binary as well as within shared objects loaded at program startup into account.

If, however, the binary loads additional shared objects during runtime via `dlopen`, addresses from within these shared objects are by default, not considered for lookup.

To change this behavior, Score-P can intercept `dlopen/dlclose` calls using the dynamic linker's auditing API (man 7 rtld-audit), if desired. To do so, prefix only the application command with `LD_AUDIT=<scorep_↩ prefix>/lib/libscorep_rtld_audit.so`. For an MPI application started with `mpiexec` where environment variables are exported to the application via `−x`, it would, e.g., look like this:

```
$ mpiexec -np 4 -x LD_AUDIT=/opt/scorep/lib/libscorep_rtld_audit.so ./jacobi
```

Note that you must not *export* `LD_AUDIT` or put it in front of launchers like `mpiexec` or `srun`; linker auditing takes place immediately after `LD_AUDIT` is set and results in error messages about missing symbols if not used together with Score-P instrumented binaries.

You can query the installation-specific value of `LD_AUDIT` via:

```
$ scorep-info ldaudit
```

Appendix I

Score-P Compiler Wrapper Usage

Usage

```
scorep-wrapper --create COMPILER [BINDIR]
scorep-<compiler> [COMPILER_FLAGS...]
```

Description

The `scorep-wrapper` script instances (like `scorep-gcc`, see below for a list of provided instances) are intended to simplify configuration and instrumentation of applications where the usual means of instrumentation, i.e., prefixing the compilation command with `scorep`, does not work out of the box. This applies, e.g., to applications that use autotools or CMake.

The intended usage of the wrapper instances is to replace the application's compiler and linker with the corresponding wrapper at configuration time so that they will be used at build time. As compiler and linker commands are usually assigned to build variables like `CC`, `CXX`, or `F77` (e.g., `CC=gcc`), using the corresponding wrapper is as simple as prefixing the value with `scorep-` (e.g., `CC=scorep-gcc`).

E.g., say the original compile command is

```
$ gcc COMPILER_FLAGS...
```

Using the wrapper instead

```
$ scorep-gcc COMPILER_FLAGS...
```

will expand to the following call:

```
$ scorep $SCOREP_WRAPPER_INSTRUMENTER_FLAGS \
    gcc $SCOREP_WRAPPER_COMPILER_FLAGS \
    COMPILER_FLAGS...
```

Used at build time, this expansion performs the desired Score-P instrumentation.

The variables `SCOREP_WRAPPER_INSTRUMENTER_FLAGS` and `SCOREP_WRAPPER_COMPILER_FLAG↩ S` can be used to pass extra arguments to `scorep` and to the compiler, respectively. Please see the *Examples* section below for details.

If the application's build system includes a configuration step (like it is the case for autotools and CMake based projects) the expansion needs to be prevented at this stage (the configuration step compiles and runs lots of small test programs; instrumenting these would in almost all cases result in failure of the configuration). To do so, one needs to set the variable `SCOREP_WRAPPER` to `off` when invoking the configuration command. The wrapper command from above will then expand to the original compile command:

```
$ gcc COMPILER_FLAGS...
```

See also the EXAMPLES section below.

Note on MPI: while many MPI distributions also allow selecting the compiler that the wrapper will invoke via environment variables, this mechanism should not be used with Score-P. Instead, users should use the command line arguments along with the Score-P wrappers for the MPI compiler wrappers:

```
$ scorep-mpicc -cc=/path/to/specific/compiler ...
```

Wrapper Instances

The installation provides wrapper instances based on the compilers used to build Score-P. Run `scorep-wrapper --help` to see a listing of all default instances of the used Score-P installation.

Additional wrapper instances can be created with `scorep-wrapper --create`.

Examples

- The most prominent use case is the CMake build system. As CMake prohibits to change the compiler after the CMake step and it also prohibits that the compiler variable value includes any flags (which renders the usual prefixing `scorep gcc` to a non-functional value). One needs to use a wrapper script which introduces the Score-P instrumenter as a compiler replacement to CMake as early as possible so that they are hard-coded into the generated build files. Apart from that one needs to make sure that the wrappers don't perform their usual instrumentation at this early stage or else the configuration step is likely to fail. However, at make time we want the wrappers to do the actual instrumentation. These goals can be achieved by invoking `cmake` like follows:

```
$ SCOREP_WRAPPER=off cmake .. \
    -DCMAKE_C_COMPILER=scorep-gcc \
    -DCMAKE_CXX_COMPILER=scorep-g++
```

  The `SCOREP_WRAPPER=off` disables the instrumentation only in the environment of the `cmake` command. Subsequent calls to `make` are not affected and will instrument the application as expected.

  Please note, that CMake 3.18.0 and 3.18.1 broke this workflow. But it was reinstated with 3.18.2.

- For autotools based build systems it is recommended to configure in the following way:

```
$ SCOREP_WRAPPER=off ../configure \
    CC=scorep-gcc \
    CXX=scorep-g++ \
    --disable-dependency-tracking
```

- Both autoconf and CMake based build systems, may automatically re-configure the build tree when calling `make`, because some build related files have changed (i.e., `Makefile.am` or `CMakeLists.txt` files). This usage is not supported by the Score-P wrapper. Please re-start the configuration from an empty build directory again as described above.

- To pass options to the `scorep` command in order to diverge from the default instrumentation or to activate verbose output, use the variable `SCOREP_WRAPPER_INSTRUMENTER_FLAGS` at make time:

```
$ make SCOREP_WRAPPER_INSTRUMENTER_FLAGS=--verbose
```

This will result in the execution of:

```
$ scorep --verbose gcc ...
```

- The wrapper also allows to pass flags to the wrapped compiler call by using the variable `SCOREP_WRAP↵PER_COMPILER_FLAGS`:

```
$ make SCOREP_WRAPPER_COMPILER_FLAGS="-D_GNU_SOURCE"
```

Will result in the execution of:

```
$ scorep gcc -D_GNU_SOURCE ...
```

- If there is a need to create additional wrapper instances, e.g., if your build system already uses compiler wrappers, you can do so by calling the `scorep-wrapper` script with the `--create` option:

```
$ scorep-wrapper --create cc
```

This will create a new wrapper instance for the `cc` compiler named `scorep-cc` in the same directory where `scorep-wrapper` resides.

Appendix J

Score-P User Library Wrapping

Usage

```
scorep-libwrap-init --name <wrappername> -x <language> [options] \
    [workspacedir]
```

Initializes the working directory for creating a user library wrapper, named `wrappername`, into `workspacedir`. The final wrapper will be installed into `directory`. `language` has to be either c or c++ plus optionally a standard, e.g., c++11. The default workspace is the current directory.

Options

- `--prefix <directory>` Directory where to install the wrapper into. Defaults to Score-P's installation directory.

- `--display-name` A more descriptive name for this wrapper. Can contain spaces and special characters. Defaults to `--name`.

- `--backend <which>` Backend compiler to be used for building the wrapper. Has to be either vanilla, mpi or shmem.

- `--cppflags "\<flags\>"` Compiler flags for the to-be-wrapped library

- `--ldflags "\<flags\>"` Linker flags for the to-be-wrapped library

- `--libs "\<libraries\>"` To-be-wrapped libraries and their dependencies, e.g., "-lm -lgmp".

- `--static-only-target-library` Add this flag if the to-be-wrapped library does not provide a shared library. This implies switching off runtime wrapping as well as not having a shared wrapper library for linktime wrapping.

- `--update` Do not exit if the working directory is already initialized. Overwrites `Makefile` and creates any file that is not already present.

- `-h, --help` Display this help message

Overview

User library wrapping enables users to install library wrappers for any C or C++ library they want.

In contrast to wrappers integrated into Score-P, e.g., PThreads, MPI, and OpenCL, user library wrappers do not extract semantic information from the library. It only intercepts function calls and provides their timing information.

Without this mechanism, in order to intercept calls to a library, users need to either build this library with Score-P or add manual instrumentation to the application using the library. Another advantage of user library wrapping is you don't need access to the source code of the to-be-wrapped library. Headers and library files suffice.

Requirements:

To enable this feature, Score-P needs to be built with `libclang` from the LLVM compiler infrastructure. More details can be found in Score-P's `INSTALL` file under "User Library Wrapping".

You can find out whether user library wrapping is enabled in the configure summary or via `scorep-info config-summary` in Section "Score-P (libwrap)".

Linktime vs. runtime wrapping:

There exist two ways to wrap calls to the to-be-wrapped (or target-) library. The main difference lies in when the actual wrapping takes place - at linktime or runtime. While they are in essence the same, they differ in which function calls can be intercepted. Specifically:

- linktime: Wraps calls originating from object files that are part of the linker command line. In addition, calls originating from static libraries are wrapped as well. The actual technique used is the `-wrap` linker flag.

- runtime: Wraps all calls that linktime wrapping would, plus those which originate from already linked shared objects. The actual technique used is replacing the original function of the target library and using `dlopen` and `dlsym` in the wrapper to open the target library plus finding and calling the original function.

You can choose linktime or runtime wrapping manually via the `--libwrap` flag by prefixing the wrapper name with either `linktime:` or `runtime:`.

When creating a wrapper, the build infrastructure automatically builds all wrapper libraries supported by your Score-P installation. If the target library only provides a static library, the runtime wrapper and the shared wrapper library of the linktime wrapper will not work. In this case you should reinitialize the workspace by appending `--update --static-only-target-library` to your `scorep-libwrap-init` command line.

Workflow for wrapping a library

Step 1: Initialize the working directory

To initialize the working directory for a new library wrapper use the `scorep-libwrap-init` command. `scorep-libwrap-init` has a number of obligatory and optional arguments. You need to supply:

- A name for the wrapper library via `--name`, and

- The used programming language (either `c` or `c++`) via `-x`. Optionally you can add a specific standard, e.g., `c++11`.

Optional arguments:

- The last argument for `scorep-libwrap-init` specifies the working directory. It defaults to the current directory.

- `--prefix` contains the installation prefix. It defaults to Score-P's installation directory.

- `--display-name` contains a more descriptive name for the wrapper.

- `--backend` lets you choose the backend compiler between vanilla/none, mpi and shmem. If applications using the library use a special compiler like `mpicc` or `oshcc`, you have to change that value to the corresponding backend.

- `--cppflags` contains the preprocessing/compiler flags required to compile an application using the target library.

- `--ldflags` contains the linker flags required to link an application using the target library.

- `--libs` contains the target library and libraries it depends on as a space-separated list of `-l`-flags.

- `--static-only-target-library` specifies whether or not the target library is only installed as a static library.

- `--update` overwrites `Makefile` in order to update the current wrapper, instead of exiting due to the working directory already being initialized.

Example:

```
$ scorep-libwrap-init --name=mylib --display-name="My Lib" -x c \
    --cppflags="-DSOME_DEFINE -I/opt/mylib/include -I/opt/somelib/include" \
    --ldflags="-L/opt/mylib/lib -L/opt/somelib/lib" \
    --libs="-lmylib -lsomelib -lz -lm" \
    $(pwd)/mylibwrapper
```

This step tries to find shared library files matching the supplied `--libs`-flags. During application runtime, these will be used for calling `dlopen` and opening the target library and depending libraries. If `scorep-libwrap-init` cannot find them, it will provide a warning. You can add these shared libraries manually to `Makefile` via the `LW_LIBLIST` variable. If you do not intend to use runtime wrapping, and only use linktime wrapping, you can ignore this warning. It is only needed for runtime wrapping via `dlopen`.

On completion, `scorep-libwrap-init` prints a short-form of how to build and use the wrapper library to the terminal.

`scorep-libwrap-init` creates a number of files in the working directory:

- `libwrap.c` just includes `libwrap.h` and should not be changed.

- `libwrap.h` needs to be edited to contain the headers typically included when using the target library.

- `main.c` will be used to compile and link test programs to make sure the target library and wrapper work correctly.

- `Makefile` can be modified and will be guide you through user library wrapper creation workflow.

- `*.filter` is a Score-P filter file used for excluding/including certain files and functions from wrapping. By default it excludes every file except the ones in the directories specified via `-I` arguments in the cppflags. This means, hopefully, only the target library will be wrapped, and sub-headers from, e.g., `/usr/include` will not be wrapped.

- `README.md` is the file containing this text

Step 2: Add library headers

The next step is to add `#include`-statements to `libwrap.h`. The included files should be headers that applications using the target library usually include.

Step 3: Create a simple example application

After this, add some basic calls to the target library to `main.c`. This file will later be compiled, linked and executed to test whether the target library and wrapper work correctly.

Step 4: Further configure the build parameters

There are a number of variables at the top of `Makefile` that can be adjusted:

- `LW_NAME` is the name of the wrapper library (same as the `--name`-argument)

- `LW_DISPLAY_NAME` is the display name of the wrapper library (same as the `--display-name`-argument)

- `LW_PREFIX` is the installation directory (same as the `--prefix`-argument)

- `LW_BACKEND` contains the compiler backend (same as the `--backend`-argument)

- `CPPFLAGS` contains the preprocessing flags (same as the `--cppflags`-argument)

- `LDFLAGS` contains the linker flags (same as the `--ldflags`-argument)

- `LIBS` contains the target library and libraries on which the target library depends (same as the `--libs`-argument)

- `LW_LIBLIST` shared libraries found according to the LIBS variable. This is internally used for runtime wrapping. This list of files will be used to call `dlopen` at runtime in order to open the target library and corresponding dependencies. File names can have paths.

There are more variables that hopefully need no manual adjustment.

Step 5: Build the wrapper

Run

```
$ make
```

Possible errors:

- Cannot find any function to wrap: Reasons for this can be that there are no include statements in libwrap.h or that the filter removes all functions. `*.filter` contains some examples on how to use it. Make sure that the header files of the target library are included/whitelisted. File names in the filter are matched against the file names and paths of headers that the preprocessing step of the compiler provides.

- There is mismatch between functions found in the header files and the symbols present in the target library: This means the header file analysis found functions that are not present in the library files. Follow the steps in the next section and find out whether these symbols ought to be wrapped or can be ignored.

- Incorrect or unexpected wrapper: Enable verbose and make anew via `make V=1`. If the LLVM/Clang prints out parsing errors, this likely means the preprocessing step executed by your compiler, by default, generates code which is of a newer language standard than what Clang's default is. You can solve this by explicitly specificying the appropriate standard via -x or the `Makefile` variable `LW_LANGUAGE`, e.g., `c++11`.

Possible warnings:

- Ignoring variadic function: variadic functions can not be wrapped, because forwarding ellipsis parameters is not possible in C. The warning describes how to adjust `Makefile` in case a `va_list`-version of this function (like what vprintf is to printf) exists. Add `ellipsis_function:va_list_function` to LI↩BWRAP_ELLIPSIS_MAPPING_SYMBOLS.

- Ignoring function with unknown argument: C functions having an empty parameter list are assumed to have an unknown number of parameters. In case this function has zero parameters, you can add it to `LIBWRA↩P_VARIADIC_IS_VOID_SYMBOLS` in `Makefile`.

This step creates a number of files:

- `main` is main.c linked to the target library. Execute it to make sure the executable works.

- `libscorep_libwrap_*` are the wrapper libraries. Up to four versions exist, depending on your Score-P configuration.

- `libwrap.i` is the preprocessed version of libwrap.h/.c used for analyzing and creating the wrapper code.

- `*.wrap` contains linker flags used for linktime wrapping. It lists every function to be wrapped.

- `scorep_libwrap_*.c` is the source of the wrapper library.

If you change `libwrap.h`, `main.c`, `Makefile` or `*.filter` repeat this step. I.e., execute `make` again. Usually the wrapper creation workflow requires to run `make`, `make check` and after adjusting the wrapper settings to account for errors and warnings you again run `make` and `make check`. It can also take more iterations.

Step 6: Verify the wrapper

Run

```
$ make check
```

For each function found in the header files this step first checks whether there is a corresponding symbol in the library files. Second, it checks whether this symbol is also present when linking without the target library, i.e., is present in the executable or system libraries itself.

A list of functions that have no corresponding symbol in the target library is provided in the filter file `missing.↩ filter`. If there are symbols in it you want to wrap, reconsider your wrapper settings, if not add these symbols to the filter file for this wrapper.

If there are symbols that are also present when linking without the target library, the file `uncertain.filter`, containing these symbols, is created. Calls to these functions might not be intercepted. It is up to you to decide whether this behavior is OK or not. You can, but don't have add some or all of these symbols to the filter file for this wrapper.

As mentioned in the last paragraph of the previous section, repeat `make` and `make check` if you adjust your wrapper.

Step 7: Install the wrapper

Once the wrapper builds without errors, you are ready to install it. Run

```
$ make install
```

This step installs the wrapper into the Score-P installation or the directory specified via `--prefix`, and prints a short description of how to use the wrapper.

Step 8: Verify the installed wrapper

Run

```
$ make installcheck
```

Links `main.c` in up to four different ways using Score-P and the `--libwrap`-flag. There are now a number of files beginning with `main_`. Execute them to make sure they are working. Executing these applications will create Score-P experiment directories with measurements of main.c and the wrapped target library. Inspect the experiments to make sure the wrapper works as expected.

Step 9: Use the wrapper

If you installed the wrapper to somewhere other than the Score-P installation via the `--prefix`-flag, add the appropriate prefix to `SCOREP_LIBWRAP_PATH`.

```
$ export SCOREP_LIBWRAP_PATH=$SCOREP_LIBWRAP_PATH:<prefix>
```

You can then use Score-P to link your application in the usual way and additionally provide `--libwrap=<wrappername>` to enable library wrapping for the target library. Note, library wrapping happens at link time. Thus, you can apply the `--libwrap` flag only to the link command of the application. No need to re-compile the whole application, if this was necessary in the first place.

Example with only relinking the application:

```
$ scorep --libwrap=mylib,myotherlib gcc -o main main.o \
    -L/opt/mylib/lib -L/opt/myotherlib/lib -L/opt/somelib/lib \
    -lmylib -myotherlib -lsomelib -lz -lm
```

Example with both recompiling and linking the application:

```
$ scorep --libwrap=mylib,myotherlib --nocompiler gcc -o main \
    -DSOME_DEFINE -I/opt/mylib/include -I/opt/myotherlib/include
    -I/opt/somelib/include main.c \
    -L/opt/mylib/lib -L/opt/myotherlib/lib -L/opt/somelib/lib \
    -lmylib -myotherlib -lsomelib -lz -lm
```

You can manually choose between linktime and runtime wrapping by prepending `linktime:` or `runtime:` to the wrapper name provided to `--libwrap`.

In order to use multiple wrappers for one application you can use `--libwrap=<wrap-mode>↵ :lib1,lib2,...`. In order to mix linktime and runtime wrapping, you add two `--libwrap`-arguments, one with each mode.

To find out which user library wrappers are installed call

```
$ scorep-info libwrap-summary
```

It lists all found wrappers, either installed into Score-P's installation directory or found via the `SCOREP_LIBWRA↵ P_PATH` environment variable. Optionally you can run

```
$ scorep-info libwrap-summary <wrappername>
```

to show the configuration of a specific wrapper.

Workflow in short

```
$ scorep-libwrap-init --name=<name> -x c --cppflags="<>" --ldflags="<>" \
      --libs="<>" <workingdir>
$ cd <workingdir>
# Add headers to libwrap.h
# Add example to main.c
$ make
$ make check
# Remove errors by adjusting the wrapper settings and adding symbols to
# filter
# Repeat make && make check until all errors are solved
$ make install
$ make installcheck
$ scorep --libwrap=<name> gcc -o myprogram ...
$ ./myprogram
# Inspect the experiment directory scorep-*
```

Implementation details

Score-P user library wrapping enables users to intercept and analyse any C or C++ library. For this we rely on *libclang* to produce a list of functions from the header files of the target library. The wrapper itself relies on symbol-based wrapping supplied by the linker and dynamic linker.

Many C++-libraries rely heavily on inlining and templates. Wrapping libraries based on symbols being present in the target library means that this technique is unable to intercept any inlined function calls.

libclang's ability to detect to-be-inlined functions in header files improved with LLVM 3.9. For LLVM versions $<=$ 3.8 the library wrapper generator will likely generate function wrappers for functions that are inlined and cannot be wrapped. The workflow will provide you with a warning and what to do in this case.

Miscellaneous

```
$ make show-summary
```

This command displays how the wrapper is currently configured. Use this information to, e.g., redo the wrapper, or update the wrapper files via `--update` if, e.g., there is new Score-P version.

FAQ

- Open issues can be found in Score-P's `OPEN_ISSUES` file under "User library wrapping"

Appendix K

Module Documentation

K.1 Score-P User Adapter

Files

- file SCOREP_User.h

  This file contains the interface for the manual user instrumentation.
- file SCOREP_User_Types.h

  This file contains type definitions for manual user instrumentation.

Macros for region instrumentation

- #define SCOREP_USER_REGION_BEGIN(handle, name, type)
- #define SCOREP_USER_REGION_INIT(handle, name, type)
- #define SCOREP_USER_REGION_END(handle) SCOREP_User_RegionEnd(handle);
- #define SCOREP_USER_REGION_ENTER(handle) SCOREP_User_RegionEnter(handle);
- #define SCOREP_USER_REGION_DEFINE(handle) static SCOREP_User_RegionHandle handle = SCO↩
  REP_USER_INVALID_REGION;
- #define SCOREP_USER_FUNC_DEFINE()
- #define SCOREP_USER_FUNC_BEGIN()
- #define SCOREP_USER_FUNC_END() SCOREP_User_RegionEnd(scorep_user_func_handle);
- #define SCOREP_USER_GLOBAL_REGION_DEFINE(handle) SCOREP_User_RegionHandle handle =
  SCOREP_USER_INVALID_REGION;
- #define SCOREP_USER_GLOBAL_REGION_EXTERNAL(handle) extern SCOREP_User_RegionHandle
  handle;

Macros for parameter instrumentation

- #define SCOREP_USER_PARAMETER_INT64(name, value)
- #define SCOREP_USER_PARAMETER_UINT64(name, value)
- #define SCOREP_USER_PARAMETER_STRING(name, value)

Macros to provide user metrics

- #define SCOREP_USER_METRIC_LOCAL(metricHandle)
- #define SCOREP_USER_METRIC_GLOBAL(metricHandle)
- #define SCOREP_USER_METRIC_EXTERNAL(metricHandle) extern SCOREP_SamplingSetHandle metricHandle;
- #define SCOREP_USER_METRIC_INIT(metricHandle, name, unit, type, context) SCOREP_User_InitMetric(&metricHandle, name, unit, type, context);
- #define SCOREP_USER_METRIC_INT64(metricHandle, value)
- #define SCOREP_USER_METRIC_UINT64(metricHandle, value)
- #define SCOREP_USER_METRIC_DOUBLE(metricHandle, value)

Macros for creation of user topologies

- #define SCOREP_USER_CARTESIAN_TOPOLOGY_CREATE(userTopology, name, nDims)
- #define SCOREP_USER_CARTESIAN_TOPOLOGY_ADD_DIM(userTopology, size, periodic, name) SCO↩ REP_User_CartTopologyAddDim(userTopology, size, periodic, name);
- #define SCOREP_USER_CARTESIAN_TOPOLOGY_INIT(userTopology) SCOREP_User_CartTopologyInit(userTopology);
- #define SCOREP_USER_CARTESIAN_TOPOLOGY_SET_COORDS(userTopology, nDims, ...) SCOREP↩ _User_CartTopologySetCoords(userTopology, nDims, __VA_ARGS__);
- #define SCOREP_USER_CARTESIAN_TOPOLOGY_DEFINE(userTopology)

C++ specific macros for region instrumentation

- #define SCOREP_USER_REGION(name, type)

Macros for measurement control

- #define SCOREP_RECORDING_ON() SCOREP_User_EnableRecording();
- #define SCOREP_RECORDING_OFF() SCOREP_User_DisableRecording();
- #define SCOREP_RECORDING_IS_ON() SCOREP_User_RecordingEnabled()

Region types

- #define SCOREP_USER_REGION_TYPE_COMMON 0
- #define SCOREP_USER_REGION_TYPE_FUNCTION 1
- #define SCOREP_USER_REGION_TYPE_LOOP 2
- #define SCOREP_USER_REGION_TYPE_DYNAMIC 4
- #define SCOREP_USER_REGION_TYPE_PHASE 8

Metric types

- #define SCOREP_USER_METRIC_TYPE_INT64 0
- #define SCOREP_USER_METRIC_TYPE_UINT64 1
- #define SCOREP_USER_METRIC_TYPE_DOUBLE 2

Metric contexts

- #define SCOREP_USER_METRIC_CONTEXT_GLOBAL 0
- #define SCOREP_USER_METRIC_CONTEXT_CALLPATH 1

K.1.1   Detailed Description

The user adapter provides a set of macros for user manual instrumentation. The macros are inserted in the source code and call functions of the Score-P runtime system. The user should avoid calling the Score-P runtime functions directly.

For every macro, two definitions are provided: The first one inserts calls to the Score-P runtime system, the second definitions resolve to nothing. Which implementation is used, depends on the definition of SCOREP_USER↩ _ENABLE. If SCOREP_USER_ENABLE is defined, the macros resolve to calls to the Score-P runtime system. If SCOREP_USER_ENABLE is undefined, the user instrumentation is removed by the preprocessor. This flag SCOREP_USER_ENABLE should be set through the instrumentation wrapper tool automatically if user manual instrumentation is enabled.

Every source file which is instrumented must include a header file with the Score-P user instrumentation header. For C/C++ programs, the header file is 'scorep/SCOREP_User.h', for Fortran files, 'scorep/SCOREP_User.inc' must be included. Because the Fortran compilers cannot expand macros, the Fortran source code must be preprocessed by a C or C++ preprocessor, to include the headers and expand the macros. Which Fortran files are passed to the preprocessor depends on the suffix. Usually, suffixes .f and .f90 are not preprocessed, .F and .F90 files are preprocessed. However, this may depend on the used compiler.

K.1.2   Macro Definition Documentation

K.1.2.1   #define SCOREP_RECORDING_IS_ON(   ) SCOREP_User_RecordingEnabled()

In C/C++ it behaves like a function call which returns whether recording is enabled or not. It returns false if the recording of events is disabled, else it returns true.

C/C++ example:

```
1 void foo()
2 {
3   if ( SCOREP_RECORDING_IS_ON() )
4   {
5     // do something
6   }
7 }
```

In Fortran, this macro has a different syntax. An integer variable must be specified as parameter, which is set to non-zero if recording is enabled, else the value is set to zero.

Fortran example:

```
1 subroutine foo
2   integer :: l
3
4   SCOREP_RECORDING_IS_ON( l )
5   if (l .eq. 0) then
6     ! do something
7   end if
8
9 end subroutine foo
```

K.1.2.2   #define SCOREP_RECORDING_OFF(   ) SCOREP_User_DisableRecording();

Disables recording of events. If already disabled, this command has no effect. The control is not restricted to events from the user adapter, but disables the recording of all events.

C/C++ example:

```
1 void foo()
2 {
3   SCOREP_RECORDING_OFF()
4
5   // do something
6
7  SCOREP_RECORDING_ON()
8 }
```

Fortran example:

```
1 subroutine foo
2
3   SCOREP_RECORDING_OFF()
4 ! do something
5   SCOREP_RECORDING_ON()
6
7 end subroutine foo
```

K.1.2.3   #define SCOREP_RECORDING_ON(   ) SCOREP_User_EnableRecording();

Enables recording of events. If already enabled, this command has no effect. The control is not restricted to events from the user adapter, but enables the recording of all events.

C/C++ example:

```
1 void foo()
2 {
3   SCOREP_RECORDING_OFF()
4
5   // do something
6
7  SCOREP_RECORDING_ON()
8 }
```

Fortran example:

```
1 subroutine foo
2
3   SCOREP_RECORDING_OFF()
4 ! do something
5   SCOREP_RECORDING_ON()
6
7 end subroutine foo
```

K.1.2.4   #define SCOREP_USER_CARTESIAN_TOPOLOGY_ADD_DIM(*userTopology,   size,   periodic,   name* **) SCOREP_User_CartTopologyAddDim(userTopology, size, periodic, name);**

This statement defines a dimension for the topology of the given handle.

Parameters

userTopology	The handle that identifies the respective topology.
size	Number of elements in the dimension.
periodic	The periodicity of the dimension; Bool values allowed.
name	The name of the dimension.

K.1.2.5 #define SCOREP_USER_CARTESIAN_TOPOLOGY_CREATE(*userTopology, name, nDims*)

Value:

```
SCOREP_User_CartesianTopologyHandle userTopology =
    SCOREP_USER_INVALID_CARTESIAN_TOPOLOGY; \
  SCOREP_User_CartTopologyCreate( &userTopology, name, nDims );
```

This statement creates an user topology by the given name and initializes it based on name and number of expected dimensions.

Parameters

userTopology	The handle that identifies the topology in subsequent calls.
name	The unique name for the user topology.
nDims	The number of dimensions.

Note

The name of each user topology has to be unique to appear as a distinct topology!

K.1.2.6 #define SCOREP_USER_CARTESIAN_TOPOLOGY_DEFINE(*userTopology*)

This statement defines an user topology in the Fortran case. As a variable definition it has to be placed in the variable section.

Parameters

userTopology	The handle that identifies the topology in subsequent calls.

K.1.2.7 #define SCOREP_USER_CARTESIAN_TOPOLOGY_INIT(*userTopology*) SCOREP_User_CartTopologyInit(userTopology);

Initialize the topology after all dimensions are set.

Parameters

userTopology	The handle that identifies the respective topology.

K.1.2.8 #define SCOREP_USER_CARTESIAN_TOPOLOGY_SET_COORDS(*userTopology, nDims, ...*) SCOREP_User_CartTopologySetCoords(userTopology, nDims, __VA_ARGS__);

This statement defines the coordinates for the topology of the given handle. This call has to be done on the thread/process that should be associated with this set of coordinates, as the mapping will be done implicitly. The number of coordinate parameters in the variable array has to be the same as the number of dimensions for the topology.

Parameters

userTopology	The handle that identifies the respective topology.
nDims	The number of dimensions, which is the number of coordinate parameters in C and the length of the coordinate array in Fortran.
variable	array Coordinate information for each of the nDims dimensions.

The order of macro execution is important. All dimensions have to be added before the init call; the coordinates have to be added after the init call.

Note

> For MPI programs, all calls to SCOREP_USER_CARTESIAN_TOPOLOGY_SET_COORDS have to happen after MPI_Init!

C/C++ Example:

```
1 SCOREP_USER_CARTESIAN_TOPOLOGY_CREATE ( mytopo, "This is a new user topology", 2 );
2 SCOREP_USER_CARTESIAN_TOPOLOGY_ADD_DIM ( mytopo, 2 , 1, "dim1_2" );
3 SCOREP_USER_CARTESIAN_TOPOLOGY_ADD_DIM ( mytopo, ( numprocs + 1 ) / 2 , 1, "dim2_4" );
4 SCOREP_USER_CARTESIAN_TOPOLOGY_INIT ( mytopo );
5 SCOREP_USER_CARTESIAN_TOPOLOGY_SET_COORDS ( mytopo, 2, rank % 2, rank / 2 );
```

Note that SCOREP_USER_CARTESIAN_TOPOLOGY_SET_COORDS takes the location into account on which it is executed. To create coordinates on a thread level, use this macro inside an OpenMP parallel region or in a POSIX thread's start_routine. It is possible to set all coordinates in a serial part of your program though.

Note, for Fortran the CREATE macro is split in a define and create step to allow its placement in the variable section.

Fortran example:

```
1 program main
2   integer :: numprocs
3   SCOREP_USER_CARTESIAN_TOPOLOGY_DEFINE( mytopo )
4   integer, dimension(2) :: coords
5
6   !...
7
8   SCOREP_USER_CARTESIAN_TOPOLOGY_CREATE( mytopo, "This is a new user topology", 2 )
9   SCOREP_USER_CARTESIAN_TOPOLOGY_ADD_DIM( mytopo, 2, 1, "dim1_2" )
10   SCOREP_USER_CARTESIAN_TOPOLOGY_ADD_DIM( mytopo, ( numprocs + 1 ) / 2 , 1, "dim2_4" )
11   SCOREP_USER_CARTESIAN_TOPOLOGY_INIT ( mytopo );
12   coords = (/ MOD( rank, 2 ), rank / 2 /)
13   SCOREP_USER_CARTESIAN_TOPOLOGY_SET_COORDS ( mytopo, 2, coords )
14 end program main
```

K.1.2.9   #define SCOREP_USER_FUNC_BEGIN(   )

Value:

```
static SCOREP_User_RegionHandle        \
    scorep_user_func_handle = SCOREP_USER_INVALID_REGION; \
    SCOREP_User_RegionBegin( &scorep_user_func_handle, &SCOREP_User_LastFileName, \
                        &SCOREP_User_LastFileHandle, SCOREP_USER_FUNCTION_NAME, \
                        SCOREP_USER_REGION_TYPE_FUNCTION, __FILE__,
        __LINE__ );
```

This macro marks the start of a function. It should be inserted at the beginning of the instrumented function. It will generate a region, with the function name.

The C/C++ version of this command takes no arguments. It contains a variable declaration and a function call. Compilers that require a strict separation between declaration block and execution block may fail if this macro is used.

In Fortran one argument is required for the name of the function. Furthermore, the handle must be declared explicitly in Fortran.

Parameters

name	Fortan only: A string containing the name of the function.	

C/C++ example:

```
1 void myfunc()
2 {
3    // declarations
4
5    SCOREP_USER_FUNC_BEGIN()
6
7    // do something
8
9    SCOREP_USER_FUNC_END()
10 }
```

Fortran example:

```
1 subroutine myfunc
2    SCOREP_USER_FUNC_DEFINE()
3    ! more declarations
4
5    SCOREP_USER_FUNC_BEGIN( "myfunc" )
6    ! do something
7    SCOREP_USER_FUNC_END()
8
9 end subroutine myfunc
```

Note that in Fortran the function need to be declared using SCOREP_USER_FUNC_DEFINE before.

K.1.2.10   #define SCOREP_USER_FUNC_DEFINE(   )

This macro is for Fortran only. It declares the handle for a function. Every function handle must be declared in the declaration part of the subroutine or function if the SCOREP_USER_FUNC_BEGIN and SCOREP_USER_FUN↩
C_END macros are used.

Example:

```
1 subroutine myfunc
2    SCOREP_USER_FUNC_DEFINE()
3    ! more declarations
4
5    SCOREP_USER_FUNC_BEGIN( "myfunc" )
6    ! do something
7    SCOREP_USER_FUNC_END()
8
9 end subroutine myfunc
```

Note that in Fortran the function need to be declared using SCOREP_USER_FUNC_DEFINE before.

K.1.2.11   #define SCOREP_USER_FUNC_END(   ) SCOREP_User_RegionEnd(scorep_user_func_handle);

This macro marks the end of a function. It should be inserted at every return point of the instrumented function.

C/C++ example:

```
1 void myfunc()
2 {
3   // declarations
4
5   SCOREP_USER_FUNC_BEGIN()
6
7   // do something
8   if ( some_expression )
9   {
10     SCOREP_USER_FUNC_END()
11     return;
12   }
13
14   SCOREP_USER_FUNC_END()
15 }
```

Fortran example:

```
1 subroutine myfunc
2   SCOREP_USER_FUNC_DEFINE()
3   ! more declarations
4
5   SCOREP_USER_FUNC_BEGIN( "myfunc" )
6   ! do something
7   SCOREP_USER_FUNC_END()
8
9 end subroutine myfunc
```

Note that in Fortran the function need to be declared using SCOREP_USER_FUNC_DEFINE before.

K.1.2.12 #define SCOREP_USER_GLOBAL_REGION_DEFINE(*handle*) SCOREP_User_RegionHandle handle = SCOREP_USER_INVALID_REGION;

This macro defines a region handle in a global scope for usage in more than one code block. If a region is used in multiple source files, only one of them must contain the definition using SCOREP_USER_GLOBAL_REGION←_DEFINE. All other files, in which the global handle is accessed, must only declare the global handle with SCO←REP_USER_GLOBAL_REGION_EXTERNAL(handle). It is possible to use the global handle in more than one code-block. However, code-blocks that share a handle, are handled as they were all the same region. Enter and exit events for global regions are created with SCOREP_USER_REGION_BEGIN and SCOREP_USER_REGIO←N_END, respectively. Its name and type is determined at the first enter event and is not changed on later events, even if other code blocks contains a different name or type in their SCOREP_USER_REGION_BEGIN statement.

This macro is not available in Fortran.

Parameters

handle	A unique name for the handle must be provided. This handle is declared in the macro. This handle is used in the SCOREP_USER_REGION_BEGIN and SCOREP_USER_REGION←_END statements to specify which region is started, or ended. If you are using a Fortran version which has a limited length of code lines, the length of the *handle* parameter must be at most 4 characters, else the declaration line exceeds the allowed length.

C/C++ example:

```
1 // In File1:
2 SCOREP_USER_GLOBAL_REGION_DEFINE( my_global_handle )
3
4 void myfunc()
5 {
6   SCOREP_USER_REGION_BEGIN( my_global_handle, "my_global", SCOREP_USER_REGION_TYPE_PHASE )
7
8   // do something
9
10   SCOREP_USER_REGION_END( my_global_handle )
11 }
```

```
1 // In File2:
2 SCOREP_USER_GLOBAL_EXTERNAL( my_global_handle )
3
4 void foo()
5 {
6   SCOREP_USER_REGION_BEGIN( my_global_handle, "my_global", SCOREP_USER_REGION_TYPE_PHASE )
7
8   // do something
9
10   SCOREP_USER_REGION_END( my_global_handle )
11 }
```

K.1.2.13 #define SCOREP_USER_GLOBAL_REGION_EXTERNAL(*handle*) extern **SCOREP_User_RegionHandle** handle;

This macro declares an externally defined global region. If a region is used in multiple source files, only one of them must contain the definition using SCOREP_USER_GLOBAL_REGION_DEFINE. All other files, in which the global handle is accessed, must only declare the global handle with SCOREP_USER_GLOBAL_REGION_EXT↩ERNAL(handle). It is possible to use the global handle in more than one code-block. However, code-blocks that share a handle, are handled as they were all the same region. Enter and exit events for global regions are created with SCOREP_USER_REGION_BEGIN and SCOREP_USER_REGION_END, respectively. Its name and type is determined at the first enter event and is not changed on later events, even if other code blocks contains a different name or type in their SCOREP_USER_REGION_BEGIN statement.

This macro is not available in Fortran

Parameters

handle	A name for a variable must be provided. This variable name must be the same like for the corresponding SCOREP_USER_GLOBAL_REGION_DEFINE statement. The handle is used in the SCOREP_USER_REGION_BEGIN and SCOREP_USER_REGION_END statements to specify which region is started, or ended.

C/C++ example:

```
1 // In File 1
2 SCOREP_USER_GLOBAL_REGION_DEFINE( my_global_handle )
3
4 void myfunc()
5 {
6   SCOREP_USER_REGION_BEGIN( my_global_handle, "my_global", SCOREP_USER_REGION_TYPE_PHASE )
7
8   // do something
9
10   SCOREP_USER_REGION_END( my_global_handle )
11 }
```

```
1 // In File 2
2 SCOREP_USER_GLOBAL_EXTERNAL( my_global_handle )
3
4 void foo()
5 {
6   SCOREP_USER_REGION_BEGIN( my_global_handle, "my_global", SCOREP_USER_REGION_TYPE_PHASE )
7
8   // do something
9
10   SCOREP_USER_REGION_END( my_global_handle )
11 }
```

K.1.2.14 #define SCOREP_USER_METRIC_CONTEXT_CALLPATH 1

Indicates that a user counter is is measured for every callpath.

K.1.2.15   #define SCOREP_USER_METRIC_CONTEXT_GLOBAL 0

Indicates that a user counter is is measured for the global context.

K.1.2.16   #define SCOREP_USER_METRIC_DOUBLE(*metricHandle, value*)

Value:

```
SCOREP_User_TriggerMetricDouble( \
        metricHandle, value );
```

Triggers a new event for a user counter of a double precision floating point data type. Each user metric must be declared with SCOREP_USER_COUNTER_LOCAL, SCOREP_USER_COUNTER_GLOBAL, or SCOREP_USE←
R_COUNTER_EXTERNAL and initialized with SCOREP_USER_COUNTER_INIT before it is triggered for the first time.

Parameters

metricHandle	The handle of the metric for which a value is given in this statement.
value	The value of the counter. It must be possible for implicit casts to cast it to a double.

Example:

```
1 SCOREP_USER_METRIC_LOCAL( my_local_metric )
2
3 int main()
4 {
5   SCOREP_USER_METRIC_INIT( my_local_metric, "My Metric", "seconds", \
6                       SCOREP_USER_METRIC_TYPE_DOUBLE, \
7                       SCOREP_USER_METRIC_CONTEXT_GLOBAL )
8   // do something
9 }
10
11 void foo()
12 {
13    double my_double = get_some_double_value();
14    SCOREP_USER_METRIC_DOUBLE( my_local_metric, my_double )
15 }
```

Fortran example:

```
1 program myProg
2   SCOREP_USER_METRIC_LOCAL( my_local_metric )
3   real (kind=selected_int_kind(14,200)):: my_real = 24.5
4 ! more declarations
5
6
7   SCOREP_USER_METRIC_INIT( my_local_metric, "My Metric", "seconds", &
8                       SCOREP_USER_METRIC_TYPE_DOUBLE, &
9                       SCOREP_USER_METRIC_CONTEXT_GLOBAL )
10
11 ! do something
12
13   SCOREP_USER_METRIC_DOUBLE( my_local_metric, my_real )
14 end program myProg
```

K.1.2.17   #define SCOREP_USER_METRIC_EXTERNAL(*metricHandle*) extern **SCOREP_SamplingSetHandle metricHandle;**

Declares an externally defined handle for a user metric. Every global metric must be declared only in one file using SCOREP_USER_METRIC_GLOBAL. All other files in which this handle is accessed must declare it with SCOREP_USER_METRIC_EXTERNAL.

This macro is not available in Fortran.

Parameters

metricHandle	The variable name of the handle. it must be the same name as used in the corresponding SCOREP_USER_METRIC_GLOBAL statement.

C/C++ example:

```
1 // In File 1
2 SCOREP_USER_METRIC_GLOBAL( my_global_metric )
3
4 int main()
5 {
6   SCOREP_USER_METRIC_INIT( my_global_metric, "My Global Metric", "seconds", \
7                         SCOREP_USER_METRIC_TYPE_UINT64, \
8                         SCOREP_USER_METRIC_CONTEXT_GLOBAL )
9   // do something
10 }
11
12 void foo()
13 {
14   uint64 my_int = get_some_int_value();
15   SCOREP_USER_METRIC_UINT64( my_global_metric, my_int )
16 }

1 // In File 2
2 SCOREP_USER_METRIC_EXTERNAL( my_global_metric )
3
4 void bar()
5 {
6   uint64 my_int = get_some_int_value();
7   SCOREP_USER_METRIC_UINT64( my_global_metric, my_int )
8 }
```

K.1.2.18   #define SCOREP_USER_METRIC_GLOBAL(*metricHandle*)

Value:

```
SCOREP_SamplingSetHandle metricHandle \
      = SCOREP_INVALID_SAMPLING_SET;
```

Declares a handle for a user metric as a global variable. It must be used if a metric handle is accessed in more than one file. Every global metric must be declared only in one file using SCOREP_USER_METRIC_GLOBAL. All other files in which this handle is accessed must declare it with SCOREP_USER_METRIC_EXTERNAL.

This macro is not available in Fortran.

Parameters

metricHandle	The variable name for the handle. If you are using a Fortran version which has a limited length of code lines, the length of the *handle* parameter must be at most 4 characters, else the declaration line exceeds the allowed length.

C/C++ example:

```
1 // In File 1
2 SCOREP_USER_METRIC_GLOBAL( my_global_metric )
3
4 int main()
5 {
6   SCOREP_USER_METRIC_INIT( my_global_metric, "My Global Metric", "seconds", \
7                         SCOREP_USER_METRIC_TYPE_UINT64, \
8                         SCOREP_USER_METRIC_CONTEXT_GLOBAL )
9   // do something
10 }
11
12 void foo()
13 {
14   uint64 my_int = get_some_int_value();
15   SCOREP_USER_METRIC_UINT64( my_global_metric, my_int )
16 }

1 // In File 2
2 SCOREP_USER_METRIC_EXTERNAL( my_global_metric )
3
4 void bar()
5 {
6   uint64 my_int = get_some_int_value();
7   SCOREP_USER_METRIC_UINT64( my_global_metric, my_int )
8 }
```

K.1.2.19   #define SCOREP_USER_METRIC_INIT(*metricHandle, name, unit, type, context* **) SCOREP_User_InitMetric(**
&metricHandle, name, unit, type, context);

Initializes a new user counter. Each counter must be initialized before it is triggered the first time. The handle must
be declared using SCOREP_USER_METRIC_LOCAL, SCOREP_USER_METRIC_GLOBAL, or SCOREP_USE↩
R_METRIC_EXTERNAL.

Parameters

metricHandle	Provides a variable name of the variable to store the metric handle. The variable is declared by the macro.
name	A string containing a unique name for the counter.
unit	A string containing a the unit of the data.
type	Specifies the data type of the counter. It must be one of the following: SCOREP_USER_↩METRIC_TYPE_INT64, SCOREP_USER_METRIC_TYPE_UINT64, SCOREP_USER_M↩ETRIC_TYPE_DOUBLE. In Fortran is SCOREP_USER_METRIC_TYPE_UINT64 not available.
context	Specifies the context for which the counter is measured. IT must be one of the following: SCOREP_USER_METRIC_CONTEXT_GLOBAL, or SCOREP_USER_METRIC_CONTE↩XT_CALLPATH.

C/C++ example:

```
1 SCOREP_USER_METRIC_LOCAL( my_local_metric )
2
3 int main()
4 {
5   SCOREP_USER_METRIC_INIT( my_local_metric, "My Metric", "seconds", \
6                         SCOREP_USER_METRIC_TYPE_UINT64, \
7                         SCOREP_USER_METRIC_CONTEXT_GLOBAL )
8   // do something
9 }
10
11 void foo()
12 {
13   uint64 my_int = get_some_int_value();
14   SCOREP_USER_METRIC_UINT64( my_local_metric, my_int )
15 }
```

Fortran example:

```
1 program myProg
2   SCOREP_USER_METRIC_LOCAL( my_local_metric )
3   integer (kind=selected_int_kind(8)):: my_int = 19
4 ! more declarations
5
6
7   SCOREP_USER_METRIC_INIT( my_local_metric, "My Metric", "seconds", &
8                         SCOREP_USER_METRIC_TYPE_INT64, &
9                         SCOREP_USER_METRIC_CONTEXT_GLOBAL )
10
11 ! do something
12
13   SCOREP_USER_METRIC_INT64( my_local_metric, my_int )
14 end program myProg
```

K.1.2.20   #define SCOREP_USER_METRIC_INT64(*metricHandle, value* **)**

Value:

```
SCOREP_User_TriggerMetricInt64( \
      metricHandle, value );
```

Triggers a new event for a user counter of a 64 bit integer data type. Each user metric must be declared with
SCOREP_USER_COUNTER_LOCAL, SCOREP_USER_COUNTER_GLOBAL, or SCOREP_USER_COUNTE↩
R_EXTERNAL and initialized with SCOREP_USER_COUNTER_INIT before it is triggered for the first time.

Parameters

metricHandle	The handle of the metric for which a value is given in this statement.
value	The value of the counter. It must be possible for implicit casts to cast it to a 64 bit integer.

C/C++ example:

```
1 SCOREP_USER_METRIC_LOCAL( my_local_metric )
2
3 int main()
4 {
5   SCOREP_USER_METRIC_INIT( my_local_metric, "My Metric", "seconds", \
6                            SCOREP_USER_METRIC_TYPE_INT64, \
7                            SCOREP_USER_METRIC_CONTEXT_GLOBAL )
8   // do something
9 }
10
11 void foo()
12 {
13   int64 my_int = get_some_int_value();
14   SCOREP_USER_METRIC_INT64( my_local_metric, my_int )
15 }
```

Fortran example:

```
1 program myProg
2   SCOREP_USER_METRIC_LOCAL( my_local_metric )
3   integer (kind=selected_int_kind(8)):: my_int = 19
4 ! more declarations
5
6
7   SCOREP_USER_METRIC_INIT( my_local_metric, "My Metric", "seconds", &
8                            SCOREP_USER_METRIC_TYPE_INT64, &
9                            SCOREP_USER_METRIC_CONTEXT_GLOBAL )
10
11 ! do something
12
13   SCOREP_USER_METRIC_INT64( my_local_metric, my_int )
14 end program myProg
```

K.1.2.21 #define SCOREP_USER_METRIC_LOCAL(*metricHandle*)

Value:

```
static SCOREP_SamplingSetHandle \
    metricHandle                                                        \
        = SCOREP_INVALID_SAMPLING_SET;
```

Declares a handle for a user metric. It defines a variable which must be in scope at all places where the metric is used. If it is used in more than one place it need to be a global definition.

Parameters

metricHandle	The name of the variable which will be declared for storing the metric handle.

C/C++ example:

```
1 SCOREP_USER_METRIC_LOCAL( my_local_metric )
2
3 int main()
4 {
5   SCOREP_USER_METRIC_INIT( my_local_metric, "My Metric", "seconds", \
6                            SCOREP_USER_METRIC_TYPE_UINT64, \
7                            SCOREP_USER_METRIC_CONTEXT_GLOBAL )
8   // do something
9 }
10
11 void foo()
12 {
13   uint64 my_int = get_some_int_value();
14   SCOREP_USER_METRIC_UINT64( my_local_metric, my_int )
15 }
```

Fortran example:

```
1 program myProg
2    SCOREP_USER_METRIC_LOCAL( my_local_metric )
3    integer (kind=selected_int_kind(8)):: my_int = 19
4 ! more declarations
5
6
7    SCOREP_USER_METRIC_INIT( my_local_metric, "My Metric", "seconds", &
8                             SCOREP_USER_METRIC_TYPE_INT64, &
9                             SCOREP_USER_METRIC_CONTEXT_GLOBAL )
10
11 ! do something
12
13   SCOREP_USER_METRIC_INT64( my_local_metric, my_int )
14 end program myProg
```

K.1.2.22 #define SCOREP_USER_METRIC_TYPE_DOUBLE 2

Indicates that a user counter is of data type double.

K.1.2.23 #define SCOREP_USER_METRIC_TYPE_INT64 0

Indicates that a user counter is of data type signed 64 bit integer.

K.1.2.24 #define SCOREP_USER_METRIC_TYPE_UINT64 1

Indicates that a user counter is of data type unsigned 64 bit integer.

K.1.2.25 #define SCOREP_USER_METRIC_UINT64(*metricHandle, value*)

Value:

```
SCOREP_User_TriggerMetricUint64( \
      metricHandle, value );
```

Triggers a new event for a user counter of a 64 bit unsigned integer data type. Each user metric must be declared with SCOREP_USER_COUNTER_LOCAL, SCOREP_USER_COUNTER_GLOBAL, or SCOREP_USER_COU↩
NTER_EXTERNAL and initialized with SCOREP_USER_COUNTER_INIT before it is triggered for the first time.

In Fortran is the unsigned integer type metric not available.

Parameters

metricHandle	The handle of the metric for which a value is given in this statement.
value	The value of the counter. It must be possible for implicit casts to cast it to a 64 bit unsigned integer.

Example:

```
1 SCOREP_USER_METRIC_LOCAL( my_local_metric )
2
3 int main()
4 {
5    SCOREP_USER_METRIC_INIT( my_local_metric, "My Metric", "seconds", \
6                             SCOREP_USER_METRIC_TYPE_UINT64, \
7                             SCOREP_USER_METRIC_CONTEXT_GLOBAL )
8    // do something
9 }
10
11 void foo()
12 {
13   uint64 my_int = get_some_int_value();
14   SCOREP_USER_METRIC_UINT64( my_local_metric, my_int )
15 }
```

K.1.2.26 #define SCOREP_USER_PARAMETER_INT64(*name, value*)

Value:

```
{ \
        static SCOREP_User_ParameterHandle scorep_param =
    SCOREP_USER_INVALID_PARAMETER; \
        SCOREP_User_ParameterInt64( &scorep_param, name, value ); }
```

This statement adds a 64 bit signed integer type parameter for parameter-based profiling to the current region. The call-tree for the region is split according to the different values of the parameters with the same name. It is possible to add an arbitrary number of parameters to a region. Each parameter must have a unique name. However, it is not recommended to use more than 1 parameter per region.

Parameters

name	A string containing the name of the parameter.
value	The value of the parameter. It must be possible for implicit casts to cast it to a 64 bit integer.

C/C++ example:

```
1 void myfunc(int64 myint)
2 {
3   SCOREP_USER_REGION_DEFINE( my_region_handle )
4   SCOREP_USER_REGION_BEGIN( my_region_handle, "my_region",SCOREP_USER_REGION_TYPE_COMMON )
5   SCOREP_USER_PARAMETER_INT64("A nice int",myint)
6
7   // do something
8
9   SCOREP_USER_REGION_END( my_region_handle )
10 }
```

K.1.2.27 #define SCOREP_USER_PARAMETER_STRING(*name, value*)

Value:

```
{ \
        static SCOREP_User_ParameterHandle scorep_param =
    SCOREP_USER_INVALID_PARAMETER; \
        SCOREP_User_ParameterString( &scorep_param, name, value ); }
```

This statement adds a string type parameter for parameter-based profiling to the current region. The call-tree for the region is split according to the different values of the parameters with the same name. It is possible to add an arbitrary number of parameters to a region. Each parameter must have a unique name. However, it is not recommended to use more than 1 parameter per region. During one visit it is not allowed to use the same name twice for two different parameters.

Parameters

name	A string containing the name of the parameter.
value	The value of the parameter. It must be a pointer to a C-string (a NULL-terminated string).

C/C++ Example:

```
1 void myfunc(char *mystring)
2 {
3   SCOREP_USER_REGION_DEFINE( my_region_handle )
4   SCOREP_USER_REGION_BEGIN( my_region_handle, "my_region",SCOREP_USER_REGION_TYPE_COMMON )
5   SCOREP_USER_PARAMETER_STRING("A nice string",mystring)
6
7   // do something
8
9   SCOREP_USER_REGION_END( my_region_handle )
10 }
```

K.1.2.28 #define SCOREP_USER_PARAMETER_UINT64(*name, value*)

Value:

```
{ \
        static SCOREP_User_ParameterHandle scorep_param =
    SCOREP_USER_INVALID_PARAMETER; \
        SCOREP_User_ParameterUint64( &scorep_param, name, value ); }
```

This statement adds a 64 bit unsigned integer type parameter for parameter-based profiling to the current region. The call-tree for the region is split according to the different values of the parameters with the same name. It is possible to add an arbitrary number of parameters to a region. Each parameter must have a unique name. However, it is not recommended to use more than 1 parameter per region.

Parameters

name	A string containing the name of the parameter.
value	The value of the parameter. It must be possible for implicit casts to cast it to a 64 bit unsigned integer.

C/C++ example:

```
1 void myfunc(uint64 myuint)
2 {
3   SCOREP_USER_REGION_DEFINE( my_region_handle )
4   SCOREP_USER_REGION_BEGIN( my_region_handle, "my_region",SCOREP_USER_REGION_TYPE_COMMON )
5   SCOREP_USER_PARAMETER_UINT64("A nice unsigned int",myuint)
6
7   // do something
8
9   SCOREP_USER_REGION_END( my_region_handle )
10 }
```

K.1.2.29 #define SCOREP_USER_REGION(*name, type*)

Instruments a code block as a region with the given name. It inserts a local variable of the type class SCOREP_↩
User_Region. Its constructor generates the enter event and its destructor generates the exit event. Thus, only one statement is necessary to instrument the code block. This statement is only in C++ available.

Parameters

name	A string containing the name of the new region. The name should be unique.
type	Specifies the type of the region. Possible values are SCOREP_USER_REGION_TYPE_↩COMMON, SCOREP_USER_REGION_TYPE_FUNCTION, SCOREP_USER_REGION_↩TYPE_LOOP, SCOREP_USER_REGION_TYPE_DYNAMIC, SCOREP_USER_REGION↩_TYPE_PHASE, or a combination of them.

Example:

```
1 void myfunc()
2 {
3   SCOREP_USER_REGION_( "myfunc", SCOREP_USER_REGION_TYPE_FUNCTION )
4
5   // do something
6 }
```

K.1.2.30 #define SCOREP_USER_REGION_BEGIN(*handle, name, type*)

Value:

```
SCOREP_User_RegionBegin( \
        &handle, &SCOREP_User_LastFileName, &SCOREP_User_LastFileHandle, name, \
        type, __FILE__, __LINE__ );
```

This macro marks the start of a user defined region. The SCOREP_USER_REGION_BEGIN and SCOREP_US↩
ER_REGION_END calls of all regions must be correctly nested.

Parameters

handle	The handle of the region to be started. This handle must be declared using SCOREP_US← ER_REGION_DEFINE or SCOREP_USER_GLOBAL_REGION_DEFINE before.
name	A string containing the name of the new region. The name should be unique.
type	Specifies the type of the region. Possible values are SCOREP_USER_REGION_TYPE_← COMMON, SCOREP_USER_REGION_TYPE_FUNCTION, SCOREP_USER_REGION_← TYPE_LOOP, SCOREP_USER_REGION_TYPE_DYNAMIC, SCOREP_USER_REGION← _TYPE_PHASE, or a combination of them.

C/C++ example:

```
1 void myfunc()
2 {
3   SCOREP_USER_REGION_DEFINE( my_region_handle )
4
5   // do something
6
7   SCOREP_USER_REGION_BEGIN( my_region_handle, "my_region",SCOREP_USER_REGION_TYPE_COMMON )
8
9   // do something
10
11   SCOREP_USER_REGION_END( my_region_handle )
12 }
```

Fortran example:

```
1 program myProg
2   SCOREP_USER_REGION_DEFINE( my_region_handle )
3   ! more declarations
4
5   SCOREP_USER_REGION_BEGIN( my_region_handle, "my_region",SCOREP_USER_REGION_TYPE_COMMON )
6   ! do something
7   SCOREP_USER_REGION_END( my_region_handle )
8
9 end program myProg
```

K.1.2.31 #define SCOREP_USER_REGION_DEFINE(*handle*) static **SCOREP_User_RegionHandle** handle = **SCOREP_USER_INVALID_REGION**;

This macro defines a user region handle in a local context. Every user handle must be defined, before it can be used.

Parameters

handle	A unique name for the handle must be provided. This handle is declared in the macro. This handle is used in the SCOREP_USER_REGION_BEGIN and SCOREP_USER_REGION← _END statements to specify which region is started, or ended. If you are using a Fortran version which has a limited length of code lines, the length of the *handle* parameter must be at most 4 characters, else the declaration line exceeds the allowed length.

C/C++ example:

```
1 void myfunc()
2 {
3   SCOREP_USER_REGION_DEFINE( my_region_handle )
4
5   // do something
6
7   SCOREP_USER_REGION_BEGIN( my_region_handle, "my_region",SCOREP_USER_REGION_TYPE_COMMON )
8
9   // do something
10
11   SCOREP_USER_REGION_END( my_region_handle )
12 }
```

Fortran example:

```
1 program myProg
2   SCOREP_USER_REGION_DEFINE( my_region_handle )
3   ! more declarations
4
5   SCOREP_USER_REGION_BEGIN( my_region_handle, "my_region",SCOREP_USER_REGION_TYPE_COMMON )
6   ! do something
7   SCOREP_USER_REGION_END( my_region_handle )
8
9 end program myProg
```

K.1.2.32 #define SCOREP_USER_REGION_END(*handle* **) SCOREP_User_RegionEnd(handle);**

This macro marks the end of a user defined region. The SCOREP_USER_REGION_BEGIN and SCOREP_USE↩
R_REGION_END calls of all regions must be correctly nested.

Parameters

handle	The handle of the region which ended here. It must be the same handle which is used as the start of the region.

C/C++ example:

```
1 void myfunc()
2 {
3   SCOREP_USER_REGION_DEFINE( my_region_handle )
4
5   // do something
6
7   SCOREP_USER_REGION_BEGIN( my_region_handle, "my_region",SCOREP_USER_REGION_TYPE_COMMON )
8
9   // do something
10
11   SCOREP_USER_REGION_END( my_region_handle )
12 }
```

Fortran example:

```
1 program myProg
2   SCOREP_USER_REGION_DEFINE( my_region_handle )
3   ! more declarations
4
5   SCOREP_USER_REGION_BEGIN( my_region_handle, "my_region",SCOREP_USER_REGION_TYPE_COMMON )
6   ! do something
7   SCOREP_USER_REGION_END( my_region_handle )
8
9 end program myProg
```

K.1.2.33 #define SCOREP_USER_REGION_ENTER(*handle* **) SCOREP_User_RegionEnter(handle);**

This macro marks the beginning of a user defined and already initialized region. The SCOREP_USER_REGI↩
ON_BEGIN/SCOREP_USER_REGION_ENTER and SCOREP_USER_REGION_END calls of all regions must be
correctly nested. To initialize the region handle, SCOREP_USER_REGION_INIT or SCOREP_USER_REGION_↩
BEGIN must be called before.

Parameters

handle	The handle of the region which ended here. It must be the same handle which is used as the start of the region.

C/C++ example:

```
1 void myfunc()
2 {
3   SCOREP_USER_REGION_DEFINE( my_region_handle )
4
5   // do something
6
7   SCOREP_USER_REGION_INIT( my_region_handle, "my_region",SCOREP_USER_REGION_TYPE_COMMON )
8   SCOREP_USER_REGION_ENTER( my_region_handle )
9
10   // do something
11
12   SCOREP_USER_REGION_END( my_region_handle )
13 }
```

Fortran example:

```
1 program myProg
2   SCOREP_USER_REGION_DEFINE( my_region_handle )
3   ! more declarations
4
5   SCOREP_USER_REGION_INIT( my_region_handle, "my_region",SCOREP_USER_REGION_TYPE_COMMON )
6   SCOREP_USER_REGION_ENTER( my_region_handle )
7   ! do something
8   SCOREP_USER_REGION_END( my_region_handle )
9
10 end program myProg
```

K.1.2.34 #define SCOREP_USER_REGION_INIT(*handle, name, type*)

Value:

```
SCOREP_User_RegionInit( \
        &handle, &SCOREP_User_LastFileName, &SCOREP_User_LastFileHandle, name, \
        type, __FILE__, __LINE__ );
```

This macro initializes a user defined region. If the region handle is already initialized, no operation is executed.

Parameters

handle	The handle of the region to be started. This handle must be declared using SCOREP_US←ER_REGION_DEFINE or SCOREP_USER_GLOBAL_REGION_DEFINE before.
name	A string containing the name of the new region. The name should be unique.
type	Specifies the type of the region. Possible values are SCOREP_USER_REGION_TYPE_←COMMON, SCOREP_USER_REGION_TYPE_FUNCTION, SCOREP_USER_REGION_←TYPE_LOOP, SCOREP_USER_REGION_TYPE_DYNAMIC, SCOREP_USER_REGION←_TYPE_PHASE, or a combination of them.

C/C++ example:

```
1 void myfunc()
2 {
3   SCOREP_USER_REGION_DEFINE( my_region_handle )
4
5   // do something
6
7   SCOREP_USER_REGION_INIT( my_region_handle, "my_region",SCOREP_USER_REGION_TYPE_COMMON )
8   SCOREP_USER_REGION_ENTER( my_region_handle )
9
10   // do something
11
12   SCOREP_USER_REGION_END( my_region_handle )
13 }
```

Fortran example:

```
1 program myProg
2   SCOREP_USER_REGION_DEFINE( my_region_handle )
3   ! more declarations
4
5   SCOREP_USER_REGION_INIT( my_region_handle, "my_region",SCOREP_USER_REGION_TYPE_COMMON )
6   SCOREP_USER_REGION_ENTER( my_region_handle )
7   ! do something
8   SCOREP_USER_REGION_END( my_region_handle )
9
10 end program myProg
```

K.1.2.35 #define SCOREP_USER_REGION_TYPE_COMMON 0

Region without any specific type.

K.1.2.36 #define SCOREP_USER_REGION_TYPE_DYNAMIC 4

Marks the regions as dynamic.

K.1.2.37 #define SCOREP_USER_REGION_TYPE_FUNCTION 1

Marks the region as being the codeblock of a function.

K.1.2.38 #define SCOREP_USER_REGION_TYPE_LOOP 2

Marks the region as being the codeblock of a loop with the same number of iterations on all processes.

K.1.2.39 #define SCOREP_USER_REGION_TYPE_PHASE 8

Marks the region as being a root node of a phase.

K.2   Public type definitions and enums used in Score-P

Macros

- #define SCOREP_ALL_TARGET_RANKS -1
- #define SCOREP_INVALID_EXIT_STATUS ((int64_t)($\sim$(($\sim$((uint64_t)0u)) $>>$ 1)))
- #define SCOREP_INVALID_LINE_NO 0
- #define SCOREP_INVALID_METRIC SCOREP_MOVABLE_NULL
- #define SCOREP_INVALID_PARADIGM SCOREP_MOVABLE_NULL
- #define SCOREP_INVALID_PID 0
- #define SCOREP_INVALID_REGION SCOREP_MOVABLE_NULL
- #define SCOREP_INVALID_ROOT_RANK -1
- #define SCOREP_INVALID_SAMPLING_SET SCOREP_MOVABLE_NULL
- #define SCOREP_INVALID_SOURCE_FILE SCOREP_MOVABLE_NULL
- #define SCOREP_INVALID_TID 0
- #define SCOREP_IO_UNKNOWN_OFFSET UINT64_MAX
- #define SCOREP_LOCATION_TYPES
- #define SCOREP_MOVABLE_NULL 0
- #define SCOREP_MPI_PROC_NULL -3
- #define SCOREP_MPI_ROOT -2

Typedefs

- typedef uint32_t SCOREP_Allocator_MovableMemory
- typedef SCOREP_Allocator_MovableMemory SCOREP_AnyHandle
- typedef int64_t SCOREP_ExitStatus
- typedef uint32_t SCOREP_LineNo
- typedef SCOREP_AnyHandle SCOREP_MetricHandle
- typedef int SCOREP_MpiRank
- typedef uint64_t SCOREP_MpiRequestId
- typedef SCOREP_AnyHandle SCOREP_ParadigmHandle
- typedef SCOREP_AnyHandle SCOREP_RegionHandle
- typedef SCOREP_AnyHandle SCOREP_SamplingSetHandle
- typedef SCOREP_AnyHandle SCOREP_SourceFileHandle
- typedef struct SCOREP_Task $*$ SCOREP_TaskHandle

Enumerations

- enum SCOREP_CollectiveType {
  SCOREP_COLLECTIVE_BARRIER,
  SCOREP_COLLECTIVE_BROADCAST,
  SCOREP_COLLECTIVE_GATHER,
  SCOREP_COLLECTIVE_GATHERV,
  SCOREP_COLLECTIVE_SCATTER,
  SCOREP_COLLECTIVE_SCATTERV,
  SCOREP_COLLECTIVE_ALLGATHER,
  SCOREP_COLLECTIVE_ALLGATHERV,
  SCOREP_COLLECTIVE_ALLTOALL,
  SCOREP_COLLECTIVE_ALLTOALLV,
  SCOREP_COLLECTIVE_ALLTOALLW,
  SCOREP_COLLECTIVE_ALLREDUCE,
  SCOREP_COLLECTIVE_REDUCE,
  SCOREP_COLLECTIVE_REDUCE_SCATTER,
  SCOREP_COLLECTIVE_REDUCE_SCATTER_BLOCK,
  SCOREP_COLLECTIVE_SCAN,
  SCOREP_COLLECTIVE_EXSCAN,
  SCOREP_COLLECTIVE_CREATE_HANDLE,
  SCOREP_COLLECTIVE_DESTROY_HANDLE,
  SCOREP_COLLECTIVE_ALLOCATE,
  SCOREP_COLLECTIVE_DEALLOCATE,
  SCOREP_COLLECTIVE_CREATE_HANDLE_AND_ALLOCATE,
  SCOREP_COLLECTIVE_DESTROY_HANDLE_AND_DEALLOCATE }

  Types to specify the used collectives in calls to SCOREP_MpiCollectiveBegin and SCOREP_RmaCollectiveBegin.

- enum SCOREP_CommunicatorFlag {
  SCOREP_COMMUNICATOR_FLAG_NONE = 0,
  SCOREP_COMMUNICATOR_FLAG_CREATE_DESTROY_EVENTS = (1 << 0) }

- enum SCOREP_IoAccessMode {
  SCOREP_IO_ACCESS_MODE_NONE = 0,
  SCOREP_IO_ACCESS_MODE_READ_ONLY,
  SCOREP_IO_ACCESS_MODE_WRITE_ONLY,
  SCOREP_IO_ACCESS_MODE_READ_WRITE,
  SCOREP_IO_ACCESS_MODE_EXECUTE_ONLY,
  SCOREP_IO_ACCESS_MODE_SEARCH_ONLY }

- enum SCOREP_IoCreationFlag {
  SCOREP_IO_CREATION_FLAG_NONE = 0,
  SCOREP_IO_CREATION_FLAG_CREATE = (1 << 0),
  SCOREP_IO_CREATION_FLAG_TRUNCATE = (1 << 1),
  SCOREP_IO_CREATION_FLAG_DIRECTORY = (1 << 2),
  SCOREP_IO_CREATION_FLAG_EXCLUSIVE = (1 << 3),
  SCOREP_IO_CREATION_FLAG_NO_CONTROLLING_TERMINAL = (1 << 4),
  SCOREP_IO_CREATION_FLAG_NO_FOLLOW = (1 << 5),
  SCOREP_IO_CREATION_FLAG_PATH = (1 << 6),
  SCOREP_IO_CREATION_FLAG_TEMPORARY_FILE = (1 << 7),
  SCOREP_IO_CREATION_FLAG_LARGEFILE = (1 << 8),
  SCOREP_IO_CREATION_FLAG_NO_SEEK = (1 << 9),
  SCOREP_IO_CREATION_FLAG_UNIQUE = (1 << 10) }

- enum SCOREP_IoOperationFlag {
  SCOREP_IO_OPERATION_FLAG_NONE = 0,
  SCOREP_IO_OPERATION_FLAG_BLOCKING = 0,
  SCOREP_IO_OPERATION_FLAG_NON_BLOCKING = (1 << 0),
  SCOREP_IO_OPERATION_FLAG_COLLECTIVE = (1 << 1),
  SCOREP_IO_OPERATION_FLAG_NON_COLLECTIVE = 0 }

- enum SCOREP_IoOperationMode {

SCOREP_IO_OPERATION_MODE_READ = 0,
SCOREP_IO_OPERATION_MODE_WRITE,
SCOREP_IO_OPERATION_MODE_FLUSH }
- enum SCOREP_IoParadigmType
- enum SCOREP_IoSeekOption {
SCOREP_IO_SEEK_FROM_START = 0,
SCOREP_IO_SEEK_FROM_CURRENT,
SCOREP_IO_SEEK_FROM_END,
SCOREP_IO_SEEK_DATA,
SCOREP_IO_SEEK_HOLE,
SCOREP_IO_SEEK_INVALID }
- enum SCOREP_IoStatusFlag {
SCOREP_IO_STATUS_FLAG_NONE = 0,
SCOREP_IO_STATUS_FLAG_CLOSE_ON_EXEC = (1 << 0),
SCOREP_IO_STATUS_FLAG_APPEND = (1 << 1),
SCOREP_IO_STATUS_FLAG_NON_BLOCKING = (1 << 2),
SCOREP_IO_STATUS_FLAG_ASYNC = (1 << 3),
SCOREP_IO_STATUS_FLAG_SYNC = (1 << 4),
SCOREP_IO_STATUS_FLAG_DATA_SYNC = (1 << 5),
SCOREP_IO_STATUS_FLAG_AVOID_CACHING = (1 << 6),
SCOREP_IO_STATUS_FLAG_NO_ACCESS_TIME = (1 << 7),
SCOREP_IO_STATUS_FLAG_DELETE_ON_CLOSE = (1 << 8) }
- enum SCOREP_Ipc_Datatype

    specifies an inter process communication data types
- enum SCOREP_Ipc_Operation

    specifies an inter process communication operation for reduce function
- enum SCOREP_LocationType { , SCOREP_INVALID_LOCATION_TYPE }
- enum SCOREP_LockType {
SCOREP_LOCK_EXCLUSIVE,
SCOREP_LOCK_SHARED,
SCOREP_INVALID_LOCK_TYPE }
- enum SCOREP_MetricOccurrence {
SCOREP_METRIC_OCCURRENCE_SYNCHRONOUS_STRICT = 0,
SCOREP_METRIC_OCCURRENCE_SYNCHRONOUS = 1,
SCOREP_METRIC_OCCURRENCE_ASYNCHRONOUS = 2,
SCOREP_INVALID_METRIC_OCCURRENCE }

    Types to be used in defining the occurrence of a sampling set.
- enum SCOREP_MetricScope {
SCOREP_METRIC_SCOPE_LOCATION = 0,
SCOREP_METRIC_SCOPE_LOCATION_GROUP = 1,
SCOREP_METRIC_SCOPE_SYSTEM_TREE_NODE = 2,
SCOREP_METRIC_SCOPE_GROUP = 3,
SCOREP_INVALID_METRIC_SCOPE }

    Types to be used in defining the scope of a scoped sampling set.
- enum SCOREP_ParadigmClass { SCOREP_INVALID_PARADIGM_CLASS }

    defines classes of paradigms that are monitored Types:
- enum SCOREP_ParadigmType { SCOREP_INVALID_PARADIGM_TYPE }

    defines paradigms that are be monitored
- enum SCOREP_ParameterType {
SCOREP_PARAMETER_INT64,
SCOREP_PARAMETER_UINT64,
SCOREP_PARAMETER_STRING,
SCOREP_INVALID_PARAMETER_TYPE }

    defines types to be used in defining a parameter for parameter based profiling (SCOREP_Definitions_New$\leftarrow$ Parameter()).
- enum SCOREP_RegionType { , SCOREP_INVALID_REGION_TYPE }

> *specifies a Region*

- enum SCOREP_RmaAtomicType

  specifies a RMA Atomic Operation Type.

- enum SCOREP_RmaSyncLevel

  specifies a RMA synchronization level, used by RMA records to be passed to SCOREP_Rma∗() functions.

- enum SCOREP_RmaSyncType { SCOREP_INVALID_RMA_SYNC_TYPE }

  Type of direct RMA synchronization call.

- enum SCOREP_RmaWindowFlag {
  SCOREP_RMA_WINDOW_FLAG_NONE = 0,
  SCOREP_RMA_WINDOW_FLAG_CREATE_DESTROY_EVENTS = (1 << 0) }

- enum SCOREP_SamplingSetClass {
  SCOREP_SAMPLING_SET_ABSTRACT,
  SCOREP_SAMPLING_SET_CPU,
  SCOREP_SAMPLING_SET_GPU }

  Class of locations which recorded a sampling set.

- enum SCOREP_Substrates_RequirementFlag {
  SCOREP_SUBSTRATES_REQUIREMENT_CREATE_EXPERIMENT_DIRECTORY,
  SCOREP_SUBSTRATES_REQUIREMENT_PREVENT_ASYNC_METRICS,
  SCOREP_SUBSTRATES_REQUIREMENT_PREVENT_PER_HOST_AND_ONCE_METRICS,
  SCOREP_SUBSTRATES_NUM_REQUIREMENTS }

K.2.1 Detailed Description

K.2.2 Macro Definition Documentation

K.2.2.1 #define SCOREP_ALL_TARGET_RANKS -1

Symbolic constant representing an invalid or inapplicable target rank for a lock event.

See also

> SCOREP_RmaAcquireLock() and similar calls.

K.2.2.2 #define SCOREP_INVALID_EXIT_STATUS ((int64_t)(∼((∼((uint64_t)0u)) >> 1)))

Symbolic constant representing an invalid or unknown exit status. Do not use INT64_C here, as this maybe accessed by C++

K.2.2.3 #define SCOREP_INVALID_LINE_NO 0

Symbolic constant representing an invalid or unknown line number.

See also

> SCOREP_Definitions_NewRegion()

K.2.2.4 **#define SCOREP_INVALID_METRIC SCOREP_MOVABLE_NULL**

Symbolic constant representing an invalid or unknown metric definition.

K.2.2.5 **#define SCOREP_INVALID_PARADIGM SCOREP_MOVABLE_NULL**

Symbolic constant representing an invalid or unknown paradigm definition.

K.2.2.6 **#define SCOREP_INVALID_PID 0**

Symbolic constant representing an invalid or unknown process identifier.

K.2.2.7 **#define SCOREP_INVALID_REGION SCOREP_MOVABLE_NULL**

Symbolic constant representing an invalid or unknown region definition.

K.2.2.8 **#define SCOREP_INVALID_ROOT_RANK -1**

Symbolic constant representing an invalid or unknown rank.

See also

SCOREP_MpiCollective()

K.2.2.9 **#define SCOREP_INVALID_SAMPLING_SET SCOREP_MOVABLE_NULL**

Symbolic constant representing an invalid or unknown metric class definition.

K.2.2.10 **#define SCOREP_INVALID_SOURCE_FILE SCOREP_MOVABLE_NULL**

Symbolic constant representing an invalid or unknown source file definition.

K.2.2.11 **#define SCOREP_INVALID_TID 0**

Symbolic constant representing an invalid or unknown thread identifier.

K.2.2.12 **#define SCOREP_IO_UNKNOWN_OFFSET UINT64_MAX**

Symbolic constant representing the offset that is unknown.

K.2.2.13 #define SCOREP_LOCATION_TYPES

Value:

```
SCOREP_LOCATION_TYPE( CPU_THREAD, "CPU thread" ) \
    SCOREP_LOCATION_TYPE( GPU,        "GPU" ) \
    SCOREP_LOCATION_TYPE( METRIC,     "metric location" ) \
```

Types to be used in defining a location (SCOREP_Definitions_NewLocation()).

K.2.2.14 #define SCOREP_MOVABLE_NULL 0

Symbolic constant representing an invalid or NULL handle of type SCOREP_Allocator_MovableMemory.

K.2.2.15 #define SCOREP_MPI_PROC_NULL -3

Symbolic constant representing the MPI constant MPI_PROC_NULL.

See also

SCOREP_MpiCollective()

K.2.2.16 #define SCOREP_MPI_ROOT -2

Symbolic constant representing the MPI constant MPI_ROOT.

See also

SCOREP_MpiCollective()

K.2.3 Typedef Documentation

K.2.3.1 typedef uint32_t SCOREP_Allocator_MovableMemory

Opaque handle to memory that can be easily moved between processes. Used for definitions as they have to be moved during unification.

K.2.3.2 typedef SCOREP_Allocator_MovableMemory SCOREP_AnyHandle

Type of a opaque handle to any definition.

K.2.3.3 typedef int64_t SCOREP_ExitStatus

Type used in specify the exit status of the program.

See also

    ProgramEnd event

K.2.3.4 typedef uint32_t SCOREP_LineNo

Type used in specifying line numbers.

See also

    SCOREP_Definitions_NewRegion()

K.2.3.5 typedef SCOREP_AnyHandle SCOREP_MetricHandle

Type of a opaque handle to a metric definition.

See also

    SCOREP_Definitions_NewMetric()

K.2.3.6 typedef int SCOREP_MpiRank

Type of MPI Ranks. Type of MPI ranks always int.

K.2.3.7 typedef uint64_t SCOREP_MpiRequestId

Type of a MPI Non-blocking communication request id.

K.2.3.8 typedef SCOREP_AnyHandle SCOREP_ParadigmHandle

Type of a opaque handle to a paradigm definition.

See also

    SCOREP_Definitions_NewParadigm()

K.2.3.9   typedef SCOREP_AnyHandle SCOREP_RegionHandle

Type of a opaque handle to a region definition.

See also

> SCOREP_Definitions_NewRegion()

K.2.3.10   typedef SCOREP_AnyHandle SCOREP_SamplingSetHandle

Type of a opaque handle to a sampling set definition.

See also

> SCOREP_Definitions_NewSamplingSet()

K.2.3.11   typedef SCOREP_AnyHandle SCOREP_SourceFileHandle

Type of a opaque handle to a source file definition.

See also

> SCOREP_Definitions_NewSourceFile()

K.2.3.12   typedef struct SCOREP_Task∗ SCOREP_TaskHandle

Task Handle

K.2.4   Enumeration Type Documentation

K.2.4.1   enum SCOREP_CollectiveType

Types to specify the used collectives in calls to *SCOREP_MpiCollectiveBegin* and *SCOREP_RmaCollectiveBegin*.

Enumerator

> ***SCOREP_COLLECTIVE_BARRIER***   The collective is a barrier, e.g., MPI_Barrier(...), shmem_barrier(...), or shmem_barrier_all(...)
>
> ***SCOREP_COLLECTIVE_BROADCAST***   The collective is a broadcast, e.g., MPI_Bcast(...), or shmem_↩ broadcast32(...)
>
> ***SCOREP_COLLECTIVE_GATHER***   The collective is a simple gather operation, e.g., MPI_Gather(...)
>
> ***SCOREP_COLLECTIVE_GATHERV***   The collective is a complex gather operation, e.g., MPI_Gatherv(...)
>
> ***SCOREP_COLLECTIVE_SCATTER***   The collective is a simple scatter operation, e.g., MPI_Scatter(...)
>
> ***SCOREP_COLLECTIVE_SCATTERV***   The collective is a complex scatter operation, e.g., MPI_Scatterv(...)
>
> ***SCOREP_COLLECTIVE_ALLGATHER***   The collective is a simple allgather operation, e.g., MPI_Allgather(...), or shmem_collect64(...)

SCOREP_COLLECTIVE_ALLGATHERV The collective is a complex allgather operation, e.g., MPI_↩
Allgatherv(...)

SCOREP_COLLECTIVE_ALLTOALL The collective is a simple all-to-all communication, e.g., MPI_↩
Alltoall(...)

SCOREP_COLLECTIVE_ALLTOALLV The collective is a all-to-all communication with more options for sizes
and displacements, e.g., MPI_Alltoallv(...)

SCOREP_COLLECTIVE_ALLTOALLW The collective is a generalized all-to-all communication, e.g., MPI↩
_Alltoallw(...)

SCOREP_COLLECTIVE_ALLREDUCE The collective is an allreduce operation, e.g., MPI_Allreduce(...)

SCOREP_COLLECTIVE_REDUCE The collective is a reduce operation, e.g., MPI_Reduce(...), or shmem↩
_longlong_max_to_all(...)

SCOREP_COLLECTIVE_REDUCE_SCATTER The collective is a reduce-scatter operation, which combines
some values and scatters the results, e.g., MPI_Reduce_scatter(...)

SCOREP_COLLECTIVE_REDUCE_SCATTER_BLOCK The collective is a reduce scatter block operation,
e.g., MPI_Reduce_scatter_block(...)

SCOREP_COLLECTIVE_SCAN The collective is a scan operation, where partial reductions of data is com-
puted, e.g., MPI_Scan(...)

SCOREP_COLLECTIVE_EXSCAN The collective is an exclusive scan operation, e.g., MPI_Exscan(...)

SCOREP_COLLECTIVE_CREATE_HANDLE This is used by the tracing substrate to work together with O↩
TF2

SCOREP_COLLECTIVE_DESTROY_HANDLE This is used by the tracing substrate to work together with
OTF2

SCOREP_COLLECTIVE_ALLOCATE This is used by the tracing substrate to work together with OTF2

SCOREP_COLLECTIVE_DEALLOCATE This is used by the tracing substrate to work together with OTF2

SCOREP_COLLECTIVE_CREATE_HANDLE_AND_ALLOCATE This is used by the tracing substrate to
work together with OTF2

SCOREP_COLLECTIVE_DESTROY_HANDLE_AND_DEALLOCATE This is used by the tracing substrate
to work together with OTF2

K.2.4.2 enum SCOREP_CommunicatorFlag

Flags for communicator definitions

Enumerator

SCOREP_COMMUNICATOR_FLAG_NONE No flag is set.

SCOREP_COMMUNICATOR_FLAG_CREATE_DESTROY_EVENTS There will be communicator event/destroy
events.

K.2.4.3 enum SCOREP_IoAccessMode

Access mode of an I/O handle in subsequent I/O operations.

Enumerator

SCOREP_IO_ACCESS_MODE_NONE Unspecified access mode.

SCOREP_IO_ACCESS_MODE_READ_ONLY Read-only access.

SCOREP_IO_ACCESS_MODE_WRITE_ONLY Write-only access.

SCOREP_IO_ACCESS_MODE_READ_WRITE Read-write access.

SCOREP_IO_ACCESS_MODE_EXECUTE_ONLY Execute-only access.

SCOREP_IO_ACCESS_MODE_SEARCH_ONLY Search-only access.

K.2.4.4 enum **SCOREP_IoCreationFlag**

Additional flags specified while creation of an I/O handle.

Enumerator

> ***SCOREP_IO_CREATION_FLAG_NONE*** No flag is set.
>
> ***SCOREP_IO_CREATION_FLAG_CREATE*** If the file does not exist, it will be created.
>
> ***SCOREP_IO_CREATION_FLAG_TRUNCATE*** Truncate file to length 0 if possible.
>
> ***SCOREP_IO_CREATION_FLAG_DIRECTORY*** Open operation will fail if pathname is not a directory.
>
> ***SCOREP_IO_CREATION_FLAG_EXCLUSIVE*** Ensure that this call creates the file.
>
> ***SCOREP_IO_CREATION_FLAG_NO_CONTROLLING_TERMINAL*** File is a terminal device and should not be promoted to a controlling terminal, if none existed before.
>
> ***SCOREP_IO_CREATION_FLAG_NO_FOLLOW*** If pathname is a symbolic link, then the open operation will fail.
>
> ***SCOREP_IO_CREATION_FLAG_PATH*** File is only a location in the filesystem tree and is not suitable for reading and writing.
>
> ***SCOREP_IO_CREATION_FLAG_TEMPORARY_FILE*** Create an unnamed temporary file.
>
> ***SCOREP_IO_CREATION_FLAG_LARGEFILE*** Ensure that the file size can be represented by a 64-bit datatype.
>
> ***SCOREP_IO_CREATION_FLAG_NO_SEEK*** Gives the advice that no reposition will happen on this I/O handle. E.g., no seek operation or similar, only sequential read or write operations.
>
> ***SCOREP_IO_CREATION_FLAG_UNIQUE*** Gives the advice that this will be the only *active* {IoHandle} of the {IoParadigmType} which will operate on the referenced {IoFile} at any time. E.g., no other {IoHandle} of the same {IoParadigmType} and the same {IoFile} will be *active*.

K.2.4.5 enum **SCOREP_IoOperationFlag**

Flags for I/O operations to indicate specific semantics of the operation. Per default any I/O operation is assumed as blocking and non-collective. You can set appropriate flag bits to indicate a deviation from this default semantic.

Enumerator

> ***SCOREP_IO_OPERATION_FLAG_NONE*** No special semantics.
>
> ***SCOREP_IO_OPERATION_FLAG_BLOCKING*** The I/O operation was performed in a blocking mode (default).
>
> ***SCOREP_IO_OPERATION_FLAG_NON_BLOCKING*** The I/O operation was performed in a non-blocking mode.
>
> ***SCOREP_IO_OPERATION_FLAG_COLLECTIVE*** The I/O operation was performed collectively over the communicator of the referenced {IoHandle} handle.
>
> ***SCOREP_IO_OPERATION_FLAG_NON_COLLECTIVE*** The I/O operation was performed in a non-collective mode. (default)

K.2.4.6 enum **SCOREP_IoOperationMode**

Mode of an I/O operation.

Enumerator

> ***SCOREP_IO_OPERATION_MODE_READ*** Read operation.
>
> ***SCOREP_IO_OPERATION_MODE_WRITE*** Write operation.
>
> ***SCOREP_IO_OPERATION_MODE_FLUSH*** Synchronization/flush operation (request and completion).

K.2.4.7   enum **SCOREP_IoParadigmType**

I/O paradigm types.

K.2.4.8   enum **SCOREP_IoSeekOption**

Options for repositioning a file offset with file seek operations.

Enumerator

    SCOREP_IO_SEEK_FROM_START   The offset is set to offset bytes.

    SCOREP_IO_SEEK_FROM_CURRENT   The offset is set to its current location plus offset bytes.

    SCOREP_IO_SEEK_FROM_END   The offset is set to the size of the file plus offset bytes.

    SCOREP_IO_SEEK_DATA   The offset is set to the next location in the file greater than or equal to offset containing data.

    SCOREP_IO_SEEK_HOLE   The offset is set to the next hole in the file greater than or equal to offset.

    SCOREP_IO_SEEK_INVALID   NON-ABI, for internal use only.

K.2.4.9   enum **SCOREP_IoStatusFlag**

Additional status flags of an I/O handle. Status flags can be set when an I/O handle is created and changed during its lifetime.

Enumerator

    SCOREP_IO_STATUS_FLAG_NONE   No flag is set.

    SCOREP_IO_STATUS_FLAG_CLOSE_ON_EXEC   Enable close-on-exec flag.

    SCOREP_IO_STATUS_FLAG_APPEND   Open file in append mode which means I/O write operations are automatically performed at the end of the file.

    SCOREP_IO_STATUS_FLAG_NON_BLOCKING   I/O operations (including the creation) will fail if they would block the issuing process.

    SCOREP_IO_STATUS_FLAG_ASYNC   Enable signal-driven I/O.

    SCOREP_IO_STATUS_FLAG_SYNC   Write operations on the file will complete according to the requirements of synchronized I/O file integrity completion (data and metadata)

    SCOREP_IO_STATUS_FLAG_DATA_SYNC   Write operations on the file will complete according to the requirements of synchronized I/O data integrity completion.

    SCOREP_IO_STATUS_FLAG_AVOID_CACHING   Instruct I/O operations to reduce caching effects, e.g., direct file I/O.

    SCOREP_IO_STATUS_FLAG_NO_ACCESS_TIME   Read access to a file won't update its last access time.

    SCOREP_IO_STATUS_FLAG_DELETE_ON_CLOSE   Delete the file when closing the {IoHandle}.

K.2.4.10   enum SCOREP_Ipc_Datatype

specifies an inter process communication data types

Types:

- SCOREP_IPC_BYTE byte

- SCOREP_IPC_CHAR char

- SCOREP_IPC_UNSIGNED_CHAR unsigned char

- SCOREP_IPC_INT int

- SCOREP_IPC_UNSIGNED unsigned int

- SCOREP_IPC_INT32_T int32_t

- SCOREP_IPC_UINT32_T uint32_t

- SCOREP_IPC_INT64_T int64_t

- SCOREP_IPC_UINT64_T uint64_t

- SCOREP_IPC_DOUBLE double

K.2.4.11   enum SCOREP_Ipc_Operation

specifies an inter process communication operation for reduce function

Types:

- SCOREP_IPC_BAND binary and

- SCOREP_IPC_BOR binary or

- SCOREP_IPC_MIN minimum

- SCOREP_IPC_MAX maximum

- SCOREP_IPC_SUM sum

K.2.4.12   enum SCOREP_LocationType

Enumerator

> ***SCOREP_INVALID_LOCATION_TYPE***   For internal use only.

K.2.4.13   enum SCOREP_LockType

General Lock Type.

Enumerator

> ***SCOREP_LOCK_EXCLUSIVE***   Exclusive lock. No other lock will be granted.
> ***SCOREP_LOCK_SHARED***   Shared lock. Other shared locks will be granted, but no exclusive locks.
> ***SCOREP_INVALID_LOCK_TYPE***   For internal use only.

K.2.4.14 enum SCOREP_MetricOccurrence

Types to be used in defining the occurrence of a sampling set.

Enumerator

> ***SCOREP_METRIC_OCCURRENCE_SYNCHRONOUS_STRICT*** Metric occurs at every region enter and leave.
>
> ***SCOREP_METRIC_OCCURRENCE_SYNCHRONOUS*** Metric occurs only at a region enter and leave, but does not need to occur at every enter/leave.
>
> ***SCOREP_METRIC_OCCURRENCE_ASYNCHRONOUS*** Metric can occur at any place i.e. it is not related to region enter and leaves.
>
> ***SCOREP_INVALID_METRIC_OCCURRENCE*** For internal use only.

K.2.4.15 enum SCOREP_MetricScope

Types to be used in defining the scope of a scoped sampling set.

Enumerator

> ***SCOREP_METRIC_SCOPE_LOCATION*** Scope of a metric is another location.
>
> ***SCOREP_METRIC_SCOPE_LOCATION_GROUP*** Scope of a metric is a location group.
>
> ***SCOREP_METRIC_SCOPE_SYSTEM_TREE_NODE*** Scope of a metric is a system tree node.
>
> ***SCOREP_METRIC_SCOPE_GROUP*** Scope of a metric is a generic group of locations.
>
> ***SCOREP_INVALID_METRIC_SCOPE*** For internal use only.

K.2.4.16 enum SCOREP_ParadigmClass

defines classes of paradigms that are monitored Types:

Types to be used in defining a region (SCOREP_Definitions_NewRegion()). In order to track the origin of a region definition, the adapter needs to provide *his* type.

- SCOREP_PARADIGM_CLASS_MPP refers to any multi processing based paradigms (e.g., MPI, SHMEM)

- SCOREP_PARADIGM_CLASS_THREAD_FORK_JOIN refers to any thread parallel fork-join based paradigms (e.g., OpenMP)

- SCOREP_PARADIGM_CLASS_THREAD_CREATE_WAIT refers to any thread parallel create wait based paradigms (e.g., PThreads)

- SCOREP_PARADIGM_CLASS_ACCELERATOR refers to any accelerator based paradigms

- SCOREP_INVALID_PARADIGM_CLASS for internal use only

Enumerator

> ***SCOREP_INVALID_PARADIGM_CLASS*** For internal use only.

K.2.4.17   enum SCOREP_ParadigmType

defines paradigms that are be monitored

Types:

- SCOREP_PARADIGM_MEASUREMENT refers to Score-P internals

- SCOREP_PARADIGM_USER refers to user instrumentation

- SCOREP_PARADIGM_COMPILER refers to compiler instrumentation

- SCOREP_PARADIGM_SAMPLING refers to sampling

- SCOREP_PARADIGM_MEMORY refers to a memory region (malloc/realloc/...)

- SCOREP_PARADIGM_LIBWRAP refers to region instrumented by user library wrapping

- SCOREP_PARADIGM_MPI refers to MPI instrumentation

- SCOREP_PARADIGM_SHMEM refers to SHMEM instrumentation

- SCOREP_PARADIGM_OPENMP refers to OpenMP instrumentation

- SCOREP_PARADIGM_PTHREAD refers to Pthread instrumentation

- SCOREP_PARADIGM_ORPHAN_THREAD refers to Pthreads that are not instrumented

- SCOREP_PARADIGM_CUDA refers to CUDA instrumentation

- SCOREP_PARADIGM_OPENCL refers to OpenCL instrumentation

- SCOREP_PARADIGM_OPENACC refers to OpenACC instrumentation

- SCOREP_PARADIGM_IO refers to I/O instrumentation

- SCOREP_INVALID_PARADIGM_TYPE for internal use only

Enumerator

> ***SCOREP_INVALID_PARADIGM_TYPE***   For internal use only.

K.2.4.18   enum SCOREP_ParameterType

defines types to be used in defining a parameter for parameter based profiling (SCOREP_Definitions_New↩
Parameter()).

Enumerator

> ***SCOREP_PARAMETER_INT64***   The parameter is an int64_t
>
> ***SCOREP_PARAMETER_UINT64***   The parameter is an uint64_t
>
> ***SCOREP_PARAMETER_STRING***   The parameter is a string
>
> ***SCOREP_INVALID_PARAMETER_TYPE***   For internal use only.

K.2.4.19   enum **SCOREP_RegionType**

specifies a Region

Types to be used in defining a region (SCOREP_Definitions_NewRegion()). These types are currently not used inside the measurement system. This *may* change in the future if we are going to implement phases/dynamic regions etc. inside the measurement system as opposed to inside the adapters or as a postprocessing step. The names should be self explanatory.

Types:

- SCOREP_REGION_UNKNOWN The type of the region is unknown / not defined

- SCOREP_REGION_FUNCTION The region is defined by compiler instrumentation/sampling and defines a code function

- SCOREP_REGION_LOOP The region represents a loop in the source code (used by Opari)

- SCOREP_REGION_USER The region is a user region, e.g., an Opari user region

- SCOREP_REGION_CODE The region represents a code region

- SCOREP_REGION_PHASE (Currently not used)

- SCOREP_REGION_DYNAMIC (Currently not used)

- SCOREP_REGION_DYNAMIC_PHASE (Currently not used)

- SCOREP_REGION_DYNAMIC_LOOP (Currently not used)

- SCOREP_REGION_DYNAMIC_FUNCTION (Currently not used)

- SCOREP_REGION_DYNAMIC_LOOP_PHASE (Currently not used)

- SCOREP_REGION_COLL_ONE2ALL Represents a collective communication region with one2all communication

- SCOREP_REGION_COLL_ALL2ONE Represents a collective communication region with all2one communication

- SCOREP_REGION_COLL_ALL2ALL Represents a collective communication region with all2all communication

- SCOREP_REGION_COLL_OTHER Represents a collective communication region that is neither one2all, nor all2one, nor all2all

- SCOREP_REGION_POINT2POINT Represents a point2point communication region

- SCOREP_REGION_PARALLEL Represents an (OpenMP) parallel region

- SCOREP_REGION_SECTIONS Represents an (OpenMP) sections region

- SCOREP_REGION_SECTION Represents an (OpenMP) section region

- SCOREP_REGION_WORKSHARE Represents an (OpenMP) workshare region

- SCOREP_REGION_SINGLE Represents an (OpenMP) single region

- SCOREP_REGION_MASTER Represents an (OpenMP) master region

- SCOREP_REGION_CRITICAL Represents an (OpenMP) critical region

- SCOREP_REGION_ATOMIC Represents an atomic region

- SCOREP_REGION_BARRIER Represents a barrier

- SCOREP_REGION_IMPLICIT_BARRIER Represents an implicit barrier (that is implicitly given but not explicitly defined)

- SCOREP_REGION_FLUSH Represents an (OpenMP) flush region

- SCOREP_REGION_CRITICAL_SBLOCK Represents an sblock within a (OpenMP) critical region

- SCOREP_REGION_SINGLE_SBLOCK Represents an sblock within a (OpenMP) single region

- SCOREP_REGION_WRAPPER Represents a wrapper region (e.g., from interpositioning)

- SCOREP_REGION_TASK Represents a (OpenMP) task region, within SCOREP_REGION_TASK_CREA←
TE

- SCOREP_REGION_TASK_UNTIED Represents a (OpenMP) untied task region

- SCOREP_REGION_TASK_WAIT Represents a (OpenMP) taskwait region

- SCOREP_REGION_TASK_CREATE Represents a created (OpenMP) task region

- SCOREP_REGION_ORDERED Represents an (OpenMP) ordered region

- SCOREP_REGION_ORDERED_SBLOCK Represents an sblock within a (OpenMP) ordered region

- SCOREP_REGION_ARTIFICIAL Represents an artificial region

- SCOREP_REGION_RMA Represents an RMA region

- SCOREP_REGION_THREAD_CREATE Represents the creation of a thread

- SCOREP_REGION_THREAD_WAIT Represents the creation of a thread

- SCOREP_REGION_ALLOCATE Represents a region where memory is allocated, e.g., MPI_Alloc_mem

- SCOREP_REGION_DEALLOCATE Represents a region where memory is deallocated

- SCOREP_REGION_REALLOCATE Represents a region where memory is reallocated

- SCOREP_REGION_FILE_IO Represents an I/O data operation region

- SCOREP_REGION_FILE_IO_METADATA Represents an I/O metadata operation region (e.g., seek)

- SCOREP_REGION_KERNEL_LAUNCH Represents a function launching a kernel on an accelerator device
(externally mapped to WRAPPER)

- SCOREP_REGION_KERNEL Represents a kernel running on an accelerator (externally mapped to FUN←
CTION)

Enumerator

> ***SCOREP_INVALID_REGION_TYPE*** For internal use only.

K.2.4.20   enum SCOREP_RmaAtomicType

specifies a RMA Atomic Operation Type.

Types:

- SCOREP_RMA_ATOMIC_TYPE_ACCUMULATE accumulate

- SCOREP_RMA_ATOMIC_TYPE_INCREMENT increment

- SCOREP_RMA_ATOMIC_TYPE_TEST_AND_SET test and set

- SCOREP_RMA_ATOMIC_TYPE_COMPARE_AND_SWAP compare and swap

- SCOREP_RMA_ATOMIC_TYPE_SWAP swap

- SCOREP_RMA_ATOMIC_TYPE_FETCH_AND_ADD fetch and add

- SCOREP_RMA_ATOMIC_TYPE_FETCH_AND_INCREMENT fetch and increment

- SCOREP_RMA_ATOMIC_TYPE_ADD add

- SCOREP_INVALID_RMA_ATOMIC_TYPE for internal use only

K.2.4.21 enum SCOREP_RmaSyncLevel

specifies a RMA synchronization level, used by RMA records to be passed to SCOREP_Rma∗() functions.

Types:

- SCOREP_RMA_SYNC_LEVEL_NONE No process synchronization or access completion (e.g., ∗ MPI_↩
  Win_post)

- SCOREP_RMA_SYNC_LEVEL_PROCESS Synchronize processes (e.g., MPI_Win_create/free)

- SCOREP_RMA_SYNC_LEVEL_MEMORY Complete memory accesses (e.g., MPI_Win_complete, MPI_↩
  Win_wait)

- SCOREP_RMA_SYNC_LEVELS for internal use only

K.2.4.22 enum SCOREP_RmaSyncType

Type of direct RMA synchronization call.

Types:

- SCOREP_RMA_SYNC_TYPE_MEMORY Synchronize memory copy.

- SCOREP_RMA_SYNC_TYPE_NOTIFY_IN Incoming remote notification.

- SCOREP_RMA_SYNC_TYPE_NOTIFY_OUT Outgoing remote notification

- SCOREP_INVALID_RMA_SYNC_TYPE for internal use only

Enumerator

> ***SCOREP_INVALID_RMA_SYNC_TYPE*** For internal use only.

K.2.4.23 enum SCOREP_RmaWindowFlag

Flags for RMA window definitions

Enumerator

> ***SCOREP_RMA_WINDOW_FLAG_NONE*** No flag is set.
> ***SCOREP_RMA_WINDOW_FLAG_CREATE_DESTROY_EVENTS*** There will be RMA window event/destroy
> events.

K.2.4.24   enum SCOREP_SamplingSetClass

Class of locations which recorded a sampling set.

Enumerator

    SCOREP_SAMPLING_SET_ABSTRACT   The sampling set is more complicated, e.g., refers to a number of locations.

    SCOREP_SAMPLING_SET_CPU   The sampling set refers to a CPU.

    SCOREP_SAMPLING_SET_GPU   The sampling set refers to a GPU.

K.2.4.25   enum SCOREP_Substrates_RequirementFlag

Enumerator

    SCOREP_SUBSTRATES_REQUIREMENT_CREATE_EXPERIMENT_DIRECTORY   Return true on this feature if your substrate needs the experiment directory. There will be no directory nor configuration log file if no substrate requests it.

    SCOREP_SUBSTRATES_REQUIREMENT_PREVENT_ASYNC_METRICS   Return true on this feature if your substrate can't handle asynchronous metrics. No asynchronous metrics will be recorded if at least on substrate prevents it.

    SCOREP_SUBSTRATES_REQUIREMENT_PREVENT_PER_HOST_AND_ONCE_METRICS   Return true on this feature if your substrate can't handle PER_HOST or ONCE metrics. No PER_HOST or ONCE metrics will be recorded if at least on substrate prevents it.

    SCOREP_SUBSTRATES_NUM_REQUIREMENTS   Non-ABI used internally

Appendix L

Data Structure Documentation

L.1 SCOREP_Metric_Plugin_Info Struct Reference

```
#include <SCOREP_MetricPlugins.h>
```

Data Fields

- int32_t(∗ add_counter)(char ∗metric_name)
- uint64_t delta_t
- void(∗ finalize)(void)
- uint64_t(∗ get_all_values)(int32_t id, SCOREP_MetricTimeValuePair ∗∗time_value_list)
- uint64_t(∗ get_current_value)(int32_t id)
- SCOREP_Metric_Plugin_MetricProperties ∗(∗ get_event_info)(char ∗token)
- bool(∗ get_optional_value)(int32_t id, uint64_t ∗value)
- int32_t(∗ initialize)(void)
- uint32_t plugin_version
- uint64_t reserved [92]
- SCOREP_MetricPer run_per
- void(∗ set_clock_function)(uint64_t(∗clock_time)(void))
- SCOREP_MetricSynchronicity sync
- void(∗ synchronize)(bool is_responsible, SCOREP_MetricSynchronizationMode sync_mode)

L.1.1 Detailed Description

Information on that defines the plugin. All values that are not explicitly defined should be set to 0

L.1.2 Field Documentation

L.1.2.1 int32_t(∗ SCOREP_Metric_Plugin_Info::add_counter) (char ∗metric_name)

Depending on run_per, this function is called per thread, per process, per host, or only on a single thread. Further it is called for each metric as returned by the calls to get_event_info.

The function sets up the measurement for the requested metric and returns a non-negative unique ID which is from now on used to refer to this metric.

Parameters

metric_name	Name of an individual metric

Returns

non-negative ID of requested metric or negative value in cased of failure

L.1.2.2 uint64_t SCOREP_Metric_Plugin_Info::delta_t

Set a specific interval for reading metric values. Score-P will request metric values of a plugin, at the earliest, after delta_t ticks after it was last read. NOTE: This is only a lower limit for the time between two reads

- there is no upper limit. This setting is used by plugins of synchronicity type SCOREP_METRIC_SYNC, SCOREP_METRIC_ASYNC_EVENT, and SCOREP_METRIC_ASYNC. In combination with SCOREP_M↩ETRIC_SYNC, it can be used for metrics that update at known intervals and therefore reduce the over head of reading unchanged values. In combination with SCOREP_METRIC_ASYNC_EVENT it can be used similarly. This value is ignored for SCOREP_METRIC_STRICTLY_SYNC metrics.

L.1.2.3 void(∗ SCOREP_Metric_Plugin_Info::finalize) (void)

This functions is called once per process to clean up all resources used by the metric plugin.

L.1.2.4 uint64_t(∗ SCOREP_Metric_Plugin_Info::get_all_values) (int32_t id, SCOREP_MetricTimeValuePair ∗∗time_value_list)

This function provides the recorded value of the selected metric. It must be implemented by asynchronous metric plugins. The timestamps in the returned list should correspond to the clock provided by set_clock_function. Further, all values (timestamps) should lie within within the following interval: synchronize(SCOREP_METR↩IC_SYNCHRONIZATION_MODE_BEGIN|SCOREP_METRIC_SYNCHRONIZATION_MODE_BEGIN_MPP), synchronize(SCOREP_METRIC_SYNCHRONIZATION_MODE_END). Score-P takes ownership of the ∗time_↩value_list memory. Make sure the memory remains valid and is never modified by the plugin. Score-P will release the memory using `free`. The pointer must be created directly by malloc/calloc etc. directly. Do not use a memory pool / pointer with offset into a larger memory etc.

Parameters

	id	Metric id (see add_counter).
out	*time_value_list*	Pointer to list with return values (pairs of timestamp and value).

See also

SCOREP_MetricSynchronizationMode

Returns

Number of elements within ∗time_value_list

L.1.2.5 uint64_t(∗ SCOREP_Metric_Plugin_Info::get_current_value) (int32_t id)

This function shall provide the current value of a metric. It must be implemented by strictly synchronous metric plugins. It is called according to the run_per specification.

Parameters

id	Metric id (see add_counter).

Returns

Current value of requested metric. For metrics of value_type other than UINT64, the data should be reinterpreted to a UINT64 using a union.

L.1.2.6 SCOREP_Metric_Plugin_MetricProperties∗(∗ SCOREP_Metric_Plugin_Info::get_event_info) (char ∗token)

A user specifies a SCOREP_METRIC_EXAMPLE_PLUGIN=token1,token2,... This function is called once per process and token. Each token can result in any number of metrics (wildcards). The function shall provide the properties of the metrics for this token.

The total list of metrics returned by the calls for all tokens comprises the metrics that will be recorded by the plugin.

Note: The properties-array must contain an additional end entry with name = NULL.

Note: The properties-array memory and all indirect pointers are managed by Score-P now. Make sure the memory remains valid and unmodified. All memory may be released with free by Score-P. Make sure that all provided pointers are created by malloc/strdup/....

Parameters

token	String that describes one or multiple metrics.

Returns

properties Meta data about the metrics available for this token.

L.1.2.7 bool(∗ SCOREP_Metric_Plugin_Info::get_optional_value) (int32_t id, uint64_t ∗value)

This function provides the current value of a metric if available. It must be implemented by synchronous metric plugins. It is called according to the run_per specification.

Parameters

	id	Metric id (see add_counter).
out	*value*	Current value of requested metric. For metrics of value_type other than UIN←↩T64, the data should be reinterpreted to a UINT64 using a union.

Returns

True if value of requested metric was written, otherwise false.

L.1.2.8 int32_t(∗ SCOREP_Metric_Plugin_Info::initialize) (void)

This function is called once per process. It should check that all requirements are met (e.g., are special libraries needed and available, has the user appropriate rights to access implemented metrics). If all requirements are met, data structures used by the plugin can be initialized within this function.

Returns

0 if successful, error code if failure

L.1.2.9   uint32_t SCOREP_Metric_Plugin_Info::plugin_version

Should be set to SCOREP_METRIC_PLUGIN_VERSION (needed for back- and forward compatibility)

L.1.2.10   uint64_t SCOREP_Metric_Plugin_Info::reserved[92]

Reserved space for future features, should be zeroed

L.1.2.11   SCOREP_MetricPer SCOREP_Metric_Plugin_Info::run_per

Defines how many threads should record the metrics of a plugin. For the available options see SCOREP_MetricPer.

L.1.2.12   void(∗ SCOREP_Metric_Plugin_Info::set_clock_function) (uint64_t(∗clock_time)(void))

When this callback is implemented, Score-P calls it once to provide a clock function that allows the plugin to read the current time in Score-P ticks. This should be used by asynchronous metric plugins.

Note: This function is called before initialize.

Parameters

clock_time	Function pointer to Score-P's clock function.

L.1.2.13   SCOREP_MetricSynchronicity SCOREP_Metric_Plugin_Info::sync

Defines how metrics are measured over time and how they are collected by Score-P. This setting influences when and which callback functions are called by Score-P. For the available options see SCOREP_MetricSynchronicity.

L.1.2.14   void(∗ SCOREP_Metric_Plugin_Info::synchronize) (bool is_responsible, SCOREP_MetricSynchronizationMode sync_mode)

This callback is used for stating and stopping the measurement of asynchronous metrics and can also be used for time synchronization purposes. This function is called for all threads in the application, but the threads that handle the metric plugin according to run_per will be marked as is_responsible. The function will be called approximately at the same time for all threads:

- Once the beginning with SCOREP_METRIC_SYNCHRONIZATION_MODE_BEGIN or SCOREP_METRI↩C_SYNCHRONIZATION_MODE_BEGIN_MPP for (non-)MPI programs respectively.

- Once at the end with SCOREP_METRIC_SYNCHRONIZATION_MODE_END For asynchronous metrics, starting and stopping a measurement should be done in this function, not in add_counter.

Parameters

is_responsible	Flag to mark responsibility as per run_per
sync_mode	Mode of synchronization point, e.g. SCOREP_METRIC_SYNCHRONIZATION_MODE↩ _BEGIN, SCOREP_METRIC_SYNCHRONIZATION_MODE_BEGIN_MPP, SCOREP_ME↩ TRIC_SYNCHRONIZATION_MODE_END

See also

SCOREP_MetricSynchronizationMode

The documentation for this struct was generated from the following file:

- SCOREP_MetricPlugins.h

L.2 SCOREP_Metric_Plugin_MetricProperties Struct Reference

Properties describing a metric. Provided by the get_event_info function.

```
#include <SCOREP_MetricPlugins.h>
```

Data Fields

- SCOREP_MetricBase base
- char ∗ description
- int64_t exponent
- SCOREP_MetricMode mode
- char ∗ name
- char ∗ unit
- SCOREP_MetricValueType value_type

L.2.1 Detailed Description

Properties describing a metric. Provided by the get_event_info function.

L.2.2 Field Documentation

L.2.2.1 SCOREP_MetricBase SCOREP_Metric_Plugin_MetricProperties::base

Base of metric: decimal, binary

L.2.2.2 char∗ SCOREP_Metric_Plugin_MetricProperties::description

Additional information about the metric

L.2.2.3   int64_t SCOREP_Metric_Plugin_MetricProperties::exponent

Exponent to scale metric: e.g., 3 for kilo

L.2.2.4   SCOREP_MetricMode SCOREP_Metric_Plugin_MetricProperties::mode

Metric mode: valid combination of ACCUMULATED|ABSOLUTE|RELATIVE + POINT|START|LAST|NEXT

See also

SCOREP_MetricMode

L.2.2.5   char∗ SCOREP_Metric_Plugin_MetricProperties::name

Plugin name

L.2.2.6   char∗ SCOREP_Metric_Plugin_MetricProperties::unit

Unit string of recorded metric

L.2.2.7   SCOREP_MetricValueType SCOREP_Metric_Plugin_MetricProperties::value_type

Value type: signed 64 bit integer, unsigned 64 bit integer, double

The documentation for this struct was generated from the following file:

- SCOREP_MetricPlugins.h

L.3   SCOREP_Metric_Properties Struct Reference

```
#include <SCOREP_MetricTypes.h>
```

Data Fields

- SCOREP_MetricBase base
- const char ∗ description
- int64_t exponent
- SCOREP_MetricMode mode
- const char ∗ name
- SCOREP_MetricProfilingType profiling_type
- SCOREP_MetricSourceType source_type
- const char ∗ unit
- SCOREP_MetricValueType value_type

L.3.1 Detailed Description

Properties describing a metric.

L.3.2 Field Documentation

L.3.2.1 SCOREP_MetricBase SCOREP_Metric_Properties::base

Base of metric values (DECIMAL or BINARY).

L.3.2.2 const char∗ SCOREP_Metric_Properties::description

Long description of the metric.

L.3.2.3 int64_t SCOREP_Metric_Properties::exponent

Exponent to scale metric values (metric_value = value $*$ base$^\wedge$exponent).

L.3.2.4 SCOREP_MetricMode SCOREP_Metric_Properties::mode

Metric mode (valid combination of ACCUMULATED|ABSOLUTE|RELATIVE and POINT|START|LAST|NEXT).

L.3.2.5 const char∗ SCOREP_Metric_Properties::name

Name of the metric.

L.3.2.6 SCOREP_MetricProfilingType SCOREP_Metric_Properties::profiling_type

Profiling type of the metric.

L.3.2.7 SCOREP_MetricSourceType SCOREP_Metric_Properties::source_type

Type of the metric source (e.g. PAPI).

L.3.2.8 const char∗ SCOREP_Metric_Properties::unit

Unit of the metric.

L.3.2.9 SCOREP_MetricValueType SCOREP_Metric_Properties::value_type

Type of the metric value (INT64, UINT64, or DOUBLE).

The documentation for this struct was generated from the following file:

- SCOREP_MetricTypes.h

L.4 SCOREP_MetricTimeValuePair Struct Reference

```
#include <SCOREP_MetricTypes.h>
```

Data Fields

- uint64_t timestamp
- uint64_t value

L.4.1 Detailed Description

Pair of Score-P timestamp and corresponding metric value (used by asynchronous metrics).

L.4.2 Field Documentation

L.4.2.1 uint64_t SCOREP_MetricTimeValuePair::timestamp

Timestamp in Score-P time!

L.4.2.2 uint64_t SCOREP_MetricTimeValuePair::value

Current metric value

The documentation for this struct was generated from the following file:

- SCOREP_MetricTypes.h

L.5 SCOREP_SubstratePluginCallbacks Struct Reference

```
#include <SCOREP_SubstratePlugins.h>
```

Data Fields

- SCOREP_CallingContextHandle(∗   SCOREP_CallingContextHandle_GetParent   )(SCOREP_Calling↩
  ContextHandle handle)
- SCOREP_RegionHandle(∗ SCOREP_CallingContextHandle_GetRegion)(SCOREP_CallingContextHandle
  handle)
- void(∗ SCOREP_ConfigManifestSectionEntry)(FILE ∗out, const char ∗fileName, const char ∗description↩
  FormatString,...)

  Create formated entry in the manifest file consisting out of a name and a description, the latter in printf style and
  terminated with a period.
- void(∗ SCOREP_ConfigManifestSectionHeader)(FILE ∗out, const char ∗section)

  Create formated header in the manifest file.
- const char ∗(∗ SCOREP_GetExperimentDirName)(void)
- int(∗ SCOREP_Ipc_Allgather)(const void ∗sendbuf, void ∗recvbuf, int count, SCOREP_Ipc_Datatype
  datatype)
- int(∗ SCOREP_Ipc_Allreduce)(const void ∗sendbuf, void ∗recvbuf, int count, SCOREP_Ipc_Datatype
  datatype, SCOREP_Ipc_Operation operation)
- int(∗ SCOREP_Ipc_Barrier)(void)
- int(∗ SCOREP_Ipc_Bcast)(void ∗buf, int count, SCOREP_Ipc_Datatype datatype, int root)
- int(∗ SCOREP_Ipc_Gather)(const void ∗sendbuf, void ∗recvbuf, int count, SCOREP_Ipc_Datatype datatype,
  int root)
- int(∗ SCOREP_Ipc_Gatherv)(const void ∗sendbuf, int sendcount, void ∗recvbuf, const int ∗recvcnts, SCO↩
  REP_Ipc_Datatype datatype, int root)
- int(∗ SCOREP_Ipc_GetRank)(void)
- int(∗ SCOREP_Ipc_GetSize)(void)
- int(∗ SCOREP_Ipc_Recv)(void ∗buf, int count, SCOREP_Ipc_Datatype datatype, int source)
- int(∗ SCOREP_Ipc_Reduce)(const void ∗sendbuf, void ∗recvbuf, int count, SCOREP_Ipc_Datatype
  datatype, SCOREP_Ipc_Operation operation, int root)
- int(∗ SCOREP_Ipc_Scatter)(const void ∗sendbuf, void ∗recvbuf, int count, SCOREP_Ipc_Datatype datatype,
  int root)
- int(∗ SCOREP_Ipc_Scatterv)(const void ∗sendbuf, const int ∗sendcounts, void ∗recvbuf, int recvcount, S↩
  COREP_Ipc_Datatype datatype, int root)
- int(∗ SCOREP_Ipc_Send)(const void ∗buf, int count, SCOREP_Ipc_Datatype datatype, int dest)
- void ∗(∗ SCOREP_Location_GetData)(const struct SCOREP_Location ∗locationData, size_t plugin_id)
- uint64_t(∗ SCOREP_Location_GetGlobalId)(const struct SCOREP_Location ∗locationData)
- uint32_t(∗ SCOREP_Location_GetId)(const struct SCOREP_Location ∗locationData)
- const char ∗(∗ SCOREP_Location_GetName)(const struct SCOREP_Location ∗locationData)
- SCOREP_LocationType(∗ SCOREP_Location_GetType)(const struct SCOREP_Location ∗locationData)
- void(∗ SCOREP_Location_SetData)(const struct SCOREP_Location ∗locationData, size_t plugin_id, void
  ∗data)
- void(∗ SCOREP_Metric_WriteAsynchronousMetrics)(struct SCOREP_Location ∗location, SCOREP_↩
  Substrates_WriteMetricsCb cb)

  Tell Score-P to write the current asynchronous metrics to cb.
- void(∗ SCOREP_Metric_WriteStrictlySynchronousMetrics)(struct SCOREP_Location ∗location, uint64_↩
  t timestamp, SCOREP_Substrates_WriteMetricsCb cb)

  Tell Score-P to write the current strictly synchronous metrics to cb.
- void(∗ SCOREP_Metric_WriteSynchronousMetrics)(struct SCOREP_Location ∗location, uint64_t times-
  tamp, SCOREP_Substrates_WriteMetricsCb cb)

  Tell Score-P to write the current synchronous metrics to cb.
- SCOREP_MetricMode(∗ SCOREP_MetricHandle_GetMode)(SCOREP_MetricHandle handle)
- const char ∗(∗ SCOREP_MetricHandle_GetName)(SCOREP_MetricHandle handle)
- SCOREP_MetricProfilingType(∗ SCOREP_MetricHandle_GetProfilingType)(SCOREP_MetricHandle han-
  dle)
- SCOREP_MetricSourceType(∗ SCOREP_MetricHandle_GetSourceType)(SCOREP_MetricHandle handle)
- SCOREP_MetricValueType(∗ SCOREP_MetricHandle_GetValueType)(SCOREP_MetricHandle handle)

- SCOREP_ParadigmClass(∗ SCOREP_ParadigmHandle_GetClass)(SCOREP_ParadigmHandle handle)
- const char ∗(∗ SCOREP_ParadigmHandle_GetName)(SCOREP_ParadigmHandle handle)
- SCOREP_ParadigmType(∗ SCOREP_ParadigmHandle_GetType)(SCOREP_ParadigmHandle handle)
- const char ∗(∗ SCOREP_ParameterHandle_GetName)(SCOREP_ParameterHandle handle)
- SCOREP_ParameterType(∗ SCOREP_ParameterHandle_GetType)(SCOREP_ParameterHandle handle)
- SCOREP_LineNo(∗ SCOREP_RegionHandle_GetBeginLine)(SCOREP_RegionHandle handle)
- const char ∗(∗ SCOREP_RegionHandle_GetCanonicalName)(SCOREP_RegionHandle handle)
- SCOREP_LineNo(∗ SCOREP_RegionHandle_GetEndLine)(SCOREP_RegionHandle handle)
- const char ∗(∗ SCOREP_RegionHandle_GetFileName)(SCOREP_RegionHandle handle)
- uint32_t(∗ SCOREP_RegionHandle_GetId)(SCOREP_RegionHandle handle)
- const char ∗(∗ SCOREP_RegionHandle_GetName)(SCOREP_RegionHandle handle)
- SCOREP_ParadigmType(∗ SCOREP_RegionHandle_GetParadigmType)(SCOREP_RegionHandle handle)
- SCOREP_RegionType(∗ SCOREP_RegionHandle_GetType)(SCOREP_RegionHandle handle)
- const SCOREP_MetricHandle ∗(∗ SCOREP_SamplingSetHandle_GetMetricHandles)(SCOREP_↩ SamplingSetHandle handle)
- SCOREP_MetricOccurrence(∗ SCOREP_SamplingSetHandle_GetMetricOccurrence)(SCOREP_↩ SamplingSetHandle handle)
- uint8_t(∗ SCOREP_SamplingSetHandle_GetNumberOfMetrics)(SCOREP_SamplingSetHandle handle)
- SCOREP_SamplingSetClass(∗ SCOREP_SamplingSetHandle_GetSamplingSetClass)(SCOREP_↩ SamplingSetHandle handle)
- SCOREP_MetricScope(∗ SCOREP_SamplingSetHandle_GetScope)(SCOREP_SamplingSetHandle handle)
- bool(∗ SCOREP_SamplingSetHandle_IsScoped)(SCOREP_SamplingSetHandle handle)
- const char ∗(∗ SCOREP_SourceFileHandle_GetName)(SCOREP_SourceFileHandle handle)
- const char ∗(∗ SCOREP_StringHandle_Get)(SCOREP_StringHandle handle)

L.5.1 Detailed Description

Callbacks that are passed to Substrate plugins via the set_callbacks(...) call. These callbacks can be used by the plugins to access Score-P internal data and functionality.

Developer notice: New functions should be appended at the end of this struct. When extending this list, increase SCOREP_SUBSTRATE_PLUGIN_VERSION

L.5.2 Field Documentation

L.5.2.1 SCOREP_CallingContextHandle(∗ SCOREP_SubstratePluginCallbacks::SCOREP_CallingContextHandle_GetParent) (SCOREP_CallingContextHandle handle)

Returns the parent of a calling context node.

Parameters

handle	handle of the calling context node

Returns

parent handle

L.5.2.2 SCOREP_RegionHandle(∗ SCOREP_SubstratePluginCallbacks::SCOREP_CallingContextHandle_GetRegion) (SCOREP_CallingContextHandle handle)

Returns the region of a calling context node. (see SCOREP_PublicTypes.h)

Parameters

handle	handle of the calling context node

Returns

handle to the region that holds the node

L.5.2.3   const char∗(∗ SCOREP_SubstratePluginCallbacks::SCOREP_GetExperimentDirName) (void)

Returns the Score-P experiment directory. This should be used to write debug/performance data and is available at and after init_mpp is called. Data should be placed under SCOREP_GetExperimentDirName()/(plugin-name)/. Per location data should be placed under SCOREP_GetExperimentDirName()/(plugin-name)/(prefix)(SCOREP↩ _Location_GetGlobalId(location))(suffix). If you want to use the experiment directory, you have to set the SC↩ OREP_SUBSTRATES_REQUIREMENT_EXPERIMENT_DIRECTORY flag in the requirements, see SCOREP_↩ SubstratePluginInfo.get_requirement The name is temporary and the directory might be renamed in the finalization stage.

L.5.2.4   int(∗ SCOREP_SubstratePluginCallbacks::SCOREP_Ipc_Allgather) (const void ∗sendbuf, void ∗recvbuf, int count, SCOREP_Ipc_Datatype datatype)

Gathers data from every process with an equal amount of sent and received data from each process and distributes it to all processes. Can be called after Plugin receives init_mpp() call.

Parameters

sendbuf	pointer to the buffer of the data that should be sent
recvbuf	pointer to buffer there the received data should be stored data is stored in rank order size of the buffer should be big enough for the data of all processes
count	number of elements in buffer
datatype	type of data

Returns

return zero on success and something else if an error occurred

L.5.2.5   int(∗ SCOREP_SubstratePluginCallbacks::SCOREP_Ipc_Allreduce) (const void ∗sendbuf, void ∗recvbuf, int count, SCOREP_Ipc_Datatype datatype, SCOREP_Ipc_Operation operation)

Perform a reduce operation (such as sum, max, logical AND, etc.) with the combined data from every process and distributes the result to all processes. Can be called after Plugin receives init_mpp() call.

Parameters

sendbuf	pointer to the buffer of the data that should be sent
recvbuf	pointer to buffer there the evaluated data should be stored

count	number of elements in buffer
datatype	type of data
operation	operation that should be performed

Returns

return zero on success and something else if an error occurred

L.5.2.6 int(∗ SCOREP_SubstratePluginCallbacks::SCOREP_Ipc_Barrier) (void)

Wait until every process that is part of the MPP paradigm entered the barrier, otherwise return. Can be called after Plugin receives init_mpp() call.

Returns

return zero on success and something else if an error occurred

L.5.2.7 int(∗ SCOREP_SubstratePluginCallbacks::SCOREP_Ipc_Bcast) (void ∗buf, int count, SCOREP_Ipc_Datatype datatype, int root)

Send data to every process within the MPP paradigm (if MPP paradigm is used), including self. Can be called after Plugin receives init_mpp() call.

Parameters

buf	pointer to the buffer of the data that should be sent
count	number of elements in buffer
datatype	type of data
root	rank of the source process

Returns

return zero on success and something else if an error occurred

L.5.2.8 int(∗ SCOREP_SubstratePluginCallbacks::SCOREP_Ipc_Gather) (const void ∗sendbuf, void ∗recvbuf, int count, SCOREP_Ipc_Datatype datatype, int root)

Gathers data from every process within the MPP paradigm (if MPP paradigm is used), root included, with an equal amount of sent for each process.

Can be called after Plugin receives init_mpp() call.

Parameters

sendbuf	pointer to the buffer of the data that should be sent

recvbuf	pointer to buffer there the received data should be stored data is stored in rank order size of the buffer should be big enough for the data of all processes (SCOREP_Ipc_Get↩ Size()∗sizeof(type(datatype)))
count	number of elements in buffer
datatype	type of data
root	rank of the process, that should receive all the data

Returns

> return zero on success and something else if an error occurred

L.5.2.9   int(∗ SCOREP_SubstratePluginCallbacks::SCOREP_Ipc_Gatherv) (const void ∗sendbuf, int sendcount, void ∗recvbuf, const int ∗recvcnts, SCOREP_Ipc_Datatype datatype, int root)

Gathers data from every process, root included, with varying amount of sent data from each process. Can be called after Plugin receives init_mpp() call.

Parameters

sendbuf	pointer to the buffer of the data that should be sent
sendcount	number of elements in sent buffer
recvbuf	pointer to buffer there the received data should be stored data is stored in rank order size of the buffer should be big enough for the data of all processes
recvcnts	array with the number of elements that should be received from each process length should be the number of process
datatype	type of data
root	rank of the process, that should receive all the data

Returns

> return zero on success and something else if an error occurred

L.5.2.10   int(∗ SCOREP_SubstratePluginCallbacks::SCOREP_Ipc_GetRank) (void)

Get the rank of the process. Can be called after Plugin receives init_mpp() call.

Returns

> If MPP paradigm is used get an identifier within the MPP paradigm (e.g., an MPI rank). If no MPP paradigm is used return 0

L.5.2.11   int(∗ SCOREP_SubstratePluginCallbacks::SCOREP_Ipc_GetSize) (void)

Get the number of processes in this parallel program. Can be called after Plugin receives init_mpp() call.

Returns

> If MPP paradigm is used we get the total number of processes that participate in the MPP paradigm, if no MPP paradigm is used return 1

L.5.2.12   int(∗ SCOREP_SubstratePluginCallbacks::SCOREP_Ipc_Recv) (void ∗buf, int count, SCOREP_Ipc_Datatype datatype, int source)

Receive data from a specific process (blocking) Can be called after Plugin receives init_mpp() call. Should not be called if no MPP paradigm is used.

Parameters

buf	pointer to buffer there the received data should be stored
count	number of elements in buffer
datatype	type of data
source	rank of the source process (must be smaller than SCOREP_Ipc_GetSize() and different from SCOREP_Ipc_GetRank())

Returns

return zero on success and something else if an error occurred

L.5.2.13 int(∗ SCOREP_SubstratePluginCallbacks::SCOREP_Ipc_Reduce) (const void ∗sendbuf, void ∗recvbuf, int count, SCOREP_Ipc_Datatype datatype, SCOREP_Ipc_Operation operation, int root)

Perform a reduce operation (such as sum, max, logical AND, etc.) with the combined data from every process. Can be called after Plugin receives init_mpp() call.

Parameters

sendbuf	pointer to the buffer of the data that should be sent
recvbuf	pointer to buffer there the evaluated data should be stored
count	number of elements in buffer
datatype	type of data
operation	operation that should be performed
root	rank of the process, that should receive the evaluated data

Returns

return zero on success and something else if an error occurred

L.5.2.14 int(∗ SCOREP_SubstratePluginCallbacks::SCOREP_Ipc_Scatter) (const void ∗sendbuf, void ∗recvbuf, int count, SCOREP_Ipc_Datatype datatype, int root)

Send data to each process, root included with an equal amount of received data for each process. Can be called after Plugin receives init_mpp() call.

Parameters

sendbuf	pointer to the buffer of the data that should be sent
recvbuf	pointer to buffer there the received data should be stored
count	number of elements in buffer
datatype	type of data
root	rank of the source process

Returns

return zero on success and something else if an error occurred

L.5.2.15 int(∗ SCOREP_SubstratePluginCallbacks::SCOREP_Ipc_Scatterv) (const void ∗sendbuf, const int ∗sendcounts, void ∗recvbuf, int recvcount, SCOREP_Ipc_Datatype datatype, int root)

Send data in parts to each process, root included with a varying amount of received data for each process. Can be called after Plugin receives init_mpp() call.

Parameters

sendbuf	pointer to the buffer of the data that should be sent
sendcounts	array with the number of elements that should be sent to each process length should be the number of process
recvbuf	pointer to buffer there the received data should be stored
recvcount	number of elements in received buffer
datatype	type of data
root	rank of the source process

Returns

> return zero on success and something else if an error occurred

L.5.2.16   int(∗ SCOREP_SubstratePluginCallbacks::SCOREP_Ipc_Send) (const void ∗buf, int count, SCOREP_Ipc_Datatype datatype, int dest)

Send data do a specific process (blocking) within the MPP paradigm. Can be called after Plugin receives init_mpp() call. Should not be called if no MPP paradigm is used.

Parameters

bufpointer	to the buffer of the data that should be sent
count	number of elements in buffer
datatype	type of data
dest	rank of the receiver process (must be smaller than SCOREP_Ipc_GetSize() and different from SCOREP_Ipc_GetRank())

Returns

> return zero on success and something else if an error occurred

L.5.2.17   void∗(∗ SCOREP_SubstratePluginCallbacks::SCOREP_Location_GetData) (const struct SCOREP_Location ∗locationData, size_t plugin_id)

Get location private data for a specific location (see SCOREP_Location_SetData) It is save to use it after the location is created (SCOREP_SubstratePluginInfo.create_location) and before the location is deleted (SCOREP↩ _SubstratePluginInfo.delete_location)

Parameters

location	handle of the location
plugin_id	the id assigned by assign_id

Returns

> the data for this location and this plugin

L.5.2.18   uint64_t(∗ SCOREP_SubstratePluginCallbacks::SCOREP_Location_GetGlobalId) (const struct SCOREP_Location ∗locationData)

Get a unique global id of a location This is only to be used after init_mpp() is called.

Parameters

location	location given from an event

Returns

global location id

L.5.2.19 uint32_t(∗ SCOREP_SubstratePluginCallbacks::SCOREP_Location_GetId) (const struct SCOREP_Location ∗locationData)

Returns the LOCAL id of the location. (see also SCOREP_Location_GetGlobalId)

Parameters

location	location given from an event, or a location function from SCOREP_SubstratePluginInfo

Returns

ID that is unique within the process

L.5.2.20 const char∗(∗ SCOREP_SubstratePluginCallbacks::SCOREP_Location_GetName) (const struct SCOREP_Location ∗locationData)

Get the name of a location

Parameters

location	handle

Returns

location name

L.5.2.21 SCOREP_LocationType(∗ SCOREP_SubstratePluginCallbacks::SCOREP_Location_GetType) (const struct SCOREP_Location ∗locationData)

Returns the type of the location.

Returns

L.5.2.22 void(∗ SCOREP_SubstratePluginCallbacks::SCOREP_Location_SetData) (const struct SCOREP_Location ∗locationData, size_t plugin_id, void ∗data)

Set location private data for a specific location (see SCOREP_Location_GetData) It is save to use it after the location is created (SCOREP_SubstratePluginInfo.create_location) and before the location is deleted (SCOREP↵ _SubstratePluginInfo.delete_location)

Parameters

location	handle of the location
plugin_id	the id assigned by assign_id
the	data for this location and this plugin

L.5.2.23 void(∗ SCOREP_SubstratePluginCallbacks::SCOREP_Metric_WriteAsynchronousMetrics) (struct SCOREP_Location ∗location, SCOREP_Substrates_WriteMetricsCb cb)

Tell Score-P to write the current asynchronous metrics to cb.

This function shall only be called while processing enters, exits, and samples. The callback might be called multiple times if multiple asynchronous sampling sets are present.

Parameters

location	the location of the runtime event, this is reported back via cb
cb	a callback that processes the asynchronous metrics.

L.5.2.24 void(∗ SCOREP_SubstratePluginCallbacks::SCOREP_Metric_WriteStrictlySynchronousMetrics) (struct SCOREP_Location ∗location, uint64_t timestamp, SCOREP_Substrates_WriteMetricsCb cb)

Tell Score-P to write the current strictly synchronous metrics to cb.

This function shall only be called while processing enters, exits, and samples.

Parameters

location	the location of the runtime event, this is reported back via cb
timestamp	the timestamp of the runtime event, this is reported back via cb
cb	a callback that processes the strictly synchronous metrics.

L.5.2.25 void(∗ SCOREP_SubstratePluginCallbacks::SCOREP_Metric_WriteSynchronousMetrics) (struct SCOREP_Location ∗location, uint64_t timestamp, SCOREP_Substrates_WriteMetricsCb cb)

Tell Score-P to write the current synchronous metrics to cb.

This function shall only be called while processing enters, exits, and samples. The callback might be called multiple times if multiple synchronous sampling sets are present.

Parameters

location	the location of the runtime event, this is reported back via cb
timestamp	the timestamp of the runtime event, this is reported back via cb
cb	a callback that processes the synchronous metrics.

L.5.2.26 SCOREP_MetricMode(∗ SCOREP_SubstratePluginCallbacks::SCOREP_MetricHandle_GetMode) (SCOREP_MetricHandle handle)

Returns the mode of a metric.

Parameters

	handle	handle of the local metric definition.

Returns

mode of a metric.

See also

SCOREP_MetricTypes.h

L.5.2.27   const char∗(∗ SCOREP_SubstratePluginCallbacks::SCOREP_MetricHandle_GetName) (SCOREP_MetricHandle handle)

Returns the name of a metric.

Parameters

	handle	handle of the local metric definition.

Returns

name of a metric.

L.5.2.28   SCOREP_MetricProfilingType(∗ SCOREP_SubstratePluginCallbacks::SCOREP_MetricHandle_GetProfilingType) (SCOREP_MetricHandle handle)

Returns the profiling type of a metric.

Parameters

	handle	handle of the local metric definition.

Returns

profiling type of a metric.

See also

SCOREP_MetricTypes.h

L.5.2.29   SCOREP_MetricSourceType(∗ SCOREP_SubstratePluginCallbacks::SCOREP_MetricHandle_GetSourceType) (SCOREP_MetricHandle handle)

Returns the source type of a metric.

Parameters

handle	handle of the local metric definition.

Returns

source type of a metric.

See also

SCOREP_MetricTypes.h

L.5.2.30 SCOREP_MetricValueType(∗ SCOREP_SubstratePluginCallbacks::SCOREP_MetricHandle_GetValueType) (SCOREP_MetricHandle handle)

Returns the value type of a metric.

Parameters

handle	handle of the local metric definition.

Returns

value type of a metric.

See also

SCOREP_MetricTypes.h

L.5.2.31 SCOREP_ParadigmClass(∗ SCOREP_SubstratePluginCallbacks::SCOREP_ParadigmHandle_GetClass) (SCOREP_ParadigmHandle handle)

Returns paradigm class

Parameters

handle	handle of the paradigm

Returns

class (e.g. SCOREP_PARADIGM_CLASS_MPP for MPI paradigm)

See also

SCOREP_PublicTypes.h

L.5.2.32 const char∗(∗ SCOREP_SubstratePluginCallbacks::SCOREP_ParadigmHandle_GetName) (SCOREP_ParadigmHandle handle)

Returns paradigm name

Parameters

handle	handle of the paradigm

Returns

name (e.g., "MPI")

L.5.2.33 SCOREP_ParadigmType(∗ SCOREP_SubstratePluginCallbacks::SCOREP_ParadigmHandle_GetType) (SCOREP_ParadigmHandle handle)

Returns paradigm type

Parameters

handle	handle of the paradigm

Returns

type (e.g. SCOREP_PARADIGM_MPI for MPI)

See also

SCOREP_PublicTypes.h

L.5.2.34 const char∗(∗ SCOREP_SubstratePluginCallbacks::SCOREP_ParameterHandle_GetName) (SCOREP_ParameterHandle handle)

Returns the name of a parameter

Parameters

handle	handle of the parameter

Returns

name

L.5.2.35 SCOREP_ParameterType(∗ SCOREP_SubstratePluginCallbacks::SCOREP_ParameterHandle_GetType) (SCOREP_ParameterHandle handle)

Returns parameter type

Parameters

handle	handle of the parameter

Returns

     type of parameter

See also

     SCOREP_PublicTypes.h

L.5.2.36 SCOREP_LineNo(∗ SCOREP_SubstratePluginCallbacks::SCOREP_RegionHandle_GetBeginLine) (SCOREP_RegionHandle handle)

Returns begin line of a function within the source file

Parameters

handle	handle of the region

Returns

     begin line

L.5.2.37 const char∗(∗ SCOREP_SubstratePluginCallbacks::SCOREP_RegionHandle_GetCanonicalName) (SCOREP_RegionHandle handle)

Returns regions mangled canonical name

Parameters

handle	of the region

Returns

     mangled region name

L.5.2.38 SCOREP_LineNo(∗ SCOREP_SubstratePluginCallbacks::SCOREP_RegionHandle_GetEndLine) (SCOREP_RegionHandle handle)

Returns end line of a function within the source file

Parameters

handle	handle of the region

Returns

     end line

L.5.2.39 const char∗(∗ SCOREP_SubstratePluginCallbacks::SCOREP_RegionHandle_GetFileName) (SCOREP_RegionHandle handle)

Returns file name where region is defined

Parameters

handle	handle of the region

Returns

file name

L.5.2.40 uint32_t(∗ SCOREP_SubstratePluginCallbacks::SCOREP_RegionHandle_GetId) (SCOREP_RegionHandle handle)

Returns parameter id

Parameters

handle	handle of the parameter

Returns

id of parameter

L.5.2.41 const char∗(∗ SCOREP_SubstratePluginCallbacks::SCOREP_RegionHandle_GetName) (SCOREP_RegionHandle handle)

Returns region demangled name

Parameters

handle	handle of the region

Returns

demangled region name

L.5.2.42 SCOREP_ParadigmType(∗ SCOREP_SubstratePluginCallbacks::SCOREP_RegionHandle_GetParadigmType) (SCOREP_RegionHandle handle)

Returns region paradigm

Parameters

handle	handle of the region

Returns

paradigm (e.g. SCOREP_PARADIGM_MPI for MPI regions)

See also

SCOREP_PublicTypes.h

L.5.2.43 SCOREP_RegionType(∗ SCOREP_SubstratePluginCallbacks::SCOREP_RegionHandle_GetType) (SCOREP_RegionHandle handle)

Returns region type

Parameters

handle	handle of the region

Returns

type (e.g. SCOREP_REGION_USER for user regions)

See also

SCOREP_PublicTypes.h

L.5.2.44 const SCOREP_MetricHandle∗(∗ **SCOREP_SubstratePluginCallbacks::SCOREP_SamplingSetHandle_Get**↩
MetricHandles) (SCOREP_SamplingSetHandle handle)

Get the metric handles in a sampling set.

Parameters

handle	the handle of the the existing sampling set

Returns

a list of associated metrics. get the length of the list with SCOREP_SamplingSet_GetNunmberOfMetrics

L.5.2.45 SCOREP_MetricOccurrence(∗ SCOREP_SubstratePluginCallbacks::SCOREP_SamplingSetHandle_GetMetric↩
Occurrence) (SCOREP_SamplingSetHandle handle)

Get the metric occurrence of a sampling set.

Parameters

handle	the handle of the the existing sampling set

Returns

the occurrence of handle.

L.5.2.46 uint8_t(∗ SCOREP_SubstratePluginCallbacks::SCOREP_SamplingSetHandle_GetNumberOfMetrics)
(SCOREP_SamplingSetHandle handle)

Get the number of metrics in a sampling set.

Parameters

handle	handle of the the existing sampling set

Returns

the number of associated metrics

L.5.2.47 SCOREP_SamplingSetClass(∗ SCOREP_SubstratePluginCallbacks::SCOREP_SamplingSetHandle_Get↩
SamplingSetClass) (SCOREP_SamplingSetHandle handle)

Returns the class of the sampling set

Parameters

handle	the handle of the the existing sampling set

Returns

sampling set class

L.5.2.48 SCOREP_MetricScope(∗ SCOREP_SubstratePluginCallbacks::SCOREP_SamplingSetHandle_GetScope)
(SCOREP_SamplingSetHandle handle)

Returns the scope of the sampling set or SCOREP_INVALID_METRIC_SCOPE if sampling set is not scoped

Parameters

handle	the handle of the the existing sampling set

Returns

scope or ($>$=SCOREP_INVALID_METRIC_SCOPE) if the sampling set is not scoped or the runtime version
of Score-P is newer than the Score-P version when compiling plugin

L.5.2.49 bool(∗ SCOREP_SubstratePluginCallbacks::SCOREP_SamplingSetHandle_IsScoped)
(SCOREP_SamplingSetHandle handle)

Check whether a sampling set is scoped (belongs to a number of locations).

Parameters

handle	the handle of the the existing sampling set

Returns

whether the sampling set is scoped or not

L.5.2.50 const char∗(∗ SCOREP_SubstratePluginCallbacks::SCOREP_SourceFileHandle_GetName)
(SCOREP_SourceFileHandle handle)

Returns name of a source file

Parameters

handle	handle of the source file

Returns

name

L.5.2.51   const char∗(∗ SCOREP_SubstratePluginCallbacks::SCOREP_StringHandle_Get) (SCOREP_StringHandle handle)

Resolve string handle

Parameters

handle	handle of string

Returns

string

The documentation for this struct was generated from the following file:

- SCOREP_SubstratePlugins.h

L.6   SCOREP_SubstratePluginInfo Struct Reference

```
#include <SCOREP_SubstratePlugins.h>
```

Data Fields

- void(∗ activate_cpu_location)(const struct SCOREP_Location ∗location, const struct SCOREP_Location ∗parentLocation, uint32_t forkSequenceCount)
- void(∗ assign_id)(size_t pluginId)
- void(∗ core_task_complete)(const struct SCOREP_Location ∗location, SCOREP_TaskHandle taskHandle)
- void(∗ core_task_create)(const struct SCOREP_Location ∗location, SCOREP_TaskHandle taskHandle)
- void(∗ create_location)(const struct SCOREP_Location ∗location, const struct SCOREP_Location ∗parent↩Location)
- void(∗ deactivate_cpu_location)(const struct SCOREP_Location ∗location, const struct SCOREP_Location ∗parentLocation)
- void(∗ delete_location)(const struct SCOREP_Location ∗location)
- void(∗ dump_manifest)(FILE ∗manifestFile, const char ∗relativeSourceDir, const char ∗targetDir)
- void(∗ finalize)(void)
- uint32_t(∗ get_event_functions)(SCOREP_Substrates_Mode mode, SCOREP_Substrates_Callback ∗∗functions)
- bool(∗ get_requirement)(SCOREP_Substrates_RequirementFlag flag)
- int(∗ init)(void)
- void(∗ init_mpp)(void)
- void(∗ new_definition_handle)(SCOREP_AnyHandle handle, SCOREP_HandleType type)
- uint32_t plugin_version
- void(∗ pre_unify)(void)
- void(∗ set_callbacks)(const SCOREP_SubstratePluginCallbacks ∗callbacks, size_t size)
- void(∗ undeclared [SCOREP_SUBSTRATE_PLUGIN_UNDEFINED_MANAGEMENT_FUNCTIONS])(void)
- void(∗ write_data)(void)

L.6.1   Detailed Description

Describes a Substrate plugin. The plugin definition should be done using the SCOREP_SUBSTRATE_PLUGIN_↩
ENTRY macro The call order of these functions is:

- Called per process

    1. (...Score-P initialization part 1, e.g. environment variables...)
    2. resolving of SCOREP_SUBSTRATE_PLUGIN_ENTRY, e.g., for SCOREP_SUBSTRATE_PLUGIN↩
       S=foo check whether there is a library called libscorep_substrate_foo.so that provides a SCOREP_S↩
       UBSTRATE_PLUGIN_ENTRY(foo) / holds the function SCOREP_SubstratePlugin_foo_get_info
    3. init()
    4. set_callbacks()
    5. get_event_functions( ) with mode == SCOREP_SUBSTRATES_RECORDING_ENABLED
    6. get_event_functions( ) with mode == SCOREP_SUBSTRATES_RECORDING_DISABLED
    7. From now on at any point in time there can be new definitions created via new_definition_handle().
       Definitions are unique in a process context. Thus, two different locations within a process can use the
       same definitions and there are no handles that are only valid for a specific location. Plugins should care
       for thread safeness when registering handles in internal data structures.
    8. From now on at any point in time there can be calls to get_requirement( )
    9. (...Score-P initialization part 2...)
    10. assign_id()
    11. create_location() for the main thread
    12. As soon as the MPP paradigm is initialized, init_mpp() is called. If the program does not use MPP
        paradigms, init_mpp() will also be called
    13. (...Score-P initialization part 3...)
    14. Called for each location (e.g., thread)
        (a) create_location() (Only new locations, remember that the main thread location is already created
            earlier!)
        (b) if it is a CPU location: activate_cpu_location()
        (c) Now there is a sequence of these events
            – if it is a CPU location:
                * (optional) calls to deactivate_cpu_location() followed by calls to activate_cpu_location()
                * (optional) core_task_create(), possibly followed by runtime events, followed by core_task↩
                  _complete()
            – (optional) runtime events
        (d) if it is a CPU location: deactivate_cpu_location()
        (e) delete_location()
    15. pre_unify()
    16. write_data()
    17. finalize()

Not implemented functions MUST point to NULL, e.g., info.assign_id = NULL;

Developer notice: When a new function is necessary, append it after the functions, but before undeclared. For each
new function, decrease SCOREP_SUBSTRATE_PLUGIN_UNDEFINED_MANAGEMENT_FUNCTIONS by one. If
this happens, increase SCOREP_PLUGIN_VERSION.

L.6.2 Field Documentation

L.6.2.1 void(∗ SCOREP_SubstratePluginInfo::activate_cpu_location) (const struct SCOREP_Location ∗location, const struct SCOREP_Location ∗parentLocation, uint32_t forkSequenceCount)

This function is called whenever a location is activated.

See also

> [create_location](#)
> [deactivate_cpu_location](#)
> [delete_location](#)

Parameters

location	location which is activated
parentLocation	parent of location. May be equal *locationData*.
forkSequence↩ Count	an increasing number which defines the current fork from parentLocation. At each fork (or omp parallel) this increases by 1.

L.6.2.2 void(∗ SCOREP_SubstratePluginInfo::assign_id) (size_t pluginId)

This function assigns a specific ID to the plugin that can be used for accessing thread local storage. However, most of the internal functionality is not available at the time this function is called. Therefore, only the id should be stored for now.

Parameters

plugin_id	a specific ID that is assigned to the plugin and can be used to access thread local storages. (see also SCOREP_Location_SetData and SCOREP_Location_GetData) This ID is only valid for the current Score-P run.

L.6.2.3 void(∗ SCOREP_SubstratePluginInfo::core_task_complete) (const struct SCOREP_Location ∗location, SCOREP_TaskHandle taskHandle)

Called when a task (e.g., an OpenMP task) is completed The taskHandle is not defined via define_handle and can not be converted to SCOREP_AnyHandle

Parameters

location	the location that completes the task
taskHandle	the completed created task

L.6.2.4 void(∗ SCOREP_SubstratePluginInfo::core_task_create) (const struct SCOREP_Location ∗location, SCOREP_TaskHandle taskHandle)

Called when a task (e.g., an OpenMP task) is creates The taskHandle is not defined via define_handle and can not be converted to SCOREP_AnyHandle

Parameters

location	the location that created the task
taskHandle	the newly created task

L.6.2.5   void(∗ SCOREP_SubstratePluginInfo::create_location) (const struct SCOREP_Location ∗location, const struct SCOREP_Location ∗parentLocation)

The location callbacks notify the subsystem about the lifetime of a location. For CPU locations:

- create_location
    - initial activate_cpu_location()
    - (optional) interim deactivate_cpu_location(), followed by activate_cpu_location()
    - final deactivate_cpu_location()
- delete_location

For none-CPU locations:

1. create_location

2. delete_location This function is called whenever a new location is created, e.g. whenever a new OpenMP thread is created.

Parameters

location	location which is created
parentLocation	location that created this location

L.6.2.6   void(∗ SCOREP_SubstratePluginInfo::deactivate_cpu_location) (const struct SCOREP_Location ∗location, const struct SCOREP_Location ∗parentLocation)

This function is called whenever a location is deactivated

See also

> create_location
> activate_cpu_location

Parameters

location	location which is deactivated
parentLocation	parent of location. May be equal *locationData*.

L.6.2.7   void(∗ SCOREP_SubstratePluginInfo::delete_location) (const struct SCOREP_Location ∗location)

This function is called whenever a location is deleted

See also

> create_location

location	location which is deleted

L.6.2.8 void(∗ SCOREP_SubstratePluginInfo::dump_manifest) (FILE ∗manifestFile, const char ∗relativeSourceDir, const char ∗targetDir)

This function is called during experiment directory creation and allows to write information about involved files to the Manifest file and initiate file copies if needed.

L.6.2.9 void(∗ SCOREP_SubstratePluginInfo::finalize) (void)

This function is called when the Score-P run finished

L.6.2.10 uint32_t(∗ SCOREP_SubstratePluginInfo::get_event_functions) (SCOREP_Substrates_Mode mode, SCOREP_Substrates_Callback ∗∗functions)

Get all functions for events, attributed to their SCOREP_Substrates_EventType

Parameters

mode	defines which function set should be returned either for disabled or enabled recording.
functions	a pointer to the functions that are assigned to the types. The returned array MUST hold SC↩OREP_SUBSTRATES_NUM_EVENTS elements. Not-implemented functions should be set to NULL. The array will NOT be free'd by Score-P.

Returns

MUST return SCOREP_SUBSTRATES_NUM_EVENTS (see SCOREP_SubstrateEvents.h)

L.6.2.11 bool(∗ SCOREP_SubstratePluginInfo::get_requirement) (SCOREP_Substrates_RequirementFlag flag)

Provide Score-P with additional information about requirements, see SCOREP_SubstratesRequirementFlag for details. If this function is not implemented, the default is assumed (0). This can be called at any time by any thread depending on the flag that is queried. Plugins must take care that they return 0 if flag is greater than SCORE↩P_SUBSTRATES_NUM_REQUIREMENT Plugins must always return the same value for a given flag during one execution.

Parameters

flag	the requirement flag that is queried

Returns

the setting for the requirement flag, which highly depends on the type of flag

L.6.2.12   int(∗ SCOREP_SubstratePluginInfo::init) (void)

This function is called before most internal Score-P data is initialized. The plugin should be initialized here and dependencies should be checked.

Returns

> 0 if initialization succeeded, otherwise !=0

L.6.2.13   void(∗ SCOREP_SubstratePluginInfo::init_mpp) (void)

This function is called after MPP paradigms are initialized. If the program does not use MPP paradigms this function is also called. To detect used paradigms check for calls to new_definition_handle with type == SCOREP_HAND↩
LE_TYPE_PARADIGM.

L.6.2.14   void(∗ SCOREP_SubstratePluginInfo::new_definition_handle) (SCOREP_AnyHandle handle,
SCOREP_HandleType type)

This function will be called whenever a new definition is created Plugins can filter the processing of definitions according to the given type. Plugins should use the callbacks passed by set_callbacks, SCOREP_Public↩
Handles.h, and SCOREP_PublicTypes.h to make sense from the handle, (e.g., to get the name of a region) The handles can be unique for every process and are not necessarily comparable. It might be that this is called within a locked region. Therefore your implementation should be as lightweight as possible.

Parameters

handle	a handle to the newly created object
type	the type of the handle

L.6.2.15   uint32_t SCOREP_SubstratePluginInfo::plugin_version

Must be set to SCOREP_SUBSTRATE_PLUGIN_VERSION (needed for back- and forward compatibility)

L.6.2.16   void(∗ SCOREP_SubstratePluginInfo::pre_unify) (void)

This function is called before the data about different threads and MPI processes is collected and unified, i.e. when definitions are synchronized.

L.6.2.17   void(∗ SCOREP_SubstratePluginInfo::set_callbacks) (const SCOREP_SubstratePluginCallbacks ∗callbacks,
size_t size)

Provide plugins with pointers to functions that can be used to get meta data about handles.

Parameters

callbacks	the provided function callbacks
sizeof(SCORE↩ P_Substrate↩ Callbacks)	The plugin should care that its version of SCOREP_SubstrateCallbacks is smaller or equal size

L.6.2.18   void(∗ SCOREP_SubstratePluginInfo::undeclared[SCOREP_SUBSTRATE_PLUGIN_UNDEFINED_MANA↩ GEMENT_FUNCTIONS])(void)

for future extensions Plugins must set all entries of this list to 0 (e.g., via memset)

When a new function is added in Score-P, SCOREP_SUBSTRATE_PLUGIN_UNDEFINED_MANAGEMENT_F↩ UNCTIONS should be decreased by 1. Score-P should check for an appropriate plugin_version before calling the new function. Otherwise, Score-P is not able to check whether this is set to 0 but might use an invalid function. This can be avoided by enforcing a correct plugin_version.

L.6.2.19   void(∗ SCOREP_SubstratePluginInfo::write_data) (void)

This function is called after the unification process when traces/profiles are written - right before finalize. This should be used to flush recorded data.

The documentation for this struct was generated from the following file:

- SCOREP_SubstratePlugins.h

Appendix M

File Documentation

M.1 SCOREP_MetricPlugins.h File Reference

Description of the metric plugin header. For information on how to use metric plugins, please refer to 'Metric Plugins'.

```
#include <stdbool.h>
#include <scorep/SCOREP_MetricTypes.h>
```

Data Structures

- struct SCOREP_Metric_Plugin_Info
- struct SCOREP_Metric_Plugin_MetricProperties

  Properties describing a metric. Provided by the get_event_info function.

Macros

- #define SCOREP_METRIC_PLUGIN_ENTRY(_name)
- #define SCOREP_METRIC_PLUGIN_VERSION 1

M.1.1 Detailed Description

Description of the metric plugin header. For information on how to use metric plugins, please refer to 'Metric Plugins'.

M.1.2 Macro Definition Documentation

M.1.2.1 #define SCOREP_METRIC_PLUGIN_ENTRY(_name)

Value:

```
EXTERN SCOREP_Metric_Plugin_Info \
    SCOREP_MetricPlugin_ ## _name ## _get_info( void )
```

Macro used for implementation of the 'get_info' function

M.1.2.2   #define SCOREP_METRIC_PLUGIN_VERSION 1

The developer of a metric plugin should provide a README file which explains how to compile, install and use the plugin. In particular, the supported metrics should be described in the README file.

Each metric plugin has to include `SCOREP_MetricPlugins.h` and implement a 'get_info' function. Therefore, use the SCOREP_METRIC_PLUGIN_ENTRY macro and provide the name of the plugin library as the argument. For example, the metric plugin libexample_plugin.so should use SCOREP_METRIC_PLUGIN_ENTRY(example←
_plugin).

It is encouraged to use the "_plugin" suffix on the name to avoid conflicts with existing libraries, e.g., libsensors_←
plugin.so using the existing libsensors.so.

M.1.3   Mandatory functions

See each function for details.

initialize

Check requirements and initialize the plugin.

get_event_info

A user specifies a SCOREP_METRIC_EXAMPLE_PLUGIN=token1,token2,... This function provides information about the metric(s) corresponding to this token. The total list of metrics returned for all tokens will then be recorded by the plugin.

add_counter

The function is called for and sets each of the metrics to be recorded by the plugin. It provides a unique ID for each metric.

finalize

Clean up the resources used by the metric plugin.

M.1.4   Mandatory variables

run_per

Defines how many threads should record the metrics of a plugin.

sync

Defines synchronicity type of a metric plugin. A metric plugin can

- provide a metric value for each event (SCOREP_METRIC_STRICTLY_SYNC)

- optionally provide a metric value for each Score-P event (SCOREP_METRIC_SYNC)

- measure metric values independently of Score-P events, but collect them in Score-p during a Score-P event (SCOREP_METRIC_ASYNC_EVENT)

- measure all metric values independently of events and collect them once at the very end of execution (SC←
OREP_METRIC_ASYNC)

plugin_version

Should be set to SCOREP_METRIC_PLUGIN_VERSION

Depending on the plugin's synchronicity type there are some optional functions and variables.

M.1.5   Optional functions

get_current_value

Used by strictly synchronous metric plugins only. Returns value of requested metric.

get_optional_value

Used by synchronous metric plugins, but not by strictly synchronous ones. This function requests current value of a metric, but it is valid that no value is returned (read: no update for this metric available).

get_all_values

Used by asynchronous metric plugins. This function is used to request values of a asynchronous metric. The metric will return an arbitrary number of timestamp-value-pairs.

set_clock_function

Used by asynchronous metric plugins. This function passes a function to the plugin, which can be used by the plugin to get a Score-P valid timestamp.

M.1.6   Optional variables

delta_t

Defines interval between two calls to update metric value. Ignored for strictly synchronous plugins.Current version of Score-P metric plugin interface

M.2   SCOREP_MetricTypes.h File Reference

Types used by metric service.

```
#include <stdint.h>
```

Data Structures

- struct SCOREP_Metric_Properties
- struct SCOREP_MetricTimeValuePair

Enumerations

- enum SCOREP_MetricBase {
  SCOREP_METRIC_BASE_BINARY = 0,
  SCOREP_METRIC_BASE_DECIMAL = 1,
  SCOREP_INVALID_METRIC_BASE }

- enum SCOREP_MetricMode {
  SCOREP_METRIC_MODE_ACCUMULATED_START = 0,
  SCOREP_METRIC_MODE_ACCUMULATED_POINT = 1,
  SCOREP_METRIC_MODE_ACCUMULATED_LAST = 2,
  SCOREP_METRIC_MODE_ACCUMULATED_NEXT = 3,
  SCOREP_METRIC_MODE_ABSOLUTE_POINT = 4,
  SCOREP_METRIC_MODE_ABSOLUTE_LAST = 5,
  SCOREP_METRIC_MODE_ABSOLUTE_NEXT = 6,
  SCOREP_METRIC_MODE_RELATIVE_POINT = 7,
  SCOREP_METRIC_MODE_RELATIVE_LAST = 8,
  SCOREP_METRIC_MODE_RELATIVE_NEXT = 9 }

- enum SCOREP_MetricPer {
  SCOREP_METRIC_PER_THREAD = 0,
  SCOREP_METRIC_PER_PROCESS,
  SCOREP_METRIC_PER_HOST,
  SCOREP_METRIC_ONCE }

- enum SCOREP_MetricProfilingType {
  SCOREP_METRIC_PROFILING_TYPE_EXCLUSIVE = 0,
  SCOREP_METRIC_PROFILING_TYPE_INCLUSIVE = 1,
  SCOREP_METRIC_PROFILING_TYPE_SIMPLE = 2,
  SCOREP_METRIC_PROFILING_TYPE_MAX = 3,
  SCOREP_METRIC_PROFILING_TYPE_MIN = 4 }

- enum SCOREP_MetricSourceType {
  SCOREP_METRIC_SOURCE_TYPE_PAPI = 0,
  SCOREP_METRIC_SOURCE_TYPE_RUSAGE = 1,
  SCOREP_METRIC_SOURCE_TYPE_USER = 2,
  SCOREP_METRIC_SOURCE_TYPE_OTHER = 3,
  SCOREP_METRIC_SOURCE_TYPE_TASK = 4,
  SCOREP_METRIC_SOURCE_TYPE_PLUGIN = 5,
  SCOREP_METRIC_SOURCE_TYPE_PERF = 6 }

- enum SCOREP_MetricSynchronicity {
  SCOREP_METRIC_STRICTLY_SYNC = 0,
  SCOREP_METRIC_SYNC,
  SCOREP_METRIC_ASYNC_EVENT,
  SCOREP_METRIC_ASYNC }

- enum SCOREP_MetricSynchronizationMode {
  SCOREP_METRIC_SYNCHRONIZATION_MODE_BEGIN,
  SCOREP_METRIC_SYNCHRONIZATION_MODE_BEGIN_MPP,
  SCOREP_METRIC_SYNCHRONIZATION_MODE_END }

- enum SCOREP_MetricValueType {
  SCOREP_METRIC_VALUE_INT64,
  SCOREP_METRIC_VALUE_UINT64,
  SCOREP_METRIC_VALUE_DOUBLE }

M.2.1 Detailed Description

Types used by metric service.

M.2.2 Enumeration Type Documentation

M.2.2.1 enum SCOREP_MetricBase

Types to be used in defining metric base (SCOREP_Definitions_NewMetric()).

Enumerator

> ***SCOREP_METRIC_BASE_BINARY*** Binary base.
>
> ***SCOREP_METRIC_BASE_DECIMAL*** Decimal base.
>
> ***SCOREP_INVALID_METRIC_BASE*** For

M.2.2.2 enum SCOREP_MetricMode

Types to be used in defining metric mode (SCOREP_Definitions_NewMetric()). The mode consists of a timing and a value semantic. The possible value semantics are:

- Accumulated for classic counters, e.g. number of floating point operations. While they are stored monotonically increasing in the trace, they are often differentiated as rate over time.

- Absolute values, e.g. temperature. They are stored as is in the trace and typically also displayed as is.

- Relative values.

The possible timing semantics are:

- Start: The value is valid for the interval from the beginning of the trace to the associated timestamp.

- Point: The value is only valid for the point in time given by the timestamp.

- Last: The value is valid for the interval from the previous to the current timestamp.

- Next: The value is valid for the interval from the current to the next timestamp.

Enumerator

> ***SCOREP_METRIC_MODE_ACCUMULATED_START*** Accumulated metric, 'START' timing.
>
> ***SCOREP_METRIC_MODE_ACCUMULATED_POINT*** Accumulated metric, 'POINT' timing.
>
> ***SCOREP_METRIC_MODE_ACCUMULATED_LAST*** Accumulated metric, 'LAST' timing.
>
> ***SCOREP_METRIC_MODE_ACCUMULATED_NEXT*** Accumulated metric, 'NEXT' timing.
>
> ***SCOREP_METRIC_MODE_ABSOLUTE_POINT*** Absolute metric, 'POINT' timing.
>
> ***SCOREP_METRIC_MODE_ABSOLUTE_LAST*** Absolute metric, 'LAST' timing.
>
> ***SCOREP_METRIC_MODE_ABSOLUTE_NEXT*** Absolute metric, 'NEXT' timing.
>
> ***SCOREP_METRIC_MODE_RELATIVE_POINT*** Relative metric, 'POINT' timing.
>
> ***SCOREP_METRIC_MODE_RELATIVE_LAST*** Relative metric, 'LAST' timing.
>
> ***SCOREP_METRIC_MODE_RELATIVE_NEXT*** Relative metric, 'NEXT' timing.

M.2.2.3 enum SCOREP_MetricPer

Enumeration to define how many threads should record the metrics of a plugin. Used by SCOREP_Metric_Plugin↩_Info::run_per.

Enumerator

> ***SCOREP_METRIC_PER_THREAD*** Metric values are recorded on all threads of all processes
>
> ***SCOREP_METRIC_PER_PROCESS*** If processes use multiple threads, the metric is recorded on the main thread of each process.
>
> ***SCOREP_METRIC_PER_HOST*** Metric values are recorded on a single thread of each node in a parallel program running on multiple nodes (hosts). Nodes are determined by the platform-specific Score-P node identifier.
>
> ***SCOREP_METRIC_ONCE*** Metric values recorded once within the parallel program. They are recorded on the first node, first process, first thread.

M.2.2.4 enum SCOREP_MetricProfilingType

Types used to define type of profiling.

Enumerator

> ***SCOREP_METRIC_PROFILING_TYPE_EXCLUSIVE*** Exclusive values (excludes values from subordinated items)
>
> ***SCOREP_METRIC_PROFILING_TYPE_INCLUSIVE*** Inclusive values (sum including values from subordinated items)
>
> ***SCOREP_METRIC_PROFILING_TYPE_SIMPLE*** Single value
>
> ***SCOREP_METRIC_PROFILING_TYPE_MAX*** Maximum values
>
> ***SCOREP_METRIC_PROFILING_TYPE_MIN*** Minimum values

M.2.2.5 enum SCOREP_MetricSourceType

Metric sources to be used in defining a metric member (SCOREP_Definitions_NewMetric()).

Enumerator

> ***SCOREP_METRIC_SOURCE_TYPE_PAPI*** PAPI counter.
>
> ***SCOREP_METRIC_SOURCE_TYPE_RUSAGE*** Resource usage counter.
>
> ***SCOREP_METRIC_SOURCE_TYPE_USER*** User metrics.
>
> ***SCOREP_METRIC_SOURCE_TYPE_OTHER*** Any other metrics.
>
> ***SCOREP_METRIC_SOURCE_TYPE_TASK*** Generated by task profiling.
>
> ***SCOREP_METRIC_SOURCE_TYPE_PLUGIN*** Plugin metrics.
>
> ***SCOREP_METRIC_SOURCE_TYPE_PERF*** Linux perf metrics

M.2.2.6 enum SCOREP_MetricSynchronicity

Enumeration to define the synchronicity type of a metric. Used by SCOREP_Metric_Plugin_Info::sync.

Enumerator

> ***SCOREP_METRIC_STRICTLY_SYNC*** The current value of each metric is queried by Score-P whenever an enter/leave event occurs via get_current_value. The plugin must always be able to provide a current value. The plugin provides the value itself, the timestamp is provided by Score-P. This setting is used for metrics that can be measured with minimal runtime costs and updated very frequently.

> ***SCOREP_METRIC_SYNC*** The current value of each metric is queried by Score-P whenever an enter/leave event occurs via get_optional_value. Providing a value is optional in case no new value is available in the plugin. The plugin provides the value itself, the timestamp is provided by Score-P. This setting is used for metrics that can be measured with minimal runtime costs but do not necessarily always change.

> ***SCOREP_METRIC_ASYNC_EVENT*** Metric values are be measured at arbitrary points in time, but are collected at enter/leave events. Whenever an enter/leave event occurs, Score-P queries the plugin via get↩ _all_values for a list of timestamp-value-pairs. This setting can be used for some special cases, SCOR↩ EP_METRIC_ASYNC is usually easier to implement.

> ***SCOREP_METRIC_ASYNC*** Metric values are be measured at arbitrary points in time. All values are collected once at the very end of the execution. Score-P collects the values and associated timestamps via get↩ _all_values. This setting is used for metrics that are recorded on external systems or within a separate thread. While it does require additional memory buffers to store the measurement, it usually reduces the overhead by decoupling the measurement from collection. It is also called post-mortem processing.

M.2.2.7 enum SCOREP_MetricSynchronizationMode

Possible modes of a synchronization point. Express the time when a synchronization happens.

Enumerator

> ***SCOREP_METRIC_SYNCHRONIZATION_MODE_BEGIN*** Synchronization at the beginning of the measurement

> ***SCOREP_METRIC_SYNCHRONIZATION_MODE_BEGIN_MPP*** Synchronization at the initialization of a multi-process paradigm (e.g., MPI)

> ***SCOREP_METRIC_SYNCHRONIZATION_MODE_END*** Synchronization at the end of the measurement

M.2.2.8 enum SCOREP_MetricValueType

Types to be used in defining type of metric values (SCOREP_Definitions_NewMetric()). The interface uses UINT64 for all values, the other types should be reinterpreted using a union.

Enumerator

> ***SCOREP_METRIC_VALUE_INT64*** 64 bit integer

> ***SCOREP_METRIC_VALUE_UINT64*** 64 bit unsigned integer

> ***SCOREP_METRIC_VALUE_DOUBLE*** double precision floating point

M.3 SCOREP_PublicHandles.h File Reference

Description of definition handles. This header defines an enumeration to map SCOREP_AnyHandle to specific handles. It also contains some of these handle definitions.

```
#include <scorep/SCOREP_PublicTypes.h>
```

Enumerations

- enum SCOREP_HandleType { ,
  SCOREP_HANDLE_TYPE_CALLING_CONTEXT,
  SCOREP_HANDLE_TYPE_GROUP,
  SCOREP_HANDLE_TYPE_INTERIM_COMMUNICATOR,
  SCOREP_HANDLE_TYPE_INTERRUPT_GENERATOR,
  SCOREP_HANDLE_TYPE_LOCATION,
  SCOREP_HANDLE_TYPE_LOCATION_GROUP,
  SCOREP_HANDLE_TYPE_LOCATION_PROPERTY,
  SCOREP_HANDLE_TYPE_METRIC,
  SCOREP_HANDLE_TYPE_PARADIGM,
  SCOREP_HANDLE_TYPE_PARAMETER,
  SCOREP_HANDLE_TYPE_REGION,
  SCOREP_HANDLE_TYPE_RMA_WINDOW,
  SCOREP_HANDLE_TYPE_SAMPLING_SET,
  SCOREP_HANDLE_TYPE_SAMPLING_SET_RECORDER,
  SCOREP_HANDLE_TYPE_SOURCE_CODE_LOCATION,
  SCOREP_HANDLE_TYPE_SOURCE_FILE,
  SCOREP_HANDLE_TYPE_STRING,
  SCOREP_HANDLE_TYPE_SYSTEM_TREE_NODE,
  SCOREP_HANDLE_TYPE_SYSTEM_TREE_NODE_PROPERTY,
  SCOREP_HANDLE_TYPE_CARTESIAN_TOPOLOGY,
  SCOREP_HANDLE_TYPE_CARTESIAN_COORDS,
  SCOREP_HANDLE_TYPE_IO_FILE,
  SCOREP_HANDLE_TYPE_IO_FILE_PROPERTY,
  SCOREP_HANDLE_TYPE_IO_HANDLE,
  SCOREP_HANDLE_TYPE_IO_PARADIGM,
  SCOREP_HANDLE_TYPE_NUM_HANDLES }

M.3.1 Detailed Description

Description of definition handles. This header defines an enumeration to map SCOREP_AnyHandle to specific handles. It also contains some of these handle definitions.

M.3.2 Enumeration Type Documentation

M.3.2.1 enum **SCOREP_HandleType**

handle types, lists all handle types that are supported.

Enumerator

SCOREP_HANDLE_TYPE_CALLING_CONTEXT The handle type is not defined/invalid

SCOREP_HANDLE_TYPE_GROUP The handle type is SCOREP_CallingContextHandle

SCOREP_HANDLE_TYPE_INTERIM_COMMUNICATOR The handle type is SCOREP_GroupHandle

SCOREP_HANDLE_TYPE_INTERRUPT_GENERATOR The handle type is SCOREP_InterimCommunicator↩
Handle

SCOREP_HANDLE_TYPE_LOCATION The handle type is SCOREP_InterruptGeneratorHandle

SCOREP_HANDLE_TYPE_LOCATION_GROUP The handle type is SCOREP_LocationHandle

SCOREP_HANDLE_TYPE_LOCATION_PROPERTY The handle type is SCOREP_LocationGroupHandle

SCOREP_HANDLE_TYPE_METRIC The handle type is SCOREP_LocationPropertyHandle

SCOREP_HANDLE_TYPE_PARADIGM The handle type is SCOREP_MetricHandle (defined in SCOREP↩
_PublicTypes.h)

SCOREP_HANDLE_TYPE_PARAMETER The handle type is SCOREP_ParadigmHandle (defined in SCO↩
REP_PublicTypes.h)

SCOREP_HANDLE_TYPE_REGION The handle type is SCOREP_ParameterHandle

SCOREP_HANDLE_TYPE_RMA_WINDOW The handle type is SCOREP_RegionHandle (defined in SCO↩
REP_PublicTypes.h)

SCOREP_HANDLE_TYPE_SAMPLING_SET The handle type is SCOREP_RmaWindowHandle

SCOREP_HANDLE_TYPE_SAMPLING_SET_RECORDER The handle type is SCOREP_SamplingSet↩
Handle (defined in SCOREP_PublicTypes.h)

SCOREP_HANDLE_TYPE_SOURCE_CODE_LOCATION The handle type is SCOREP_SamplingSet↩
RecorderHandle

SCOREP_HANDLE_TYPE_SOURCE_FILE The handle type is SCOREP_SourceCodeLocationHandle

SCOREP_HANDLE_TYPE_STRING The handle type is SCOREP_SourceFileHandle (defined in SCORE↩
P_PublicTypes.h)

SCOREP_HANDLE_TYPE_SYSTEM_TREE_NODE The handle type is SCOREP_StringHandle

SCOREP_HANDLE_TYPE_SYSTEM_TREE_NODE_PROPERTY The handle type is SCOREP_System↩
TreeNodeHandle

SCOREP_HANDLE_TYPE_CARTESIAN_TOPOLOGY The handle type is SCOREP_SystemTreeNode↩
PropertyHandle

SCOREP_HANDLE_TYPE_CARTESIAN_COORDS The handle type is SCOREP_CartesianTopology↩
Handle

SCOREP_HANDLE_TYPE_IO_FILE The handle type is SCOREP_CartesianCoordsHandle

SCOREP_HANDLE_TYPE_IO_FILE_PROPERTY The handle type is SCOREP_IoFileHandle

SCOREP_HANDLE_TYPE_IO_HANDLE The handle type is SCOREP_IoFilePropertyHandle

SCOREP_HANDLE_TYPE_IO_PARADIGM The handle type is SCOREP_IoHandleHandle

SCOREP_HANDLE_TYPE_NUM_HANDLES The handle type is SCOREP_AnyHandle Not ABI

M.4 SCOREP_PublicTypes.h File Reference

Defines public definitions that are used internally and externally (e.g., by metric plugins, user functions, substrate plugins)

```
#include <stdint.h>
```

Macros

- #define SCOREP_ALL_TARGET_RANKS -1
- #define SCOREP_INVALID_EXIT_STATUS ((int64_t)(∼((∼((uint64_t)0u)) >> 1)))
- #define SCOREP_INVALID_LINE_NO 0
- #define SCOREP_INVALID_METRIC SCOREP_MOVABLE_NULL
- #define SCOREP_INVALID_PARADIGM SCOREP_MOVABLE_NULL
- #define SCOREP_INVALID_PID 0
- #define SCOREP_INVALID_REGION SCOREP_MOVABLE_NULL
- #define SCOREP_INVALID_ROOT_RANK -1
- #define SCOREP_INVALID_SAMPLING_SET SCOREP_MOVABLE_NULL
- #define SCOREP_INVALID_SOURCE_FILE SCOREP_MOVABLE_NULL
- #define SCOREP_INVALID_TID 0
- #define SCOREP_IO_UNKNOWN_OFFSET UINT64_MAX
- #define SCOREP_LOCATION_TYPES
- #define SCOREP_MOVABLE_NULL 0
- #define SCOREP_MPI_PROC_NULL -3
- #define SCOREP_MPI_ROOT -2

Typedefs

- typedef uint32_t SCOREP_Allocator_MovableMemory
- typedef SCOREP_Allocator_MovableMemory SCOREP_AnyHandle
- typedef int64_t SCOREP_ExitStatus
- typedef uint32_t SCOREP_LineNo
- typedef SCOREP_AnyHandle SCOREP_MetricHandle
- typedef int SCOREP_MpiRank
- typedef uint64_t SCOREP_MpiRequestId
- typedef SCOREP_AnyHandle SCOREP_ParadigmHandle
- typedef SCOREP_AnyHandle SCOREP_RegionHandle
- typedef SCOREP_AnyHandle SCOREP_SamplingSetHandle
- typedef SCOREP_AnyHandle SCOREP_SourceFileHandle
- typedef struct SCOREP_Task ∗ SCOREP_TaskHandle

Enumerations

- enum SCOREP_CollectiveType {
  SCOREP_COLLECTIVE_BARRIER,
  SCOREP_COLLECTIVE_BROADCAST,
  SCOREP_COLLECTIVE_GATHER,
  SCOREP_COLLECTIVE_GATHERV,
  SCOREP_COLLECTIVE_SCATTER,
  SCOREP_COLLECTIVE_SCATTERV,
  SCOREP_COLLECTIVE_ALLGATHER,
  SCOREP_COLLECTIVE_ALLGATHERV,
  SCOREP_COLLECTIVE_ALLTOALL,
  SCOREP_COLLECTIVE_ALLTOALLV,
  SCOREP_COLLECTIVE_ALLTOALLW,
  SCOREP_COLLECTIVE_ALLREDUCE,
  SCOREP_COLLECTIVE_REDUCE,
  SCOREP_COLLECTIVE_REDUCE_SCATTER,
  SCOREP_COLLECTIVE_REDUCE_SCATTER_BLOCK,
  SCOREP_COLLECTIVE_SCAN,
  SCOREP_COLLECTIVE_EXSCAN,
  SCOREP_COLLECTIVE_CREATE_HANDLE,
  SCOREP_COLLECTIVE_DESTROY_HANDLE,
  SCOREP_COLLECTIVE_ALLOCATE,
  SCOREP_COLLECTIVE_DEALLOCATE,
  SCOREP_COLLECTIVE_CREATE_HANDLE_AND_ALLOCATE,

SCOREP_COLLECTIVE_DESTROY_HANDLE_AND_DEALLOCATE }

Types to specify the used collectives in calls to SCOREP_MpiCollectiveBegin and SCOREP_RmaCollectiveBegin.

- enum SCOREP_CommunicatorFlag {
SCOREP_COMMUNICATOR_FLAG_NONE = 0,
SCOREP_COMMUNICATOR_FLAG_CREATE_DESTROY_EVENTS = (1 << 0) }

- enum SCOREP_IoAccessMode {
SCOREP_IO_ACCESS_MODE_NONE = 0,
SCOREP_IO_ACCESS_MODE_READ_ONLY,
SCOREP_IO_ACCESS_MODE_WRITE_ONLY,
SCOREP_IO_ACCESS_MODE_READ_WRITE,
SCOREP_IO_ACCESS_MODE_EXECUTE_ONLY,
SCOREP_IO_ACCESS_MODE_SEARCH_ONLY }

- enum SCOREP_IoCreationFlag {
SCOREP_IO_CREATION_FLAG_NONE = 0,
SCOREP_IO_CREATION_FLAG_CREATE = (1 << 0),
SCOREP_IO_CREATION_FLAG_TRUNCATE = (1 << 1),
SCOREP_IO_CREATION_FLAG_DIRECTORY = (1 << 2),
SCOREP_IO_CREATION_FLAG_EXCLUSIVE = (1 << 3),
SCOREP_IO_CREATION_FLAG_NO_CONTROLLING_TERMINAL = (1 << 4),
SCOREP_IO_CREATION_FLAG_NO_FOLLOW = (1 << 5),
SCOREP_IO_CREATION_FLAG_PATH = (1 << 6),
SCOREP_IO_CREATION_FLAG_TEMPORARY_FILE = (1 << 7),
SCOREP_IO_CREATION_FLAG_LARGEFILE = (1 << 8),
SCOREP_IO_CREATION_FLAG_NO_SEEK = (1 << 9),
SCOREP_IO_CREATION_FLAG_UNIQUE = (1 << 10) }

- enum SCOREP_IoOperationFlag {
SCOREP_IO_OPERATION_FLAG_NONE = 0,
SCOREP_IO_OPERATION_FLAG_BLOCKING = 0,
SCOREP_IO_OPERATION_FLAG_NON_BLOCKING = (1 << 0),
SCOREP_IO_OPERATION_FLAG_COLLECTIVE = (1 << 1),
SCOREP_IO_OPERATION_FLAG_NON_COLLECTIVE = 0 }

- enum SCOREP_IoOperationMode {
SCOREP_IO_OPERATION_MODE_READ = 0,
SCOREP_IO_OPERATION_MODE_WRITE,
SCOREP_IO_OPERATION_MODE_FLUSH }

- enum SCOREP_IoParadigmType

- enum SCOREP_IoSeekOption {
SCOREP_IO_SEEK_FROM_START = 0,
SCOREP_IO_SEEK_FROM_CURRENT,
SCOREP_IO_SEEK_FROM_END,
SCOREP_IO_SEEK_DATA,
SCOREP_IO_SEEK_HOLE,
SCOREP_IO_SEEK_INVALID }

- enum SCOREP_IoStatusFlag {
SCOREP_IO_STATUS_FLAG_NONE = 0,
SCOREP_IO_STATUS_FLAG_CLOSE_ON_EXEC = (1 << 0),
SCOREP_IO_STATUS_FLAG_APPEND = (1 << 1),
SCOREP_IO_STATUS_FLAG_NON_BLOCKING = (1 << 2),
SCOREP_IO_STATUS_FLAG_ASYNC = (1 << 3),
SCOREP_IO_STATUS_FLAG_SYNC = (1 << 4),
SCOREP_IO_STATUS_FLAG_DATA_SYNC = (1 << 5),
SCOREP_IO_STATUS_FLAG_AVOID_CACHING = (1 << 6),
SCOREP_IO_STATUS_FLAG_NO_ACCESS_TIME = (1 << 7),
SCOREP_IO_STATUS_FLAG_DELETE_ON_CLOSE = (1 << 8) }

- enum SCOREP_Ipc_Datatype

specifies an inter process communication data types

- enum SCOREP_Ipc_Operation

specifies an inter process communication operation for reduce function

- enum SCOREP_LocationType { , SCOREP_INVALID_LOCATION_TYPE }
- enum SCOREP_LockType {
  SCOREP_LOCK_EXCLUSIVE,
  SCOREP_LOCK_SHARED,
  SCOREP_INVALID_LOCK_TYPE }
- enum SCOREP_MetricOccurrence {
  SCOREP_METRIC_OCCURRENCE_SYNCHRONOUS_STRICT = 0,
  SCOREP_METRIC_OCCURRENCE_SYNCHRONOUS = 1,
  SCOREP_METRIC_OCCURRENCE_ASYNCHRONOUS = 2,
  SCOREP_INVALID_METRIC_OCCURRENCE }

  Types to be used in defining the occurrence of a sampling set.
- enum SCOREP_MetricScope {
  SCOREP_METRIC_SCOPE_LOCATION = 0,
  SCOREP_METRIC_SCOPE_LOCATION_GROUP = 1,
  SCOREP_METRIC_SCOPE_SYSTEM_TREE_NODE = 2,
  SCOREP_METRIC_SCOPE_GROUP = 3,
  SCOREP_INVALID_METRIC_SCOPE }

  Types to be used in defining the scope of a scoped sampling set.
- enum SCOREP_ParadigmClass { SCOREP_INVALID_PARADIGM_CLASS }

  defines classes of paradigms that are monitored Types:
- enum SCOREP_ParadigmType { SCOREP_INVALID_PARADIGM_TYPE }

  defines paradigms that are be monitored
- enum SCOREP_ParameterType {
  SCOREP_PARAMETER_INT64,
  SCOREP_PARAMETER_UINT64,
  SCOREP_PARAMETER_STRING,
  SCOREP_INVALID_PARAMETER_TYPE }

  *defines types to be used in defining a parameter for parameter based profiling (SCOREP_Definitions_New↩
  Parameter()).*
- enum SCOREP_RegionType { , SCOREP_INVALID_REGION_TYPE }

  specifies a Region
- enum SCOREP_RmaAtomicType

  specifies a RMA Atomic Operation Type.
- enum SCOREP_RmaSyncLevel

  specifies a RMA synchronization level, used by RMA records to be passed to SCOREP_Rma∗() functions.
- enum SCOREP_RmaSyncType { SCOREP_INVALID_RMA_SYNC_TYPE }

  Type of direct RMA synchronization call.
- enum SCOREP_RmaWindowFlag {
  SCOREP_RMA_WINDOW_FLAG_NONE = 0,
  SCOREP_RMA_WINDOW_FLAG_CREATE_DESTROY_EVENTS = (1 << 0) }
- enum SCOREP_SamplingSetClass {
  SCOREP_SAMPLING_SET_ABSTRACT,
  SCOREP_SAMPLING_SET_CPU,
  SCOREP_SAMPLING_SET_GPU }

  Class of locations which recorded a sampling set.
- enum SCOREP_Substrates_RequirementFlag {
  SCOREP_SUBSTRATES_REQUIREMENT_CREATE_EXPERIMENT_DIRECTORY,
  SCOREP_SUBSTRATES_REQUIREMENT_PREVENT_ASYNC_METRICS,
  SCOREP_SUBSTRATES_REQUIREMENT_PREVENT_PER_HOST_AND_ONCE_METRICS,
  SCOREP_SUBSTRATES_NUM_REQUIREMENTS }

M.4.1 Detailed Description

Defines public definitions that are used internally and externally (e.g., by metric plugins, user functions, substrate plugins)

M.5 SCOREP_SubstrateEvents.h File Reference

Description of the substrate plugin events header. For information on how to use substrate plugins, please refer to section 'Substrate Plugins'.

```
#include <stdbool.h>
#include <stddef.h>
#include <scorep/SCOREP_MetricTypes.h>
#include <scorep/SCOREP_PublicTypes.h>
#include <scorep/SCOREP_PublicHandles.h>
```

Typedefs

- typedef void(∗ SCOREP_Substrates_Callback) (void)
- typedef void(∗ SCOREP_Substrates_CallingContextEnterCb) (struct SCOREP_Location ∗location, uint64←
  _t timestamp, SCOREP_CallingContextHandle callingContext, SCOREP_CallingContextHandle previous←
  CallingContext, uint32_t unwindDistance, uint64_t ∗metricValues)
- typedef void(∗ SCOREP_Substrates_CallingContextExitCb) (struct SCOREP_Location ∗location, uint64←
  _t timestamp, SCOREP_CallingContextHandle callingContext, SCOREP_CallingContextHandle previous←
  CallingContext, uint32_t unwindDistance, uint64_t ∗metricValues)
- typedef void(∗ SCOREP_Substrates_DisableRecordingCb) (struct SCOREP_Location ∗location, uint64_←
  t timestamp, SCOREP_RegionHandle regionHandle, uint64_t ∗metricValues)
- typedef void(∗ SCOREP_Substrates_EnableRecordingCb) (struct SCOREP_Location ∗location, uint64_←
  t timestamp, SCOREP_RegionHandle regionHandle, uint64_t ∗metricValues)
- typedef void(∗ SCOREP_Substrates_EnterRegionCb) (struct SCOREP_Location ∗location, uint64_t times-
  tamp, SCOREP_RegionHandle regionHandle, uint64_t ∗metricValues)
- typedef void(∗ SCOREP_Substrates_EnterRewindRegionCb) (struct SCOREP_Location ∗location, uint64_t
  timestamp, SCOREP_RegionHandle regionHandle)
- typedef void(∗ SCOREP_Substrates_ExitRegionCb) (struct SCOREP_Location ∗location, uint64_t times-
  tamp, SCOREP_RegionHandle regionHandle, uint64_t ∗metricValues)
- typedef void(∗ SCOREP_Substrates_ExitRewindRegionCb) (struct SCOREP_Location ∗location, uint64_←
  t timestamp, SCOREP_RegionHandle regionHandle, bool doRewind)
- typedef void(∗ SCOREP_Substrates_IoChangeStatusFlagsCb) (struct SCOREP_Location ∗location,
  uint64_t timestamp, SCOREP_IoHandleHandle handle, SCOREP_IoStatusFlag statusFlags)
- typedef void(∗ SCOREP_Substrates_IoCreateHandleCb) (struct SCOREP_Location ∗location, uint64_←
  t timestamp, SCOREP_IoHandleHandle handle, SCOREP_IoAccessMode mode, SCOREP_IoCreationFlag
  creationFlags, SCOREP_IoStatusFlag statusFlags)
- typedef void(∗ SCOREP_Substrates_IoDeleteFileCb) (struct SCOREP_Location ∗location, uint64_t times-
  tamp, SCOREP_IoParadigmType ioParadigm, SCOREP_IoFileHandle ioFile)
- typedef void(∗ SCOREP_Substrates_IoDestroyHandleCb) (struct SCOREP_Location ∗location, uint64_←
  t timestamp, SCOREP_IoHandleHandle handle)
- typedef void(∗ SCOREP_Substrates_IoDuplicateHandleCb) (struct SCOREP_Location ∗location, uint64_←
  t timestamp, SCOREP_IoHandleHandle oldHandle, SCOREP_IoHandleHandle newHandle, SCOREP_Io←
  StatusFlag statusFlags)
- typedef void(∗ SCOREP_Substrates_IoOperationBeginCb) (struct SCOREP_Location ∗location, uint64_t
  timestamp, SCOREP_IoHandleHandle handle, SCOREP_IoOperationMode mode, SCOREP_IoOperation←
  Flag operationFlags, uint64_t bytesRequest, uint64_t matchingId, uint64_t offset)
- typedef void(∗ SCOREP_Substrates_IoOperationCancelledCb) (struct SCOREP_Location ∗location,
  uint64_t timestamp, SCOREP_IoHandleHandle handle, uint64_t matchingId)
- typedef void(∗ SCOREP_Substrates_IoOperationCompleteCb) (struct SCOREP_Location ∗location,
  uint64_t timestamp, SCOREP_IoHandleHandle handle, SCOREP_IoOperationMode mode, uint64_t bytes←
  Result, uint64_t matchingId)
- typedef void(∗ SCOREP_Substrates_IoOperationIssuedCb) (struct SCOREP_Location ∗location, uint64_←
  t timestamp, SCOREP_IoHandleHandle handle, uint64_t matchingId)

- typedef void(∗ SCOREP_Substrates_IoOperationTestCb) (struct SCOREP_Location ∗location, uint64_↩
  t timestamp, SCOREP_IoHandleHandle handle, uint64_t matchingId)
- typedef void(∗ SCOREP_Substrates_IoSeekCb) (struct SCOREP_Location ∗location, uint64_t timestamp,
  SCOREP_IoHandleHandle handle, int64_t offsetRequest, SCOREP_IoSeekOption whence, uint64_t offset↩
  Result)
- typedef void(∗ SCOREP_Substrates_MpiCollectiveBeginCb) (struct SCOREP_Location ∗location, uint64_t
  timestamp)
- typedef void(∗ SCOREP_Substrates_MpiCollectiveEndCb) (struct SCOREP_Location ∗location, uint64_↩
  t timestamp, SCOREP_InterimCommunicatorHandle communicatorHandle, SCOREP_MpiRank rootRank,
  SCOREP_CollectiveType collectiveType, uint64_t bytesSent, uint64_t bytesReceived)
- typedef void(∗ SCOREP_Substrates_MpiIrecvCb) (struct SCOREP_Location ∗location, uint64_t timestamp,
  SCOREP_MpiRank sourceRank, SCOREP_InterimCommunicatorHandle communicatorHandle, uint32_↩
  t tag, uint64_t bytesReceived, SCOREP_MpiRequestId requestId)
- typedef void(∗ SCOREP_Substrates_MpiIrecvRequestCb) (struct SCOREP_Location ∗location, uint64_↩
  t timestamp, SCOREP_MpiRequestId requestId)
- typedef void(∗ SCOREP_Substrates_MpiIsendCb) (struct SCOREP_Location ∗location, uint64_t times-
  tamp, SCOREP_MpiRank destinationRank, SCOREP_InterimCommunicatorHandle communicatorHandle,
  uint32_t tag, uint64_t bytesSent, SCOREP_MpiRequestId requestId)
- typedef void(∗ SCOREP_Substrates_MpiIsendCompleteCb) (struct SCOREP_Location ∗location, uint64_t
  timestamp, SCOREP_MpiRequestId requestId)
- typedef void(∗ SCOREP_Substrates_MpiNonBlockingCollectiveCompleteCb) (struct SCOREP_Location
  ∗location, uint64_t timestamp, SCOREP_InterimCommunicatorHandle communicatorHandle, SCOREP_↩
  MpiRank rootRank, SCOREP_CollectiveType collectiveType, uint64_t bytesSent, uint64_t bytesReceived,
  SCOREP_MpiRequestId requestId)
- typedef void(∗ SCOREP_Substrates_MpiNonBlockingCollectiveRequestCb) (struct SCOREP_Location
  ∗location, uint64_t timestamp, SCOREP_MpiRequestId requestId)
- typedef void(∗ SCOREP_Substrates_MpiRecvCb) (struct SCOREP_Location ∗location, uint64_t timestamp,
  SCOREP_MpiRank sourceRank, SCOREP_InterimCommunicatorHandle communicatorHandle, uint32_↩
  t tag, uint64_t bytesReceived)
- typedef void(∗ SCOREP_Substrates_MpiRequestCancelledCb) (struct SCOREP_Location ∗location,
  uint64_t timestamp, SCOREP_MpiRequestId requestId)
- typedef void(∗ SCOREP_Substrates_MpiRequestTestedCb) (struct SCOREP_Location ∗location, uint64_↩
  t timestamp, SCOREP_MpiRequestId requestId)
- typedef void(∗ SCOREP_Substrates_MpiSendCb) (struct SCOREP_Location ∗location, uint64_t times-
  tamp, SCOREP_MpiRank destinationRank, SCOREP_InterimCommunicatorHandle communicatorHandle,
  uint32_t tag, uint64_t bytesSent)
- typedef void(∗ SCOREP_Substrates_OnTracingBufferFlushBeginCb) (struct SCOREP_Location ∗location,
  uint64_t timestamp, SCOREP_RegionHandle regionHandle, uint64_t ∗metricValues)
- typedef void(∗ SCOREP_Substrates_OnTracingBufferFlushEndCb) (struct SCOREP_Location ∗location,
  uint64_t timestamp, SCOREP_RegionHandle regionHandle, uint64_t ∗metricValues)
- typedef void(∗ SCOREP_Substrates_ProgramBeginCb) (struct SCOREP_Location ∗location, uint64_↩
  t timestamp, SCOREP_StringHandle programName, uint32_t numberOfProgramArgs, SCOREP_String↩
  Handle ∗programArguments, SCOREP_RegionHandle programRegionHandle, uint64_t processId, uint64_t
  threadId)
- typedef void(∗ SCOREP_Substrates_ProgramEndCb) (struct SCOREP_Location ∗location, uint64_t times-
  tamp, SCOREP_ExitStatus exitStatus, SCOREP_RegionHandle programRegionHandle)
- typedef void(∗ SCOREP_Substrates_RmaAcquireLockCb) (struct SCOREP_Location ∗location, uint64_↩
  t timestamp, SCOREP_RmaWindowHandle windowHandle, uint32_t remote, uint64_t lockId, SCOREP_↩
  LockType lockType)
- typedef void(∗ SCOREP_Substrates_RmaAtomicCb) (struct SCOREP_Location ∗location, uint64_t times-
  tamp, SCOREP_RmaWindowHandle windowHandle, uint32_t remote, SCOREP_RmaAtomicType type,
  uint64_t bytesSent, uint64_t bytesReceived, uint64_t matchingId)
- typedef void(∗ SCOREP_Substrates_RmaCollectiveBeginCb) (struct SCOREP_Location ∗location, uint64↩
  _t timestamp, SCOREP_RmaSyncLevel syncLevel)
- typedef void(∗ SCOREP_Substrates_RmaCollectiveEndCb) (struct SCOREP_Location ∗location, uint64_↩
  t timestamp, SCOREP_CollectiveType collectiveOp, SCOREP_RmaSyncLevel syncLevel, SCOREP_Rma↩
  WindowHandle windowHandle, uint32_t root, uint64_t bytesSent, uint64_t bytesReceived)

- typedef void(∗ SCOREP_Substrates_RmaGroupSyncCb) (struct SCOREP_Location ∗location, uint64_↩
  t timestamp, SCOREP_RmaSyncLevel syncLevel, SCOREP_RmaWindowHandle windowHandle, SCOR↩
  EP_GroupHandle groupHandle)
- typedef void(∗ SCOREP_Substrates_RmaOpCompleteRemoteCb) (struct SCOREP_Location ∗location,
  uint64_t timestamp, SCOREP_RmaWindowHandle windowHandle, uint64_t matchingId)
- typedef void(∗ SCOREP_Substrates_RmaOpTestCb) (struct SCOREP_Location ∗location, uint64_t times-
  tamp, SCOREP_RmaWindowHandle windowHandle, uint64_t matchingId)
- typedef void(∗ SCOREP_Substrates_RmaReleaseLockCb) (struct SCOREP_Location ∗location, uint64_↩
  t timestamp, SCOREP_RmaWindowHandle windowHandle, uint32_t remote, uint64_t lockId)
- typedef void(∗ SCOREP_Substrates_RmaRequestLockCb) (struct SCOREP_Location ∗location, uint64_↩
  t timestamp, SCOREP_RmaWindowHandle windowHandle, uint32_t remote, uint64_t lockId, SCOREP_↩
  LockType lockType)
- typedef void(∗ SCOREP_Substrates_RmaSyncCb) (struct SCOREP_Location ∗location, uint64_t timestamp,
  SCOREP_RmaWindowHandle windowHandle, uint32_t remote, SCOREP_RmaSyncType syncType)
- typedef void(∗ SCOREP_Substrates_RmaTryLockCb) (struct SCOREP_Location ∗location, uint64_t times-
  tamp, SCOREP_RmaWindowHandle windowHandle, uint32_t remote, uint64_t lockId, SCOREP_LockType
  lockType)
- typedef void(∗ SCOREP_Substrates_RmaWaitChangeCb) (struct SCOREP_Location ∗location, uint64_↩
  t timestamp, SCOREP_RmaWindowHandle windowHandle)
- typedef void(∗ SCOREP_Substrates_RmaWinCreateCb) (struct SCOREP_Location ∗location, uint64_↩
  t timestamp, SCOREP_RmaWindowHandle windowHandle)
- typedef void(∗ SCOREP_Substrates_RmaWinDestroyCb) (struct SCOREP_Location ∗location, uint64_↩
  t timestamp, SCOREP_RmaWindowHandle windowHandle)
- typedef void(∗ SCOREP_Substrates_SampleCb) (struct SCOREP_Location ∗location, uint64_t timestamp,
  SCOREP_CallingContextHandle callingContext, SCOREP_CallingContextHandle previousCallingContext,
  uint32_t unwindDistance, SCOREP_InterruptGeneratorHandle interruptGeneratorHandle, uint64_t ∗metric↩
  Values)
- typedef void(∗ SCOREP_Substrates_ThreadForkJoinForkCb) (struct SCOREP_Location ∗location, uint64↩
  _t timestamp, SCOREP_ParadigmType paradigm, uint32_t nRequestedThreads, uint32_t forkSequence↩
  Count)
- typedef void(∗ SCOREP_Substrates_ThreadForkJoinJoinCb) (struct SCOREP_Location ∗location, uint64_t
  timestamp, SCOREP_ParadigmType paradigm)
- typedef void(∗ SCOREP_Substrates_ThreadForkJoinTaskCreateCb) (struct SCOREP_Location ∗location,
  uint64_t timestamp, SCOREP_ParadigmType paradigm, SCOREP_InterimCommunicatorHandle thread↩
  Team, uint32_t threadId, uint32_t generationNumber)
- typedef void(∗ SCOREP_Substrates_ThreadForkJoinTaskSwitchCb) (struct SCOREP_Location ∗location,
  uint64_t timestamp, uint64_t ∗metricValues, SCOREP_ParadigmType paradigm, SCOREP_Interim↩
  CommunicatorHandle threadTeam, uint32_t threadId, uint32_t generationNumber, SCOREP_TaskHandle
  taskHandle)
- typedef void(∗ SCOREP_Substrates_TrackAllocCb) (struct SCOREP_Location ∗location, uint64_t timestamp,
  uint64_t addrAllocated, size_t bytesAllocated, void ∗substrateData[ ], size_t bytesAllocatedMetric, size_↩
  t bytesAllocatedProcess)
- typedef void(∗ SCOREP_Substrates_TrackFreeCb) (struct SCOREP_Location ∗location, uint64_t timestamp,
  uint64_t addrFreed, size_t bytesFreed, void ∗substrateData[ ], size_t bytesAllocatedMetric, size_t bytes↩
  AllocatedProcess)
- typedef void(∗ SCOREP_Substrates_TrackReallocCb) (struct SCOREP_Location ∗location, uint64_t times-
  tamp, uint64_t oldAddr, size_t oldBytesAllocated, void ∗oldSubstrateData[ ], uint64_t newAddr, size_t new↩
  BytesAllocated, void ∗newSubstrateData[ ], size_t bytesAllocatedMetric, size_t bytesAllocatedProcess)
- typedef void(∗ SCOREP_Substrates_TriggerParameterStringCb) (struct SCOREP_Location ∗location,
  uint64_t timestamp, SCOREP_ParameterHandle parameterHandle, SCOREP_StringHandle string_handle)
- typedef void(∗ SCOREP_Substrates_WriteMetricsCb) (struct SCOREP_Location ∗location, uint64_t times-
  tamp, SCOREP_SamplingSetHandle samplingSet, const uint64_t ∗metricValues)

- typedef void(∗ SCOREP_Substrates_RmaPutCb) (struct SCOREP_Location ∗location, uint64_t timestamp,
  SCOREP_RmaWindowHandle windowHandle, uint32_t remote, uint64_t bytes, uint64_t matchingId)

- typedef void(∗ SCOREP_Substrates_RmaOpCompleteBlockingCb) (struct SCOREP_Location ∗location, uint64_t timestamp, SCOREP_RmaWindowHandle windowHandle, uint64_t matchingId)

- typedef void(∗ SCOREP_Substrates_ThreadAcquireLockCb) (struct SCOREP_Location ∗location, uint64_t timestamp, SCOREP_ParadigmType paradigm, uint32_t lockId, uint32_t acquisitionOrder)

- typedef void(∗ SCOREP_Substrates_TriggerCounterInt64Cb) (struct SCOREP_Location ∗location, uint64_t timestamp, SCOREP_SamplingSetHandle counterHandle, int64_t value)

- typedef void(∗ SCOREP_Substrates_TriggerParameterInt64Cb) (struct SCOREP_Location ∗location, uint64_t timestamp, SCOREP_ParameterHandle parameterHandle, int64_t value)

- typedef void(∗ SCOREP_Substrates_ThreadForkJoinTeamBeginCb) (struct SCOREP_Location ∗location, uint64_t timestamp, SCOREP_ParadigmType paradigm, SCOREP_InterimCommunicatorHandle thread↩Team, uint64_t threadId)

- typedef void(∗ SCOREP_Substrates_ThreadForkJoinTaskBeginCb) (struct SCOREP_Location ∗location, uint64_t timestamp, SCOREP_RegionHandle regionHandle, uint64_t ∗metricValues, SCOREP_↩ParadigmType paradigm, SCOREP_InterimCommunicatorHandle threadTeam, uint32_t threadId, uint32_t generationNumber, SCOREP_TaskHandle taskHandle)

- typedef void(∗ SCOREP_Substrates_ThreadCreateWaitCreateCb) (struct SCOREP_Location ∗location, uint64_t timestamp, SCOREP_ParadigmType paradigm, SCOREP_InterimCommunicatorHandle thread↩Team, uint32_t createSequenceCount)

- typedef void(∗ SCOREP_Substrates_IoAcquireLockCb) (struct SCOREP_Location ∗location, uint64_t times-tamp, SCOREP_IoHandleHandle handle, SCOREP_LockType lockType)

- typedef void(∗ SCOREP_Substrates_CommCreateCb) (struct SCOREP_Location ∗location, uint64_t times-tamp, SCOREP_InterimCommunicatorHandle communicatorHandle)

- typedef void(∗ SCOREP_Substrates_CommDestroyCb) (struct SCOREP_Location ∗location, uint64_t times-tamp, SCOREP_InterimCommunicatorHandle communicatorHandle)

Enumerations

- enum SCOREP_Substrates_EventType {
  SCOREP_EVENT_ENABLE_RECORDING = 0,
  SCOREP_EVENT_DISABLE_RECORDING,
  SCOREP_EVENT_ON_TRACING_BUFFER_FLUSH_BEGIN,
  SCOREP_EVENT_ON_TRACING_BUFFER_FLUSH_END,
  SCOREP_EVENT_ENTER_REGION,
  SCOREP_EVENT_EXIT_REGION,
  SCOREP_EVENT_SAMPLE,
  SCOREP_EVENT_CALLING_CONTEXT_ENTER,
  SCOREP_EVENT_CALLING_CONTEXT_EXIT,
  SCOREP_EVENT_ENTER_REWIND_REGION,
  SCOREP_EVENT_EXIT_REWIND_REGION,
  SCOREP_EVENT_MPI_SEND,
  SCOREP_EVENT_MPI_RECV,
  SCOREP_EVENT_MPI_COLLECTIVE_BEGIN,
  SCOREP_EVENT_MPI_COLLECTIVE_END,
  SCOREP_EVENT_MPI_ISEND_COMPLETE,
  SCOREP_EVENT_MPI_IRECV_REQUEST,
  SCOREP_EVENT_MPI_REQUEST_TESTED,
  SCOREP_EVENT_MPI_REQUEST_CANCELLED,
  SCOREP_EVENT_MPI_ISEND,
  SCOREP_EVENT_MPI_IRECV,
  SCOREP_EVENT_RMA_WIN_CREATE,
  SCOREP_EVENT_RMA_WIN_DESTROY,
  SCOREP_EVENT_RMA_COLLECTIVE_BEGIN,
  SCOREP_EVENT_RMA_COLLECTIVE_END,
  SCOREP_EVENT_RMA_TRY_LOCK,
  SCOREP_EVENT_RMA_ACQUIRE_LOCK,
  SCOREP_EVENT_RMA_REQUEST_LOCK,
  SCOREP_EVENT_RMA_RELEASE_LOCK,
  SCOREP_EVENT_RMA_SYNC,
  SCOREP_EVENT_RMA_GROUP_SYNC,
  SCOREP_EVENT_RMA_PUT,
  SCOREP_EVENT_RMA_GET,
  SCOREP_EVENT_RMA_ATOMIC,
  SCOREP_EVENT_RMA_WAIT_CHANGE,
  SCOREP_EVENT_RMA_OP_COMPLETE_BLOCKING,
  SCOREP_EVENT_RMA_OP_COMPLETE_NON_BLOCKING,
  SCOREP_EVENT_RMA_OP_TEST,
  SCOREP_EVENT_RMA_OP_COMPLETE_REMOTE,
  SCOREP_EVENT_THREAD_ACQUIRE_LOCK,
  SCOREP_EVENT_THREAD_RELEASE_LOCK,
  SCOREP_EVENT_TRIGGER_COUNTER_INT64,
  SCOREP_EVENT_TRIGGER_COUNTER_UINT64,
  SCOREP_EVENT_TRIGGER_COUNTER_DOUBLE,
  SCOREP_EVENT_TRIGGER_PARAMETER_INT64,
  SCOREP_EVENT_TRIGGER_PARAMETER_UINT64,
  SCOREP_EVENT_TRIGGER_PARAMETER_STRING,
  SCOREP_EVENT_THREAD_FORK_JOIN_FORK,
  SCOREP_EVENT_THREAD_FORK_JOIN_JOIN,
  SCOREP_EVENT_THREAD_FORK_JOIN_TEAM_BEGIN,
  SCOREP_EVENT_THREAD_FORK_JOIN_TEAM_END,
  SCOREP_EVENT_THREAD_FORK_JOIN_TASK_CREATE,
  SCOREP_EVENT_THREAD_FORK_JOIN_TASK_SWITCH,
  SCOREP_EVENT_THREAD_FORK_JOIN_TASK_BEGIN,
  SCOREP_EVENT_THREAD_FORK_JOIN_TASK_END,
  SCOREP_EVENT_THREAD_CREATE_WAIT_CREATE,
  SCOREP_EVENT_THREAD_CREATE_WAIT_WAIT,
  SCOREP_EVENT_THREAD_CREATE_WAIT_BEGIN,
  SCOREP_EVENT_THREAD_CREATE_WAIT_END,
  SCOREP_EVENT_TRACK_ALLOC,
  SCOREP_EVENT_TRACK_REALLOC,
  SCOREP_EVENT_TRACK_FREE,

SCOREP_SUBSTRATES_NUM_EVENTS }

> *Substrate events. Lists every event that is going to be used by the substrate mechanism. More details can be found in the respective functions. To maintain API stability, new events need to be added at the end of the enum.*

- enum SCOREP_Substrates_Mode {
SCOREP_SUBSTRATES_RECORDING_ENABLED = 0,
SCOREP_SUBSTRATES_RECORDING_DISABLED,
SCOREP_SUBSTRATES_NUM_MODES }

M.5.1 Detailed Description

Description of the substrate plugin events header. For information on how to use substrate plugins, please refer to section 'Substrate Plugins'.

M.5.2 Typedef Documentation

M.5.2.1 typedef void(∗ SCOREP_Substrates_Callback) (void)

M.5.3 Advice

Do not include this file directly, but include SCOREP_SubstratePlugins.h Generic void function pointer for substrate functions.

M.5.3.1 typedef void(∗ SCOREP_Substrates_CallingContextEnterCb) (struct SCOREP_Location ∗location, uint64_t timestamp, SCOREP_CallingContextHandle callingContext, SCOREP_CallingContextHandle previousCallingContext, uint32_t unwindDistance, uint64_t ∗metricValues)

called when entering a region via a instrumentation adapter and Score-P is recording calling contexts (i.e., unwinding is enabled), alternative to SCOREP_Substrates_EnterRegionCb

Parameters

location	location which creates this event
timestamp	timestamp for this event
callingContext	callstack at timestamp
previous↩ CallingContext	calling context of the last SCOREP_Substrates_SampleCb
unwindDistance	number of stack levels changed since the last sample
metricValues	synchronous metrics at timestamp The synchronous metric belong to the last sampling set definition whose metric occurrence is SCOREP_METRIC_OCCURRENCE_SYNCHRON↩ OUS_STRICT and whose class is SCOREP_SAMPLING_SET_CPU

M.5.3.2 typedef void(∗ SCOREP_Substrates_CallingContextExitCb) (struct SCOREP_Location ∗location, uint64_t timestamp, SCOREP_CallingContextHandle callingContext, SCOREP_CallingContextHandle previousCallingContext, uint32_t unwindDistance, uint64_t ∗metricValues)

called when exiting a region via a instrumentation adapter and Score-P is recording calling contexts (i.e., unwinding is enabled). alternative to SCOREP_Substrates_ExitRegionCb

See also

> SCOREP_Substrates_CallingContextEnterCb

M.5.3.3 typedef void(∗ SCOREP_Substrates_CommCreateCb) (struct SCOREP_Location ∗location, uint64_t timestamp, SCOREP_InterimCommunicatorHandle communicatorHandle)

Records communicator creation, tracking Score-P communicator handles.

Parameters

location	location which creates this event
timestamp	timestamp for this event
communicator↩ Handle	communicator handle of the created communicator.

M.5.3.4   typedef void(∗ SCOREP_Substrates_CommDestroyCb) (struct SCOREP_Location ∗location, uint64_t timestamp, SCOREP_InterimCommunicatorHandle communicatorHandle)

Records communicator destruction, tracking Score-P communicator handles.

Parameters

location	location which creates this event
timestamp	timestamp for this event
communicator↩ Handle	communicator handle of the destroyed communicator.

M.5.3.5   typedef void(∗ SCOREP_Substrates_DisableRecordingCb) (struct SCOREP_Location ∗location, uint64_t timestamp, SCOREP_RegionHandle regionHandle, uint64_t ∗metricValues)

called when disabling the recording on a process and its sub-locations This is triggered by user instrumentation and can not be called when in parallel This event will be generated for SCOREP_Substrates_Mode ENABLED, right before the disabling starts. It is currently not called for SCOREP_Substrates_Mode DISABLED. This might change in the future.

See also

> SCOREP_Substrates_EnableRecordingCb

M.5.3.6   typedef void(∗ SCOREP_Substrates_EnableRecordingCb) (struct SCOREP_Location ∗location, uint64_t timestamp, SCOREP_RegionHandle regionHandle, uint64_t ∗metricValues)

called when enabling the recording on a process and its sub-locations This is triggered by user instrumentation and can not be called when in parallel The default mode is enabled. If the default mode is set to disabled by user instrumentation, an enabled event is called before the substrate is finalized. This event will be generated for SCOREP_Substrates_Mode ENABLED, right after the enabling finished. It is currently not called for SCOREP_↩ Substrates_Mode DISABLED. This might change in the future.

Parameters

location	location which creates this event.
timestamp	timestamp for this event
regionHandle	"MEASUREMENT OFF" region
metricValues	synchronous metrics at timestamp

M.5.3.7   typedef void(∗ SCOREP_Substrates_EnterRegionCb) (struct SCOREP_Location ∗location, uint64_t timestamp, SCOREP_RegionHandle regionHandle, uint64_t ∗metricValues)

called when entering a region via some instrumentation adapter. If unwinding is enabled, the event CALLING_C↩ ONTEXT_ENTER will be called instead.

Parameters

location	location which creates this event
timestamp	timestamp for this event
regionHandle	region that is entered
metricValues	synchronous metrics at timestamp. The synchronous metric belong to the last sampling set definition whose metric occurrence is SCOREP_METRIC_OCCURRENCE_SYNCHRON↩OUS_STRICT and whose class is SCOREP_SAMPLING_SET_CPU

M.5.3.8 typedef void(∗ SCOREP_Substrates_EnterRewindRegionCb) (struct SCOREP_Location ∗location, uint64_t timestamp, SCOREP_RegionHandle regionHandle)

called when the user adapter enters a rewind region. The recording of the region and any following region information after this should be discarded when the next SCOREP_Substrates_ExitRewindRegionCb for this regionHandle is called with do_rewind == true

Parameters

location	location which creates this event
timestamp	timestamp for this event
regionHandle	region that is entered

M.5.3.9 typedef void(∗ SCOREP_Substrates_ExitRegionCb) (struct SCOREP_Location ∗location, uint64_t timestamp, SCOREP_RegionHandle regionHandle, uint64_t ∗metricValues)

called when exiting a region via some instrumentation adapter If unwinding is enabled, the event CALLING_CON↩TEXT_EXIT will be called instead.

See also

SCOREP_Substrates_EnterRegionCb

M.5.3.10 typedef void(∗ SCOREP_Substrates_ExitRewindRegionCb) (struct SCOREP_Location ∗location, uint64_t timestamp, SCOREP_RegionHandle regionHandle, bool doRewind)

called when the user adapter exits a rewind region. The recording of the region and any previous regions since SCOREP_Substrates_ExitRewindRegionCb for this regionHandle should be discarded if do_rewind == true

Parameters

location	location which creates this event
timestamp	timestamp for this event
regionHandle	region that is entered
doRewind	whether to discard previously recorded data or not

M.5.3.11 typedef void(∗ SCOREP_Substrates_IoAcquireLockCb) (struct SCOREP_Location ∗location, uint64_t timestamp, SCOREP_IoHandleHandle handle, SCOREP_LockType lockType)

Process I/O lock events (e.g., apply or remove a POSIX advisory lock on an open file)

Parameters

location	The location where this event happened.
timestamp	The time when this event happened.
handle	An active I/O handle.
lockType	Type of the lock (e.g., exclusive or shared.

M.5.3.12 typedef void(∗ SCOREP_Substrates_IoChangeStatusFlagsCb) (struct SCOREP_Location ∗location, uint64_t timestamp, SCOREP_IoHandleHandle handle, SCOREP_IoStatusFlag statusFlags)

Records a change to the status flags associated with an active I/O handle.

Parameters

location	The location where this event happened.
timestamp	The time when this event happened.
handle	An active I/O handle.
statusFlags	Set flags (e.g., close-on-exec, append, etc.).

M.5.3.13 typedef void(∗ SCOREP_Substrates_IoCreateHandleCb) (struct SCOREP_Location ∗location, uint64_t timestamp, SCOREP_IoHandleHandle handle, SCOREP_IoAccessMode mode, SCOREP_IoCreationFlag creationFlags, SCOREP_IoStatusFlag statusFlags)

Records the creation of a new active I/O handle that can be used by subsequent I/O operation events. An IoHandle is active between a pair of consecutive IoCreateHandle and IoDestroyHandle events. All locations of a location group have access to an active IoHandle.

Parameters

location	The location which creates this event.
timestamp	The timestamp, when the event event occurred.
handle	A reference to the affected I/O handle which will be activated by this record.
mode	Determines which I/O operations can be applied to this I/O handle (e.g., read-only, write-only, read-write)
creationFlags	Requested I/O handle creation flags (e.g., create, exclusive, etc.).
statusFlags	I/O handle status flags which will be associated with the handle attribute (e.g., append, create, close-on-exec, async, etc).

M.5.3.14 typedef void(∗ SCOREP_Substrates_IoDeleteFileCb) (struct SCOREP_Location ∗location, uint64_t timestamp, SCOREP_IoParadigmType ioParadigm, SCOREP_IoFileHandle ioFile)

Records the deletion of an I/O file.

Parameters

location	The location where this event happened.
timestamp	The time when this event happened.

ioParadigm	The I/O paradigm which induced the deletion.
ioFile	File identifier.

M.5.3.15 typedef void(∗ SCOREP_Substrates_IoDestroyHandleCb) (struct SCOREP_Location ∗location, uint64_t timestamp,
SCOREP_IoHandleHandle handle)

Records the end of an active I/O handle's lifetime.

Parameters

location	The location where this event happened.
timestamp	The time when this event happened.
handle	A reference to the affected I/O handle which will be activated by this record.

M.5.3.16 typedef void(∗ SCOREP_Substrates_IoDuplicateHandleCb) (struct SCOREP_Location ∗location, uint64_t timestamp,
SCOREP_IoHandleHandle oldHandle, SCOREP_IoHandleHandle newHandle, SCOREP_IoStatusFlag statusFlags)

Records the duplication of an already existing active I/O handle.

Parameters

location	The location where this event happened.
timestamp	The time when this event happened.
oldHandle	An active I/O handle.
newHandle	A previously inactive I/O handle which will be activated by this record.
statusFlags	The status flag for the new I/O handle newHandle. No status flags will be inherited from the I/O handle oldHandle.

M.5.3.17 typedef void(∗ SCOREP_Substrates_IoOperationBeginCb) (struct SCOREP_Location ∗location, uint64_t timestamp,
SCOREP_IoHandleHandle handle, SCOREP_IoOperationMode mode, SCOREP_IoOperationFlag
operationFlags, uint64_t bytesRequest, uint64_t matchingId, uint64_t offset)

Records the begin of a file operation (read, write, etc.).

Parameters

location	The location where this event happened.
timestamp	The time when this event happened.
handle	An active I/O handle.
mode	Mode of an I/O handle operation (e.g., read or write).
operationFlags	Special semantic of this operation.
bytesRequest	Requested bytes to write/read.
matchingId	Identifier used to correlate associated event records of an I/O operation. This identifier is unique for the referenced I/O handle.

M.5.3.18 typedef void(∗ SCOREP_Substrates_IoOperationCancelledCb) (struct SCOREP_Location ∗location, uint64_t
timestamp, SCOREP_IoHandleHandle handle, uint64_t matchingId)

Records the successful cancellation of a non-blocking operation (read, write etc.) on an active I/O handle.

Parameters

location	The location where this event happened.
timestamp	The time when this event happened.
handle	An active I/O handle.
matchingId	Identifier used to correlate associated event records of an I/O operation. This identifier is unique for the referenced I/O handle.

M.5.3.19 **typedef void(∗ SCOREP_Substrates_IoOperationCompleteCb) (struct SCOREP_Location ∗location, uint64_t timestamp, SCOREP_IoHandleHandle handle, SCOREP_IoOperationMode mode, uint64_t bytesResult, uint64_t matchingId)**

Records the end of a file operation (read, write etc.) on an active I/O handle.

Parameters

location	The location where this event happened.
timestamp	The time when this event happened.
handle	An active I/O handle.
mode	Mode of an I/O handle operation (e.g., read or write).
bytesResult	Number of actual transferred bytes.
matchingId	Identifier used to correlate associated event records of an I/O operation. This identifier is unique for the referenced I/O handle.

M.5.3.20 **typedef void(∗ SCOREP_Substrates_IoOperationIssuedCb) (struct SCOREP_Location ∗location, uint64_t timestamp, SCOREP_IoHandleHandle handle, uint64_t matchingId)**

Records the successful initiation of a non-blocking operation (read, write etc.) on an active I/O handle.

Parameters

location	The location where this event happened.
timestamp	The time when this event happened.
handle	An active I/O handle.
matchingId	Identifier used to correlate associated event records of an I/O operation. This identifier is unique for the referenced I/O handle.

M.5.3.21 **typedef void(∗ SCOREP_Substrates_IoOperationTestCb) (struct SCOREP_Location ∗location, uint64_t timestamp, SCOREP_IoHandleHandle handle, uint64_t matchingId)**

Records an unsuccessful test whether an I/O operation has already finished.

Parameters

location	The location where this event happened.
timestamp	The time when this event happened.
handle	An active I/O handle.

matchingId	Identifier used to correlate associated event records of an I/O operation. This identifier is unique for the referenced I/O handle.

M.5.3.22   typedef void(∗ SCOREP_Substrates_IoSeekCb) (struct SCOREP_Location ∗location, uint64_t timestamp, SCOREP_IoHandleHandle handle, int64_t offsetRequest, SCOREP_IoSeekOption whence, uint64_t offsetResult)

Records a change of the position, e.g., within a file.

Parameters

location	The location where this event happened.
timestamp	The time when this event happened.
handle	An active I/O handle.
offsetRequest	Requested offset.
whence	Position inside the file from where offsetRequest should be applied (e.g., absolute from the start or end, relative to the current position).
offsetResult	Resulting offset, e.g., within the file relative to the beginning of the file.

M.5.3.23   typedef void(∗ SCOREP_Substrates_MpiCollectiveBeginCb) (struct SCOREP_Location ∗location, uint64_t timestamp)

Called when an MPI collective is recognized by the MPI adapter before it is started. See also the MPI specifications at https://www.mpi-forum.org/docs/ More information on the type is passed with the SCOREP_↩Substrates_MpiCollectiveEndCb callback

Parameters

location	location which creates this event
timestamp	timestamp for this event

M.5.3.24   typedef void(∗ SCOREP_Substrates_MpiCollectiveEndCb) (struct SCOREP_Location ∗location, uint64_t timestamp, SCOREP_InterimCommunicatorHandle communicatorHandle, SCOREP_MpiRank rootRank, SCOREP_CollectiveType collectiveType, uint64_t bytesSent, uint64_t bytesReceived)

called when an MPI collective is recognized by the MPI adapter after it is finished. see also the MPI specifications at https://www.mpi-forum.org/docs/

Parameters

location	location which creates this event
timestamp	timestamp for this event
communicator↩Handle	communicator handle to which the location rank and the destinationRank belong
rootRank	rank that sent the message that is to be received
collectiveType	type of the collective operation
bytesSent	number of bytes received with this message
bytesReceived	number of bytes received with this message

M.5.3.25   typedef void(∗ SCOREP_Substrates_MpiIrecvCb) (struct SCOREP_Location ∗location, uint64_t timestamp, SCOREP_MpiRank sourceRank, SCOREP_InterimCommunicatorHandle communicatorHandle, uint32_t tag, uint64_t bytesReceived, SCOREP_MpiRequestId requestId)

Finishes an MpiIrecvRequest see also the MPI specifications at https://www.mpi-forum.org/docs/

Parameters

location	location which sends an MPI message
timestamp	timestamp for this event
sourceRank	rank that sent the message
communicator↩ Handle	communicator of this location and the source rank
tag	MPI tag
bytesReceived	number of bytes received with this message
requestId	request ID of the non blocking communication to be canceled (see MPI standard)

M.5.3.26   typedef void(∗ SCOREP_Substrates_MpiIrecvRequestCb) (struct SCOREP_Location ∗location, uint64_t timestamp, SCOREP_MpiRequestId requestId)

Called from MPI adapter when an MPI_Irecv is initialized see also the MPI specifications at https://www.↩ mpi-forum.org/docs/ Previously there should be a SCOREP_Substrates_MpiIrecvCb with the same requestId

Parameters

location	location which creates this event
timestamp	timestamp for this event
requestId	request ID of the MPI_Irecv (see MPI standard)

M.5.3.27   typedef void(∗ SCOREP_Substrates_MpiIsendCb) (struct SCOREP_Location ∗location, uint64_t timestamp, SCOREP_MpiRank destinationRank, SCOREP_InterimCommunicatorHandle communicatorHandle, uint32_t tag, uint64_t bytesSent, SCOREP_MpiRequestId requestId)

Initialize a non blocking send via MPI see also the MPI specifications at https://www.mpi-forum.↩ org/docs/

Parameters

location	location which creates this event
timestamp	timestamp for this event
destinationRank	rank that will receive the message
communicator↩ Handle	communicator of this location and the target rank
tag	MPI tag
bytesSent	number of sent bytes
requestId	request ID of the non blocking communication to be canceled (see MPI standard)

M.5.3.28   typedef void(∗ SCOREP_Substrates_MpiIsendCompleteCb) (struct SCOREP_Location ∗location, uint64_t timestamp, SCOREP_MpiRequestId requestId)

Called from MPI adapter when an MPI_Isend is completed see also the MPI specifications at https://www.↩ mpi-forum.org/docs/ Previously there should be a SCOREP_Substrates_MpiIsendCb with the same requestId

Parameters

location	location which creates this event
timestamp	timestamp for this event
requestId	request ID of the MPI_Isend (see MPI standard)

M.5.3.29   typedef void(∗ SCOREP_Substrates_MpiNonBlockingCollectiveCompleteCb) (struct SCOREP_Location ∗location, uint64_t timestamp, SCOREP_InterimCommunicatorHandle communicatorHandle, SCOREP_MpiRank rootRank, SCOREP_CollectiveType collectiveType, uint64_t bytesSent, uint64_t bytesReceived, SCOREP_MpiRequestId requestId)

Called when a MPI non-blocking collective operation is completed by the MPI adapter. see also the MPI specifications at `https://www.mpi-forum.org/docs/`

Parameters

location	location which creates this event
timestamp	timestamp for this event
communicator↩ Handle	communicator handle to which the location rank and the destinationRank belong
rootRank	rank that sent the message that is to be received
collectiveType	type of the collective operation
bytesSent	number of bytes received with this message
bytesReceived	number of bytes received with this message
requestId	request ID (see MPI standard)

M.5.3.30   typedef void(∗ SCOREP_Substrates_MpiNonBlockingCollectiveRequestCb) (struct SCOREP_Location ∗location, uint64_t timestamp, SCOREP_MpiRequestId requestId)

Called when a non-blocking MPI collective operation is initiated by the MPI adapter. See also the MPI specifications at `https://www.mpi-forum.org/docs/` More information on the type is passed with the SCOREP_↩ Substrates_MpiNonBlockingCollectiveCompleteCb callback

Parameters

location	location which creates this event
timestamp	timestamp for this event
requestId	request ID (see MPI standard)

M.5.3.31   typedef void(∗ SCOREP_Substrates_MpiRecvCb) (struct SCOREP_Location ∗location, uint64_t timestamp, SCOREP_MpiRank sourceRank, SCOREP_InterimCommunicatorHandle communicatorHandle, uint32_t tag, uint64_t bytesReceived)

called when an MPI_Recv is recognized by the MPI adapter see also the MPI specifications at `https://www.↩ mpi-forum.org/docs/` There should be a SCOREP_Substrates_MpiSendCb call on the location of the sourceRank.

Parameters

location	location which creates this event
timestamp	timestamp for this event
sourceRank	rank that sent the message that is to be received
communicator↩ Handle	communicator handle to which the location rank and the destinationRank belong
tag	provided MPI tag for this message
bytesReceived	number of bytes received with this message

M.5.3.32   typedef void(∗ SCOREP_Substrates_MpiRequestCancelledCb) (struct SCOREP_Location ∗location, uint64_t timestamp, SCOREP_MpiRequestId requestId)

Called from MPI adapter when a non-blocking communication is cancelled see also the MPI specifications at `https://www.mpi-forum.org/docs/` Previously there should be an initialization of the non-blocking communication with the same requestId

Parameters

location	location which creates this event
timestamp	timestamp for this event
requestId	request ID of the non blocking communication to be canceled (see MPI standard)

M.5.3.33   typedef void(∗ SCOREP_Substrates_MpiRequestTestedCb) (struct SCOREP_Location ∗location, uint64_t timestamp, SCOREP_MpiRequestId requestId)

Called from MPI adapter when the status of a non-blocking communication is tested see also the MPI specifications at `https://www.mpi-forum.org/docs/` Previously there should be an initialization of the non-blocking communication with the same requestId

Parameters

location	location which creates this event
timestamp	timestamp for this event
requestId	request ID of the non blocking communication to be tested (see MPI standard)

M.5.3.34   typedef void(∗ SCOREP_Substrates_MpiSendCb) (struct SCOREP_Location ∗location, uint64_t timestamp, SCOREP_MpiRank destinationRank, SCOREP_InterimCommunicatorHandle communicatorHandle, uint32_t tag, uint64_t bytesSent)

called when an MPI_Send is recognized by the MPI adapter see also the MPI specifications at `https://www.↩ mpi-forum.org/docs/` There should be a SCOREP_Substrates_MpiRecvCb call on the location of the destinationRank.

Parameters

location	location which creates this event
timestamp	timestamp for this event

destinationRank	rank that should receive the message
communicator↩ *Handle*	communicator handle to which the location rank and the destinationRank belong
tag	provided MPI tag for this message
bytesSent	number of bytes sent with this message

M.5.3.35 typedef void(∗ SCOREP_Substrates_OnTracingBufferFlushBeginCb) (struct SCOREP_Location ∗location, uint64_t timestamp, SCOREP_RegionHandle regionHandle, uint64_t ∗metricValues)

called when flushing the tracing buffer of a location

Parameters

location	location which creates this event
timestamp	timestamp for this event
regionHandle	"TRACE BUFFER FLUSH" region
metricValues	synchronous metrics at timestamp

M.5.3.36 typedef void(∗ SCOREP_Substrates_OnTracingBufferFlushEndCb) (struct SCOREP_Location ∗location, uint64_t timestamp, SCOREP_RegionHandle regionHandle, uint64_t ∗metricValues)

called after flushing the tracing buffer of a location,

See also

    SCOREP_Substrates_OnTracingBufferFlushBeginCb

M.5.3.37 typedef void(∗ SCOREP_Substrates_ProgramBeginCb) (struct SCOREP_Location ∗location, uint64_t timestamp, SCOREP_StringHandle programName, uint32_t numberOfProgramArgs, SCOREP_StringHandle ∗programArguments, SCOREP_RegionHandle programRegionHandle, uint64_t processId, uint64_t threadId)

called on each process when the application starts

Parameters

location	location which creates this event
timestamp	timestamp for this event
programName	name of the application
numberOf↩ *ProgramArgs*	number of arguments of the application
program↩ *Arguments*	the list of arguments
program↩ *RegionHandle*	the program region that is entered
processId	process identifier of the operating system If processId is not set, it has the value SCOREP↩ _INVALID_PID.

threadId	thread identifier of the operating system

M.5.3.38   typedef void(∗ SCOREP_Substrates_ProgramEndCb) (struct SCOREP_Location ∗location, uint64_t timestamp, SCOREP_ExitStatus exitStatus, SCOREP_RegionHandle programRegionHandle)

called on each process when the application ends

Parameters

location	location which creates this event
timestamp	timestamp for this event
exitStatus	exit status of the application
program↩ RegionHandle	the program region that is entered

M.5.3.39   typedef void(∗ SCOREP_Substrates_RmaAcquireLockCb) (struct SCOREP_Location ∗location, uint64_t timestamp, SCOREP_RmaWindowHandle windowHandle, uint32_t remote, uint64_t lockId, SCOREP_LockType lockType)

Marks the time that a lock is granted. This is the typical situation. It has to be followed by a matching *SCOREP_↩ Substrates_RmaRequestLockCb* record later on.

Parameters

location	location which creates this event
timestamp	timestamp for this event
windowHandle	Memory window.
remote	Rank of target in context of window.
lockId	Lock id in context of window.
lockType	Type of lock (shared vs. exclusive).

M.5.3.40   typedef void(∗ SCOREP_Substrates_RmaAtomicCb) (struct SCOREP_Location ∗location, uint64_t timestamp, SCOREP_RmaWindowHandle windowHandle, uint32_t remote, SCOREP_RmaAtomicType type, uint64_t bytesSent, uint64_t bytesReceived, uint64_t matchingId)

The atomic RMA operations are similar to the get and put operations. As an additional field they provide the type of operation. Depending on the type, data may be received, sent, or both, therefore, the sizes are specified separately. Matching the local and optionally remote completion works the same way as for get and put operations.

Parameters

location	location which creates this event
timestamp	timestamp for this event
windowHandle	Window.
remote	Rank of target in context of window.
type	Type of atomic operation (see *SCOREP_RmaAtomicType*).
bytesSent	Number of bytes transferred to remote target.

bytesReceived	Number of bytes transferred from remote target.
matchingId	Matching number.

M.5.3.41 **typedef void(∗ SCOREP_Substrates_RmaCollectiveBeginCb) (struct SCOREP_Location ∗location, uint64_t timestamp, SCOREP_RmaSyncLevel syncLevel)**

begin a collective operation on an MPI remote memory access window see also the MPI specifications at `https↩`
`://www.mpi-forum.org/docs/` More information is passed with the next call SCOREP_Substrates_Rma↩
CollectiveEndCb

Parameters

location	location which creates this event
timestamp	timestamp for this event
syncLevel	synchronization level

M.5.3.42 **typedef void(∗ SCOREP_Substrates_RmaCollectiveEndCb) (struct SCOREP_Location ∗location, uint64_t timestamp, SCOREP_CollectiveType collectiveOp, SCOREP_RmaSyncLevel syncLevel, SCOREP_RmaWindowHandle windowHandle, uint32_t root, uint64_t bytesSent, uint64_t bytesReceived)**

end a collective operation on an MPI remote memory access window see also the MPI specifications at `https↩`
`://www.mpi-forum.org/docs/`

Parameters

location	location which creates this event
timestamp	timestamp for this event
collectiveOp	type of the collective operation
syncLevel	synchronization level
windowHandle	the previously defined and created window handle
root	Root process/rank if there is one
bytesSent	number of bytes sent
bytesReceived	number of bytes received

M.5.3.43 **typedef void(∗ SCOREP_Substrates_RmaGroupSyncCb) (struct SCOREP_Location ∗location, uint64_t timestamp, SCOREP_RmaSyncLevel syncLevel, SCOREP_RmaWindowHandle windowHandle, SCOREP_GroupHandle groupHandle)**

This record marks the synchronization of a sub-group of the locations associated with the given memory window. It needs to be recorded for all participating locations.

Parameters

location	location which creates this event
timestamp	timestamp for this event
syncLevel	Synchronization level.
windowHandle	Memory window.

groupHandle	Group of participating processes or threads.

M.5.3.44   **typedef void(∗ SCOREP_Substrates_RmaOpCompleteBlockingCb) (struct SCOREP_Location ∗location, uint64_t timestamp, SCOREP_RmaWindowHandle windowHandle, uint64_t matchingId)**

The completion records mark the end of RMA operations. Local completion for every RMA operation (get, put, or atomic operation) always has to be marked with either *SCOREP_Substrates_RmaOpCompleteBlockingCb* or *SCOREP_Substrates_RmaOpCompleteNonBlockingCb* using the same matching number as the RMA operation record. An RMA operation is blocking when the operation completes locally before leaving the call, for non-blocking operations local completion has to be ensured by a subsequent call.

Parameters

location	location which creates this event
timestamp	timestamp for this event
windowHandle	Memory window.
matchingId	Matching number.

M.5.3.45   **typedef void(∗ SCOREP_Substrates_RmaOpCompleteRemoteCb) (struct SCOREP_Location ∗location, uint64_t timestamp, SCOREP_RmaWindowHandle windowHandle, uint64_t matchingId)**

An optional remote completion point can be specified with SCOREP_Substrates_RmaOpCompleteRemoteCb. It is recorded on the same location as the RMA operation itself. Again, multiple RMA operations may map to the same SCOREP_Substrates_RmaOpCompleteRemoteCb. The target locations are not explicitly specified but implicitly as all those that were referenced in matching RMA operations.

Parameters

location	location which creates this event
timestamp	timestamp for this event
windowHandle	Memory window.
matchingId	Matching number.

M.5.3.46   **typedef void(∗ SCOREP_Substrates_RmaOpTestCb) (struct SCOREP_Location ∗location, uint64_t timestamp, SCOREP_RmaWindowHandle windowHandle, uint64_t matchingId)**

This record indicates a test for completion. It is only useful for non-blocking RMA calls where the API supports such a test. The test record stands for a negative outcome, otherwise a completion record is written (see *SCOREP_↩ Substrates_RmaOpCompleteRemoteCb*).

Parameters

location	location which creates this event
timestamp	timestamp for this event
windowHandle	Memory window.
matchingId	Matching number.

M.5.3.47   **typedef void(∗ SCOREP_Substrates_RmaPutCb) (struct SCOREP_Location ∗location, uint64_t timestamp, SCOREP_RmaWindowHandle windowHandle, uint32_t remote, uint64_t bytes, uint64_t matchingId)**

The get and put operations access remote memory addresses. The corresponding get and put records mark when they are issued. The actual start and the completion may happen later.

Parameters

location	location which creates this event
timestamp	timestamp for this event
windowHandle	Memory window.
remote	Rank of target in context of window.
bytes	Number of bytes transferred.
matchingId	Matching number.

Note

The matching number allows to reference the point of completion of the operation. It will reappear in a completion record on the same location.

M.5.3.48   typedef void(∗ SCOREP_Substrates_RmaReleaseLockCb) (struct SCOREP_Location ∗location, uint64_t timestamp, SCOREP_RmaWindowHandle windowHandle, uint32_t remote, uint64_t lockId)

Marks the time the lock is freed. It contains all fields that are necessary to match it to either an earlier *SCOR↩EP_Substrates_RmaAcquireLockCb* or *SCOREP_Substrates_RmaRequestLockCb* event and is required to follow either of the two.

Parameters

location	location which creates this event
timestamp	timestamp for this event
windowHandle	Memory window.
remote	Rank of target in context of window.
lockId	Lock id in context of window.

M.5.3.49   typedef void(∗ SCOREP_Substrates_RmaRequestLockCb) (struct SCOREP_Location ∗location, uint64_t timestamp, SCOREP_RmaWindowHandle windowHandle, uint32_t remote, uint64_t lockId, SCOREP_LockType lockType)

This record marks the time that a request for a lock is issued where the RMA model ensures that the lock is granted eventually without further notification. As of now this is specific for MPI. In this case, the *SCOREP_RmaAcquireLock* event is not present.

Parameters

location	location which creates this event
timestamp	timestamp for this event
win	Memory window.
remote	Rank of target in context of window.
lockId	Lock id in context of window.
lockType	Type of lock (shared vs. exclusive).

M.5.3.50   typedef void(∗ SCOREP_Substrates_RmaSyncCb) (struct SCOREP_Location ∗location, uint64_t timestamp, SCOREP_RmaWindowHandle windowHandle, uint32_t remote, SCOREP_RmaSyncType syncType)

This record marks a simple pairwise synchronization.

Parameters

location	location which creates this event
timestamp	timestamp for this event
windowHandle	Memory window.
remote	Rank of target in context of window.
syncType	Type of direct RMA synchronization call (e.g. SCOREP_RMA_SYNC_TYPE_MEMORY, S↵COREP_RMA_SYNC_TYPE_NOTIFY_IN, SCOREP_RMA_SYNC_TYPE_NOTIFY_OUT).

M.5.3.51 typedef void(∗ SCOREP_Substrates_RmaTryLockCb) (struct SCOREP_Location ∗location, uint64_t timestamp, SCOREP_RmaWindowHandle windowHandle, uint32_t remote, uint64_t lockId, SCOREP_LockType lockType)

An attempt to acquire a lock which turns out negative can be marked with SCOREP_Substrates_RmaTryLockCb. In this case, no release record may follow. With this a series of unsuccessful locking attempts can be identified. If an lock attempt is successful, it is marked with *SCOREP_Substrates_RmaAcquireLockCb* right away instead of a pair of *SCOREP_Substrates_RmaTryLockCb* and @ SCOREP_Substrates_RmaAcquireLockCb. see also the MPI specifications at https://www.mpi-forum.org/docs/

Parameters

location	location which creates this event
timestamp	timestamp for this event
windowHandle	the previously defined and created window handle
remote	Rank of target in context of window
lockId	Lock id in context of window.
lockType	Type of lock (shared vs. exclusive).

M.5.3.52 typedef void(∗ SCOREP_Substrates_RmaWaitChangeCb) (struct SCOREP_Location ∗location, uint64_t timestamp, SCOREP_RmaWindowHandle windowHandle)

The SCOREP_EVENT_RMA_WAIT_CHANGE event marks a synchronization point that blocks until a remote operation modifies a given memory field. This event marks the beginning of the waiting period. The memory field in question is part of the specified window.

Parameters

location	location which creates this event
timestamp	timestamp for this event
windowHandle	Memory window.

M.5.3.53 typedef void(∗ SCOREP_Substrates_RmaWinCreateCb) (struct SCOREP_Location ∗location, uint64_t timestamp, SCOREP_RmaWindowHandle windowHandle)

create a remote memory access window see also the MPI specifications at https://www.mpi-forum.↵org/docs/

Parameters

location	location which creates this event
timestamp	timestamp for this event
windowHandle	previously defined window handle

M.5.3.54   typedef void(∗ SCOREP_Substrates_RmaWinDestroyCb) (struct SCOREP_Location ∗location, uint64_t timestamp,
SCOREP_RmaWindowHandle windowHandle)

destroy a remote memory access window see also the MPI specifications at <code>https://www.mpi-forum.↵</code>
<code>org/docs/</code>

Parameters

location	location which creates this event
timestamp	timestamp for this event
windowHandle	previously defined and created window handle

M.5.3.55   typedef void(∗ SCOREP_Substrates_SampleCb) (struct SCOREP_Location ∗location, uint64_t timestamp,
SCOREP_CallingContextHandle callingContext, SCOREP_CallingContextHandle previousCallingContext, uint32_t
unwindDistance, SCOREP_InterruptGeneratorHandle interruptGeneratorHandle, uint64_t ∗metricValues)

called when a sampling adapter interrupts the workload and records a sample. Called from a signal handler, so
used functions should be async-signal safe. If a function is not signal safe, but is interrupted by a signal (i.e., a
sample event) and used within the signal context, its behavior is unpredictable.

Parameters

location	location which creates this event
timestamp	timestamp for this event
callingContext	callstack at timestamp
previous↵CallingContext	calling context of the last SCOREP_Substrates_SampleCb
unwindDistance	number of stack levels changed since the last sample
interrupt↵Generator↵Handle	source that interrupted the workload
metricValues	synchronous metrics at timestamp The synchronous metric belong to the last sampling set definition whose metric occurrence is SCOREP_METRIC_OCCURRENCE_SYNCHRON↵OUS_STRICT and whose class is SCOREP_SAMPLING_SET_CPU

M.5.3.56   typedef void(∗ SCOREP_Substrates_ThreadAcquireLockCb) (struct SCOREP_Location ∗location, uint64_t timestamp,
SCOREP_ParadigmType paradigm, uint32_t lockId, uint32_t acquisitionOrder)

Process a thread acquire/release lock event in the measurement system.

Parameters

location	location which creates this event

timestamp	timestamp for this event
paradigm	the underlying parallelization paradigm of the lock
lockId	A unique ID to identify the lock, maintained by the caller.
acquisitionOrder	A monotonically increasing id to determine the order of lock acquisitions. Same for corresponding acquire-release events.

M.5.3.57 **typedef void(∗ SCOREP_Substrates_ThreadCreateWaitCreateCb) (struct SCOREP_Location ∗location, uint64_t timestamp, SCOREP_ParadigmType paradigm, SCOREP_InterimCommunicatorHandle threadTeam, uint32_t createSequenceCount)**

process a thread event for a create/wait thread model instrumentation adapter e.g., pthreads

Parameters

location	location which creates this event
timestamp	timestamp for this event
paradigm	One of the predefined threading models.
threadTeam	previously defined thread team
create↩ SequenceCount	a process unique increasing number that is increased at every SCOREP_Substrates_↩ ThreadCreateWaitCreateCb and SCOREP_Substrates_ThreadForkJoinForkCb
threadId	thread identifier of the operating system If threadId is not set, it has the value SCOREP_IN↩ VALID_TID.

M.5.3.58 **typedef void(∗ SCOREP_Substrates_ThreadForkJoinForkCb) (struct SCOREP_Location ∗location, uint64_t timestamp, SCOREP_ParadigmType paradigm, uint32_t nRequestedThreads, uint32_t forkSequenceCount)**

called from threading instrumentation adapters before a thread team is forked, e.g., before an OpenMP parallel region

Parameters

location	location which creates this event
timestamp	timestamp for this event
paradigm	threading paradigm
nRequested↩ Threads	number of threads to be forked. Note that this does not necessarily represent actual threads but threads can also be reused, e..g, in OpenMP runtimes.
forkSequence↩ Count	an increasing number, unique for each process that allows to identify a parallel region

M.5.3.59 **typedef void(∗ SCOREP_Substrates_ThreadForkJoinJoinCb) (struct SCOREP_Location ∗location, uint64_t timestamp, SCOREP_ParadigmType paradigm)**

called from threading instrumentation after a thread team is joined, e.g., after an OpenMP parallel region

Parameters

location	location which creates this event

timestamp	timestamp for this event
paradigm	threading paradigm

M.5.3.60 **typedef void(∗ SCOREP_Substrates_ThreadForkJoinTaskBeginCb) (struct SCOREP_Location ∗location, uint64_t timestamp, SCOREP_RegionHandle regionHandle, uint64_t ∗metricValues, SCOREP_ParadigmType paradigm, SCOREP_InterimCommunicatorHandle threadTeam, uint32_t threadId, uint32_t generationNumber, SCOREP_TaskHandle taskHandle)**

Process a task begin/end event

Parameters

location	location which creates this event
timestamp	timestamp for this event
metricValues	synchronous metrics
paradigm	One of the predefined threading models.
regionHandle	Region handle of the task region.
threadTeam	previously defined thread team
threadId	Id of the this thread within the team of threads that constitute the parallel region.
generation↩ Number	The sequence number for this task. Each task created gets a thread private generation number attached. Combined with the *threadId*, this constitutes a unique task ID inside the parallel region.
taskHandle	A handle to the executed task

M.5.3.61 **typedef void(∗ SCOREP_Substrates_ThreadForkJoinTaskCreateCb) (struct SCOREP_Location ∗location, uint64_t timestamp, SCOREP_ParadigmType paradigm, SCOREP_InterimCommunicatorHandle threadTeam, uint32_t threadId, uint32_t generationNumber)**

Process a task create event in the measurement system.

Parameters

location	location which creates this event
timestamp	timestamp for this event
paradigm	One of the predefined threading models.
threadTeam	previously defined thread team
threadId	Id of the this thread within the team of threads that constitute the parallel region.
generation↩ Number	The sequence number for this task. Each task gets a thread private generation number of the creating thread attached. Combined with the *threadId*, this constitutes a unique task ID inside the parallel region.

M.5.3.62 **typedef void(∗ SCOREP_Substrates_ThreadForkJoinTaskSwitchCb) (struct SCOREP_Location ∗location, uint64_t timestamp, uint64_t ∗metricValues, SCOREP_ParadigmType paradigm, SCOREP_InterimCommunicatorHandle threadTeam, uint32_t threadId, uint32_t generationNumber, SCOREP_TaskHandle taskHandle)**

Process a task switch event

Parameters

location	location which creates this event
timestamp	timestamp for this event
metricValues	synchronous metrics
paradigm	One of the predefined threading models.
threadTeam	previously defined thread team
threadId	Id of the this thread within the team of threads that constitute the parallel region.
generation↩Number	The sequence number for this task. Each task gets a thread private generation number of the creating thread attached. Combined with the *threadId*, this constitutes a unique task ID inside the parallel region.
taskHandle	A handle to the resumed task.

M.5.3.63   typedef void(∗ SCOREP_Substrates_ThreadForkJoinTeamBeginCb) (struct SCOREP_Location ∗location, uint64_t timestamp, SCOREP_ParadigmType paradigm, SCOREP_InterimCommunicatorHandle threadTeam, uint64_t threadId)

called from threading instrumentation after a thread team is created/before it is joined.

Parameters

location	location which creates this event
timestamp	timestamp for this event
paradigm	threading paradigm
threadTeam	previously defined thread team
threadId	thread identifier of the operating system If threadId is not set, it has the value SCOREP_IN↩VALID_TID.

M.5.3.64   typedef void(∗ SCOREP_Substrates_TrackAllocCb) (struct SCOREP_Location ∗location, uint64_t timestamp, uint64_t addrAllocated, size_t bytesAllocated, void ∗substrateData[ ], size_t bytesAllocatedMetric, size_t bytesAllocatedProcess)

Event for allocating memory using malloc/calloc

Parameters

location	location which creates this event
timestamp	timestamp for this event
addrAllocated	allocated address, should be converted to void∗
bytesAllocated	number of bytes allocated
substrateData	Internal substrates may register data for this address. They should use their substrate id to access their specific item. Substrate plugins should NOT access this variable.
bytesAllocated↩Metric	The total size of the metric. E.g., all memory regions tracked with the memory adapters count into a specific metric
bytesAllocated↩Process	total number of bytes allocated in the process

M.5.3.65   typedef void(∗ SCOREP_Substrates_TrackFreeCb) (struct SCOREP_Location ∗location, uint64_t timestamp, uint64_t addrFreed, size_t bytesFreed, void ∗substrateData[ ], size_t bytesAllocatedMetric, size_t bytesAllocatedProcess)

Event for freeing memory using free

Parameters

location	location which creates this event
timestamp	timestamp for this event
addrFreed	address passed to free
bytesFreed	number of bytes freed
substrateData	Internal substrates get previously registered data for addrFreed. They should use their substrate id to access their specific item. Substrate plugins should NOT access this variable.
bytesAllocated↩Metric	The total size of the metric. E.g., all memory regions tracked with the memory adapters count into a specific metric
bytesAllocated↩Process	total number of bytes allocated in the process

M.5.3.66 typedef void(∗ SCOREP_Substrates_TrackReallocCb) (struct SCOREP_Location ∗location, uint64_t timestamp, uint64_t oldAddr, size_t oldBytesAllocated, void ∗oldSubstrateData[ ], uint64_t newAddr, size_t newBytesAllocated, void ∗newSubstrateData[ ], size_t bytesAllocatedMetric, size_t bytesAllocatedProcess)

Event for allocating/freeing memory using realloc

Parameters

location	location which creates this event
timestamp	timestamp for this event
oldAddr	address passed to realloc
oldBytes↩Allocated	size allocated to oldAddr before calling realloc
oldSubstrate↩Data	Internal substrates get their previously registered data for oldAddr. They should use their substrate id to access their specific item. Substrate plugins should NOT access this variable.
newAddr	address gained from realloc
newBytes↩Allocated	size of object at newAddr after realloc
newSubstrate↩Data	Internal substrates can register data for newAddr. They should use their substrate id to access their specific item. Substrate plugins should NOT access this variable.
bytesAllocated↩Metric	The total size of the metric. E.g., all memory regions tracked with the memory adapters count into a specific metric
bytesAllocated↩Process	total number of bytes allocated in the process

M.5.3.67 typedef void(∗ SCOREP_Substrates_TriggerCounterInt64Cb) (struct SCOREP_Location ∗location, uint64_t timestamp, SCOREP_SamplingSetHandle counterHandle, int64_t value)

Trigger a counter, which represents more or less a metric

See also

also SCOREP_User_TriggerMetricInt64 SCOREP_User_TriggerMetricUint64 SCOREP_User_Trigger↩MetricDouble

Parameters

location	location which creates this event
timestamp	timestamp for this event
counterHandle	previously defined counter handle
value	value of the counter when triggered. The datatype depends on the called function

M.5.3.68   typedef void(∗ SCOREP_Substrates_TriggerParameterInt64Cb) (struct SCOREP_Location ∗location, uint64_t timestamp, SCOREP_ParameterHandle parameterHandle, int64_t value)

Trigger a user defined parameter with a specific value

See also

>   also SCOREP_User_ParameterInt64 SCOREP_User_ParameterUint64

Parameters

location	location which creates this event
timestamp	timestamp for this event
parameter↩ Handle	previously defined parameter handle
value	value of the parameter when triggered. The datatype depends on the called function

M.5.3.69   typedef void(∗ SCOREP_Substrates_TriggerParameterStringCb) (struct SCOREP_Location ∗location, uint64_t timestamp, SCOREP_ParameterHandle parameterHandle, SCOREP_StringHandle string_handle)

Trigger a user defined parameter with a specific value

See also

>   also SCOREP_User_ParameterString

Parameters

location	location which creates this event
timestamp	timestamp for this event
parameter↩ Handle	previously defined parameter handle
string_handle	previously defined string when the parameter is triggered.

M.5.3.70   typedef void(∗ SCOREP_Substrates_WriteMetricsCb) (struct SCOREP_Location ∗location, uint64_t timestamp, SCOREP_SamplingSetHandle samplingSet, const uint64_t ∗metricValues)

Used to record metrics. Provided by substrates as arguments to SCOREP_Metric_WriteStrictlySynchronous↩ Metrics(), SCOREP_Metric_WriteSynchronousMetrics(), and SCOREP_Metric_WriteAsynchronousMetrics(). These functions are supposed to be called during following events: [CallingContext]Enter, [CallingContext]Exit, Sample. Will also be used for writing post mortem asynchronous metrics during SCOREP_EVENT_WRITE_PO↩ ST_MORTEM_METRICS.

Parameters

location	A pointer to the thread location data of the thread that executed the metric event.
timestamp	The timestamp, when the metric event occurred.
samplingSet	The sampling set with metrics
metricValues	Array of the metric values.

M.5.4   Enumeration Type Documentation

M.5.4.1   enum SCOREP_Substrates_EventType

Substrate events. Lists every event that is going to be used by the substrate mechanism. More details can be found in the respective functions. To maintain API stability, new events need to be added at the end of the enum.

Enumerator

> ***SCOREP_EVENT_ENABLE_RECORDING***   enable recording of events, see SCOREP_Substrates_Enable↩RecordingCb()

> ***SCOREP_EVENT_DISABLE_RECORDING***   disable recording of events, see SCOREP_Substrates_↩DisableRecordingCb()

> ***SCOREP_EVENT_ON_TRACING_BUFFER_FLUSH_BEGIN***   start flushing trace buffer to disk, see SCOR↩EP_Substrates_OnTracingBufferFlushBeginCb()

> ***SCOREP_EVENT_ON_TRACING_BUFFER_FLUSH_END***   end flushing trace buffer to disk, see SCOREP↩_Substrates_OnTracingBufferFlushEndCb()

> ***SCOREP_EVENT_ENTER_REGION***   enter an instrumented region, see SCOREP_Substrates_Enter↩RegionCb()

> ***SCOREP_EVENT_EXIT_REGION***   exit an instrumented region, see SCOREP_Substrates_ExitRegionCb()

> ***SCOREP_EVENT_SAMPLE***   record a calling context from sampling, see SCOREP_Substrates_SampleCb()

> ***SCOREP_EVENT_CALLING_CONTEXT_ENTER***   enter an instrumented region with calling context information, replaces SCOREP_EVENT_ENTER_REGION when unwinding is enabled, see SCOREP_↩Substrates_CallingContextEnterCb()

> ***SCOREP_EVENT_CALLING_CONTEXT_EXIT***   exit an instrumented region with calling context information, replaces SCOREP_EVENT_EXIT_REGION when unwinding is enabled, see SCOREP_Substrates_↩CallingContextExitCb()

> ***SCOREP_EVENT_ENTER_REWIND_REGION***   enter rewinding, see SCOREP_Substrates_EnterRewind↩RegionCb()

> ***SCOREP_EVENT_EXIT_REWIND_REGION***   exit rewinding, see SCOREP_Substrates_ExitRewindRegion↩Cb()

> ***SCOREP_EVENT_MPI_SEND***   MPI_Send, see SCOREP_Substrates_MpiSendCb()

> ***SCOREP_EVENT_MPI_RECV***   MPI_Recv, see SCOREP_Substrates_MpiRecvCb()

> ***SCOREP_EVENT_MPI_COLLECTIVE_BEGIN***   starts an MPI collective, see SCOREP_Substrates_Mpi↩CollectiveBeginCb()

> ***SCOREP_EVENT_MPI_COLLECTIVE_END***   ends an MPI collective, see SCOREP_Substrates_Mpi↩CollectiveEndCb()

> ***SCOREP_EVENT_MPI_ISEND_COMPLETE***   marks the completion of an MPI_Isend, see SCOREP_↩Substrates_MpiIsendCompleteCb()

> ***SCOREP_EVENT_MPI_IRECV_REQUEST***   marks the request for an MPI_Irecv, see SCOREP_Substrates↩_MpiIrecvRequestCb()

> ***SCOREP_EVENT_MPI_REQUEST_TESTED***   marks the test of an MPI request (e.g., in an MPI_Waitsome(...)), see SCOREP_Substrates_MpiRequestTestedCb()

SCOREP_EVENT_MPI_REQUEST_CANCELLED marks the cancellation of an MPI request (e.g., an MP↩
I_Test_cancelled(...) call that returned true in its second parameter), see SCOREP_Substrates_Mpi↩
RequestCancelledCb()

SCOREP_EVENT_MPI_ISEND marks the start of an MPI_ISend, see SCOREP_Substrates_MpiIsendCb()

SCOREP_EVENT_MPI_IRECV marks the start of an MPI_IRecv, see SCOREP_Substrates_MpiIrecvCb()

SCOREP_EVENT_RMA_WIN_CREATE marks the creation of an RMA window (used by cuda, opencl, and
shmem), see SCOREP_Substrates_RmaWinCreateCb()

SCOREP_EVENT_RMA_WIN_DESTROY marks the destruction of an RMA window (used by cuda, opencl,
and shmem), see SCOREP_Substrates_RmaWinDestroyCb()

SCOREP_EVENT_RMA_COLLECTIVE_BEGIN marks the start of an RMA collective (used by shmem), see
SCOREP_Substrates_RmaCollectiveBeginCb()

SCOREP_EVENT_RMA_COLLECTIVE_END marks the start of an RMA collective (used by shmem), see
SCOREP_Substrates_RmaCollectiveEndCb()

SCOREP_EVENT_RMA_TRY_LOCK marks an RMA trylock (used by shmem), see SCOREP_Substrates↩
_RmaTryLockCb()

SCOREP_EVENT_RMA_ACQUIRE_LOCK marks the acquisition of an RMA lock (used by shmem), see S↩
COREP_Substrates_RmaAcquireLockCb()

SCOREP_EVENT_RMA_REQUEST_LOCK marks a request for an RMA lock (used by shmem), see SCO↩
REP_Substrates_RmaRequestLockCb()

SCOREP_EVENT_RMA_RELEASE_LOCK marks a release of an RMA lock (used by shmem), see SCO↩
REP_Substrates_RmaReleaseLockCb()

SCOREP_EVENT_RMA_SYNC marks a simple pairwise RMA synchronization, see SCOREP_Substrates↩
_RmaSyncCb()

SCOREP_EVENT_RMA_GROUP_SYNC marks an RMA synchronization of a sub-group of locations on a
given window, see SCOREP_Substrates_RmaGroupSyncCb()

SCOREP_EVENT_RMA_PUT marks a put operation to an RMA memory (used by cuda, opencl, and shmem),
see SCOREP_Substrates_RmaPutCb()

SCOREP_EVENT_RMA_GET marks a get operation from an RMA memory (used by cuda, opencl, and
shmem), see SCOREP_Substrates_RmaGetCb()

SCOREP_EVENT_RMA_ATOMIC marks an atomic RMA operation (used by shmem), see SCOREP_↩
Substrates_RmaAtomicCb()

SCOREP_EVENT_RMA_WAIT_CHANGE marks a blocks until a remote operation modifies a given RMA
memory field (used by shmem), see SCOREP_Substrates_RmaWaitChangeCb()

SCOREP_EVENT_RMA_OP_COMPLETE_BLOCKING marks completion of a blocking RMA operation
(used by cuda, opencl, and shmem), see SCOREP_Substrates_RmaOpCompleteBlockingCb()

SCOREP_EVENT_RMA_OP_COMPLETE_NON_BLOCKING marks completion of a non-blocking RMA op-
eration, see SCOREP_Substrates_RmaOpCompleteNonBlockingCb()

SCOREP_EVENT_RMA_OP_TEST marks a test for completion of a non-blocking RMA operation, see SC↩
OREP_Substrates_RmaOpTestCb()

SCOREP_EVENT_RMA_OP_COMPLETE_REMOTE marks a remote completion point, see SCOREP_↩
Substrates_RmaOpCompleteRemoteCb()

SCOREP_EVENT_THREAD_ACQUIRE_LOCK marks when a thread acquires a lock (pthreads, explicit and
implicit OpenMP locks), see SCOREP_Substrates_ThreadAcquireLockCb()

SCOREP_EVENT_THREAD_RELEASE_LOCK marks when a thread releases a lock (pthreads, explicit and
implicit OpenMP locks), see SCOREP_Substrates_ThreadreleaseLockCb()

SCOREP_EVENT_TRIGGER_COUNTER_INT64 called when an int64 counter is triggered, see SCOREP↩
_Substrates_TriggerCounterInt64Cb()

SCOREP_EVENT_TRIGGER_COUNTER_UINT64 called when an uint64 counter is triggered, see SCOR↩
EP_Substrates_TriggerCounterUint64Cb()

SCOREP_EVENT_TRIGGER_COUNTER_DOUBLE called when an double counter is triggered, see SCO↩
REP_Substrates_TriggerCounterDoubleCb()

SCOREP_EVENT_TRIGGER_PARAMETER_INT64 called when an int64 parameter is triggered, called from user instrumentation, see SCOREP_Substrates_TriggerParameterInt64Cb()

SCOREP_EVENT_TRIGGER_PARAMETER_UINT64 called when an uint64 parameter is triggered, called from user instrumentation, see SCOREP_Substrates_TriggerParameterUint64Cb()

SCOREP_EVENT_TRIGGER_PARAMETER_STRING called when an string parameter is triggered, called from user instrumentation, see SCOREP_Substrates_TriggerParameterStringCb()

SCOREP_EVENT_THREAD_FORK_JOIN_FORK called before a fork-join based thread-parallel programming model (e.g., OpenMP) forks its threads logically, see SCOREP_Substrates_ThreadForkJoinFork←Cb()

SCOREP_EVENT_THREAD_FORK_JOIN_JOIN called after a fork-join based thread-parallel programming model (e.g., OpenMP) joins its threads logically, see SCOREP_Substrates_ThreadForkJoinJoinCb()

SCOREP_EVENT_THREAD_FORK_JOIN_TEAM_BEGIN begin of a parallel execution on a thread created by either SCOREP_ThreadForkJoin_Fork, is called by all created threads, see SCOREP_Substrates_←ThreadForkJoinTeamBeginCb()

SCOREP_EVENT_THREAD_FORK_JOIN_TEAM_END end of a parallel execution on a thread created by either SCOREP_ThreadForkJoin_Fork, is called by all created threads, see SCOREP_Substrates_←ThreadForkJoinTeamBeginCb()

SCOREP_EVENT_THREAD_FORK_JOIN_TASK_CREATE creation of a task in a fork-join based thread-parallel programming model (e.g., OpenMP), see SCOREP_Substrates_ThreadForkJoinTaskCreateCb()

SCOREP_EVENT_THREAD_FORK_JOIN_TASK_SWITCH switching of tasks in a fork-join based thread-parallel programming model (e.g., OpenMP), see SCOREP_Substrates_ThreadForkJoinTaskSwitchCb()

SCOREP_EVENT_THREAD_FORK_JOIN_TASK_BEGIN begin of a task in a fork-join based thread-parallel programming model (e.g., OpenMP), see SCOREP_Substrates_ThreadForkJoinTaskBeginCb()

SCOREP_EVENT_THREAD_FORK_JOIN_TASK_END end of a task in a fork-join based thread-parallel programming model (e.g., OpenMP), see SCOREP_Substrates_ThreadForkJoinTaskEndCb()

SCOREP_EVENT_THREAD_CREATE_WAIT_CREATE create a new thread in a create-wait based thread-parallel programming model (e.g., pthreads), called by parent, see SCOREP_Substrates_ThreadCreate←WaitCreateCb()

SCOREP_EVENT_THREAD_CREATE_WAIT_WAIT wait and join a thread in a create-wait based thread-parallel programming model (e.g., pthreads), usually called by parent, see SCOREP_Substrates_←ThreadCreateWaitWaitCb()

SCOREP_EVENT_THREAD_CREATE_WAIT_BEGIN begin a new thread in a create-wait based thread-parallel programming model (e.g., pthreads), called by new thread, see SCOREP_Substrates_Thread←CreateWaitBeginCb()

SCOREP_EVENT_THREAD_CREATE_WAIT_END end a thread in a create-wait based thread-parallel programming model (e.g., pthreads), see SCOREP_Substrates_ThreadCreateWaitEndCb()

SCOREP_EVENT_TRACK_ALLOC track malloc/calloc memory allocation, see SCOREP_Substrates_←TrackAllocCb()

SCOREP_EVENT_TRACK_REALLOC track realloc memory (de-)allocation, see SCOREP_Substrates_←TrackReallocCb()

SCOREP_EVENT_TRACK_FREE track realloc memory deallocation, see SCOREP_Substrates_Track←FreeCb()

SCOREP_EVENT_WRITE_POST_MORTEM_METRICS write post mortem metrics before unify, see SCO←REP_Substrates_WriteMetricsCb()

SCOREP_EVENT_PROGRAM_BEGIN begin of program and measurement, see SCOREP_Substrates_←ProgramBeginCb()

SCOREP_EVENT_PROGRAM_END end of program and measurement, see SCOREP_Substrates_←ProgramEndCb()

SCOREP_EVENT_IO_CREATE_HANDLE marks the creation of a new I/O handle, the handle gets active and can be used in subsequent I/O operations, see SCOREP_Substrates_IoCreateHandleCb

SCOREP_EVENT_IO_DESTROY_HANDLE marks the deletion of a, the I/O handle can't be used in subsequent I/O operations any longer, see SCOREP_Substrates_IoDestroyHandleCb

SCOREP_EVENT_IO_DUPLICATE_HANDLE marks the duplication of an already existing I/O handle, the new one gets active and can be used in subsequent I/O operations, see SCOREP_Substrates_Io↩DuplicateHandleCb

SCOREP_EVENT_IO_SEEK marks a seek operation, sets the file position indicator, see SCOREP_↩Substrates_IoSeekCb

SCOREP_EVENT_IO_CHANGE_STATUS_FLAGS called when status information of an I/O handle are changed, see SCOREP_Substrates_IoChangeStatusFlagsCb

SCOREP_EVENT_IO_DELETE_FILE marks the deletion of a file, see SCOREP_Substrates_IoDeleteFileCb

SCOREP_EVENT_IO_OPERATION_BEGIN marks the begin of an I/O operation, see SCOREP_↩Substrates_IoOperationBeginCb

SCOREP_EVENT_IO_OPERATION_ISSUED called when an asynchronous I/O operation is submitted to the I/O subssystem, see SCOREP_Substrates_IoOperationIssuedCb

SCOREP_EVENT_IO_OPERATION_TEST called when an asynchronous I/O operation is tested for completion but is not finished yet, see SCOREP_Substrates_IoOperationTestCb

SCOREP_EVENT_IO_OPERATION_COMPLETE called when an asynchronous I/O operation is successfully tested for completion, see SCOREP_Substrates_IoOperationCompleteCb

SCOREP_EVENT_IO_OPERATION_CANCELLED called when an asynchronous I/O operation is aborted, see SCOREP_Substrates_IoOperationCancelledCb

SCOREP_EVENT_IO_ACQUIRE_LOCK marks the acquisition of an I/O lock, see SCOREP_Substrates_↩IoAcquireLockCb

SCOREP_EVENT_IO_TRY_LOCK called when , see SCOREP_Substrates_IoTryLockCb

SCOREP_EVENT_IO_RELEASE_LOCK marks the release of an I/O lock, see SCOREP_Substrates_Io↩ReleaseLockCb

SCOREP_EVENT_MPI_NON_BLOCKING_COLLECTIVE_REQUEST marks the initiation of a non-blocking MPI collective operation, see SCOREP_Substrates_MpiNonBlockingCollectiveRequestCb()

SCOREP_EVENT_MPI_NON_BLOCKING_COLLECTIVE_COMPLETE marks the completion of a nonblocking MPI collective operation, see SCOREP_Substrates_MpiNonBlockingCollectiveCompleteCb()

SCOREP_EVENT_COMM_CREATE marks the creation of a communicator, providing the respective handle

SCOREP_EVENT_COMM_DESTROY marks the destruction of a communicator, providing the respective handle

SCOREP_SUBSTRATES_NUM_EVENTS Non-ABI, marks the end of the currently supported events and can change with different versions of Score-P (increases with increasing Score-P version)

M.5.4.2 enum **SCOREP_Substrates_Mode**

Substrates need to provide two sets of callbacks for the modes SCOREP_SUBSTATES_RECORDING_ENABLED and SCOREP_SUBSTRATES_RECORDING_DISABLED. This enum is used as an array index.

Enumerator

SCOREP_SUBSTRATES_RECORDING_ENABLED The recording of events is enabled (default)

SCOREP_SUBSTRATES_RECORDING_DISABLED The recording of events is disabled

SCOREP_SUBSTRATES_NUM_MODES Non-ABI

M.6 SCOREP_SubstratePlugins.h File Reference

Description of the substrate plugin header. For information on how to use substrate plugins, please refer to section 'Substrate Plugins'.

```
#include <stdlib.h>
#include <stdio.h>
#include <stddef.h>
#include <scorep/SCOREP_PublicTypes.h>
#include <scorep/SCOREP_PublicHandles.h>
#include <scorep/SCOREP_SubstrateEvents.h>
```

Data Structures

- struct SCOREP_SubstratePluginCallbacks
- struct SCOREP_SubstratePluginInfo

Macros

- #define SCOREP_SUBSTRATE_PLUGIN_ENTRY(_name)
- #define SCOREP_SUBSTRATE_PLUGIN_UNDEFINED_MANAGEMENT_FUNCTIONS 99
- #define SCOREP_SUBSTRATE_PLUGIN_VERSION 3

M.6.1 Detailed Description

Description of the substrate plugin header. For information on how to use substrate plugins, please refer to section 'Substrate Plugins'.

M.6.2 Macro Definition Documentation

M.6.2.1 #define SCOREP_SUBSTRATE_PLUGIN_ENTRY(_name)

Value:

```
EXTERN SCOREP_SubstratePluginInfo \
    SCOREP_SubstratePlugin_ ## _name ## _get_info( void )
```

Macro used for implementation of the 'get_info' function

M.6.2.2 #define SCOREP_SUBSTRATE_PLUGIN_UNDEFINED_MANAGEMENT_FUNCTIONS 99

This should be reduced by 1 for each new function added to SCOREP_SubstratePluginInfo

M.6.3   Advice for developers

The developer of a substrate plugin should provide a README file which explains how to compile, install and use the plugin. In particular, the supported substrates should be described in the README file.

Each substrate plugin has to include `SCOREP_SubstratePlugins.h` and implement a 'get_info' function. Therefore, use the SCOREP_SUBSTRATE_PLUGIN_ENTRY macro and provide the name of the plugin library as the argument. The plugin library must be called libscorep_substrate_<libraryname>.so For example, the example substrate plugin libscorep_substrate_example.so should use SCOREP_SUBSTRATE_PLUGIN_ENTRY(example). Substrate plugins that implement event functions should also include `SCOREP_SubstrateEvents.h` Plugin writers should also refer to `SCOREP_PublicHandles.h` and `SCOREP_PublicTypes.h` to handle SCOR↩ EP handles given in event functions.

M.6.4   Functions

See each function for details. All functions except init are optional!

init

Check requirements and initialize the plugin.

assign_id

The plugin gets an id that can be used later to store location specific data.

init_mpp

If an MPP paradigm is used, it will be initialized when this call occurs. If no MPP paradigm is used, this function will be called as well.

finalize

Finalization of Score-P.

create_location

Create a new location (e.g., when a thread is created)

delete_location

Delete an existing location

activate_cpu_location

Activate a CPU location to write events. Called, for example, after create_location on CPU locations.

deactivate_cpu_location

Deactivate a CPU location. Called, for example, before delete_location.

pre_unify

Called before the unify step, after the measurement.

write_data

Called after the measurement when writing data.

core_task_create

Create a task (e.g., an OpenMP task)

core_task_complete

Complete a task (e.g., an OpenMP task)

new_definition_handle

Called when a handle is defined, which could define, for example a region or a metric. Plugins can use callbacks to get meta data for this handle.

get_event_functions

Called twice with different modes. Get a list of events that shall be passed to the plugin.

set_callbacks

Called before get_event_functions. Set a list of callbacks so that the plugin can get meta data for handles.

undeclared

MUST be set to zero. Added for extendability.

M.6.5 Mandatory variable

plugin_version

Must be set to SCOREP_SUBSTRATE_PLUGIN_VERSIONCurrent version of Score-P substrate plugin interface

M.7 SCOREP_User.h File Reference

This file contains the interface for the manual user instrumentation.

```
#include <scorep/SCOREP_User_Variables.h>
#include <scorep/SCOREP_User_Functions.h>
```

Macros

Macros for region instrumentation

- #define SCOREP_USER_FUNC_DEFINE()
- #define SCOREP_USER_REGION_DEFINE(handle) static SCOREP_User_RegionHandle handle = S↩
  COREP_USER_INVALID_REGION;
- #define SCOREP_USER_REGION_ENTER(handle) SCOREP_User_RegionEnter(handle);
- #define SCOREP_USER_REGION_BEGIN(handle, name, type)
- #define SCOREP_USER_REGION_INIT(handle, name, type)
- #define SCOREP_USER_REGION_END(handle) SCOREP_User_RegionEnd(handle);
- #define SCOREP_USER_FUNC_BEGIN()
- #define SCOREP_USER_FUNC_END() SCOREP_User_RegionEnd(scorep_user_func_handle);
- #define SCOREP_USER_GLOBAL_REGION_DEFINE(handle) SCOREP_User_RegionHandle handle =
  SCOREP_USER_INVALID_REGION;
- #define SCOREP_USER_GLOBAL_REGION_EXTERNAL(handle) extern SCOREP_User_Region↩
  Handle handle;

Macros for parameter instrumentation

- #define SCOREP_USER_PARAMETER_INT64(name, value)
- #define SCOREP_USER_PARAMETER_UINT64(name, value)
- #define SCOREP_USER_PARAMETER_STRING(name, value)

Macros to provide user metrics

- #define SCOREP_USER_METRIC_LOCAL(metricHandle)
- #define SCOREP_USER_METRIC_GLOBAL(metricHandle)
- #define SCOREP_USER_METRIC_EXTERNAL(metricHandle) extern SCOREP_SamplingSetHandle
  metricHandle;
- #define SCOREP_USER_METRIC_INIT(metricHandle, name, unit, type, context) SCOREP_User_Init↩
  Metric(&metricHandle, name, unit, type, context);
- #define SCOREP_USER_METRIC_INT64(metricHandle, value)
- #define SCOREP_USER_METRIC_UINT64(metricHandle, value)
- #define SCOREP_USER_METRIC_DOUBLE(metricHandle, value)

Macros for creation of user topologies

- #define SCOREP_USER_CARTESIAN_TOPOLOGY_CREATE(userTopology, name, nDims)
- #define SCOREP_USER_CARTESIAN_TOPOLOGY_ADD_DIM(userTopology, size, periodic, name) S↩
  COREP_User_CartTopologyAddDim(userTopology, size, periodic, name);
- #define SCOREP_USER_CARTESIAN_TOPOLOGY_INIT(userTopology) SCOREP_User_Cart↩
  TopologyInit(userTopology);
- #define SCOREP_USER_CARTESIAN_TOPOLOGY_SET_COORDS(userTopology, nDims, ...) SCOR↩
  EP_User_CartTopologySetCoords(userTopology, nDims, __VA_ARGS__);
- #define SCOREP_USER_CARTESIAN_TOPOLOGY_DEFINE(userTopology)

C++ specific macros for region instrumentation

- #define SCOREP_USER_REGION(name, type)

Macros for measurement control

- #define SCOREP_RECORDING_ON() SCOREP_User_EnableRecording();
- #define SCOREP_RECORDING_OFF() SCOREP_User_DisableRecording();
- #define SCOREP_RECORDING_IS_ON() SCOREP_User_RecordingEnabled()

M.7.1 Detailed Description

This file contains the interface for the manual user instrumentation.

M.8 SCOREP_User_Types.h File Reference

This file contains type definitions for manual user instrumentation.

```
#include <scorep/SCOREP_PublicTypes.h>
```

Macros

- #define SCOREP_USER_INVALID_PARAMETER -1
- #define SCOREP_USER_INVALID_REGION NULL
- #define SCOREP_USER_INVALID_TOPOLOGY -1

Region types

- #define SCOREP_USER_REGION_TYPE_COMMON 0
- #define SCOREP_USER_REGION_TYPE_FUNCTION 1
- #define SCOREP_USER_REGION_TYPE_LOOP 2
- #define SCOREP_USER_REGION_TYPE_DYNAMIC 4
- #define SCOREP_USER_REGION_TYPE_PHASE 8

Metric types

- #define SCOREP_USER_METRIC_TYPE_INT64 0
- #define SCOREP_USER_METRIC_TYPE_UINT64 1
- #define SCOREP_USER_METRIC_TYPE_DOUBLE 2

Metric contexts

- #define SCOREP_USER_METRIC_CONTEXT_GLOBAL 0
- #define SCOREP_USER_METRIC_CONTEXT_CALLPATH 1

Typedefs

- typedef struct SCOREP_User_Topology ∗ SCOREP_User_CartesianTopologyHandle
- typedef uint32_t SCOREP_User_MetricType
- typedef uint64_t SCOREP_User_ParameterHandle
- typedef struct SCOREP_User_Region ∗ SCOREP_User_RegionHandle
- typedef uint32_t SCOREP_User_RegionType

M.8.1 Detailed Description

This file contains type definitions for manual user instrumentation.

M.8.2   Macro Definition Documentation

M.8.2.1   #define SCOREP_USER_INVALID_PARAMETER -1

Marks an parameter handle as invalid or uninitialized

M.8.2.2   #define SCOREP_USER_INVALID_REGION NULL

Value for uninitialized or invalid region handles

M.8.2.3   #define SCOREP_USER_INVALID_TOPOLOGY -1

Marks a topology handle as invalid or uninitialized

M.8.3   Typedef Documentation

M.8.3.1   typedef struct SCOREP_User_Topology∗ SCOREP_User_CartesianTopologyHandle

Type for topology handles in the user adapter

M.8.3.2   typedef uint32_t SCOREP_User_MetricType

Type for the user metric type

M.8.3.3   typedef uint64_t SCOREP_User_ParameterHandle

Type for parameter handles

M.8.3.4   typedef struct SCOREP_User_Region∗ SCOREP_User_RegionHandle

Type for region handles in the user adapter.

M.8.3.5   typedef uint32_t SCOREP_User_RegionType

Type for the region type

Index