# StarPU profiling interface

Olivier Aumage, Camille Coti

# StarPU profiling interface

Olivier Aumage[1] and Camille Coti[2]

[1]Inria Research Center at the University of Bordeaux and LaBRI, France
[2]Université du Québec à Montréal, Canada

November 2022

**Abstract**

This document introduces the programming interface exposed by the StarPU task-based parallel runtime system for performance monitoring tools to collect performance-related metrics on the execution of StarPU applications. It defines the entry point, the callback methods, the performance counters available. It also presents the usage of the mechanisms.

## Contents

# 1 Introduction to the StarPU callback interface

The StarPU callback interface can be used by third-party libraries for profile and trace data collection, to conduct performance analysis. It provides a set of callbacks that are triggered at specific points of the execution, as described in this document. Data is passed through these callbacks for these external libraries to collect information about the execution.

First, such a library must be loaded, as described in section 2. Second, the library initializes the callbacks by registering the function it provides onto StarPU's callback points, as described in section 3.

StarPU provides two categories of callbacks: the first one gives information on the execution of the StarPU tasks on the available resources, as described in section 4; the second one gives information on the internal state of the scheduler, as described in section 5.

# 2 Loading a tooling library

The library must be passed using the `STARPU_PROF_TOOL` environment variable or preloaded using `LD_PRELOAD`, upon the launch of the compiled StarPU program. For instance:

```
STARPU_PROF_TOOL=./libstarpu_myprof.so  ./myprogram
```

# 3 Library initialization

The library *must* implement a function `starpu_prof_tool_library_register`. This function registers callback routines using the `register` and `unregister` functions, whose pointers are passed as parameters. Its signature is the following one:

```
typedef void (*starpu_prof_tool_entry_func)(starpu_prof_tool_entry_register_func reg,
                                            starpu_prof_tool_entry_register_func unreg);
```

For example:

```
void starpu_prof_tool_library_register(starpu_prof_tool_entry_register_func reg,
                                       starpu_prof_tool_entry_register_func unreg)
{
        enum  starpu_prof_tool_command info = 0;
        reg(starpu_prof_tool_event_driver_init, &driver_init_cb, info);
        reg(starpu_prof_tool_event_driver_init_start, &driver_init_start_cb, info);
        /* ... */
}
```

The signature of these registration and unregistration functions is:

```
typedef void
        (*starpu_prof_tool_entry_register_func)(enum starpu_prof_tool_event event_type,
                                                starpu_prof_tool_cb_func cb,
                                                enum starpu_prof_tool_command info);
```

# 4 Task performance callbacks

This section presents the task performance callbacks that can be used for performance analysis of the *application*. The performance counters used to monitor the performance of StarPU and, in particular, retrieve information from the scheduler, are defined in section 5.

## 4.1 Callbacks signature

Callback functions have the following prototype:

```
typedef void (*starpu_prof_tool_cb_func)(struct starpu_prof_tool_info*,
                                         union starpu_prof_tool_event_info*,
                                         struct starpu_prof_tool_api_info*);
```

For instance:

```
void myfunction_cb( starpu_prof_tool_info* prof_info,
                    starpu_event_tool_info* event_info,
                    starpu_api_tool_info* api_info );
```

The arguments contain information that are passed to the callback (see section 4.2).

## 4.2 Data structures

### 4.2.1 Profiling information

This data structure provides the main information about the event concerned by the callback.

```
struct starpu_prof_tool_info
{
        struct starpu_conf *conf;
        enum starpu_prof_tool_event event_type;
        unsigned int starpu_version[3];
        int thread_id;
```

```
        int worker_id;

        int device_number;
        enum starpu_prof_tool_driver_type driver_type;

        unsigned memnode;
        unsigned bytes_to_transfer;
        unsigned bytes_transferred;

        void* fun_ptr;
}
```

- `conf`: a pointer to the current internal configuration;
- `event_type`: the type of the event that triggered this callback, see section 4.3;
- `starpu_version`: the current version of starpu (major, minor, release) and can be used to make a tool; compatible across versions taking into account API changes;
- `thread_id`: id of the current thread;
- `worker_id`: id of the current worker;
- `device_number`: number of the device related to the event that triggered this callback;
- `device_type`: type of device used by the event that triggered this callback, see section ;
- `memnode`: memory node, used in data transfers;
- `bytes_to_transfer`: how many bytes are being transferred by this memory movement, used in data transfers;
- `bytes_transfered`: how many bytes have been transferred by this memory movement, used in data transfers;
- `fun_ptr`: function pointer that tells us which function is about to be executed. It has to be resolved in the profiling tool.

Please note that not all the fields are relevant for all events. For instance, `memnode` is used for data transfers but not for executions. Likewise, `fun_ptr` is used for executions but not for data transfers.

For example, a callback function that would print the name of the event and the worker id would be:

```
void myfunction_cb( starpu_prof_tool_info* prof_info,
                    starpu_event_tool_info* event_info,
                    starpu_api_tool_info* api_info )
{
  if( prof_info != NULL ){
    printf("%d on worker %d\n", prof_info->event_type, prof_info->worker_id);
  }
}
```

#### 4.2.2 Driver types

This data structure provides information on how the event concerned by the callback is being executed.

```
  enum starpu_prof_tool_driver_type
  {
    starpu_prof_tool_driver_cpu,
    starpu_prof_tool_driver_gpu
  };
```

For the moment, we define two types of events: on a GPU and on a CPU.

#### 4.2.3 Event info

This data structure provides information on the event itself. It can be used to determine which event was triggered when the same function is registered for all the callbacks.

```
  union starpu_prof_tool_event_info
{
        enum starpu_prof_tool_event event_type;
};
```

#### 4.2.4 API info

This data structure is empty for the moment.

```
  struct starpu_prof_tool_api_info
  {
  };
```

### 4.3 List of events

The list of events for which StarPU provides callbacks is given by the following `enum`:

```
enum starpu_prof_tool_event
{
  starpu_prof_tool_event_none = 0,
  starpu_prof_tool_event_init,
  starpu_prof_tool_event_terminate,
  starpu_prof_tool_event_init_begin,
  starpu_prof_tool_event_init_end,
  starpu_prof_tool_event_driver_init,
  starpu_prof_tool_event_driver_deinit,
  starpu_prof_tool_event_driver_init_start,
  starpu_prof_tool_event_driver_init_end,
  starpu_prof_tool_event_start_cpu_exec,
  starpu_prof_tool_event_end_cpu_exec,
  starpu_prof_tool_event_start_gpu_exec,
  starpu_prof_tool_event_end_gpu_exec,
  starpu_prof_tool_event_start_transfer,
  starpu_prof_tool_event_end_transfer,
  starpu_prof_tool_event_user_start,
```

```
  starpu_prof_tool_event_user_end
};
```

#### 4.3.1 Initialization and shutdown

When the StarPU runtime environment is initialized, it triggers the callback corresponding to the event:

```
starpu_prof_tool_event_init
```

When it is shut down, it triggers the callback corresponding to the event:

```
starpu_prof_tool_event_terminate
```

The initialization itself begins and ends with the following events, respectively:

```
starpu_prof_tool_event_init_begin
starpu_prof_tool_event_init_end
```

The driver, one for every computation resource, begins and ends with the following events, respectively:

```
starpu_prof_tool_event_driver_init
starpu_prof_tool_event_driver_deinit
```

#### 4.3.2 Tasks

Tasks start and end on CPU resources and trigger the following callbacks:

```
starpu_prof_tool_event_start_cpu_exec
starpu_prof_tool_event_end_cpu_exec
```

They start and end on GPU resources and trigger the following callbacks:

```
starpu_prof_tool_event_start_gpu_exec
starpu_prof_tool_event_end_gpu_exec
```

#### 4.3.3 Memory transfers

The active part of memory transfers (*i.e.*, when StarPU is in memory transfer-related routines) start and end triggering the following callbacks, respectively:

```
starpu_prof_tool_event_start_transfer
starpu_prof_tool_event_end_transfer
```

#### 4.3.4 Other callbacks

Users can define their own callbacks:

```
starpu_prof_tool_event_user_start
starpu_prof_tool_event_user_end
```

In addition, the following callback is defined as a dummy event and always set to 0:

```
starpu_prof_tool_event_none
```

# 5 Scheduler performance counters

This chapter presents the StarPU performance monitoring framework. It summarizes the objectives of the framework. It then introduces the entities involved in the framework. It presents the API of the framework, as well as some implementation details. It exposes the typical sequence of operations to plug an external tool to monitor a performance counter of StarPU.

## 5.1 Objectives

The objectives of this framework are to let external tools interface with StarPU to collect various performance metrics at runtime, in a generic, safe, extensible way. For that, it enables such tools to discover the available performance metrics in a particular StarPU build as well as the type of each performance counter value. It lets these tools build sets of performance counters to monitor, and then register listener callbacks to collect the measurement samples of these sets of performance counters at runtime.

## 5.2 Entities

The performance monitoring framework is built on a series of concepts and items, organized in a consistent way. The corresponding C language objects should be considered opaque by external tools, and should only be manipulated through proper function calls and accessors.

### 5.2.1 Performance Counter

The performance counter entity is the fundamental object of the framework, representing one piece of performance metrics, such as for instance the total number of tasks submitted so far, that is exported by StarPU and can be collected through the framework at runtime. A performance counter has a type and belongs to a scope. A performance counter is designated by a unique name and unique ID integer.

### 5.2.2 Performance Counter Type

A performance counter has a type. A type is designated by a unique name and unique ID number. Currently supported types include:

| Type Name | Type | Definition |
|---|---|---|
| `int32` | 32-bit | signed integers |
| `int64` | 64-bit | signed integers |
| `float` | 32-bit | single-precision floating point |
| `double` | 64-bit | double-precision floating point |

### 5.2.3 Performance Counter Scope

A performance counter belongs to a scope. The scope of a counter defines the context considered for computing the corresponding performance counter. A scope is designated with a unique name and unique ID number. Currently defined scopes include:

| Scope Name | Scope Definition |
|---|---|
| `global` | Counter is global to the StarPU instance |
| `per_worker` | Counter is within the scope of a thread worker |
| `per_codelet` | Counter is within the scope of a task codelet |

### 5.2.4 Performance Counter Set

A performance counter set is a subset of the performance counters belonging to the same scope. Each counter of the scope can be in the enabled or disabled state in a performance counter set. A performance counter set enables a performance monitoring tool to indicate the set of counters to be collected for a particular listener callback.

### 5.2.5 Performance Counter Sample

A performance counter sample corresponds to one sample of collected measurement values of a performance counter set. Only the values corresponding to enabled counters in the sample's counter set should be observed by the listener callback. Whether the sample contains valid values for counters disabled in the set is unspecified.

### 5.2.6 Performance Counter Listener

A performance counter listener is a callback function registered by some external tool to monitor a set of performance counters in a particular scope. It is called each time a new performance counter sample is ready to be observed. The sample object should not be accessed outside of the callback.

## 5.3 Application Programming Interface

The API of the performance monitoring framework is defined in the `starpu_perf_monitoring.h` public header file of StarPU. This header file is automatically included with `starpu.h`. An example of use of the routines is given in Section 3.6.

### 5.3.1 Scope Related Routines

| Function Name | Function Definition |
|---|---|
| `starpu_perf_counter_scope_name_to_id` | Translate scope name constant string to scope id |
| `starpu_perf_counter_scope_id_to_name` | Translate scope id to scope name constant string |

### 5.3.2 Type Related Routines

| Function Name | Function Definition |
|---|---|
| `starpu_perf_counter_type_name_to_ib` | Translate type name constant string to type id |
| `starpu_perf_counter_type_id_to_namb` | Translate type id to type name constant string |

### 5.3.3 Counter Related Routines

| Function Name | Function Definition |
|---|---|
| `starpu_perf_counter_nb` | Return the number of performance counters for the given scope |
| `starpu_perf_counter_name_to_id` | Translate a performance counter name to its id |
| `starpu_perf_counter_nth_to_id` | Translate a performance counter rank in its scope to its counter id |
| `starpu_perf_counter_id_to_name` | Translate a counter id to its name constant string |
| `starpu_perf_counter_get_type_id` | Return the counter's type id |
| `starpu_perf_counter_get_help_string` | Return the counter's help string |
| `starpu_perf_counter_list_avail` | Display the list of counters defined in the given scope |
| `starpu_perf_counter_list_all_avail` | Display the list of counters defined in all scopes |

### 5.3.4 Counter Set Related Routines

| Function Name | Function Definition |
|---|---|
| `starpu_perf_counter_set_alloc` | Allocate a new performance counter set |
| `starpu_perf_counter_set_free` | Free a performance counter set |
| `starpu_perf_counter_set_enable_id` | Enable a given counter in the set |
| `starpu_perf_counter_set_disable_id` | Disable a given counter in the set |

#### 5.3.5 Listener Related Routines

| Function Name | Function Definition |
|---|---|
| `starpu_perf_counter_listener_init` | Initialize a new performance counter listener |
| `starpu_perf_counter_listener_exit` | End a performance counter listener |
| `starpu_perf_counter_set_global_listener` | Set a listener for the global scope |
| `starpu_perf_counter_set_per_worker_listener` | Set a listener for the per-worker scope on a given worker |
| `starpu_perf_counter_set_all_per_worker_listeners` | Set a common listener for all workers |
| `starpu_perf_counter_set_per_codelet_listener` | Set a per-codelet listener for a codelet |
| `starpu_perf_counter_unset_global_listener` | Unset the global listener |
| `starpu_perf_counter_unset_per_worker_listener` | Unset the per-worker listener |
| `starpu_perf_counter_unset_all_per_worker_listeners` | Unset all per-worker listeners |
| `starpu_perf_counter_unset_per_codelet_listener` | Unset a per-codelet listener |

#### 5.3.6 Sample Related Routines

| Function Name | Function Definition |
|---|---|
| `starpu_perf_counter_sample_get_int32_value` | Read an int32 counter value from a sample |
| `starpu_perf_counter_sample_get_int64_value` | Read an int64 counter value from a sample |
| `starpu_perf_counter_sample_get_float_value` | Read a float counter value from a sample |
| `starpu_perf_counter_sample_get_double_value` | Read a double counter value from a sample |

### 5.4 Implementation Details

#### 5.4.1 Performance Counter Registration

Each module of StarPU can export performance counters. In order to do so, modules that need to export some counters define a registration function that is called at StarPU initialization time. This function is responsible for calling the `_starpu_perf_counter_register()` function once for each counter it exports, to let the framework know about the list of counters managed by the module. It also registers performance sample updater callbacks for the module, one for each scope for which it exports counters.

#### 5.4.2 Performance Sample Updaters

The updater callback for a module and scope combination is internally called every time a sample for a set of performance counter must be updated. Thus, the updated callback is responsible for filling the sample's selected counters with the counter values found at the time of the call. Global updaters are currently called at task submission time, as well as any blocking tasks management function of the StarPU API, such as the `starpu_task_wait_for_all()`, which waits for the completion of all tasks submitted up to this point. Per-worker updaters are currently called at the level of StarPU's drivers, that is, the modules in charge of task execution of hardware-specific worker threads. The actual calls occur in-between the execution of tasks. Per-codelet updaters are currently called both at task submission time, and at the level of StarPU's drivers together with the per-worker updaters. A performance sample object is locked during the sample collection. The locking prevents the following issues:

- The listener of sample being changed during sample collection;
- The set of counters enabled for a sample being changed;
- Conflicting concurrent updates;
- Updates while the sample is being read by the listener.

The location of the updaters' calls is chosen to minimize the sequentialization effect of the locking, in order to limit the level of interference of the monitoring process. For Global updaters, the calls are performed only on the application thread(s) in charge of submitting tasks. Since, in most cases, only a single application thread submits tasks, the sequentialization effect is moderate. Per-worker updates are local to their worker, thus here again the sample lock is un-contented, unless the external monitoring tool frequently changes the set of enabled counters in the sample.

#### 5.4.3 Counter operations

In practice the sample updaters only take snapshots of the actual performance counters. The performance counters themselves are updated with ad-hoc procedures depending on each counter. Such procedures typically involve atomic operations. While operations such as atomic increments or decrements on integer values are readily available, this is not the case for more complex operations such as min/max for computing peak value counters (for instance in the global and per-codelet counters for peak number of submitted tasks and peak number of ready tasks waiting for execution), and this is also not the case for computations on floating point data (used for instance in computing cumulated execution time of tasks, either per worker or per codelet). The performance monitoring framework therefore supplies such missing routines, for the internal use of StarPU.

#### 5.4.4 Runtime checks

The performance monitoring framework features a comprehensive set of runtime checks to verify that both StarPU and some external tool do not access a performance counter with the wrong typed routines, to quickly detect situations of mismatch that can result from the evolution of multiple pieces of software at distinct paces. Moreover, no StarPU data structure is accessed directly either by the external code making use of the performance monitoring framework. The use of the C enum constants is optional; referring to values through constant strings is available when more robustness is desired. These runtime checks enable the framework to be extensible. Moreover, while the framework's counters currently are permanently compiled in, they could be made optional at compile time, for instance to suppress any overhead once the analysis and optimization process has been completed by the programmer. Thanks to the runtime discovery of available counters, the applicative code, or an intermediate layer such as skeleton layer acting on its behalf, would then be able to adapt to performance analysis builds versus optimized builds.

### 5.5 Exported Counters

#### 5.5.1 Global Scope

| Counter Name | Counter Definition |
|---|---|
| `starpu.task.g_total_submitted` | Total number of tasks submitted |
| `starpu.task.g_peak_submitted` | Maximum number of tasks submitted, waiting for dependencies resolution at any time |
| `starpu.task.g_peak_ready` | Maximum number of tasks ready for execution, waiting for an execution slot at any time |

#### 5.5.2 Per-worker Scope

| Counter Name | Counter Definition |
|---|---|
| `starpu.task.w_total_executed` | Total number of tasks executed on a given worker |
| `starpu.task.w_cumul_execution_time` | Cumulated execution time of tasks executed on a given worker |

### 5.5.3 Per-Codelet Scope

| Counter Name | Counter Definition |
|---|---|
| starpu.task.c_total_submitted | Total number of submitted tasks for a given codelet |
| starpu.task.c_peak_submitted | Maximum number of submitted tasks for a given codelet waiting for dependencies resolution at any time |
| starpu.task.c_peak_ready | Maximum number of ready tasks for a given codelet waiting for an execution slot at any time |
| starpu.task.c_total_executed | Total number of executed tasks for a given codelet |
| starpu.task.c_cumul_execution_time | Cumulated execution time of tasks for a given codelet |

# 6 Using the StarPU performance interface

## 6.1 In an external library

**Step 0: Initialize the library**

First, the library needs to be initialized. StarPU opens it, looks for the function `starpu_prof_tool_library_register`, and calls it. This is the place where callbacks must be registered; a registration and an un-registration function are provided when this function returns.

```
void starpu_prof_tool_library_register(starpu_prof_tool_entry_register_func reg,
                                       starpu_prof_tool_entry_register_func unreg)
{
        /* ... */
}
```

**Step 1: Register callbacks**

We can either use a different function for each callback, or the same function for each callback. If we follow the latter option, the even type that triggered the callback can be identified using the parameters passed to the callback.

```
void starpu_prof_tool_library_register(starpu_prof_tool_entry_register_func reg,
                                       starpu_prof_tool_entry_register_func unreg)
{
        enum  starpu_prof_tool_command info = 0;
        reg(starpu_prof_tool_event_driver_init, &driver_init_cb, info);
        reg(starpu_prof_tool_event_driver_init_start, &driver_init_start_cb, info);
        reg(starpu_prof_tool_event_driver_init_end &driver_init_end_cb, info);
        reg(starpu_prof_tool_event_driver_start_cpu_exec, &myFunction_cb, info);
        reg(starpu_prof_tool_event_driver_end_cpu_exec, &myFunction_cb, info);
        /* ... */
}
```

**Step 2: Use the information passed to the callbacks**

The callback functions are passed some information about the context in which the event has happened. The fields of the data structures are described in this document. For instance, we can get the event type and the worked it happened on; for callbacks related to the execution of tasks, we can get a pointer to the function executed by the task (which can be resolved, for instance, using a tool like BFD).

```
void myfunction_cb(struct starpu_prof_tool_info* prof_info,
                   union starpu_prof_tool_event_info* event_info,
                   struct starpu_prof_tool_api_info* api_info )
{
    switch(  prof_info->event_type ) {
    case starpu_prof_tool_event_start_cpu_exec:
        printf( "Start function %p on worker %d\n", prof_info->fun_ptr, prof_info->worker_id );
        break;
    case starpu_prof_tool_event_end_cpu_exec,
        printf( "End function %p on worker %d\n", prof_info->fun_ptr, prof_info->worker_id );
        break;
    }
}
```

**Step 3: Load the library at run-time**

The library can be loaded by StarPU using either the `LD_PRELOAD` or the `STARPU_PROF_TOOL` environment variable:

```
STARPU_PROF_TOOL=./libstarpu_myprof.so  ./myprogram
```

## 6.2 In a StarPU application

This section presents a typical sequence of operations to interface an external tool with some StarPU performance counters. In this example, the counters monitored are the per-worker total number of executed tasks (`starpu.task.w_total_executed`) and the tasks' cumulated execution time (`starpu.task.w_cumul_execution_time`).

**Step 0: Initialize StarPU** StarPU must first be initialized, by a call to `starpu_init()`, for performance counters to become available, since each module of StarPU registers the performance counters it exports during that initialization phase.

```
int ret = starpu_init(NULL);
```

**Step 1: Allocate a counter set**

A counter set has to be allocated on the per-worker scope. The per-worker scope id can be obtained by name, or with the pre-defined enum value `starpu_perf_counter_scope_per_worker`.

```
enum starpu_perf_counter_scope w_scope = starpu_perf_counter_scope_per_worker;
struct starpu_perf_counter_set *w_set = starpu_perf_counter_set_alloc(w_scope);
```

**Step 2: Get the counter IDs**

Each performance counter has a unique ID used to refer to it in subsequent calls to the performance monitoring framework.

```
int id_w_total_executed = starpu_perf_counter_name_to_id(w_scope,
                                                         "starpu.task.w_total_executed");
```

```
int id_w_cumul_execution_time = starpu_perf_counter_name_to_id(w_scope,
                                                    "starpu.task.w_cumul_execution_time");
```

**Step 3: Enable the counters in the counter set**

This step indicates which counters will be collected into performance monitoring samples for the listeners referring to this counter set.

```
starpu_perf_counter_set_enable_id(w_set, id_w_total_executed);
starpu_perf_counter_set_enable_id(w_set, id_w_cumul_execution_time);
```

**Step 4: Write a listener callback**

This callback will be triggered when a sample becomes available. Upon execution, it reads the values for the two counters from the sample and displays these values, for the sake of the example.

```
void w_listener_cb(struct starpu_perf_counter_listener *listener,
                   struct starpu_perf_counter_sample   *sample,
                   void *context)
{
  int32_t w_total_executed =
             starpu_perf_counter_sample_get_int32_value(sample, id_w_total_executed);

  double w_cumul_execution_time =
      starpu_perf_counter_sample_get_double_value(sample, id_w_cumul_execution_time);

  printf("worker[%d]: w_total_executed = %d, w_cumul_execution_time = %lf\n",
                                  starpu_worker_get_id(),
                                  w_total_executed,
                                  w_cumul_execution_time);
}
```

**Step 5: Initialize the listener**

This step allocates the listener structure and prepares it to listen to the selected set of per-worker counters. However, it is not actually active until Step 6, once it is attached to one or more worker.

```
struct starpu_perf_counter_listener * w_listener =
                        starpu_perf_counter_listener_init(w_set, w_listener_cb, NULL);
```

**Step 6: Set the listener on all workers**

This step actually makes the listener active, in this case on every StarPU worker thread.

```
starpu_perf_counter_set_all_per_worker_listeners(w_listener);
```

After this step, any task assigned to a worker will be counted in that worker selected performance counters, and reported to the listener.