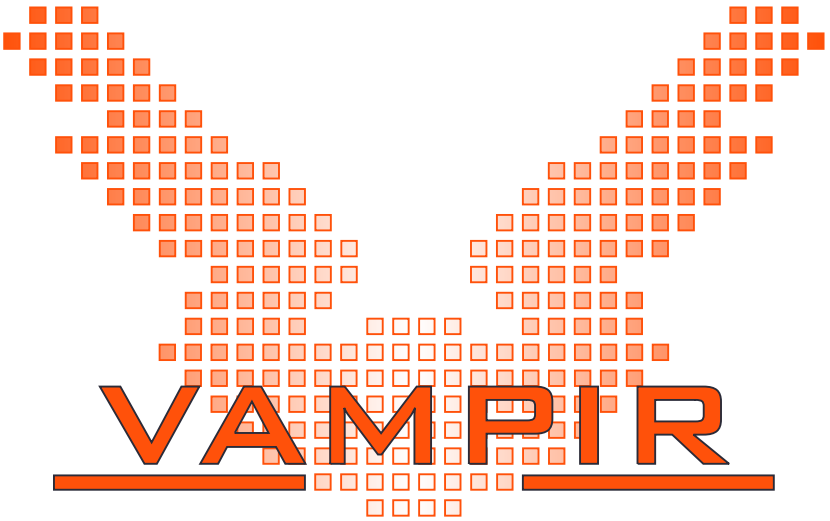
VampirTrace

Plugin Counter Manual



TU Dresden
Center for Information Services and
High Performance Computing (ZIH)
01062 Dresden
Germany

http://www.tu-dresden.de/zih
http://www.tu-dresden.de/zih/vampirtrace

Contact: vampirsupport@zih.tu-dresden.de

Contents

This documentation describes the usage of plugin counters for VampirTrace. It describes the general idea of these plugins as well as how to enable them or to implement a new one.

1 General Information

Plugin counters are an easy way to extend the functionality of VampirTrace to your needs. They allow to define additional counters as external library, which is then loaded when tracing your application. So there is no need to recompile VampirTrace and neither to instrument your application manually with VT_USER counters.

In the following, the word process describes a task within the operating system that consists of a number of threads (at least one) which share some process resources. A thread is a smaller unit of processing and therefor much more light weight then a process. Threads within a process have shared (e.g. heap memory) and private resources (e.g. stack memory). The implementation of threads and processes is highly platform dependend.

1.1 Audience

The target audience of this feature are people who want to trace:

- system behaviour, that cannot be related to a specific thread or process.
- repeatingly the same metric, which is not specified in another feature of VampirTrace (Consider using PAPI, it's much faster and more comfortable then writing this on your own)
- a specific metric, but are too scared of the VampirTrace source (which is open-source btw.)

Cases when you should use something else are:

- if additional overhead is inacceptable for you.
- if you only trace ONE program. You should use the VT_USER related functions and instrument your code manually.

1.2 Installing a Plugin Counter

Plugin counters are provided as shared libraries. To use them, they should be placed in a directory which is part of the `LD_LIBRARY_PATH` (like `/lib` or `/usr/lib`).

1.3 Information about Plugin Counters

If the developer of a plugin did his job, there should be a README on how to compile it, install it, use it, and what events the plugin supports.

1.4 Enabling a Plugin Counter

To enable a plugin, set the environment variable `VT_PLUGIN_CNTR_METRICS`. As an example, if you have a library named libKswEvents.so, with the event page_faults, set it with

```
export VT_PLUGIN_CNTR_METRICS=KswEvents_page_faults
```

Multiple plugins and counters can be selected by separating them with :'s.

2 Implementing a new Plugin

To define a new plugin include `vt_plugin_cntr.h` and implement the function `get_info()`.

2.1 General Functions

Some functions are mandatory and have to be implemented for every plugin, others are optional and some have to be implemented for special types of plugins.

2.1.1 Mandatory functions

init

The initializing function should check, whether counting of this plugin is generally available for the current system, whether the user has the right to count, whether an external database is available, and so on. Also it should initialize most of the data structures used from now on.

```
int32_t init(void);
Return: Whether the plugin could be initialized
        correctly (0) or not (!= 0)
```

Note: On VampirTrace Version 5.10 (VT_PLUGIN_CNTR_VERSION 1), this function is called once per process. Even on processes where the event might not be traced!

get_event_info

Plugins may provide the functionality of wildcards, so you might have multiple events for one event string containing such a wildcard.
Example: The user sets `VT_PLUGIN_CNTR_METRICS=myPlugin_*`, the developer of the myPlugin might want to expand the passed string "$*$" to "real" counters like "counter 1" and "counter 2". This should be done in this function.
The returned list should end with an element whose name is NULL, for all other elements, the `name` and the `cntr_property` are mandatory, the `unit` is optional and might be NULL.

The cntr_property define informations about the given metric. To set them combine the definitions of a data type:

```
VT_PLUGIN_CNTR_[FLOAT|DOUBLE|SIGNED|UNSIGNED]
```

with the definition where in time the value is relevant:

```
VT_PLUGIN_CNTR_[START|POINT|LAST|NEXT]
```

and the information about whether the value is absolute or accumulated:

```
VT_PLUGIN_CNTR_[ABS|ACC].
```

To combine these definitions, use the bitwise or "|". An example would be:

```
/*data type is double,the value is valid from now until
* the next value is reported, the value is absolute.
*/
element[0].cntr_property=VT_PLUGIN_CNTR_DOUBLE |
                         VT_PLUGIN_CNTR_NEXT |
                         VT_PLUGIN_CNTR_ABS
```

More information on the specifications can be found in vt_plugin_cntr.h. To write a value, which does not have the data type uint64_t (VT_PLUGIN_CNTR_UNSIGNED) it is advised to use unions to write the actual date. If you use for example the data type double, you might want to use this:

```
  //...
  union{
    uint64_t u64;
    double dbl;
  } value;
  //...
  value.dbl=measure_value();
  // return value.u64 at some point

vt_plugin_cntr_metric_info *get_event_info(
  char * cmd_line_name)
Input: Metric name read from the command line
Return: List of information about the selected metrics.
```

Note: This function is called once per process after init() is called. It is called once for every command line metric, which refers to the plugin. The function is also called on processes where the event might not be monitored!

add_counter

The add counter function is used to add counters, which may be called per thread, per process, per host or only once. This depends on the run_per variable defined in the info struct. However, this should initialize the counting procedure, but not start it. The returned counter ID is eminent for the further measurement process, since VampirTrace will use this ID from now on to get results, en- and disable the counting and so on. The plugin has to be aware of this counter and the related counting facility structure.

```
int32_t add_counter(char * metric_name)
Input:metric_name, name of the selected metric (provided
       by get_event_info)
Return: a unique ID (unique within the plugin)
        or -1 if adding the counter failed
```

Note: The generated and returned ID has to be unique, multiple threads may call this function at the same time, so if you have a PER_THREAD plugin, write the ID generation thread-save!

finalize

This method should free all resources, finalize the internal counting infrastructure and so on.

```
void finalize()
```

Note: This function is also called per process even on those where no event from this plugin is traced.

2.1.2 Mandatory variables

This variable defines for what type of threads/processes the counters defined in this plugin are measured. `int32_t run_per`

- VT_PLUGIN_CNTR_ONCE: It is measured only once: for the first process , first thread on the first node used.

  Example: SAN accesses from a cluster

- VT_PLUGIN_CNTR_PER_HOST: The counters of this plugin will be measured on the first process, first thread, but for all nodes

  Example: Energy consumption of a node

- VT_PLUGIN_CNTR_PER_PROCESS: The counters of this plugin will be measured for all processes, but only for the first thread.

  Example: Active threads per process.

- VT_PLUGIN_CNTR_PER_THREAD: The counters of this plugin will be measured for all threads of all processes.

  Example: Page faults per thread.

The implications deriving from this variable are described in the section “Types of plugins” `int32_t synch`

- VT_PLUGIN_CNTR_SYNCH: A current value of the events of this plugins is queried whenever a VampirTrace event occurs.

- VT_PLUGIN_CNTR_ASYNCH_EVENT: The events are collected asynchronously by the plugin. Whenever there’s a VampirTrace event, all ASYNCH plugin events are collected by VampirTrace.

- VT_PLUGIN_CNTR_ASYNCH_POST_MORTEM: The events are collected asynchronously by the plugin. When VampirTrace terminates, all events are collected from such plugins by VampirTrace.

- VT_PLUGIN_CNTR_ASYNCH_CALLBACK: The events are collected asynchronously by the plugin. The plugin calls a callback function for every event which occurs.

`uint32_t vt_plugin_cntr_version` should always be set to `VT_PLUGIN_CNTR_VERSION` to allow compatiblity checks.

2.1.3 Optional functions

These functions can be implemented in the plugin and will be evaluated by VampirTrace. However, if they are not implemented, they will be ignored but the tracing is still valid.

enable_counter

This function should enable the counting of an event. It should run fast (if possible). The function might be called per thread, so be thread save within it.

```
int32_t enable_counter(int32_t counter_id)
Input: ID, id of the counter to enable (was generated
       by add_counter(...))
Return: whether successfull (0) or not (!=0)
```

Note: There might be multiple enable_counter calls which are called consecutively.

disable_counter

This function should disable the counting of an event. It should run fast (if possible). The function might be called per thread, so be thread save within it.

```
int32_t disable_counter(int32_t counter_id)
Input: ID, id of the counter to disable (was generated
       by add_counter(...))
Return: whether successfull (0) or not (!=0)
```

Note: There might be multiple disable_counter calls which are called consecutively.

2.2 Types of plugins and functions per type

There are several types of plugins, which may be implemented. These vary in the way they are called as well as in the functions to implement.
The type of plugin has to be specified in the info variable “synch” and has to be one of the values defined in `enum vt_plugin_cntr_synch`.

2.2.1 Synchronous plugins

Synchronous plugins are called whenever an event is generated by VampirTrace. E.g. when a function is entered and exited, the current value is get and stored for all available plugins.
Pro:

- There is no need to measure the time (and convert it), but only to report the current value.
- The implementation is pretty easy, since most functionality is not needed.

Contra:

- The functionality is pretty limited, since no counter events between VampirTrace events can be counted.

Synchronous plugins have to implement the following function:

```
uint64_t get_current_value(int32_t counter_id)
Input: counter, defines the counter id provided by
       add_counter(...)
Return: the current value of the counter
```

2.2.2 Asynchronous events

Asynchronous events collect data themselves, reporting it to VampirTrace whenever it is needed. They provide VampirTrace with both: time stamp and value, defined in the `vt_plugin_cntr_timevalue` struct. They are available since VampirTrace version 5.11. As additional funtions, they have to implement:

```
void set_pform_wtime_function(
                              uint64_t (*pform_wtime)(void))
Input: A function, that creates VampirTrace
       compatible timestamps
```

Note: This is the only function that is called before init().

Providing timestamps

All times within VampirTrace are reported using an internal timing method, which generates timestamps that are not compatible with yours. Never. (Since they may change from platform to platform).
Changing your time getting method can be more or less complicated, depending on how you get timestamps and how you use them. It is simple if you get the timestamps yourself within the asynchronous thread. Just store the wtime function, VampirTrace passes to you before starting the init function. Use this wtime function to get a valid timestamp.
If you get the time stamp from some sort of external process or another task it is not that simple. In this case, you need to take timestamps with wtime and your timer, to build up a formula, which recalculates the original time stamps. Example: your timer (gettime()) delivers milliseconds

```
uint64_t start_my_time=gettime();
uint64_t start_vt_time=wtime();
// do initialization (hopefully it will be long enough)
. . .
uint64_t end_my_time=gettime();
uint64_t end_vt_time=wtime();
// how much faster/slower is your timer compared to ours
double factor=
  (end_vt_time-start_vt_time)/(end_my_time-start_my_time);
// when did the timer start
double additional=start_vt_time-factor*start_my_time;
//generate future timestamps with:
uint64_t vt_timestamp=additional+my_timestamp*factor;
```

To gain more precision for the timer, you might do the time measuring of start and end times over a longer period. For `ASYNCH_POST_MORTEM` or `ASYNCH` plugins, you should get the end_time when `get_all_values` is called for the first time.

Types of asynchronous events and reporting

There are different types of asynchronous events, which are described here:

- `ASYNCH` Plugins of this type will be asked to report new events whenever VampirTrace generates an event. The plugin should buffer the events which were generated to this point. It may implement an own environment variable, leting the user define the buffer size to prevent too large buffers.

  These plugins will be at least asked for new events when VampirTrace finishes. Asynch plugins have to implement the following function:

```
uint64_t get_all_values(int32_t counter
  vt_plugin_cntr_timevalue ** result_vector)
Input:
counter: defines the counter id generated by
         add_counter(...)
Output:
result_vector: a pointer to the available
        results (will be freed by VampirTrace,
        may be NULL if no data is provided)
Return: the number of results in result_vector
```

- `ASYNCH_POST_MORTEM` are comparable to `ASYNCH` plugins. The only difference is that the `get_all_values` function is only called once (maybe once per thread) after the tracing. Again, the buffer for data should be large enough or configurable for these cases.

- `ASYNCH_CALLBACK` These plugins generate and report events themselves. They allow to use the internal VampirTrace buffer, avoiding too much memory overhead.
  As additional functions, they have to implement:

```
int32_t set_callback_function(
     void * ID,
     int32_t counter_id,
     int32_t (*callback_function)
          (void *,vt_plugin_cntr_timevalue) callback)
     )
Input:
ID: VampirTrace will provide a void pointer, which
    has to be passed in every subsequent callback
    for this counter
counter_id: defines the counter id generated by
            add_counter(...)
callback: defines the function to call when an event
```

```
                shall be reported
  Return: whether setting the callback was okay (0) or not
```

Note: As the callback function is provided some time after the counter is added, you should check whether the callback function and the callback ID was set before you try to use it. **Return values of the callback function:** When the plugin calls the callback funtion, a value from the `enum vt_plugin_callback_return` is returned.

These values are:

- `VT_PLUGIN_CNTR_CALLBACK_OK` - the value has been stored in a buffer
- `VT_PLUGIN_CNTR_CALLBACK_BUFFER_FULL` - the value could not be stored as the buffer is already full (try setting the environment variable `VT_PLUGIN_CNTR_CALLBACK_BUFFER`).
- `VT_PLUGIN_CNTR_CALLBACK_TRACE_OFF_PERMANENT` - the plugin can shut down, as tracing for the current task is permanently disabled. This is the case when the number of maximal flushes is reached.

Note: For older VampirTrace versions (¡5.11), this enum does not exist. Other return values are possible (e.g. for unexpected errors).

- All plugins which create an additional thread to generate the events should implement the following function to prevent sampling threads from being traced.

```
int32_t is_thread_registered()
Return: Whether the calling thread was created
        by this plugin (1) or not (0).
```

2.3 Order of function calling

The funtions described above are called in the following order:

1. `get_info()` - once per plugin
2. `set_pform_wtime_function()` - once per plugin for asynchr. plugins
3. `init()` - once per plugin
4. `get_event_info()` - per selected event in the environment variable
5. `add_counter()` - per selected event per thread received by get_event_info()
6. `enable_counter()` - per added counter per thread
7. sequence of the following function calls in any order per thread

   a) `get_all_values()` - per added asynchr. event counter
   b) `get_current_value()` - per added synchr. counter
   c) `disable_counter()` followed by an `enable_counter()` - per added counter

8. `get_all_values()` - per added asynchr. post mortem counter per thread
9. `disable_counter()` - per added counter per thread
10. `finalize()` - once per plugin

3 Building a plugin

All plugins have to be build as shared library, so it can be load dynamically while executing the tracing. This is how a Makefile might look like:

```
gcc -c -fPIC myPlugin.c -o libMyPlugin.o \
  -I/home/user/vampirtrace/include
gcc -shared -Wl,-soname,libMyPlugin.so \
  -o libMyPlugin.so libMyPlugin.o
```

First the plugin is compiled to an object. The include path of the VampirTrace installation has to be adapted that it matches the compiling system. Second the object file is linked to a shared library. Most plugins will make use of thread synchronization calls like `pthread_mutex_(un)lock` and external libraries. These have to be passed to the linker (e.g. `-lpthread`)

4 Challenges

4.1 Task Availability

VampirTrace reuses thread IDs. If you have for example a per thread counter which measures hardware performance events and use it on an OpenMP or pthread parallel program, the task, which calls the plugins functions and pass counter ids may change during execution. In this case it is the developers job to check the calling pid. This should be done whenever the counter is enabled. The developer should close the previously set up counter for the non-existent thread in this case and use the new one. A task is defined as working and available only between enable- and disable_counters. After a counter is disabled, the operating system task id (tid) of the calling thread may change. For processes a plugin developer can assume that the task is valid from init to finalize.

4.2 Limited Memory for Callback Plugins

This problem can occur when callback plugins generate events much faster then VampirTrace events occur. The internal buffer is flushed every time such an event is recorded. To circumvent this limitation, one can adapt the buffer size available for callback values. Each callback value has a size of 16 byte. Set the environment variable `VT_PLUGIN_CNTR_CALLBACK_BUFFER` to change the amount of available buffer per threaed. The value is passed in bytes. The developer may post a message with a hint to this variable when reporting a value via the callback function fails with return value `VT_PLUGIN_CNTR_CALLBACK_BUFFER_FULL`.

4.3 Locking Code Sections Within a Plugin

A useful per thread plugin itself is not thread safe by default. This limits the overhead of function calling, but hardens the work for the plugin developer. Thread locking mechanisms are highly platform dependend. Most systems support pthreads, so we recommend to use `pthread_mutex_lock` and `pthread_mutex_unlock` from `pthread.h`.

Known functions which are concurring are all those which are called per thread. The plugin developer has to guarantee that these are thread save (for critical sections).

5 Example

This section presents a small example for synchronous counters:

```
/*
* This plugin will add random values for every VampirTrace
* event in your trace
*
* Compile it with
*   cc -c -fPIC random_plugin.c -o RandomPlugin.o
*     -I/path/to/vtrace
* Link it with
*   cc -L/path/to/vtrace -shared\
*     -Wl,-soname,libRandomPlugin.so \
*     -o libRandomPlugin.so RandomPlugin.o
* Copy it to a folder in LD_LIBRARY_PATH or add the
* current path with
*   export LD_LIBRARY_PATH=$PWD:$LD_LIBRARY_PATH
* Use it with setting:
*   export VT_PLUGIN_CNTR_METRICS="RandomPlugin_banana"
* To add one counter that is named "banana" including
* random numbers.
* Set it to "RandomPlugin_*" to add NUMBER_RANDOM_COUNTER
* counters numbered from 0 to NUMBER_RANDOM_COUNTER-1
*/

#include <vampirtrace/vt_plugin_cntr.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
/* mutex to be thread save */
#include <pthread.h>

#define NUMBER_RANDOM_COUNTER 16

static pthread_mutex_t add_counter_mutex;

static int32_t nr=0;
```

```
int32_t init(){
  /* check if pthread mutex can be created */
  return pthread_mutex_init( &add_counter_mutex, NULL );
}



int32_t add_counter(char * event_name){
  /* id generation has to be thread save */
  int id;
  pthread_mutex_lock( &add_counter_mutex );
  id=nr;
  nr++;
  pthread_mutex_unlock( &add_counter_mutex );
  return id;
}

vt_plugin_cntr_metric_info * get_event_info(
      char * event_name){
  vt_plugin_cntr_metric_info * return_values;
  char name_buffer[255];
  int i;
  /* if wildcard, add some random counters */
  if (strcmp(event_name,"*")==0){
    return_values= malloc((NUMBER_RANDOM_COUNTER+1) *
        sizeof(vt_plugin_cntr_metric_info) );
    for (i=0;i<NUMBER_RANDOM_COUNTER;i++){
      sprintf(name_buffer,"random counter #%i",i);
      return_values[i].name=strdup(name_buffer);
      return_values[i].unit=NULL;
      return_values[i].cntr_property=
        VT_PLUGIN_CNTR_LAST|VT_PLUGIN_CNTR_ABS|
        VT_PLUGIN_CNTR_UNSIGNED;
    }
    return_values[NUMBER_RANDOM_COUNTER].name=NULL;
  /* if no wildcard is given create one random counter
     with the passed name */
  } else{
    return_values= malloc(2*
        sizeof(vt_plugin_cntr_metric_info) );
    sprintf(name_buffer,
        "random counter #%s",event_name);
    return_values[0].name=strdup(name_buffer);
    return_values[0].unit=NULL;
```

```
    return_values[0].cntr_property=
      VT_PLUGIN_CNTR_LAST|VT_PLUGIN_CNTR_ABS|
      VT_PLUGIN_CNTR_UNSIGNED;
    return_values[1].name=NULL;
  }
  return return_values;
}

uint64_t get_value(int32_t counterIndex){
  return rand();
}

void fini(){
  pthread_mutex_destroy( &add_counter_mutex );
}

vt_plugin_cntr_info get_info(){
  vt_plugin_cntr_info info;
  memset(&info,0,sizeof(vt_plugin_cntr_info));
  info.init=init;
  info.vt_plugin_cntr_version = VT_PLUGIN_CNTR_VERSION;
  info.add_counter=add_counter;
  info.run_per=VT_PLUGIN_CNTR_PER_THREAD;
  info.synch=VT_PLUGIN_CNTR_SYNCH;
  info.get_event_info=get_event_info;
  info.get_current_value=get_value;
  info.finalize=fini;
  return info;
}
```