OCERA 2nd phase Deliverables

OCERA 2nd phase Deliverables

Table of Contents

Chapter 1. Metrics

Summary

Name

Metrics (*METRICS*)

Description

This component is a library capable of extracting measurements of system metrics from POSIX trace streams.

Author/s

Agustin Espinosa, Andres Terrasa, Ana Garcia-Fornes.

Reviewer

Layer

Library level RTLinux and Linux

Version

1.0

Status

Alpha

Dependencies

Lightweight POSIX Trace

Release Date

??

Description

The POSIX tracing services specify a set of interfaces to allow for portable access to underlying trace management services by application programs. Programmers may use these services to get a sequence of trace events generated by the system during the execution of their application. These trace events are kept in a POSIX trace stream. The contents of a trace stream can be analyzed while the tracing activity takes place or it can be analyzed later, once the tracing activity has been completed. A trace event is generated when some action takes place in the system and this trace event may be stored in one or several trace streams. Each trace event contains data which is relative to the action that has generated it. The POSIX tracing services require that data such as the following be associated to each trace event: event type, time stamp, process identifier and thread identifier. Using these data, we can get time related system metrics such as the execution time of a given system call, the response time of a periodic job, etc.

Unfortunately, the interpretation of the events which are stored in trace streams may be difficult for programmers who do not know the system implementation in detail. Events stored in a trace stream represent system actions such as context switches, hardware interrupts, state changes, etc. In order to extract metrics from these events it is necessary to know how the execution of the system generates these events, and

normally who has implemented the system is the only one which knows this information. In order to solve this problem, this component implements a metrics extraction engine and provides an application interface for using this engine. This interface allows the programmer to obtain predefined system metrics from trace streams without it being necessary for the programmer to know the system implementation.

API / Compatibility

The application interface for this component is as follows:

Metrics are used in this interface by means of metrics identifiers, which are objects of the `metrics_metric_id_t` type. This library offers a fixed set of metrics and each metric has its own name, "M_JOB_RESPONSE_TIME" by example. The user of this library shall provide a predefined metric name in order to get a valid metric identifier. These identifiers can be retrieved by using the following function:

```
int
metrics_metric_open (const char          *metric_name,
       metrics_metric_id_t *metric_id);
```

Metrics identifiers can be grouped in metrics sets, which are objects of the `metrics_metricset_t` type. A program uses these sets in order to define the metrics which shall be extracted from a trace stream. The following functions can be used in order to manipulate metric sets:

```
int
metrics_metricset_empty (metrics_metricset_t *set);

int
metrics_metricset_fill (metrics_metricset_t *set);

int
metrics_metricset_add (metrics_metric_id_t  metric_id,
         metrics_metricset_t *set);

int
metrics_metricset_del (metrics_metric_id_t  metric_id,
         metrics_metricset_t *set);

int
metrics_metricset_ismember (metrics_metric_id_t        metric_id,
       const metrics_metricset_t *set,
       int                       *ismember);
```

The metrics extraction engine implemented by this library shall be initialized before it can be used to extract metrics from a trace stream. This initialization is carried out by the following function:

```
int metrics_init (trace_id_t trid, const metrics_metricset_t *set);
```

This initialization action binds the metrics extraction engine with a trace stream and a metric set. The trace stream identified by the *`trid`* argument will be used later by the metrics extraction engine in order to search metrics and retrieve measurements

for this metrics. Both pre-recorded or a active trace stream can be binded to the metrics extraction engine and the identifiers for these trace stream shall be retrieved by using the appropriate functions available in the POSIX Tracing interface. The metric set identified by the *set* argument is used to select the metrics that the metrics extraction engine will search in the trace stream.

Once the metrics extraction engine is initialized, measurements can be retrieved from the trace stream. This measurements are objects of the *struct metrics_measurement_t* type and can be retrieved by using the following functions:

```
int
metrics_getnext_measurement (metrics_measurement_t *result,
        int                  *unavailable);


int
metrics_trygetnext_measurement (metrics_measurement_t *result,
    int                  *unavailable);


int
metrics_timedgetnext_measurement (metrics_measurement_t *result,
      int                  *unavailable,
      const struct timespec *abs_timeout);

typedef struct {
  metrics_metric_id_t  metric_id;
  pthread_t            thread_id;
  struct timespec      duration;
  struct timespec      begin;
  struct timespec      end;
  int                  events_count;
  int                  id;

} metrics_measurement_t;
```

These functions search metrics in the trace stream and, when a metric is found, a measurement for the metric found is reported. The argument *unavailable* is set to zero when no more metrics can be retrieved from the trace stream. This occurs when all the trace events in the trace stream has been retrieved, for pre-recorded trace streams, or when the trace stream is destroyed, for active trace streams.

The `metrics_getnext_measurement` function retrieves as many trace events from the trace stream as necessary in order to find a metric and returns when a metric is found. The calling process may be suspended if no trace events are stored in the trace stream and this trace stream is an active trace stream. The execution of the calling process is restarted when a new trace event is stored in the trace stream.

The `metrics_trygetnext_measurement` function retrieves only a trace event from the trace stream at once in order to find a metric. If no metric is found, then an error code is returned indicating this situation. This function is applicable to active trace streams only.

The `metrics_timedgetnext_measurement` function behaves as the `metrics_getnext_measurement` function, but if the calling process is suspended, it is restarted at the time indicated in the *abs_timeout* argument. In this case this function returns an error code indicating this situation. This function is applicable to active trace streams only.

Whenever a metric is found, these functions return a measurement for this metric. A measurement includes the following data:

metric_id	The metric associated to this measurement
thread_id	The thread associated to this measurement
duration	The amount of time in which the system has been in the state which corresponds with the metric found
begin	The time stamp of the first trace event which corresponds to the metric found
end	The time stamp of the last trace event which corresponds to the metric found
event_count	The number of trace events generated by the trace system while the system was in the state which corresponds with the metric found

In the current stage of the Metrics library implementation, the following metrics are currently defined:

RUNNING_SECTION

The amount of time elapsed since a thread is dispatched until another thread is dispatched

CLOCK_NANOSLEEP_NO_SUSPENDS

Kernel execution time used to serve a `clock_nanosleep` function when the calling thread is not suspended

CLOCK_NANOSLEEP_UNTIL_SUSPENDS

Kernel execution time used to serve a `clock_nanosleep` function until the calling thread is suspended

CLOCK_NANOSLEEP_AFTER_SUSPENDS

Kernel execution time used to serve a `clock_nanosleep` function since the calling thread is akakened until this function returns

JOB_RESPONSE_TIME_CLOCK_NANOSLEEP

Job response time for a thread which uses the `clock_nanosleep` to implement its periodic behaviour

JOB_EXECUTION_TIME_CLOCK_NANOSLEEP

Job execution time for a thread which uses the `clock_nanosleep` to implement its periodic behaviour

JOB_INPUT_JITTER_CLOCK_NANOSLEEP

Input jitter for a thread which uses the `clock_nanosleep` to implement its periodic behaviour

JOB_OUTPUT_JITTER_CLOCK_NANOSLEEP

Output jitter for a thread which uses the `clock_nanosleep` to implement its periodic behaviour

Implementation issues

Measurements are obtained by searching sequences of events stored in a trace stream that correspond to the metrics the programmer wants to obtain. Each metric provided by the metrics search engine is analyzed by an automaton, or, more precisely, for a group of equivalent automata, each of which is associated to a different thread identifier. These automata are internal to the implementation and therefore the users of the Metrics interface don't use them directly.

We decided to use automata because they adapt very well to the nature of the information to be processed, which is formed by a sequence of trace events. Another advantage of using automata is they allow to implement new metrics in a very easy way.

These automata are defined using the TCL scripting language. Each automaton is defined as a TCL list, in which the automaton states and transitions are declared. An example of this automaton definition is shown in the next code fragment.

```
{RUNNING_SECTION
    {   ## States

         #  Name           Type

 {NOT_RUNNING      OUT }
 {RUNNING          IN  }
 {DONE             END }
    }
    {   ## Transitions

         # From        To              Label

 {NOT_RUNNING  RUNNING     {NORMAL M_CONTEXT_SWITCH  SELF_THREAD }} #1
 {RUNNING      DONE        {NORMAL M_CONTEXT_SWITCH  OTHER_THREAD}} #2
 {RUNNING      DONE        {NORMAL M_LAST_EVENT      ANY_THREAD  }} #3
 {DONE         NOT_RUNNING {LAMBDA}}                                #4
    }
}
```

This automaton example implements the metric *RUNNING_SECTION*. This metric represents the amount of time elapsed since a thread is dispatched until another thread is dispatched.

This definition declares an automaton class, and this class is instantiated at run time for any thread which is detected in the trace stream which is being scanned. In this way, if an application has ten threads, then ten automata for the metric *RUNNING_SECTION* are generated, each one of them binded to a specific thread.

This automaton class has two main states: *NOT_RUNNING* and *RUNNING*. The *NOT_RUNNING* state is the initial state of the automaton and indicates that the thread binded to the corresponding automaton instance is not running. This state is of the type *OUT*, that means that the automaton is not recognizing the metric *RUNNING_SECTION*.

When the system performs a context switch to the thread binded to the corresponding automaton instance, then the automaton switches to the *RUNNING* state. This state change is mandated for transition 1. This transition applies when the trace event *M_CONTEXT_SWITCH* is detected in the trace stream and the system selects the thread binded (*SELF_THREAD*) to the automaton instance. The *RUNNING* state is of type *IN* that means that the automaton is now recognizing the metric *RUNNING_SECTION*.

Finally, when the automaton detects in the trace stream that the system performs a context switch to other thread or the last event of the trace stream has been retrieved, the automaton changes to the *DONE* state, which is of the type *END*. This change is mandated by transitions 2 or 3. When a state of type *END* is reached, the automaton produces a measurement. Next, due to the lambda transition 4, the automaton changes to the *NOT_RUNNING* state.

Automata instances have two internal modes of operation: recognizing or not recognizing a metric. An automaton instance is in the not recognizing mode initially. Being

the automaton in the not recognizing mode, it enters in the recognizing mode when an *IN* state is reached. An automaton instance switches to the not recognizing mode again when it reaches an *END* state, and produces a measurement with the following data:

metric_id	The metric corresponding to the automaton instance
thread_id	The thread binded to the automaton instance
duration	The sum of the length of the segments detected by the automaton while it was in the recognition mode. A segment is formed by a sequence of *IN* states and the length of a segment is the difference between the time stamps of the trace events that have determined the initial and final states of the segment
begin	The time stamp of the trace event which initiates the metric recognition mode
end	The time stamp of the trace event which makes the automaton instance to switch to the not recognition mode
event_count	The number of trace events detected while the automaton instance was in a state of the type *IN*

The previous automaton class example is quite simple since a measurement is comprised by a single segment, and so not all the functionality of these automata is shown. By example, when more complex metrics are defined, several segments are part of single measurement normally. There are two more state types also, *CANCEL* and *CANCEL_SEGMENT*. These states allow to cancel the current measurement or the current segment respectively and they are used when the automaton detects that the current situation detected in the trace stream doesn't really corresponds to the metric which is being analyzed.

Once the automata used by the metrics extraction engine have been described, let us see how these automata are implemented.

An automaton class is represented internally as an array of state descriptors. Each one of these state descriptor holds the type of the state and a pointer to the head of a list of transition descriptors, which is formed by the output transitions of the state. A transition descriptor holds its target state, its label and a pointer to the next transition in its transition list. At compile time, a TCL script reads the definitions of the automata classes and generates C code that declares the corresponding data structures. An example of the generated C code is shown in the next code fragment, which corresponds with the automaton which scans the metric *RUNNING_SECTION* described above.

```
mtri_transition_descriptor_t tr_1 =
      {1, NORMAL, M_CONTEXT_SWITCH, SELF_THREAD, NULL};
mtri_transition_descriptor_t tr_2 =
      {2, NORMAL, M_CONTEXT_SWITCH, OTHER_THREAD, NULL};
mtri_transition_descriptor_t tr_3 =
      {2, NORMAL, M_LAST_EVENT, ANY_THREAD, &tr_2};
mtri_transition_descriptor_t tr_4 =
      {0, LAMBDA, M_ANY_EVENT, ANY_THREAD, NULL};


mtri_metric_descriptor_t mtri_metric [] =
{
{{OUT}, {IN}, {END}},
{&tr_1, &tr_3, &tr_4}
},
{
```

```
  // Other automata
};
```

As you can see, automata are declared statically in the C code. This approach has the advantage that no code is necessary at run time to initialize automata classes, making the memory size of the library smaller.

By the other hand, automaton instances are generated using dynamic memory at run time only when they are required. Particularly, when a new thread identifier is detected in the trace stream which is being scanned, automaton instances binded to this thread are created for all the metrics selected in the metrics engine initialization. Each automaton instance is represented by a automaton instance descriptor. This descriptor holds information about the current state of the automaton instance, a pointer to its class automaton and accounting information related to measurement being performed by the automaton instance. All of these automaton descriptors are holded in a single linked list. When an trace event is retrieved from the trace stream, this event is delivered sequentially to each one of the automaton instances in this list.

The temporal cost of retrieving a measurement is as follows. Processing a single event from a single automaton instance has a cost of *O(1)* . The `metrics_getnext_measurement` function processes as many trace event as necessary in order to get a measurement, and each trace event is delivered to all the automaton instances of the automaton instances list. The size of this list is *M x T*, being *M* the number of selected metrics and *T* the number of threads detected. The total cost for this function is *O(E x M x T)*, being *E* the number of trace events required to get a measurement. The cost of the `metrics_timedgetnext_measurement` function is the same, since it has the same definition. Finally, the cost of the `metrics_trygetnext_measurement` function is *O(M x T)*, since this function processes only a trace event at time.

Tests and validation

Validation criteria

The main validation criteria for this component are the following: the quality of the interface, its correctness and its efficiency, mainly when it is used for on-line metrics retrieval.

The quality of the interface includes aspects such as clarity, ease to use, compatibility with other interfaces and to be appropriate to implement it efficiently. A great effort has been done in order to design a high quality interface. This effort has been based in following the same design principles used in modern POSIX interfaces and to develop several application programs which uses this interface in order perform useful tasks, such as generating metrics reports or supervising the temporal behavior of real-time applications.

Correctness is an obvious validation criteria. In order to meet this requirement more easily, the Metrics library has been implemented as two different parts: an automata definition system and a generic core capable to use the previously defined automata. The advantages of this approach is that the automata definition system and the generic core are small and simple programs, and so their correctness is easy to validate.

By having enough conficence about the correctness of this implementation, we can deal with the more difficult aspect of the overall correctness: to achive that a particular automaton class corresponds whith an unique path in the RT_Linux system execution and this execution path let us to detect the metric for which the automaton

class is designed. This correctness can be achieved by having a precise knowledge of the RT-Linux system and by intensive testing and examination of the trace events stored in the generated trace streams.

Respecting efficiency, in this first stage of implementation we are trying to meet this criteria by selecting the more appropriate data structures. The overhead of this Metric library will be measured in the following implementation stages, and a more fine tuning of the implementation will be done.

Tests

The Metrics library is currently in the testing phase, specially in respect to the implementation of new metrics. Test programs are always a set a real-time threads in which the metrics which are being tested are present. Trace streams are obtained from the execution of these programs and they are analyzed for the metrics extraction engine, in order to test if the atomata corresponding to the metrics which are being tested are well defined. <\para>

Example

The following code fragment shows the expected usage of the Metrics application interface. First, a set formed by two predefined metrics is built and the metrics extraction engine is initialized. Next, all the measurements are extracted from the trace stream and processed.

```
  metrics_metricset_empty (&set);
  res = metrics_metric_open ("JOB_EXECUTION_TIME_CLOCK_NANOSLEEP",
        &JOB_EXECUTION_TIME_CLOCK_NANOSLEEP);
  res = metrics_metric_open ("JOB_RESPONSE_TIME_CLOCK_NANOSLEEP",
        &JOB_RESPONSE_TIME_CLOCK_NANOSLEEP);

  metrics_metricset_add (JOB_EXECUTION_TIME_CLOCK_NANOSLEEP, &set);
  metrics_metricset_add (JOB_RESPONSE_TIME_CLOCK_NANOSLEEP, &set);

  metrics_init (trid, &set);

  metrics_getnext_measurement (&result, &unavailable);

  while (! unavailable) {

    process_measurement (&result);

    metrics_getnext_measurement (&result, &unavailable);
  }
}
```

The above code fragment can be used both for off-line and for on-line metrics retrieval. For off-line retrieval, the trace stream used to initialize the metrics extraction engine should be a pre-recorded trace stream. In this case, the trace stream should be created by using the POSIX tracing `posix_trace_open` function, such is shown in the following code fragment:

```
  fd = open ("trace.log", O_RDWR);
  posix_trace_open (fd, &trid);
```

A typical usage for off-line retrieval is the generation of a data file in which all the measurements are stored in order to generate a metrics report later. A program which generates this data file is currently available. This program generates an XML formatted data file with all the measurements found. An example of a fragment of this data file is the following:

```
<MEASUREMENT>
  <METRIC>     JOB_EXECUTION_TIME_CLOCK_NANOSLEEP   </METRIC>
  <THREAD>                        1   </THREAD>
  <DURATION>           0.003412928   </DURATION>
  <BEGIN>              1.000565376   </BEGIN>
  <END>                1.063999008   </END>
  <EV_COUNT>                    52   </EV_COUNT>
  <ID      >                    37   </ID>
</MEASUREMENT>
```

When on-line retrieval is used, the trace stream should be an active trace stream, and it should be created by using the tracing `posix_trace_create` function:

```
  posix_trace_create (0, &attr, &trid);
```

By using the on-line metrics retrieval it is possible, for example, to implement a task which supervises at run time that temporal requirements, such as the deadline or the worst-case execution time, are meet for the normal application tasks.