

Percept

Copyright © 2007-2016 Ericsson AB, All Rights Reserved
Percept 0.9
December 6, 2016

Copyright © 2007-2016 Ericsson AB, All Rights Reserved
Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0 Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License. The Initial Developer of the Original Code is Ericsson AB. Ericsson AB, All Rights Reserved.

December 6, 2016

1 Percept User's Guide

Percept is an acronym for **P**ercept - **er**lang **c**oncurr**e**ncy **p**rofiling **t**ool.

It is a tool to visualize application level concurrency and identify concurrency bottlenecks.

1.1 Percept

Percept, or Percept - Erlang Concurrency Profiling Tool, utilizes trace informations and profiler events to form a picture of the processes's and ports runnability.

1.1.1 Introduction

Percept uses `erlang:trace/3` and `erlang:system_profile/2` to monitor events from process states. Such states are,

- waiting
- running
- runnable
- free
- exiting

There are some other states too, `suspended`, `hibernating`, and garbage collecting (`gc`). The only ignored state is `gc` and a process is considered to have its previous state through out the entire garbage collecting phase. The main reason for this, is that our model considers the `gc` as a third state neither active nor inactive.

A waiting or suspended process is considered an inactive process and a running or runnable process is considered an active process.

Events are collected and stored to a file. The file can be moved and analyzed on a different machine than the target machine.

Note, even if percept is not installed on your target machine, profiling can still be done via the module *percept_profile* located in runtime_tools.

1.1.2 Getting started

Profiling

There are a few ways to start the profiling of a specific code. The command `percept:profile/3` is a preferred way.

The command takes a filename for the data destination file as first argument, a callback entry-point as second argument and a list of specific profiler options, for instance `procs`, as third argument.

Let's say we have a module called example that initializes our profiling-test and let it run under some defined manner designed by ourself. The module needs a start function, let's call it go and it takes zero arguments. The start arguments would look like:

```
percept:profile("test.dat", {test, go, []}, [procs]).
```

For a semi-real example we start a tree of processes that does sorting of random numbers. In our model below we use a controller process that distributes work to different client processes.

```
-module(sorter).
-export([go/3,loop/0,main/4]).

go(I,N,M) ->
    spawn(?MODULE, main, [I,N,M,self()]),
    receive done -> ok end.

main(I,N,M,Parent) ->
    Pids = lists:foldl(
 fun(_,Ps) ->
     [ spawn(?MODULE,loop, []) | Ps]
 end, [], lists:seq(1,M)),

    lists:foreach(
 fun(_) ->
     send_work(N,Pids),
     gather(Pids)
 end, lists:seq(1,I)),

    lists:foreach(
 fun(Pid) ->
     Pid ! {self(), quit}
 end, Pids),

    gather(Pids), Parent ! done.

send_work(_,[]) -> ok;
send_work(N,[Pid|Pids]) ->
    Pid ! {self(),sort,N},
    send_work(round(N*1.2),Pids).

loop() ->
    receive
 {Pid, sort, N} -> dummy_sort(N),Pid ! {self(), done},loop();
 {Pid, quit} -> Pid ! {self(), done}
    end.

dummy_sort(N) -> lists:sort([ random:uniform(N) || _ <- lists:seq(1,N)]).

gather([]) -> ok;
gather([Pid|Pids]) -> receive {Pid, done} -> gather(Pids) end.
```

We can now start our test using percept:

```
Erlang (BEAM) emulator version 5.6 [async-threads:0] [kernel-poll:false]

Eshell V5.6  (abort with ^G)
1> percept:profile("test.dat", {sorter, go, [5, 2000, 15]}, [procs]).
Starting profiling.
ok
```

Percept sets up the trace and profiling facilities to listen for process specific events. It then stores these events to the `test.dat` file. The profiling will go on for the whole duration until `sorter:go/3` returns and the profiling has concluded.

Data viewing

To analyze this file, use `percept:analyze("test.dat")`. We can do this on any machine with Percept installed. The command will parse the data file and insert all events in a RAM database, `percept_db`. The initial command will only prompt how many processes were involved in the profile.

```
2> percept:analyze("test.dat").
Parsing: "test.dat"
Parsed 428 entries in 3.81310e-2 s.
    17 created processes.
    0 opened ports.
ok
```

To view the data we start the web-server using `percept:start_webserver/1`. The command will return the hostname and the a port where we should direct our favorite web browser.

```
3> percept:start_webserver(8888).
{started,"durin",8888}
4>
```

Overview selection

Now we can view our data. The database has its content from `percept:analyze/1` command and the webserver is started.

When we click on the `overview` button in the menu percept will generate a graph of the concurrency and send it to our web browser. In this view we get no details but rather the big picture. We can see if our processes behave in an inefficient manner. Dips in the graph represents low concurrency in the erlang system.

We can zoom in on different areas of the graph either using the mouse to select an area or by specifying min and max ranges in the edit boxes.

> **Note:**
>
> Measured time is presented in seconds if nothing else is stated.

Figure 1.1: Overview selection

Processes selection

To get a more detailed description we can select the process view by clicking the `processes` button in the menu.

The table shows process id's that are click-able and direct you to the process information page, a lifetime bar that presents a rough estimate in green color about when the process was alive during profiling, an entry-point, its registered name if it had one and the process's parent id.

We can select which processes we want to compare and then hit the `compare` button on the top right of the screen.

Figure 1.2: Processes selection

Compare selection

The activity bar under the concurrency graph shows each process's runnability. The color green shows when a process is active (which is running or runnable) and the white color represents time when a process is inactive (waiting in a receive or is suspended).

To inspect a certain process click on the process id button, this will direct you to a process information page for that specific process.

Figure 1.3: Processes compare selection

Process information selection

Here we can some general information for the process. Parent and children processes, spawn and exit times, entry-point and start arguments.

We can also see the process' inactive times. How many times it has been waiting, statistical information and most importantly in which function.

The time percentages presented in process information are of time spent in waiting, not total run time.

Figure 1.4: Process information selection

1.2 egd

1.2.1 Introduction

The egd module is an interface for 2d-image rendering and is used by Percept to generate dynamic graphs to its web pages. All code is pure erlang, no drivers needed.

The library is intended for small to medium image sizes with low complexity for optimal performance. The library handles horizontal lines better then vertical lines.

The foremost purpose for this module is to enable users to generate images from erlang code and/or datasets and to send these images to either files or web servers.

1.2.2 File example

Drawing examples:

```
-module(img).

-export([do/0]).

do() ->
    Im = egd:create(200,200),
    Red = egd:color({255,0,0}),
    Green = egd:color({0,255,0}),
    Blue = egd:color({0,0,255}),
    Black = egd:color({0,0,0}),
    Yellow = egd:color({255,255,0}),

    % Line and fillRectangle

    egd:filledRectangle(Im, {20,20}, {180,180}, Red),
    egd:line(Im, {0,0}, {200,200}, Black),
```

```
   egd:save(egd:render(Im, png), "/home/egil/test1.png"),

   egd:filledEllipse(Im, {45, 60}, {55, 70}, Yellow),
   egd:filledEllipse(Im, {145, 60}, {155, 70}, Blue),

   egd:save(egd:render(Im, png), "/home/egil/test2.png"),

   R = 80,
   X0 = 99,
   Y0 = 99,

   Pts = [ {  X0 + trunc(R*math:cos(A*math:pi()*2/360)),
 Y0 + trunc(R*math:sin(A*math:pi()*2/360))
    } || A <- lists:seq(0,359,5)],
   lists:map(
fun({X,Y}) ->
    egd:rectangle(Im, {X-5, Y-5}, {X+5,Y+5}, Green)
end, Pts),

   egd:save(egd:render(Im, png), "/home/egil/test3.png"),

   % Text
   Filename = filename:join([code:priv_dir(percept), "fonts", "6x11_latin1.wingsfont"]),
   Font = egd_font:load(Filename),
   {W,H} = egd_font:size(Font),
   String = "egd says hello",
   Length = length(String),

   egd:text(Im, {round(100 - W*Length/2), 200 - H - 5}, Font, String, Black),

   egd:save(egd:render(Im, png), "/home/egil/test4.png"),

   egd:destroy(Im).
```

First save.

Figure 2.1: test1.png

Second save.

Figure 2.2: test2.png

Third save.

Figure 2.3: test3.png

Fourth save.

Figure 2.4: test4.png

1.2.3 ESI example

Using egd with inets ESI to generate images on the fly:

```
-module(img_esi).

-export([image/3]).

image(SessionID, _Env, _Input) ->
    mod_esi:deliver(SessionID, header()),
    Binary = my_image(),
    mod_esi:deliver(SessionID, binary_to_list(Binary)).

my_image() ->
    Im = egd:create(300,20),
    Black = egd:color({0,0,0}),
    Red = egd:color({255,0,0}),
    egd:filledRectangle(Im, {30,14}, {270,19}, Red),
    egd:rectangle(Im, {30,14}, {270,19}, Black),

    Filename = filename:join([code:priv_dir(percept), "fonts", "6x11_latin1.wingsfont"]),
    Font = egd_font:load(Filename),
    egd:text(Im, {30, 0}, Font, "egd with esi callback", Black),
    Bin = egd:render(Im, png),
    egd:destroy(Im),
    Bin.

header() ->
    "Content-Type: image/png\r\n\r\n".
```

Figure 2.5: Example of result.

For more information regarding ESI, please see inets application *mod_esi*.

2 Reference Manual

Percept is an acronym for **P**ercept - **er**lang **c**oncurr**e**ncy **p**rofiling **t**ool.

It is a tool to visualize application level concurrency and identify concurrency bottlenecks.

egd

Erlang module

egd - erlang graphical drawer

DATA TYPES

```
color()
egd_image()
font()
point() = {integer(), integer()}
render_option() = {render_engine, opaque} | {render_engine, alpha}
```

Exports

color(Color::Value | Name) -> color()

Types:

Value = {byte(), byte(), byte()} | {byte(), byte(), byte(), byte()}

Name = black | silver | gray | white | maroon | red | purple | fuchia | green | lime | olive | yellow | navy | blue | teal | aqua

Creates a color reference.

create(Width::integer(), Height::integer()) -> egd_image()

Creates an image area and returns its reference.

destroy(Image::egd_image()) -> ok

Destroys the image.

filledEllipse(Image::egd_image(), P1::point(), P2::point(), Color::color()) -> ok

Creates a filled ellipse object.

filledRectangle(Image::egd_image(), P1::point(), P2::point(), Color::color()) -> ok

Creates a filled rectangle object.

line(Image::egd_image(), P1::point(), P2::point(), Color::color()) -> ok

Creates a line object from P1 to P2 in the image.

rectangle(Image::egd_image(), P1::point(), P2::point(), Color::color()) -> ok

Creates a rectangle object.

render(Image::egd_image()) -> binary()

Equivalent to *render(Image, png, [{render_engine, opaque}])*.

render(Image::egd_image(), Type::png | raw_bitmap) -> binary()

Equivalent to *render(Image, Type, [{render_engine, opaque}])*.

**render(Image::egd_image(), Type::png | raw_bitmap, Options::
[render_option()]) -> binary()**

Renders a binary from the primitives specified by egd_image(). The binary can either be a raw bitmap with rgb tripplets or a binary in png format.

save(Binary::binary(), Filename::string()) -> ok

Saves the binary to file.

**text(Image::egd_image(), P::point(), Font::font(), Text::string(),
Color::color()) -> ok**

Creates a text object.

percept

Erlang module

Percept - Erlang Concurrency Profiling Tool

This module provides the user interface for the application.

DATA TYPES

```
percept_option() = procs | ports | exclusive
```

Exports

analyze(Filename::string()) -> ok | {error, Reason}

Analyze file.

profile(Filename::string()) -> {ok, Port} | {already_started, Port}

See also: *percept_profile*.

profile(Filename::string(), Options::[percept_option()]) -> {ok, Port} | {already_started, Port}

See also: *percept_profile*.

profile(Filename::string(), MFA::mfa(), Options::[percept_option()]) -> ok | {already_started, Port} | {error, not_started}

See also: *percept_profile*.

start_webserver() -> {started, Hostname, Port} | {error, Reason}

Types:

Hostname = string()

Port = integer()

Reason = term()

Starts webserver.

start_webserver(Port::integer()) -> {started, Hostname, AssignedPort} | {error, Reason}

Types:

Hostname = string()

AssignedPort = integer()

Reason = term()

Starts webserver. If port number is 0, an available port number will be assigned by inets.

stop_profile() -> ok | {error, not_started}

See also: *percept_profile*.

stop_webserver() -> ok | {error, not_started}

Stops webserver.

percept_profile

Erlang module

Percept Collector

This module provides the user interface for the percept data collection (profiling).

DATA TYPES

```
percept_option() = procs | ports | exclusive
```

Exports

start(Filename::string()) -> {ok, Port} | {already_started, Port}

Equivalent to *start(Filename, [procs])*.

start(Filename::string(), Options::[percept_option()]) -> {ok, Port} | {already_started, Port}

Types:

Port = port()

Starts profiling with supplied options. All events are stored in the file given by Filename. An explicit call to stop/0 is needed to stop profiling.

start(Filename::string(), MFA::mfa(), Options::[percept_option()]) -> ok | {already_started, Port} | {error, not_started}

Types:

Port = port()

Starts profiling at the entrypoint specified by the MFA. All events are collected, this means that processes outside the scope of the entry-point are also profiled. No explicit call to stop/0 is needed, the profiling stops when the entry function returns.

stop() -> ok | {error, not_started}

Stops profiling.