RELEASE

SEVENTH FRAMEWORK PROGRAMME

ICT-287510
RELEASE
A High-Level Paradigm for Reliable Large-Scale Server Software
A Specific Targeted Research Project (STReP)

D5.2 (WP5): Online SD Erlang Profiling and Refactoring Tools

Due date of deliverable: 30th September 2013
Actual submission date: 14th October 2013

Start date of project: 1st October 2011 Duration: 36 months

Lead contractor: University of Kent Revision: 0.1

Purpose: To design and deliver an online visualisation tool to support the interactive exploration of the data from a system during execution; to extend the Wrangler refactoring tool to support the refactoring from Erlang/OTP to SD Erlang.

Results: The main results of this deliverable, reported here and in three published papers, are

- A prototype online visualisation tool to support the interactive exploration of runtime profiling data.
- Various improvements to the DTrace/SystemTap backend for offline and online profiling, as well as the design and implementation of DTrace/SystemTap probes for SD Erlang.
- Various improvements to the offline profiling tool Percept2.
- Extension of the Wrangler refactoring tool for refactoring from Erlang/OTP to SD Erlang.

Conclusion: We have delivered a first set of tools to support the online profiling for SD Erlang and refactoring from Erlang/OTP to SD Erlang.

Project funded under the European Community Framework 7 Programme (2011-14)			
Dissemination Level			
PU	Public		✳
PP	Restricted to other programme participants	(including the Commission Services)	
RE	Restricted to a group specified by the consortium	(including the Commission Services)	
CO	Confidential only for members of the consortium	(including the Commission Services)	

Online SD Erlang Profiling and Refactoring Tools

Contents

1 Executive Summary

This is what we said in the future work section of D5.1:

In the second year of the project we have developed tools for support of online monitoring and visualisation of SD Erlang programs, while at the same time enhancing the offline monitoring tools presented in D5.1. In particular we have

- Enhanced Percept2 so that it acts as a single front-end for the results of both Erlang trace and DTrace/SystemTap.
- Included further support for distribution in Percept2.
- Explicitly supported the constructs of SD Erlang in offline and online visualisation.
- Worked with experts in the field of data visualisation from the University of Kent's Computational Intelligence research group to develop innovative approaches to HTML5 for web-based graphical visualisation.

This work is reported in this document and in more detail in three published papers. The tools are available in from github as open source repositories.

Change Log

Version	Date	Comments
0.1	30/9/2013	First Version
0.2	7/10/2013	Revised in the light of reviewer comments

2 Introduction

At the heart of the RELEASE project is the development of SD Erlang, that uses Erlang distribution to handle large-scale multicore systems in Erlang. SD Erlang has been designed, and has been implemented in the second year of the project.

To build SD Erlang systems of any size, it is necessary to monitor and visualise the behaviour of these systems, and Work Package 5 of RELEASE is designed to do this. While many-core systems in SD Erlang will necessarily be distributed, it is also the case that they will contain single multicore processing elements that deploy Erlang. Monitoring and visualising SD Erlang will therefore necessarily require in-node observation as well as monitoring what goes on between nodes.

The deliverable D5.2 outlines the progress of the project in meeting this goals, and in particular describes in detail the innovations that have been made in *online monitoring and visualisation* for Erlang multicore and distributed systems, as well as refactoring support for refactoring Erlang programs using global_groups to programs using s_groups. These include a new online visualisation tool for the online monitoring of profiling data, the further development of the facilities for tracing by means of DTrace/SystemTap, the enhancement of Percept2 to support the constructs in SD Erlang, and to act as a single front-end for the results of both Erlang trace and DTrace/SystemTap, and finally the extension of Wrangler, the Erlang refactoring tool, to support the migration from Erlang/OTP to SD Erlang.

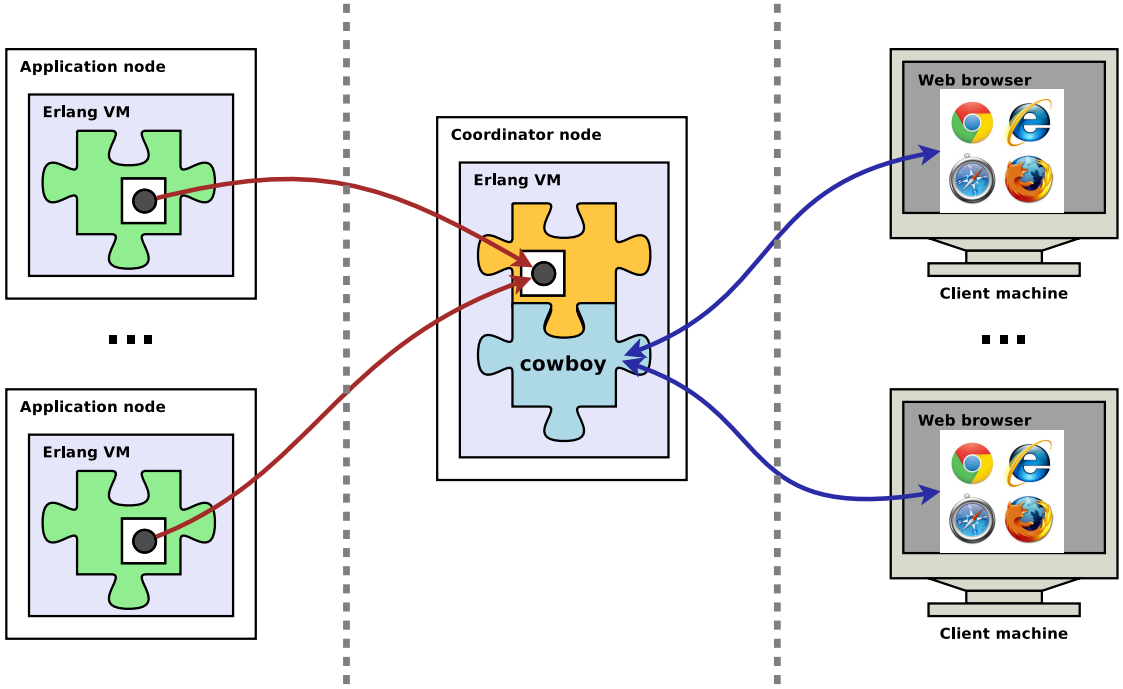

Figure 1: Online profiling of a distributed Erlang application using the Erlang tracing mechanism.

3 Overall Architecture of Online SD Erlang Profiling

SD Erlang applications are expected to run on several Erlang nodes, distributed over a network of computers. Scalable online profiling of such applications, when the number of Erlang nodes increases, is a real challenge for the RELEASE project, because of the vast amount of profiling data that have to be collected and appropriately visualized in a short time period, real-time.

On the other hand, making things a bit easier for online profiling in comparison to offline profiling, there is a limit to the amount of profiling information that is useful for viewing online. Typically, online profiling tools focus on a number of characteristic measures and quantities of the system under surveillance, such as the utilization of application nodes, the number of processes in run queues, the amount of information exchanged through the network, etc. In this respect, online profiling tools have an easier job than offline profiling tools, as the size of profiling data that need to be collected is significantly smaller. This in general alleviates the need for having multiple nodes collecting this profiling information, which is absolutely required for the offline case, and allows us to implement an overall architecture such as the one that we are presenting in this section. If the size of profiling data that needs to be collected is larger than what we now expect, the central coordinator node can be replaced by a hierarchical network of coordinator nodes, in exactly the same way as proposed for offline profiling.

The overall architecture of online profiling tools suitable for SD Erlang applications is given in Figures 1 and 2. The two figures differ only slightly, in the back-end that is used for collecting the profiling information: the Erlang tracing mechanism (Figure 1) and DTrace/SystemTap (Figure 2). We begin by describing the common parts and we will come to the differences when they become important.

The overall architecture for the online profiling of an SD Erlang application is structured in three sections, clearly separated in both figures by vertical dashed lines.

- The left section is the environment where the SD Erlang application is executed. It consists of one or more Erlang VMs, running on one or more computing devices that we call *application nodes*. Each Erlang VM runs a part of the SD Erlang application.

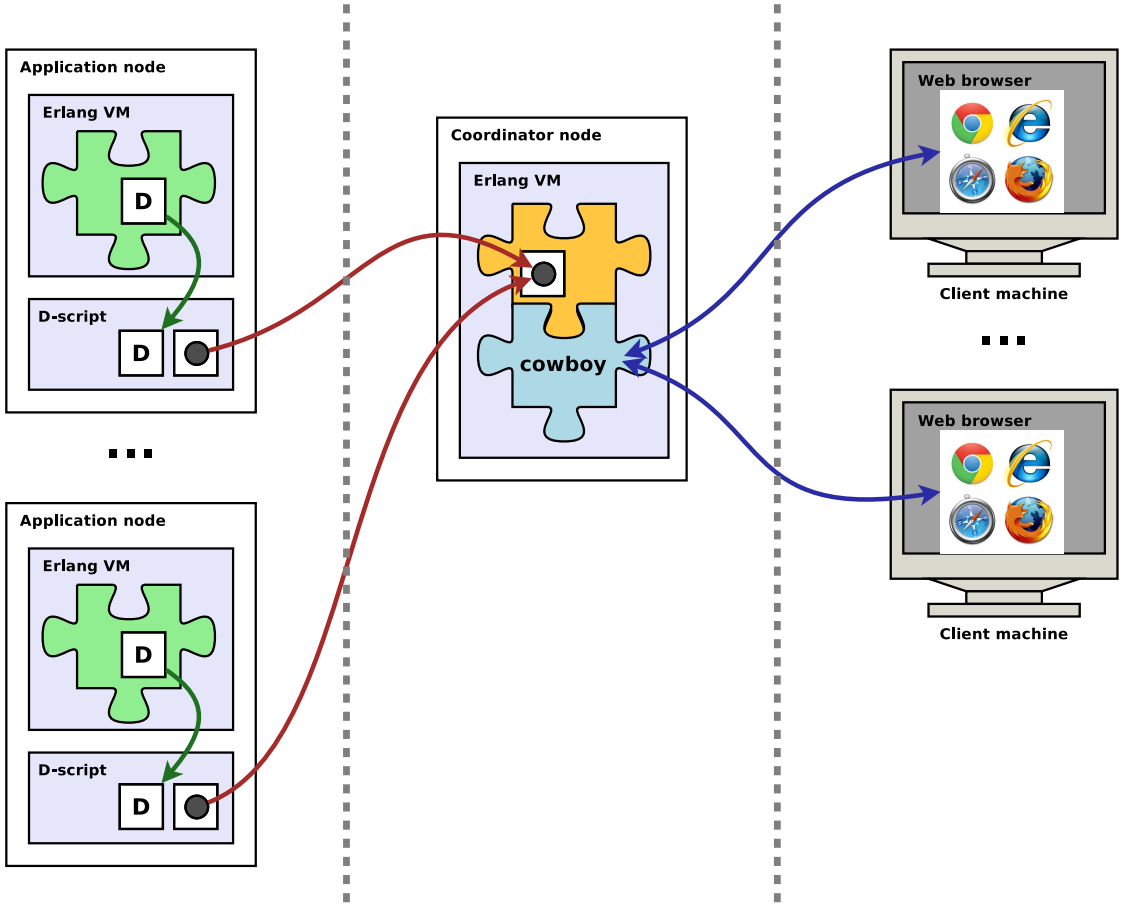

Figure 2: Online profiling of a distributed Erlang application using DTrace/SystemTap.

- The middle section is the environment where the coordination of the entire profiling process takes place. The job of the *coordinator node* is to gather profiling data from all application nodes, over the network, and to store them in an appropriate database. Furthermore, the coordinator node hosts a web server that will be used for the visualisation of profiling information.

  The coordinator node requires four pieces of software: one that coordinates the process, one that receives profiling data, the database where these are stored, and the web server. All four can be written in Erlang and run in a single Erlang VM, as suggested in the figures. The web server suggested for this purpose is `Cowboy` [cow].

- The right section is the environment that hosts the *clients* of the online profiling tool. There can be several such clients, each one connecting to the coordinator node through the web server, in order to retrieve and visualise the information that is needed.

The coordinator node is responsible for coordinating the gathering of profiling data. It should communicate with the application nodes and direct them to start collecting (or stop collecting) the information that is requested to be visualised by the clients. It is also responsible for specifying various parameters that control the gathering of information, such as the polling frequency.

As in the architecture suggested for offline profiling tools, which is reported in the revised deliverable D.5.1, there are two alternative back-ends for generating the profiling information: one using the Erlang tracing mechanism, which is built in the Erlang/OTP, and one using DTrace/SystemTap, a separate light-weight dynamic tracing framework that is available for Unix/Linux machines and nicely integrates

with recent Linux kernels ($\geq 3.5.x$).

When the back-end that is based on the Erlang tracing mechanism is used, the Erlang VMs running on the application nodes communicate directly with the Erlang VM running on the coordinator node. This is implemented by sending messages to a dedicated process on the coordinator node. On the other hand, when the DTrace/SystemTap back-end is used, execution of the SD Erlang application causes probes to be fired on the application nodes. These probes are perceived by D or SystemTap scripts that are executed on the application nodes with the use of Erlang ports. The profiling data that are collected by the scripts are sent, through the corresponding Erlang port, to a dedicated process on the coordinator node.

4 Offline and online visualisation of Profiling Data

Developing tools for support of online monitoring and visualisation of SD Erlang programs is one of the tasks in WP5, so we have prototyped a web-based online monitoring and visualisation system for concurrent and distributed Erlang applications. We give an overview of the tool in this report, more details about the techniques used to support visualisation are described in the paper *Multi-level visualisation of Concurrent and Distributed Computation in Erlang* [RBL13].

Like Percept2, the online monitoring and visualisation tool has a web-based interface. It makes use of `Cowboy` [cow] as the web server, and in particular cowboy's support for websocket to connect the visualisation/monitoring on the client side with the live trace/profile data sent to the server side. Both event-based tracing and sampling techniques are used to collect live data from running nodes. Sampling is used to collect data that is not accessible through Erlang's built-in trace, such as the lengths of run queues in an Erlang node. An Erlang node is needed to host the web server; this node is also used as a 'trace control node' to start/stop the tracing/sampling on remote nodes, and as the receiver of live data from remote nodes. To reduce the traffic between the web client and the web server, live data is aggregated over a time interval, say 100 milliseconds, before being sent to the client.

Our tool is able to provide two granularities of view of a running Erlang system: low level and high level.

The low level view shows the concurrent behaviour of Erlang processes on a single Erlang node, focussing in particular on scheduling aspects. The schedulers are laid out in a circle: we indicate the size of their run queues by the length of bars which are drawn next to the schedulers. We also visualize the migration of Erlang concurrent processes from one scheduler to another, as work is redistributed to fully exploit the hardware: this migration is represented by animated edges between schedulers.

An example snapshot of low level visualisation in action is shown in Fig 3. In this example, the parallel machine is split into two six-core processors, represented by the two pink semicircles. Erlang schedulers are shown as a number that represents the unique ID of the related threads. Pairs of Erlang schedulers that run on a single real core are shown grouped by the blue ellipses. Each individual run queue is represented by a bar radiating from the visualisation. The height and colour are used to indicate the size of the run queue. The run queue size can be displayed on screen as a value by selecting the relevant option in the interface, or by hovering a cursor over the bar or Erlang scheduler in question. The list of numbers radiating from a Erlang scheduler shows the unique ID for each process spawned on that scheduler.

Process migration from one scheduler to another is shown by edges between various schedulers. These are curved to avoid occlusion with other schedulers. The edges are coloured to indicate the three types of migration: within a single core (green), between different cores on the same processor (grey), and from a core on one processor to a core on the other (blue). These edges fade away after a few seconds to prevent the visualisation from becoming occluded.

The high level view visualizes the distributed aspects of the system, with each graph node representing an Erlang node. We show message passing between nodes as edges, laying nodes out according

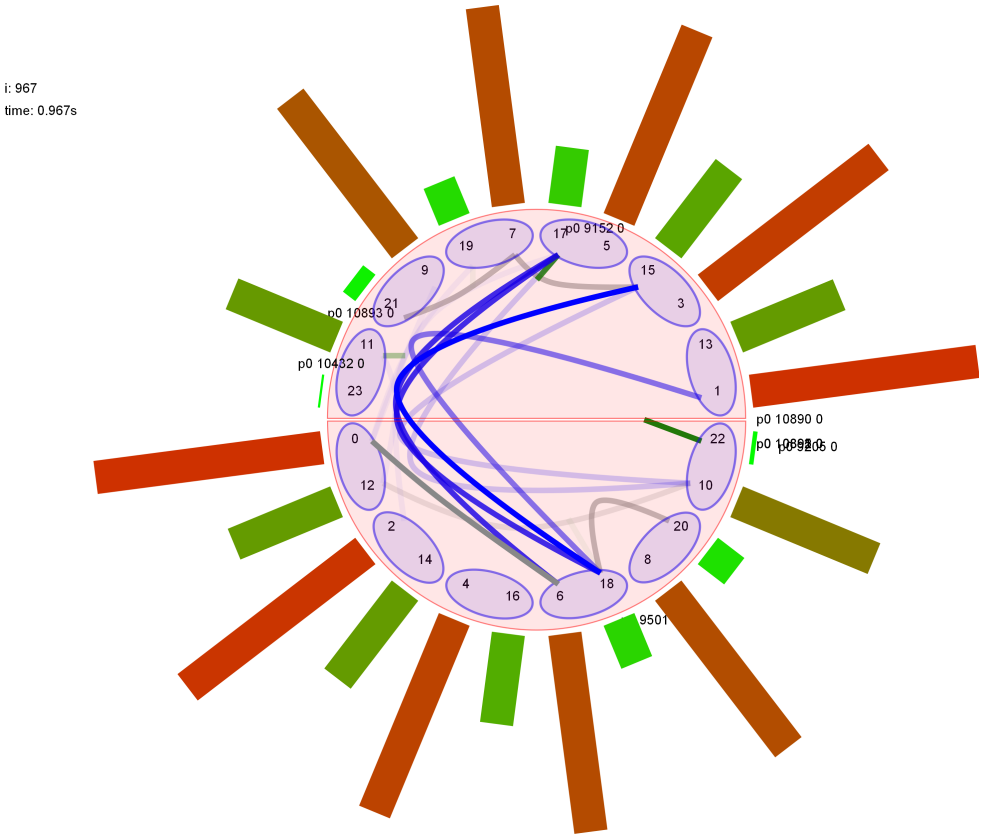

Figure 3: Low Level visualisation

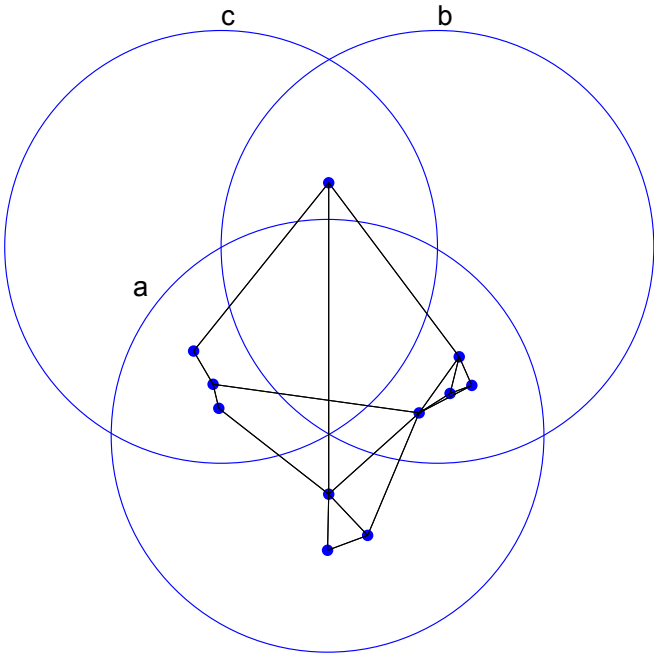

Figure 4: High Level visualisation

to their current connections. The grouping of nodes into s_groups is shown by surrounding each s_group by an interlinking circle.

An example snapshot of the high level visualisation is shown in Fig 4. In which, each s_group is represented by a circle. As a node may be the member of more than one s group, the circles typically intersect. A force-directed layout is then applied to the nodes to im- prove the layout. Communication between nodes is represented by edges and these may change during the programs execution. The shading of the edges is related to the frequency of messages within the timeframe. The darker the edge, the more frequently messages are sent. Edges are removed if no communication exists between them and new edges are introduced when communication is initiated between nodes. The use of a force directed layout method allows the visualisation to be dynamic, so that the repositioning of the nodes can be shown in an animated manner as the diagram reaches a new equilibrium.

Downloading and using the systems

The offline visualisation tool is available from `https://github.com/RefactoringTools/percept2`, with the relevant code contained in

`https://github.com/RefactoringTools/percept2/tree/master/visualizations/low`
`https://github.com/RefactoringTools/percept2/tree/master/visualizations/high`

respectively.

Low-level visualisation

To run the low level offline visualization, navigate to `parse.html` in your HTML5 compliant browser.

There, files must be uploaded for the visualization to run and these must be in the required format.

Example of run queue migration (see `rq_migration_by_time.txt`):

```
{0.004082,{pid,{0,9011,0}},4}.
{0.004279,{pid,{0,24,0}},1}.
{0.004293,{pid,{0,22,0}},1}.
{0.00431,{pid,{0,34,0}},1}.
{0.006194,{pid,{0,17,0}},1}.
{0.017307,{pid,{0,17,0}},4}.
{0.021344,{pid,{0,17,0}},1}.
{0.023292,{pid,{0,34,0}},4}.
{0.026142,{pid,{0,17,0}},4}.
```

Example of run queue size (see `sample_run_queues.txt`):

```
0.0   0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0.138095   2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  1
0.141939   0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0.143956   0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0.145075   1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
0.147073   0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0.149071   0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
```

[This information is also contained in the `README.md` file.]

High-level visualisation

To run the offline high level visualization, there are several requirements:

All files need to be stored in a webserver, specifically one that can run PHP. The webserver also needs Java 1.6 or greater.

Once these requirements are met, navigate to `regions.html` in your HTML5 compliant browser.

It requires three files in various formats, and these formats must be exact.

Example of sgroup format (see `sgroup.txt`):

```
a b c ab bc ac abc
```

Example of node format (see `nodes.txt`):

```
node1 abc
node2 a
node3 ab
node4 ab
node5 b
node6 bc
```

Example of communication format (see `inter_node_sum.txt`):

```
{200,
 [{{node1,node11},1,240},
  {{node10,node10},3,152},
  {{node5,node5},3,150},
  {{node2,node2},22,46589}]}.
{400,
 [{{node1,node11},2,318},
  {{node1,node5},11,1091},
  {{node5,node5},19,46439}]}.
```

[This information is also contained in the `README.md` file.]

Dynamic Erlang visualisation online: `devo`

The online visualisation tool, `devo`, is available from `https://github.com/RefactoringTools/devo`. It is used as follows.

To compile the code you need `rebar` in your `PATH`.

Type the following command:

```
$ make
```

You can then start the online visualisation from an Erlang node with the following command:

```
devo:start().
```

Then point your browser to the indicated URL to open the Percept2 online visualisation page.
To stop the application, type:

```
devo:stop().
```

At the time of writing, the tool supports low-level visualisation only; the high-level visualisation will also be available from the end of October 2013.

[This information is also contained in the `README.md` file.]

5 Further Development on Erlang DTrace/SystemTap

This section presents the results related to the DTrace/SystemTap back-end, both for offline profiling (using Percept2) and for online profiling and monitoring. It also reports on the design and implementation of DTrace/SystemTap probes for SD Erlang.

5.1 DTrace/SystemTap back-end for the offline profiling tool Percept2

In the first year of the project, we designed, implemented and delivered the first version of a new back-end for *Percept* that uses DTrace or SystemTap to collect the information that *Percept*'s current back-end collects using Erlang built-in tracing and profiling mechanism. This section describes our efforts during the second year to enhance this back-end, as well as the results of these efforts.

We first re-designed our back-end, so that it can be used to profile distributed applications. The revised architecture is shown in Figure 7 of the revised deliverable D.5.1. As soon as the profiling of a distributed application starts, a process is spawned on the coordinator node. The coordinator process starts tracing all the involved application nodes, collects the tracing and profiling information from each one of these nodes, eventually stops tracing the nodes, and writes the collected information to a trace file.

In order to start tracing a specific application node, the coordinator process needs to remotely start the execution of a D or a SystemTap script on that node. The script is executed on the application node through an Erlang port, and the information that it collects is sent back to the coordinator process. In order to stop tracing an application node, the coordinator process needs to remotely stop the execution of the script (i.e., close the port).

The users can specify the events that they need to trace. There is a number of profile and trace flags that they can specify for this purpose. Each one of these flags is mapped to a set of DTrace probes that need to be enabled. So, based on the specified profile and trace flags, the D or SystemTap script, which is to be executed on the application nodes, is generated dynamically.

While profiling an application, the user can at any point start tracing one or more application nodes, stop tracing one or more other nodes, or modify the events that they need to trace.

Finally, we have extended our back-end, so that it can serve as a back-end not only for *Percept*, but also for its enhanced version, *Percept2*. In order to do that, we had to support a number of extra trace and profile flags.

5.2 DTrace/SystemTap back-end for online profiling and monitoring tools

Section 4 describes the prototype of a web-based online monitoring and visualisation system. Information is collected from one or more application nodes using the Erlang built-in tracing mechanism and other sampling techniques, and is fed to the system, in order to be visualized. This section presents an alternative back-end for this system that collects the same information using DTrace or SystemTap.

The new back-end was designed based on the architecture described in Section 3. One or more application nodes are traced. On each node that needs to be traced, a D or SystemTap script is executed (using Erlang ports). The script that needs to be executed on each application node is generated dynamically, based on the events that need to be traced.

On the coordinator node, there is a process that acts as the tracing coordinator. More specifically, the following tasks are assigned to this process:

- to start tracing a node;
- to stop tracing a node; and
- to collect information from a traced node and to forward this information to the collecting process (which is the same for both back-ends) in the expected format.

As mentioned in Section 4, the events that are currently traced by the prototype tool are process migrations, run queue sizes, inter-node message passing and s_group-related events. In order to support the first two and the fourth type of events, we had to define new probes. A summary of the probes for the first two types of events is given below. For each probe, the description contains the event that causes it, the names and the types of its parameters, and the kind of information that is passed to them. The probes that we added to support the tracing of s_group-related events are described in Section 5.3.

Probe: run_queue-create

Fired: whenever a run queue is created

Header:

```
probe run_queue__create(int rqid)
```

Parameters:

- `rqid`: the ID of the run queue

Probe: run_queue-enqueue

Fired: whenever a process is added to a run queue

Header:

```
probe run_queue__enqueue(char *p, int priority, int rqid, int rqsize)
```

Parameters:

- `p`: the PID of the process
- `priority`: the priority of the process
- `rqid`: the ID of the run queue
- `rqsize`: the size of the run queue (after the addition)

Probe: run_queue-dequeue

Fired: whenever a process is removed from a run queue

Header:

```
probe run_queue__dequeue(char *p, int priority, int rqid, int rqsize)
```

Parameters:

- `p`: the PID of the process
- `priority`: the priority of the process

- `rqid`: the ID of the run queue
- `rqsize`: the size of the run queue (after the removal)

Probe: process-migrate

Fired: whenever a process migrates from one run queue to another

Header:

```
probe process__migrate(char *p, int priority, int fromrqid, int fromrqsize,
                       int torqid, int torqsize)
```

Parameters:

- `p`: the PID of the process
- `priority`: the priority of the process
- `fromrqid`: the ID of the run queue, from which the process was removed
- `fromrqsize`: the size of the run queue, from which the process was removed (after the removal)
- `torqid`: the ID of the run queue, to which the process was added
- `torqsize`: the size of the run queue, to which the process was added (after the addition)

By adding new DTrace probes in order to export from the runtime system information that was not supported by the existing probes (e.g., the run queue sizes), we avoided the use of any sampling techniques.

5.3 DTrace probes for SD Erlang

SD Erlang has been essentially designed as a tweaked version of Erlang's *kernel* application. This leads to the observation that tracing specific events that are related to s_groups implies tracing calls of specific Erlang functions. So, in order to trace s_group-related events using DTrace or SystemTap, we had two options.

Our first option was to use the `global__function__entry` probe, and to use our D or SystemTap script to filter out any irrelevant function calls.

Probe: global-function-entry

Fired: whenever an external function is called

Header:

```
probe global__function__entry(char *p, char *mfa, int depth, uint64_t ts)
```

Parameters:

- `p`: the PID of the caller
- `mfa`: the MFA for the called function
- `depth`: the stack depth
- `ts`: the timestamp (in microseconds)

Our second option was to use one or more of the 951 `user_trace__n*` "user" probes (`user_trace__n0`, ..., `user_trace__n950`). The integer that follows n in the name of the probe is the *probe number*. These probes can be triggered from inside the `s_group` module, whenever something interesting happens (e.g., a new s_group is created). The user tag for all these probes is set to "s_group".

Event	Probe number	Arguments
Creation s_group	0	s1: the group name
Deletion of s_group	1	s1: the group name
Addition of a node into an s_group	2	s1: the group name, s2: the node name
Removal of a node from an s_group	3	s1: the group name, s2: the node name

Table 1: Correspondence between s_group events and probe numbers.

Probe: user_trace-n0

Fired: whenever any of the pn functions of the dyntrace module is invoked with 0 as its first argument

Header:

```
probe user_trace__n0(char *proc, char *user_tag, int i1, int i2, int i3, int i4,
    char *s1, char *s2, char *s3, char *s4)
```

Parameters:

- `proc`: the PID of the process that fired the probe
- `user_tag`: a user tag
- `i1`: an integer
- `i2`: an integer
- `i3`: an integer
- `i4`: an integer
- `s1`: a string
- `s2`: a string
- `s3`: a string
- `s4`: a string

We finally chose to take advantage of the "user" probes, and moreover to connect each type of s_group-related events that we are interested in to one such probe. The correspondence between s_group events and probe numbers, as well as the information that these probes are expected to carry are shown in Table 1.

Note that based on the above table, the call of an s_group function might cause more than one probe to be fired. For example, a call to the `s_group:new_s_group/2` will fire one `user_trace-n0` probe for the creation of the new s_group, and one more `user_trace-n2` probe for each one of the nodes that are contained in the list that is passed as a second argument to this function call.

The advantages of using "user" probes to trace s_group events are that it does not involve any irrelevant probe firings, which need to be ignored, and that it allows us to have an even more fine-grained control on the s_group events we want to trace (e.g., we can trace only creations of s_groups). On the other hand, our approach has the disadvantage that, since at the moment there is no way to reserve a specific probe label, anyone can end up using the same probe number to trace some other type of events.

```
-type profile_option()::procs  | ports  | schedulers
                        |running| message| migration
                        |garbage_collection
                        |s_group
                        |all
                        |{callgraph, [module_name()]}.
```

Figure 5: Profiling Options

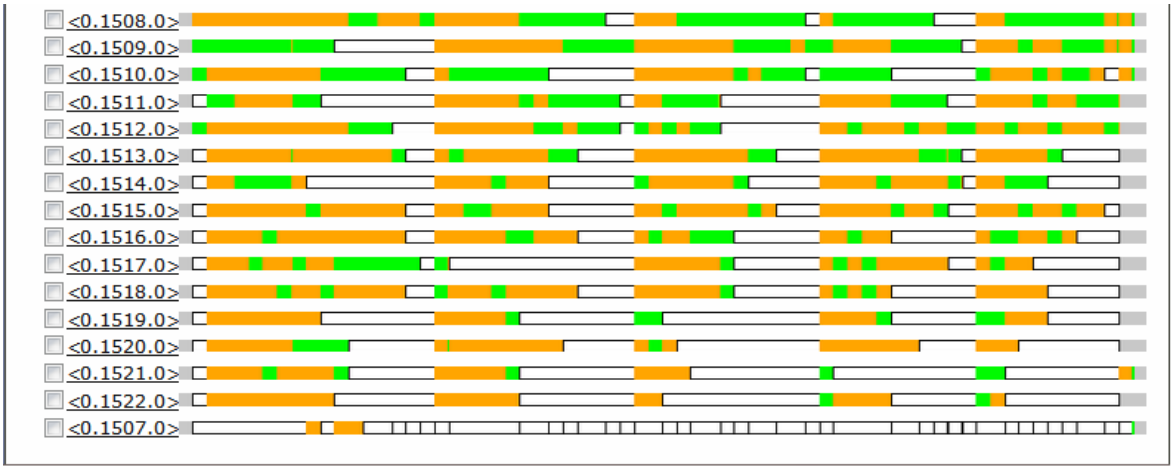

Figure 6: Percept2: process runnability comparison

6 Enhanced Percept2

We have further developed Percept2, downloadable from `https://github.com/RefactoringTools/percept2`, after its first release in month 12. Various aspects of Percept2 have been improved, and some new features have been added. More details about the tool is reported in the paper *Multicore Profiling for Erlang Programs Using Percept2* [LT13]; here is a summary of the major enhancements made to Percept2 in the last year.

- User-command interface improved to allow the profiling of a particular aspect of the execution. The user-command interface of Percept2 has been further refined to allow a more flexible control over what to profile. As shown in the type definition of `profile_option` in Fig 5, the profiling of port activities, schedulers activities, message passing, process migration, garbage_collection and s_group activities can be enabled/disabled, while the profiling of process runnability (indicated by the 'proc' flag) is always enabled. The profiling of calls to functions defined in a particular module is enabled if the 'callgraph' flag together with the module name(s) is supplied.

- Distinguish between running and runnable time for each process. Percept2 distinguishes between process states of *running* and *runnable* (i.e. the process is ready to run, but another process is currently running). This is achieved by enabling the tracing of process scheduling events. As a result, we are able to calculate the accumulated runtime of a process more accurately.

  The distinction between *running* and *runnable* is now also reflected in the process runnability comparison as shown in Fig 6, where *orange* represents *runnable* and *green* represents *running*.

- Selective function profiling of processes. In Percept2, we have built a version of `fprof`. Compared to `fprof`, Percept2's support for function profiling does not measure a function's own

execution time, but measures everything else that `fprof` measures. Eliminating measurement of a function's own execution time gives a user the freedom of not profiling all the function calls invoked during the program execution. For example, they can choose to profile only functions defined in the user's application code, and not those in libraries.

To further reduce the impact on program execution time and the size of trace data, a user can replace the uses of standard `spawn/spawn_link` functions in the application code with the `spawn/spawn_link` functions provided by Percept2, so that Percept2 could selectively trace those processes that exhibit a common behaviour, i.e. function activities are recorded for a processes only if it has been selected to be traced by Percept2. The selectivity only applies to function activities, and does not affect the tracing of other process activities in any way.

- Improved dynamic function callgraph. With the dynamic function callgraph, a user is able to understand the causes of certain events, such as heavy calls of a particular function, by examining the region around the node for the function, including the path to the root of the graph. Each edge in the callgraph is annotated with the number of times the target function is called by the source function. Latest additions to the callgraph are that each node in the callgraph is annotated with the time the function took as a percentage of the process's life time, as well as nodes for pseudo functions `suspend` and `garbage_collect` indicating the time a function spends on suspension and garbage collection respectively.

- Process communication graph. Support for the static graph visualisation of process-to-process message passing has been added to Percept2. As shown in Fig 7, each node in this graph represents a process, and the label on the link from one process to another indicates the total number of messages sent from the originating node to the target node, and the average size of the messages sent. The graph can be simplified by increasing the threshold values as shown at the top of the figure, so that one can focus on those processes with heavy communication.

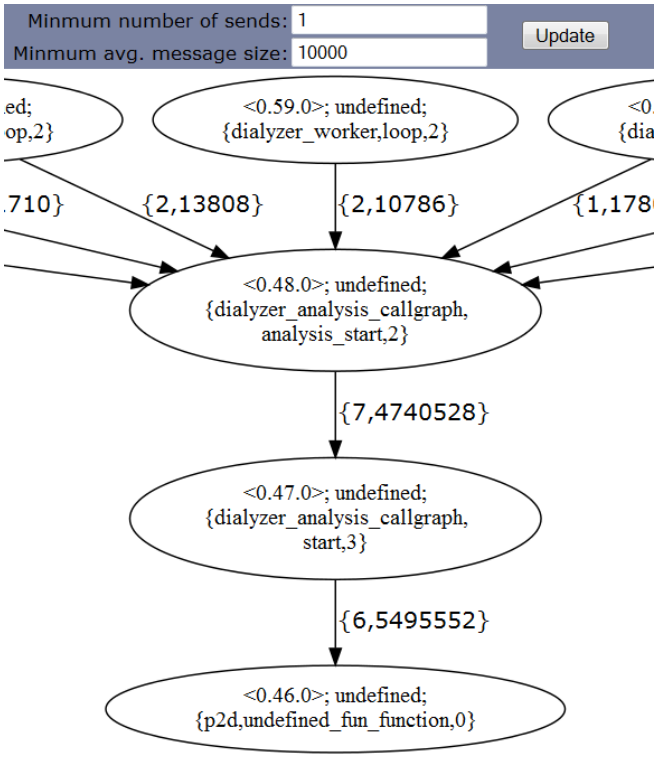

Figure 7: Percept2: process communication graph

Information about inter-scheduler message passing can be derived from inter-process message passing events and process scheduling events, i.e. which process is scheduled to run and on which scheduler.

- Inter-node message passing. The message passing between nodes in a distributed system can be profiled using Percept2, and an offline replay of the communication between nodes is supported by our offline web-based graphical visualisation tool developed by experts in the field of data visualisation from the University of Kent's Computational Intelligence research group. How the tool works has been briefly described in Section 4, more details can be found in the paper *Multi-level visualisation of Concurrent and Distributed Computation in Erlang*[RBL13].

- Tracing of `s_group` activities in a distributed system. Unlike `global_group`, `s_group` allows dynamic changes to the s_group structure of a distributed Erlang system. In order to support SD Erlang, we have also extended Percept2 to allow the profiling of `s_group` related activities, so that the dynamic changes to the s_group structure of a distributed Erlang system can be captured. For this purpose, we added to Percept2 the profiling of 4 `s_group` functions, i.e. `s_group:new_s_group/2`, `s_group:delete_s_group/1`, `s_group:add_nodes/2` and `s_group:remove_nodes/2`. These functions are traced only if the `s_group` flag is supplied when the profiling is started. For each function call traced, its arguments are also recorded. The trace events collected, together with the initial s_group configuration if there is any, can be used to generate an offline replay of the s_group structure evolution, as well as the online visualisation of the current s_group structure, of an Erlang system.

Downloading and using the system

The system is available from

`https://github.com/RefactoringTools/percept2`

and the software is built and installed in the usual way:

```
$ ./configure
$ make
$ (sudo) make install
```

By default Percept2 is installed under the directory `/usr/local`; to install Percept2 in a different directory, you need to explicitly specify the directory using the `'--prefix=...'` flag of `'configure'`.

NOTE: percept2 uses the 'graphviz' tool to generate the graph representation of process tree and dynamic function callgraph. While we are trying to remove this dependence, the current version does require it.

7 Refactoring Support for SD Erlang

SD Erlang replaces Erlang's original `global_group` library with a new `s_group` library. As a result, Erlang programs using `global_group` will have to be refactored to use `s_group`. This kind of API migration problem is not uncommon, as software evolves and this often changes the API of a library.

While we could have extended Wrangler, the Erlang refactoring tool, with a number of refactorings especially for the migration from `global_group` to `s_group`, we thought that the Erlang community would benefit more if our approach could support API migration in general. For this reason, we have extended Wrangler with a framework for the automatic generation of API migration refactorings from a user-defined adaptor module.

Our approach to automatic API migration works in this way: when an API function's interface is changed, the author of this API function implements an *adaptor function*, defining calls to the old API in terms of the new. From this definition we automatically generate the refactoring, that transforms the client code to use the new API. This refactoring can be supplied by the API writer to clients on library upgrade, allowing users to upgrade their code automatically.

An *adaptor* function is a single-clause function that implements the 'old' API function using the 'new' API. A number of constraints should be satisfied by adapter functions:

- The definition should have only one clause, and the name/arity should be the same as the 'old' function.
- The parameters of the function should all be variables.
- If the function definition is a `case` expression, then the last expression of every clause body should be a simple expression that can be used as a pattern.

For example, the adaptor functions for `global_group:global_groups/0` and `global_group:send/2` are shown in Fig 8.

```
%%   Example (a): The adapter function for global_group:global_groups/0

global_groups() ->
  case s_group:s_groups() of
       {[OwnGroupName], GroupNames} ->
            {OwnGroupName, GroupNames};
        undefined -> undefined
  end.

%%    Example (b): The adapter function for global_group:send/2

send(Name, Msg) ->
    case s_group:send(hd(s_group:own_s_groups()), Name, Msg) of
        {'badarg', {_GrpName, Name, Msg}} ->
            {'badarg', Name, Msg};
        Pid -> Pid
    end.
```

Figure 8: Adapter Function Examples

Once an adapter module has been defined, a rule generator takes the adapter module as input, and generates a number of rules and meta_rules (at most 3, typically) for each adaptor function defined. For instance, Fig 9 shows one of the rules generated for migrating from `global_group:global_groups/0` to `s_group:s_groups/0`.

The rule- and template-based API for program transformation and analysis is part of Wrangler extension framework. The API allows programmers to express program analysis and transformation in Erlang concrete syntax. The API has been used by Wrangler authors and others to define a collection of refactorings/program transformations so far. More details about the API are described in [LT11].

API migration refactorings are a special kind of refactorings, whose preconditions are always met. The way in which the refactoring rules are applied is also different from the way in which general rule-based refactorings are applied. As a matter of fact, the API migration process is a combination of rule application and refactoring. Rule application makes sure the old API is replaced by the new API, and refactoring steps are to avoid generation of expressions that are too complex. In order to keep the

```
?META_RULE(?T("case global_group:global_groups() of
                  {OwnGroupName@,GroupNames@} when Guard1@@ -> Body1@@;
                  undefined when Guard2@@ -> Body2@@
              end"),
           api_refac:anti_quote(refac_api_migration:mk_str(
             "case s_group:s_groups() of
                {[OwnGroupName@],GroupNames@} when Guard1@@ ->
                    Body1@@;
                undefined when Guard2@@ ->
                    Body2@@
             end", [])),
           (api_refac:free_vars(Guard1@@) --
             (api_refac:bound_vars(GroupNames@) ++
                api_refac:bound_vars(OwnGroupName@))
             ==
           api_refac:free_vars(Guard1@@))).
```

Figure 9: A migration rule for migrating `gobal_groups:global_groups/0`

code it generates as tidy as possible, refactorings such as *introduce a new variable*, *removal of unused expressions/variables* are applied after a rule has been applied.

As a design principle, we try to limit the scope of changes as much as possible, so that only the places where the 'old' API function is called are affected, and the remaining part of the code is unaffected. One could argue that the migration can be done by *unfolding* the function applications of the old API function using the adaptor function once it is define, however, the code produced by this approach would be a far cry from what a user would have written. Instead, we aim to produce code that meets users' expectation.

More details about Wrangler's support for API migration is reported in the paper *Automated API Migration in a User-Extensible Refactoring Tool for Erlang Programs* [LT12], in which a more complex API migration example, i.e. the migration from `regexp` to `re`, is studied. Wrangler is downloadable from `https://github.com/RefactoringTools/wrangler`.

8 Future Plans

In the next period we will continue to work on these tools in the following ways.

- Develop exemplars and training materials that clearly explain the typical usage and workflow for users of Percept2 and the online visualisation tools.
- These materials will include descriptions of how the tools can be used in a situation where there is the potential for vast amounts of data to be generated, and so the data needs to be throttled and generated selectively.
- The WP5 team will liaise with the ESL team to integrate the SD Erlang visualisation tool with the Wombat deployment and monitoring system, and in particular we will use the data gathering aspects of Wombat to drive the SD Erlang visualisations.
- Looking towards D5.5, integrate offline and online motoring information with debugging systems.

9 Conclusions

This report describes the second deliverable in the fifth work package of the RELEASE project, and delivers a number of tools and mechanisms supporting the online visualisation of SD Erlang many/-multicore systems, as well the refactoring from Erlang/OTP to SD Erlang.

References

[cow] Cowboy - A small, fast, modular HTTP server. `http://ninenines.eu/`.

[LT11] Huiqing Li and Simon Thompson. A User-extensible Refactoring Tool for Erlang Programs. Technical Report 4-11, School of Computing, Univ. of Kent, UK, 2011.

[LT12] Huiqing Li and Simon Thompson. Automated api migration in a user-extensible refactoring tool for erlang programs. *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, 0:294–297, 2012.

[LT13] Huiqing Li and Simon Thompson. Multicore Profiling for Erlang Programs Using Percept2. In *Proceedings of the Twelfth ACM SIGPLAN Erlang Workshop*, Boston, USA, September 2013.

[RBL13] Simon Thompson Robert Baker, Perter Rodgers and Huiqing Li. Multi-level Visualization of Concurrent and Distributed Computation in Erlang. In *Visual Languages and Computing (VLC) in The 19th International Conference on Distributed Multimedia Systems (DMS 2013)*, Brighton, UK, August 2013.