# Multicore Profiling for Erlang Programs Using Percept2

Huiqing Li
Computing Laboratory, University of Kent, UK
H.Li@kent.ac.uk

Simon Thompson
Computing Laboratory, University of Kent, UK
S.J.Thompson@kent.ac.uk

## Abstract

Erlang is a functional programming language with built-in support for concurrency based on share-nothing processes and asynchronous message passing. The design of Erlang makes it suitable for writing concurrent and parallel applications, taking full advantage of the computing power of modern multicore machines. However many existing Erlang applications are sequential, in need of parallelisation.

In this paper, we present the Erlang concurrency profiling tool Percept2, and demonstrate how the information provided by it can help the user to explore the potential parallelism in an Erlang application and how the system performs on the Erlang multicore system. Percept2 thus allows users improve the performance and scalability of their applications.

***Categories and Subject Descriptors*** D.2.3 [*SOFTWARE ENGINEERING*]: Coding Tools and Techniques; D.2.6 []: Programming Environments; D.2.7 []: Distribution, Maintenance, and Enhancement; D.3.2 [*PROGRAMMING LANGUAGES*]: Language Classifications—Applicative (functional) languages; Concurrent, distributed, and parallel languages; D.3.4 []: Processors

***Keywords*** Erlang, multicore, profiling, tracing, parallelisation.

## 1. Introduction

The demise of Moore's Law has shifted programmers' attention to parallelism in order to improve the performance and scalability of an application when the language in which the application is written supports multicore programming. Among such languages, Erlang [7, 10] has been gaining popularity recently, largely because of its lightweight concurrency model, in which share-nothing processes communicate through asynchronous message passing.

Erlang allows programmers to express concurrent processes and communications in a concise way. A process executing a function is created using the `spawn` primitive with the function name and its arguments as arguments; message passing between processes is expressed as `Pid!Message`, where `Pid` is the process identifier of the target process and `Message` is the message being sent.

When an Erlang Virtual Machine (VM) with symmetric multiprocessing (SMP) support is started, by default it creates a scheduler for each CPU core available. Each scheduler has its own process run-queue, and handles the scheduling of processes in its run-queue, independent of other schedulers. A process migration mechanism is applied periodically to balance the workload between schedulers. Erlang applications, especially those that are CPU-intensive, scale very well in a multicore environment with SMP enabled [9, 22].

Although many concurrent Erlang applications have been developed, less effort has been made to develop parallel Erlang applications. One of the reasons for this is that SMP support was only added to Erlang less than ten years ago, and it has taken a few years since then for the system to mature. Without SMP, it is still possible to write concurrent programs, in which case each Erlang process would have its own slice of time to run; however to get parallelism on a multicore computer, it would instead be necessary explicitly to start multiple instances of the Erlang VM. Nowadays there is still a lack of tools that allow users to develop such applications by exploring the potential parallelism in an Erlang application. We have therefore developed *Percept2*, an enhanced version of the Erlang concurrency profiling tool, Percept [11].

We have given the tool a new name instead of a new version number because the original Percept tool is part of the standard Erlang distribution, whereas Percept2 is an evolution of it, and whether or nit it is going to replace Percept in the distribution is not under our control. Giving them different names also allows the continued evolution and development of both tools.

Percept is a tool for offline visualisation of Erlang application level concurrency and identification of concurrency bottlenecks. It utilises Erlang's built-in support for tracing to monitor events from process states. Trace events are collected and stored in a file, which can then be analysed offline. Once the analysis is done, the data can be viewed through a web-based interface.

Percept is able to give a picture of application-level parallelism, as well as how much time processes spend waiting for messages, but it does not provide any core/scheduler-related information, nor does it support the process of parallelising existing sequential programs. In `Percept2`, we aim to meet these goals:

- Expose scheduler-related activities to end-users. These include message communication between schedulers, process migration from one scheduler to another, scheduler activity, etc.
- Provide finer-grained profiling information about the existing parallelism of an application, and help the user to explore the potential parallelism.
- Parallelisation increases the number of processes and therefore the number of events traced in a given period of time. Percept2 should therefore scale well to parallel applications running on multiple cores.

The rest of the paper is organised as follows: Section 2 gives an overview of Erlang and Percept, and Section 3, introduces Percept2. The scalability of Percept2, as well as how this is achieved, is discussed in Section 4. The usage of the tool is described in Sec-


*Erlang '13*, September 28, 2013, Boston, MA, USA.

http://dx.doi.org/10.1145/2505305.2505311

tion 5. A case study showing how Percept2 is used to guide the parallelisation of an Erlang application in described in Section 6. The Erlang built-in trace mechanisms are discussed in Section 7. Finally, Sections 8 and 9 conclude after addressing related and future work.

## 2. Erlang and Percept

**Erlang** [10] makes parallel programming easy by modelling the world as sets of parallel processes that can interact only by exchanging messages. Erlang concurrency is directly supported in the virtual machine, rather than indirectly by operating system threads. Erlang processes are very lightweight, and as a result an Erlang program can be made up of thousands or millions of processes that may run on a single processor, a multicore processor or a manycore system.

The main implementation of Erlang is the Erlang/OTP (Open Telecom Platform) system [4], and this was equipped with symmetric multi-processing (SMP) capabilities in 2006; SMP support has been improved continuously since then. In the current release (R16B), the VM detects the CPU topology automatically at startup, and creates a scheduler for each CPU core available. Each scheduler has it own process run-queue, and processes are migrated between run-queues if scheduler loads need to be balanced [19, 21].

**Built-in Tracing.** The Erlang runtime system has built-in support for tracing many types of events, allowing an application to be traced while being executed. No special compilation or instrumentation of the program is needed.

Erlang's built-in support for tracing is exposed to end users through a number of built-in functions (BIFs): `erlang:trace/3`, `erlang:trace_pattern/3`, and `erlang:trace_info/2`. The function `erlang:trace/3` *enables* and *disables* the low-level tracing. When enabling tracing, the user can specify which processes to trace, and which events they are interested in. The process that makes the call to `erlang:trace/3` to enable the tracing is known as the *tracer process*. In Erlang, at any one time, any process can only be traced by one tracer process.

Events that can be traced include: global and local function calls, process-related activities, message passing, garbage collection and memory usage. When tracing is enabled, trace events are sent as messages of the following format:

```
{trace, Pid, Tag, Data1 [,Data2]}
```

where `[,Data2]` denotes an optional field dependent on the trace message type. If the `timestamp` flag is given, the first element of the tuple will be `trace_ts` instead and the timestamp is added as the last element of the tuple.

The `erlang:trace_pattern/3` BIF, used in conjunction with `erlang:trace/3`, is for enabling the tracing of local and global function calls, with which, a user can specify the subset of functions to be traced using Erlang's *match specification* mechanism. A match specification consists of an Erlang term describing a small program that expresses a condition to be matched over a set of arguments. If the matching succeeds, a trace event is generated and some predefined actions can be executed. A function `call` or `return_to` trace event will only be generated if a *traced* process executes a *traced* function.

**Percept** [11] is a offline profiling tool included in the Erlang/OTP distribution. It utilises trace information and profiler events to form a picture of the runnability of processes, from which one can infer the level of parallelism the application supports, and hopefully identify bottlenecks in the application.

In Percept, process states are monitored. A waiting or suspended process is considered to be *inactive* and a running or runnable process is considered *active*. Process events are collected and stored to a file that can then be analysed; the analyser parses the data file

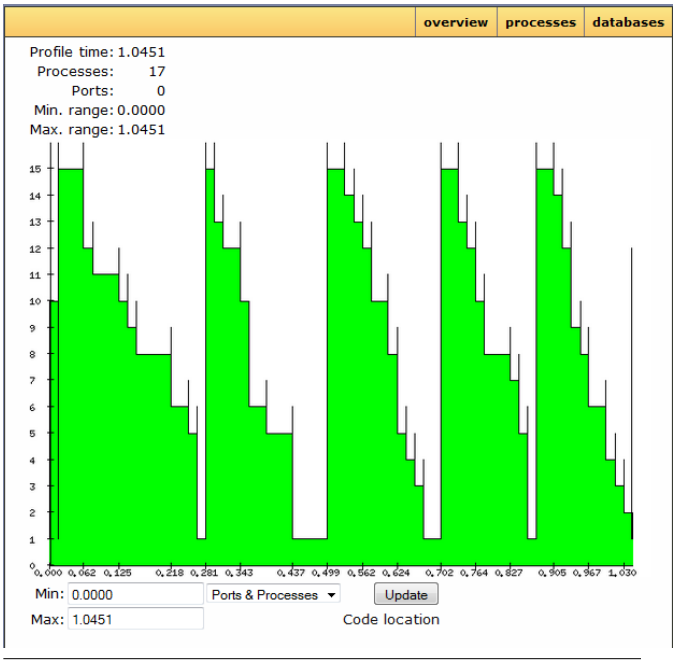



**Figure 1.** Percept: concurrency overview

and inserts all events into the `percept_db` database. After analysis the data can be viewed through a browser-based interface.

Percept generates an application-level concurrency profile, as shown in Fig 1, which is generated by profiling the example module `sorter.erl` used in the Percept documentation [11]. The profile shows the number of active processes at any point during profiling; dips represents low concurrency/parallelism. It is possible to zoom in on different parts of the profile to see the data for a specific time interval. The ratio between the number of runnable processes and the number of Erlang schedulers available can be used as an indicator of resource utilisation. For example, a very low ratio (less than 1) indicates under-utilisation of available Erlang schedulers, whereas a very high ratio indicates the high number of unfinished processes, which could potentially be a problem if the amount of memory used by each process is large.

Selecting the *processes* option from the menu leads to the process table page, as shown in Fig 2. Each row in the table shows information about a process, identified by its *P*rocess *id* (Pid): a lifetime bar that shows a rough estimate of when the process is alive during the profile, the entry-point, the process's registered name if it has one, and process's parent Pid. Pids in the table are clickable, and clicking on a Pid leads to the information page for this particular process, as shown in Fig 3. Apart from the basic process information shown in process table, this page also shows the Pids of the process' children, if any, and the process' inactive times. This latter includes how many times the process has been waiting for a message and in which function.

It is possible to select a number of processes of interest by ticking the *select* box in the process table, as shown in Fig 2, and compare their runnability during the execution by pressing the *compare* button. Fig 4 shows the runnablity of each of the processes selected during the period of profiling. In a runnability bar, *green* means the process is active (i.e running or runnable), and *white* means the process is inactive.

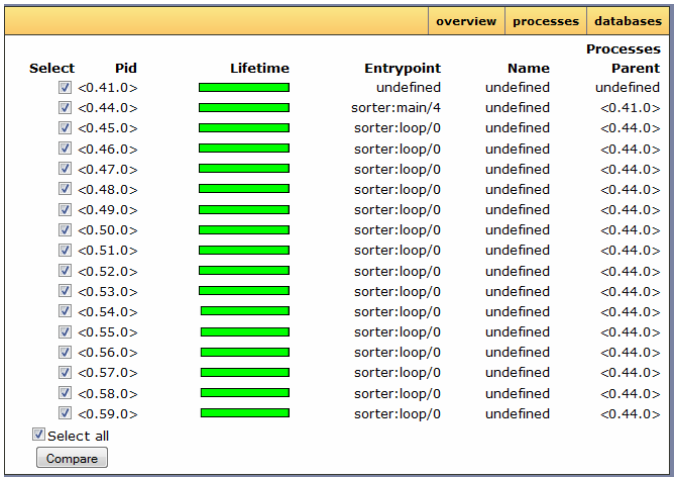



**Figure 2.** Percept: process table

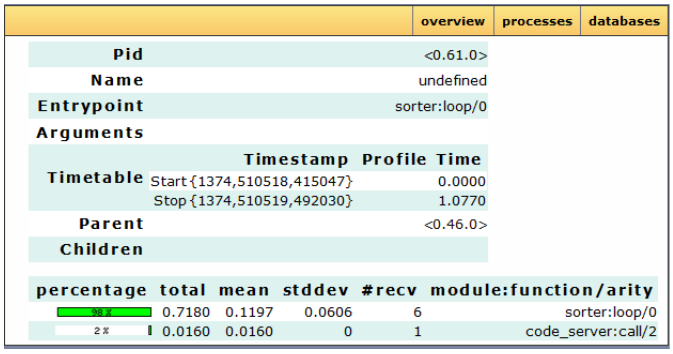



**Figure 3.** Percept: process information page

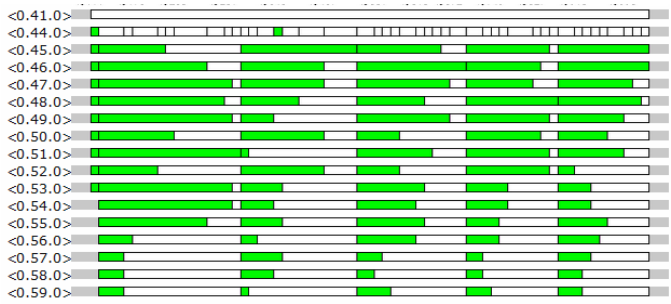



**Figure 4.** Percept: process runnability comparison

## 3. Percept2

The information provided by Percept is useful, but also limited. We have therefore extended Percept in two dimensions: functionality and scalability, and named the enhanced version *Percept2*. Percept2 is open source and downloadable from `https://github.com/huiqing/percept2`. The new functionalities added to Percept2 and the rationale behind the changes are discussed in this section; the performance and scalability of Percept2 is discussed in the next section.

***Scheduler Activity.*** Profiling the number of active schedulers at any time during program execution is a first step towards exposing multicore-related information to end users. In order to make full use of the multicore resource, an Erlang application should have enough processes to keep all the schedulers busy to avoid (as much as possible) any schedulers being inactive. In Percept2, a user can view the activeness of schedulers from the overview page by selecting the *scheduler* option from the drop list. The scheduler activity graph, as shown in Fig 5 generated by profiling type checking the Erlang `erts` library using Dialyzer [18], has a similar layout to the process activity graph shown in Fig 1, except that the Y-axis represents the number of active schedulers.

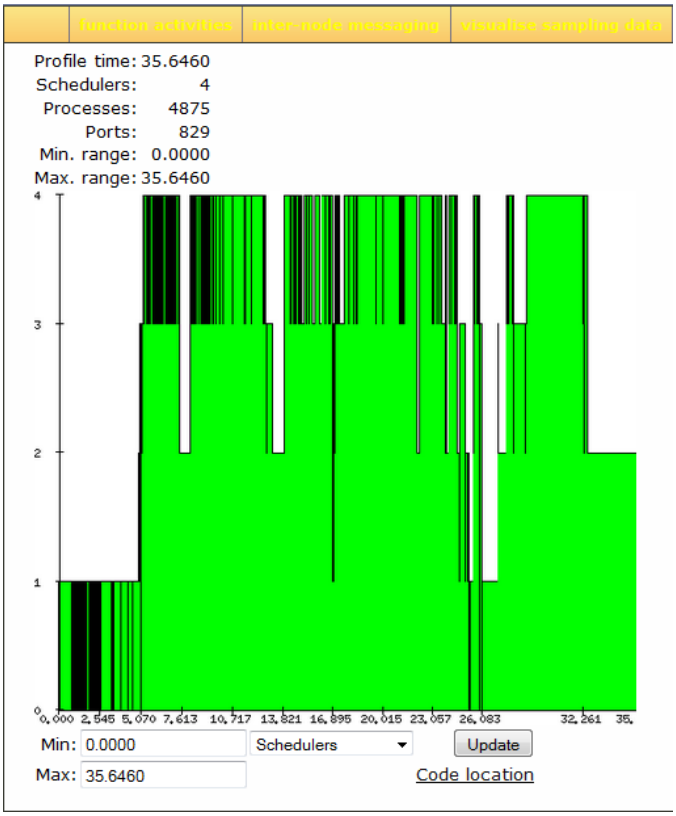



**Figure 5.** Percept2: active schedulers

***Process Migration***. One of the most important aspects of the multicore implementation is balancing workload between schedulers. In the Erlang VM, both work sharing and stealing approaches are employed. Once migration paths have been decided, schedulers with less work will pull processes from their counterparts, while schedulers with more work will push processes to others [19]. In Percept2, the migration history of a process from one scheduler to another is recorded, as is shown in the *RQ_history* row in the process information page in Fig 6. This information is not only useful for end-users, but also for the implementers of the virtual machine.

***Message Passing Between Processes/Schedulers.*** Message passing between processes can be monitored by enabling the tracing of *send* and *receive* events. This information shows how frequent the communication is between processes and the amount of data being sent/received by a process. In Erlang, messages between processes are copied, so frequent big messages being passed could affect the performance of an application. As shown in Fig 6, the number, as well as the average size, of messages sent/received by a process is profiled.

With Percept2, it is also possible to view process-to-process message passing, as shown in Fig 7. Each node in this graph represents a process, and the label on the link from one process to

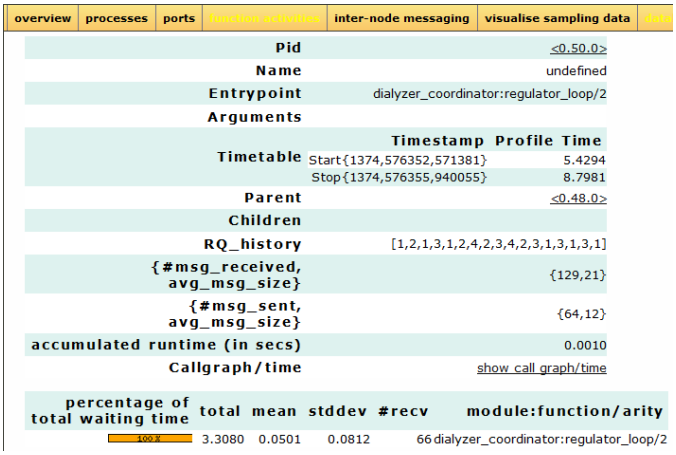



**Figure 6.** Percept2: process information page

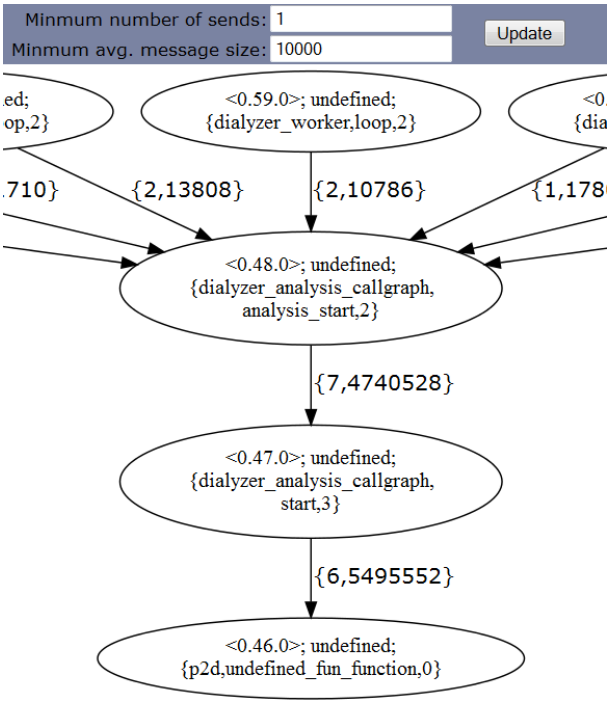



**Figure 7.** Percept2: process communication graph

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.

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.

***Accumulated Process Runtime.*** Unlike Percept, 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, also shown in Fig 6, more accurately.

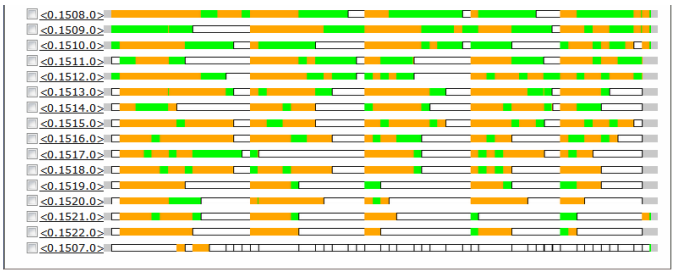



**Figure 8.** Percept2: process runnability comparison

The distinction between *running* and *runnable* is also reflected in the process runnability comparison. For instance, after profiling the same program used in Section 2 using Percept2, the process runnability comparison table is as shown in Fig 8, where *orange* represents *runnable* and *green* represents *running*.

***Structured Process Display.*** Processes are shown in an expandable/collapsible tree structure in Percept2, as in Fig 9, which is generated by profiling Wrangler [17]'s similar code detection functionality, so the parent-child relationships between processes are made explicit.

The number of processes spawned by an application may well increase significantly when parallelism is added, challenging the scalability of the tool, and making it difficult for users to navigate through the process tree and to spot interesting processes. Based on the observation that processes with the same entry function and spawned by the same parent process (possibly with different arguments) in general exhibit common behaviour, Percept2 displays a *compressed* version of the process tree for users to explore. For those sibling processes which share the same entry function, only one representative process is displayed, and all the remaining are combined into one artificial process.

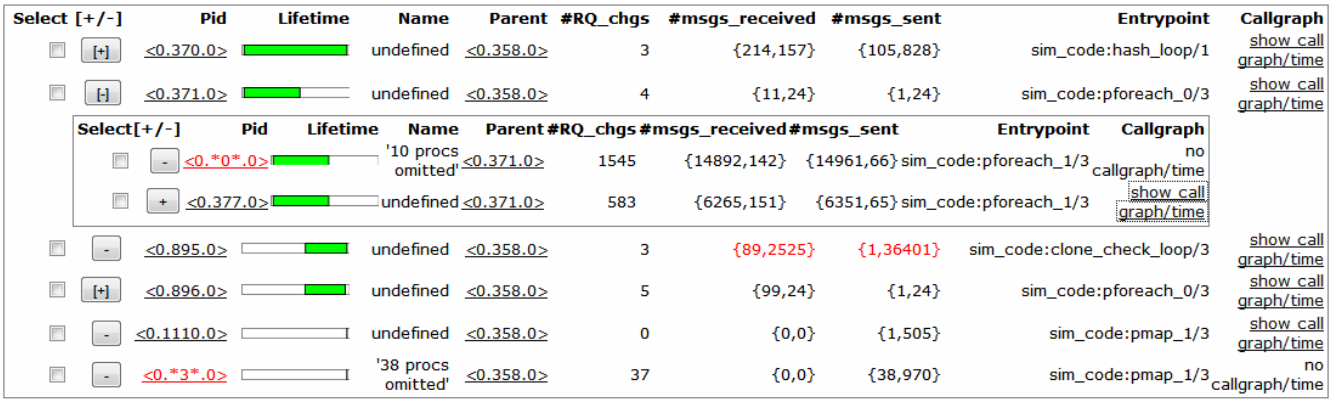
For instance, in Fig 9, the process with *pid* `<0.371.0>` spawned 11 processes because of the use of the parallel version of the `lists:foreach` function. Instead of listing all the 11 processes, Percept2 shows the process with the *pid* `<0.377.0>`, and collapses the remaining 10 processes into the artificial entry `<0.*0*.0>`. The uncompressed process (`<0.377.0>` in this case) is not chosen randomly, instead it is the process with the most accumulated runtime among the siblings. Although not shown in the process tree, information about those compressed processes is reachable by clicking on the artificial pid, i.e. `<0.*0*.0>`, which leads to a process information page containing links to those processes compressed.

A graph representation of the process tree structure is also generated by Percept2, and can be reached by clicking on the 'Process Tree Graph' link from the process tree page.

***Function Profiling.*** Process information alone does not say much about how a process's execution time is used, or how much of a process's execution time is spent on a particular computation of an Erlang application.

To address this problem, Percept2 allows the tracing of function activities, and is able to provide the user with profiling information including: dynamic, *calling-context-aware*, callgraphs; the total amount of time the process spent on a particular function, information about which functions are active during a specific time-interval, etc.

There are already a number of function profiling tools available as a part of the Erlang distribution, including `fprof` [5], `eprof` [3], `cover` [1] and `cprof` [2]. Among these profiling tools, `fprof` provides the most detailed information about functions. It measures the execution time for each function, both own time and accumulated time, and also records the amount of time a process spends



| Select | [+/-] | Pid | Lifetime | Name | Parent | #RQ_chgs | #msgs_received | #msgs_sent | Entrypoint | Callgraph |
|---|---|---|---|---|---|---|---|---|---|---|
| ☐ | [+] | <0.370.0> | | undefined | <0.358.0> | 3 | {214,157} | {105,828} | sim_code:hash_loop/1 | show call graph/time |
| ☐ | [-] | <0.371.0> | | undefined | <0.358.0> | 4 | {11,24} | {1,24} | sim_code:pforeach_0/3 | show call graph/time |
| Select | [+/-] | Pid | Lifetime | Name | Parent | #RQ_chgs | #msgs_received | #msgs_sent | Entrypoint | Callgraph |
| ☐ | - | <0.*0*.0> | | '10 procs omitted' | <0.371.0> | 1545 | {14892,142} | {14961,66} | sim_code:pforeach_1/3 | no callgraph/time |
| ☐ | + | <0.377.0> | | undefined | <0.371.0> | 583 | {6265,151} | {6351,65} | sim_code:pforeach_1/3 | show call graph/time |
| ☐ | - | <0.895.0> | | undefined | <0.358.0> | 3 | {89,2525} | {1,36401} | sim_code:clone_check_loop/3 | show call graph/time |
| ☐ | [+] | <0.896.0> | | undefined | <0.358.0> | 5 | {99,24} | {1,24} | sim_code:pforeach_0/3 | show call graph/time |
| ☐ | - | <0.1110.0> | | undefined | <0.358.0> | 0 | {0,0} | {1,505} | sim_code:pmap_1/3 | show call graph/time |
| ☐ | - | <0.*3*.0> | | '38 procs omitted' | <0.358.0> | 37 | {0,0} | {38,970} | sim_code:pmap_1/3 | no callgraph/time |

**Figure 9.** Percept2: compressed process tree

on garbage collection and suspension while executing a function. The caller of a function call is recorded, hence a callgraph for a process can be built from the trace information. The disadvantage of `fprof` is that it slows down program execution significantly, and the trace file can be very large. `eprof` has less impact on the program execution, but it also provides less profiling information. For example, `eprof` does not measure the time spent on garbage collection or suspension, and does not record the caller function of a function call, which means a callgraph cannot be built from the trace data. `cover` and `cprof` provide even less profiling information; both tools collect information on a per module basis, hence are not suitable for use in Percept2 either.

In order to provide 'good enough' profiling information to end users while avoiding slowing down program execution significantly, we decided to build into Percept2 a simplified 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 her/his 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.

To enable the tracing of functions, both exported and local, defined in a particular module, the user needs to supply Percept2 with the name of that module when the profiling is started. We elaborate more about the calling-context-aware callgraph next.

***Calling-context-aware Callgraph.*** Function callgraphs are a commonly used representation of function invocation behaviour, in which nodes representing functions are connected by directed edges representing caller-callee relationships. In general, callgraphs are succinct because each function is represented as a single node regardless of the different paths on which the node occurs; however this feature also limits the amount of information provided to the user, such as the calling-context information. In Percept2, a partial *calling-context-aware* callgraph is used to represent the dynamic function invocation behaviour.

Instead of merging every occurrence of the same function node, Percept2 only merges identical function nodes that share the same parent node, i.e. the same calling context, and identical function nodes in a repeated call path sequence due to recursion (hence partial).

With the 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. As shown in graph in Fig 14, a slice of a callgraph generated from the case study to be discussed in Section 5, each edge is annotated with the number of times the target function is called by the source function, and each node is annotated with the time the function took as a percentage of the process's life time. In the callgraph, pseudo functions `suspend` and `garbage_collect` are used to indicate the time a function spends on suspension and garbage collection respectively.

***Support for Distribution.*** With Percept2, it is also possible to profile the inter-node message passing in a distributed Erlang system. When multiple Erlang nodes are profiled, each node produces its own trace data file, which can then be passed on to a Percept2 trace analyser. Percept2 is able to analyse multiple trace files in parallel, as will be discussed in the next section, hence there is no need to merge trace files from different nodes into a single trace file.

## 4. Implementation and Scalability

The trace analyser in Percept is a sequential program, hence does not scale well on multicore machines. To improve the performance and scalability of Percept2, we have parallelised the trace analyser in two ways: first, Percept2 is parallelised so that multiple trace files can be analysed in parallel; second, the analysis of each single trace file is parallelised.

***Parallel analysis of multiple trace files***. In Percept2, trace data can be written to a number of files, each with a limited size. When the file specification for the data destination is specified in the format of {`FileName, wrap, Suffix, WrapSize`}, trace data is initially written to `FileName++"0"++Suffix` until a trace message written to the file makes it longer than `WrapSize`; at which point, the file is closed and the trace following will be written to `FileName++"1"++Suffix` until it reaches its size limit, and so on.

When multiple trace files are analysed, Percept2 starts a trace analyser process for each trace file. A trace analyser reads a trace message each time, process it, and update the database, which is a collection of ETS (Erlang built-in Term Storage) tables used to store the trace data internally. To reduce potential lock contention on some frequently visited ETS tables, each trace analyser creates its own database to hold its local data, while ETS tables that hold

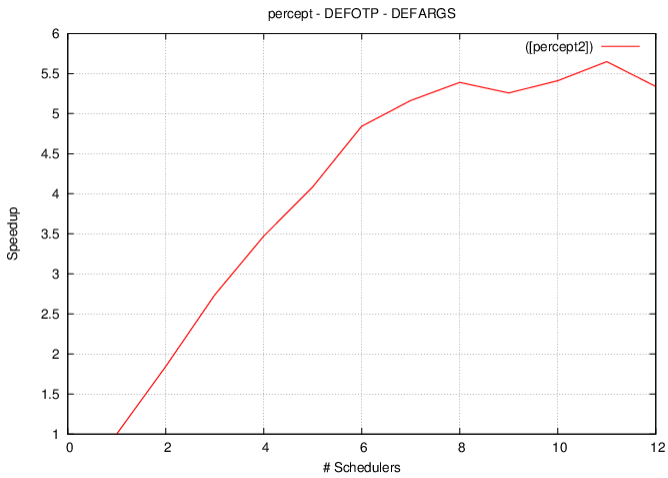



**Figure 10.** Percept2 Speedup Diagram

global data, or are updated less frequently are shared among the trace analysers.

***Parallel analysis of a single trace file***. The processing of a single trace file is parallelised in the way that a process is spawned for each particular kind of messages. For instance, there is a process for handling process activities, scheduler activities, and function activities respectively. The process handling function activities is further parallelised so that a process is spawned for every process in the application to handle the function trace events for that particular process. With this parallelisation, the main trace analyser process is only responsible for the reading and dispatching of trace messages to other processes.

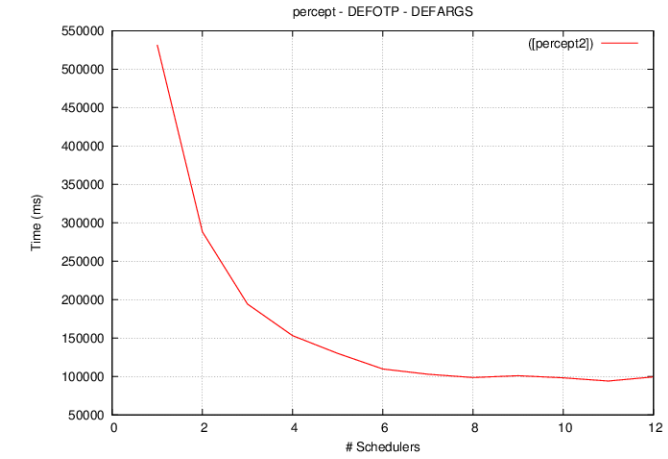



**Figure 11.** Percept2 Time Diagram

***Performance and Scalability***. The scalability of Percept2 on a 12-core processor running CentOS 6.3 is shown in Fig 10, and time spent is shown in Fig 11. The figures were generated using Bencherl [8] – a scalability benchmark suite for Erlang/OTP. In this experiment, Bencherl was configured to execute Percept2's trace analyser with five trace files as input. The total size of the five trace files is 1.36G, and there are 11,008,609 trace messages in total. Bencherl first runs the trace analyser on an Erlang VM using only one scheduler, then adds a scheduler each run until there are 12 schedulers. The results shows that the trace analyser scales reasonably well. Timewise, the execution time with 1 core is 531 seconds, and this reduced to 99.5 seconds when the number of cores reaches 12.

***Profiling Overhead***. To reduce the overhead of applying profiling to an Erlang application, Percept2 allows a user to selectively control which feature, or set of features, to profile. For instance, if message passing is not of interest to the user, they can choose not to include the feature `'message'` in the profiling options when starting profiling. When function activities are not traced, the overhead introduced by Percept2 is reasonable. For instance, one of the Erlang applications that we profiled creates 14848 processes and 672 ports during its execution. The running time without profiling on a 12-core machine is 99.19 seconds. Using Percept2 to profile this application with all the features supported by Percept2 (apart from function profiling) enabled, the running time is 111.09 seconds, a profiling overhead of 11% of the original execution time. When the profiling of function activities is enabled, the overhead could increase significantly depending on how many functions, and how many processes executing these functions, are being profiled. As mentioned earlier, our approach to reducing the overhead introduced by function profiling is *selective profiling*, i.e., selectively tracing only the function activities of a subset of processes that exhibit common behaviour.

## 5. How to use Percept2

Percept2 follows Percept's API interface style with only minor changes to the types of the arguments accepted by the `profile` and `analyze` functions. As for Percept, the profiling process consists of three steps: starting/stopping the profiling, analysing trace data and visualising profiling result.

Depending on the size of the application being profiled, and the parts of the application that a user is interested in, there are three ways to invoke Percept2 profiling:

**Profile a complete run of an application.** Performing this requires the entry function of the application, and the command for starting profiling is:

```
percept2:profile(FileSpec, Entry, Options).
```

where `FileSpec` specifies the files used to store trace data, `Entry` is the `{Module, Function, Args}` representation of the entry function of the application, and `Options` specifies which aspects of the application should be profiled.

`FileSpec` can be the name of a single file, or in the format of `{FileName, wrap, Suffix, WrapSize}` specifying a collection of files to store the trace data. `Options` has the type of `[profile_option()]`, where the type `profile_option()` is defined in Fig 12. Profiling starts with execution of the entry function, and goes on for the whole duration until the entry function returns and profiling has concluded.

**Profile a time slice of the application run.** The commands for starting and stopping profiling while an application is already up and running are:

```
percept2:profile(FileSpec, Options)
percept2:stop_profile().
```

**Profile a particular part of an application.** This can be achieved by inserting the commands for starting and stopping profiling at appropriate points in the application code. Obviously, the application code needs to be recompiled after these changes.

Once profiling has been finished, the following commands used to analyse the trace data: the former for a single file, the latter for multiple trace files.

```
percept2:analyze([FileName])
percept2:analyze
        (FileBaseName, Suffix, StartIndex, EndIndex).
```

The last step is to visualise the profiling result. To do this, use `percept2:start_webserver(8888)` to start the web server, then go to 'localhost:8888' in your web browser. If the port number is not given, an available port number is assigned automatically.

## 6. A Case Study

In this section we use Percept2 to guide the parallelisation of the code clone detection functionality in Wrangler [17], an Erlang refactoring tool. The clone detection tool is able to find code fragments that have a non-trivial common abstraction in Erlang systems. The algorithm consists of a number of phases: parse Erlang files into abstract syntax trees (ASTs); generalise, flatten and hash ASTs; identify clone candidates, and finally check clone candidates to identify genuine clones. Apart from the identification of clone candidates, which is implemented in C, all the other phases are implemented in Erlang. The original implementation of the clone detection algorithm is sequential, although with some concurrency built-in; Fig 13 shows the process activity profile of this implementation. As can be seen, most of computation is done by a single process, i.e., the process `<0.37.0>`. Examining the dynamic callgraph of process `<0.37.0>` in top-down order, we noticed that the function `generalise_and_hash_file_ast_1/5` and its direct parent have a larger number of call counts compared to their caller functions, and consume 78% of the process lifetime. This portion of the callgraph is shown in Fig 14. The actual code shows that this function is called in a list comprehension over a list of files. We refactored the sequential list comprehension to a parallel version, as shown in Fig 15, where `para_lib` is a library of parallel utility functions (written by us).

The previous refactoring improves the parallelism of the algorithm, however profiling the refactored program still shows that some processes are more heavily loaded than others. We examined the callgraph of one of those processes, and this leads to the refactoring step 2 (see Fig 16), which turns the use of sequential `lists:foreach/2` into a parallel version. The macro `?PARALLEL` in the new code is used to control the maximum number of new processes to be spawned; if this value is smaller than the number of elements to be processed in the list, this list will be chopped into smaller sublists and a process spawned for each sublist.

Repeating this profiling and examination process leads to our third refactoring, as shown in Fig 17, which involves the refactoring of a recursive function definition. This refactoring is slightly more complex than the previous two steps because of the need to eliminate the data dependency between two consecutive recursions. On a 4-core machine, the scalability of the clone detection algorithm after the three refactoring steps is shown in Fig 18. We note however that parallelisation of Erlang programs is by no means straight forward due to the fact that Erlang allows expressions to have side-effects and raise exceptions.

## 7. Extensions to Erlang Trace

The amount of data that needs to be collected and analysed limits the scalability of profiling tools. Reducing data collection will thus improve scalability, and we have written prototype extensions of Erlang tracing to support this.

***Message Size vs. Message Content***. Currently Erlang tracing logs the complete messages *sent/received* by traced processes when the *send/receive* flags are on. While message content may be useful for some cases, it will also increase the size of trace data significantly, especially when large messages are sent between processes, and there are frequent messages between processes. We extended Erlang tracing with the option of logging the size of messages, to be used in conjunction with the *send/receive* trace flags.

***Dynamic Filtering and Manipulating of Trace Messages***. Erlang's built-in tracing provides powerful support for controlling the function call events to be traced by means of *match specification*. While match specifications can be used to filter/manipulate function-related trace events, it is not applicable to other trace events; this does not mean that there is no such a need for dynamic filtering/manipulating of other trace events however, actually it is quite the contrary.

We have therefore prototyped an extension to the Erlang built-in trace and match specification implementation to allow the dynamic filtering of general trace messages, and added `erlang:trace_filter/2` as a new BIF function, with the following specification:

```
erlang:trace_filter(PidSpec,MatchSpec)->integer()>=0
```

where the spec uses the following types:

```
PidSpec   = pid() | existing | new | all
MatchSpec = MatchSpecList | boolean()
```

`erlang:trace_filter/2` works like this: for each trace event generated by the process(es) represented by `PidSpec`, the trace message is matched over the match specifications specified by `MatchSpec`; the trace message is logged only if the match succeeds. Setting `MatchSpec` to `true` does not filter out any trace messages, whereas setting it to `false` will filter out all the trace messages, i.e. no trace messages will be recorded.

For instance, suppose we have a system using many processes, and we would like to log the 'send' events when a process sends the atom 'true' to another process. The following commands, when used together, will make sure that only the sending of message 'true' is logged, whereas all others are ignored.

```
erlang:trace(all, true, [send],
erlang:trace_filter(all,
    [{['trace', '_', 'send', 'true' '_'],
       true, [{message, true}]}]).
```

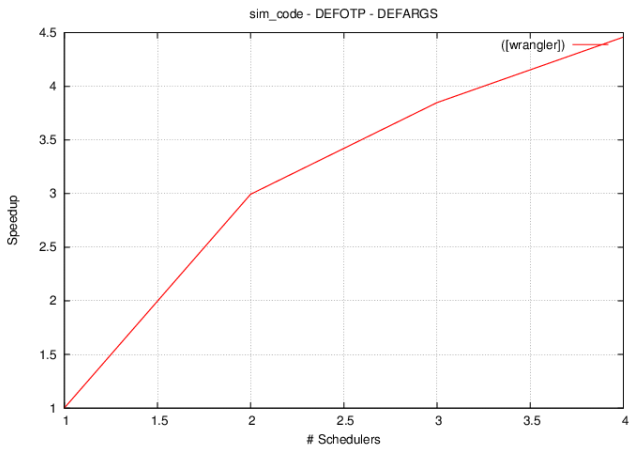



**Figure 18.** Clone Detection Speedup Diagram

```
-type profile_option():: procs       %% profile process concurrency
                       | ports       %% profile ports concurrency
                       | schedulers  %% profile scheduler activity
                       | running     %% profile process concurrency, also
                                     %% distinguish running and runnable.
                       | message     %% profile process concurreny and message passing.
                       | migration   %% profile process concurreny and process migration.
                       | all         %% enable all the above options.
                       |{callgraph, [module_name()]}. %% profile process concurrency, as well as
                                                      %% the 'call/return_to' activties of funcitons
                                                      %% defined in the modules given in the list.
```

**Figure 12.** Type definition of `profile_option()`

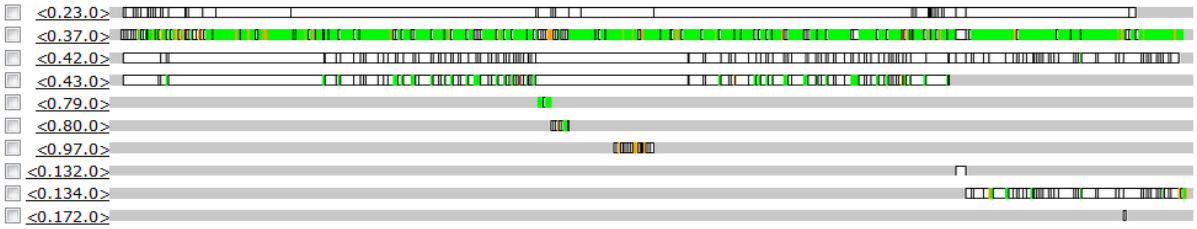



**Figure 13.** Clone detection: process activity comparison

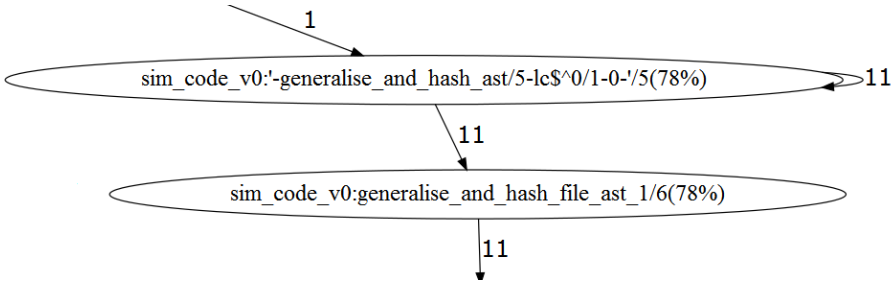



**Figure 14.** Percept2: dynamic callgraph

```
%Before:
[generalise_and_hash_file_ast_1(File, Threshold, ASTPid, true,SearchPaths, TabWidth)||File<-Files].
%After:
para_lib:pmap(fun(File) ->
                generalise_and_hash_file_ast_1(File, Threshold, ASTPid, true, SearchPaths, TabWidth)
              end, Files)
```

**Figure 15.** Refactoring 1: list comprehension to parallel map

```
% Before:
lists:foreach(fun(Form)-> F(Form) end, Forms).
% After:
para_lib:pforeach(fun(Form) ->F(Form) end, Forms, ?PARALLEL).
```

**Figure 16.** Refactoring 2: sequential `foreach` to parallel `foreach`

## 8. Related Work

Related to our work, but with a different approach, is the work led by N.S.Papaspyrou from National Technical University of Athens. Instead of using Erlang's built-in trace, their work has been focusing on using and extending the existing DTrace probes to profile Erlang. DTrace [14] is a dynamic tracing framework, originally developed for Solaris, for observing performance issues of a running system at all levels of the software stack. It allows users to define

```
% Before:
examine_clone_candidates([],_Thresholds,CloneCheckerPid,_Num) ->
     get_final_clone_classes(CloneCheckerPid);
examine_clone_candidates([C|Cs],Thresholds,CloneCheckerPid,Num) ->
     output_progress_msg(Num),
  NewClones = examine_a_clone_candidate(C, Thresholds),
     add_new_clones(CloneCheckerPid, {C, NewClones}),
     examine_clone_candidates(Cs,Thresholds,CloneCheckerPid,Num+1).

% After:
examine_clone_candidates(Cs, Thresholds, CloneCheckerPid) ->
      NumberedCs = lists:zip(Cs, lists:seq(1, length(Cs))),
      para_lib:pforeach(fun({C, Nth}) ->
                          examine_a_clone_candidate({C,Nth},Thresholds,CloneCheckerPid)
                        end,NumberedCs),
      get_final_clone_classes(CloneCheckerPid).
examine_a_clone_candidate({C,_Nth},Thresholds,CloneCheckerPid) ->
     output_progress_msg(Nth),
     NewClones = examine_a_clone_candidate(C, Thresholds),
     add_new_clones(CloneCheckerPid, {C, NewClones}).
```

**Figure 17.** Refactoring 3: recursive function to parallel foreach.

what information is desired, how it should processed and formatted. The work of extending with DTrace probes the Erlang VM source code (implemented in C) was initiated in 2011 by Fritchie [12], and is part of the official Erlang/OTP source code from release R15B01; this forms the foundation of Papaspyrou's current research on Erlang profiling.

VampirTrace/Vampir [20] is a tool set and a runtime library for instrumentation and tracing of software applications. It is particularly tailored to parallel and distributed High Performance Computing (HPC) applications. The instrumentation part modifies a given application in order to inject additional measurement calls during runtime. The tracing part provides the actual measurement functionality used by the instrumentation calls. By this means, a variety of detailed performance properties can be collected and recorded during runtime. This includes function enter and leave events, MPI communication, OpenMP [6] events, and performance counters. After a successful tracing run, VampirTrace writes all collected data to a trace file in the Open Trace Format (OTF) [16]. As a result, the information is available for post-mortem analysis and visualisation by various tools. Most notably, VampirTrace provides the input data for the Vampir analysis and visualization tool.

Scalasca [13] is another representative trace-based tools that support performance optimization of parallel programs by measuring and analysing their runtime behaviours. The analysis identifies potential performance bottlenecks, in particular those concerning communication and synchronization, and offers guidance in exploring their causes. Scalasca targets mainly scientific and engineering applications based on the programming interfaces MPI and OpenMP, including hybrid applications based on a combination of the two.

In the functional programming paradigm, there is ThreadScope [15] for performance profiling of parallel Haskell programs. ThreadScope reads GHC-generated tracing events from a log file, and displays the thread profile information in a graphical viewer. The tool allows users to check if work is well balanced across the available processors and spot performance issues relating to garbage collection and poor load balancing.

## 9. Conclusions and Future Work

We have presented the Erlang concurrency profiling tool Percept2, an extended version of Percept in both functionality and scalability. Percept2 helps programmers to understand the parallelism inherent in their programs and to identify potential parallelism that they might introduce. The scalability of Percept2 allows it to handle larger trace data more efficiently.

Our work on Percept2 is currently going on in a number of directions. We are working on online visualisation of certain trace events, such as the process migration, message passing between schedulers, etc; Percept2's support for distribution is being improved, so that profiling information of multiple nodes can be viewed and compared in one page. Automatic mining of trace data and reporting of possible concurrency bottlenecks is another focus of our work on improving Percept2.

This research is supported by EU FP7 collaborative project RELEASE (`http://www.release-project.eu/`), grant number 287510. We are very grateful to the developers of Percept, on whose work we have built.

## References

[1] cover - A Coverage Analysis Tool for Erlang. `http://www.erlang.org/doc/man/cover.html`.

[2] cprof - A simple Call Count Profiling Tool. `http://www.erlang.org/doc/man/cprof.html`.

[3] eprof - A Time Profiling Tool for Erlang. `http://www.erlang.org/doc/man/eprof.html`.

[4] Erlang/OTP. `http://www.erlang.org`.

[5] fprof - An Erlang File Trace Profiler. `http://www.erlang.org/doc/man/fprof.html`.

[6] The OpenMP API specification for parallel programming. `http://openmp.org/wp/`.

[7] J. Armstrong. *Programming Erlang*. Pragmatic Bookshelf, 2007.

[8] S. Aronis, N. Papaspyrou, et al. A scalability benchmark suite for Erlang/OTP. In *Proceedings of the eleventh ACM SIGPLAN workshop on Erlang*, 2012.

[9] S. Aronis and K. Sagonas. On Using Erlang for Parallelization Experience from Parallelizing Dialyzer. In *Draft Procs. of Symp. on Trends in Funct. Prog.*, 2012.

[10] F. Cesarini and S. Thompson. *Erlang Programming*. O'Reilly Media, Inc., 2009.

[11] B.-E. Dahlberg. Percept - An Erlang Concurrency Profiling Tool. `http://www.erlang.org/doc/man/percept.html`.

[12] S. L. Fritchie. DTrace and Erlang: A new beginning. Erlang User Conference 2011.

[13] M. Geimer, F. Wolf, B. J. N. Wylie, E. brahm, D. Becker, B. Mohr, and F. Jlich. The SCALASCA performance toolset architecture. *Concurr. Comput. : Pract. Exper.*, 22(6):702–719, Apr. 2010.

[14] B. Gregg and J. Mauro. *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X, and FreeBSD*. Prentice Hall, 2011.

[15] D. Jones Jr., S. Marlow, and S. Singh. Parallel performance tuning for Haskell. In *Haskell Symposium 2009*, Edinburgh, Scotland, Sept. 2009. ACM Press.

[16] A. Knpfer, R. Brendel, H. Brunst, H. Mix, and W. Nagel. Introducing the Open Trace Format (OTF). In V. Alexandrov, G. Albada, P. Sloot, and J. Dongarra, editors, *Computational Science, ICCS 2006*, volume 3992 of *Lecture Notes in Computer Science*, pages 526–533. Springer Berlin Heidelberg, 2006.

[17] H. Li and S. Thompson. Incremental Code Clone Detection and Elimination for Erlang Programs. In *Fundamental Approaches to Software Engineering (FASE'11)*, 2011.

[18] T. Lindahl and K. Sagonas. Practical type inference based on success typings. In *Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, pages 167–178, New York, NY, USA, 2006. ACM Press.

[19] K. Lundin. About Erlang/OTP and Multi-core Performance in Particular. Erlang Factory London 2009.

[20] M. S. Mller, A. Knpfer, M. Jurenz, M. Lieber, H. Brunst, H. Mix, and W. E. Nagel. Developing Scalable Applications with Vampir, VampirServer and VampirTrace. In *PARCO*, volume 15 of *Advances in Parallel Computing*, pages 637–644. IOS Press, 2007.

[21] P. Nyblom. Erlang SMP Support. Erlang User Conference 2009.

[22] J. Zhang. Characterizing the Scalability of Erlang VM on Manycore Processors. Technical Report 5, KTH, School of Info. and Comm. Tech., 2011.