National Technical University of Athens
School of Electrical and Computer Engineering
Division of Computer Science

Race Condition Detection in Concurrent Erlang Applications Using Static Analysis

Diploma Thesis

by

MARIA I. CHRISTAKIS

Supervisor: Konstantinos Sagonas
N.T.U.A. Associate Professor

Software Laboratory
Athens, September 2009

National Technical University of Athens
School of Electrical and Computer Engineering
Division of Computer Science
Software Laboratory

Race Condition Detection in Concurrent Erlang Applications Using Static Analysis

Diploma Thesis

by

MARIA I. CHRISTAKIS

Supervisor: Konstantinos Sagonas
N.T.U.A. Associate Professor

This thesis was approved by the examining committee on the 25th of September, 2009.

(Signature)	*(Signature)*	*(Signature)*
....	....	....
Nectarios Koziris	Nikolaos Papaspyrou	Konstantinos Sagonas
N.T.U.A. Associate Professor	N.T.U.A. Assistant Professor	N.T.U.A. Associate Professor

Athens, September 2009

(Signature)

....

Maria I. Christakis
Graduate Electrical and Computer Engineer, N.T.U.A.

National Technical University of Athens
School of Electrical and Computer Engineering
Division of Computer Science
Software Laboratory

Abstract

In safety-critical and high-reliability concurrent systems, software development and maintenance require great effort. This effort could be significantly reduced if concurrency defects, among other software errors, were identified through automatic tools such as program analyzers and compile-time checkers. We therefore address the problem of finding some commonly occurring kinds of race conditions in Erlang programs using static analysis. Our analysis is completely automatic, fast and scalable, and avoids false alarms by taking language characteristics into account. We have integrated our analysis in a publicly available, commonly used tool for detecting software defects in Erlang programs and evaluate its effectiveness and performance on a suite of widely used industrial and open source programs of considerable size. The number of previously unknown race conditions that we have detected in them is significant.

Categories and Subject Descriptors

- D.1.3 [*Programming Techniques*]: Concurrent Programming
- D.2.5 [*Software Engineering*]: Testing and Debugging – Diagnostics
- D.3.2 [*Programming Languages*]: Language Classifications – Concurrent, distributed, and parallel languages

General Terms

Algorithms, Design, Languages, Measurement, Performance

Keywords

static race detection, concurrent languages, Erlang

To all those who seek knowledge
for its own sake

Contents

List of Figures

List of Tables

Acknowledgements

First and foremost, my thanks go to my advisor, Kostis Sagonas. His work in software development has been a great source of inspiration since the first years of my undergraduate studies; his deep scientific insight and ceaseless enthusiasm about research have made my working environment quite enviable; his essential advice has definitely brought me closer to my goal of becoming a creditable software researcher and developer myself; his friendship and care over the last year have been immensely supportive. I cannot express how much I look forward to the next five years!

Thanks are also to my parents for their love and affection and for convincing me that even a difficult endeavour may be accomplished if taken one step at a time.

Finally, I would like to extend my thanks to my fellow students. It would not have been the same without the enjoyment and stimulation I got out of our interaction.

Μαρία Χρηστάκη

Chapter 1

Introduction

1.1 Statement of the Problem

1.1.1 *"There ain't no such thing as a free lunch"* [1]

In recent years, an interesting phenomenon, known as "Andy giveth, and Bill taketh away", has indicated the need for software development to make a turn toward concurrency [30]. No matter how fast processors become, software always has its way to eat up the extra speed, either by finding a lot more to do or by doing it inefficiently enough. After all, we cannot expect processor speed to keep growing exponentially forever. It seems that this "free ride" era, when programmers could lay back and wait for Moore's law to take effect, has given the nod to its successor and a whole new ball game has begun!

1.1.2 No Pain, No Gain: Concurrency Problems

Concurrency is fundamental in computer programming, both as a method to better structure programs and as a means to speed up their execution. Nowadays concurrent programming is also becoming a necessity in order to take advantage of multi-core machines which are ubiquitous. The only catch is that concurrent programming is harder and more error-prone than its sequential counterpart. We have all heard and sometimes even experienced horror stories of memory violations, race conditions, shared-memory corruption and the like, when programming with multiple execution threads.

More specifically, the typical problems with concurrency can be outlined as follows:

- *Race condition*: A strange interleaving of processes has an unintended effect.
- *Deadlock*: Two or more processes stop and wait for each other.
- *Livelock*: Two or more processes keep executing without making any progress.

These problems are usually *heisenbugs* [28] – they can alter their behaviour or completely disappear when one tries to isolate them – since they go hand in hand with the order of execution of the processes involved.

[1] R. A. Heinlein, *The Moon Is a Harsh Mistress*

These entirely new difficulties/bugs have a few characteristics:

- They are more likely to strike on today's multicore processors where the process interleavings can become quite complex.
- The nondeterminism of concurrency makes the task of tracking them down a notoriously difficult one. We just never know when the last bug has been removed.
- Even experienced programmers sometimes have a hard time reasoning about the correctness of concurrent code. It is rather challenging to conceptualize all the possible interleavings of processes when looking at a piece of even straight-line code.
- They are very hard to reproduce since a program may be working just fine under most interleavings.

1.1.3 Erlang Is Not Immune

To make concurrent programming simpler and more natural for some tasks, different programming languages support different concurrency models. Some of them totally avoid some hazards associated with concurrent execution. One such language is Erlang, a language whose concurrency model is based on user-level processes that communicate using asynchronous message passing [2]. Erlang considerably simplifies the programming of some tasks and has been proven very suitable for some kinds of highly-concurrent applications; however, it does not avoid all problems associated with concurrent execution. In particular, the language currently provides no atomicity construct and its implementation in the Erlang/OTP system allows for many kinds of *race conditions* in programs, i.e., situations where one execution thread accesses some data value while some other thread tries to update this value [17]. In fact, there is documented evidence that race conditions are a serious problem when developing and troubleshooting large Erlang applications in industry [9].

1.2 Overview of the Solution

To ameliorate the situation and by building upon successful prior work on detecting software defects on the sequential part of Erlang [19, 25], we have embarked on a project aiming to detect concurrency errors in Erlang programs using static analysis. In this thesis we take a very important first step in that direction by presenting an effective analysis that detects race conditions in Erlang. So far, analyses for race detection have been developed for languages that support concurrency using lock-based synchronization and their techniques rely heavily on the presence of locking statements in programs. Besides tailoring the analysis to the characteristics of concurrency in Erlang, the main challenges for our work have been to develop an analysis that: 1) is completely automatic and requires no guidance from its user; 2) strikes a proper balance between soundness and precision;

3) is fast and scalable and thus able to handle large and possibly open programs; and 4) integrates smoothly with the existing defect detection analyses of the underlying tool. As we will see, we have achieved these goals.

The contributions of this thesis are as follows:

- It documents the most important kinds of data races in Erlang programs;
- it presents an effective and scalable analysis that detects these races, and
- it demostrates the effectiveness of the analysis by running it against a suite of widely used industrial and open source applications of significant size and reports on the number of race conditions that were detected.

The next chapter overviews the Erlang language and the defect detection tool which is the vehicle for our work. Chapter 3 describes what data races are in Erlang, followed by Chapter 4 which presents in detail the analysis we use to detect them. The effectiveness and performance of our analysis is evaluated in Chapter 5 and the thesis ends by reviewing related work and some final remarks.

Chapter 2

Preliminaries

2.1 Erlang and Erlang/OTP

Erlang is a strict, dynamically typed functional programming language with support for concurrency, communication, distribution, fault-tolerance, on-the-fly code reloading, automatic memory management and support for multiple platforms [3]. The number of areas where Erlang is actively used is increasing. However, its primary application area is still in large-scale embedded control systems developed by the telecom industry. The main implementation of the language, the Erlang/OTP (Open Telecom Platform) system from Ericsson, has been open source since 1998 and has been used quite successfully both by Ericsson and by other companies around the world to develop software for large commercial applications. Nowadays, applications written in the language are significant both in number and in code size making Erlang one of the most industrially relevant declarative languages.

2.1.1 Sequential Erlang

Every file of Erlang code is a module. Declarations within the file name the module after the file and declare which functions may be exported – called from other modules. Comments begin with the percent sign (`%`) and run to the end of the line.

In Erlang, terms are either variables, simple terms, structured terms, or function closures. Variables are *single-assignment* and always begin with a capital letter or an underscore. Simple terms include atoms, process identifiers, integers and floating point numbers. Structured terms are lists (enclosed in brackets) and tuples (enclosed in braces). Structured terms are constructed explicitly and deconstructed using pattern matching. Pattern matching is also used to select function clauses or different branches of `case` statements; the two forms are equivalent and choosing between them is a matter of taste. Figure 2.1 shows all the above. It also shows how Erlang code is organized in modules, how the code can contain calls to exported functions of some other module (the call to function `math:sqrt/1` in our example), how distinct functions may have the same name as long as they have different arities (`factorial/1` and `factorial/2`) and how pattern matching is enriched by the presence of flat guards such as type tests and arithmetic comparisons.

example.erl

```
-module(example).
-export([factorial/1, factorial/2, nth/2, area/1]).

factorial(0) -> 1;
factorial(N) -> N * factorial(N - 1).
factorial(0, Acc) -> Acc;
factorial(N, Acc) -> factorial(N - 1, N * Acc).

nth(1, [H|_]) -> H;
nth(N, [_H|T]) when is_integer(N), N > 1 ->
    nth(N - 1, T).

area(Shape) ->
    case Shape of
        {square, Side} when is_number(Side) ->
            Side * Side;
        {circle, Radius} ->
            3.14 * Radius * Radius; %% well, almost
        {triangle, A, B, C} ->
            S = (A + B + C) / 2,
            math:sqrt(S * (S - A) * (S - B) * (S - C))
    end.
```

Figure 2.1: A sequential Erlang program used as example

The Erlang language is rather small, but it has evolved from an even smaller language which over the years has been enriched with new language constructs [1]. For instance, for some years now Erlang supports a notation for function closures (known as *funs* in the Erlang lingo) when older Erlang versions only supported `apply`. Similarly, modern Erlang comes with language constructs to perform pattern matching directly on *binaries* and *bit streams* [14] when older Erlang required a conversion of binaries to lists first. Current Erlang comes with a notation for *records* as well, which allows referring to tuple elements by name instead of positions. Employing record notation and some appropriate declaration, we could then write the first `case` clause of the `area/1` function of our example program as follows:

```
#square{side = Side} when is_number(Side) ->
    Side * Side;
```

Over the years, Erlang has also adopted various constructs from other programming languages, most notably *list comprehensions*, which are a convenient shorthand for a combination of the `map`, `filter` and `append` functions on lists. List comprehension generators also serve as filter expressions. For example, the following list comprehension:

```
List = [{1, 2.56}, {3.14, 4}, some_atom, {5, 6}],
[Y * (Y + 1) || {X, Y} <- List, is_integer(X), X > 1].
```

will silently filter out the `some_atom`, `{1, 2.56}` and `{3.14, 4}` elements of the list and will produce the list `[42]`. On the other hand, in non-filter expressions, the evaluation of list comprehensions might throw a runtime exception since Erlang is dynamically typed. For example, the list comprehension we just showed would throw an exception if `List` also contained the term `{7, eleven}`.

2.1.2 Concurrent Erlang

Erlang's main strength is that it has been built from the ground up to support concurrency. In fact, its concurrency model differs from most other programming languages out there. Processes in Erlang are extremely light-weight (lighter than OS threads), their number in typical applications is quite large and their allocated memory starts very small (currently, 233 bytes) and can vary dynamically. Erlang's concurrency primitives `spawn`, `!` (send) and `receive` allow a process to spawn new processes and communicate with others through asynchronous message passing. Any data can be sent as a message and processes may be located on any machine. Each process has a *mailbox*, essentially a message queue, where each message sent to the process will arrive. Message selection from the mailbox occurs through pattern matching. To support robust systems, a process can register to receive a message if another one terminates. Erlang also provides mechanisms for allowing a process to timeout while waiting for messages and a `try`/`catch`-style exception mechanism for error handling. There is no process-noticeable shared memory and distribution is almost invisible in Erlang.

Note that Erlang processes differ from both OS processes and OS threads: An OS process usually has a separate address space implemented in hardware resulting in the need of TLB flushes and the like, while OS threads usually communicate through shared memory. Finally, OS processes and threads are often implemented in such a way that they can be executed in parallel.

In Erlang, on the other hand, the scheduling of processes is primarily the responsibility of the runtime system of the language. In the single-threaded version of the runtime system, there is a single scheduler which picks up processes from a single ready queue. The selected process gets assigned a number of reductions to execute. Each time the process does a function call, a reduction is consumed. A process gets suspended when the number of remaining reductions reaches zero, or when the process tries to execute a `receive` statement and there are no matching messages in its mailbox, or when it gets stuck waiting for I/O. In the multi-threaded version of the system, which nowadays is more common and the default on multi-core architectures, there are multiple schedulers (typically one for each core) each having its own ready queue. On top of that, the runtime system of Erlang/OTP R13B (the version released on March 2009) also employs a redistribution scheme based on *work stealing* when some scheduler's run queue becomes empty. A side-effect of all this is that the multi-threaded version of Erlang/OTP makes many more process interleavings possible and more likely to occur than in earlier versions. Indeed, in some applications written long ago, concurrency bugs that have laid hidden for a number of years have been recently exposed.

Let's take a look at concurrent Erlang by introducing a small example, illustrated in Figure 2.2 [12], in which we create two processes that practice at "ping pong".

```erlang
-module(ping_pong).
-export([start/0]).

ping(0) ->
    pong ! finished,
    io:format("Ping finished~n", []);
ping(N) ->
    pong ! {ping, self()},
    receive
        pong ->
            io:format("Ping received pong~n", [])
    end,
    ping(N-1).

pong() ->
    receive
        finished ->
            io:format("Pong finished~n", []);
        {ping, Ping_PID} ->
            io:format("Pong received ping~n", []),
            Ping_PID ! pong,
            pong()
    end.

start() ->
    register(pong, spawn(fun pong/0)),
    spawn(fun () -> ping(3) end).
```

Figure 2.2: A concurrent Erlang program used as example

The output of this example program is:

```
> ping_pong:start().
<0.38.0>
Pong received ping
Ping received pong
Pong received ping
Ping received pong
Pong received ping
Ping received pong
Ping finished
Pong finished
```

The `start/0` function spawns the *pong* process and gives it the name `pong`:

```erlang
register(pong, spawn(fun pong/0))
```

It then creates the *ping* process and returns its process identifier, or pid, which uniquely identifies the process (<0.38.0>). *Pong* is now waiting for messages. If the atom `finished` is received, *pong* writes `Pong finished` to the output and as it has nothing more to do, it terminates. If it receives a message of the form:

```erlang
{ping, Ping_PID}
```

it writes `Pong received ping` to the output and sends the atom `pong` to the *ping* process:

```erlang
Ping_PID ! pong
```

After sending this message to *ping*, *pong* calls function `pong` again and has to wait for another message. At the same time, *ping*'s second clause sends a message to *pong* which contains its process identifier:

```erlang
pong ! {ping, self()}
```

and waits for a reply:

```erlang
receive
    pong ->
        io:format("Ping received pong~n", [])
end
```

As soon as this reply arrives, it writes `Ping received pong` to the output and calls function `ping` again:

```
ping(N-1)
```

The argument of function `ping` is consecutively decremented until it becomes zero. When this occurs, its first clause will be executed:

```
ping(0) ->
    pong ! finished,
    io:format("Ping finished~n", []);
```

The atom `finished` is sent to *pong*, causing it to terminate, and `Ping finished` is written to the output. *Ping* then itself terminates as it has nothing left to do.

Though concurrency using message passing avoids some kinds of concurrency errors, Erlang is not immune to races. For example, if we had implemented a server-client messaging framework to read and increment a counter, as shown in Figure 2.3, we would be just as exposed to races as a shared-state implementation that forgot to take locks. It is obvious that some of the clients in the example may want to update the counter at the same time. In this case, they will send `get_counter` messages to the server and wait for its reply. The server will respond with the current counter value and immediately following this response, the clients on hold update the counter to the same new value since their read *and* write operations are not conducted atomically. A serious race condition has just happened.

2.1.3 Erlang/OTP

The main implementation of the language is the Erlang/OTP (Open Telecom Platform) system by Ericsson. At the time of this writing, the most recent Erlang/OTP system is R13B01 (release 13B01). Besides libraries containing a large set of built-in functions (BIFs) for the language, the Erlang/OTP system comes with a number of ready-to-use components and design patterns – known as *behaviours* – (such as finite state machines, generic servers, supervisors, etc.), providing a set of design principles for developing fault-tolerant Erlang applications. Indeed, a fair number of commercial and/or open-source applications have been written over the years using the Erlang/OTP system and making Erlang both one of the most industrially relevant declarative languages as well as a language with a significant body of existing code out there.

2.2 Dialyzer: A Brief Overview

Since 2007 the Erlang/OTP distribution includes a static analysis tool, called `dialyzer` [19, 25], for finding discrepancies (i.e., type errors, software defects such as exception-raising

```erlang
%% BAD - race-prone implementation - do not use - BAD
-module(bad_counter).
-export([server/0]).

server() ->
    Self = self(),
    spawn(fun () -> client(Self) end),
    spawn(fun () -> client(Self) end),
    spawn(fun () -> client(Self) end),
    loop(0).

loop(N) ->
    io:format("N = ~w~n", [N]),
    receive
        {get_counter, Pid} ->
            Pid ! N,
            loop(N);
        {set_counter, C} ->
            loop(C)
    end.

client(Server) ->
    Server ! {get_counter, self()},
    receive
        N ->
            Server ! {set_counter, N+1} %% BAD: race!
    end,
    client(Server).
```

Figure 2.3: A race-prone Erlang program used as example

code, hidden failures, unsatisfiable conditions, redundancies such as unreachable code, etc.) in single Erlang modules or entire applications. Dialyzer[1] is totally automatic, extremely easy to use and particularly successful in identifying software defects which may be hidden in Erlang code, especially in program paths which are not exercised by testing [26]. In fact, since its release, dialyzer has been applied to a significantly large number of programs consisting of several thousand lines of code from real-world telecom applications, and has been surprisingly effective in locating discrepancies in heavily used, well-tested code.

Dialyzer can analyze programs without having to alter their source in any way. The analysis does not even need access to the source, since its starting point is debug-compiled virtual machine bytecode. However, if the source code is indeed available, it can provide

[1] DIscrepancy AnaLYZer for ERlang; www.it.uu.se/research/group/hipe/dialyzer.

the analysis with additional information and perhaps benefit from various kinds of user annotations. The internal language of the analysis to which bytecode and source code are translated, is Core Erlang [7], the official core language for Erlang and the language used internally in the bytecode compiler. Since Core Erlang is on a level close enough to the original source, where it is nearly smooth to reason about the programmer's intentions, it provides `dialyzer` with the means to produce precise and self-explanatory warning messages. This is because Core Erlang introduces a `let` construct which makes the binding occurrence and scope of all variables explicit and helps in retaining line numbers as well as deriving, in a very accurate way, information about the possible values used as arguments to functions that are local to a module.

`Dialyzer` must have its ways to extract some limited form of implicit type information from Erlang code in order to statically find definite type clashes and report them to the user in the form of warnings. Experience with `dialyzer` and its current uses show that it is a sure-fire tool in inferring various forms of non-trivial type information for Erlang programs in a completely automatic and scalable way. This was made viable by `dialyzer`'s *soft type system*, a limited form of type checking that has been developed using *success typings* [20, 16]. Soft typing will not reject any program, but will instead inform the user that the program has some provable type errors. Unlike most soft typing systems that have previously been proposed, success typings [2] allow for compositional, bottom-up type inference which appears to scale well in practice. Moreover, by taking control-flow into account and exploiting properties of the language, such as its module system, success typings can be refined [3] and become accurate and precise.

The details of `dialyzer`'s analyses are beyond the scope of this thesis – we refer the interested reader to the relevant publications [19, 20] – but notable characteristics of its core analysis are:

- `Dialyzer` is a *sound* defect detector – though of course not guaranteed to find all errors – in the sense that it does not report any false positives.

- Currently, `dialyzer` is *a push-button technology and completely automatic*. In particular, it accepts Erlang code "as is" and does not require any annotations from the user, it is very easy to customize and supports various modes of operation (GUI vs. command-line, module-local vs. application-global analysis, using analyses of different power, focusing on certain types of discrepancies only, etc.).

- Its basic analysis is typically *quite fast*, making `dialyzer` an integrated component of Erlang development.

The core analysis is supported by various components for creating and manipulating function call graphs for a higher-order language (which also requires *escape* analysis), taking

[2]If the arguments of an application are in the function domain, the application *might succeed*, but if they are not, the application will *definitely fail*.

[3]A refined success typing is also a success typing but the domain has been constrained and thus, the range can possibly be constrained as well.

control-flow into account, efficiently representing sets of values and computing fixpoints, etc. Nowadays, `dialyzer` is used extensively in the Erlang programming community and is often integrated in the build environment of many applications.[4] However, we note that `dialyzer`'s analysis was restricted to detecting defects in the sequential part of Erlang when we started our work. Before we see how we extended its analysis to also detect races, let us first make more precise what exactly race conditions are in Erlang.

[4]A survey of tools for developing and testing Erlang programs [22], published in the fall of 2008, showed that `dialyzer` is by a wide margin the software tool which is the most widely known (70%) and used (47%) by Erlang developers.

Chapter 3

Race Conditions in Erlang

Naïvely, one may think that race conditions are impossible in Erlang. After all, the language is often advertized as supporting a *shared nothing concurrency* model [2, 18]. A Google search on the term might even convince some readers that this is indeed the case. For example, the Wikipedia article on concurrent computing currently mentions that "Erlang uses asynchronous message passing with nothing shared"[1]. If nothing is shared between processes, how can there be race conditions? In reality, the "nothing shared" slogan is an oversimplification: both of the language's *copying semantics*, which e.g. allows for a shared memory implementation of processes, and of its actual implementation by Ericsson. While it is indeed the case that the Erlang language does not provide any constructs for processes to create and modify shared memory, applications written in Erlang/OTP often employ (and rely upon) built-in features, such as message passing and term storage, which allow processes to share data, make decisions based on the values of this data and destructively update them.

This is exactly what leads to data races in programs and the definition of race conditions we adopt in this thesis: "a race occurs when two threads (or processes) can access (read or write) a data variable simultaneously, and at least one of the two accesses is a write" [15]. Intuitively, we think of race conditions occurring when a process reads some variable and then decides to take some action based on the value of that variable. If it is possible for another process to succeed in changing the value stored on that variable in between the read and the action in such a way that the action about to be taken is no longer appropriate, then we say that our program has experienced a race condition.

In this respect, it is easily understood that the nature of errors caused by race conditions can by very subtle – races do not often crash the system; they manifest themselves as corrupt or wrong memory contents instead, and make debugging extremely frustrating. As a result, race detection tools are valuable both to programmers, by offering them a helping hand in needy times, and to concurrent programs, by increasing their reliability.

In the context of Erlang programs, use of certain Erlang/OTP built-ins leads to data races between processes. Let's first see the simplest of them.

[1]`http://en.wikipedia.org/wiki/Concurrent_computing` (September 2009).

3.1 Data Races in the Process Registry

In Erlang, each created process has a unique identifier (known as its "pid"), which is dynamically assigned to the process upon its creation. To send a message to a process one must know its pid. Besides addressing a process by using its pid, there is also a mechanism, called the *process registry*, which acts as a node-local name server, for registering a process under a certain name so that messages can be sent to this process using that name. Names of processes are currently restricted to atoms. The virtual machine of Erlang/OTP provides built-ins:

`register(Name,Pid)` which adds a table entry associating a certain `Pid` with a given `Name` and generates a run-time exception if the `Name` already appears in the registry,

`registered()` which returns the list of names of all registered processes, and

`whereis(Name)` which returns the pid associated with `Name` or the special value `undefined` if no process is currently registered under the given `Name`.

The registry holds only *live* processes; processes that finish their execution or crash (e.g., due to some uncaught exception) get automatically unregistered.

Many programs manipulating the process registry are written in a defensive programming style similar to the code shown in Figure 3.1.

```
proc_reg(Name) ->
    ...
    case whereis(Name) of
      undefined ->
        Pid = spawn(...),
        register(Name,Pid);
      Pid -> % already
        ok   % registered
    end,
    ...
```

Figure 3.1: A function manipulating the process registry which contains a race condition

This code contains a race condition if executed concurrently by two or more processes. Figure 3.2 shows an interleaving of the concurrent execution of two processes running the code of the `proc_reg` function. This interleaving will result in a runtime exception at the point when P_2 will attempt to register the process with pid $\texttt{Pid}_2$ under a name which has already been inserted in the process registry by process P_1. As a result of this exception, P_2 will crash.

That process P_2 will crash is bad alright, but this is not the only problem of this code. Another problem here is that any action that P_2 has taken between the `whereis` and

P_1	P_2
`proc_reg(gazonk)`	
`...`	`proc_reg(gazonk)`
`whereis(gazonk)`	
	`...`
`Pid`$_1$ `= spawn(...)`	
	`whereis(gazonk)`
`register(gazonk,Pid`$_1$`)`	
	`Pid`$_2$ `= spawn(...)`
	`register(gazonk,Pid`$_2$`)`

Figure 3.2: An interleaving of two processes running the code of the `proc_reg` function

`register` calls that affects the state needs to be undone. In our example execution Pid_2 is now a ghost process. In more involved examples, many more actions affecting the state may occur in between these two calls.

The real problem with the program of Figure 3.1 is that the code between the `whereis` and the `register` calls needs to execute *atomically* but Erlang currently lacks a construct that allows programmers to express this intention. Not only there is no construct like `atomic` in Erlang, but there is also nothing that can be conveniently used as a mutex to protect blocks containing sequences of built-in function calls. In the single-threaded implementation of Erlang/OTP, the probability of a process exhausting its reductions somewhere between the `whereis` and `register` calls is really low, especially if the two calls are so close to each other as in our example, so the race condition is there alright but the actual race is quite unlikely to occur in practice. Not so in the multi-threaded version of Erlang/OTP which nowadays is more or less ubiquitous. Similar problems exist in code that uses a call to the `registered` built-in to make a decision whether to register some process under a name or not, although such code is considerably less common.

3.2 Data Races in the Erlang Term Storage

The second category of data races are races related to the Erlang Term Storage (ETS) facility of Erlang/OTP. This facility provides the ability to store very large quantities of data, organized as a set of dynamic tables in memory, and to have effectively constant time access to this data. Each ETS table is created by a process using the `ets:new(Name,Options)` built-in and is given a `Name` which then can be used to refer to this table (in addition to the table identifier, "tid", which is the return of the `ets:new/2` built-in). Access rights can also be specified for the table by declaring it (in `Options`) as `private`, `protected`, or `public`. Any process can read from or write to tables that are public. Reading and writing happens primarily with the built-ins:[2]

`ets:lookup(Table,Key)` which returns a list of objects currently associated with the given `Key` in the `Table` (which is a name or a tid), and

`ets:insert(Table,Object)` which inserts an `Object` (a tuple with its first position designated as a key) to a given `Table`.

The program of Figure 3.3 shows a made up example of Erlang code which contains an ETS-related race condition. Note that function `ets_inc` has a race condition only if the ETS table, which is created outside this function, is designated as `public`.

```
run() ->
    Tab = ets:new(some_tab_name,[public]),
    Inc = compute_inc(),
    Fun = fun () -> ets_inc(Tab,Inc) end,
    spawn_some_processes(Fun).

ets_inc(Tab,Inc) ->
    case ets:lookup(Tab,some_key) of
      [] ->
        ets:insert(Tab,{some_key,Inc});
      [{some_key,OldValue}] ->
        NewValue = OldValue + Inc,
        ets:insert(Tab,{some_key,NewValue})
    end.
```

Figure 3.3: Program containing a race condition related to ETS

3.3 Data Races in the Mnesia Database

The last category of race conditions we examine are those related to mnesia [21], the distributed Database Management System of Erlang/OTP. Being a database system, mnesia actually contains constructs for enclosing series of table manipulation operations into atomic transactions and there is support to automatically deal with data races which are part of a transaction. However, for performance reasons, mnesia also provides a whole bunch of *dirty* operations – among them `mnesia:dirty_read(Table,Key)` and `mnesia:dirty_write(Table,Record)` – which, as their name suggests, perform database reads and writes without any guarantees that they will not cause data races when executed concurrently. Despite the warning in their name, these dirty operations are used

[2]The `ets` module contains more built-ins for reading from and updating ETS tables, e.g., `ets:lookup_element(Table,Key,Pos)` and `ets:insert_new(Table,Object)`, but we do not describe them here as their treatment is similar to `lookup` and `insert`.

more often than they really need to in applications. Figure 3.4 shows a function from the code of the **snmp** application of Erlang/OTP R13B01.

```
-export([table_func/2]).

table_func(...) ->
    create_time_stamp_table(), ...

create_time_stamp_table() ->
    Props = [{type,set}, ...],
    create_table(time_stamp,Props,ram_copies,false),
    NRef =
      case mnesia:dirty_read(time_stamp,ref_count) of
        [] -> 1;
        [#time_stamp{data = Ref}] -> Ref + 1
      end,
    mnesia:dirty_write(#time_stamp{data = NRef}).
```

Figure 3.4: Program containing a race condition related to mnesia

Having presented the most commonly occurring race conditions in Erlang, which also happen to be the categories of race conditions that our tool currently detects, let us now present the static analysis that we use to detect them.

Chapter 4

Architecture and Implementation

4.1 Desiderata

Before we describe the core of our analysis, we enumerate the goals and requirements we set for its implementation before we embarked on it:

- Our method should be *sound*: it should aim to maximize the number of reported race conditions, but should not generate any false positives meaning that it should not warn for race conditions that are impossible to occur.

- All race conditions detected by the tool should be reported to the user in the form of *precise* warnings. No great ability should be required for their understanding.

- The tool should request no effort or guidance from its user. In particular, the user should not be required to do changes to existing code like providing unsafe operation information, specifying which processes may be interleaved and which must run atomically or writing other such annotations. Instead, the tool should be *completely automated* and able to analyze large, concurrent Erlang applications on its own.

- The analysis should be *fast and scalable* so as to constitute a consistent and smoothly integrated component of `dialyzer`.

4.2 The Analysis

No doubt the reader has noticed that all the examples of race conditions we presented in the previous chapter have some characteristics in common. They all involve a built-in that reads a data item, some decision is then taken based on the value which was read, and execution continues with a built-in performing a write operation of the same data item on either some (Figure 3.1) or on all execution paths (Figure 3.3) following the read. Of course, that our examples follow this pattern is not a coincidence. After all, this pattern reflects the definition of race conditions we gave in the beginning of Chapter 3. However, one should not conclude that detecting this small code pattern is all that our analysis needs to do. In the programs we want to handle, the built-ins performing the reads and writes may be spatially far apart, they may be hidden in the code of higher-order functions, or even be located in different modules. In short, race detection in Erlang requires *control-flow analysis*. Also, the race detection needs to be able to reason about *data-flow*: if at some program point the analysis locates a call to say `whereis(N)` and from that point on control reaches a program point where a call to `register(M,Pid)` appears, the analysis has to determine whether `N` and `M` can possibly refer to the same process name or not. If they can, we have detected a possible race condition; otherwise, there is none. Finally, to avoid a large number of false alarms, the analysis has to take language characteristics into account. For example, the fact that in Erlang only *escaping* functions (i.e., functions that are exported from a module or function closures returned as results) can be used in some `spawn`.

Conceptually, the analysis has three distinct phases: an initial phase that scans the code to collect information needed by the subsequent phases, a phase where all code points with possible race conditions are identified as suspects, and a phase where suspects that are clearly innocent are filtered out. For efficiency reasons, the actual implementation blurs the lines separating these phases and also employs some optimizations. Let's see all these in detail.

4.2.1 Collecting Information for the Analysis

We have integrated our analysis in `dialyzer` because many of the components that it relies upon were already available or could be easily extended to provide the information that the analysis needs. The analysis starts by the user specifying a set of directories/files to be analyzed. Rather than operating directly on Erlang source, all of `dialyzer`'s passes operate at the level of Core Erlang [7], the language also used internally by the Erlang compiler. Core Erlang significantly eases analysis and optimization by removing all syntactic sugar and by introducing a `let` construct which makes the binding occurrence and scope of all variables explicit.

As the source code is translated to Core Erlang, `dialyzer` constructs the *control-flow graph* (CFG) of each function or function closure and then uses a simplified version of the escape analysis of Carlsson et al. [8] to determine closures that escape their defining function. For example, for the code of Figure 3.3 the escape analysis will determine that function `run` defines a function closure that escapes this function as it is used as an argument to function `spawn_some_processes`, which presumably uses this argument in some `spawn`. Given this information, `dialyzer` also constructs the *inter-modular call graph* of all functions and closures, so that subsequent analyses can use this information to speed up their fixpoint computations. For the example in the same figure, the call graph will contain three nodes for functions whose definitions appear in the code (functions `run`, `ets_inc`, and the closure) and an edge from the node of the function closure to that of `ets_inc`.

Besides control-flow, the analysis also needs data-flow information and more specifically it needs information whether variables can possibly refer to the same data item or not. Without race detection this information is not explicitly maintained by `dialyzer`, so we added a *sharing/alias analysis* component that computes and maintains this information. The precision of this analysis is often helped by the fact that `dialyzer` computes type information at a very fine-grained level. For example, different atoms $a_1, \ldots, a_n$ are represented as different *singleton types* in the type domain and their union $a_1 | \ldots | a_n$ is mapped to the supertype $atom()$ only when the size of the union exceeds a relatively high limit [20]. We will see how this information is used by the race analysis in Section 4.2.3.

4.2.2 Determining Code Points with Possible Race Conditions

The second phase of the analysis collects pairs of program points possibly involved in a race condition. These pairs are of the form $\langle P_1, P_2 \rangle$ where P_1 is a program point containing a read built-in (e.g., `whereis`, `ets:lookup`, ...) and P_2 is a program point containing a write built-in (e.g., `register`, `ets:insert`, ...) and such that there is a control-flow path from P_1 to P_2.

In order to collect these pairs, we need to inspect every possible execution path of the program. To this end, we find the root nodes in the inter-modular call graph and start by traversing their CFGs using depth-first search. This depth-first search starts by identifying program points containing a read built-in and then tries to find a program point “deeper” in the graph containing a write built-in. In case a call to some other function is encountered and this function is statically known, the traversal continues by examining its CFG. The case of unknown higher-order calls, as in the code of Figure 4.1 where the `Fun(N)` call is a call to some unknown closure, requires special care: for soundness, the search needs to continue the traversal starting from all root nodes corresponding to a function of arity one. The analysis continues until every path is traversed. Loops also require special attention. A pre-processing step detects cycles in the call graph and checks whether a write built-in is followed by a read built-in in some path in that cycle. Eventually, this exhaustive traversal creates the complete set of pairs of program points where race conditions are possible.

```
foo(Fun,N,M) ->
  ...
  case whereis(N) of
    undefined ->
      ...,
      Fun(M);
    Pid -> ...
  end,
  ...
```

Figure 4.1: Example of an unknown higher-order call

4.2.3 Filtering False Alarms

There are two main problems in what we have just described. There is an obvious performance problem related to the search being exhaustive and there is a precision problem in that the candidate set of race conditions may contain many false alarms. We deal with the latter problem in this section.

We can filter out the majority of false alarms by taking variable sharing, type information, and the characteristics of the race conditions we aim to detect into account. For example, for the case of function `foo` above consider the set of functions that `Fun` can possibly refer to which directly or indirectly lead to a call to `register`. The set of possible race conditions will consist of pairs $\langle P_w, P_{r_i} \rangle$ where P_w denotes the program point corresponding to the `whereis` call in `foo` and P_{r_i} denotes the program points corresponding to the `register` calls. For simplicity, let us assume that in all these `register` calls their first argument is a term which shares with `M` (i.e., it is `M` or a variable which is an alias of `M`). Finally let $A_{\texttt{N}}$ and $A_{\texttt{M}}$ denote the set of atoms that type analysis has determined as possible values for `N` and `M` respectively. If $A_{\texttt{N}} \cap A_{\texttt{M}} = \emptyset$ then all these race conditions are clearly false alarms and can be filtered out. Note that what we have just described is actually the complicated case where the call leading to the write built-in is a call to some unknown function. In most cases, function calls are to known functions which makes the filtering process much simpler. Similarly, there are many cases where $A_{\texttt{N}}$ or $A_{\texttt{M}}$ are singleton sets, which also simplifies the process. Similar filtering criteria, regarding the name of the table, are applied to race conditions related to ETS and mnesia. In addition, ETS-related possible data races which do not involve a `public` table or that involve objects associated with different keys are also filtered out in this analysis phase.

4.2.4 Some Optimizations

Although we have described the computing and filtering phases of the analysis as being distinct, our implementation blurs this distinction, thereby avoiding the exhaustive search and speeding up the analysis. In addition, we also employ the following optimizations:

Control-flow graph and call graph minimization. The CFGs that `dialyzer` constructs by default contain the complete Core Erlang code of functions. This makes sense as most of its analyses, including the type and sharing analyses, need this information. However, note that the path traversal procedure of Section 4.2.2 requires only part of this information. For example, in the program illustrated in Figure 3.4, both the `Props` variable assignment and the list construction on the same line, as well as the complete code of the `case` statement are irrelevant for determining the candidate set of race conditions. Our analysis takes advantage of this by a pre-processing step that removes all this code from the CFGs and by recursively removing CFGs of *leaf* functions that do not contain any calls to the built-ins we search for. In the same spirit, CFGs of functions that are not reachable from some escaping function (i.e., from a root node of the traversal) are also removed.

Avoiding repeated traversals and benefiting from temporal locality. After the call graph is minimized as described above, the depth-first CFG traversal starts from some root. The traversal of all paths from this root often encounters a split in the CFG (e.g., a point where a `case` statement begins) which is followed by a CFG join (the point where the `case` statement ends). All the "straight-line code" which lies between the join point and the next split, including any straight-line code in CFGs of functions called there, does not need to be repeatedly traversed if it is found to contain no built-ins during the traversal of its first depth-first search path. This optimization effectively prunes common sub-paths by condensing them to a single program point. Another optimization is to collect, during the construction of the CFGs of functions, the set of program points containing read and write built-ins that result in race conditions and perform a search focussed around these points, effectively exploiting the fact that in most programs pairs of program points that are involved in race conditions are temporally close to each other (i.e., not necessarily in the same function but only a small number of function calls apart).

Making unknown function calls less unknown. When we described how unknown higher-order calls like `Fun(N)` are handled, we made the pessimistic assumption that `Fun` can refer to any function with arity one. This is correct but way too conservative. By taking into account information about the type of `N` and of the return value of the function, the set of these functions can be reduced, often significantly so. Even though in Erlang there is no guarantee that calls will respect the type discipline, calls that do not do so will result in a crash which is a defect that `dialyzer` will report to its user anyway, albeit in another defect category. The user can correct this first and re-run the analysis.

Clearly, the implementation of these optimization ideas has a heavy impact on the effectiveness and performance of our method. Let us therefore evaluate it on a suite of large, widely used Erlang applications.

Chapter 5

Experimental Evaluation

5.1 Dialyzer in Action

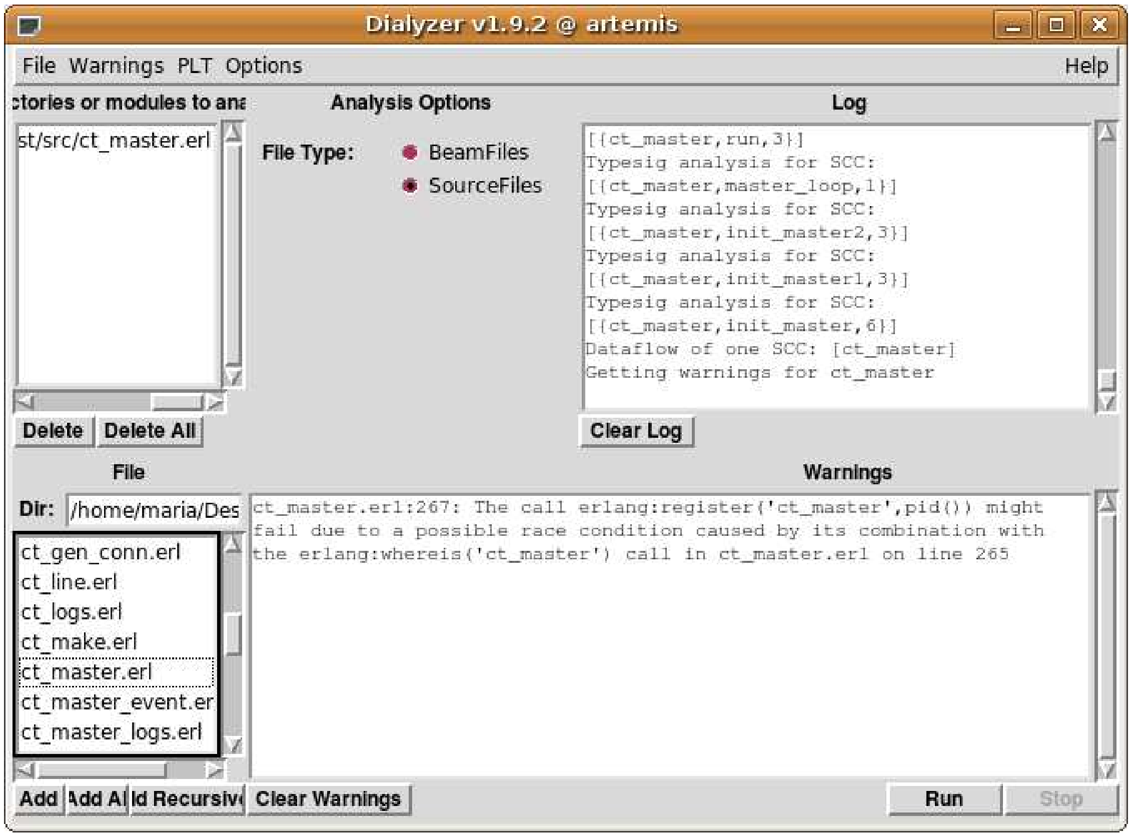
5.1.1 Using Dialyzer from its Graphical User Interface

Figure 5.1 shows `dialyzer`'s GUI in action. In fact, the snapshot depicts `dialyzer` detecting the possible race conditions in `ct_master.erl`, a module of the **common_test** application of Erlang/OTP R13B01.

In the "File" window, there is a listing of the current directory and we can either click our way to the directories or modules we want to "dialyze" or type the correct path in the entry. We can mark the directories or modules for discrepancy analysis and click "Add". In other words, we can add `.beam` and `.erl`-files directly, or indirectly, by adding directories that contain these kinds of files. However, we can only add one type of files to be analyzed, which is specified by the current analysis mode controlled in the top-middle part of the main window under "Analysis Options" – we cannot mix `.beam` and `.erl`-files.

Under the "Warnings" pull-down menu, there are buttons that control which discrepancies are reported to the user in the "Warnings" window. By clicking on these buttons, one can enable/disable a whole class of warnings – race condition detection being one of them.

Once we have chosen the modules or directories for the analysis, we can click on the "Run" button to start the analysis. If, for some reason, we want to stop the analysis while it is running, we can push the "Stop" button. In every case, the information from the analysis will be displayed in the "Log" and "Warnings" windows as shown in Figure 5.1.



```
[{ct_master,run,3}]
Typesig analysis for SCC:
[{ct_master,master_loop,1}]
Typesig analysis for SCC:
[{ct_master,init_master2,3}]
Typesig analysis for SCC:
[{ct_master,init_master1,3}]
Typesig analysis for SCC:
[{ct_master,init_master,6}]
Dataflow of one SCC: [ct_master]
Getting warnings for ct_master
```

```
ct_master.erl:267: The call erlang:register('ct_master',pid()) might
fail due to a possible race condition caused by its combination with
the erlang:whereis('ct_master') call in ct_master.erl on line 265
```

Figure 5.1: `Dialyzer`'s GUI version

5.1.2 Using Dialyzer from the Command Line

There is also a command line version of `dialyzer` for automated use. Below follows a brief description of the tool options that are most relevant to this thesis.
Usage:

```
dialyzer [-Wpossible_races] [--src] [-c applications] [-r applications]
```

Options:

`-c applications` (or `--command-line applications`)
: Use `dialyzer` from the command line (no GUI) to detect defects in the specified applications (directories or `.erl` or `.beam` files).

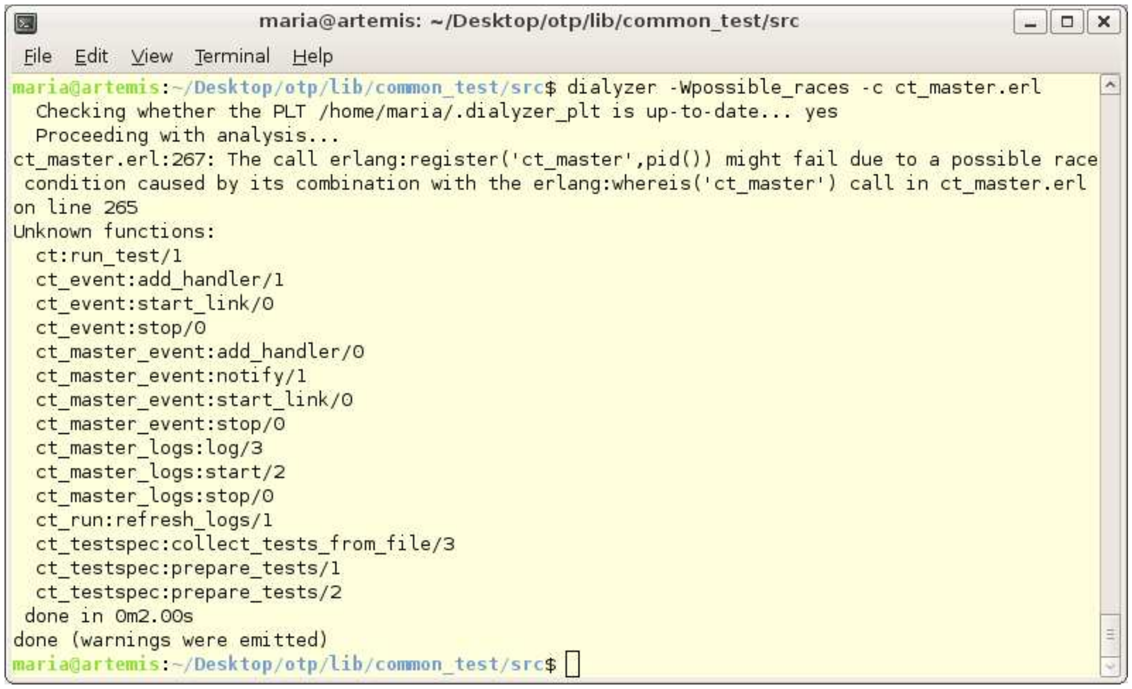
`-r applications`
: Same as `-c` only that directories are searched recursively for subdirectories containing `.erl` or `.beam` files (depending on the type of analysis).

`--src`
: Override the default, which is to analyze `.beam` files, and analyze starting from Erlang source code instead.

The exit status of the command line version is:

```
0 - No problems were encountered during the analysis and no
    warnings were emitted.
1 - Problems were encountered during the analysis.
2 - No problems were encountered, but warnings were emitted.
```

Figure 5.2 shows `dialyzer`'s command line version in action.


maria@artemis: ~/Desktop/otp/lib/common_test/src
File Edit View Terminal Help

```
maria@artemis:~/Desktop/otp/lib/common_test/src$ dialyzer -Wpossible_races -c ct_master.erl
  Checking whether the PLT /home/maria/.dialyzer_plt is up-to-date... yes
  Proceeding with analysis...
ct_master.erl:267: The call erlang:register('ct_master',pid()) might fail due to a possible race
 condition caused by its combination with the erlang:whereis('ct_master') call in ct_master.erl
on line 265
Unknown functions:
  ct:run_test/1
  ct_event:add_handler/1
  ct_event:start_link/0
  ct_event:stop/0
  ct_master_event:add_handler/0
  ct_master_event:notify/1
  ct_master_event:start_link/0
  ct_master_event:stop/0
  ct_master_logs:log/3
  ct_master_logs:start/2
  ct_master_logs:stop/0
  ct_run:refresh_logs/1
  ct_testspec:collect_tests_from_file/3
  ct_testspec:prepare_tests/1
  ct_testspec:prepare_tests/2
 done in 0m2.00s
done (warnings were emitted)
maria@artemis:~/Desktop/otp/lib/common_test/src$
```

Figure 5.2: `Dialyzer`'s command line version

5.1.3 Using Dialyzer from Erlang

Naturally, `dialyzer` can also be used directly from an Erlang shell like any other Erlang application.

5.2 Measurements

The analysis we described in the previous chapter has been implemented and incorporated in the development version of `dialyzer`. We have paid special attention to integrate it smoothly with the existing analyses, reuse as much of the underlying infrastructure as possible, and fine-tune the race detection so that it incurs relatively little additional overhead to `dialyzer`'s default mode of use. The main module of the race analysis is about

2,200 lines of Erlang code and the user can turn on race detection either via a GUI button or a command-line option.

We have measured the effectiveness and performance of the analysis by applying it on a corpus of Erlang code of significant size: more than a million lines of code. In this thesis we restrict our attention to the Erlang/OTP libraries described below. Needless to mention that Erlang/OTP libraries are heavily used in Erlang programs.

asn1 Provides support for Abstract Syntax Notation One

common_test A portable framework for automatic testing

gs A Graphics System used to write platform independent user interfaces

hipe The native code compiler of Erlang (HIgh Performance Erlang)

kernel Functionality necessary to run the Erlang system itself

mnesia A heavy duty real-time distributed database

otp_mibs SNMP management information base for Erlang/OTP nodes

percept A concurrency profiler tool

runtime_tools Runtime tools, tools to include in a production system

snmp Simple Network Management Protocol (SNMP) support including a Management Information Base compiler and tools for creating SNMP agents

stdlib The Erlang standard libraries

tv An Erlang Term Store (ETS) and mnesia graphical Table Visualizer

Table 5.1 shows the lines of code (LOC) of each application, the number of race conditions detected (total and categorized as being related to the process registry, to ETS or to mnesia), and the elapsed wall clock time (in minutes) and memory requirements (in MB) for running `dialyzer` without and with the race condition detection on these programs. The performance evaluation was conducted on a machine with a dual processor[1] Intel Pentium 2GHz CPU with 3GB of RAM, running Linux.

5.3 Performance Analysis

We will now attempt to evaluate the performance of our race detection method in terms of the measurements shown in Table 5.1.

Regarding the total memory use, our analysis, when invoked, requires an amount of space that depends on the application we want to “dialyze”. This is a direct consequence

[1]However, most of `dialyzer`’s analyses, including the race detection, use only one core.

Table 5.1: `Dialyzer`'s race detection performance on Erlang/OTP applications

		Number of Races				Time (mins)		Space (MB)	
Application	LOC	Total	ProcR	ETS	Mnesia	w/o race	w race	w/o race	w race
asn1	38,965	2	2	-	-	3:30	4:04	182	282
common_test	15,573	1	1	-	-	0:22	0:22	74	78
gs	15,819	2	2	-	-	1:00	2:01	111	170
hipe	95,652	0	-	-	-	3:03	3:06	183	333
kernel	36,618	6	4	-	2	1:00	1:05	86	130
mnesia	24,730	0	-	-	-	0:56	3:23	87	157
otp_mibs	196	2	-	-	2	0:00	0:00	32	33
percept	4,457	3	3	-	-	0:11	0:11	40	43
runtime_tools	8,277	2	2	-	-	0:28	0:28	62	71
snmp	52,071	6	-	3	3	1:54	2:00	141	192
stdlib	72,297	1	1	-	-	6:23	6:45	189	310
tv	20,050	1	1	-	-	0:13	0:13	71	72

of the fact that the analysis needs to examine both control and data-flow information about an application in order to be able to detect any race conditions. As one might expect, larger applications usually have more control and data-flow information and thus, the memory use for these applications is greater.

Considering the total execution time, the "dialyzing" of applications is practically unaffected by the extra overhead of collecting information for the analysis. However, the execution time naturally increases when our analysis identifies more than a few program points containing a read built-in in the depth-first search of the CFGs. This is because for each one of these read built-ins our method tries to find a program point "deeper" in the graph containing a write built-in even if this generally involves repeated traversals of the same paths. The execution time also depends on the number and length of these paths as well as on the number of function or closure calls encountered in them. At this point, it is reminded that when the search finds a statically known function or closure call, the traversal continues by examining its CFG. In case of an unknown call, the depth-first search continues the traversal starting from all root nodes corresponding to a function of the same arity and type information about the argument and return values. Consequently, the larger the number of function or closure calls, the larger the size of the graph that needs to be investigated for write built-ins. Table 5.2 shows the measurements that were conducted to confirm the exact causes of the execution time diversity observed in the previous section – we measured the number of read built-ins identified in the depth-first search, the number of function or closure calls between the read and write built-ins in the graph and the race analysis space overhead compared to the default indicating the size of the CFGs.

For example, although the read built-ins in the **snmp** application are many more than

Table 5.2: `Dialyzer`'s execution time affecting factors

Application	Read Built-Ins	Function or Closure Calls	Space Overhead (MB)	Time (mins) w/o race	Time (mins) w race
asn1	15	6,264	100	3:30	4:04
common_test	13	1,705	4	0:22	0:22
gs	28	4,623	59	1:00	2:01
hipe	1	11	150	3:03	3:06
kernel	132	1,433	44	1:00	1:05
mnesia	85	16,041	70	0:56	3:23
otp_mibs	5	48	1	0:00	0:00
percept	17	203	3	0:11	0:11
runtime_tools	12	234	9	0:28	0:28
snmp	136	11,150	51	1:54	2:00
stdlib	60	2,929	121	6:23	6:45
tv	2	436	1	0:13	0:13

in the **mnesia** application, the race analysis execution time is much closer to the default. This is because the CFGs that are traversed are smaller in the **snmp** application and there are less function or closure calls to be investigated.

Overall, given that context-independent control-flow analysis is cubic in the worst case, the tool performs quite satisfactorily, with the exception of a few outliers, leaving very little reason not to use it regularly when developing Erlang programs.

5.4 Current Experiences

As it is probably obvious by now, during its development, our race detection analysis has been repeatedly tested. Most notably, it has been applied to all Erlang/OTP applications, the largest of which consists of about 200,000 lines of Erlang code. In the pursuit of experiences from benchmarks besides the Erlang/OTP system, we have also applied `dialyzer` on various other open source and often widely used applications written in Erlang:

CouchDB A distributed, fault-tolerant and schema-free document-oriented database via a RESTful HTTP/JSON API

ejabberd A distributed, fault-tolerant Jabber server

Erlang Web A framework for applications based on HTTP protocols

Scalaris A scalable, transactional, distributed key-value store

Yaws An HTTP, high-performance 1.1 web server, particularly well-suited for dynamic-content web applications

For these applications we used the code from their public repositories at the end of August 2009. Table 5.3 illustrates our measurements.

Table 5.3: `Dialyzer`'s race detection performance on open source applications

		Number of Races				Time (mins)		Space (MB)	
Application	LOC	Total	ProcR	ETS	Mnesia	w/o race	w race	w/o race	w race
CouchDB	22,611	0	-	-	-	0:47	0:53	115	187
ejabberd	72,788	6	1	4	1	0:39	0:40	113	142
Erlang Web	22,229	7	-	7	-	0:33	0:35	115	122
Scalaris	38,770	0	-	-	-	1:13	1:20	179	278
Yaws	37,270	3	3	-	-	1:33	1:39	167	245

Notice that the number of race conditions is significant, especially considering that our technique currently tracks only some specific categories of possible data races in Erlang. Also, in most cases, data race detection adds only a small overhead, both in time and in space, to `dialyzer`'s default analysis.

A more detailed experience report on using `dialyzer` on open source projects is beyond the scope of this thesis, however, it suffices to say that when a code defect is as frustrating to detect as race conditions are, a tool that is able to automatically track them down should be extremely useful.

Chapter 6

Related Work

In this chapter, we discuss related work, including dynamic and static race detection techniques [6, 23] as well as race detection in Erlang.

6.1 Dynamic Race Detection

State-of-the-art race detection tools are primarily dynamic. Although dynamic race detectors are inherently unsound and associated with a serious memory and performance overhead when used on very large applications, they have the big advantage that they require no user interaction since they discover all they need to know from the execution. Of course, these tools are limited only to the runtime execution paths that occur during testing. They can be broadly classified into *happens-before-based*, *lockset-based* and *hybrid*.

Happens-before-based dynamic race detectors rely on a partial ordering of all events of all threads in a concurrent execution such that any violation of this ordering is taken for a race condition. Every thread has a *vector clock* – a timestamp that increases as synchronization events occur – of its own and vector clocks of other threads. For instance, when a synchronization event occurs, all the threads involved have to update and exchange their vector clocks. Shared data also has vector clocks so that it may keep track of the last time it was accessed by each thread. After all these vector clocks, a simple property must hold for a concurrent execution to be race free: when a thread *speedy* performs a read operation on shared data, *speedy*'s vector clocks for all other threads that have previously accessed the data must be less than or equal to the corresponding data vector clocks. The key drawbacks of this technique are that it is difficult to implement efficiently and, although it produces no false positives, it produces many false negatives.

The lockset algorithm was originally implemented in the `Eraser` tool [27] and relies on the common lock-based synchronization discipline: in a concurrent execution, all shared data should be accessed under lock and key. Every shared variable has a *candidate set* of protection locks. When a thread tries to access a shared variable, the algorithm determines the intersection of all the locks held by the thread with the candidate set of the shared variable. Consequently, if at any point this intersection turns out to be empty, the algorithm signals a possible race condition. The primary problem with lockset-based race detection is that it produces many false positives either when synchronization mechanisms other than locks are employed or when the programmer knows that at a certain program point there is no way that a shared variable can be accessed by more than one threads, hence leaves it unprotected. Another problem with the technique is that, as any dynamic analysis, it cannot avoid false negatives.

Since combining the happens-before-based and the lockset-based race detection was first proposed [10], several hybrid algorithms have emerged that benefit from both approaches without exhibiting the disadvantages of either. Two popular hybrid implementations are `MultiRace` [24] and `RaceTrack` [31]. Both tools managed to bring their analyses up to a more practical level than the existing dynamic tools of the time. For instance, existing tools monitored the shared memory at the granularity of an object, rather than a word, with means to avoid vastly increasing their time and space overhead. However, locks may naturally be used on object fields as well, resulting in numerous false positives. Consequently, once an object becomes a race suspect, `RaceTrack` and `MultiRace` dynamically reduce the memory granularity and proceed with the analysis. An equally important engineering choice that these tools made was the phased enforcement of the lockset and happens-before analyses: the cheaper lockset algorithm is applied first and the validity of the potential race conditions it detects is then verified by the happens-before analysis, increasing both the precision and the synchronization discipline expressiveness of these hybrid dynamic race detectors.

6.2 Static Race Detection

Static race detectors employ either flow-insensitive analyses based on types, or flow-sensitive static versions of the lockset algorithm, or are based on model checking.

Type-based systems specify the synchronization discipline by means of types; namely, a program that type-checks is race free. In these systems, race condition prevention relies on the idea that shared data must be protected with a lock, ergo by integrating the lock into the shared data type, we can ensure that any subsequent appearances of that data are confined only in code that acquired the particular lock. For files and classes to be compilation independent, type systems also use function *effects clauses* that suggest which locks must be acquired by any caller of a function. Although these systems impose no performance penalty and are by natural means familiar to the programmer, they tend to limit the expressiveness and precision of their programming languages; race free programs that may obviate the need for locking either because their thread interleavings are known in advance or because their shared data is being initialized before the threading begins, are not allowed. Besides, a fair number of annotations is required, thus making their use for large applications quite unlikely despite their scalability.

Soundness, precision and the amount of annotations required seem to be correlative dimensions of comparison for the flow-sensitive tools. For instance, a high precision analysis would certainly have to sacrifice either its soundness or a possibly small number of annotations. Furthermore, these analyses tend to be both expressive when the programmer knowledge is hard-coded in the form of annotations as well as scalable since they are usually aimed for industrial use. Two static versions of the lockset algorithm we are aware of for C are `Warlock` [29] and `RacerX` [11]. `Warlock` does not trace paths through loops or recursive functions while `RacerX` pushes the envelope by using heuristics, statistical analysis and ranking to produce a high quality set of concurrency bugs.

Model checking is a simple, yet powerful, technique. Broadly speaking, it performs some sort of simulation of the application by exploring all possible execution paths for all possible data in order to detect any undesirable behaviours. Examining all possible thread interleavings and for each interleaving every possible control and data-flow is the naïve way to extend model checking to concurrent applications. Undoubtedly, we are talking exponential; this algorithm could soon bring about a combinatorial explosion even for small applications. In short, model checking must find ways to ally with pruning and fight against time.

Other static race detection approaches include language-based ones such as `nesC` [13] for C and `Guava` [5] for Java.

Table 6.1 presents a comparison and overview of the race detection techniques discussed so far in this chapter.

Table 6.1: Rough characterizations of different race detection techniques

Technique	Annotation	Expressiveness	Scalability	Soundness	Precision
Type-based system	High	Low	High	Yes	Low
Dynamic race detection	None	Medium	Medium	No	Medium
Model checking	None	High	Varies	Yes	High
Flow-sensitive analysis	Varies	Medium	High	Varies	Varies

6.3 Race Detection in Erlang

`QuickCheck` [4] is an Erlang *property-based testing* tool. `QuickCheck` users write properties that should hold and test their running code against them. The tool uses controllable random test case generation combined with automated test case simplification for a more painless error diagnosis. Very recently, `QuickCheck` came with an extension to unit test Erlang programs for race conditions. A concurrent test case passes successfully, if there exists some ordering of the test's function calls for which the user specification still holds. In case the test fails, `PULSE` may help out in tracing what really happened. `PULSE` is a ProTest User Level Scheduler for Erlang that randomly schedules the test case processes and records a detailed trace. This method is only semi-automatic as it relies on the user to specify, using a special QuickCheck module (`eqc_par_statem`) that models a parallel state machine, the properties for which to test for possible atomicity violations. As a case study, the method was applied to a small (200-line) Erlang program detecting two race conditions.

While we prefer our method because it is completely automatic and more scalable, the two methods are actually complementary to each other. `Dialyzer` cannot detect one of the two race conditions in that program because this race depends on the semantics of the operations which are supplied by the user (in the form of properties that should hold). The other race condition could be detected by `dialyzer` if its analysis were enhanced with information about the behaviour of the `gen_server` module of Erlang/OTP. More generally, it is clear that in both tools the more the information which is supplied to them about which operations and built-ins can cause atomicity violations, the more the race conditions that they can detect. But a fundamental difference between them is that in our tool this responsibility lies in the hands of the tool implementor while in `QuickCheck`'s case in the programmer's.

With the exception of `QuickCheck`, we are not aware of any other concurrency error detector in Erlang. We hope to see more research and tool development in this direction.

Chapter 7

Concluding Remarks

In this thesis, we characterized the kinds of data races that Erlang programs can exhibit and presented an effective static analysis technique that detects them. By implementing this analysis in a publicly available and commonly used tool for detecting software defects in Erlang programs not only were we able to measure its effectiveness and performance by applying it to several large applications, but we also contribute in a concrete way to raising the awareness of the Erlang programming community on these issues and helping programmers fix the corresponding bugs. Data races are subtle and notoriously difficult for programmers to avoid and reason about, independently of language. In Erlang there are fewer potential race conditions and they are less likely to occur during testing, which unfortunately also makes it less likely that programmers will be paying special attention to watch out for them when programming. Despite the restricted nature of data races in Erlang, our experimental results have shown that the number of race conditions is not negligible even in widely used applications. Tools to detect them definitely have their place in the developer's tool suite.

Various additions to `dialyzer`'s functionality are already planned. We intend to extend our race analysis for the tool to become capable of detecting more of the shifty concurrency errors that may push a programmer over the edge. Enriching `dialyzer`'s specification language [16] for programmers to specify which code fragments are intended to be atomic is another thought on the table since many concurrency errors are manifested as atomicity violations. In every case, our research will continue to be based on the feedback that we seek from experienced Erlang users and large, concurrent Erlang applications.

References

[1] J. Armstrong. A history of Erlang. In *Proceedings of the 3rd ACM SIGPLAN Conference on History of Programming Languages*, pages 1–26, New York, NY, USA, 2007. ACM.

[2] J. Armstrong. *Programming Erlang: Software for a Concurrent World*. The Pragmatic Bookshelf, Raleigh, NC, 2007.

[3] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang*. Prentice Hall Europe, Herfordshire, Great Britain, second edition, 1996.

[4] T. Arts, J. Hughes, J. Johansson, and U. Wiger. Testing telecoms software with Quviq QuickCheck. In *Proceedings of the 5th ACM SIGPLAN Workshop on Erlang*, pages 2–10, New York, NY, USA, 2006. ACM.

[5] D. Bacon, R. Strom, and A. Tarafdar. Guava: A dialect of Java without data races. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 382–400, New York, NY, USA, 2000. ACM.

[6] N. E. Beckman. A survey of methods for preventing race conditions, 2006.

[7] R. Carlsson. An introduction to Core Erlang. In *Proceedings of the PLI'01 Workshop on Erlang*, 2001.

[8] R. Carlsson, K. Sagonas, and J. Wilhelmsson. Message analysis for concurrent programs using message passing. *ACM Trans. Prog. Lang. Syst.*, 28(4):715–746, July 2006.

[9] M. Cronqvist. Troubleshooting a large Erlang system. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Erlang*, pages 11–15, New York, NY, USA, 2004. ACM.

[10] A. Dinning and E. Schonberg. Detecting access anomalies in programs with critical sections. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 85–96, New York, NY, USA, 1991. ACM.

[11] D. Engler and K. Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 237–252, New York, NY, USA, 2003. ACM.

[12] Erlang web site. http://www.erlang.org.

[13] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–11, New York, NY, USA, 2003. ACM.

[14] P. Gustafsson and K. Sagonas. Bit-level binaries and generalized comprehensions in Erlang. In *Proceedings of the 4th ACM SIGPLAN Workshop on Erlang*, pages 1–8, New York, NY, USA, 2005. ACM.

[15] T. Henzinger, R. Jhala, and R. Majumdar. Race checking by context inference. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–13, New York, NY, USA, 2004. ACM.

[16] M. Jimenez, T. Lindahl, and K. Sagonas. A language for specifying type contracts in Erlang and its interaction with success typings. In *Proceedings of the 6th ACM SIGPLAN Workshop on Erlang*, pages 11–17, New York, NY, USA, 2007. ACM.

[17] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[18] J. Larson. Erlang for concurrent programming. *ACM Queue*, 6(5):26–38, 2008.

[19] T. Lindahl and K. Sagonas. Detecting software defects in telecom applications through lightweight static analysis: A war story. In C. Wei-Ngan, editor, *Programming Languages and Systems: Proceedings of the Second Asian Symposium*, volume 3302 of *LNCS*, pages 91–106, Berlin, Germany, 2004. Springer.

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

[21] H. Mattsson, H. Nilsson, and C. Wikström. Mnesia - a distributed robust DBMS for telecommunications applications. In G. Gupta, editor, *Practical Applications of Declarative Languages: Proceedings of the PADL Symposium*, volume 1551 of *LNCS*, pages 152–163, Berlin, Germany, 1999. Springer.

[22] T. Nagy and A. Nagyné Víg. Erlang testing and tools survey. In *Proceedings of the 7th ACM SIGPLAN Workshop on Erlang*, pages 21–28, New York, NY, USA, 2008. ACM.

[23] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 308–319, New York, NY, USA, 2006. ACM.

[24] E. Pozniansky and A. Schuster. Efficient on-the-fly data race detection in multi-threaded C++ programs. In *Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 179–190, New York, NY, USA, 2003. ACM.

[25] K. Sagonas. Experience from developing the Dialyzer: A static analysis tool detecting defects in Erlang applications. In *Proceedings of the ACM SIGPLAN Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

[26] K. Sagonas and D. Luna. Gradual typing of Erlang programs: A Wrangler experience. In *Proceedings of the 7th ACM SIGPLAN Workshop on Erlang*, pages 73–82, New York, NY, USA, 2008. ACM.

[27] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 27–37, New York, NY, USA, 1997. ACM.

[28] G. Steele and E. Raymond. *The New Hacker's Dictionary*. The MIT Press, Cambridge, MA, third edition, 1996.

[29] N. Sterling. Warlock: A static data race analysis tool. In *Proceedings of the Usenix Winter Technical Conference*, pages 97–106, 1993.

[30] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 2005.

[31] Y. Yu, T. Rodeheffer, and W. Chen. RaceTrack: Efficient detection of data race conditions via adaptive tracking. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, pages 221–234, New York, NY, USA, 2005. ACM.