Static Detection of Race Conditions in Erlang

Maria Christakis[1] and Konstantinos Sagonas[1,2]

[1] School of Electrical and Computer Engineering,
National Technical University of Athens, Greece
[2] Department of Information Technology, Uppsala University, Sweden
{mchrista,kostis}@softlab.ntua.gr

Abstract. We address the problem of detecting 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 dialyzer, a commonly used tool for detecting software defects in Erlang programs which is part of Erlang/OTP, and evaluate its effectiveness and performance on a suite of widely used industrial and open source programs of considerable size. The analysis has detected a significant number of previously unknown race conditions.

1 Introduction

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.

To make concurrent programming simpler and better suited 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 [1]. 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 [2]. In fact, there is documented evidence that race conditions are a serious problem when developing and troubleshooting large industrial Erlang applications [3].

To ameliorate the situation and building upon successful prior work on detecting software defects on the sequential part of Erlang [4,5], we have embarked on a project aiming to detect concurrency errors in Erlang programs using static analysis. In this paper we take a very important first step in that direction by

M. Carro and R. Peña (Eds.): PADL 2010, LNCS 5937, pp. 119–133, 2010.

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 paper 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 demonstrates 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 section overviews the Erlang language and the defect detection tool which is the vehicle for our work. Section 3 describes commonly occurring kinds of data races in Erlang programs, followed by Sect. 4 which presents in detail the analysis we use to detect them. The effectiveness and performance of our analysis is evaluated in Sect. 5 and the paper ends by reviewing related work and some final remarks.

2 Erlang and Dialyzer

Erlang [1] 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. Erlang's primary application area has been 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.

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 provides mechanisms for allowing a process to timeout while waiting for messages and a `try`/`catch`-style exception mechanism for error handling.

In Erlang, 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 recently been exposed.

Since 2007 the Erlang/OTP distribution includes a static analysis tool, called dialyzer [4,5], for finding software defects (such as type errors, exception-raising code, code which has become unreachable due to some logical error, etc.) in single Erlang modules or entire applications. In fact, dialyzer has been surprisingly effective in locating software bugs even in heavily used, well-tested code. Dialyzer[1] is totally automatic, extremely easy to use and supports various modes of operation: command-line vs. GUI, starting the analysis from source vs. byte code, focussing on some kind of defects only, etc. The details of dialyzer's analyses are beyond the scope of this paper — we refer the interested reader to the relevant publications [4,6] — but notable characteristics of its core analysis are that it is *sound for defect detection* (i.e., it produces no false positives), *fast* and *scalable*. 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 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.[2] However, we note that dialyzer's analysis was restricted to detecting defects in the sequential part of Erlang when

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

[2] A survey of tools for developing and testing Erlang programs [7], 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.

we started this work. Before we see how we extended its analysis to also detect data races, let us first see the kinds of race conditions that exist in Erlang.

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 [1]. 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".[3] If nothing is shared between processes, how can there be race conditions? In reality, the "shared nothing" 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 — system built-ins 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 paper: "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". 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 a race condition.

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.

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,

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

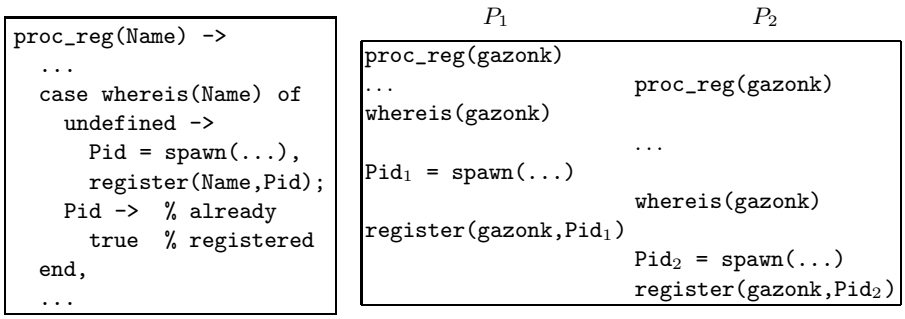

Fig. 1. A function manipulating the process registry which contains a race condition (left) and an interleaving of two processes that shows the race (right)

`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 on the left box of Fig. 1. This code contains a race condition if executed concurrently by two or more processes. The right box of the same figure 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 where 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 unfortunate, 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 `register` calls which affects the state needs to be undone. In our example run, $\texttt{Pid}_2$ is now a ghost process. In more involved examples, many more actions affecting the state may have occurred in code that executed between these two calls.

The real problem with the program of Fig. 1 is that the code that lays 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 is there currently 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 small, especially if the two calls are as close to each other as in our example, thus the race condition is there alright in the program 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`

```
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.
```

```
-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}).
```

Fig. 2. Programs containing race conditions related to ETS and Mnesia

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 those 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:[4]

`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 on the left box of Fig. 2 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`.

3.3 Data Races in the Mnesia Database

The last category of race conditions we examine are those related to mnesia [8], the distributed Database Management System of Erlang/OTP. Being a database

[4] 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`.

system, mnesia actually contains constructs for enclosing series of table manipulation operations into atomic transactions and support to automatically deal with data races which take place within 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 by programmers more often than they really need to in applications. The right box of Fig. 2 shows a function from the code of the `snmp` application of Erlang/OTP R13B01.

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

4 Detecting Race Conditions Using Static Analysis

No doubt the reader has noticed that all the examples of race conditions we presented in the previous section 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 (Fig. 1) or on all execution paths (Fig. 2) 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 Sect. 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 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.

4.1 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 [9], the language used internally by the Erlang compiler. Core Erlang significantly eases analysis and optimization by removing 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. [10] to determine closures that escape their defining function. For example, for the code on the left box of Fig. 2 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 [6]. We will see how this information is used by the race analysis in Sect. 4.3.

4.2 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 on the right where the `Fun(N)` call is a call to some unknown closure, gives us an implementation choice. One option is to ignore such calls. This gives an analysis which is sound for defect detection (i.e., an analysis that completely avoids false alarms). The other option, which gives an analysis sound for correctness (i.e., an analysis that finds all data races but may also produce some false alarms), is to continue the traversal starting from all root nodes corresponding to a function of arity one and continue the analysis until every path is traversed. This exhaustive traversal creates the complete set of pairs of program points where race conditions are possible. Loops 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.

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

4.3 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 false alarms. We deal with the latter problem in this section.

False alarms are avoided by taking variable sharing, type information, and the characteristics of the race conditions we aim to detect into account. Suppose we opt for an analysis that finds all data races. Then, 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 $A_{\mathtt{N}}$ and $A_{\mathtt{M}}$ denote the set of atoms that type analysis has determined as possible values for `N` and `M` respectively. If $A_{\mathtt{N}} \cap A_{\mathtt{M}} = \emptyset$ 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, $A_{\mathtt{N}}$ or $A_{\mathtt{M}}$ are often 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.

The method we have described has the following property. In programs where the function call graph is precise (i.e., when there are no unknown calls or when

the escape analysis offers precise information about these calls) the analysis produces no false alarms.

4.4 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 Sect. 4.2 requires only part of this information. For example, in the program illustrated on the right box of Fig. 2, 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)` could be 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 is 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 these defects first and re-run the race analysis.

5 Experimental Evaluation

The analysis we described in the previous section 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. Another analysis option controls whether the analysis will examine calls to unknown functions or not (Sect. 4.2).

With this option off, 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 paper we restrict our attention to Erlang/OTP libraries and open source applications which were found to contain race conditions in their code. A short description of the code bases we focus on appears in Table 1. All of them are heavily used. For open source applications we used the code from their public repositories at the end of August 2009.

Table 1. Brief description of applications found to contain race conditions

Application libraries from the Erlang/OTP R13B01 distribution	
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
kernel	Functionality necessary to run the Erlang/OTP system itself
otp_mibs	SNMP Management Information Base for Erlang/OTP nodes
percept	A concurrency profiler tool
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 agents
stdlib	The Erlang standard libraries
tv	An Erlang term store and mnesia graphical Table Visualizer
Open source Erlang applications	
ejabberd	A distributed, fault-tolerant Jabber/XMPP application server
Erlang Web	A framework for applications based on HTTP protocols
yaws	(Yet another web server) An HTTP, high-performance 1.1 web server, particularly well-suited for dynamic-content web applications

Table 2. Effectiveness and performance of the race analysis

		Num Race Conditions				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
kernel	36,618	6	4	2	-	1:00	1:05	86	130
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
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
yaws	37,270	3	3	-	-	1:33	1:39	167	245

Table 2 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 analysis that detects race conditions on these programs. The performance evaluation was conducted on a machine with a dual processor Intel Pentium 2GHz CPU with 3GB of RAM, running Linux. (Currently, the analysis utilizes only one core.)

In analyzing these results, first 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. Since the analysis does not examine execution paths starting from statically unknown function calls, it produces no false alarms. In fact, we have manually examined all these race conditions and confirmed that indeed all are possible. Regarding performance, in most cases, data race detection adds only a small overhead, both in time and in space, to dialyzer's default analysis. The only outliers are gs where the analysis time is doubled and stdlib where analysis with race condition detection on requires 66% more space than analysis without. Still, viewed in absolute terms, both the time and the space overhead are reasonable given the size of these applications. Since the analysis is totally automatic, we see very little reason not to use it regularly when developing Erlang programs.

6 Related Work

The problem of detecting data races and other concurrency errors in programs is fundamental and well studied. In the literature one can find various approaches, which can be broadly classified as *static*, *dynamic*, or *hybrid*.

Dynamic race detectors instrument the program and monitor its execution during runtime either using some variant of the *lockset* algorithm [11,12] to see whether the locking discipline (i.e., the assumption that all shared variables must be accessed within the protection of a lock) is violated or by checking whether Lamport's *happens-before* relation between thread accesses to a given piece of data holds. State-of-the-art dynamic detectors are scalable and easy to use but cannot guarantee the absence of races and require comprehensive test suites. Their efficiency and precision can be improved with static analysis, thereby yielding hybrid race detectors [13]. For more information on dynamic and hybrid approaches to race detection we refer the reader to a recent survey [14].

Static approaches either prevent some kinds of races completely by imposing a type system to the language that guarantees the absence of these races if the program type checks, or use path sensitive model checkers or flow sensitive static analyzers to detect them. The latter techniques are more related to what we do, so we examine them more closely. Model checkers find race conditions by considering all possible interleavings in a model of the software which is under scrutiny and try to fight combinatorial explosion by using various clever representations of the search space and heuristics to cut down the number of interleavings that need to be explored. The key advantage of model checkers is that they detect actual data races and often also produce counterexamples for them. On the other hand, existing software model checkers do not scale to the size of programs we need to handle. Moreover, it is not clear what the property to check should be since the kinds of atomicity violations that our tool detects are not easily expressible in the language of most model checkers. Static analyzers have been shown to be more scalable. They either employ a static version of the lockset algorithm [15,16], flow sensitive analysis [17,18,19], or are based on abstract interpretation [20]. A big challenge for static analyzers is to strike a proper balance between soundness and precision. Soundness is often threatened by how well they abstract certain nasty features of the language [16] or by the effectiveness of the alias and escape analyses that they employ [17,18]. Most analyzers try to reduce the number of false alarms either using heuristics inspired from common programming idioms [16] or by using a carefully thought out sequence of analysis stages and taking context sensitivity into account [18]. In this respect they are very much related to what we do. However, all these approaches have been developed and investigated in the context of imperative languages (C, C++, and Java), where the implementation of multi-threading is via locks and synchronization, so naturally the techniques on which they are based differ significantly from ours.

Very recently, Claessen et al. proposed a method to detect race conditions in Erlang programs by employing *property-based testing* using QuickCheck and a special purpose randomizing user-level scheduler for Erlang called PULSE [21]. Their 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 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 QuickCheck properties that should hold). The other race condition is detectable by dialyzer when enhancing its analysis 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 the tools can detect. But a fundamental difference between them is that in our tool the responsibility for supplying this information lies in the hands of the tool implementor while in QuickCheck's case in the programmer's.

7 Concluding Remarks

In this paper we showed 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 manifest themselves during testing, which unfortunately also makes it less likely that programmers will be paying special attention to be watching 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.

References

1. Armstrong, J.: Programming Erlang: Software for a Concurrent World. The Pragmatic Bookshelf, Raleigh (2007)
2. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Communications of the ACM 21(7), 558–565 (1978)
3. Cronqvist, M.: Troubleshooting a large Erlang system. In: Proceedings of the 3rd ACM SIGPLAN Workshop on Erlang, pp. 11–15. ACM, New York (2004)
4. Lindahl, T., Sagonas, K.: Detecting software defects in telecom applications through lightweight static analysis: A war story. In: Chin, W.-N. (ed.) APLAS 2004. LNCS, vol. 3302, pp. 91–106. Springer, Heidelberg (2004)
5. Sagonas, K.: 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)

6. Lindahl, T., Sagonas, K.: Practical type inference based on success typings. In: Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming, pp. 167–178. ACM, New York (2006)
7. Nagy, T., Nagyné Víg, A.: Erlang testing and tools survey. In: Proceedings of the 7th ACM SIGPLAN Workshop on Erlang, pp. 21–28. ACM, New York (2008)
8. Mattsson, H., Nilsson, H., Wikström, C.: Mnesia - a distributed robust DBMS for telecommunications applications. In: Gupta, G. (ed.) PADL 1999. LNCS, vol. 1551, pp. 152–163. Springer, Heidelberg (1999)
9. Carlsson, R.: An introduction to Core Erlang. In: Proceedings of the PLI 2001 Workshop on Erlang (2001)
10. Carlsson, R., Sagonas, K., Wilhelmsson, J.: Message analysis for concurrent programs using message passing. ACM Transactions on Programming Languages and Systems 28(4), 715–746 (2006)
11. Dinning, A., Schonberg, E.: Detecting access anomalies in programs with critical sections. In: Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, pp. 85–96. ACM, New York (1991)
12. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.: Eraser: A dynamic data race detector for multithreaded programs. In: Proceedings of the 16th ACM Symposium on Operating Systems Principles, pp. 27–37. ACM, New York (1997)
13. O'Callahan, R., Choi, J.D.: Hybrid dynamic data race detection. In: Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 167–178. ACM, New York (2003)
14. Beckman, N.E.: A survey of methods for preventing race conditions (2006)
15. Sterling, N.: Warlock: A static data race analysis tool. In: Proceedings of the Usenix Winter Technical Conference, pp. 97–106 (1993)
16. Engler, D., Ashcraft, K.: RacerX: Effective, static detection of race conditions and deadlocks. In: Proceedings of the 19th ACM Symposium on Operating Systems Principles, pp. 237–252. ACM, New York (2003)
17. Choi, J.D., Lee, K., Loginov, A., O'Callahan, R., Sarkar, V., Shidharan, M.: Efficient and precise datarace detection for multithreaded object oriented programs. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 258–269. ACM, New York (2002)
18. Naik, M., Aiken, A., Whaley, J.: Effective static race detection for Java. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 308–319. ACM, New York (2006)
19. Voung, J.W., Jahla, R., Lerner, S.: Relay: static race detection of million of lines of code. In: Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp. 205–214. ACM, New York (2007)
20. Mathworks: Code verification and run-time error detection through abstract interpretation. White paper (2004)
21. Claessen, K., Pałka, M., Smallbone, N., Hughes, J., Svensson, H., Arts, T., Wiger, U.: Finding race conditions in Erlang with QuickCheck and PULSE. In: Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming. ACM, New York (2009)