Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

Parallelizing Dialyzer: a Static Analyzer that Detects Bugs in Erlang Programs

Διπλωματική Εργασία

της

Υπατίας Τσαβλίρη

Επιβλέπων: Κωστής Σαγώνας
Αν. Καθηγητής Ε.Μ.Π.

Εργαστήριο Τεχνολογίας Λογισμικού
Αθήνα, Ιούλιος 2010

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστήριο Τεχνολογίας Λογισμικού

Parallelizing Dialyzer: a Static Analyzer that Detects Bugs in Erlang Programs

Διπλωματική Εργασία

της

Υπατίας Τσαβλίρη

Επιβλέπων: Κωστής Σαγώνας
Αν. Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 16η Ιουλίου, 2010.

(Υπογραφή)	*(Υπογραφή)*	*(Υπογραφή)*
.....	.....	.....
Κωστής Σαγώνας	Νικόλαος Παπασπύρου	Νεκτάριος Κοζύρης
Αν. Καθηγητής Ε.Μ.Π.	Επ. Καθηγητής Ε.Μ.Π.	Αν. Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2010

(Υπογραφή)

.....

Υπατία Τσαβλίρη

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστήριο Τεχνολογίας Λογισμικού

Ευχαριστίες

Ευχαριστώ όλους όσους συνέβαλλαν λίγο ή πολύ με την υποστήριξη τους και τις συμβουλές τους στην ολοκλήρωση της διπλωματικής αυτής.

Πρώτα από όλα τον καθηγητή μου Κωστή Σαγώνα, που από την αρχή βρήκε το θέμα που μου ταίριαζε και που με βοήθησε χωρίς δισταγμό όποτε τον χρειάστηκα. Επίσης τον ευχαριστώ για τα meetings που καθιέρωσε κάθε εβδομάδα με τα υπόλοιπα παιδιά που μας έφεραν πιο κοντά και μας έκαναν μια πραγματική ομάδα. Ευχαριστώ λοιπόν όλα τα παιδιά της ομάδας αυτής που ένας προς έναν με βοήθησαν όταν το χρειάστηκα, το Θανάση που έφυγε, τη Μαρία που ήταν πάντα εκεί, το Μιχάλη που ήξηρε πάντα την "εντολή", το Μανώλη, το Σταύρο και τον Άλκη που χρειάζονταν οργάνωση και διοίκηση για να συντονιστούν.

Περισσότερο από όλους όμως ευχαριστώ την Έλλη που τόσα χρόνια είναι δίπλα μου και χωρίς εκείνη δε θα είχα καταφέρει τίποτα και ίσως να μην το είχα προσπαθήσει εξαρχής.

Ευχαριστώ επίσης το φιλόξενο Softlab που είχε πάντα ένα γραφείο, έστω και σκονισμένο, για εμένα και την Έλλη και αρκετή όρεξη για κουβέντα όταν χρειαζόταν.

Τέλος ευχαριστώ την οικογένεια μου που ανέχτηκε τη γκρίνια μου, τους καλούς μου φίλους που με άκουγαν και με ενθάρρυναν πάντα, και όποιον άλλον ήταν δίπλα μου και με στήριζε με αγάπη.

Υπατια Τσαβλίρη

Περίληψη

Καθώς οι υπολογιστές που διαθέτουν πολυπηρυνικές αρχιτεκτονικές καθιερώνονται, η ανάγκη για αποδοτική παραλληλοποιήση των υπαρχόντων σειριακών εφαρμογών έχει γίνει επιτακτική. Η γλώσσα προγραμματισμού Erlang είναι εξ αρχής σχεδιασμένη για ταυτοχρονισμό, έτσι θα ανέμενε κανείς ότι η παραλληλοποίηση προγραμμάτων που είναι γραμμένα σε Erlang είναι ένα σχετικά εύκολο έργο. Η παρούσα διπλωματική εργασία πραγματεύεται τη διερεύνηση μεθόδων για την αποτελεσματική παραλληλοποίηση μιας μεγάλης και μη τετριμμένης σειριακής εφαρμογής γραμμένης σε Erlang, που ονομάζεται Dialyzer. Ο Dialyzer είναι ενα εργαλείο ανοιχτού κώδικα, που ανακαλύπτει αυτομάτα σφάλματα λογισμικού σε μεγάλες εφαρμογές γραμμένες σε Erlang. Αφού εντοπίσουμε τα τμήματα των αναλύσεων που πραγματοποιεί ο Dialyzer τα οποία μπορούν να εκτελεστούν παράλληλα, περιγράφουμε κάποιες τεχνικές χονδρόκοκκης παραλληλοποίησης που επωφελούνται από την βασισμένη σε δένδρα δομή των αναλύσεων αυτών. Περιγράφουμε επίσης τους τρόπους αντιμετώπισης των προβλημάτων μνήμης που προκύπτουν από το γεγονός ότι πολλές παράλληλες διεργασίες χρειάζονται ταυτόχρονη πρόσβαση σε μεγάλες δομές δεδομένων. Αφού εφαρμόσουμε στην πράξη όλες αυτές τις τεχνικές, παρουσιάζουμε τα πειραματικά αποτελέσματα που δείχνουν ότι η παράλληλη έκδοση του Dialyzer που αναπτύξαμε καταφέρνει να επιταχύνει σημαντικά το χρόνο εκτέλεσής των αναλύσεων του, ακόμα και σε διπύρηνα ή τετραπύρηνα υπολογιστικά συστήματα που είναι πολύ κοινά στις μέρες μας. Πέρα από τη συγκεκριμένη εφαρμογή, η παρούσα εργασία προσφέρει πολύτιμη εμπειρία, καθώς και σημαντικά συμπεράσματα, τόσο για τα πλεονεκτήματα όσο και για τα εμπόδια που ένας προγραμματιστής μπορεί να συναντήσει όταν προσπαθεί να παραλληλοποιήσει μια σειριακή εφαρμογή γραμμένη σε Erlang.

Keywords

παραλληλοποιήση, ταυτοχρονισμός, επιτάχυνση, απόδοση, Dialyzer, ανάλυση, μοιραζόμενα δεδομένα, Erlang

Abstract

As parallel platforms, like multi-core machines, become mainstream, the need to efficiently parallelize existing serial applications has become pressing. Erlang is a programming language designed for concurrency from the ground up so one might expect that parallelization of Erlang programs is a relatively easy task. This thesis is concerned with investigating methods to effectively parallelize a big and non-trivial single-threaded application written in Erlang, called Dialyzer. Dialyzer is an open source software defect detection tool, that automatically discovers software bugs in big applications written in Erlang. After identifying which parts of the analyses Dialyzer performs can be run in parallel, we describe coarse-grained parallelization schemes which take advantage of the tree-based nature of these analyses. We also describe ways of dealing with memory issues that emanate from many concurrent processes needing efficient concurrent access to large data structures. We implemented all these schemes and we report experimental results showing that the parallel implementation of Dialyzer manages to significantly speed up its execution, even on 2-core or 4-core machines which are very common nowadays. Beyond this particular application, this thesis offers useful experience as well as important conclusions concerning both the advantages and the obstacles a programmer is likely to encounter when parallelizing a serial Erlang application.

Keywords

parallelization, concurrency, speed up, performance, Dialyzer, analysis, shared data, Erlang

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

In recent years, major processor manufacturers have massively focused on developing multicore architectures and integrating them in every single machine to improve its performance. This trend is due mainly to the fact it has become really hard to further increase the speed of a single processor, leaving no room for the traditional approach that assured that single-threaded code executed faster on newer processors with no modification[12]. As a consequence parallelism is becoming ubiquitous, and parallel programming is becoming central to the programming enterprise. However, performance improvement in multicore architectures requires hard work and fundamental changes to the way we design and implement algorithms and programs. Software has to be written in a multi-threaded or multi-process manner to take full advantage of the new hardware.

Still, currently there is an enormous number of existing applications that remain single-threaded, since they were designed years ago when multicores were considered rare and exotic. Some of them are inherently non-parallel, but others are just condemned by the fact that they were not developed with concurrency in mind. To stop those applications from wasting performance improvement by ignoring multiple cores availability there is only one solution: turn them from sequential to parallel, i.e. parallelize them. This transition can be made in an easier way if the language in which the application is written offers a satisfying support for concurrent programming. An example of such a programming language is Erlang.

Erlang is a language designed for concurrency from the ground up; it was a central concern when the language was developed. Its built-in support for concurrency, which uses the process concept to get a clean separation between tasks, allows programmers to create fault tolerant architectures and fully utilize the multi-core hardware that is available to us today[10]. However, since concurrent programming is considered hard and often avoided by programmers, not all big applications written in Erlang take advantage of the language infrastructure, being serial or mostly serial. Hence, we think of such an Erlang application as the perfect case to investigate ways of efficiently parallelizing a single-threaded application.

We choose for our study a big and non trivial serial Erlang application Dialyzer, an open source software defect detection tool, that automatically discovers software bugs in big applications written in Erlang. Study of serial Dialyzer's function, profiling, memory and performance evaluation are some of the methods we use in this thesis to effectively parallelize this application. We also describe coarse-grained parallelization schemes, ways to deal with memory issues and we actually introduce a parallel implementation for Dialyzer. Our aim is to share valuable experience and perhaps some guidelines with anyone who wishes to paralellize a big, real-world, single-threaded Erlang application and

draw some conclusions about both the advantages and the obstacles of such an acting.

To introduce the reader to our subject, the next chapter briefly describes some characteristics of concurrent Erlang as well as some details about the Dialyzer application. Chapters 3 and 4 present information about serial Dialyzer: In Chapter 3 we introduce the main data structures used whereas in Chapter 4 we explain in detail the way Dialyzer performs static analysis. The main chapter of this thesis, Chapter 5, comes next and presents the whole procedure of Dialyzer's parallelization, the problems we encountered and the solutions we figured out, as well as some implementation details. In Chapter 6 we present the performance results followed by some general conclusions and remarks stated in Chapter 7.

Chapter 2

Preliminaries

2.1 The Erlang Language

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 [2]. 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 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.

In Erlang scheduling of processes is primarily a 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, 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 also employs a redistribution scheme based on work stealing when some scheduler's run queue becomes empty[4].

To sum up, Erlang has many traits that make it nearly ideal for multi-core, at least for a certain class of problems[13].

2.1.2 Shared Memory in Erlang

The central problem in all concurrent systems, that all implementers have had to solve, is the problem of sharing information. If one separates a problem into different tasks, how should those tasks communicate with one another[10]? Erlang is often advertized as supporting a shared nothing concurrency model[5, 1], where process communication is exclusicely done by message passing. Actually, this is not quite true, but there are only two ways to share memory, thus sharing can be easily avoided. These two ways of sharing memory have to do with shared ETS or DETS tables.

ETS and DETS are two system modules that are used for efficient storage of large numbers of Erlang terms. ETS is short for Erlang term storage, and DETS is short for disk ETS. Since ETS tables are commonly used in Erlang applications, it is worth taking a closer look to them.

ETS tables are data structures for associating keys with values. The most common operations we will perform on tables are insertions and lookups. An ETS table is just a collection of Erlang tuples.

ETS Table Efficiency Considerations

ETS is highly efficient—using ETS, we can store colossal amounts of data (if we have enough memory) and perform lookups in constant (or in some cases logarithmic) time. ETS tables look as if they were implemented in Erlang, but in fact they are implemented in the underlying runtime system and have different performance characteristics than ordinary Erlang objects. In particular, ETS tables are not garbage collected; this means there are no garbage collection penalties involved in using extremely large ETS tables, though slight penalties are incurred when we create or access ETS objects. However, to protect against the simultaneous modification of shared memory, a locking mechanism is used. Call this a mutex, a synchronized method, or whatever else, it's still a lock.

When it comes to memory issues, ETS tables are stored in a separate storage area that is not associated with normal process memory. An ETS table is said to be owned by the process that created it— when that process dies or when ets:delete is called, then the table is deleted.

When a tuple is inserted into an ETS table, all the data structures representing the tuple are copied from the process stack and heap into the ETS table. When a lookup operation is performed on a table, the resultant tuples are copied from the ETS table to the stack and heap of the process. This is true for all data structures except large binaries that are stored in their own off-heap storage area.

Types of ETS Tables

- *private*: Create a private table. Only the owner process can read and write this table.

- *public*: Create a public table. Any process that knows the table iden- tifier can read and write this table.

- *protected*: Create a protected table. Any process that knows the table identifier can read this table, but only the owner process can

ETS Tables As Blackboards

Protected tables provide a type of "blackboard system." We can think of an protected ETS table as a kind of named blackboard. Anybody who knows the name of the blackboard can read the blackboard, but only the owner can write on the blackboard. In contrast, an ETS table that has been opened in public mode can be written and read by any process that knows the table name. In this case, the user must ensure that reads and writes to the table are performed in a consistent manner[1].

2.2 Dialyzer: A Brief Overview

Since 2007 the Erlang/OTP distribution includes a static analysis tool, called DIALYZER, for finding discrepancies (i.e., type errors, software defects such as exception-raising code, hidden failures, unsatisfiable conditions, redundancies such as unreachable code, etc.) in single Erlang modules or entire applications. DIALYZER 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. 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[6].

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 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, 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 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[8]. 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 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 and become accurate and precise.

Notable characteristics of DIALYZER are[11]:

- 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 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[3].

Chapter 3

Basic Data Structures

In order to be able to present the steps and methods we implemented towards DIALYZER's parallelization, it is necessary to introduce the reader to the way the serial analysis is currently performed. But first it is necessary to describe the three fundamental data structures of DIALYZER, their implementation as well as the role they play in the analysis.

3.1 Call Graph

3.1.1 Function Call Graph

For efficiency reasons, the analysis first constructs a global function Call Graph, which describes the dependencies between the functions of the modules to be analyzed. Specifically, the Call Graph is a directed graph with functions as nodes and an edge (f, g) whenever f calls g. We can easily notice that this is not an acyclic graph since mutually dependent functions, that means functions that call each other recursively, can form cycles. To represent this data structure best, the digraph module, which implements a version of labeled directed graphs, is used.[8]

For example, consider the module in Listing 3.1. This is a list processing function included in the stdlib library of the standard Erlang distribution that returns the sorted list formed by merging two lists. Both first and second list must be sorted according to some ordering function prior to evaluating this function. The Call Graph that DIALYZER constructs in order to analyze this piece of code is shown in Figure 3.1:

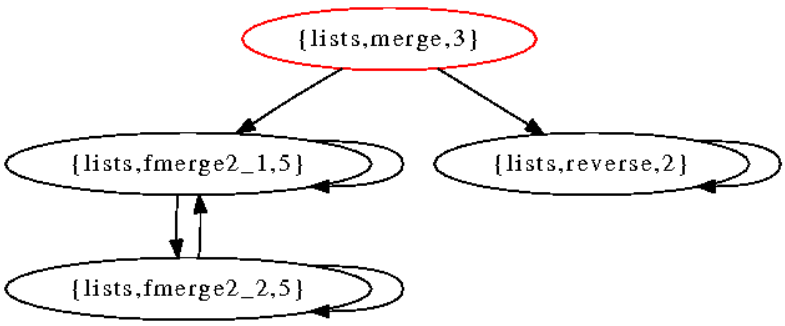

Figure 3.1: lists:merge/3 function call graph

As we will describe in the next chapter, the analysis uses the information represented in this function Call Graph to perform the type inference in a bottom-up fashion, propagating

```
1  -module(lists).
2
3  -export([merge/3]).
4
5  merge(Fun, T1, [H2 | T2]) when is_function(Fun, 2) ->
6      reverse(fmerge2_1(T1, H2, Fun, T2, []), []);
7  merge(Fun, T1, []) when is_function(Fun, 2) ->
8      T1.
9
10 reverse([H|T], Y) ->
11     reverse(T, [H|Y]);
12 reverse([], X) -> X.
13
14 %% Elements from the first list are prioritized.
15 fmerge2_1([H1 | T1], H2, Fun, T2, M) ->
16     case Fun(H1, H2) of
17         true ->
18             fmerge2_1(T1, H2, Fun, T2, [H1 | M]);
19         false ->
20             fmerge2_2(H1, T1, Fun, T2, [H2 | M])
21     end;
22 fmerge2_1([], H2, _Fun, T2, M) ->
23     lists:reverse(T2, [H2 | M]).
24
25 fmerge2_2(H1, T1, Fun, [H2 | T2], M) ->
26     case Fun(H1, H2) of
27         true ->
28             fmerge2_1(T1, H2, Fun, T2, [H1 | M]);
29         false ->
30             fmerge2_2(H1, T1, Fun, T2, [H2 | M])
31     end;
32 fmerge2_2(H1, T1, _Fun, [], M) ->
33     lists:reverse(T1, [H1 | M]).
```

Listing 3.1: Function merge from lists.erl of stdlib library

type information from callees to callers. To be able to do so, some kind of topological ordering of the graph is needed, a fact that requires our Call Graph to be acyclic. For this purpose, the Call Graph we described is condensed to its strongly connected components (SCCs), i.e. its maximal strongly connected subgraphs, and we end up with a directed acyclic graph (DAG) in which each and every cycle is included in some SCC[8]. The condensed Call Graph of the function `merge/3` is shown in Figure 3.2.

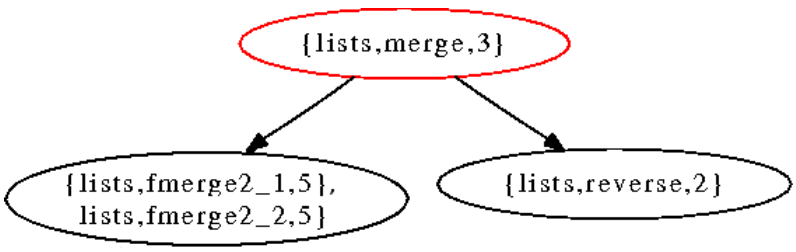

Figure 3.2: lists:merge/3 call graph of function SCCs

However, the representation of the graph itself is not enough for the type analysis DIALYZER performs, as more pieces of information regarding the calls, the self recursive

or escaping functions as well as the mappings between functions' names and labels are indispensable. All this information is organized and stored as a record named callgraph, which is part of the program state, and thus is passed as an argument to every function that contributes to the type inference.

3.1.2 Module Call Graph

Apart from the function Call Graph we described above, the analysis also uses another directed graph, that describes the dependencies between the modules to be analyzed instead of those of the functions. This Module Call Graph is constructed in the same way as the function callgraph except that each node now represents a module. DIALYZER uses this graph to get the necessary information about function calls across module boundaries. It is condensed to its SCCs so that it can be sorted and then used to propagate analysis information in the control flow, as we are going to describe in detail in the next chapter.

For instance, in Figure 3.3 we can see the Module Call Graph of OTP's `typer` application condensed to its SCCs. Examining that graph we can figure out that `typer` consists of six modules. The module typer_map does not contain a call to a function that belongs to any of the other, whereas the rest of them form a cycle, i.e. every module includes at least one call to a function that belongs to some of the others, in a way that a path from each module in the uncondensed graph to every other module exists. Moreover, in some of those modules forming the large SCC there is a call to a typer_map function.

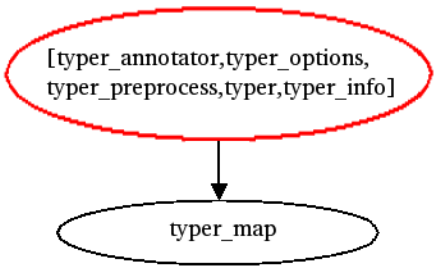

Figure 3.3: lists:merge/3 call graph of module SCCs

3.2 PLT

DIALYZER stores the information of the analyzed functions in a Persistent Lookup Table (PLT). To understand the role of this structure we should describe how it is initially constructed as well as how it is actually used during the analysis.

Building PLT

Upon first start-up DIALYZER automatically triggers the analysis of the ERLANG/OTP standard library, stdlib, to construct a persistent lookup table which can be used as a basis for all subsequent analyses.[6] However, the programmer is free to add to this PLT the DIALYZER results about any other frequently used library modules or applications he frequently uses, in order to speed up his future analyses. Using this table as a starting point, the analysis will not have to find once again the types of those stored and pre-analyzed functions, but will just have to fetch them from that persistent lookup table, which is stored in the users home directory(unless otherwise configured)[9].

Using PLT

When the user runs the analysis for the modules he selected, the information stored in the PLT file is loaded and kept as a record throughout program execution. The most important information this record contains is a dictionary, called info, with function name or label as a Key and both the inferred arguments and return types of that function as a Value. Other important information included in the PLT record is a dictionary, called contracts, that maps function names to their contracts(type specifications), and a set of the exported types of all analyzed functions.

All this information is not only read during the analysis but also updated every time the analysis figures out the type of a function. As a consequence the copy of the stored PLT our analysis uses is step by step populated with information about every single function included in the modules we chose to analyze. This way the persistent lookup table works as a type database and is fundamental to type propagation among dependent functions, like those we encountered when studying callgraph. In the following chapter, when describing the analysis steps, we are going to explain in a more detailed way how PLT information is used.

3.3 Code Server

The last major data structure that we are going to present is Code Server, where the compiled source code of the modules to be analyzed, as well as some important information about it, are stored. Actually, Code Server is not exactly a single data structure. It consists of a record with exports, exported types, module records and contracts information and a spawned server process which manages the code. It is worth taking a closer look to the way the server process works.

New Code Server

First of all, the code is loaded at the beginning of the analysis and it is compiled to Core Erlang, an intermediate representation of Erlang. Then a new process is spawned and starts to work as the Code Server. This server stores the Core Erlang code in a dictionary with module name as a Key and module code as a value. First, each module code is encoded as a compressed binary in order to significantly reduce memory usage and then is stored by the Code Server.

Code Lookups

When at some point of the analysis the code of some module is needed, Code Server receives a lookup request message and immediately fetches and decodes the code from binary back to Erlang term format, in order to send it as a response. However, when the analysis asks for the code of just one function, which is often the case, fetching and expanding every time the code of the whole module has turned out to be inefficient. For this purpose, Code Server currently implements a method of data caching. Specifically, in such a case the code server does indeed fetch and decode the whole module code, but does not throw it away after picking out the code for the single function. Instead, it keeps it cached, passing it as an argument to its next loop iteration(lines 4-20 of Listing 3.2). This way if there are consecutive lookup requests for functions that belong to the same module, which is also a common phenomenon, the server has already available the required code and no longer has to stall by re-fetching and re-expanding it.

This method offers a far from being negligible speed up to the code server functionality, as well as reduction in memory consumption. For instance, when running dialyzer to analyze three important OTP applications, stdlib, compiler and kernel the size of the code we need to store if we perform no compression is about 39961378 words. However, if we keep this code cashed with module-level compression its size is dramatically reduced to 3400 words. It is worth mentioning that if we store the same code with function-level compression the size of code would be somewhere between, reaching 225126 words for the same applications.

```
table__loop(Cached, Map) ->
  receive
    stop -> ok;
    {Pid, lookup, {M, F, A} = MFA} ->
      {NewCached, Ans} =
         case Cached of
           {M, Tree} ->
             [Val] = [VarFun || {Var, _Fun} = VarFun <- cerl:module_defs(Tree),
                                cerl:fname_id(Var) =:= F,
                                cerl:fname_arity(Var) =:= A],
             {Cached, Val};
           _ ->
             Tree = fetch_and_expand(M, Map),
             [Val] = [VarFun || {Var, _Fun} = VarFun <- cerl:module_defs(Tree),
                                cerl:fname_id(Var) =:= F,
                                cerl:fname_arity(Var) =:= A],
             {{M, Tree}, Val}
         end,
         Pid ! {self(), MFA, Ans},
         table__loop(NewCached, Map);
    {Pid, lookup, Mod} when is_atom(Mod) ->
      Ans = case Cached of
              {Mod, Tree} -> Tree;
                        _ -> fetch_and_expand(Mod, Map)
            end,
      Pid ! {self(), Mod, Ans},
      table__loop({Mod, Ans}, Map);
    {insert, List} ->
      NewMap = lists:foldl(fun({Key, Val}, AccMap) ->
                           dict:store(Key, Val, AccMap)
                           end, Map, List),
      table__loop(Cached, NewMap)
  end.
```

Listing 3.2: Dialyzer Code Server loop

Chapter 4

Sequential Analysis

Now that the reader is familiar with the main data structures Dialyzer uses, it is time to present the way the sequential static analysis is performed. Therefore, in this chapter we are going to describe step by step the algorithm that Dialyzer employs in order to identify type clashes in Erlang code.

4.1 Preparation step

In the beginning of the analysis, the basic infrastructure is built in order to facilitate and support the analysis. It consists of the data structures we described in the previous chapter, which are initialized and then stored in a record named State which is thereafter used in all steps of the program. In fact, during this preparation period :

- PLT is loaded from the input file and its information is stored in the corresponding record field.
- Code of the modules to be analyzed is loaded, compiled and stored in Code Server's dictionary.
- Call Graph for all functions included in the modules to be analyzed is constructed and condensed. Then, this directed acyclic graph is sorted topologically. In particular, a postorder depth-first traversal is performed, that results in a linear order of its nodes in which each node comes before all nodes to which it has outbound edges. In other words, we end up with an ordered list where function SCCs always come before their parents, i.e. the SCCs that call them and thus depend on them. The ordering of the analysis, at least of the first part of it, is based on that sorted list since it can ensure that functions are analyzed in a bottom-up fashion. That way, it is also ensured that all type information about the functions that a SCC depends on has already been gathered[7], which is very useful for type propagation during the type inference Dialyzer employs.

4.2 Success Typings Inference

This is perhaps the most important phase of Dialyzer analysis, as it results in conclusions about the success typings of the arguments and the return value of every included function. We have explained the notion of success typings in Chapter 2, but now we are going to take a closer look to their inference process.

4.2.1 Overview

The success typings inference is done by analyzing one SCC of the postorder list of SCCs we have constructed after the other. Once a function has been analyzed, the resulting type signature is put into PLT. ingWhenever a call to this function is encountered, we can use the current value in the persistent lookup table to get its type information. This way, types are propagated from callees to callers.

4.2.2 Analysis Steps

We will now describe in more detail the steps of finding the success typings of the functions to be analyzed:

- Take the first SCC of the topologically ordered Call Graph and request its code, contracts and records from the Code Server.

- Apply the success typings inference algorithm to the specified SCC. This algorithm has two phases. In the first, the code is traversed and constraints are generated using derivation rules. In the second, we try to find a solution to the constraints and this solution constitutes the success typing[8]. This process uses information from Call Graph, PLT and function code who are passed as arguments throughout the analysis process.

- If the inferred success typing compare equal to the one stored in the PLT about that specific function (if it exists), then this SCC has reached a fixpoint. In other words, Dialyzer cannot find more specific type information than that stored in the PLT, so the SCC does not need to be analyzed any more. Otherwise, it means that the success typing for that SCC has been further specialized, because of some refinement performed on a previous step, and they get stored in the PLT in order to replace the previous ones - if any.

- The previous steps are repeated until there are no more SCCs in the sorted list to be analyzed. Then, all functions that have not reached a fixpoint are gathered in a list. If this list is empty then every single function has been assigned the most specific success typings possible and the analysis is over. Otherwise, the analysis has to proceed to the refinement step, but only for the functions in the list, until no function has been left without a fixpoint. Obviously, the number of SCCs to be analyzed is constantly reduced and by avoiding redundant re-analyses the analysis gets more efficient.

4.3 Refinement of Success Typings

This is the other fundamental phase of Dialyzer analysis. A refined success typing is a success typing under some additional constraints. In particular, a refined success typing of a function is a success typing where the domain is restricted by taking information from all call sites into account. In other words, the refinement of the success typings Dialyzer performs is a type specialization of all non-escaping functions based on information which manifests how the function is actually used in a program[8]. In this section we are going to present some details of this procedure.

4.3.1 Overview

ERLANG functions are organized in modules with a specified interface, declared with an -export() statement. Non-exported, called internal or module-local, functions are protected against arbitrary calls from other modules. This means that for module-local functions we can employ a closed world assumption about their intended uses and exploit information about their calling contexts to derive better typings for them[7].

For instance, in Figure 4.1 we see a case where types are refined due to a call. Particularly, function `f/1` is only called in function t/1 with a floating point number as an argument. As `f/1` is not an exported function this is the only case this function is used and therefore both its argument and return value type can be resticted from number to float. For `n/1`, a similar process, refinement does not occur since this function is called both with an integer and a floating point number. As for `t/1`, its argument cannot be refined since it is an exported function and therefore its call sites are not known.

```
-module(arith).
-export([t/1]).

t(N) ->
  X = f(3.14) + N,
  n(42) + n(X).

n(N) -> N + 1.

f(N) -> N + 2.
```

Success typings	
t	:: $(number()) \rightarrow number()$
n	:: $(number()) \rightarrow number()$
f	:: $(number()) \rightarrow number()$

Refined success typings	
t	:: $(number()) \rightarrow number()$
n	:: $(number()) \rightarrow number()$
f	:: $(float()) \rightarrow float()$

Figure 4.1: Success Typing Refinement on module arith

4.3.2 Refinement Steps

As described in the previous section the refinement of success typings works in a topdown fashion over the function call graph, propagating information from callers to callees. This data flow analysis steps are:

- Construction and condensation of the module Call Graph for the functions that have not reached a fixpoint. It is then sorted topologically and results in a list were module SCCs are placed in a postorder way. However, those SCCs are also ordered internally in a postorder way.
- Take the first module SCC of the topologically ordered Module Call Graph.

  If it only contains one module then:

  - Request the module code and records from the Code Server.
  - Apply the function types refinement algorithm to the specified module. Since we are now interested in the information flow from caller to callee, this algorithm performs an analysis that traverses the module code and propagates information forward in the control flow. Function Call Graph, PLT and module code are necessary for this analysis, so they are passed as arguments to the function in charge.

  - If the refined success typings that resulted from the data flow analysis compare equal to the ones stored in the PLT about that specific module, then the analysis has reached a fixpoint for this module. In other words, Dialyzer found no calls that can restrict the previously inferred type information stored in the PLT, so this module does not need to be analyzed any more. Otherwise, it means that the refinement process has further specialized some function types of that module which are therefore stored in the PLT in order to replace the previous ones.

  Else if the SCC contains more than one module

  - Analyze the modules one after another, the way we described previously, in the order they are stored in the SCC list.
  - Collect the functions of the modules that have not reached a fixpoint in a list. If this list is empty then the refinement for this SCC is completed. Otherwise, rerun the refinement analysis for this SCC (repeat from previous step) until the fixpoint is empty. This means that no further information can be inferred by studying the calls between these modules.

- The previous steps are repeated until there are no more module SCCs in the sorted list to be analyzed. Then, all functions of the modules or the SCCs that have not reached a fixpoint are gathered in a list. If this list is empty then every single function has been assigned the most specific success typings possible and the analysis is over. Otherwise, the analysis has to repeat the success typings inference step, but only for the functions in the list, until no function has been left without a fixpoint. Obviously, the number of functions to be analyzed is once again reduced, so a new function Call Graph is constructed out of the remaining functions on which the rest of the analysis will be based.

The outline of the analysis we described so far is smoothly expressed in the Dialyzer's analysis main function shown in Listing 4.1.

4.4 Warnings collection

When all functions have reached a fixpoint, i.e. there cannot be further specialization of the types Dialyzer has inferred for them, the static analysis is over. Now, all the required information Dialyzer needs to identify the code discrepancies is available and stored in the persistent lookup table. Therefore, it is time to track any software defects or hidden failures and present them in the form of warnings to the user.

To do so, Dialyzer has to traverse the code of each module one last time and watch out for type clashes, using PLT as a reference. Dialyzer traverses the code of a module the same way as in data flow analysis, using the same functions, but paying attention to identify type errors, since functions types are already as refined as possible. However, during this step there is no need for special ordering of the module set, since there will not occur any type propagation. In other words, the dependencies between the modules are of no interest to the warning collection procedure, which is therefore applied to each module in a simple alphabetical order.

```
get_refined_success_typings(State) ->
  case find_succ_typings(State) of
    {fixpoint, State1} ->  State1;
    {not_fixpoint, NotFixpoint1, State1} ->
      Callgraph = State1#st.callgraph,
      NotFixpoint2 = [lookup_name(F, Callgraph) || F <- NotFixpoint1],
      ModulePostorder =
        dialyzer_callgraph:module_postorder_from_funs(NotFixpoint2, Callgraph),
      case refine_succ_typings(ModulePostorder, State1) of
    {fixpoint, State2} ->
      State2;
    {not_fixpoint, NotFixpoint3, State2} ->
      Callgraph1 = State2#st.callgraph,
      %% Need to reset the callgraph.
      NotFixpoint4 = [lookup_name(F, Callgraph1) || F <- NotFixpoint3],
      Callgraph2 = dialyzer_callgraph:reset_from_funs(NotFixpoint4,
                                Callgraph1),
      get_refined_success_typings(State2#st{callgraph = Callgraph2})
      end
  end.
```

Listing 4.1: Serial analysis main function

4.5 Analysis Summary

The way DIALYZER performs the its analysis can be summarized in the following basic steps:

1. Construct the call graph for the functions and sort it topologically based on the dependencies between its strongly connected components (SCCs).
2. Analyze the SCCs in a bottom-up fashion using the constraint based type inference to find their most general success typings under the current constraints.
3. Analyze the SCCs in a top-down order using the data flow analysis to propagate information from the call sites to module local functions, and then add new constraints for the type inference.
4. If a fix-point has been reached, annotate the program with the derived type signatures, otherwise repeat from step 2.
5. Identify and collect the warnings from modules.

Chapter 5

Parallelization

Dialyzer has some features that have rendered it a quite promising candidate for parallelization. Now that the reader is aware of the way the analysis is done, it is time to present our ideas and efforts to make this application run efficiently on shared memory multi-core architectures. Since Dialyzer's parallelization required a lot of work, there is a lot of experience to share.

First, we are going to introduce our initial idea, pointing out details about which parts of the program can be parallelized and how. Then we will describe the design of the parallel version of Dialyzer we implemented, with information about how coordination and communication between processes is done. Finally, we will state and explain the cause of the problems we encountered, as well as the solutions we figured out and implemented.

5.1 Before Parallelization: Design and Profiling

5.1.1 Basic Idea

The main reason we decided to develop some techniques to convert Dialyzer from a single-threaded application to a multi-threaded one is that we have noticed that some specific parts of its execution can be executed independently. As we may remind, Dialyzer analysis consists of three main parts: success typings inference, success typings refinement and identification and collection of warnings.

From an algorithmic point of view, all of those parts have a common characteristic; they apply a specific analysis function, different for each part, to every SCC of a Call Graph. In any case, this function does not mutate any global variables, only manipulates state by returning new data. So, the though that immediately comes to mind is: why not apply this function to every SCC of the graph concurrently? Well, this may sound quite reasonable but it is only applicable for the collecting warnings phase where the dependencies between those SCCs do not matter.

However, in the first two type inference related phases the order in which those tree SCCs are analyzed play a fundamental role. As we saw in the previous chapter, in both those cases the analysis is based on a topological ordering of that tree because there are dependencies between the SCCs. The actual constraint that prevents us from performing a parallelization as described above is that every SCC should be analyzed before its parent, to allow types to be upwards propagated. Still, we are now going to describe a satisfying and relatively simple solution to this problem.

5.1.2 Parallelization method

We recall that the tree we are talking about is actually a Call Graph where the SCCs are either function or module SCCs, depending on whether we are performing success typings inference or refinement respectively. We will continue referring to them simply as SCCs since the parallelization technique we are going to describe is general and applicable to both the analyses.

Method Outline

Taking a closer look at this Call Graph, we notice that the only SCCs that do not depend on any other are those represented as leaves. These SCCs are ready to be analyzed, and since they have no dependencies their analysis may be done in parallel. In the meantime, whenever the analysis for one leaf is completed, any other SCC that used to depend exclusively on it is ready to be analyzed, taking as input the output of the leaf analysis. So, a queue of ready for analysis SCCs is gradually formed and whenever one completes its analysis, the analysis of its parents gets queued, provided that it has been their only unanalyzed child. This way, this queue serves as a pool of independent tasks that can be executed in parallel from the system CPUs.

For instance, consider the following example Function Call Graph in Figure 5.1. To perform the parallel approach we described above in the success typings inference analysis, we should first look for the Call Graph leaves. We notice that those are the functions `bar/1` and `boo/2`, which are therefore ready to be analyzed in parallel. When `boo/2` analysis is completed, boo/1 that depends only on that function, is added to the ready queue, whereas when `bar/1` analysis finishes, no function gets ready since `foo/2` has to wait for the results of `boo/1` too. Finally, when those results are ready, the analysis of the root of the Call Graph, function `foo/2`, begins. In the end, all Call Graph functions have been analyzed, and the inferred types have been propagated in the right bottom up way. There is no need to present an example of a refinement step of a Module Call Graph too, as the approach would be completely similar.

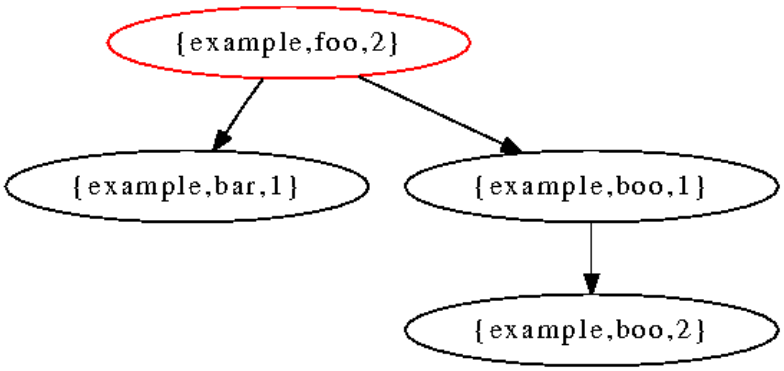

Figure 5.1: Example function call graph

Serial Analysis Profiling

Before reorganizing Dialyzer to put the idea we described above into action, it was wise to gather some profiling information about its sequential execution behaviour. Identifying which parts of a sequential program are most time-consuming can help us realize

Applications	Success Typings Inference	Refinement of Success Typings	Warnings Collection	Rest of Analysis	Total time
stdlib	228.09	38.56	147.19	14.41	428.25
kernel	23.65	8.58	23.89	4.92	61.04
erts	14.33	4.61	9.21	1.39	29.53
compiler	241.25	23.29	41.43	6.27	312.24

Table 5.1: Dialyzer profiling for four OTP applications

the performance impact we should expect by applying our parallelization method. In this direction, we ran serial Dialyzer for some applications included in the Erlang/OTP to be able to argue about what portion of the total execution time is due to each step of the analysis. The results of this profiling on a multiprocessor machine with four Intel Xeon E7340 CPUs (2.40 GHz) and 16 GB of RAM, are shown in Table 5.1 and the corresponding charts are presented in Figure 5.2. We remind that during the analysis, the success typings inference as well as the refinement steps can be executed more than one time until a fixpoint is reached. Wherever this is the case, the time reported in the table is actually the sum of the time of every execution of such a step.

The information we get from those charts is very useful. It is obvious that in all cases the phase of success typings inference is where most analysis time is spent. In particular, it varies from 39% to even 77% of the total execution time. Next time-consuming phase is the refinement of success typings that ranges betwenn 13% and 39%, depending on the application. The warnings collection step occupies a smaller time slice of the total analysis time, with a minimum of 8% and a maximum of 16%. The rest of the analysis, that includes reading and compiling source code as well as constructing the basic data structures, consumes in all cases less time than every other phase since it varies from 2% to 8% of the total execution time.

Therefore it is obvious, that in order to significantly speed up this application we should first focus on parallelizing success typings inference, then the refinement step and finally the warnings collection. However, we may notice that the fact that warnings identification and collection is the least time-consuming among the three main analysis phases, is a little disappointing. To be specific, as we mentioned in the previous section, the order of execution of the analyses performed during this phase, unlike the others, is not restricted by any kind of dependencies, rendering them highly parallelizalbe. Still, since the time it consumes is small, its parallelization, no matter how efficient, is expected to have a small impact on the overall execution time.

Synchronization Points

Another thing that is essential to do before converting a serial algorithm to a parallel one is identifying the synchronization points, meaning the program points where all processes have to meet or where some of them have to wait for a resource to get ready. Studying serial execution can give us valuable information about those points and help us this way design how the coordination and communication of parallel processes should be performed to preserve program correctness and enhance its efficiency.

In our case, it is not very difficult to trace those points by recalling some parts of the analysis we described in Chapter 4. First of all, reasonably, every phase of the analysis has to be completed before we can proceed to the next one. In other words, only when the

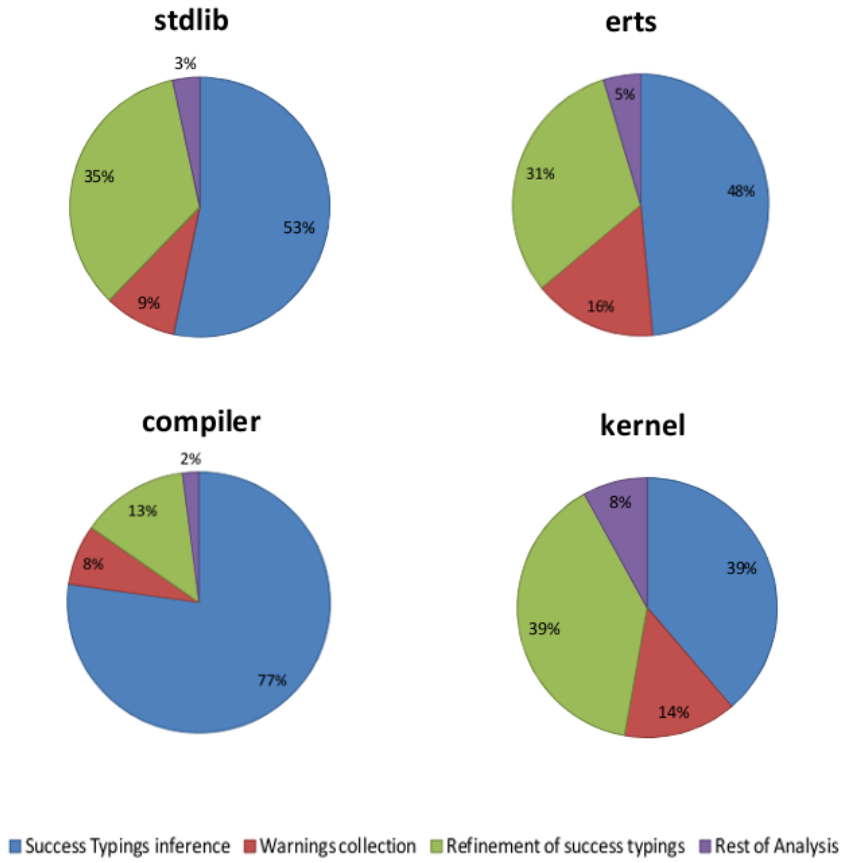

Figure 5.2: Time profiling on four OTP applictions

success typings inference step has analyzed every SCC in its list, Dialyzer can calculate the new Call Graph from the fixpoint and proceed to the refinement step. The same also applies the other way around; only when all modules have been refined, the success typings inference step can start again, if necessary. Finally, Dialyzer must have finished the type analysis in order to be able identify the warnings of the code using the final function types stored in the PLT. Therefore the warnings collection step has to wait for the type inference to end, in order to prepare and start its execution.

The synchronization points we just described are actually the parts of the program that cannot be parallelized (they remain sequential). There is no way we can avoid or ameliorate them in our parallel implementation since they are required from the nature of the analysis performed. However, Dialyzer also has some other synchronization points due to the data dependencies that appear in the analysis. Even though these points definitely restrain the parallel execution of some processes, their performance impact can be tuned to some extend by performing proper and efficient coordination and communication of the involved processes.

Specifically the parallelization method we described at the beginning of this section, applies a sophisticated process coordination pattern that restrains as much as it gets the performance damage because of the data dependencies. In particular, the coordinator we designed detects and queues an SCC for analysis as soon as its last child has been analyzed, resulting in increased parallelism. A simpler implementation would wait for all Call Graph leaves to finish their parallel analysis, then delete them from the graph and trigger the concurrent analysis of the leaves of the new Call Graph. The advantage of

this method is that it is lighter since it does not require checking all the time for ready to be analyzed SCCs, but on the other hand it unwisely wastes parallelism. In fact, the problem of this method lies on the fact that when waiting for the last leaves to complete their analysis, there may be cores that remain idle even though ready SCCs exist, since the coordinator does not know about them. The performance of this method gets worse as the number of cores the parallel program is running on increases.

5.2 Implementation Description

After having studied every detail of Dialyzer's sequential execution and profiled its behaviour, we came up with a parallel implementation that puts into action the methods described in the previous section. We will first describe the way the coordination is done for the inference and refinement of success typings, since they are very similar, but we will present both their implementations. Additionally, we will describe the trivial coordination that was needed for the parallelization of the warnings phase.

5.2.1 Parallel Inference and Refinement of Success Typings

The outline of our parallel implementation of both these analysis steps is the same, since it is based on the technique we described in the previous section which is applicable to both cases. Our implementation includes two main actors which undertake the tasks of spawning, coordinating and collecting the results of the processes that actually perform the analysis of the Call Graph SCCs. The communication between actors and processes is performed by message passing, which is also the means used to coordinate the whole parallel analysis as we will explain below.

Actors and Roles

Trying to apply our parallelization method we immediately realize that we need an actor that will trigger the concurrent analysis of the ready to be analyzed SCCs. In other words, this actor would be, first of all, responsible for spawning processes to analyze the Call Graph leaves. Later, this actor would have to wait to receive information about which SCCs are each time ready to be analyzed and then spawn processes to analyze them as well. From now on, let us refer to this actor as Spawn Server.

Furthermore, there is a need for an actor that is able to decide when an SCC is ready to be analyzed, i.e. when the analysis of its children is over, and to propagate type information properly. In this direction, this actor would need to be notified from the analysis function whenever the analysis of a SCC is completed, and access the type information that resulted from it. This way this actor could perform a simple check every time he receives the results of a SCC analysis to determine whether any other SCC used to depend exclusively on it. If such one is found, then PLT has to be updated with the new function types and then a message has to be sent to the Spawn Server, informing it about the new ready SCCs. Otherwise the actor just has to update the type information and continue waiting to receive the results of another SCC. In fact, this actor actually coordinates the analysis, since it defines the order in which SCCs are analyzed, so let us refer to it as Coordinator.

Communication

Communication between actors and processes plays a fundamental role in our parallel implementation. In particular, message passing actually defines the order in which the tasks are executed. Processes that perform the analysis of an SCC send a message to inform Coordinator about their results and Coordinator sends a message to Spawn Server to let it know which SCCs are ready to be analyzed. From another point of view, Coordinator blocks until it receives the result of an analysis and Spawn Server blocks while waiting for ready SCCs.

Success Typings Inference Impelementation

Now it is time to present the exact implementation of the parallel success typings inference analysis. In Listing 5.1 we can see the code of the Coordinator and in Listing 5.2 the code of Spawn Process. Coordinator functions the way we described above, except for an additional checking it performs to decide when analysis is over. In fact, every time a SCC finishes the analysis and sends its results, Coordinator after receiving them deletes this SCC from the Call Graph. This way, when all analyses are over, this graph is empty and Coordinator realizes it is time to return the result of the analysis and stop Spawn Server's execution.

```
1  find_coordinator (NotFixpoint, CondensedDigraph, State, Fast, ServerPid) ->
2    receive
3      {results, SCC, SuccTypes, PltContracts, NewNotFixPoint1} ->
4        NewNotFixpoint2 = ordsets:union(NewNotFixPoint1, NotFixpoint),
5        ReadyParents = get_ready_parents(SCC, CondensedDigraph),
6        State1 = dialyzer_succ_typings:insert_into_plt(SuccTypes, State, true),
7        NewState = dialyzer_plt:insert_contract_list(State1, PltContracts, true),
8        case ReadyParents of
9          [] ->
10           ok;
11         _->
12           ServerPid ! {ReadyParents, NewState}
13       end,
14       true = digraph:del_vertex(CondensedDigraph, SCC),
15       case digraph:no_vertices(CondensedtheDigraph) =:= 0 of
16         true ->
17             ServerPid ! stop,
18             case dialyzer_succ_typings:check_fixpoint(NewNotFixpoint2,
19                                                       undefined,
20                                                       Fast, true) of
21                 false -> {not_fixpoint, NewNotFixpoint2, NewState};
22                 true ->  {fixpoint, NewState}
23             end;
24         false ->
25             find_coordinator(NewNotFixpoint2, CondensedDigraph, NewState, Fast,
                   ServerPid)
26       end
27   end.
```

Listing 5.1: Coordinator of Success typings inference analysis

Spawn Server also works as described above, spawning the function in Listing 5.3 for each ready SCC. This function actually calls the same function that is used in this analysis by the serial DIALYZER, but then sends the results as a message to Coordinator instead of

just returning them.

```
find_spawn_server(Parent, CondensedDigraph, ReadyList, State
                  Fast, CoordinatorPid) ->
  case digraph:no_vertices(CondensedDigraph) =:= 0 of
    true ->
      receive stop ->
      exit(normal)
      end;
    false ->
      _Pids = [spawn(
         fun() ->
             dialyzer_succ_typings:analyze_scc_parallel(SCC , State
                                                        CoordinatorPid, Fast)
         end) || SCC <- ReadyList],
      [update_info(SCC, Parent) || SCC <- ReadyList],
      receive {NewReadyList, NewState} -> ok end,
      find_spawn_server(Parent, CondensedDigraph, NewReadyList, NewState
                        Fast, CoordinatorPid)
  end.
```

Listing 5.2: Spawn Server of success typings inference

```
analyze_scc_parallel(SCC, State, CoordinatorPid, Fast) ->
  {SuccTypes, PltContracts, NotFixpoint, _NewState} =
    analyze_scc(SCC, State, Fast),
  CoordinatorPid ! {results, SCC, SuccTypes, PltContracts, NotFixpoint}.
```

Listing 5.3: Type inference function for a SCC in parallel analysis

Success Typings Refinement Impelementation

The code of the Coordinator and the Spawn Server of our parallel implementation of the refinement step are shown in Listing 5.4 and Listing 5.5 respectively. We notice that the only difference from the Coordinator of the success typings inference is that it separates the received results in two cases. If it receives a tuple starting with false, it means that the types of that module have been refined, so Coordinator stores them in the PLT, otherwise they have remained the same and they do not get stored.

However, there is also a difference in the implementation of the Spawn Server. When spawning the analysis of an SCC, the server checks if it includes one module or more than one and decides which of the analysis function shown in Listing 5.6 to spawn. In both cases the functions call the analysis function that was used in the serial analysis, but return its result by sending a message.

5.2.2 Parallel Warnings Collection

As we have mentioned several times in this chapter, parallelizing the warnings collection phase was the easiest part of our work. This simplicity is also obvious in the implementation of the Coordinator that we present in Listing 5.7. In this case there is no need for a Spawn Server, since spawning and result collection can be done by a single process without any impact in the performance. In fact this process, let us call it Coordinator, spawns a warning collection function (Listing 5.8) for each and every module of the

```
refine_coordinator(NotFixpoint, ModuleDigraph, State, ServerPid) ->
  receive
    {true, M, FixpointFromScc} ->
      ok;
    {false, M, DictFixpoint, FixpointFromScc} ->
      NewState = dialyzer_succ_typings:insert_into_plt(DictFixpoint, State,
                                                       true)
  end,
  NewFixpoint = ordsets:union(NotFixpoint, FixpointFromScc),
  ReadyParents = get_ready_parents(M, ModuleDigraph),
  case ReadyParents of
    [] ->
      ok;
    _->
      ServerPid ! {ReadyParents, NewState}
  end,
  true = digraph:del_vertex(ModuleDigraph, M),
  case digraph:no_vertices(ModuleDigraph) =:= 0 of
    true ->
      ServerPid ! stop,
      case NewFixpoint =:= [] of
    false -> {not_fixpoint, NewFixpoint, NewState};
    true ->  {fixpoint, NewState}
      end;
    false ->
      refine_coordinator(NewFixpoint, ModuleDigraph, NewState, ServerPid)
  end.
```

Listing 5.4: Coordinator of Success typings refinement analysis

Module Call Graph at once. Then Coordinator waits to receive the produced warnings for all modules, it gathers them in a list and returns them.

5.3 First Performance Results: Problems and Solutions

Having finished with replacing the serial analysis functions with those presented in the previous section, our parallel Dialyzer was ready. After checking the parallel analysis correctness, particularly that it produces the same warnings as the serial one, we got some time measurements to evaluate the speedup we achieved. However, those measurements were totally disappointing, since they revealed there was no performance improvement. On the contrary, results indicated that actually we had slowdown for every single benchmark we ran, together with very large memory consumption. In this chapter we study both the performance and memory problem and present the solutions we devised to deal with them and make parallel Dialyzer scale.

5.3.1 Problem Identification and Initial Solution

To deal with the lack of parallel speedup we had to understand what exactly was going on with our implementation. It was obvious that no conclusions could be made about time efficiency of our parallel program if we did not first eliminate the memory issue. So, we focused on it and tried to explain it.

```
refine_spawn_server(Parent, ModuleDigraph, State,
                    ReadyList, CoordinatorPid) ->
  case digraph:no_vertices(ModuleDigraph) =:= 0 of
    true ->
      receive stop ->
      exit(normal)
      end;
    false ->
       _Pids =
         [spawn(fun() ->
           case SCC of
             [M] ->
                dialyzer_succ_typings:refine_one_module_parallel(M,
                                                                 State,
                                                                 CoordinatorPid
                                                                     );
             [_|_] ->
                dialyzer_succ_typings:refine_one_scc_parallel(SCC,
                                                              State,
                                                              CoordinatorPid)
           end
         end) || SCC <- ReadyList],
      [update_info(SCC, Parent, "Dataflow of one SCC:") || SCC <- ReadyList],
      receive {NewReadyList, NewState} -> ok end,
      refine_spawn_server(Parent, ModuleDigraph, NewState,
                          NewReadyList, CoordinatorPid)
  end.
```

Listing 5.5: Spawn Server of success typings refinement

```
refine_one_module_parallel(M, State, CoordinatorPid) ->
  case refine_one_module(M, State) of
    {true, []} ->
      CoordinatorPid ! {true, [M], ordsets:new()};
    {false, NotFixpoint} ->
      CoordinatorPid ! {false, [M], NotFixpoint,
                        ordsets:from_list([FunLbl || {FunLbl,_Type} <-
                            NotFixpoint])}
  end.

refine_one_scc_parallel(SCC, State, CoordinatorPid) ->
  {_NewState, FixpointFromScc} = refine_one_scc(SCC, State),
  CoordinatorPid ! {true, SCC, FixpointFromScc}.
```

Listing 5.6: Type refinement function for a SCC in parallel analysis

Memory Problem

Being now aware that something is wrong with memory, we studied once again our implementation to identify the cause of the problem. What we immediately noticed is that the spawned processes we use to perform each analysis in parallel take record State as input, since their function needs the greatest part of the information stored in it. Specifically, the State record includes Call Graph, Code Server and PLT records, i.e. all fundamental structures where each step on the analysis is based. As a consequence, State turns out to be a quite large data structure as it contains information that ranges from

```
get_warnings_from_modules_parallel(Modules, State, BehavioursChk, CWarns) ->
  CoordinatorPid = self(),
  [spawn(
     fun() ->
     dialyzer_succ_typings:get_warnings_from_module_parallel(M, State,
                                                             BehavioursChk,
                                                             CoordinatorPid)
     end) || M <- Modules],
  Warns = [receive {warnings, Warnings} -> Warnings end || _M <- Modules],
  lists:flatten(CWarns++Warns).
```

Listing 5.7: Coordinator of warnings collection

```

get_warnings_from_module_parallel(M, State, BehavioursChk, CoordinatorPid) ->
  {Warnings1, Warnings2, Warnings3} = get_warnings_from_module(M, State,
                                                               BehavioursChk),
  CoordinatorPid ! {warnings, [Warnings1, Warnings2, Warnings3]}.
```

Listing 5.8: Function of warnings collection for a SCC in parallel analysis

types, to Call Graph and source code details.

Moreover, passing State as an argument in a spawned function equals to creating a local copy of this large structure for the new process to use it. Hence, passing State as an argument to many spawned processes at the same time will reasonably cause increased memory consumption due to the multiple large copies. In addition, it is not hard to realize that all this copying is also a time consuming procedure that prevents the parallel implementation from achieving a speedup.

Solution Presentation

To eliminate this problem we definitely need to stop passing the State as an argument to the analysis functions. However, even if we replace the argument passing with a message, no difference will be made since message passing involves local copying as well. Obtaining local copies of State, a classic approach in functional languages, is therefore out of the question. Therefore we can think of no other solution than maintaining a single shared copy of this State, where all processes could have concurrent access.

As we described in Chapter 2, even though it is said that Erlang does not have shared memory, there is a way of sharing data between processes using ETS tables. This data structure is known to assure highly efficient interprocess access to common data and is often an alternative when it is too costly to program with nondestructive assignment and "pure" Erlang data structures[1], as it is in our case. Still, when a lookup operation is performed on such a table, the resultant tuples are actually copied from the ETS table to the heap of the process. As a consequence, the language provide us with no solution that completely avoids copying. However, in this case it is possible to copy the minimum required data from each main data structure included in State rather than the whole record.

Therefore, to deal with memory overconsumption we ended up converting the dictionaries included in Call Graph, Code Server and PLT records into ETS tables. When inserting tuples to such an ETS table we tried to use as much refined key values as possible, in order to prevent processes from copying redundant information when performing a

lookup at this table. Every spawned process was then able to read, whenever required, the exact information it needed from the corresponding global table, performing fine-grained copying.

Solution Evaluation

Implementing this solution managed to significantly reduce memory consumption but it was not enough to help us totally avoid the slowdown. We thought ETS table locking could be a possible cause for it. Locking is required on this structures to protect against the simultaneous modification of shared memory. To reduce locking we defined our ETS tables as *protected*, meaning that only one process, named the Coordinator, has write access to them whereas every other process read-only access only. This way locks are not eliminated but are reduced since concurrent modifications of a table entry cannot happen. Still, the time of execution was not affected from this change and copying was not any more a sufficient explanation for this performance problem. So, we started once again studying our implementation, identifying in the end the Code Server as the one responsible of this weird performance behaviour.

5.3.2 Code Server Bottleneck

Particularly, a big part of each phase of the analysis execution time was found to be spent without an obvious reason in code fetch. Although we have used ETS tables to represent all basic data records, Call Graph, Code Server and PLT, we have not made any changes to the Code Server process that manages the code, since it was not a data structure included in State record but a separate process. However, this server turned out to be the actual culprit for parallel Dialyzer's slowdown.

First we remind that, as explained in Chapter 3, the Code Server process is in charge of fetching and expanding the code of a module or a function many times during each phase of the analysis, performing some form of data caching. However, in the parallel implementation this server receives too many simultaneous messages asking for code fetch since many processes analyze a module or a function in the same time and need to traverse its code. This way a big waiting list is created in server's inbox and it takes a long time to reply to all of them; it is an actual bottleneck. An additional reason for Code Server's delay in sending back the corresponding code is that we can no longer take advantage of the code cashing performed. In particular, during the success typings inference of the parallel implementation, code requests come in quite a random order and it is very rare for them to belong to the same Module. The way we used to perform the serial analysis, favoured successive request for functions belonging to the same module, but this is not the case any more. Finally, Call Graph server is also partially responsible for the memory overloading since the code sent back in form of a message is actually copied locally to be used by the function that asked for it.

To face this series of problems caused by the Code Server, we decided to replace it, once again, with a protected ETS table. This way all interested processes can copy the piece of code they need simultaneously, without waiting in a queue. To reduce copying we store the code with function name as a Key, so that processes can copy the code of a single function if they prefer instead of a whole module.

After applying all those changes, the problem was eventually solved: we got speed up! We present detailed performance results as well as comments about the parallel implementation behaviour in the next Chapter.

5.4 Parallelization Summary

The steps we took until we managed to efficiently parallelize Dialyzer, are summarized below:

1. Study of the serial application and understand of they way it works.
2. Description of the outline of the most suitable parallelization method.
3. Profiling of the sequential implementation and identification of any existing synchronization points.
4. Definition of the parallelization actors and the responsibilities they have.
5. Implementation of the designed parallel method and first testing of its performance.
6. Identification and handling of any problems that may have emerged.
7. Performance measurements and evaluation.

Chapter 6

Performance Results

In this chapter we are going to present some performance results of our parallel implementation of Dialyzer to demonstrate the speed up we achieved. Those results are actually measurements of the time Dialyzer consumes to analyze standard OTP applications as well as real world applications written in Erlang. Initially, we are going to provide some information about the size of each analyzed application and then present performance charts for each phase of the analysis as well as for the total execution time. Finally, we will try to explain the behaviour of the parallel program as reflected in those charts, drawing related conclusions whenever possible.

6.1 Analyzed applications

First of all we provide some information about the applications we run Dialyzer's analysis on, in order to evaluate the time results we got for each one. We also provide information about the number of lines of source code of each application, which gives some indication about their size. Additionally, it is useful to know the number of modules, functions and SCCs that are included in them. This information is shown in Table 6.1. As "set of applications" we refer to the following OTP applications analyzed together : *inets, percept, pman, public_key, reltool, ssl, test_server, toolbar, tools* and *webtool*.

Applications	Lines of Code	Number of Functions	Number of Function SCCs	Number of Modules	Number of Module SCCs
stdlib	74594	6667	4790	79	52
compiler	40096	3124	2023	41	41
kernel	38381	3022	2643	65	35
erts	7923	669	606	8	7
mnesia	24840	2002	1657	29	9
couchdb	17743	1799	1215	53	43
set of applications	3694	8009	6847	178	168

Table 6.1: Code information for analyzed applications

6.2 Performance Charts

The benchmarks were performed on a multiprocessor machine with four Intel Xeon E7340 CPUs (2.40 GHz), having a total of 16 cores and 16 GB of RAM, running Linux 2.6.26-2-amd64 and GCC 4.3.2. We run parallel Dialyzer for the applications presented in Table 6.1, timing each phase of the analysis separately and varying the number of used cores. In particular, using +S *Schedulers* argument of Dialyzer, we were able to set the number of logical processors that would be used during the analysis. This way we created a chart for each phase of the analysis as well as for the whole analysis, showing the changes in execution time of each benchmark while the number of CPUs grows. In those charts we include execution times for both the analysis of serial Dialyzer, labeled as "ser", and parallel Dialyzer running on a single core, labeled as "1".

Figures 6.1, 6.2 and 6.3 show the performance results for success typings inference, success typings refinement and warnings collection step respectively, whereas Figure 6.4 shows the overall performance speed up of parallel Dialyzer's execution.

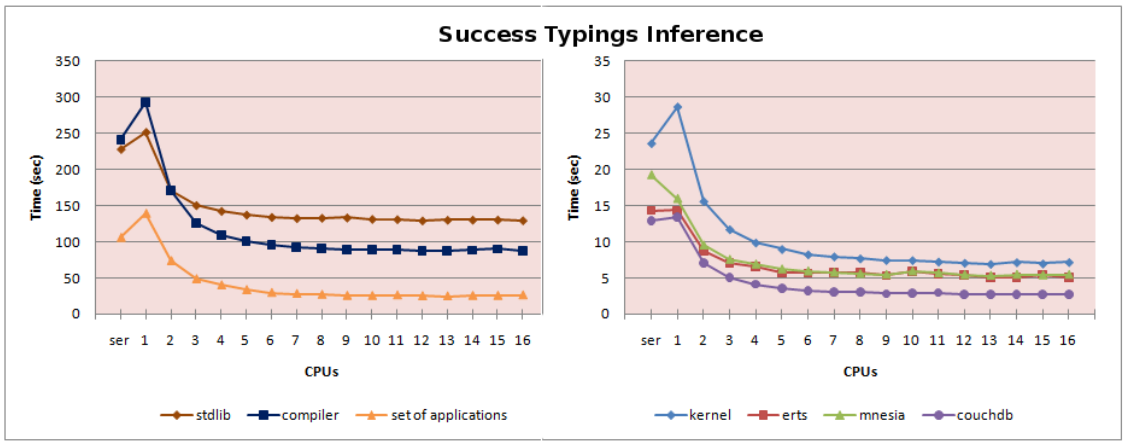

Figure 6.1: Performance results for success typings inference step

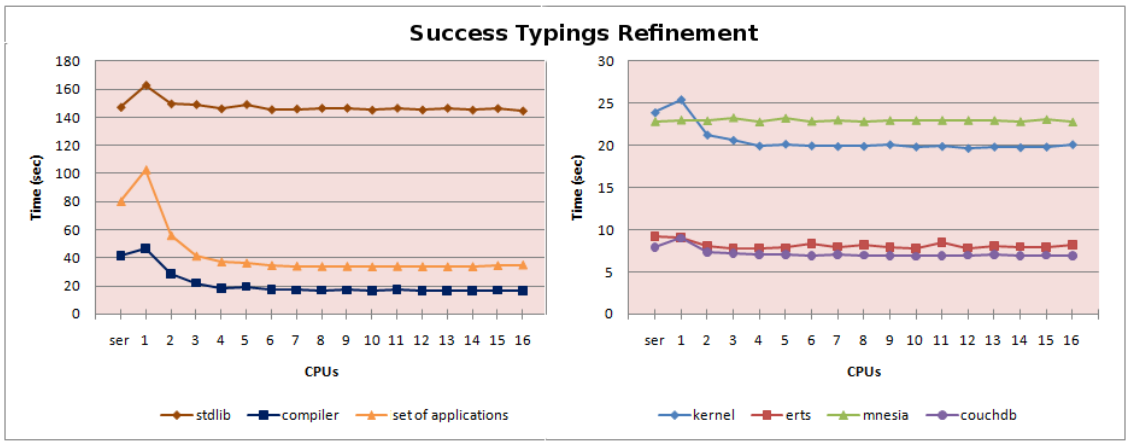

Figure 6.2: Performance results for success typings refinement step

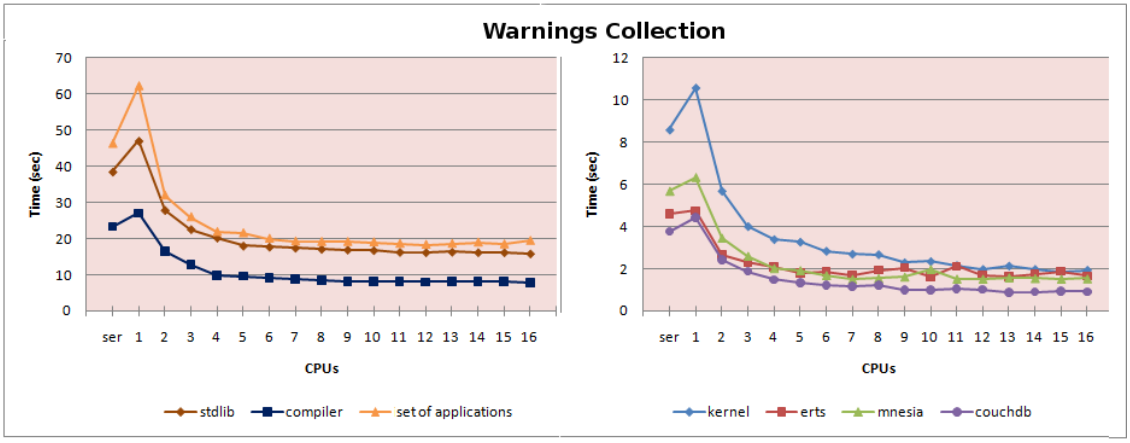

Figure 6.3: Performance results for warnings collection step

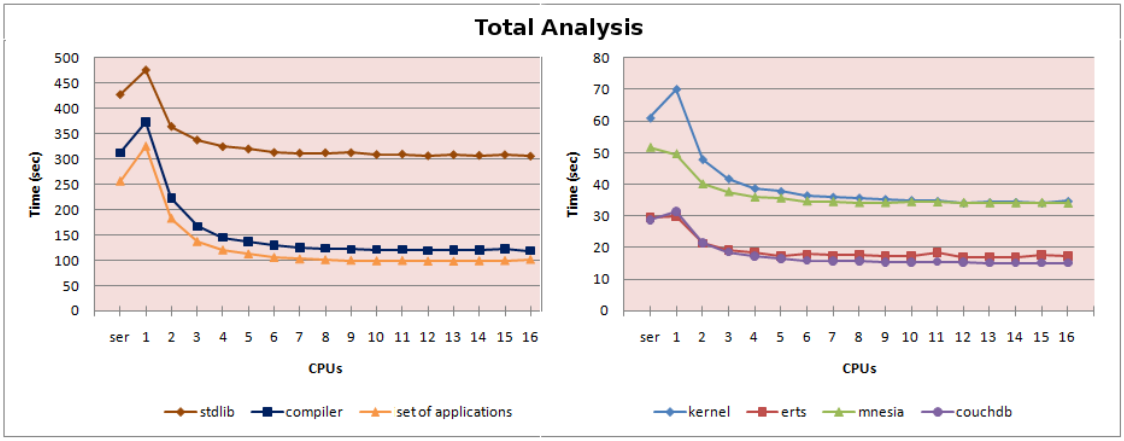

Figure 6.4: Performance results for total DIALYZER execution time

6.3 Performance Analysis: Comments and Conclusions

The performance results summarized in the charts of the previous chapter indicate that our parallel implementation of DIALYZER manages to significantly reduce analysis execution time. However, the speedup we get depends up to a certain level on the application DIALYZER is analyzing. Note that coordination of parallel processes has certain overhead which is clearly shown in the performance charts in the difference execution time for serial DIALYZER and execution time for the parallel version on one processor.

Particularly, in some cases such as *compiler* application (6.4) our implementation achieves a very satisfying absolute speedup[1] of appoximately 1.4 on 2 cores and 1.68 on 4 cores, as well as a very good relative speed up of appoximately 1.94 on 2 cores and 2.31 on 4 cores. However, in other cases such as *mnesia* application performance results are not so impressive since both absolute and relative speedup seems to be approximately 1.25 on 2 cores and 1.31 on 4-cores.

[1]We call "absolute speedup" the speedup of a parallel algorithm compared to the corresponding sequential algorithm, and "relative speedup" the speedup of a parallel algorithm compared to the same parallel algorithm on one processor.

6.3.1 Coarse-grained Parallelization

This variation in parallel Dialyzer's scalability is due to the input application characteristics. To explain that behaviour we first need to realize what exactly we have done. Our implementation was based on coarse-grained parallelization of the analysis Dialyzer performs, since the parallelization has not altered in any case the type or warning inference algorithm, but only the way it is applied. In other words, we simply focused on applying a certain function to a set of graph nodes in parallel, and not on modifying this function to increase its parallelism when called for the analysis of a single node. This immediately poses a natural bound to Dialyzer's performance: the analysis cannot last less than the time required to analyze the most-time consuming SCC of the Call Graph. This conclusion applies in all steps of the analysis.

For instance, if we take a deeper insight in the analysis of *mnesia* application we note that its reduced speedup is due to the absence of scaling at the refinement step(Figure 6.2). Studying *mnesia* characteristics in Table 6.1 we see that its module digraph consists of 9 module SCCs that include 29 modules. Further study of this application has shown that each module SCC contains one module except for one that contains 21! As a consequence the refinement of this SCC consumes far more time than the others, and is actually the one that determines how much this particular step is going to last. Indeed, preforming measurement we figured that 99.75% of the refinement step time is spent on the analysis of that 21-module SCC. Therefore, no further speedup can result since a single SCC creates a lower bound in execution time.

6.3.2 More Speedup Restraining Factors

Our performance results would have been even better if a number of factors that can limit its speedup was not present. If we try to identify most of them, we will better understand Dialyzer parallel behaviour.

Runnable Processes

It is a good practice, in order to achieve good scalability in SMP Erlang, to always have enough runnable processes to keep all schedulers busy. However, in our coarse-grained implementation it often happens for all schedulers to run out of work to do, except for one. This is the case when one big SCC is being analyzed and no other analysis process can be spawned, since every remaining unanalyzed SCC depends on it.

However, when the program causes frequent jumps in the number of runnable processes, the scheduler threads will be put to sleep by the Linux kernel if there are no available jobs, and a kernel call will be needed to wake them up again. Therefore, if a program quickly moves between having many runnable processes to few, some scheduler threads will constantly go to sleep and wake up, causing tremendous overhead[13].

Copying

As we explained in Chapter5 copying large amounts of data can cause both memory overconsumption and performance slowdown. In our implementation we managed eventually to significantly limit the amount of data that get copied by the concurrent processes. Using global data structures was an effective solution to excessive copying, but still a serious amount data is copied in every lookup to an ETS table, for example when a process needs to read the code of a module. The overhead of copying cannot be neglected and seriously affects parallel Dialyzer's performance.

Locks

Using global data structures even in ERLANG, means that at the implementation level locks are used for process synchronization. This is also the case with ETS tables that use a locking mechanism to protect against concurrent modifications, but locks cause overhead even when the chances of collisions are low[10]. We have tried to reduce locking by using *protected* ETS tables, but we could not eliminate them even though in our implementation the majority of ETS used are read-only so no simultaneous modifications can occur.

Chapter 7

Conclusion

This diploma thesis described the steps towards parallelization of Dialyzer including the obstacles encountered as well as the solutions applied. Since Dialyzer is a real-world application of significant size, widely used among Erlang programmers, this work offers useful experience and reliable conclusions to anyone who would like to try something similar: parallelize a big and non-trivial application written in Erlang.

To begin with a general conclusion, we can say that parallelizing an application written in Erlang can be hard and time-consuming. Although Erlang has very good support for concurrency, offering simplicity combined with performance, the language itself cannot do magic. Not all applications are the same, so not all applications can take full advantage of the infrastrusture a language offers. For example, Erlang is generally proud of its shared-nothing, message-passing philosophy. However in our case, those features did not manage to help us achieve good performance in the parallel implementation.

In fact, shared memory proved to be the only way of communication between processes that ensured speedup in the parallel analysis. This indicates that message passing, despite all its advantages, is not panacea. In fact, there are parallel applications, such as the one we implemented, where shared structures are indispensable and irreplaceable, usually because concurrent processes need to exchange large pieces of information that is too expensive to copy. Messages are an excellent means of communication provided they remain lightweight, therefore are not suitable for this kind of applications that use and manipulate a large state.

Apart from the language support for concurrency, programming parallel applications can also benefit from profiling tools that can reveal how a program is actually executed. In Erlang there is a very limited number of such tools at the time, especially tools that can contribute to a better understandin of what is going on in a parallel application of non-trivial size and functionality. This makes things even more difficult for the programmer who ends up spending most of his time using custom methods, trying to figure out the bottlenecks and their reasons.

However, it would be unfair to neglect the important advantages of parallel programming in Erlang. One of the most important is the fact that processes are lightweight and the scheduling of processes is automatic. Being a responsibility of the runtime system, it helps the programmer concentrate on his work without caring about the way or the order the processes he spawns are going to be executed. As we saw in this thesis, there was no concern at any time of the parallelization process about scheduling; we just spawned a process when needed to.

To sum up, parallelizing this important Erlang application managed to reach its initial goal: efficiently parallelize Dialyzer. Further work, that includes perhaps using

shared data structures with less locking and copying, in other words more suitable to our needs, is part of our plans. However, this thesis beyond the actual speedup of Dialyzer it achieved, it also presented in a complete way the whole process of parallelization of an important Erlang application.

Bibliography

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

[2] Joe Armstrong, Robert Virding, Claes Wikstrom, and Mike Williams. *Concurrent Programming in ERLANG - Second Edition.* 1996.

[3] Thomas Arts, John Hughes, Joakim Johansson, and Ulf Wiger. Testing telecoms software with quviq quickcheck. In *ERLANG '06: Proceedings of the 2006 ACM SIGPLAN workshop on Erlang*, pages 2–10, New York, NY, USA, 2006. ACM.

[4] Maria I. Christakis. Race condition detection in concurrent erlang applications using static analysis. *Diploma Thesis*, September 2009.

[5] Jim Larson. Erlang for concurrent programming. *ACM Queue*, 2008.

[6] Tobias Lindahl and Konstantinos Sagonas. Detecting software defects in telecom applications through lightweight static analysis: A war story. volume 3302 of LNCS, pages 91–106, Berlin, Germany, September 2004. Springer.

[7] Tobias Lindahl and Konstantinos Sagonas. Typer: A type annotator of erlang code. *In Proceedings of the Fourth ACM SIGPLAN Erlang Workshop*, pages 17–25, September 2005.

[8] Tobias Lindahl and Konstantinos Sagonas. Practical type inference based on success typings. pages 167–178, New York, NY, USA, 2006. ACM.

[9] Manouk-Vartan Manoukian. Detection of opaque type violations in erlang using static analysis. *Diploma Thesis*, September 2009.

[10] Richard Carlsson Martin Logan, Eric Merritt. *Erlang and OTP in Action.* Manning, 2010.

[11] Konstantinos 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.

[12] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. 2005.

[13] Ulf Wiger. Erlang programming for multicore. Georgia, USA, January 2009.