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

Σχεδίαση και υλοποίηση ενός εργαλείου ανάλυσης της εκτέλεσης ενός προγράμματος σε πολυπύρηνες αρχιτεκτονικές για τη γλώσσα Erlang

Διπλωματική Εργασία

του

Αθανάσιου Τιντινίδη

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

Εργαστήριο Τεχνολογίας Λογισμικού
Αθήνα, Σεπτέμβριος 2012

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

Σχεδίαση και υλοποίηση ενός εργαλείου ανάλυσης της εκτέλεσης ενός προγράμματος σε πολυπύρηνες αρχιτεκτονικές για τη γλώσσα Erlang

Διπλωματική Εργασία

του

Αθανάσιου Τιντινίδη

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

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 19η Σεπτεμβρίου, 2012.

.....	.....	.....
Κωστής Σαγώνας	Νικόλαος Παπασπύρου	Νεκτάριος Κοζύρης
Αν. Καθηγητής Ε.Μ.Π.	Επικ. Καθηγητής Ε.Μ.Π.	Αν. Καθηγητής Ε.Μ.Π.

Αθήνα, Σεπτέμβριος 2012

....

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

Περίληψη

Η μετάβαση σε πολυπύρηνες αρχιτεκτονικές παρουσιάζει διάφορες προκλήσεις - τα προγράμματα πρέπει να παραλλοποιηθούν αποδοτικά, αποφεύγοντας προβλήματα όπως αμοιβαίος αποκλεισμός και σημεία συμφόρησης. Ακόμα και σε γλώσσες προγραμματισμού όπως η Erlang, της οποίας το μοντέλο ταυτοχρονισμού και οι βάσεις της στο συναρτησιακό προγραμματισμό επιτρέπουν στο προγραμματιστή να αποφύγει τις περισσότερες μορφές αμοιβαίου αποκλεισμού και συνθηκών ανταγωνισμού και να δημιουργήσει ασφαλείς, επεκτάσιμες και αποδοτικές εφαρμογές, πολλές φορές είναι αναγκαίο να εξεταστεί η πορεία εκτέλεσης ώστε να εντοπιστούν σημεία συμφόρησης και άλλα προβλήματα που περιορίζουν την επιτάχυνση λόγω των πολλαπλών πυρήνων.

Σε αυτή τη διπλωματική εργασία θα περιγράψουμε την αρχιτεκτονική, σχεδιαστικές αποφάσεις και λεπτομέρειες υλοποίησης ενός νέου εργαλείου profiling για την Erlang με το όνομα Sched-plot. Το εργαλείο αυτό συλλέγει δεδομένα κατά τη διάρκεια εκτέλεσης ενός προγράμματος Erlang και στη συνέχεια παρουσιάζει το βαθμό αξιοποίησης ανά το χρόνο του κάθε scheduler (ο οποίος συνήθως αντιστοιχεί σε ένα νήμα ΚΜΕ που αντιστοιχεί σε ένα πυρήνα) και του συλλέκτη σκουπιδιών. Προσπαθήσαμε να σχεδιάσουμε την αναπαράσταση ώστε ο μεγάλος όγκος δεδομένων, που παράγεται από την εκτέλεση προγραμμάτων Erlang με υψηλό βαθμό παραλληλοποίησης σε πολυπύρηνες αρχιτεκτονικές, να είναι εύκολα κατανοητός. Ένα από τα κυριότερα ζήτημα ήταν η δειγματοληψία του χρονοδιαγράμματος ώστε να προκύψει μια σμίκρυνση που να μην αλλοιώνει τη σημασιολογία. Επίσης, για τη γρήγορη αποθήκευση των δεδομένων χωρίς σημαντική επιβάρυνση του χρόνου εκτέλεσης σχεδιάστηκε μια δυαδική κωδικοποίηση. Τέλος, μελετήσαμε την επιβάρυνση εξαιτίας της χρήσης της erlang:trace/3 που έδειξε ότι περαιτέρω βελτίωση θα επιτύχουν σημαντική βελτίωση της επεκτασιμότητας και μείωση της επιβάρυνσης.

Λέξεις Κλειδιά

πολυπύρηνες αρχιτεκτονικές, παραλληλοποίηση, ταυτοχρονισμός, Erlang, profiler, scalability, σημείο συμφόρησης

Abstract

The shift towards multi-core architectures poses several challenges in software development as programs should be efficiently parallelized while avoiding related dangers such as deadlocks and bottlenecks. Even in languages such as Erlang, whose concurrency model and functional foundations enable the programmer to avoid most deadlocks and race conditions and build safe, scalable and efficient applications, it is often required to examine the execution in order to detect bottlenecks and other factors that limit the speedup.

This thesis describes the architecture, design choices and implementation details of a new profiling tool for Erlang named Schedplot. Schedplot gathers data during the execution time and then, offline, visualizes the workload of each scheduler (which most of the time maps in a CPU-thread which maps in a core) and the garbage collector. We tried to optimize the graphical representation in order to aid the user's comprehension of the large volume of data generated by highly scalable Erlang programs run in multicore architectures; one of the main problems encountered was sampling the timeline to provide a zoomed-out view without altering the semantics of the produced representation. Another challenge was storing the produced data sufficiently fast and minimize the overhead caused; according to the evaluation, by using a custom binary encoding, it was reduced to the point of being insignificant. Last, we examined the overhead caused by using erlang:trace/3 concluding that further improvements should be made to improve the scalability and reduce the overall overhead.

Keywords

multi-core architectures, parallel, concurrency, Erlang, profiler, scalability, bottleneck, visualization

Acknowledgements

I would like to express my gratitude to my parents for their love and support as well as their dedication and focus on my education; for teaching me how to learn; for introducing me to the crystal clarity of mathematics and logic; for inspiring me to seek reason and proof behind everything and, finally, for, no matter the physical distance, being always there for me.

I thank my advisor, Kostis Sagonas, for his advice and insights which, even though not always by following them, taught me important lessons about software engineering. But I would be forever grateful for introducing me, since my first undergraduate years, to the mind-boggling world of functional and logical programming and inspiring me to pursue this vastly rewarding path.

I would also like to thank the professors and students in NTUA's Software Engineering Lab for all their support and input.

A big thanks to my friend and mentor Lefteris for his *pure* enthusiasm and support all these years and particularly for all his help with Haskell.

Last, I am forever indebted to all cosmic rays that powered M-x butterfly, enabling me to write this thesis.

Thanos Tintinidis

Contents

List of Figures

List of Listings

Chapter 1

Introduction

For years, most applications have enjoyed free and regular performance gain with minimal to no alterations, since CPU manufacturers have created faster and faster systems, eager to fulfill Moore's Law. However, all good things come to an end; exponential growth cannot continue for long before approaching physical barriers and hence slow down and eventually stop. There is already an observable trend: most processor manufacturers turn from improving clock speed and optimizing execution flow to multicore architectures[19]. Unfortunately, for programs written in most programming languages, having 16 cores of 1GHz is not equivalent to 1 core of 16GHz without major changes to their structure; at least not until an algorithm for automatic parallelization is found - which is not something expected in the foreseeable future. This naturally leads, in order to get the much desired performance improvement due to advances in new architectures, to the adoption of paradigms with emphasis on – or at least awareness of – concurrency and to focus on writing concurrent applications with good scalability.

Alas, this is easier said than done, especially for problems with not inherent parallelizable nature and the complications that may arise due to locks and race conditions in some programming paradigms. Various tools have been developed to aid the programmer in debugging and optimizing a parallelized application, both for Erlang and other programming languages. However, the is a lack of tools for Erlang that focus on visualizing the scheduler workload over time.

This thesis focus on the building of such a tool, Schedplot, which, hopefully, sheds light to the exact execution pattern of a Erlang program by depicting the schedulers' and the garbage collector's activity over time. For our work we use Erlang's tracing mechanism, erlang:trace/3 and we studied ways to minimize the overhead caused as well as parallelizing the profiling and the data analysis in order to make the tool as scalable as possible. Special consideration was given to the design of the plotting part not only to provide a intuitive graph but also to enable a fast, real-time environment. Last, a message printing mechanism was created that allows the user to orchestrate the code and print messages directly on the graph to better understand the execution flow.

1.1 Outline of the thesis

The rest of the thesis is organized as follows:

- Chapter 2 gives an introduction of the basic concepts regarding concurrency, parallelism, related metrics and limitations as well as an overview of the support of SMP in the Erlang VM.
- Chapter 3 presents a detailed guide for using Schedplot
- Chapter 4 provides an in-depth description of the Schedplot implementation including design choices
- Chapter 5 focuses on the overhead of Schedplot and other aspects related to the evaluation of the current implementation
- Chapter 6 gives an overview of related tools in Erlang and in other programing languages
- Chapter 7 describes improvements that can be made to increase the functionality as well as the performance of Schedplot in the future

Chapter 2

Background

2.1 Concurrency vs Parallelism

The concept of concurrency may be easily mistaken with parallelism; however they are vastly different. Concurrency is a property of a system: multiple computations may be performed simultaneously (with the possibility of interactions between them); parallelism on the other hand is a run-time behavior of executing some tasks at the same time.

It is worth noting that support for multiprocessing in Erlang was added in 2006[13], 20 years after the appearance of the language, further illustrating the difference between concurrency and parallelism. Of course, one might point out that most concurrent Erlang programs demonstrated good parallel behaviour, without any further modifications, when the support for multiprocessing was added but that would rather indicate the suitability of the model for multi-core architectures.

2.2 Performance Metrics of Parallel Programs

There are various performance metrics used in evaluating a parallel algorithm such as speedup, efficiency and the Karp-Flatt metric[17].

1. Speedup expresses how faster a parallel algorithm runs when more processors are added; it is defined by the following formula:

   $$S_p = T_p/T_1$$

   where

   p is the number of processors

   T_1 is the execution time of the sequential algorithm

   T_p is the execution time of the parallel algorithm with p processors

   Alternatively, T_1 may be the execution time of the parallel algorithm when only 1 processor is being used; that value may be more representative of the performance improvement since it factors in some overhead due to the parallelization of the algorithm.

The values of the speedup usually range from 0 to p; values less than 1 imply that the overhead of parallelization outweighs the improvement. Ideally, speedup would be equal to p (linear or ideal speedup).

It is interesting to note that there are cases of super-linear speedup[3] (when the speedup is greater than `p`). This is often due to the cache effect: in a multi-core architecture, the addition of one core is accompanied with the addition of extra cache that could increase the execution speed.

2. Efficiency measures how well the processors are utilized; it is defined by the formula:

$$E_p = S_p/p$$

where

p is the number of processors

S_p is the speedup

Naturally, the values range from 0 to 1; the efficiency of algorithms with linear speedup is 1 while sequential algorithms' efficiency is $1/\log(p)$.

Efficiency is often used for graphs instead of speedup since:

(a) all of the area in the graph is useful (while in a speedup curve half of the space is wasted)

(b) it is easy to see how well parallelization is working

(c) there is no need to plot a "perfect speedup" line

3. The Karp-Flatt metric is a measure of parallelization of the algorithm that indicates the extend of which the algorithm is parallelized[11]. It is defined by the formula:

$$e = \frac{\frac{1}{S_p} - \frac{1}{p}}{1 - \frac{1}{p}} = \frac{\frac{p}{S_p} - 1}{p - 1}$$

where

p is the number of processors

S_p is the speedup

2.3 Reasons for non-linear speedup

As mentioned above, the speedup of a parallel program is not always ideal; there are many reasons for that[12]:

1. overhead caused by the parallelization and the addition of safety mechanisms such as locks

2. overhead caused by the need for communication between processes

3. there is a number of operations in the program that must be performed sequentially (for example we have to open a file before we read it and write to it before we close it). This is known as Amdahl's law: the speedup of a parallel program is limited by its sequential part. Assuming that:

$$T_1 = T_{seq} + T_{par}$$

$$T_p = T_{seq} + T_{par}/p$$

where

p is the number of processors

T_{seq} is the execution time of the sequential part of the program

T_{par} is the execution time of the parallelized part of the program

The speedup would be:

$$S_p = \frac{T_1}{T_p} = \frac{T_{seq} + T_{par}}{T_{seq} + T_{par}/p}$$

As a result, the speedup is limited to:

$$\lim_{p\to\infty} S = \lim_{p\to\infty} \frac{T_{seq} + T_{par}}{T_{seq} + T_{par}/p} = \frac{T_{seq} + T_{par}}{T_{seq}} = 1 + \frac{T_{par}}{T_{seq}}$$

Note that this is just an upper limit since the speedup may be further reduced due to overhead and thus fall bellow 1.

4. the size of the program's input: while a program may have a high speedup with a large input, it may not be so efficient with a smaller input even if it is an embarrassingly parallel problem. This is known as Gustafson's Law which states that computations involving arbitrarily large data sets can be efficiently parallelized. For example, a program that generates a list of random numbers may little to no speedup when only a few random numbers are requested while its speedup could be very close to ideal when a large number (in comparison to the number of cores) of random numbers is requested instead.

2.4 Support for SMP in the Erlang VM

Researching ways to provide support for symmetrical multi processor (SMP) in Erlang started as early as 1997-1998[8] yet it was not continued until 2005 when it was restarted as part of the ordinary development; the first release of a stable runtime system with support for SMP came in OTP R11B in May 2006.

The Erlang VM without SMP support has 1 scheduler running in the main process thread (note that there may be more threads, for example, for asynchronous IO). The scheduler picks runnable processes and IO-jobs from the run-queue – there is no need to lock datastructures as only one thread access them (Figure 2.1).

In order to support SMP, multiple schedulers were introduced; usually, the number of schedulers used is equal to the number of physical cores of the machine as there is no point of introducing more. It should be noted, however, that it is possible that more

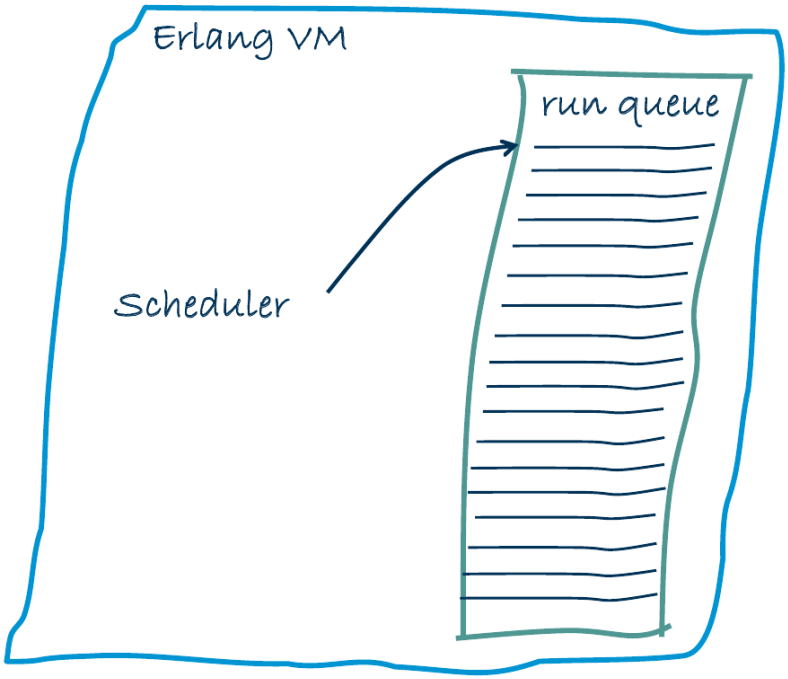

Figure 2.1: Erlang VM without SMP support

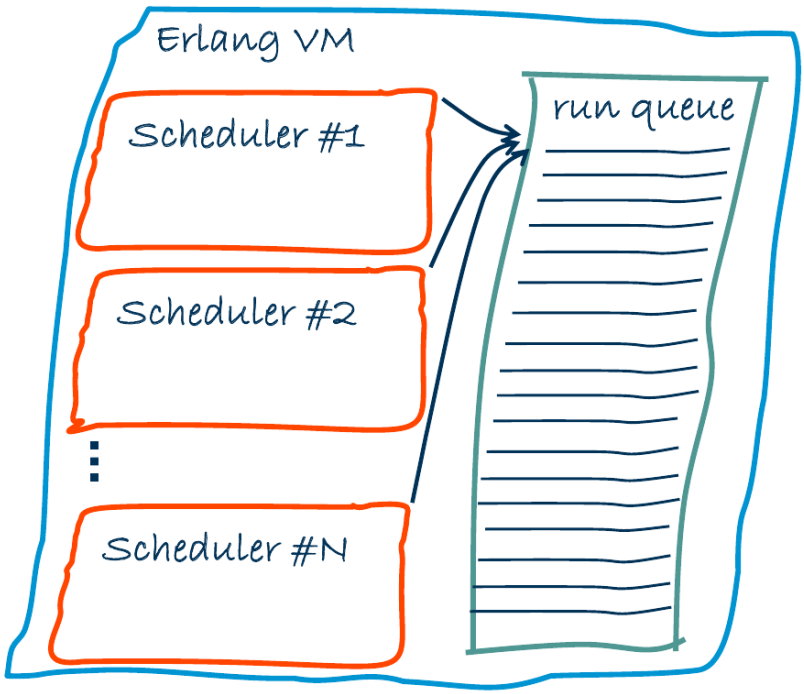

Figure 2.2: Erlang VM with SMP support

schedulers would sometimes slow down a program. Due to the sharing by the multiple schedulers, it is required to protect datastructures, such as the run queue, with locks; the run queue is one of those (Figure 2.2).

As the number of schedulers increases, the single common queue becomes a major bottleneck; to fix this, separate run queues, one per scheduler, were introduced, decreasing the number of lock conflicts dramatically for systems with many cores. In order to avoid imbalance in the schedulers' load, a migration strategy was adopted (Figure 2.3).

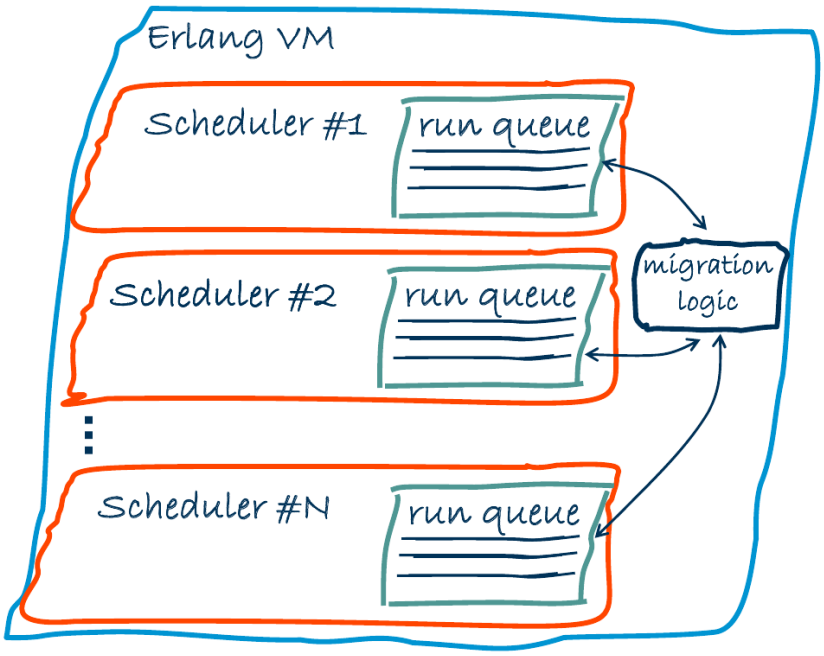

Figure 2.3: Improved Erlang VM with SMP support

It is important to note that the number of the schedulers may not correspond to the number of physical cores available to the Erlang VM since on some operating systems, the number of cores used by an application can be restricted. Moreover, the schedulers run on one OS-thread each and therefore, whether or not each scheduler runs in a different core is up to the operating system, as well as whether the thread will remain on the same core during the execution[13].

Chapter 3

Using Schedplot

Schedplot is normally used in three sequential steps: profile, analyze, view.

3.1 Profile

In this step, Schedplot profiles the desired program using `erlang:trace/3`. The simplest way to start the profiling is by using `schedplot:start/[1-3]`

`start(Fun) → 'ok'.`
Same as `start(Fun, ''schedplot_trace'', [])`

`start(Fun, Dir) → 'ok'`
Same as `start(Fun, Dir, [])`

`start(Fun, Dir, Flags) → 'ok'`

Types:

```
Fun = fun()

Dir = file:filename() | atom() | integer()

Flags = [start_flag()]

start_flag = 'gc' | 'no_auto_stop' | 'trace_all' | 'trace_mfa'.
```

`Dir` should be an (existing or not) directory name. If the directory does not exist it will be (attempted to be) created. Missing parent directories will not be created. In this directory all the trace files will be stored – if there are any files with the names used by schedplot to store the trace files, they will be overwritten.

After the initalizations, `schedplot:start/3` will use `erlang:apply/2` to run `Fun` in a separate process. The process that applied `Fun` as well as its children will be the only ones traced (unless the `trace_all` flag is used). Once `Fun` returns, the profiling will stop (unless the `no_auto_stop` flag is given) and the trace will be saved. It is therefore of utmost importance to make certain that `Fun` will not return before we want to end the tracing. Consider for example the following code:

```
1 fib(N, M) ->
2     \_ = [spawn(fun() -> fib(N) end) || \_ <- L],
3     ok.
```

Listing 3.1: Wrong way of profiling a program

which spawns M processes that will calculate the nth fibonacci number. One might expect to trace it by calling `schedplot(module,fib,[N,M])` but that would only trace the spawning part (a few micro-seconds).

To trace the whole program something like the following should be done:

```
1 fib(N, M) ->
2     L = lists:seq(1, M),
3     \_ = [spawn(fun() -> fib\_w(self(), N) end) || \_ <- L],
4     \_ = [receive ok -> ok end || \_ <- L],
5     ok.
```

Listing 3.2: Correct way of profiling a program

where `fib_w/2 is` a function wrapper that would send ok after calculating fib(N).

Of course, there could be different approaches but the gist is that the first Fun termination will signal the end of the tracing.

If the `no_auto_stop` flag is used however, the tracing will continue until `stop/0` is called. It is up to the user to either call it in the program under whatever conditions he wishes or by hand during execution.

The profiling will produce N files, one per scheduler trace plus one if the gc flag was used, named ‘‘`raw_trace`’’`++N` where N is an integer denoting the scheduler ID. It will also produce a header file named raw_trace_header

Note that `schedplot:start/[1-3]` may be called from within a program multiple times; however, if the directory name is the same only the last trace will remain It is also impossible at the moment to combine multiple traces in one.

Flags

gc: enables the tracing of the garbage collector.

trace_all: will trace all processes in the node (including the tracer)

trace_mfa: the tracer will store the `PID` and `MFArgs` when a traced process enters or leaves a scheduler (of course the `PID` will be the same both times). Currently there is no use for those data from schedplot's viewer but they could be used later or by the user

no_auto_stop: the tracing will not stop when `Fun` returns

3.1.1 Using `schedplot:print/1`

```
print(Label) → 'ok'
Label = string() | atom() | integer()
```

There is no need to use any flags in `start/[1-3]` or start other tracing processes; just call `schedplot:print/1` from the desired points of the profiled program. Note however that it is not recommended to make too many calls (over 100 thousand)

3.2 Analyze

Analyzing is a straight-forward procedure; simply use the following functions to produce the files that will be used by the viewer.

```
analyze() → 'ok'
```
same with `analyze(''schedplot_trace'')`

```
analyze(Dir) → 'ok'
```
same with `analyze(Dir, [])`

```
analyze(Dir, Flags)
```

`Dir` is the name of the directory where the raw trace files are stored and where the analyzed trace files will be stored (therefore it is required to have both read and write privileges in that folder).

Analyzing the traces usually is almost instantaneous and can be done in a different machine than the one used to do the profiling; generally it will be faster when the number of schedulers increases up to the point that we reach the number of schedulers used in the profiled program. It is also interesting to note that the speedup will be similar to the speedup of the profiled program meaning that analyzing the trace of a sequential program that ran in N schedulers where N>1 will yield little to no speedup when run in M>1 schedulers. Typically the analyzed files would be larger (yet not a lot larger) than the raw trace files, so, considering the speed of the analyzer, it might be better to store them for future use instead of the analyzed files if memory is an issue.

The analyzer will produce N files, one per scheduler trace plus one if the gc flag was used, named `''analyzed_trace''++N` where N is an integer denoting the scheduler ID. Afterwards the raw trace files are no longer required so it is safe to remove them.

3.3 View

To start the viewer use one of the following functions:

`view() → 'ok'` same with `view("schedplot_trace")`

`view(Dir) → 'ok'` same with `view(Dir, 1000,700)`

`view(Dir, Max_Width, Max_Height)` starts the viewer displaying the trace stored in `Dir` directory; the window's max width will be `Max_Width` and max height will be `Max_Height`.

Note that the final width and height will depend on the size of the graph: the viewer will start with a size that fits the graph at the maximum zoom possible without exceeding the max size given. It is not possible at the moment to detect the maximum screen size in a trivial way due to wxErlang limitations; however, it is possible to change the macro values to fit the screen resolution used in the schedplot.hrl file (macros WIDTH and HEIGHT).

It is suggested to run the viewer in a local machine to avoid any network latency

Once the viewer started, the following keys are used for navigation:

- Move right/left:
  use the arrow keys (right/left) or 4/6 numpad keys to move (50px)
  if the alt key is pressed the movement will be smaller (10px)
  if the control key is pressed the movement will be larger (200px)
- Move up/down:
  use the arrow keys (up/down) or 8/2 numpad keys to move 1 scheduler up or down
  if the alt key is pressed then it will move 10 schedulers up or down
- Zoom in/out:
  use the numpad +/- or the regular +/- or =/_ to zoom in or out
  zooming in will result in 2x while zooming out in 0.5x
  the zoom will preserve the same starting position; that is, if the graph started at 12.4 sec and ended at 22.4 sec the zoomed-in graph will start in 12.4sec and end at 17.4sec and the zoomed-out graph will start in 12.4sec and end at 32.4sec
- Select an area:
  Click the right mouse button and drag the mouse. Release the right button at the desired point. A cyan box will appear around the selected area (it does not matter if the dragging is from left to right or otherwise or if it constantly to the same direction; all that matters is the first and last point). It does not matter if an area was selected previously
- Cancel a selection:
  Press escape
- Zoom in to selection:
  While having selected an area press a zoom in key. The viewer will display an area starting at the start of the selection and using the maximum zoom (from the valid zoom levels) that would display the whole selection. It is therefore possible to have some extra data displayed at the end of the graph.

- Reset:
  Press the Home key to return to the first state of the viewer.

Note it is not possible to zoom in or out more than a certain number while displaying times before zero is possible although it has little practical meaning.

Chapter 4

Implementation Details

4.1 Profiler

The profiler is the most crucial part of the tool; careful consideration was given to make it as lightweight as possible and minimize overhead.

For extracting the required information from the Erlang VM, `erlang:trace/3` was used with the following flags:

running to record when a process enters or exits a scheduler

scheduler_id to get the ID of the scheduler the process entered or exited

timestamp (or cpu_timestamp) to get the time the process entered or exited

set_on_spawn to trace the children of the process (redundant if all the processes are being traced)

trace, MasterTracerPID to send the trace messages to the MasterTracer process

gc (optionally) to monitor garbage collection activity

The above flags result in messages of the following format:

```
1 {trace_ts,PID,IO,SID,MFA,Time}
```

Listing 4.1: trace message record

where

PID: the PID of the process that entered/exited the schedulers

IO: `'in'` or `'out'` depending on whether the process entered or exited

SID: the scheduler ID

MFA: the MFA the process was executing at that moment

Time: a timestamp (MS,S,US)

The messages are delivered to a process named master tracer which in turn forwards them to tracers depending on the `SID`. An alternative structure would be using only one process in order to minimize the influence of the tracing - this would be improved if we could also bind the process in a scheduler. However, this structure is not scallable as one process has to encode all the messages. The master trace forwards a message with the form `IO, PID,MFA,Time` if the user requested to store the `MFA` and `PID` information otherwise the message is just `IO, Time`.

In case garbage collection tracing has been enabled the master tracer also receives messages with the format:

```
1 {trace\_ts,PID,SE,GC\_Info,Time}
```

Listing 4.2: GC trace message record

where

SE: `gc_start` or `gc_end`

GC_Info: garbage collection information that are ignored

In that case the forwarded message has the following format:

```
1 {SE, {PID, {gc, gc, 0}, Time}} or {SE, {Time}}
```

As it can be easily seen, the master tracer has a severe bottleneck; however, it cannot be avoided since the current implementation of `erlang:trace/3` cannot send the messages in different tracers, let alone sending messages in tracers depending on the `scheduler_id` field of the trace message record.

Each tracer receives messages regarding the activity of one scheduler. The tracer combines every `'in'` message with its equivalent `'out'` message (or `gc_start` with `gc_end`). The messages are delivered in chronological order therefore the task of matching them is reduced to simply waiting the next message.

It has been observed that sometimes it appears that some procedures only enter the scheduler and never exit; it was found that all them was executing the MFA `io,wait_io_mon_reply,2`. Therefore, master tracer was altered so that it will not forward such messages.

The main issue regarding the profiler is the volume of the messages: it is too large to store, even in a compressed format, in the RAM; we have tried lists, ETS tables and other datastructures but the results indicated that, even for small programs, it is not possible to avoid storing them on the disk.

At first we tried to use a DETS table and `io:write/2` but that was way too slow; after some experimentation we decided to use the flag raw when opening the file (with `file:open/2`) which allowed faster access. Further benchmarks were used to compare the flags used

to open the file; it was decided to use the flag compressed since it was slightly faster and the file size was reduced. We did not use the flag delayed_write, which buffers the calls to `file:write/2` and performs the call when a time or size limit is reached, despite the fact that it improved the speed by almost 50% (since amount of system calls are decreased). Instead we implemented manually a buffering system which accumulates N messages which, to our surprise, was faster than the build-in delayed write by around 50%. We suspect that the reason for that may be calls to `erlang:now/0` or similar timer functions that could require locking.

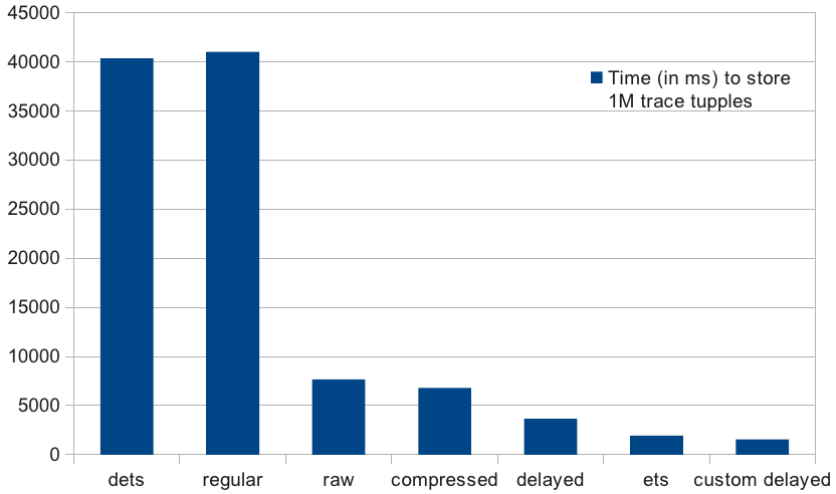

Figure 4.1: *

Time spent for I/O

- DETS/ETS stores the tuples in a dets/ets table
- Regular opens the file with no extra flags and uses `io:write/2`
- The rest open the file with the raw flag (and compressed/delayed) and use `term_to_binary/1` each time they write a tuple with `file:write/2`
- Custom delayed stores the trace messages after encoding them as binaries in a list and writes them when the list has 1000 elements.

The "drawback" of using the raw flag is not being able to use the io module; instead, we have to write the data using `file:write/2`; the data should also be binaries. First, we used term_to_binary/1 to convert the messages, however tests showed that the conversion was inefficient when dealing with a specific format such as the format of the messages. For example, a single pid, lets say <0.42.0> would be encoded as a binary of 27 bytes:

```
1 <<131,103,100,0,13,110,111,110,111,100,101,64,
2       110,111,104,111,115,116,0,0,0,42,0,0,0,0,0>>
```

The structure of a PID in Erlang on the other hand is <A.B.C>[1] where:

A: node id

B: process index which refers to the internal index in the proctab (15 bits)

C: Serial which increases every time MAXPROCS has been reached (2 bits)

Since we are tracing only processes in the local node, A would be always 0 and therefore 3 bytes would be more than enough; that's 9 times less than used by `term_to_binary/1`

Motivated by this it was decided to implement a binary encoding, reducing the size of the average message from 80 bytes to 5. The encoding is described bellow:

1. Instead of storing the times the process entered and exited a scheduler, we store the time difference between the time the process entered the scheduler and the time the previous process exited it as well as the duration the process was in the scheduler (in us). This not only reduces the bits used in the average case but also encodes the desired information with less bits in the case of heavy load (since then the duration of the processes as well as the time spend between one process exiting and the next entering will be smaller) and with more bits when the load is less resulting in less overhead due to encoding and writing the traces when the VM is busy and more when the schedulers are not so active. To strengthen this property we decided to use 6 bits to encode duration and 8 bits for time; in each case, if time is not lower than 2^{n-1} (n=6,8) then we set those bits to 1 and add another 8 bits to encode the rest $T–2^{n-1}$. While this could lead in an inefficient encoding for large numbers it was observed that in the average case, the encoding would be larger if we dedicated 1 or more bits as flags on every packet. A possible improvement could be done using duration as a flag: Duration should always be non-zero (while time between processes may be zero); therefore, we may encode larger times by setting duration (the 6 bits) to zero and add extra bits.

2. We observed that, in most cases, the MFA executed when the process enters and leaves the scheduler is the same so we can encode it just once. Therefore we use a flag that is set to 0 if the MFAs are the same and store it only once and 1 if there are different and we store both. Another optimization is done by not storing the actual names but storing the names in a dictionary and then storing just the key each time. The dictionary is created the following way:

   Each function (by function we mean the function name and arity) is inserted to the dictionary and assigned a 8-bit key. This key is returned each time we encounter

this function. We also store the module name. Naturally, there will be a conflict if another function (same name and arity) is called from a different module; in that case we return the 8-bit key along with a 4-bit key that denotes the module. Of course, to avoid using the 4-bit key all the time, the absence of 4-bit key indicates the module of the first function that was added to the table. This way we can encode functions of 17 different modules. This encoding achieves maximum efficiency when there are no close calls of functions with the same name and arity from different modules; something that is often true. When the dictionary is filled, we store it and start a new one.

3. The PID is, normally, stored in 16 bits: 15 bits for the PID and 1 bit for the flag. if the PID is larger then the flag is set to 1 and we add 8 more bits for the PID.

In the end, a regular pack would take 5 bytes have the following structure:

```
1 <<0:1, 0:1, Duration:6, TimeIn:8, MFA:8, 0:1, PID:15>>
```

Listing 4.3: format of encoded pack

If the first flag is 1 (MFAin is not the same with MFAout) then instead of MFA:8 we would have MFAin:8, MFAout:8. If the second flag is 1 (we need to encode the module) then 8 bits would be added: the first 4 to encode the module of MFA or MFAin and the last 4 to encode the MFAout (they are 0 if the MFAs are the same) If flag before the PID is 1 then 8 bits are added to the encoding of the PID Last, Duration and TimeIn can require more bits as described.

Note: if the user does not use the flag to save MFA and PID information, those will not be encoded neither stored (resulting in much smaller trace files and less overhead).

4.2 Analyzer

The encoded files have to be decoded and the information organized before it is possible to present the graph; at least if we want real-time interaction. The analyzer converts the encoded trace into a more useful format and also does additional calculations to avoid doing them -repeatedly- in real time.

The representation of the scheduler activity is a list of integers (timeline), each one corresponding to 1us; 0 when the scheduler is inactive and 1 when it is active. However, storing information this way consumes not only a lot of time during decoding but also disk space while providing little to no use, as, in most of the cases, we do no need to examine the schedulers' activity in such detail. Therefore we decided to limit the initial resolution by grouping a number of values; the parameter that defines the number of values is called grouping unit (gu) and the default value is 8, meaning that each integer of the list corresponds to 8us. A further optimization to reduce the disk size is using a variation of run-length encoding:

If we encounter the same value two or more times we encode it as `V* N` where `V* = «1:1, V:7»` where V is the value encountered and N is the number of repetitions

(this dictates that the value range is 0-127 so it can be stored in 7 bits resulting in 1 byte per value; this range is actually sufficient).

Otherwise we simply keep the values as it is: `«0:1,V:7»)`

N is kept in 1 byte and hence varies from 2 to 255; if there is a larger sequence of same numbers it is encoded in multiple packets.

This results in significant compression, mainly due to large sections of total activity (127) or inactivity (0).

Storing this information in one list is inefficient since we would have to read the whole list to use just a section so we split it in blocks of 4096 values and store it in a DETS table with appropriate keys.

Last, multiple DETS tables are used, not only because each DETS table has an upper limit of 2GB size which would become small in the case of a highly scalable program running in a machine with many cores, but also to improve the scalability of the analyzer since each process will read and write to different tables and would not have to wait to gain access to the table (DETS tables, since they are a shared resource, have locking mechanisms). A drawback to this is being limited by the maximum allowed number of simultaneously open files which depends on the maximum number of Erlang ports (size of one word as in R15) but as it is -for the current standards- large enough (and it can also be changed) this is not a significant drawback.

4.2.1 First phase: decoding

At the heart of the decoding process (Figure 4.2) there is a function that reads one encoded packet; this is achieved by reading bytes (to avoid multiple system calls to read bytes from the file, an input buffer is used) until we finish reading all the packet's values: Duration, TimeIn, MFA (in/out) and PID in case of a full packet or just Duration and TimeIn in case the user did not save information about the MFAs and PIDs.

After successfully reading a packet (in case of a malformed packet we ignore it and stop decoding while storing the already decoded packets; a malformed packet could only occur due to termination of the profiler before the packet storing was complete so there is no reason to continue decoding since the end of file is reached) we create the timeline that corresponds to the packet (time between this and the previous packet and duration) the length of which depends on the grouping unit and then the process continues to the next packet. When the produced list reaches the maximum size, it is compressed and then stored to the dets table. Using an output buffer for writing to the dets table provided minimal preformance improvement (bellow 1%) since the writes are already sparse. A possible optimization could be done by creating the timeline directly in the compressed format.

4.2.2 Second Phase: Generating the Zoom Levels

To ensure that the reaction time of plotting the graph will be small we decided to calculate the values that result from a zoom-out. The trade-off is limiting the scale to quantized values, to powers of two. After the creation of the initial timeline we create the rest of

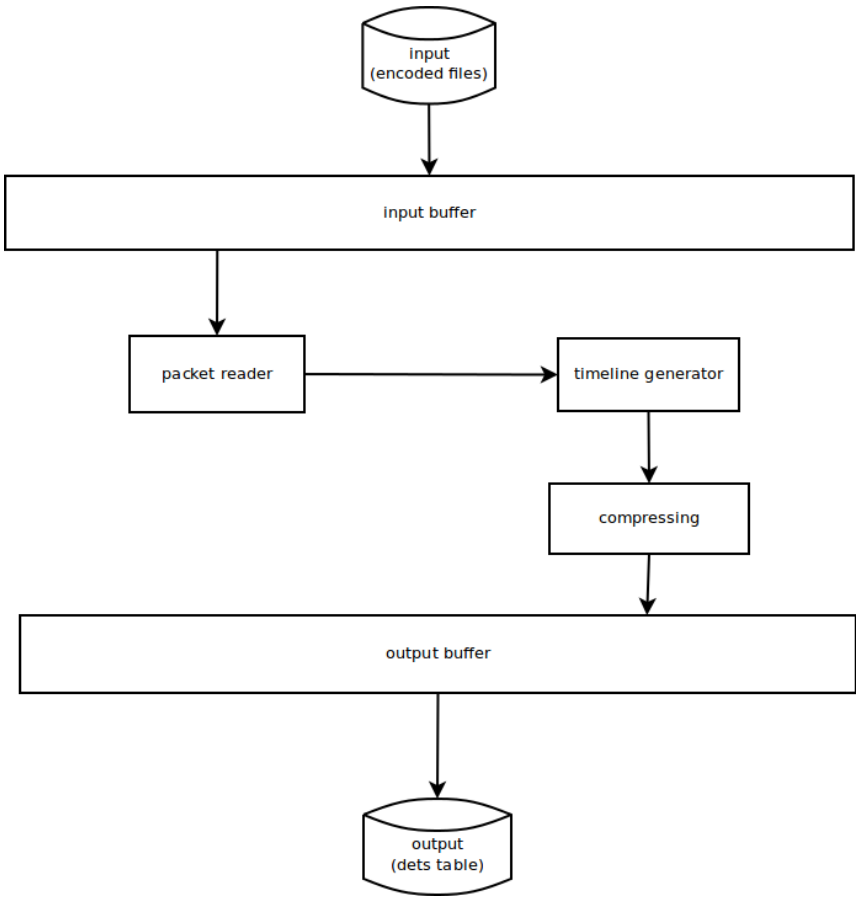

Figure 4.2: Decoding process

zoom-levels (1:2, 1:4,...) until the whole duration of the program's execution is represented in one value. The resulting structure is displayed in Figure 4.3 (assuming that we separated the timeline in blocks of size 4).

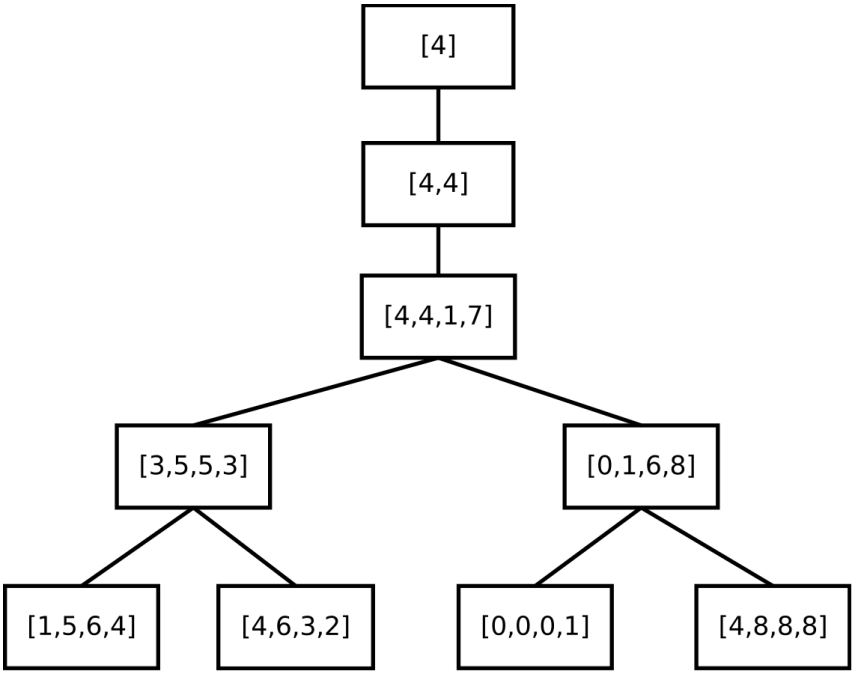

Figure 4.3: Zoom level generation process

Naturally this results in more disk space consumption; however, the overhead is never over 100%: assuming N initial values, the total number of values would be:

$$N + N/2 + N/4 + ... + N/2^n = (2–2^{-n}) * N < 2 * N$$

This does not take into consideration the extra space consumed by the keys for the last zoom levels when the number of values are less than 4096.

For example, the base values of the diagram above are 16 and the total number of values is 31; however, the number of keys required for storing the data as DETS table entries is $9 > 8 = 2 * 4$. The number of extra keys is $\log_2(size\_of\_block)$ and since the size of block is 4096 the extra keys are 12.

Each key consists of a tupple of 2 integers, one denoting the zoom level and the other the x position of the block so this overhead is insignificant.

The process of generating the zoom levels is simple: We read two blocks of data, concatenate them, combine the values in pairs of two and then write the block in the DETS table. When the compression scheme was introduced there was a significant overhead due to having to decode and re-encode the data before combining the values; however this was resolved by combining the encoded data without decoding them which not only eliminated the overhead of the compression but also sped up the process of generating the zoom levels since, instead of, for example, combining the elements of a list with $2 * N$ zeros in pairs

to create a list with N zeros, we simply replace $2 * N$ with N. This is one more reason for choosing a run-length based compression algorithm.

However, there could be a further optimization that is currently impossible due to using a DETS table. One might have noticed the redundancy of reading two blocks, creating the zoom-outed block and then writing it back to the disk only to read it in a later iteration; instead we could read one block and generate all the data that could be created as displayed in figures 4.4 and 4.5.

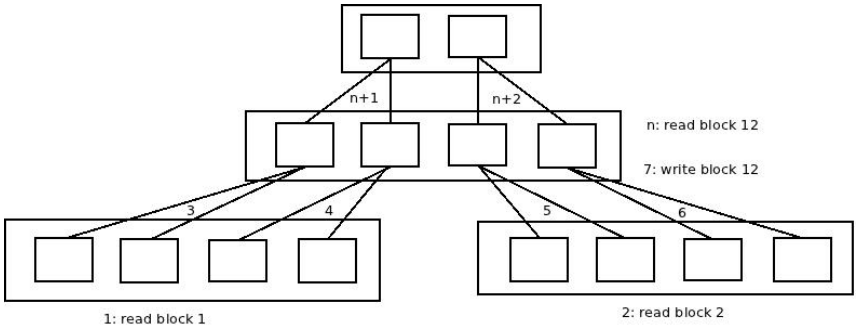

Figure 4.4: Current algorithm: each value is written and then read again

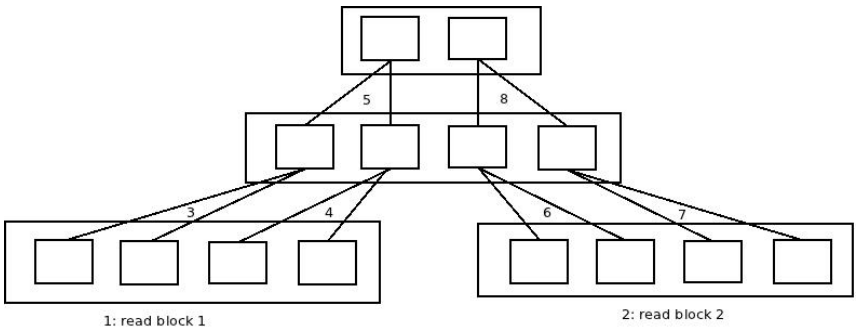

Figure 4.5: Optimization: the datablocks are read only once

The reason that this algorithm cannot be used with a DETS table is, as the diagram shows, that we would need to be able to write a not completed DETS entry and later fill it without reading the data already stored.

Note that besides the overhead of the disk operations there could be additional overhead avoided in case a more complex compression algorithm was used in the future that would not allow combining the data without decompressing them.

4.2.3 Zooming-Out Algorithm

The way data are zoomed-out in Schedplot differs from a lot of similar projects. Lets examine the following example:

Assume that we store the information in a list with zeros and ones, one denoting that there was a traced process in the scheduler that time interval (1us for erlang) and zero

that there was not. The raw data would be a list, [0, 0, 11, 11, 0, 0, 0, 11, 0, 11, 0, 11] for example.

Figure 4.6: zoom 1:1

For longer traces it is mandatory to compress this information by presenting the plot zoomed-out. A simple approach would be to combine the data in sets, calculate the average and round up. The resulting list (for combining in sets of 2) would be: [0, 11, 0, 11, 11, 11]. For higher zoom-out it would be [11, 0, 11] and eventually [11].

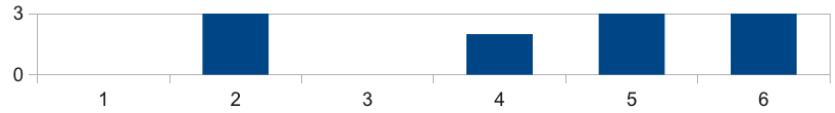

Figure 4.7: round, zoom 1:2

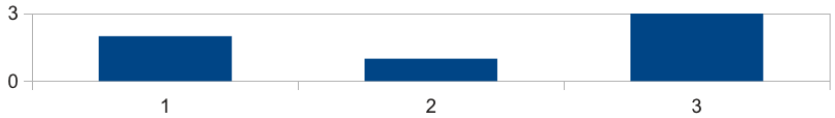

Figure 4.8: round, zoom 1:4

The problem is obvious: an activity that should be displayed as 58% was displayed as 100%. This is of course erroneous since it displays a program with 58% speedup as having ideal speedup. Even worse than that would be an approach where we use trunc (or ceiling) instead of rounding.

On the other hand however we cannot use floats; the graph is displayed in a pixelated screen and furthermore, if we desire to give the user the bigger picture we cannot have a lot of pixels to provide big accuracy, especially considering that, for a program that lasts just 1 second, the zoom-out to fit the screen would be 1:1000 for a moderately-sized screen. The approach we used to digitize the signal is a variation of rounding; there are three different values that we should make sure that are displayed at any scale at all costs: (1) the activity is maximum (2) the activity is 0 (3) the activity is somewhere in-between. Therefore, the minimum height for the display of one scheduler is 2 pixels: 11 for the first case, 00 for the second and 01 for the third. More available pixels would simply provide further information about how active is the core. So, assuming a display with 2 pixels length the original list would become: [00, 00, 11, 11, 00, 00, 00, 11, 11, 11, 11, 11] and the zoomed-out list would be [00, 11, 00, 01, 10, 10] then [11, 01, 10] and finally [01].

As it was mentioned earlier, the values are encoded in 7 bits (0-127): that way the produced graph will be accurate even if each core is displayed in 127 pixels. Considering that we aim to study the program's behavior in multi- and many-core architectures and the usual

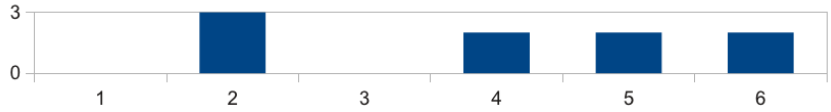

Figure 4.9: schedplot method, zoom 1:2

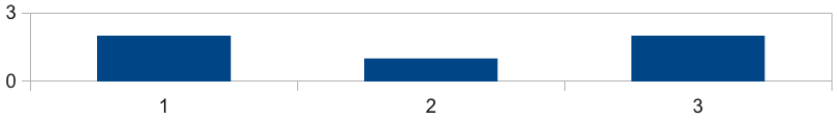

Figure 4.10: schedplot method, zoom 1:4

screen sizes, it is a sufficient resolution (or even too big). Of course, one more reason for using 7 bits instead of 8 is to improving the run-length compression by avoiding the overhead caused by having to encode single values as `1 c` and therefore doubling the size of the worst case scenario.

4.3 Plotter

To create the graphical representation of the scheduler activity we used wxErlang, the Erlang port of the wxWidgets, a widget toolkit for creating graphical user interfaces. By using wxWidgets, the program's GUI code can compile and run smoothly on several computer platforms such as GNU/Linux, Microsoft Windows and OS X. It is free and open source with a license compatible to GNU GPL[18].

The plotter consists of 3 main parts:

1. viewer: a module responsible for the creation of the windows, handling of events and general coordination
2. wxplot: a module that provides functions for drawing the actual graph
3. bets: a module that provides support for buffered dets tables

In the early stages of development the data were fetched upon request from the disk (where they were stored in the dets tables structure described previously). While the impact of the delay due to reading from the disk and, later, decoding the data, have been small, it was still noticeable, especially while scrolling left or right. To counter that, we used buffers which solved the problem.

The buffering strategy is not complicated: upon the first request, the dets tables are first queried only for the desired data and the fetched data are decoded and returned. Meanwhile, a process is spawned that fetches the data around to the current position, both in the x-axis and the zoom-axis. The data are then decoded, stored in a proper list and returned to the main process. Thus, in subsequent requests, the buffering module will first check the buffer resulting in a much smaller respond time if it's not a miss. Naturally,

in case of a miss, the dets table will be queried and a new buffer will be created around the new position; however this would rarely happen (mostly in very fast zoom-in/out or zoom to selection) since it is easy to predict the range of the data that will be requested and pre-fetch them (by spawning a process to query the dets table and decode them) if they are outside the current buffered range.

Chapter 5

Evaluation

5.1 Overhead

In this section we discuss the overhead introduced by the profiling a program; this may be caused by the following reasons:

1. the tracing itself (erlang:trace/3)
2. forwarding messages from the master tracer to tracers
3. encoding of messages
4. writing the messages to the disk

The total time spent for profiling a program can be separated in two parts:

1. the time the profiler and the program were executed in parallel
2. the time after the program ended during which the profiler encodes and saves the produced trace that was not saved during the program's execution

Our main interest is how the overhead affects the profiling of the program; so, while measuring the total time is important, our main concern is the time it takes for the program to execute when it is being profiled.

The worst case scenario, regarding overhead, is a program that achieves ideal speedup keeping all the schedulers busy and thus bombarding the tracers with messages which have to be encoded and stored. To study this case we used the following program:

which spawns N processes that recurse M million times. During the testing M was set to 10 and N varied from 1 to 64 in steps of 3 plus three extra tests where N was 128, 256 and 512. The schedulers used by the VM varied from 1 to 64 also in steps of 3. the time the profiler and the program were executed in parallel We run six tests measuring ten variables (all times are wall-clock):

1. run the program and measure its execution time (T_0)

```
seq(N, M) ->
    MM = M*1000000,
    Self = self(),
    L = lists:seq(1,N),
    \_ = [spawn(fun() -> seq\_(Self, MM) end || \_ <- L],
    \_ = [receive ok->ok end || \_ <- L],
    ok.

seq\_(PID, 0) -> PID ! ok;
seq\_(PID, M) -> seq\_(PID, M-1).
```

Listing 5.1: worst case scenario program

2. profile the program and measure
   - the time the profiler and the program were executed in parallel (T_p)
   - the full time (T_{fp})
   - the average size of the trace files (S_{rt})
3. analyze the raw trace files and measure:
   - time required to decode (T_d)
   - time required to generate the zoom levels (T_z)
   - the average size of the analyzed trace files (S_{at})
4. monitor the message queues of the tracers:
   - maximum length of the message queue of the master tracer (MQL_{mt})
   - maximum length of the message queue of the tracers (MQL_t)

Finally, at tests 5&6 we profile the program but use modified master tracers with the following code:

```
master\_tracer6() ->
        timer:sleep(infinity).

master\_tracer7() ->
        receive \_ → master\_tracer7() end.
```

Listing 5.2: alternative master tracers

and measure the time the profiler and the program were executed in parallel (T6, T7) in order to measure the overhead caused by simply using `erlang:trace/3`.

Each test is repeated three times; when time is being measured we keep the minimum value while for the rest we keep the maximum value.

5.1.1 Overhead due to `erlang:trace/3`

Table 5.1 displays the min, max, average and median values of the overhead(%) introduced during tests 2a, 6, 7.

Overhead (%)	storing	sleeping	ignoring
min	24	12	17
max	2747	2786	2775
average	809	805	813
median	752	727	750

Table 5.1:

The differences between the three methods are small; moreover, in some cases, the values of 6, 7 are greater than the values of 2a. This is further illustrated in figures 5.1, 5.2 which display the difference between the overhead for 2a and 6,7: the overhead of Schedplot's profiler is slightly greater (not even 1% at the worst case) than the overhead of a profiler that sleeps or ignores the messages most of the times (especially when the number of schedulers is small) while, in some cases it is smaller. This came as a suprise since one would expect that storing the messages will have greater overhead than simply ignoring them or doing nothing at all. From that we may conclude that the overhead is introduced by the use of erlang:trace/3 and not from the way the traces are encoded or stored, meaning that further improvements would require to make changes in the way the processes are being traced internally.

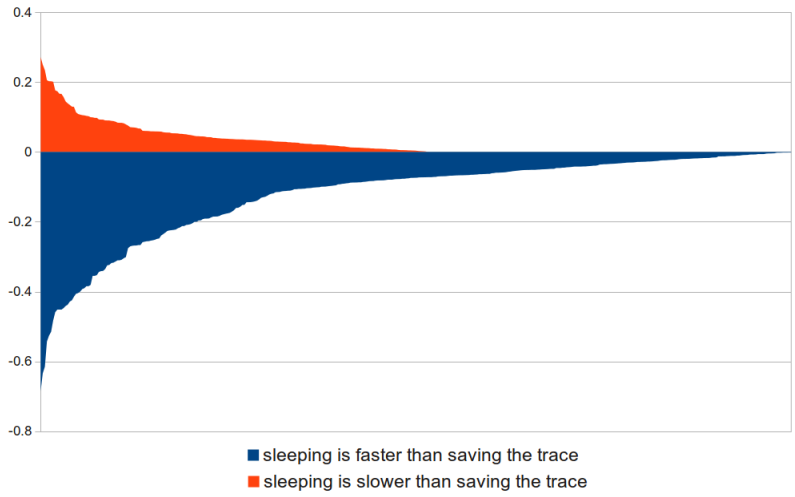

Figure 5.1: Difference of overhead(%): sleeping - saving

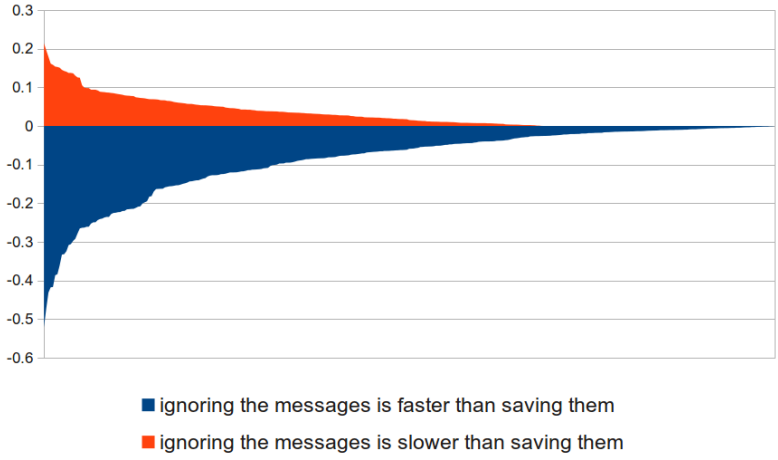

Figure 5.2: Difference of overhead(%): ignoring - saving

5.1.2 Overhead during profiling

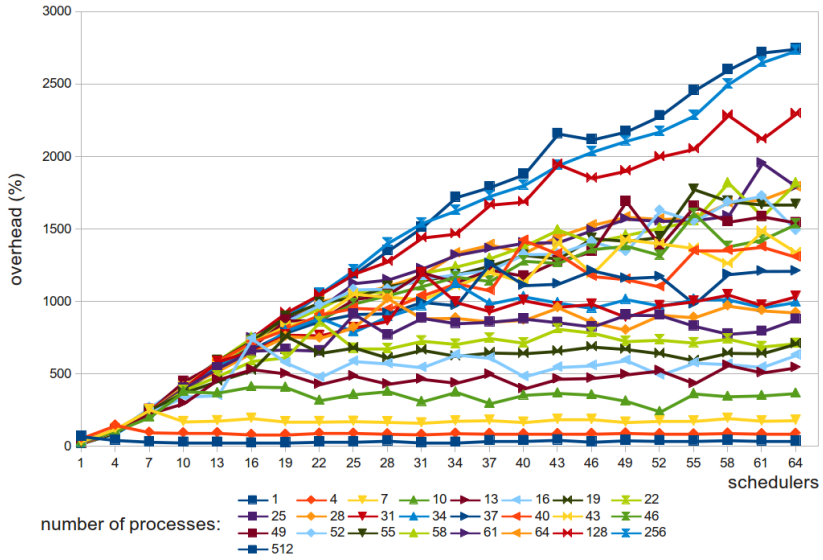

Figure 5.3: Overhead (%) of profiling

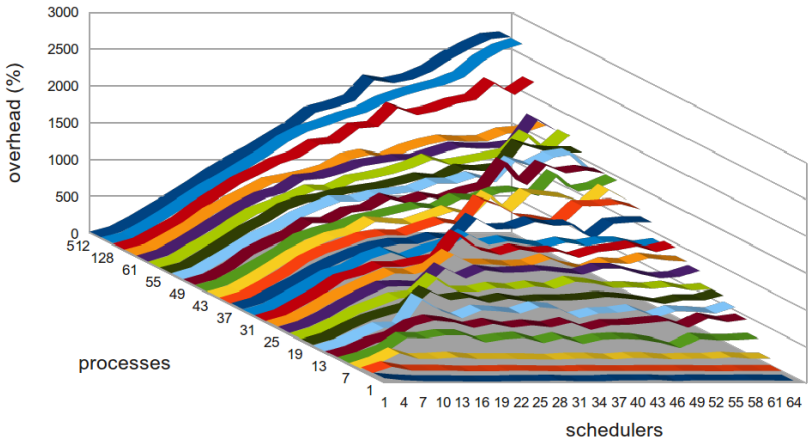

Figure 5.4: Overhead (%) of profiling (3D)

As we observe in the figures 5.3, 5.4 the overhead increases linearly as the number of cores increases and maximizes when the number of active processes is equal with the number of schedulers used; then it remains almost the same. We can observe a noticeable peak when the number of active processes is equal with the number of schedulers used; this is expectable since at this point the program achieves its maximum speedup and therefore, the overhead due to profiling is most noticeable.

5.1.3 Additional overhead cause by storing the trace

Figure 5.5 displays the extra time required to finish storing the trace as a percentage of the time the profiled program was running.

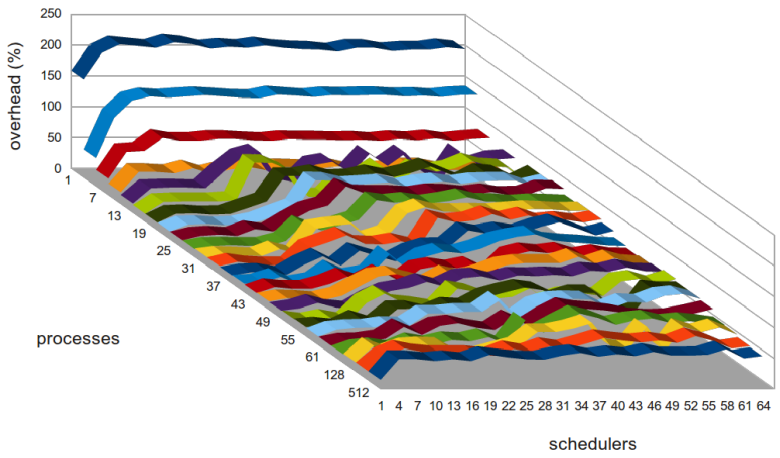

Figure 5.5: Addictional overhead (%)

The average overhead is 67% and has a maximum of 220% when a low number of processes are spawned and significant more schedulers are used; this is due to overhead caused by parallelizing the message encoding and storage. If we exclude these cases, the average overhead falls to 58% and the standard deviation reduces from 41% to 24%

5.1.4 Message queue length

Figure 5.6 displays the maximum length (in thousand of messages) of master tracer's message queue; the data were gathered by polling at regular intervals in order to avoid causing significant overhead, with `erlang:process_info/2` using the flag `message_queue_len`. A similar method was used to measure the length of the message queues of the tracers which are displayed in Figure 5.7; for simplicity, we only displayed the maximum message queue length of all the tracers and not for each tracer:

As expected, we observe from Figure 5.6 that the maximum length of master tracer's message queue increases as the number of schedulers increases until the number of schedulers reaches the number of active processes. We also observe a linear relationship between the number of active processes and the length of master tracer's message queue; this is particularly notable at the last 4 series with 64, 128, 256 and 512 processes. This relationship is better observed in Figures 5.8, 5.9 which display the average queue length correlated to the number of active processes.

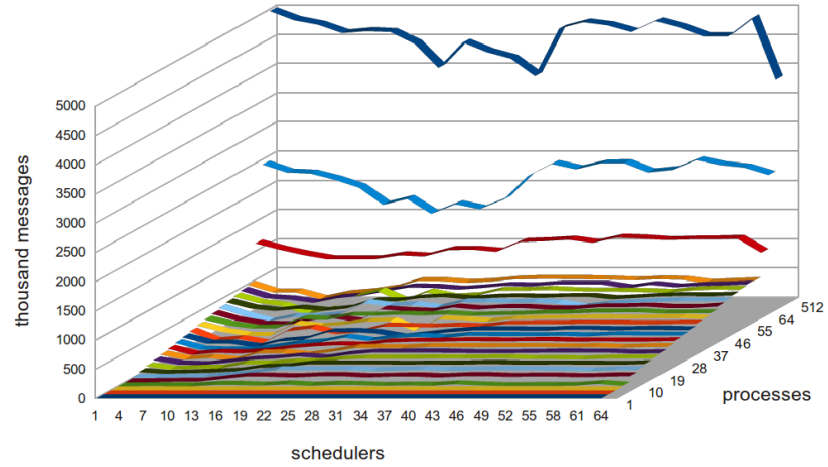

Figure 5.6: Message Queue Length: Master Tracer

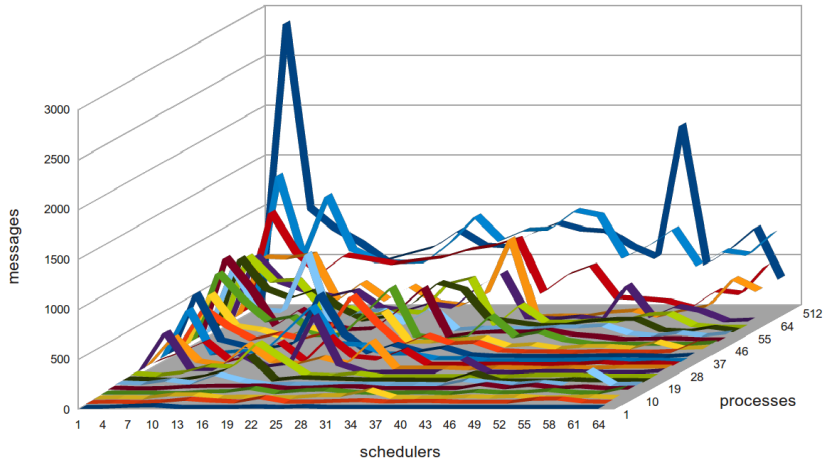

Figure 5.7: Message Queue Length: Tracers

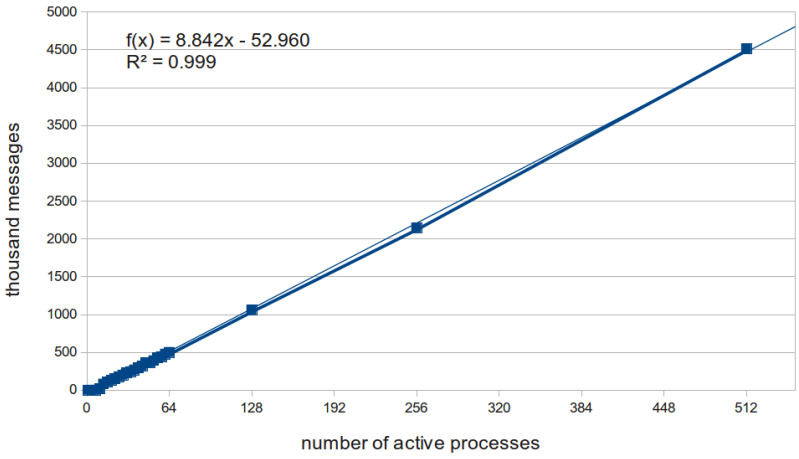

Figure 5.8: Active processes vs master tracer's message queue length

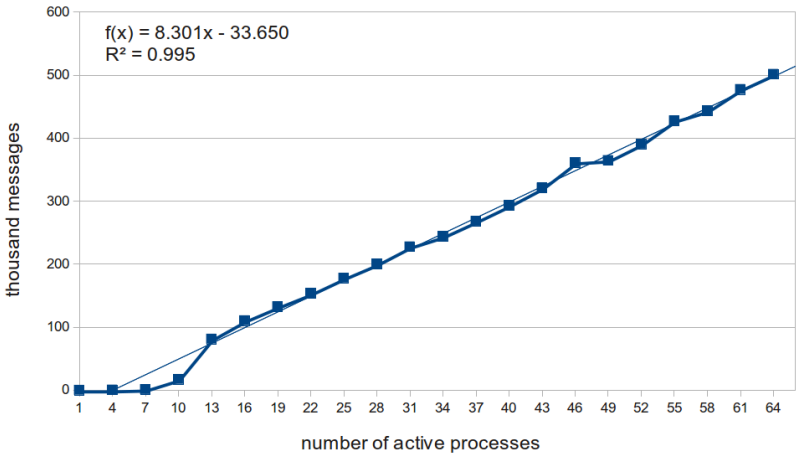

Figure 5.9: Active processes vs master tracer's message queue length (detail)

As we can see in Figure 5.9, when the number of processes is not above 10, master tracer is able to forward the trace messages to the traces; however, after this threshold, the rate of incoming messages increases resulting in larger message queue as the master tracer is forwarding them in a slower rate. This eventually limits the scalability of the profiling since the messages are not stored in the disk and therefore consume memory.

From Figure 5.7 we see that the length message queues of the tracers are a lot lower (1000 times lower) than the length of master tracer. We observe a general increase as the number of schedulers and processes increases but it is not as noticeable as the increase in master tracer. Indeed, the necessary forwarding of messages through the master tracer limits the rate of incoming messages and the fact that the tracers can encode and store the messages concurrently further reduces the problem. It is clear that the major bottleneck is caused by the restriction imposed by erlang:trace/3 which sends the messages to only one process resulting in an unscalable architecture. One might wonder if there is any real issue with having a large message queue of unread messages besides the memory cost; our main concern is that large message queues may significantly increase the overhead of sending messages due to reorganizations in the message queue's datastructure

5.2 Design choices

Special consideration was given in the way the data would be presented in a way that would convey the efficiency of the parallelization of the program; the information gathered was vast yet it should be presented in such a way that the human mind would be able not only to understand by also process it. Although we first considered displaying information about which process was running at any time, such as the PID or the MFA when the process entered or exited the scheduler, this thought was soon abandoned when we realized that in the highly concurrent programs, in which we are mainly interested in profiling, the processes' duration was very small (100us) and so, having a label for each time a process entered or left a scheduler or using different colours would be impossible even when profiling a program that lasts for less than one second. For the same reason it would also be impossible to have labels pop up when the user hovered over a part of the graph when a single pixel could be equivalent of a few milliseconds. It was decided to implement a functionality similar to printing messages during debuging: a function called by the user that would print messages on the graph, on the time of the call and on the scheduler in which the process run when the call was made. To find on which scheduler the process ran, `erlang:system_info/2` was used with the flag `scheduler_id`. The call was designed to have small overhead but it should be used sparingly; it is advised not to exceed 100,000 calls.

5.3 Alternative representations

While deciding the type of the graph that would display the data, besides the classic, implemented graph, a few other options were considered such as a radar or spider chart but it was dismissed as counter-intuitive.

Another interesting possibility for the display of the data is the fisheye view visualization. Fisheye views magnify the objects in a focal point while reducing the size of objects farther

away, thus achieving views that show local detail and global context together in a single view. It is also possible to support multiple focal points[16]. The reason it was rejected, at least for now, is the unfamiliarity it might cause to the user as well as the debatable usability when the level of detail that should be provided is taken into consideration.

Chapter 6

Related Work

6.1 Related Erlang Tools

Erlang has truly powerful trace primitives: `erlang:trace/3`, `erlang:trace_pattern/2` and `erlang:system_profile/2`, which enable the programmer to extract information about almost everything, as well as a variety of modules that help the programmer examine various aspects of the program's performance. However, it was surprising to find the absence of tools focusing on visualizing the parallel performance besides percept; one would expect a plethora of such tools in a language pioneering concurrency and scalability.

Percept (Percept - Erlang Concurrency Profiling Tool) is a tool that utilizes trace informations and profiler events to form a picture of the processes' and ports' runnability, showing the potential speedup of a program[5]. It uses `erlang:trace/3` and `erlang:system_profile/2` to monitor events from process states such as waiting, running, runnable, free and exiting; a process in running or runnable state is considered active while it is inactive in any of the other states. Then percept produces a time-graph, where the value of the y-axis is the number of active processes; this allows an overview of the concurrency of the erlang system, with peaks representing high concurrency and dips low concurrency (Figure 6.1). The user may also choose and inspect the activity of a process or a set of processes as well as more detailed statistical information about them such as spawn and exit times, entry points and start arguments etc.

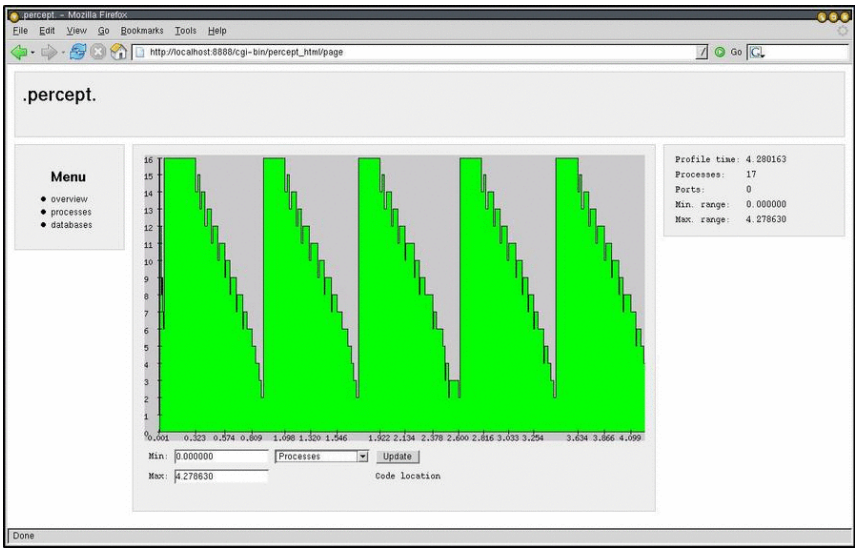

Figure 6.1: percept execution overview

6.2 Related Tools for Other Languages

An intuitive way to inspect a program's parallel activity is to create a graph of the activity of the schedulers/cores during its execution. Indeed, there are implementations of this tool for several languages but not for Erlang; while percept provides a similar view of the program's performance, which is useful, it does not quite visualize the actual execution of the program. The aforementioned tools are:

- Threadscope – Haskell
- Eden Trace Viewer – Eden (a Haskell extension focusing on parallelism)
- Thread Scheduling Visualizer – Java
- Threads View – C++AMP

The above programs have similar functionality, adjusted, of course, to fit the language's structure; we will focus on Threadscope since Haskell is the language that is closest to Erlang.

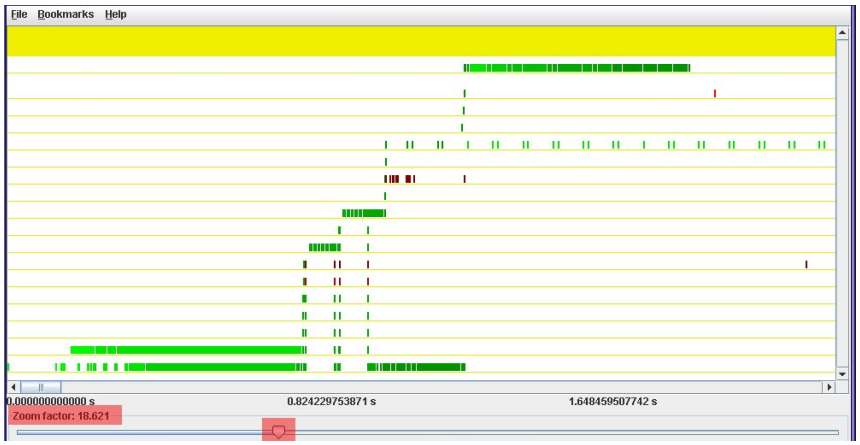

Figure 6.2: Thread Scheduling Visualizer graph (java)

6.2.1 A few words about Haskell's concurrency model

Haskell provides a mechanism to allow the user to indicate calculations that may be useful to executed in parallel by using functions from the Control.Parallel module[10]:

```
par :: a → b → b

pseq :: a → b → b
```

The function par indicates to the Haskell run-time system that it may be beneficial to evaluate the first argument in parallel with the second argument. The par function returns as its result the value of the second argument; the semantics of the program remain unaltered if we replace `par a b` with `b`.

It is important to note that the Haskell runtime system does not necessarily create a thread to compute the value of the expression a; instead, it creates a spark which has the potential to be executed on a different thread from the parent thread. It could be said therefore that the equivalent in Erlang would be spawning a process to evaluate a. Haskell uses Haskell Execution Contexts (HEC) which roughly corresponds to an operating system thread (and would be the equivalent of Erlang schedulers)[4].

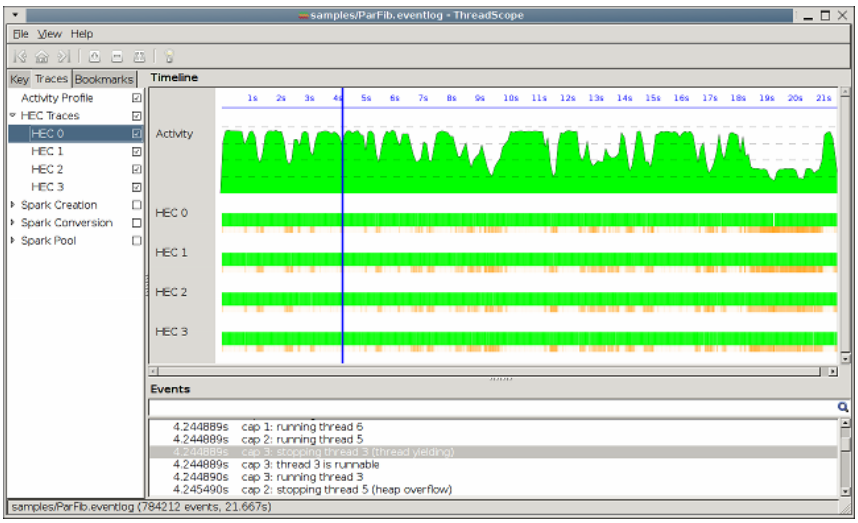

Figure 6.3: Threadscope display

Threadscope displays the activity on each HEC; for each thread the user can see whether it is running a Haskell thread or performing garbage collection; a green vertical line of standard length indicates that the HEC was running a Haskell thread on that timeframe while an orange line that it was performing garbage collection[9]. There is also information about when Haskell threads are ready to run and information about why a Haskell thread was suspended as well as a summarizing activity graph which resembles Percept's functionality. There is also information about function calls and spark generation and usage and recently there was implemented a source code mapping feature that allows the user to inspect the code that was executed at a certain moment (Figure 6.4)[21].

All in all, Threadscope is an exceptional tool that would be really useful to have implemented for Erlang; while this thesis aspires to create a profiling tool implementing a subset of its functionality there are still much to be done.

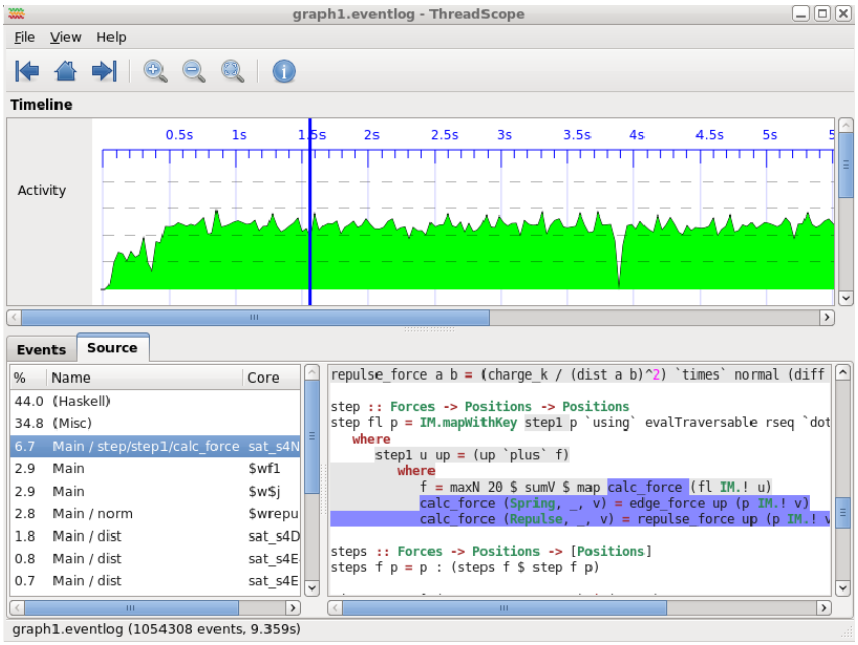

Figure 6.4: Code mapping in Threadscope

Chapter 7

Future Work

There are several ways to improve Schedplot, either functionality- or performance-wise.

7.1 Reducing and controlling overhead

Alas, there is only so much one can do to reduce the overhead inside the virtual machine; on the other hand, deciding to alter it may massively reduce the overhead. One of the major issues currently is the bottleneck imposed by having to use a master tracer to redirect the messages sent by `erlang:trace/3`. If instead we could just send the messages to the correct scheduler this bottleneck would be resolved. Furthermore, we would not need to forward the messages for a second time. Another improvement could be done by encoding and storing the messages directly to the disk instead of sending them to process first

Another possible improvement is to control the overhead; if we know when and where the profiler runs we can simply ignore it. The optimal way to do that is to pin the profiler processes to one (or more) scheduler (and make sure that no traced process uses it) and ignore that scheduler later. This could be done either just for the profiling process that runs inside the VM or for even the VM processes that generate the trace messages. Currently though it is not possible to pin a process to a scheduler and this unlikely to change since it is intentional. Note however that if the number of schedulers devoded to profiling is smaller than the number of schedulers that are used by the program, a scalability issue is created since the messages of multiple schedulers will have to be encoded and sorted by processes running in one scheduler.

7.2 Automatic Data Processing

At the moment the produced graph relies on the data collected only from one run of the profiled test. However, especially since wall clock time is used, single runs of the program may result in misleading graphs. Furthermore, it could be of great use to suitably combine results from runs with different input sizes of the programs or number of schedulers.

7.2.1 Combination of multiple runs

The first improvement is simple: profile the program multiple times and then display the average activity of the schedulers over time. By doing this we hope to eliminate or at least reduce the impact of the profiler: it will be indeed unlikely that in every execution the profiler worked at the exact same moment causing the user to believe that that moment the scheduler was inactive. Consider the following example:

Lets assume that in a program's activity in a scheduler is:

```
[4, 4, 4, 4, 0, 0, 0, 4]
```

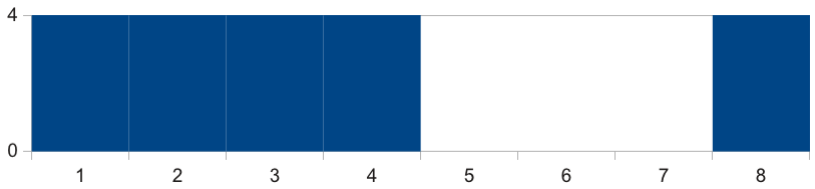

Figure 7.1: Ideal representation

With 100% overhead, there will be added 8 more zeros since the scheduler is running the profiler at those moments. This could be done in several ways (the following combinations were created by randomly inserting 4 zeros in the list).

```
1 [4, 4, 0, 4, 4, 0, 0, 0, 0, 0, 0, 0, 0, 0, 4, 0]
2 [4, 0, 0, 0, 4, 4, 4, 0, 0, 0, 0, 0, 0, 0, 0, 4]
3 [4, 4, 4, 0, 4, 0, 0, 0, 0, 0, 0, 0, 0, 4, 0, 0]
4 [4, 0, 4, 4, 0, 0, 0, 4, 0, 0, 0, 0, 0, 0, 4, 0]
5 [4, 4, 4, 0, 0, 0, 4, 0, 0, 0, 0, 0, 0, 0, 4, 0]
```

By calculating the average and zoom-out the values will be:

```
[4, 3, 2, 2, 0, 0, 1, 2]
```

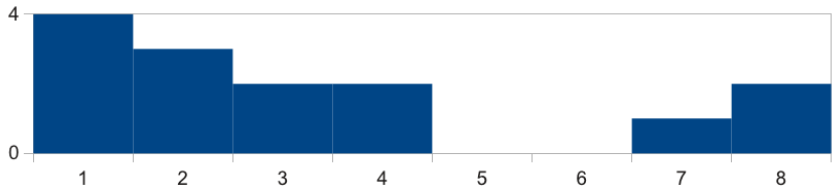

Figure 7.2: Combined representations

which resembles the actual execution of the program more than most individual runs (displayed in Figure 7.3)

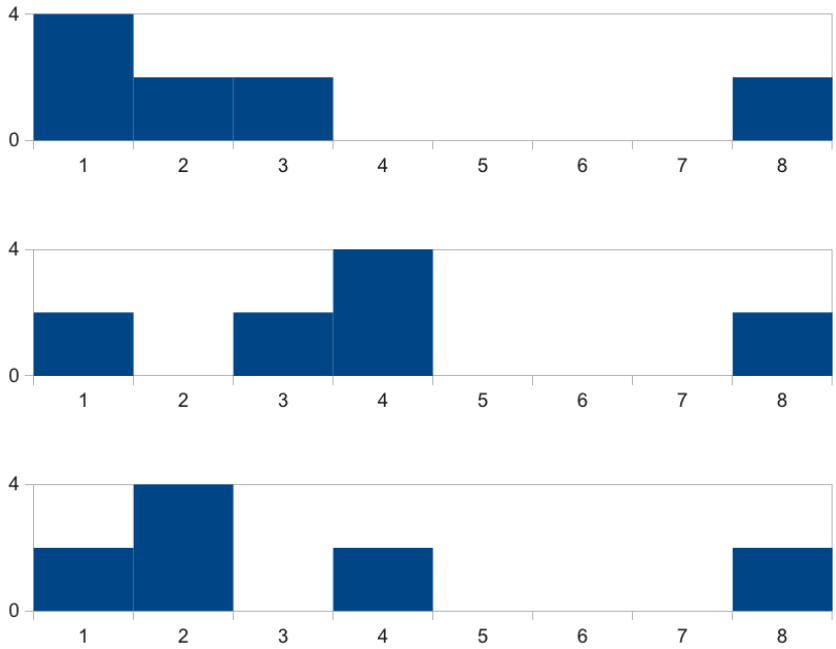

Figure 7.3: Individual representations

7.2.2 Pattern Detection

The second improvement might require more complex techniques; a few ideas that come to mind is analyzing the execution trace to detect similar patterns (such as a set of functions that are always called together, followed by another set of functions that are called later) and thus generating a mapping function between different executions of the same program. For example:

Assume that we want to trace `foo/1`. Lets assume that the execution of `foo(X)` is the following (2 schedulers; `000` denotes innactivity):

```
1 [foo, bar, foo, 000, 000, baz, baz, foo]
2 [000, 000, bar, bar, bar, foo, 000, 000]
3
4 [foo, foo, bar, bar, foo, 000, 000, 000, baz, baz, baz, foo]
5 [000, 000, 000, bar, bar, bar, bar, bar, foo, 000, 000, 000]
```

The pattern we would like to be noticed is that first foo is called, then we have subsequent, parallelized calls of bar and then a serial part of baz calls that starts only after every core has finished the bar calls. We would like to have pointed out that not all schedulers finish the bar calls at the same time and also that the gap increases when the input increases. Of course this could be a coincidence; a metric of the chance that this assumption is correct should be calculated based on factors such as the times this pattern occurs. With this information we could colour the graph; areas where there is a high chance of bottleneck existence should be highlighted while areas with a low chance should be faded.

Bibliography

[1] J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

[2] J. Armstrong, R. Virding, C. Wikstrom, and M. Williams. *Concurrent Programming in Erlang*. Prentice-Hall, 1996.

[3] N. Butler. Superlinear: an investigation into concurrent speedup, Octomber 2009.

[4] D. Coutts. Spark Visualization in ThreadScope. 2011.

[5] B. E. Dahlberg. Profiling applications in Erlang Symmetrical MultiProcessing system. Master's thesis, Uppsala University, 2007.

[6] S. Few. Practical Rules for Using Color in Charts. *Visual Business Intelligence Newsletter*, 2008.

[7] M. Heath and J. Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, 8(5), September 1991.

[8] P. Hedqvist. A parallel and multithreaded ERLANG implementation. Master's thesis.

[9] D. Jones, S. Marlow, and S. Singh. Parallel Performance Tuning for Haskell. ACM, September 2009.

[10] S. P. Jones and S. Singh. A tutorial on Parallel and Concurrent Programming in Haskell. *Lecture Notes in Computer Science*, May 2008.

[11] A. Karp and H. Flatt. Measuring parallel processor performance. *Communications of the ACM*, 33(5), May 1990.

[12] S. W. Kim and R. Eigenmann. Where Does the Speedup Go: Quantitative Modeling of Performance Losses in Shared-Memory Programs. *Parallel Processing Letters*, 10(2,3), 2000.

[13] K. Lundin. Inside the Erlang VM, focusing on SMP. November 2008. `http://www.erlang.se/euc/08/euc_smp.pdf`.

[14] K. Nevelsteen. Analyzing Performance of Multicore Applications in Erlang. Master's thesis, Uppsala University, 2011.

[15] M. O. Persson. wxErlang - a GUI library for Erlang. Master's thesis, University of Gothenburg, November 2005.

[16] T. Reinhard, S. Meier, and M. Glinz. An Improved Fisheye Zoom Algorithm for Visualizing and Editing Hierarchical Models. *REV '07 Proceedings of the Second International Workshop on Requirements Engineering Visualization*, 2007.

[17] T. Saad. Paralllel Computing with MPI: Part VII: Measuring Parallel Performance. `http://pleasemakeanote.blogspot.gr/2008/07/parallel-computing-with-mpi-part-vii.html`.

[18] J. Smart, K. Hock, and S. Csomor. *Cross-Platform GUI Programming with wxWidgets*. Prentice-Hall, 2005.

[19] H. Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software.

[20] Y. Tsavliri. Parallelizing Dialyzer: a Static Analyzer that Detects Bugs in Erlang Programs. Master's thesis, National Technical University of Athens, 2010.

[21] P. Wortman. Weaving Source Code into ThreadScope. 2011.