CHALMERS

Exploring the Elixir Ecosystem
Testing, Benchmarking and Profiling

Degree project report in Computer Engineering

SEBASTHIAN KARLSSON

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden 2015

DEGREE PROJECT REPORT

Exploring the Elixir Ecosystem

Testing, Benchmarking and Profiling

SEBASTHIAN KARLSSON

Department of Computer Science and Engineering
Division of Computer Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY

Göteborg, Sweden 2015

Exploring the Elixir Ecosystem
Testing, Benchmarking and Profiling
SEBASTHIAN KARLSSON

Examiner: Lars Svensson

Department of Computer Science and Engineering
Division of Computer Engineering
Chalmers University of Technology
SE-412 96 Göteborg
Sweden
Telephone: +46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden 2015

Exploring the Elixir Ecosystem

Testing, Benchmarking and Profiling

SEBASTHIAN KARLSSON
Department of Computer Science and Engineering, Chalmers University of Technology

Degree project report

Abstract

The success of a new programming language is to a large extent determined by the quality of its ecosystem. Among the most important parts of any given ecosystem are tools for testing, benchmarking and profiling code. This paper explores the tools available for Elixir, a new language targeting the Erlang virtual machine. A hands-on approach is used, by using these tools to develop and incrementally improve a fast Fourier transform algorithm. The final goal is a fast Fourier transform implementation fast enough for a real-time vocoder. Between the first and the last iteration a ten-fold increase in speed is achieved. While it is still not fast enough for a real-time vocoder, it demonstrates the utility of the development tools available in the Elixir ecosystem.

Keywords: Elixir, Erlang, Testing, Benchmarking, Profiling, FFT

Preface

Working on this degree project has been an interesting journey. It has been rough at times, but also very rewarding. I still need to work on my technical writing skills, but I think this report turned out acceptable in the end.

Hopefully, Elixir (along with Erlang) will continue to grow and gain popularity.

Anyway, acknowledgments. Thanks to Lars Svensson, for his straightforward criticism and insightful comments, and also for asking questions that caught me completely off guard. It was a humbling experience. Thanks also to my advisor, Roger Johansson, for letting me do my own thing. Finally, many thanks to Sakib Sistek, who provided the inspiration necessary for starting this project.

Abbreviations

BEAM Bogdan/Björn's Erlang Abstract Machine. The main implementation of the Erlang virtual machine.

BIF Built-In Function

CPU Central Processing Unit

DFT Discrete Fourier Transform

DSL Domain-Specific Language

FFT Fast Fourier Transform

FP Functional Programming

HiPE High Performance Erlang. Native code compiler for Erlang.

OOP Object-oriented Programming

PRNG Pseudo-Random Number Generator

TCP Transport Control Protocol

ULP Unit of Least Precision / Unit in the Last Place

Contents

1 Introduction

1.1 Background

"A poor craftsman blames his tools", as the traditional saying goes. While this is often taken to mean that you should focus on your own ability and not the tools you use, I prefer a different interpretation: Good craftsmen choose good tools. The tools we use do make a difference in the quality of our work. To steal a quote from Jeremy Bowers: "Da Vinci with a mop and a bucket of mud may be a better painter than you, but he would never beat Da Vinci with quality tools" [1].

For us programmers, the languages we use can be seen as our primary tools. Being able to choose the right language for a certain task is an important skill. Programming languages are however much more than just their syntax and semantics. When evaluating a language, we must also look at the whole ecosystem, the tools and the libraries available. There are many different ways to go about this, but a hands-on approach is often the best way to learn about a new language and its toolset.

In this paper we will take a look at Elixir [2], a new programming language targeting the Erlang virtual machine (BEAM). We will use the example of a fast Fourier transform (FFT) algorithm to explore the tools available for testing, benchmarking and profiling Elixir code, as well as different techniques for improving performance. To get a concrete goal to work towards, our target will be a FFT that is fast enough for a real-time vocoder.

1.2 Goals

1. Explore the tools available for testing, benchmarking and profiling Elixir code
2. Explore available techniques for improving performance of Elixir code
3. Use the above tools and techniques to determine if a FFT implementation fast enough for a real-time vocoder can be written in Elixir
4. If possible, use the FFT implementation from step 3 to create a simple real-time vocoder

Since Elixir, like Erlang, has a reputation for being slow at "number-crunching", using a FFT algorithm as the example for this exploration might seem a bit odd. This choice does however have several advantages. The algorithm is simple enough that testing, benchmarking, and profiling will not take unreasonably long time, while still being complex enough that the use of these tools to improve an implementation is warranted. There are also several known good FFT implementations in various languages that can serve as references and aid in testing. Finally, measuring and comparing the performance of different implementations is easy: a faster implementation is better. All these together give ample opportunities to test the different tools available for software development in Elixir. To trot out another old saying: "The journey is more important than the destination".

1.3 Restrictions

Elixir's support for distributed programming will not be considered. Splitting a FFT calculation across multiple machines may be useful when working with huge datasets, but the latency introduced will most likely be too much for a real-time vocoder. Using a natively compiled language such as C to implements parts of the algorithm will not be considered either, since the goal is to explore tools for working with Elixir. Finally, while the focus will be on tool written in Elixir for Elixir, some tools might still be under development or missing completely due to the young age of the language. Because of this, we will have to use some Erlang tools as well.

2 Technical Background

2.1 Functional Programming

In contrast with object-oriented programming (OOP), functional programming (FP)—as the name implies—emphasizes functions over data. The basis for OOP is to define specialized data structures, classes, for the problem at hand and then tie functions to those structures in the form of methods. FP prefers more generic data structures and smarter functions that can operate on those structures. This along with the ability to treat functions as data and use them as input to other functions give a different kind of flexibility as compared to OOP. Either paradigm can be used to emulate the other, making them basically equivalent. Certain problems, however, might be easier to reason about and solve by using one or the other.

"Traditional" OOP languages still rank high in various programming language popularity indices [3] [4] [5]. This might change however, if the current trend towards functional programming continues. With Clojure [6] and Scala [7] on the JVM and F♯ [8] on the .NET platform, interest in FP is definitely on the rise. Through the addition of lambdas—anonymous functions—to both Java [9] and C++ [10], FP concepts are seeping into OOP languages as well. Hybrid or multi-paradigm languages seem to be the next stage in the evolution of programming languages.

2.1.1 Important Concepts

All concepts mentioned below can be found in OOP as well, but they are generally more prominent in FP languages.

Higher Order Function

A function that take other functions as input. An example is `map` that takes a list and a function as input, applies the function to each item in the list, and returns a list of the results. This is equivalent to an imperative for-loop that steps through each item in a list and applies a transformation.

Lambda

An anonymous function, often created on the spot as input to a higher order function. A convenient way to pass code around as data.

Recursion

Often the only way to perform iteration, or looping, in a functional language. Many language implementations provide tail call optimization, or tail recursion, which enables an iterative process to be written recursively while still only requiring constant memory [11]. This also allows endless loops to be written in a recursive fashion. Some languages, for example Scheme [12], mandate tail call optimization for all implementations.

Immutability

Once a value has been created, it cannot be changed by any means. The member data of OOP objects is often mutable, which means that querying the same object for its state at different times can give different answers. An immutable value on the other hand always guarantees the same answer. To "change" an immutable value, a copy must be made and the change performed as a transformation during the copy operation. Immutable data solves some of the problems related to concurrent programming, but it is not a silver bullet [13].

2.2 Erlang

The Erlang programming language has seen a relatively recent increase in popularity thanks to the likes of WhatsApp [14] and Klarna [15]. The language was developed at Ericsson in 1986 for use in telephone exchange systems. This specialized use case made certain properties highly desirable: concurrency, asynchronous message passing, fault tolerance, and live updating of running systems [16]. Erlang was designed with these properties in mind, which led to a language capable of creating highly reliable systems. The most notable such system is probably the AXD301 telephone switch, with its almost legendary availability of nine 9's (99.9999999% uptime) [17].

Erlang began its life as a proprietary language, but was released as open source in 1998. The same features that made it a good fit for telephone exchanges also turned out to be useful for other massively concurrent systems. The WhatsApp messaging system, written in Erlang, can handle over two million concurrent TCP connections [14].

2.2.1 Language Features

Erlang code is compiled to bytecode and runs on a virtual machine. The de facto standard implementation is called BEAM (Bogdan/Björn's Erlang Abstract Machine).

The sequential parts of Erlang are functional. All values and variables are immutable and recursion is the only loop construct. It is dynamically typed and garbage collected.

Erlang has language level support for spawning concurrent processes (threads) within the virtual machine. These processes have no shared memory and can only communicate by sending and receiving asynchronous messages. This communication works the same way for communication between networked Erlang nodes running on different computers, making distribution completely transparent. Processes are extremely lightweight and running upwards of 30000 concurrent processes is not uncommon [16].

Listing 2.1 shows two simple functions in Erlang.

2.3 Elixir

Elixir [2] is a dynamic, functional programming language created by José Valim. It targets the Erlang virtual machine (BEAM), which gives it the same properties of scalability, distribution and fault-tolerance that has made Erlang famous. The language is very young, the first commit

```erlang
-module(greeting).

-export([hello_world/0, hello/1]).

hello_world() ->
  io:format("Hello, World!~n").

hello(Name) ->
  io:format("Hello, ~s!~n", [Name]).
```

Listing 2.1: "Hello, World!" and a simple greeting function in Erlang.

in the Git repository is dated January 9th 2011 and version 1.0 was released as recently as September 18th 2014 [18]. Thanks to a solid foundation in form of the Erlang ecosystem and the addition of tools for building, testing, package management, and documentation, Elixir already feels very polished and developer friendly.

With Valim being a prolific Rubyist and one of the top contributors to Ruby on Rails [19], it is not unexpected that Elixir's syntax is heavily inspired by Ruby. Parenthesis on function calls can be omitted and blocks are delineated with `do...end`. Listing 2.2 shows the same functions as Listing 2.1, but this time in Elixir.

```elixir
defmodule Greeting do
  def hello_world() do
    IO.puts "Hello, World!"
  end

  def hello(name) do
    IO.puts "Hello, #{name}!"
  end
end
```

Listing 2.2: "Hello, World!" and a simple greeting function in Elixir.

Elixir functions reside in modules. The Greeting module would generally be located in a file named `greeting.ex`, but the filename can be anything. The `hello/1` function showcases string interpolation, another feature that borrows its syntax from Ruby.

The general structure still stays close to Erlang, making it easy to move between the two languages. On top of that Elixir adds macros in the lisp sense, which allows for advanced metaprogramming, and protocols for polymorphism. Another smaller addition that nonetheless can make Elixir code look quite different from Erlang is the pipe operator `|>`. The expression `foo |> bar` means to evaluate `foo` and add the result from that as the first argument to the function call `bar`. This makes it easier to chain function calls in a way similar to object-oriented method chains.

Listing 2.3 shows two functions that calculate the sum of the squares of all odd numbers in a list. While the pipe-less version could be improved by a few well-placed line breaks, it still must be read from the inside out. The version using pipes on the other hand has the operations in the same order they are performed, with the subject at the very top of the pipeline. This works well since all functions in the standard library take the subject as the first parameter.

```elixir
defmodule PipeExample do
  require Integer

  def odd_square_sum(list) do
    Enum.reduce(Enum.map(Enum.filter(list, &Integer.is_odd/1), &(&1 * &1)), &+/2)
  end

  def odd_square_sum_with_pipe(list) do
    list
    |> Enum.filter(&Integer.is_odd/1)
    |> Enum.map(&(&1 * &1))
    |> Enum.reduce(&+/2)
  end
end
```

Listing 2.3: Cleaner code with the Elixir pipe operator.

The `Enum` module functions all take an enumerable collection first, the `String` functions all take a string and so on. This allows for the creation of a hidden extra variable, which can then be passed in to the next call in a pipeline. In the words of Joe Armstrong: "It's kind-of what a monad does in Haskell, but keep this a secret" [20].

2.3.1 Mix and Hex

Mix [21] is Elixir's default build tool. It provides tasks for creating, compiling and testing Elixir projects, as well as dependency management. Project details are specified in what is called a mixfile, an Elixir script file named `mix.exs` in the project base directory. Mix is controlled through a simple command line interface, with detailed instructions available by invoking `mix help`.

For dependency management, Mix offers integrated support for Hex, "a package manager for the Erlang ecosystem" [22]. Hex dependencies can be specified using name and version number only. Other dependency sources are git repositories, with a shorthand notation for Github, and local paths.

2.3.2 ExUnit

Elixir provides ExUnit [23] for basic unit testing. Tests are written using a simple DSL which provides setup, teardown and assert functionality. Mix integration makes it possible to run all tests for a Mix project using the command `mix test`, and a list of test files to run can also be specified. When a single test file is specified, a line number can also be specified to only run one particular test. Functionality for measuring test coverage is also included, by default a thin wrapper around Erlang's `cover` module.

2.3.3 Concurrency

Like Erlang, Elixir provides concurrency through processes and message passing. The default position of no shared state between processes removes a whole class of problems traditionally

associated with concurrent computing. No shared state means no need for explicit locking through semaphores or other means. Unfortunately, problems such as deadlock and livelock still persists.

Message passing also lends itself well to an asynchronous programming style. This does however mean that synchronous communication is harder, necessitating the creation of custom message protocols.

2.3.4 HiPE Native Code Compiler

The High Performance Erlang (HiPE) native code compiler [24] [25] allows Erlang—and Elixir—to be compiled to native code instead of bytecode. The code is still run inside the BEAM, but in a different mode that allows direct execution of native code. HiPE was initially created by the High-Performance Erlang Project at Uppsala University as a standalone compiler. It has since become integrated in the default Erlang distribution.

While natively compiled Erlang does speed up code execution in most cases, it is not completely without drawbacks. Tracing, single-stepping and breakpoint functionality that is normally provided by the BEAM is not available in native compiled code. There is also a small overhead when jumping between bytecode and native code, so very frequent mode switches must be avoided. Finally, stack traces provide much less detail for natively compiled code, which makes debugging harder.

2.4 Discrete and Fast Fourier Transforms

The discrete Fourier transform (DFT) is defined as

$$X_k = \sum_{n=0}^{N-1} x_n e^{-i2\pi k \frac{n}{N}} \qquad k = 0, \cdots, N-1 \tag{2.1}$$

where $x_0, \cdots, x_{N-1}$ are complex numbers. Listing 2.4 shows a naive implementation in Elixir.

```
1  defmodule Naive do
2    import Complex, only: [j: 0]
3
4    use Complex.Operators
5
6    def dft([]), do: []
7    def dft(xs) do
8      big_n = Enum.count xs
9      xs = Enum.with_index xs
10
11     Enum.map 0..(big_n-1), fn(k) ->
12       Enum.reduce(xs, 0, fn({x, n}, acc) ->
13         acc + x * Complex.exp(-j * 2 * :math.pi * k * n / big_n)
14       end)
15     end
16   end
17 end
```

Listing 2.4: Naive DFT in Elixir

The DFT is frequently used in a large number of fields and disciplines, such as signal processing, data compression, spectroscopy and quantum computing [26]. Its usefulness was however limited by the quadratic runtime of the naive implementation until the invention of several fast Fourier transform (FFT) algorithms with $\mathcal{O}(n \log n)$ runtime complexity, the most well-known being the Cooley-Tukey algorithm [27]. FFTs made the use of discrete Fourier transforms feasible, opening up new possibilities across multiple fields. This has led to the FFT being described as "the most important numerical algorithm of our lifetime" [28].

2.5 Vocoder

A vocoder (voice encoder) [29] is a system originally developed in the 1930s to code speech for transmission in telecommunications applications. By splitting an audio signal into a number of frequency bands and recording the signal level in each band the amount of information required to store speech can be greatly reduced. This in turn leads to a reduction in the bandwidth needed for transmitting speech while also simplifying encryption, which was greatly appreciated by the military. In more modern times, the vocoder has seen extensive use in music, filmmaking and games. It is often used to produce an electronic or robotic voice, but other effects are also possible.

While the classical vocoder is an analog device, digital vocoders can be created solely in software. Digital vocoding can be performed on pre-recorded input or on a live input stream. A real-time vocoder can operate on input streams with minimal delay, which makes it usable for live performances.

A simple vocoder takes two inputs, the modulator and the carrier. The modulator is usually speech or song, while the carrier is a signal that is rich in overtones such as a sawtooth wave or similar. The most basic version uses FFTs to split the modulator into frequency bands just like its analog counterpart. The amplitude of each band can then be mapped onto the corresponding band of the carrier signal. Thereafter, the result is inverse transformed to put it back in the time domain. More advanced versions might add sibilance channels to better handle sibilants ('s', 'f', 'ch' and similar sounds), or allow modifications in the time domain as well as in the frequency domain, among other functionality. The Auto-Tune [30] pitch correction software is based on one variant, the phase vocoder, and the singing voice synthesizer Vocaloid [31] uses similar technology.

3 A Pragmatic Approach

There are certain areas where only the best software is acceptable. The importance of being able to trust medical technology 100% is something most people can agree upon. The space sector is another area where correctness is paramount, and mistakes can lead to great losses of both money and lives. Aiming for perfection in cases like these makes sense, even though the cost of doing so is quite high. The effects of the Pareto principle and the law of diminishing returns, when applied to software development, makes the search for perfection prohibitively expensive for most more mundane areas. The time and money needed for the best solution can not be justified. For most applications, a "good enough" implementation is, well, good enough.

In the case of a real-time vocoder, the ability to produce output faster does not matter after a certain threshold, since the throughput will be capped by the input rate anyway. This means that once we have a solution that can handle the input, with some margin, searching for the fastest possible solution is meaningless. No implementation can produce output faster than the input stream, since there simply will not be any input to vocode. The same holds true for many other applications. A web server only needs to keep up with the rate of incoming requests. A program that interacts with a user only needs to respond fast enough that the user does not notice any delay. The average human will not feel any noticeable difference between one and ten milliseconds. In general, IO-bound programs only have to keep up with the IO-speed, anything above that does not improve anything.

There might well be applications where that level of optimization is desirable anyway. For instance, a more efficient implementation that uses fewer cycles might also increase battery life for mobile devices. Reducing energy consumption is also interesting from an environmental perspective. With mobile devices being as ubiquitous as they are, any change that reduces their environmental footprint can be seen as a positive change. We will however not consider these aspects directly, but instead focus solely on performance.

This leaves two questions that must be answered:

- What are the criteria for a "good enough" application?
- Does the application fulfill those criteria?

The first question is usually not all too hard to answer. For the vocoder, it must be capable of keeping up with the input. There is a hard limit on how long vocoding an input chunk can take. The second question can be quite a bit harder, however. There are several different techniques that can be used to check different aspects of a solution. Testing is one way to check correctness, while benchmarking and profiling can be used to check and improve performance. The next section uses the example of the real-time vocoder to explain these techniques as applied to Elixir, using several of the tools available.

4 Development Tools

With the goal of creating a real-time vocoder we will first need a FFT to convert the input to the frequency domain. Is it possible to write a FFT implementation in Elixir that is "good enough"? Before we can answer that question, we must define "good enough" for this application.

A vocoder takes two input signals, the carrier and the modulator. Both signals are chunked into windows of a certain size, in the simplest case just by taking a set number of samples from each. To be able to handle input in real-time, each window must be fully vocoded in less time than the window time. If we assume a window size of 2048 samples—a common value—and a sample rate of $48000Hz$, the window time would be $\frac{2048}{48000}s \approx 42.67ms$. We must thus be able to Fourier transform the carrier and the modulator, split the results up into bands and map the modulator onto the carrier, and perform an inverse transform in less than approximately $40ms$. Assuming the mapping stage require as much time as a single transform, this gives a maximum time of $10ms$ for a transform of size 2048. Most vocoders use overlapping windows to improve audio quality, which would further reduce this time. It is also common to use a window function such as a Hamming window to chunk the inputs signals. This makes the calculations above a bit too simple, but we can still use them to estimate the absolute lower bound on the performance of an FFT implementation. If an implementation can achieve sub-millisecond times it is most likely good enough for a real-time vocoder. If the times hover just under $10ms$ it will probably not work.

We can now define "good enough" to be the ability to perform a FFT of size 2048 in $1ms$ or less. The next step is to gather the tools needed to check if our implementation is "good enough".

4.1 Testing

We must first find a way to convince ourselves that our implementation is correct. A lightning fast FFT that produces incorrect output is a waste of storage space and processor cycles. Although it would be good practice to employ several different testing methods, for the sake of brevity we will focus on one in particular.

The general idea is to compare the output of our Elixir FFT against the output from a know good FFT implementation, in this case the one from NumPy [32]. When implementing the above method, two problems appear. We need a way to move data between Elixir and Python, as well as a way to compare floating point numbers for equality correctly. Additionally, there is the question of how to generate test inputs, a problem common to all test methods. When testing a function that takes a single 32-bit int or float as input the answer is simple: Test all of them. There are only about four billion different inputs, which on a modern machine will not take very long at all. Unfortunately, this approach does not work for functions with multiple or unbounded parameters, for instance the FFT. In these more complicated cases, we must somehow find a representative input sample.

4.1.1 ErlPort

Erlang and Elixir has the concept of ports for calling out to code in other languages. The basic interface provided, while not too complicated, does require a bit of work to use. ErlPort [33] is a library that simplifies external calls, currently supporting Python and Ruby. It takes care of data type mapping and provides an interface for calling arbitrary functions in either direction. It is also possible to provide encoder and decoder functions for custom data types, which makes it easy to translate between Elixir structs and Python or Ruby classes.

4.1.2 Floating Point Equality

> "[Floating-point] math is hard." (Bruce Dawson [34])

Never compare two floats for equality. Most, if not all, programmers have stumbled upon this advice at some point in their life. The problem lies within the limited precision of floating point numbers and the rounding errors this can introduce. Bruce Dawson has written many great articles about floats and their quirks (and about other subjects as well). In one of them [34] he elaborates on the equality problem, and details the problems with standard methods such as epsilon comparison and ULP (Units in the Last Place) based comparison. His proposed solution is based on a combination of absolute epsilon and ULP comparison. Take note that while Dawson uses single precision floats for all examples, the method works just as well for double precision. This means that it works with Elixir floats, which are all double precision. An Elixir implementation can be found in Appendix A.

4.1.3 Random Input

Erlang provides the `random` module for generating pseudo-random numbers. The PRNG state can be handled in two different ways. If you do not mind implicitness, the module can store its state in the process dictionary and update it as a side-effect of every call. If you prefer the 'pure' functional way, the state can be sent in as an explicit argument and the updated state will be returned alongside the random number.

Audio data is almost always normalized to a stream of floating point numbers in the range $[-1, 1]$. We will assume that this is the case for the input to our FFT and generate our test data in this interval. Since we will only be working with real input, we can take a shortcut by not testing complex input.

4.1.4 Putting it All Together

The final procedure for testing the Elixir FFT implementation is as follows:

1. Generate a list of random real numbers in the range $[-1, 1]$
2. Apply Elixir FFT function on the list
3. Use ErlPort to apply the `numpy.fft.fft` [35] function on the list
4. Compare the results of step 2 and 3 using Bruce Dawson's float comparison method [34] with `epsilon = 1e-9` and `max_ulps = 10`

5. Repeat steps 1-4 for inputs of varying size, ten times for each size

The ten repetitions in step 5 was chosen more or less arbitrarily. A higher number means better testing overall, but also more time required for each test run. Ten times per input size gives a nice balance between the number of different inputs tested and the time it takes to run the tests.

The above is not enough to prove that the implementation is correct beyond all doubt. It will however give a decent amount of confidence in the implementation's correctness, which for this application is all we need.

4.2 Benchmarking

Many FFT benchmarks, such as the ones found on the FFTW benchFFT website [36], measure their results in "mflops". While this might be a measurement of actual flops, the results are often scaled in some way to make it easier to compare the performance of different size FFTs on the same graph. This might be convenient for comparing different FFT implementations, but it is not as useful when you want to know if an implementation is fast enough for a certain application. Actual wall-clock time is more useful for applications with real-time constraints, such as a real-time vocoder.

Two benchmarking tools that are useful for this type of measurements are Benchfella [37] and Bmark [38]. Both provide simple DSLs for defining benchmarks and functionality for comparing benchmark runs, and both integrate with Mix. Benchfella also provides a graphing functionality that can plot the results of multiple runs on an HTML page. Bmark on the other hand uses statistical hypothesis testing when comparing benchmark runs, making it easier to see if a result is statistically significant.

4.3 Profiling

When looking for code that can be improved, profiling is an invaluable tool. While benchmarking can put concrete numbers on the performance characteristics of a piece of code, it can not explain *why* the code is slow or fast. Code profiling is one way to find out more details.

Erlang provides built-in functions (BIF) for tracing through code, which Elixir also can take advantage of. There are several modules and applications built upon these BIFs that simplifies profiling of different aspects. Two of them that are particularly useful are ExProf and eflame.

4.3.1 ExProf

ExProf [39] is a wrapper for Erlang's eprof profiler. By using Erlang's built-in trace functionality it records the number of calls to each function, a well as the time consumed by each call. Listing 4.1 shows the abridged output from running ExProf on a non-optimized mixed radix FFT.

It is easy to see that more than a quarter of the running time is spent creating and multiplying complex numbers. One reason for this is that the twiddle factors are computed when needed,

```
FUNCTION                                      CALLS      %    TIME  [uS / CALLS]
--------                                      -----    ---    ----  [----------]
math:exp/1                                    45056   1.71    7047  [      0.16]
'Elixir.Enum':with_index/1                    22527   1.72    7084  [      0.31]
'Elixir.Complex':neg/1                        45056   1.74    7180  [      0.16]
'Elixir.Complex':exp/1                        45056   1.75    7190  [      0.16]
'Elixir.Complex.Operators':'/'/2              45056   1.95    8023  [      0.18]
'Elixir.Complex':'div'/2                      45056   1.95    8045  [      0.18]
'Elixir.Complex':add/2                        43008   2.00    8231  [      0.19]
'Elixir.Basic':omega/2                        40960   2.04    8391  [      0.20]
'Elixir.Complex.Operators':'-'/1              45056   2.06    8488  [      0.19]
lists:reverse/1                               44027   2.07    8524  [      0.19]
math:sin/1                                    45056   2.08    8561  [      0.19]
'Elixir.Basic':'-butterfly/2-fun-1-'/4        40960   2.13    8789  [      0.21]
'Elixir.Basic':zip_each/2                     73726   2.15    8863  [      0.12]
'Elixir.Complex':cis/1                        45056   2.24    9224  [      0.20]
math:cos/1                                    45056   2.40    9899  [      0.22]
'Elixir.Basic':'-zip/2-fun-0-'/2              73726   3.18   13100  [      0.18]
'Elixir.Enum':'-with_index/1-fun-0-'/2        63488   3.35   13790  [      0.22]
lists:mapfoldl/3                              98298   4.92   20243  [      0.21]
'Elixir.Complex.Operators':'*'/2             226303   9.55   39340  [      0.17]
'Elixir.Complex':new/2                       407552  12.68   52198  [      0.13]
'Elixir.Complex':mul/2                       271359  13.12   54013  [      0.20]
```

Listing 4.1: Sample ExProf output (abridged)

which results in a large amount of duplicate work. Reducing the number of calls to the complex functions is the most straightforward way to reduce the running time in this case.

4.3.2 eflame

Another common way to find performance bottlenecks is to look at sampled stack traces. Using these stack traces it is possible to identify hot code paths, the parts of the program where the most CPU time is spent. However, stack traces are often long and cumbersome to read and interpret. Flame graphs [40] are a way to visualize these stack traces, giving a better overview of the data. The name comes from the shape of the graphs, which resemble flames, and the fact that they show which code paths are hot. eflame [41] is an Erlang module for producing such graphs.

Figure 4.1 shows a flame graph for the same mixed radius FFT implementation as in the ExProf example above.

Each box represents a particular function on the stack. Functions are stacked on top of their ancestors, making the y-axis show the stack depth. The top-most box is the function that was on-CPU at the point of a certain sample. The x-axis spans the total sample population. The width of a box represents the total amount of samples it was on-CPU or part of the ancestry of a function that was on-CPU. The left-to-right ordering is not necessarily meaningful. eflame is capable of preserving the order of calls, while the reference implementation sorts the calls alphabetically. The colors of the graph are not significant either, but are usually randomly picked warm colors that make the graph even more flame-like. Finally, flame graphs are interactive. When opened in a browser, it is possible to zoom in on parts of a graph by

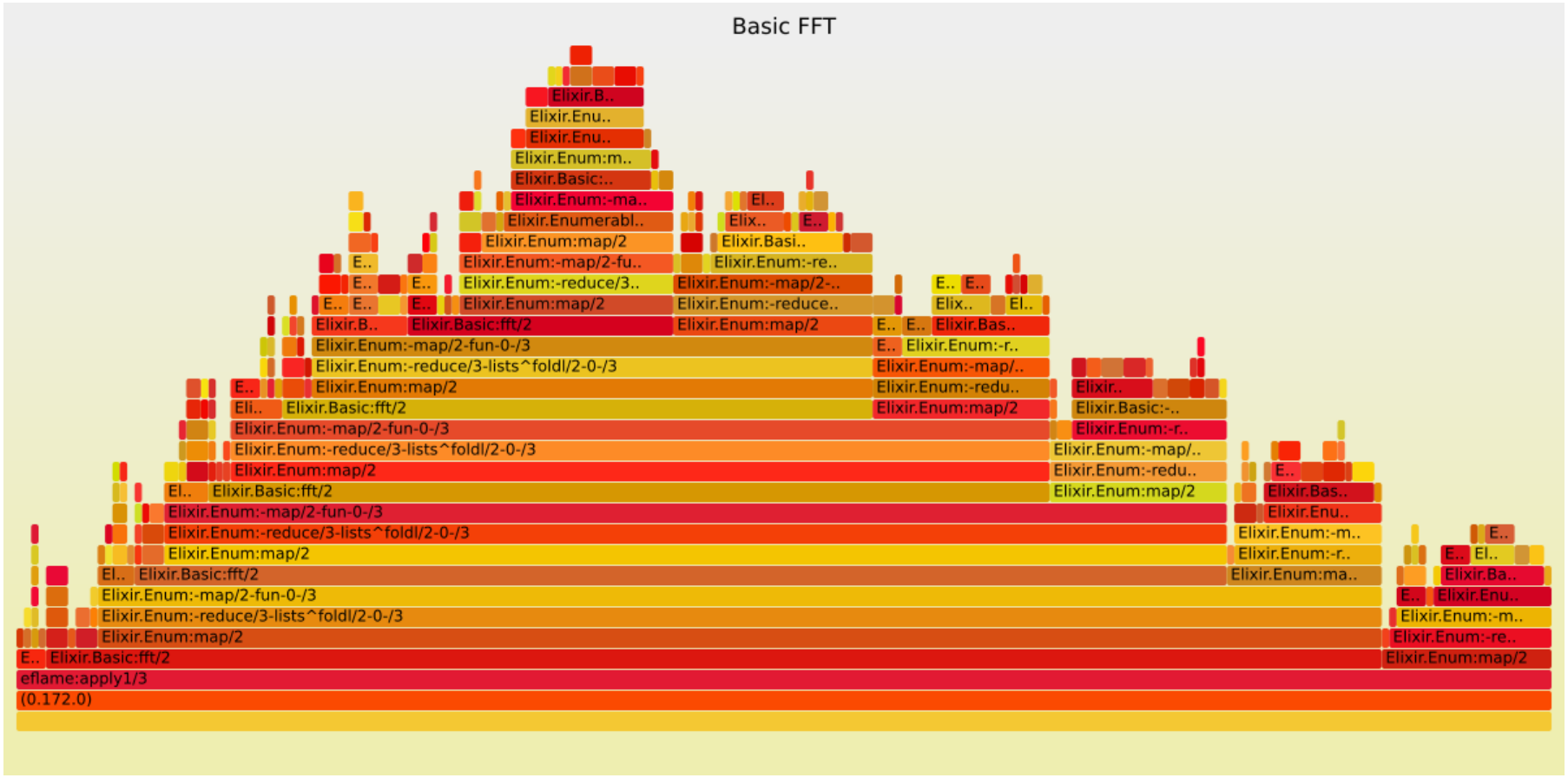

Figure 4.1: Sample flame graph generated by eflame.

clicking on one of the boxes, and mouseover reveals additional statistics.

The graph in figure 4.1 clearly shows the recursive structure of the FFT, with multiple calls to `Basic.fft/2` stacked on top of each other. The flat-ish peaks along the right side of the graph contain most of the calls to the functions operating on complex numbers. As expected, most time is spent performing complex operations.

5 Implementing a Fast Fourier Transform

5.1 Mixed-radix FFT

The algorithm we will use is the mixed-radix FFT, which can be seen as a generalization of the Cooley-Tukey algorithm [27]. Remember that the DFT is defined as

$$X_k = \sum_{n=0}^{N-1} x_n e^{-i2\pi k \frac{n}{N}} \qquad k = 0, \cdots, N-1 \tag{5.1}$$

Assuming the input size is a power of two, the Cooley-Tukey algorithm first rearranges this into two sums, one over the even indexes and one over the odd indexes:

$$X_k = \sum_{m=0}^{N/2-1} x_{2m} e^{-i2\pi k \frac{2m}{N}} + \sum_{m=0}^{N/2-1} x_{2m+1} e^{-i2\pi k \frac{2m+1}{N}} \tag{5.2}$$

From the sum over the odd indexes we can then factor out a common multiplier of $e^{\frac{-i2\pi k}{N}}$, commonly called a twiddle factor:

$$X_k = \sum_{m=0}^{N/2-1} x_{2m} e^{-i2\pi k \frac{m}{N/2}} + e^{\frac{-i2\pi k}{N}} \sum_{m=0}^{N/2-1} x_{2m+1} e^{-i2\pi k \frac{m}{N/2}} \tag{5.3}$$

We have now broken down the sum into two DFTs, one of the even-indexed inputs and one of the odd-indexed inputs. This decomposition, sometimes called the Danielson-Lanczos lemma, can then be performed recursively for the smaller DFTs. The lemma can be generalized to work for any composite input size $N = N_1 N_2$, and it is this generalized lemma that forms the basis of the mixed-radix FFT. The basic flow of the mixed-radix FFT is

1. Perform N_1 DFTs of size N_2
2. Multiply by twiddle factors
3. Perform N_2 DFTs of size N_1, often called butterflies

The name 'butterfly' in the last step comes from the case where $N_1 = 2$. The shape of a data-flow diagram for a DFT of size 2 resembles a butterfly, hence the name.

The mixed-radix FFT can handle inputs of any size, but it works best for sizes that are highly composite. Since prime-sized DFT cannot be broken down further using this method, base cases are usually computed using the naive $\mathcal{O}(n^2)$ method. Listing B.1 shows a rather inefficient implementation of this algorithm in Elixir.

5.2 Iterative Improvements

We can now begin improving the FFT implementation, using the tools from above. The first step is to test the current implementation. When we are confident that it is correct, we can benchmark the code to see how we are doing in terms of performance. We can thereafter

profile the code and look for areas we can improve. The process can then be repeated until we are happy with the performance or until we cannot improve it any further.

Listing 5.1 shows the benchmarking results for the first implementation and nine successive improvements. The first column is the name of the benchmark, the second is the number of runs and the last is the average time per run. The benchmark labeled 'basic' was the first working implementation.

```
## FFTBench
concurrent                100   11578.33 µs/op
all_butterfly             100   15642.31 µs/op
combined_complex_op       100   15691.89 µs/op
tuple_complex_native      100   16963.70 µs/op
tuple_complex              50   37938.70 µs/op
precompute                 50   42042.40 µs/op
array_data                 50   45845.20 µs/op
no_operators               50   69827.70 µs/op
memoized                   50   70610.68 µs/op
basic                      10   103750.60 µs/op
```

Listing 5.1: FFT benchmark results, output from Benchfella

All benchmarks were made using 2048 samples of a 440 hz sawtooth wave generated by Yoshimi [42] and captured through JACK [43]. The benchmarking machine was a Lenovo G570 laptop with an Intel Core i5-2430M 2.40GHz CPU. The steps taken to improve performance, working from the bottom of Listing 5.1 and up, were as follows:

- Memoize twiddle factors to greatly reduce the number of complex exponentiations needed
- Remove operator overloading[1]for complex numbers
- Use Erlang arrays [44] instead of Elixir Lists for all data
- Precompute twiddle factors to remove the overhead of memoization
- Use bare Tuples to represent complex numbers instead of Structs
- Compile to native code using HiPE
- Combine the complex multiplication and addition in the butterfly in a single function, reducing the amount of allocations of new complex numbers
- Use a DFT of size 1 as the base case instead of the naive DFT implementation used previously
- Spawn multiple processes for the top level of the recursive decomposition

While the benchmark times fluctuated a bit from run to run, the final version was always 9-10 times faster than the first version. Unfortunately, a time of around 11ms for a FFT of size 2048 is not fast enough for our real-time vocoder. For the sake of comparison, the corresponding time for Python's `NumPy.fft.fft`, written in well-optimized C, on the same machine is around 7μs.

[1]Elixir does not actually support operator overloading, but something similar can be achieved by unimporting `Kernel` operator functions such as `Kernel.+/2` and replacing them with your own. This is kind of hacky and should probably not be used in production code, but in this case it made the complex arithmetic look so much nicer.

The final code can be found in Listing B.2.

6 Conclusions

6.1 The Elixir Ecosystem

For any language, the quality of the tools is important, maybe even more so than the quality of the language itself. Despite being such a young language, Elixir already has many useful tools available. The advantage of having a build system included in the default distribution cannot be overstated, especially when the build system can be extended as easily as Mix. This creates a uniform way to perform common tasks such as testing and benchmarking, often as simple as executing `mix <task>`. Another benefit of Mix is that it dictates the general structure of Elixir projects. This allows developers moving between projects to concentrate on the code, making it easier to get up to speed on a new codebase. Of course, there are still some rough edges. For instance, it is hard to find information on how to treat projects with parts implemented in Erlang or native code. In any case, Valim's decision to include Mix from the very start can only be seen as a net positive.

For basic testing, ExUnit works well enough. Getting started is easy since Mix includes a skeleton test file with every new project, and running the tests is a simple `mix test`. Running the tests every so often to confirm that the latest changes did not break anything quickly became a habit.

Trying to improve the performance of FFTs in a language without mutable data types was interesting, to say the least. Most, if not all, openly available implementation make heavy use of mutable arrays, even in other languages that prefer immutability such as Haskell. The Erlang `array` module helped, since it is quite a bit faster for random access than lists. Even so, the execution time in the final version was pretty evenly split between complex arithmetic on one hand, and element access and insertion on the other.

Using HiPE to compile Elixir code was easier than expected. It did not however provide as much of a performance boost as I had hoped, just 2-3 times instead of the 10-20 that I have seen mentioned in several places. It is also a shame that code compiled with HiPE cannot be traced, making it hard to debug and profile.

On top of all this, having full access to the Erlang ecosystem is incredibly useful. If there is a tool or library missing in Elixir, it can probably be found in Erlang.

6.2 Optimization Techniques

Among the different optimization steps described in Section 5.2 two steps gave better results than expected. First, removing redundant computations through the memoization and later precomputation of the twiddle factors reduced the original runtime by more than 30%. In hindsight, it is clear that removing a large number of expensive floating point operations would result in a large speedup. Secondly, changing from lists to arrays cut the runtime by another 15%. While Erlang arrays don't have constant time lookup, they were still leaps and bounds faster than the $\mathcal{O}(n)$ lookup of linked lists. As a side effect of changing to arrays, the number and size of memory allocation for copies could also be reduced.

Two other optimization steps did not live up to expectations. Native compilation using HiPE, while providing a boost, was not nearly as effective as I had been led to believe. One reason for this seems to be that the way the BEAM handles floats does not translate well to native code. Finally, introducing concurrency in an efficient way proved to be quite difficult. The simple solution used—splitting the top level of the recursion into multiple processes—did work, but surely there are better ways to do it. The fact that the test machine only had two CPU cores did not help either.

To summarize, avoiding redundant calculations and using proper data structures are important, native compilation can help certain types of programs more than others, and while Elixir makes concurrency easier it can still be tricky to write good concurrent code.

6.3 Number-crunching in Elixir

Elixir's and Erlang's reputation for being slow at numerical computing seems to have a solid foundation. Seeing as this is not the primary purpose of the languages, this does not come as much of a surprise. An Erlang/Elixir expert could probably squeeze out a bit more performance, but Elixir would still not be a good fit for a real-time vocoder. When speed is needed, choosing a different language is the proper course of action.

Still, Elixir might be an interesting choice for applications that must handle huge datasets. When your input data does not fit in RAM, raw language performance is less relevant since IO will be the most likely bottleneck. Taking advantage of the distributed nature of Elixir to split the data across a network of nodes could give some interesting results. Using native code for the numerical computations and Elixir to direct the data flow would definitely be more in line with the strengths of the language. FFTs with input sizes on the scale of 10s of gigasamples or larger might benefit from this technique.

6.4 Planning vs. Reality

The original schedule for this project can be found in Appendix C. Everything went according to plan up until the Python FFT benchmarks in the fourth week of the project. The difference in benchmark times between Elixir and Python was too great. I knew that Elixir would be slower than an optimized C implementation, but I did not think it would be this slow. Investigating this took priority over benchmarking further implementations, I started the optimization work scheduled for weeks 6 and 7 immediately. This took about twice as much time as expected. Due to the fact that there are few highly optimized functional FFT implementations around, most available optimization techniques were not usable in Elixir. I had to delve deeper into both the intricacies of functional programming and the gory details of the BEAM to find out why my code was slow.

Once I had determined that an Elixir FFT would not be fast enough for the real-time vocoder, my focus shifted to this report. I had dedicated a large part of the three weeks before the presentation to writing, and I am glad that I did. Researching and documenting sources required more time than expected, but it was balanced out by the fact that I could not work

on the vocoder. If I could change anything, I would set off more time to have the drafts proof read by someone other than just myself.

References

[1] Jeremy "jerf" Bowers. *Hacker News Comment Thread*. URL: https://news.ycombinator.com/item?id=2379866 (visited on 2015-05-31).

[2] *Elixir*. URL: http://elixir-lang.org (visited on 2015-03-04).

[3] *The RedMonk Programming Language Rankings: January 2015*. URL: https://redmonk.com/sogrady/2015/01/14/language-rankings-1-15/ (visited on 2015-06-29).

[4] *Langpop.com — Programming Language Popularity*. URL: http://www.langpop.com/ (visited on 2015-06-29).

[5] *TIOBE Programming Community Index*. URL: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html (visited on 2015-06-29).

[6] *Clojure Home Page*. URL: http://clojure.org/ (visited on 2015-06-29).

[7] *Scala Home Page*. URL: http://www.scala-lang.org/ (visited on 2015-06-29).

[8] *F♯ Home Page*. URL: http://fsharp.org/ (visited on 2015-06-29).

[9] *What's New in JDK 8*. URL: http://www.oracle.com/technetwork/java/javase/8-whats-new-2157071.html (visited on 2015-06-29).

[10] *Lambda Functions (since C++11)*. URL: http://en.cppreference.com/w/cpp/language/lambda (visited on 2015-06-29).

[11] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. 2nd ed. MIT Press, 1996. Chap. 1.2.1. ISBN: 0-262-51087-1. URL: https://mitpress.mit.edu/sicp/full-text/book/book-Z-H-11.html#%_sec_1.2.1 (visited on 2015-07-10).

[12] Alex Shinn, John Cowan, and Arthur A. Gleckler, eds. *Revised⁷ Report on the Algorithmic Language Scheme ("R7RS")*. 2013-06-06. Chap. 3.5 Proper tail recursion, p. 11. URL: http://trac.sacrideo.us/wg/raw-attachment/wiki/WikiStart/r7rs.pdf (visited on 2015-07-14).

[13] Joe Duffy. *Thoughts on immutability and concurrency*. 2010-07-11. URL: http://joeduffyblog.com/2010/07/11/thoughts-on-immutability-and-concurrency/ (visited on 2015-07-10).

[14] *1 million is so 2011*. 2012-01-06. URL: https://blog.whatsapp.com/196/1-million-is-so-2011 (visited on 2015-07-05).

[15] *Klarna Engineering Insights*. 2015-01-09. URL: http://engineering.klarna.com/article/klarna-engineering-insights/ (visited on 2015-07-05).

[16] Bjarne Däcker. "Concurrent Functional Programming for Telecommunications: A Case Study of Technology Introduction." Master's Thesis. Royal Institute of Technology, 2000. Chap. 3-4, pp. 9–22. URL: http://www.erlang.se/publications/bjarnelic.pdf (visited on 2015-07-07).

[17] Joe Armstrong. *What's all this fuss about Erlang?* URL: https://pragprog.com/articles/erlang (visited on 2015-07-05).

[18] José Valim. *Elixir v1.0 released*. URL: http://elixir-lang.org/blog/2014/09/18/elixir-v1-0-0-released/ (visited on 2015-05-07).

[19] *Rails Contributors - #5 José Valim - All time*. URL: http://contributors.rubyonrails.org/contributors/jose-valim/commits (visited on 2015-05-26).

[20] Joe Armstrong. *A Week With Elixir*. URL: http://joearms.github.io/2013/05/31/a-week-with-elixir.html (visited on 2015-05-26).

[21] *Mix Documentation*. URL: http://elixir-lang.org/docs/stable/mix/ (visited on 2015-05-17).
[22] *Hex*. URL: https://hex.pm/ (visited on 2015-05-17).
[23] *ExUnit Documentation*. URL: http://elixir-lang.org/docs/stable/ex_unit/ (visited on 2015-05-19).
[24] *The High-Performance Erlang Project*. URL: https://www.it.uu.se/research/group/hipe/index.shtml (visited on 2015-05-17).
[25] K. Sagonas et al. "All you wanted to know about the HiPE compiler (but might have been afraid to ask)". In: *Proceedings of the PLI'03 Erlang Workshop*. 2003-08. URL: http://www.erlang.org/workshop/2003/paper/p36-sagonas.pdf (visited on 2015-05-17).
[26] Larry Hardesty. *Explained: The Discrete Fourier Transform*. 2009-11-25. URL: http://newsoffice.mit.edu/2009/explained-fourier (visited on 2015-02-12).
[27] James W. Cooley and John W. Tukey. "An Algorithm for the Machine Calculation of Complex Fourier Series". In: *Mathematics of Computation* 19 (1965), pp. 297–301. URL: http://dx.doi.org/10.2307/2003354.
[28] Gilbert Strang. "Wavelets". In: *American Scientist* 82.3 (1994), p. 253. URL: http://www.jstor.org/stable/29775194.
[29] Wikipedia. *Vocoder — Wikipedia, The Free Encyclopedia*. 2015. URL: http://en.wikipedia.org/w/index.php?title=Vocoder&oldid=658938750 (visited on 2015-05-26).
[30] *Auto-Tune 8*. URL: http://www.antarestech.com/products/detail.php?product=Auto-Tune_8_66 (visited on 2015-05-26).
[31] *Vocaloid Home Page*. URL: http://www.vocaloid.com/en/ (visited on 2015-05-26).
[32] *NumPy Home Page*. URL: http://www.numpy.org/ (visited on 2015-05-31).
[33] *ErlPort. Connect Erlang to other languages*. URL: http://erlport.org/ (visited on 2015-05-19).
[34] Bruce Dawson. *Comparing Floating Point Numbers, 2012 Edition*. 2012-02-25. URL: https://randomascii.wordpress.com/2012/02/25/comparing-floating-point-numbers-2012-edition/ (visited on 2015-05-19).
[35] *numpy.fft.fft — NumPy Manual*. URL: http://docs.scipy.org/doc/numpy/reference/generated/numpy.fft.fft.html (visited on 2015-03-24).
[36] *The benchFFT Home Page*. URL: http://www.fftw.org/benchfft/ (visited on 2015-05-21).
[37] Alexei Sholik. *Benchfella Github Repository*. URL: https://github.com/alco/benchfella (visited on 2015-05-28).
[38] Joseph Kain. *Bmark Github Repository*. URL: https://github.com/joekain/bmark (visited on 2015-05-28).
[39] parroty. *ExProf Github Repository*. URL: https://github.com/parroty/exprof (visited on 2015-05-11).
[40] Brendan D. Gregg. *CPU Flame Graphs*. URL: http://www.brendangregg.com/FlameGraphs/cpuflamegraphs.html (visited on 2015-05-13).
[41] Vladimir Kirillov. *eflame Github Repository*. URL: https://github.com/proger/eflame (visited on 2015-05-13).
[42] *Yoshimi Home Page*. URL: http://yoshimi.sourceforge.net/ (visited on 2015-05-21).
[43] *JACK Audio Connection Kit*. URL: http://jackaudio.org (visited on 2015-03-04).

[44] *Erlang array Module Documentation*. URL: `http://www.erlang.org/doc/man/array.html` (visited on 2015-05-28).

A Dawson's Float Equality Comparison

```elixir
@spec close_enough?(number, number, number, non_neg_integer) :: boolean
def close_enough?(a, b, epsilon, max_ulps) do
  a = :erlang.float a
  b = :erlang.float b

  cond do
    abs(a - b) <= epsilon -> true

    signbit(a) != signbit(b) -> false

    ulp_diff(a, b) <= max_ulps -> true

    true -> false
  end
end

#`signbit/1` is used instead of a regular `sign/1` function in order to
#properly handle -0.0 (negative zero). It shouldn't appear in ordinary
#code, but it *can* be constructed using Elixir binary syntax.
@spec signbit(float) :: boolean
def signbit(x) do
  case <<x :: float>> do
    <<1 :: 1, _ :: bitstring>> -> true
    _ -> false
  end
end

@spec ulp_diff(float, float) :: integer
defp ulp_diff(a, b), do: abs(as_int(a) - as_int(b))

@spec as_int(float) :: non_neg_integer
defp as_int(x) do
  <<int :: 64>> = <<x :: float>>
  int
end
```

Listing A.1: Equality comparison for floating point numbers, based on Bruce Dawson's method [34]

B FFT Implementations

```elixir
defmodule FFT do
  require Integer
  import Complex, only: [j: 0]

  use Complex.Operators

  def naive([]), do: []
  def naive(xs) do
    big_n = Enum.count xs
    xs = Enum.with_index xs

    Enum.map 0..(len-1), fn(k) ->
      Enum.reduce(xs, 0, fn({x, n}, acc) ->
        acc + x * Complex.exp(-j * 2 * :math.pi * k * n / big_n)
      end)
    end
  end

  def fft([]), do: []
  def fft([x]), do: [x]
  def fft(xs) do
    fft xs, ExMath.factors(Enum.count(xs))
  end

  defp fft(xs, [_last]), do: naive(xs)
  defp fft(xs, [f | fs]) do
    xs
    |> group_every(f)
    |> Enum.map(&fft(&1, fs))
    |> butterfly(f)
  end

  defp butterfly(rows, f) do
    big_n = Enum.count(rows) * Enum.count(hd(rows))

    rows
    |> Enum.map(fn(row) ->
         row |> List.duplicate(f) |> Enum.concat
    end)
    |> zip
    |> Enum.with_index
    |> Enum.map(fn({col, k}) ->
         col
         |> Enum.with_index
         |> Enum.reduce(0, fn({x, n}, acc) ->
              acc + omega(big_n, n*k) * x
         end)
    end)
  end

  #Twiddle factors
  defp omega(n, k), do: Complex.exp(-j * 2 * :math.pi * k / n)

```

```
54    defp group_every(collection, n) do
55      collection
56      |> Enum.chunk(n, n, [])
57      |> zip
58    end
59
60    #Multilist zip. Same as `List.zip/1` but without converting the resulting
61    #lists to tuples.
62    defp zip([]), do: []
63    defp zip(list_of_lists) when is_list(list_of_lists) do
64      zip(list_of_lists, [])
65    end
66
67    defp zip(list, acc) do
68      converter = fn(x, acc) -> zip_each(x, acc) end
69      {mlist, heads} = :lists.mapfoldl converter, [], list
70      case heads do
71        nil -> :lists.reverse acc
72        _ -> zip mlist, [:lists.reverse(heads)|acc]
73      end
74    end
75
76    defp zip_each(_, nil) do
77      {nil, nil}
78    end
79    defp zip_each([h|t], acc) do
80      {t, [h|acc]}
81    end
82    defp zip_each([], _) do
83      {nil, nil}
84    end
85  end
```

Listing B.1: First implementation of the mixed-radix FFT algorithm.

```elixir
defmodule FFT do
  @compile :native
  @compile {:hipe, [:o3]}

  def twiddle_factors(big_n) do
    0..(big_n-1)
    |> Enum.map(fn(i) ->
         cis -2*:math.pi*i/big_n
    end)
    |> List.to_tuple
  end

  #`:nat_array` is the Erlang `array` module compiled with HiPE.
  #If the input length includes a factor 2, 3 or 4, split the decomposition
  #over that many parallel processes using `Task.async/1`.
  def fft(xs, twiddles) do
    if :nat_array.size(xs) <= 1 do
      xs
    else
      {f, fs} = case ExMath.factors(:nat_array.size(xs)) do
        [2, 2 | fs] -> {4, fs}
        [3 | fs] -> {3, fs}
        [2 | fs] -> {2, fs}
        fs -> {1, fs}
      end

      children = 0..(f-1)

      tasks = Enum.map(children, fn(n) ->
                Task.async fn() -> fft(xs, f, n, twiddles, fs) end
              end)
            |> :nat_array.from_list

      :nat_array.map(fn(_, task) ->
        Task.await task
      end,
      tasks)
      |> butterfly(f, 1, twiddles)
    end
  end

  defp fft(xs, _stride, offset, _twiddles, []) do
    :nat_array.from_list([:nat_array.get(offset, xs)])
  end
  defp fft(xs, stride, offset, twiddles, [f | fs]) do
    :nat_array.map(fn(n, _) ->
      fft(xs, stride*f, offset+stride*n, twiddles, fs)
    end,
    :nat_array.new(f))
    |> butterfly(f, stride, twiddles)
  end

  defp butterfly(ts, f, stride, twiddles) do
    t_len = :nat_array.size(:nat_array.get(0, ts))
    size = f * t_len

```

```
57      :nat_array.map(fn(k, _) ->
58        rem_k = rem(k, t_len)
59
60        :nat_array.foldl(fn(n, t, acc) ->
61          :nat_array.get(rem_k, t)
62          |> c_op(elem(twiddles, rem(n*k, size)*stride), acc)
63        end,
64        {0, 0},
65        ts)
66      end,
67      :nat_array.new(size))
68    end
69
70    defp cis(x), do: {:math.cos(x), :math.sin(x)}
71
72    #c_op(a, b, c) := a*b + c, for three complex numbers
73    defp c_op({r1, i1}, {r2, i2}, {r3, i3}) do
74      {r1*r2 - i1*i2 + r3, r1*i2 + r2*i1 + i3}
75    end
76  end
```

Listing B.2: Final implementation of the mixed-radix FFT algorithm.

C Original Project Schedule

Week	Activity
1	Planning and preliminary research.
2	Research algorithms, libraries, prior work. Write Elixir port to SciPy.
3	Implement and benchmark FFT algorithms in Elixir.
4	Setup benchmarking environment for Python and C. Prepare benchmarking of SciPy and FFTW.
5	Benchmark SciPy and FFTW. Finish first draft.
6	Analyze benchmark results. Research performance improvements.
7	Benchmark and document eventual improved implementations.
8	Finish second draft.
9	Begin implementation of vocoder. Thesis writing.
10	Thesis writing. Continue work on vocoder if time left.
11	Finish final draft before presentation. Prepare for presentation. Write opposition report.
12	Presentation. Finish thesis.