Aalto University
School of Science
Master's Programme in Mobile Computing – Services and Security

Nicholas Kariniemi

Clojure on Android
Challenges and Solutions

Master's Thesis
Espoo, April 13, 2015

Supervisor: Professor Jukka Nurminen
Advisor: Vesa Hirvisalo D.Sc. (Tech.)

Aalto University
School of Science
Master's Programme in Mobile Computing – Services and Security

ABSTRACT OF MASTER'S THESIS

Author:	Nicholas Kariniemi		
Title	Clojure on Android: Challenges and Solutions		
Date:	April 13, 2015	**Pages:**	79
Major:	Mobile Computing – Services and Security	**Code:**	T-110
Supervisor:	Professor Jukka Nurminen		
Advisor:	Vesa Hirvisalo D.Sc. (Tech.)		

Mobile operating systems are rapidly expanding into new areas and the importance of mobile apps is rising with them. As the most popular mobile operating system, Android is at the forefront of this development. However, while other mobile operating systems have introduced newer, officially-supported languages for app development, the only supported language for Android app development is an older dialect of Java. Android developers are unable to take advantage of the features and styles available in alternative and more modern languages.

The Clojure language compiles to Android-compatible bytecode and is a promising language to fill this gap. However, the development of Android apps with Clojure is hindered by performance concerns. One recognized problem is the slow startup time of Clojure on Android apps. Alternative "lean" Clojure compiler projects promise to improve Clojure performance including startup time. However, the performance of Clojure on Android and the lean compiler projects has not been systematically analyzed and evaluated.

We benchmarked and analyzed the startup and run time performance of Android apps written in Clojure and compiled using both the standard Clojure compiler and experimental lean Clojure implementations. In our experiments the run time performance of Clojure on Android is similar to that of Clojure on the desktop. However, Clojure on Android apps take a significant amount of time to start, even on relatively new hardware and the latest Android versions. Long startup times scale upwards quickly with larger apps and the problem is closely tied to the Clojure compiler implementation. We also found that while the Skummet lean Clojure compiler project significantly reduces Clojure on Android startup times, more changes are necessary to make Clojure practical for general Android app development.

Keywords:	Clojure, Android, Java, JVM, benchmarking, performance, startup
Language:	English

Acknowledgements

I would like to thank my supervisor, Professor Jukka Nurminen, and advisor, Doctor Vesa Hirvisalo, for their support in writing this thesis. Thank you to Daniel Solano Gómez for the thesis idea and encouragement and to Alexander Yakushev and Reid McKenzie for the Clojure compiler projects that are the basis for large parts of this thesis. Finally, a huge thank you to my wife for both putting up with and not putting up with countless evenings working on the thesis and for reminding me about what's important.

Espoo, April 13, 2015

Nicholas Kariniemi

Contents

Chapter 1

Introduction

Clojure is a promising alternative language for Android application development. Clojure inter-operates well with Java code, allowing developers to use familiar tools from the Android Java ecosystem, while bringing modern functional, dynamic, and concurrent features. However, the performance of Clojure on Android is largely unknown. Slow startup time is a recognized issue but Clojure on Android performance has received little study. Alternative "lean" Clojure compiler projects attempt to address these performance concerns but their results have not been systematically evaluated. This thesis fills these gaps by benchmarking and analyzing Clojure on Android startup and run time performance using standard Clojure and alternative lean Clojure implementations.

Android and other mobile platforms are increasingly important targets for software development. Smart phones with modern operating systems have spread quickly in recent years and provide widespread access to a broad array of software services. Mobile platforms are also expanding rapidly beyond phones and tablets and into other realms such as watches, media centers, home automation, and automobiles. The Internet of Things is growing rapidly and mobile operating systems strive to be their platform of choice.

Android is the most widely used mobile operating system. As such Android is also a strong contender in the expansion into new computing domains. Android has historically benefited from its relatively open ecosystem. Since Android was released, the Android ecosystem has seen substantial improvements in the breadth and quality of tools and libraries for application development.

One area that has seen relatively little progress, however, is in programming languages for developing Android software. Android apps are typically developed using a Google implementation of Oracle's Java programming language. Since Android was originally released, the Oracle Java language has

advanced from version 6 to 8, bringing numerous incremental improvements along with larger features such as lambda expressions, streams, and better type inference. While some of these changes have slowly made it into Android Java, many have not and the process has been slow. Meanwhile, competing mobile platforms iOS and Windows Phone have introduced official support for the alternative functional and more modern languages Swift and F#, respectively. Android has no official alternatives to Java for typical app development. Unofficial alternatives exist but no single contender has achieved widespread support.

One promising contender for Android app development is Clojure. Clojure, like Java, compiles to bytecode that can be executed on Android. It inter-operates well with Java code, allowing developers to continue to use familiar tools and libraries from the Android ecosystem. In addition, Clojure has a number of potential advantages over Java due to its functional style, concurrency support, and dynamic development style. As a functional language, Clojure encourages the use of first-class functions and immutable data structures. This can make programs easier to reason about and test. Android is an inherently concurrent environment for development, and Clojure's immutable data structures can simplify concurrent programs. Clojure also has strong support for concurrency through various built-in constructs such as software transactional memory. Finally, the dynamic style of development typically used in Clojure development can shorten development feedback loops and potentially increase development speed.

1.1 Problem

As an alternative to Java, Clojure could provide Android developers with powerful tools for faster development of more reliable, maintainable, and performant software. However, Clojure on Android performance is largely unknown. Clojure on Android apps are known to start more slowly than Java apps but the problem has received little study. Possible solutions to the slow startup time problem for Android exist but the solutions have not been evaluated.

This thesis bridges this gap by attempting to answer the following questions:

1. How well does Clojure perform on Android?
2. What are the largest performance barriers to use of Clojure on Android?

3. How much do the lean Clojure compiler projects improve Clojure on Android performance?

4. Can Clojure ever become a practical language for Android development?

We focus on the development of *standard* Android apps using *JVM* Clojure. By standard apps we mean apps that would typically be developed using Java and do not require native C or C++ tools or third party frameworks such as PhoneGap [31] or Cordova [7]. By JVM Clojure we mean the implementation of Clojure for the JVM and not for other platforms such as the Microsoft Common Language Runtime or JavaScript. This is the most common case both from the Android perspective, where apps are typically developed using only the standard Java tools, and from the Clojure perspective, where the most popular Clojure implementation is for the JVM. This is also the most promising approach for Clojure on Android. It reaches the most common Android development use cases, allows access to the standard Android ecosystem of tools and support, and uses the most stable and performant Clojure implementation.

1.2 Methodology

This thesis analyzes the performance of Clojure on Android using experimental benchmarks and the YourKit Java profiler. Specifically, we provide the following contributions.

Startup benchmarking

We benchmark the startup time of minimal Java, Clojure, and lean Clojure apps on both Android and the desktop. We found that the minimum Clojure startup time is significantly longer than Java startup time and that the Skummet lean Clojure compiler reduces this time considerably.

Startup profiling analysis

We profile the startup process of minimal Clojure and lean Clojure apps on Android and the desktop. On both platforms we break down startup time based on class loading event timestamps. On the desktop we further examine startup performance using the YourKit Java profiler. We found that the large majority of Clojure startup time is consumed by the Clojure

runtime in loading the core Clojure libraries. This is true both on Android and the desktop and for both standard and lean Clojure.

Execution performance benchmarking

We benchmark the run times of a selection of programs from the Computer Language Benchmarks Game [20], supplemented with a few of our own benchmarks. The benchmarks are written in Java and Clojure and compiled using the Java, Clojure, and Skummet lean Clojure compilers. We found Clojure on Android execution performance to be worse than Java performance but comparable to Clojure desktop performance. Clojure startup times were found to scale upward markedly for larger programs, suggesting that typical Clojure app startup times would be much larger than the minimum times found in the prior experiments. The Skummet lean Clojure compiler started up much more quickly than Clojure across the benchmarks and had mixed execution performance compared to Clojure.

1.3 Structure of the thesis

In the first half of the thesis we provide sufficient background information about Android and the Clojure language to understand the experiments and analysis presented in the second half. Chapter 2 provides an overview of the Android app ecosystem and the way Android apps are developed and run. Chapter 3 introduces the Clojure language, with a focus on possible benefits for Android development and the features relevant to the experimental portions of the thesis. Chapter 4 describes in more detail aspects of the compilation and execution of Clojure programs relevant to later chapters. The limited research about Clojure performance is summarized in Chapter 5, along with relevant works on Android, JVM, and Java performance.

The second half presents experimental analysis of Clojure on Android performance and conclusions about the state and direction of Clojure on Android. The startup time performance of Clojure on the JVM and Android is characterized and analyzed in Chapter 6. Chapter 7 analyzes the execution performance of Clojure on Android. Chapter 8 provides broader discussion of the performance results and implications for the future. Finally, Chapter 9 summarizes the thesis work and its implications.

Chapter 2

Android Apps

The use of mobile phones has grown dramatically in recent years. Mobile phones are used more than any other electronic device and by a wider range of people. The most common mobile phone operating systems are Windows, iOS, and Android. Of the three, Android is the most popular, with 84% of all mobile phone shipments in the third quarter of 2014 [11].

This section describes the the environment in which Android applications are run first on a general level for mobile applications and then more specifically from both Android user and developer perspectives.

2.1 Mobile platforms

Mobile platforms are an increasingly important target for software development both because of the wide and growing using of mobile phones and because of the unique opportunities present in mobile applications. The ubiquity and availability of mobile phones means they are used in a wider variety of situations than traditional computers such as desktops or specialized computers such as servers. Mobile phones typically contain hardware allowing applications to see a user's location, take pictures, detect movement, record sound, interpret touch gestures, and connect to different devices. These are features that in most cases cannot be accessed on traditional computers or can be used only in limited contexts. Phones are also much more personal than computers and will usually have personal information such as contacts.

In addition, consumer-centric computing is spreading beyond phones to other devices. This spread is primarily happening not through desktop operating systems but through mobile operating systems. Mobile operating systems are used in tablets, TVs, cars, media centers, and other devices. All three major mobile operating system companies, Google, Apple, and Mi-

crosoft, have released watches that use or inter-operate with their mobile operating systems in 2014.

The primary way that companies interact with users in mobile operating systems is through *apps*. All three of the major mobile platforms, Android, Windows, and iOS, package and distribute programs to users as apps. Apps perform many different functions. Widely used apps such as Facebook or Twitter are a more accessible interface to the company's web applications. Apps such as Google Maps provide a map of the world and GPS navigation to users, supplanting other services such as automobile GPS navigators. Other popular apps such as WhatsApp, Snapchat, or Skype provide different ways for people to communicate. There are also countless games, from Clash of Clans to Angry Birds to Candy Crush Soda Saga. All of this functionality is accessed by users as apps.

Users typically find and install apps through an app store like the Google Play Store on Android or the Apple App Store. Apps may be free or paid or have in-app purchases or advertisements. Once installed, the app icon appears in a prominent place on the phone. On Android devices the icon appears on the home screen and within the user's list of apps. A user opens the app by selecting its icon and the app displays and loads. An app may also be opened in other contexts. On Android apps may directly launch other apps for purposes such as sharing.

In both of these cases the time it takes for the app to display is crucial. Jakob Nielsen in his book on software usability titled *Usability Engineering* gives three rule of thumb times for responding to user actions. Times under about 0.1 seconds are perceived as instantaneous and do not require special feedback because the user can clearly see that what happens is a direct and instantaneous result of their action. Up to one second of delay is noticeable but not enough to lose the user's train of thought. Delays of up to ten seconds will probably not lose the user's attention. Longer delays cause users to find other tasks to do while waiting for the operation to complete [44]. Studies of web site users similarly suggest a limit of two seconds for web pages to load to reduce risk of user abandonment and corresponding revenue loss [43].

User expectations about response times are likely to be much stricter for mobile phone apps. Some actions such as saving an item to another service or sharing a picture with friends must happen quickly or users are unlikely to perform the action.

2.2 Android software stack

The Android software stack can be divided into four main levels: applications, the application framework, the libraries and runtime, and the Linux kernel.

At the top level of Android are the applications themselves. Android apps as discussed in the previous section present most of the interface that the user interacts with on mobile and newer embedded devices. Android takes this idea even further than competing platforms by implementing many functions typically performed in the operating systems as apps instead. The home screen, where users can view, launch, and organize installed apps, is itself just an app like any other. The contacts list is its own app, as is the phone dialer for making calls and the messaging app for sending text messages.

Android apps are typically written in the Java language and rely heavily on the application framework on the next level of the software stack. The application framework provides a large number of APIs and services for UI elements, making phone calls, finding the user's location, displaying notifications to the user, interacting with other apps, and so forth. The APIs are provided in Java.

Below the application framework level are native libraries and the Android runtime. Native libraries are libraries compiled specifically for the mobile phone hardware and provide tools for creating secure connections, storing SQL data in SQLite, rendering web pages, and displaying graphics for games, among other things. Apps can take advantage of these native libraries. The Android runtime is a virtual machine on which apps are run. It provides an interface similar to the Java Virtual Machine, which allows apps to be compiled once for the virtual machine and run in many different environments. Most current Android phones use the Dalvik runtime and newer devices use the Android runtime (ART). The Android runtime is described in more detail in Section 4.

At the lowest level of Android is the Linux kernel. The Linux kernel provides a large number of low level functions for interacting with hardware components such as the screen, the camera, or USB devices.

2.3 Android app development

Android apps run on a virtual machine. The use of a virtual machine in Android means apps can be developed in any language that compiles to the virtual machine and executed on any device that has an implementation of the virtual machine.

Android apps are typically written in Java. A developer writes Java that

is transformed by a Java compiler into bytecode. On a desktop computer this bytecode is executed directly by the Java Virtual Machine. On Android, there are a few additional steps to package and convert the code to a format suitable for Dalvik or ART. The entire process of converting Java code to a package executable on Android is shown in Figure 2.1.

Figure 2.1: Android Build Process

Android apps are started by users but are typically stopped by the Android system. When a user leaves an application, the app is paused in the background but kept in memory. When a user returns to a paused application, the application is restored. If a user never returns to an application, at some point the Android system may decide to shut down the app and free up the memory the app is using [22]. This has important implications for the memory consumption of apps, as apps that use more memory will be terminated more quickly.

One important aspect for developing Android applications is concurrency. Android apps are inherently concurrent. Most Android apps interact directly with the user. By default all action in an Android app occurs on a single thread. This means that if an action needs to do a lot of processing or needs to wait for a network call to return the thread can be blocked. Blocking this main thread freezes the app for the user until the action is complete. This is not only poor for the user experience but may cause Android to prompt the user to shut down the app [23]. For this reason any processor-intensive or asynchronous tasks need to be handled on another thread. When results need to be displayed on the UI, the other thread needs to communicate the results to the UI thread.

As writing concurrent applications properly at a low level is notoriously difficult, Android provides a few different tools for handling actions off the UI thread and communicating results back to the UI thread. As in Java, Android provides low level tools such as the Thread and Future classes and locks. The low abstraction level makes them difficult to work with correctly, and the developer has to carefully consider concurrency concerns such as deadlock, starvation, and liveness.

The main tool suggested by Google for handling concurrency is the AsyncTask class [22]. AsyncTask allows one to define a sub-class which performs a specific task off the main thread. When the task is complete it provides

a way to call another method on the UI thread with the results of the task. An example is shown in Figure 2.2.

```
private class DownloadFilesTask extends AsyncTask<URL, Integer, Long> {
    protected Long doInBackground(URL... urls) {
        // Long-running work on background thread
    }

    // This is called when doInBackground() is finished
    protected void onPostExecute(Long result) {
        // Result obtained on UI thread
        showNotification("Downloaded " + result + " bytes");
    }
}
```

Figure 2.2: AsyncTask

There are several problems with using AsyncTask. AsyncTask is easy for simple, one-off tasks where the app does a single request and returns a response. For more complicated pictures it becomes much more difficult. For instance, a login process may require three different calls. If one of the calls fails, the whole process may need to be aborted, or the call may need to be repeated. There may be dependencies between the asynchronous tasks, where one call depends on a previous call or must be canceled if a user cancels the action. Using AsyncTask the logic for this login call is scattered across many methods and several different classes. There is no straightforward way to handle errors.

Android developers have come up with different solutions to this problem. Promises are one approach used in many other languages, where the results of an asynchronous call are wrapped in an object that can be later unwrapped in order to get the results. Callbacks are another approach. With callbacks, asynchronous methods are passed an anonymous class which is called on the resulting thread when the action completes. Another more flexible approach is the use of the Reactive Extensions (Rx) library [41] originally developed by Microsoft and in use at companies such as Netflix [10].

Other languages that compile to JVM bytecode can also provide additional tools for managing concurrency. One such language is Clojure, which is discussed in the next section.

Chapter 3

The Clojure Language

This chapter briefly introduces the Clojure language and provides an overview of Clojure features relevant to later sections of the thesis.

3.1 Clojure overview

Clojure is a functional, dynamic, Lisp-based language with strong support for concurrency that runs on the JVM [16, 27, 28].

As a functional language, Clojure has first-class functions and emphasizes pure functions and immutable state. Functions can be defined and passed around in Clojure like any other variable, which provides a large amount of expressiveness and flexibility. Clojure encourages pure functions, which have no side effects, but does not require them. Pure functions reduce the mental burden of the developer, who only has to know what values are passed in to know what value a function will return. One way Clojure encourages pure functions is by making most program state immutable by default. Immutability can simplify software development by reducing the number of ways in which a program state can change. A program with fewer moving parts is conceptually easier to understand. Immutable state also simplifies concurrency challenges because immutable state can be passed freely between threads without fear of one thread changing another thread's data.

Clojure is a dynamic language, allowing the developer to perform many actions at run time that are typically available only at compile time. This is discussed in greater detail in Section 3.2.

Clojure also belongs to the Lisp family of languages. Like other Lisps, Clojure uses the same representations for code and data and has strong support for macros. This makes the language very extensible. For example, some features can be implemented using macros in Clojure that in other

languages require extensions to the language itself.

Clojure has strong concurrency support. Besides immutable data as a default, Clojure provides a number of different tools for handling state changes safely. *Vars*, *atoms*, *refs*, and *software transactional memory* each provide a different thread-safe way to alter program state.

Finally, Clojure is a JVM language and inter-operates well with Java code. As a JVM language, Clojure can be executed in most environments that support Java, including Android. In addition, Clojure has various tools to ease the use of Java libraries and features from Clojure. Java classes can be extended, referenced, and used directly. Existing Java libraries can be used directly in Clojure code. These are important features for Android development, as the Android framework and libraries are provided almost exclusively in Java. Clojure also provides some support for the other direction, using Clojure from Java.

3.2 Dynamic features

Clojure is a dynamic language. The term dynamic is overloaded but static and dynamic are frequently used to distinguish between compile time and run time. Clojure is a dynamic language because it gives the user a larger amount of flexibility at run time than many languages.

Important dynamic features relevant to later discussion in the thesis are the REPL, dynamic compilation, reified language constructs, and dynamic binding of vars and namespaces. Together these features allow Clojure developers to use a dynamic, REPL-driven style of development.

Read-Eval-Print-Loop

The Read-Eval-Print-Loop or REPL is a console for interacting with Clojure. Clojure forms can be entered at the console to be read and evaluated and their results are printed. The REPL is the primary programming interface for developing Clojure programs and derives much of its power from the dynamic features presented subsequently.

Dynamic compilation

Clojure supports dynamic compilation and evaluation. Any Clojure code can be loaded, compiled, and evaluated at run time. The code is compiled to JVM bytecode on the fly when the forms are loaded. One implication of this is that the Clojure runtime comes with a fully functional compiler. A

second implication is that the entire Clojure language is available to Clojure programs at run time.

Reified language constructs

Clojure makes a number of constructs available at run time that are in many languages only available implicitly at compile time. This process of making implicit language constructs explicit is called *reification*. Symbols, vars, and namespaces are all examples of reified constructs in Clojure. A symbol is an identifier typically used to refer to another object. A var is a pointer to a storage location holding for example an integer or a function. A namespace is a mapping of symbols to vars, which in turn map to other values. All three of these constructs are explicitly available at run time in Clojure and can be used and manipulated like other language constructs. This is in contrast to many other languages, where similar concepts may be used internally during compilation without being available to the developer.

Dynamic binding of vars and namespaces

One important example of dynamism in Clojure relevant to this thesis is the dynamic binding of vars and namespaces. A var is a pointer to a value such as a function or an integer and is usually defined using the "def" special form or "defn" macro. The "defn" macro is a wrapper around "def" used to define functions. In practice most functions in a program are defined using "def" as a var pointing to an anonymous function value.

Clojure vars reside in namespaces, which are the basic unit of organization for Clojure programs. A namespace is a mapping from symbol identifiers to vars or Java classes.

The following example presents namespaces and vars in a simple but complete Clojure program. The first Clojure form defines a namespace named "hello". The next two forms define two vars, "foo" and "print-foo", pointing to an integer value "1" and a function to print the value of "foo".

```
(ns hello)

(def foo 1)

(defn print-foo []
    (println foo))
```

Vars and namespaces are reified and can be inspected at run time. They are also dynamically bound, and can be added, removed, or given new values

at run time. For example, if the statements of the previous example were executed in a REPL console, the user could later switch back to the "hello" namespace and inspect the values of the vars. The var "foo" could also be given a new value.

```
=> (in-ns 'hello)
#<Namespace hello>
=> foo
1
=> (print-foo)
1
nil
=> (def foo 5)
#'hello/foo
=> foo
5
=> (print-foo)
5
nil
```

Anywhere the var "foo" is used elsewhere in the program the new value will now be used. This happens without recompiling the namespace "hello" or the function "print-foo". This point has important implications. For development it makes it easier to write and test new code. If something doesn't work it can be redefined, recompiled efficiently, and tested until it does. The second important implication is more subtle. All places that use the "foo" var in the code will see the new value immediately, without being recompiled. This means that the value of var "foo" is not used directly but is looked up every time it is used. Every use of "foo" must first fetch the value of "foo" from its namespace "hello" in order to use it.

Dynamic binding works similarly for namespaces. Every time a var in a namespace is used, the namespace is looked up from a global table mapping namespace identifiers or symbols to the namespace objects themselves. Then the var is fetched from the namespace based on its own identifier, and finally the value of var can be used.

Dynamic binding of vars and namespaces is an exceptional feature for a language that otherwise emphasizes immutability and pure functions. In most cases it is recommended to define namespaces and vars only once in production code, and to use dynamic binding of vars and namespaces only during development. This convention allows vars and namespaces to be generally treated as immutable in production code. Treating them as immutable gives developers much of the simplicity and maintainability of actual immutable objects, though without the same guarantees.

Dynamic development

The REPL, dynamic compilation, reified language constructs, and dynamic binding of vars and namespaces together provide Clojure developers with a powerful environment for a dynamic style of development. Clojure programs are typically developed in an iterative, interactive fashion that makes heavy use of the REPL. Developers may write, evaluate, and test individual pieces of functionality interactively in the REPL. Reified language constructs mean program values can be inspected at the REPL. Dynamic binding of vars and namespaces allows program functions and values to be changed on the fly. Dynamic compilation and evaluation means programs can be redefined and recompiled efficiently on the fly or in the REPL as needed.

This style of dynamic or "REPL-driven" development is an integral part of the Clojure language. It deeply affects the way Clojure code is evaluated, as discussed in Chapter 4, and, ultimately, the performance of the Clojure language (Chapter 6 and Chapter 7).

Chapter 4

Clojure Compilation & Runtime

This section provides background on Clojure compilation and the Clojure runtime helpful for understanding the analysis and conclusions presented in following sections. It shows how the dynamic features discussed in Chapter 3.2 connect to design choices in the compiler and runtime, in particular with respect to how Clojure programs start. These design choices have an impact on the performance of Clojure on Android as will be seen in Chapter 6.

Program compilation and execution is discussed first on a general level and then from the perspective of virtual machines and the Java Virtual Machines. This is followed by more specific discussion of Clojure compilation, the Clojure runtime, and the Clojure startup process.

4.1 Compilation and interpretation

At a low level computers understand and execute commands expressed in binary machine language. For convenience these binary operations are given corresponding names in assembly language. For instance, a computer may have a command to add two numbers together and put the result in a specific memory location. For many tasks this is too low of an abstraction level to do useful work, however, and for that reason programs are developed in more powerful and expressive high level languages. High level languages need to be converted to machine language in order to be executed.

Two broad approaches for executing code written in high level languages are compilation and interpretation. Compilation takes program source code written in a high level language and converts it ahead of time to low level machine code understandable by a computer. The machine code can then later be executed by the machine.

Interpretation takes an opposite approach. Instead of converting the source code to binary code ahead of time, the machine may interpret the original high level language source code directly. One extreme form of interpretation is *decode-and-dispatch* interpretation [54]. In decode-and-dispatch interpretation the interpreter steps through the original program source code one line at a time, decodes each instruction, and dispatches it to the appropriate routine to be executed. For each source code command there are corresponding routines that execute the command in machine code. The program source code is executed directly without the need of a complete intermediate machine code representation.

One basic tradeoff between compilation and interpretation is between execution speed and implementation flexibility. Compilation can produce highly optimized machine code that executes quickly. The whole source code is available during compilation, which allows the compiler to take shortcuts not available to interpreters. A drawback of ahead of time compilation is that the source code must be separately compiled for each targeted machine architecture. Writing the compiler code itself can also be quite complex. Interpreters can be much easier to write and modify, allowing easier innovation with new languages. For the same reason interpreted languages can be easy to port to different architectures. Interpreted code typically executes more slowly than compiled code, however, because interpreters do not have the entire source code available and can make fewer optimizations.

4.2 Virtual machines

Between the extremes of ahead of time compilation and decode-and-dispatch interpretation are a large number of variations. One such variation is to insert a layer between the source program and the underlying hardware called a *virtual machine* [54]. Program code can be compiled or interpreted to an intermediate virtual machine code representation. The virtual machine then executes this code on the underlying machine.

Virtual machines can provide portability and flexibility without sacrificing all of the speed of strict interpreters. Portability is achieved because the program source code only needs to be compiled once for the virtual machine. The virtual machine is compiled for different underlying architectures, and the program can be executed without recompilation on any architecture supported by the virtual machine. The virtual machine provides an abstraction over the underlying hardware and the executing program uses this abstraction in place of the hardware.

The intermediate abstraction layer of a virtual machine gives a large

amount of flexibility to language developers and users. A language developer can develop a language for a single virtual machine that is able to execute on all machine architectures that the virtual machine supports. Virtual machines provide their own run time environment for programs. A language may take advantage of the virtual machine's handling of threads, the file system, and security, for example, without needing to implement these features at a low level within the language.

Performance in virtual machines can be obtained through techniques such as just-in-time (JIT) compilation. In JIT compilation program source code is compiled to machine code on the fly as needed during execution of the program. A JIT compiler may take advantage of run time information about how a program is executing in order to optimize the code. It may initially compile source code quickly to inefficient code, and then recompile code that is used often to be more efficient. When using JIT compilation in virtual machines, the original source code will typically be compiled to virtual machine code and then JIT-compiled by the virtual machine to machine code.

One widely used virtual machine is the Java Virtual Machine used by Java, Clojure, and other languages and discussed in the following section.

4.2.1 Java Virtual Machine

The Java Virtual Machine (JVM) is a layer between high level languages such as Java, Scala, JRuby, or Clojure, and the underlying platform [37]. Programs written in high level languages are compiled to JVM bytecode contained in class files. The JVM reads and executes bytecode from the class files.

The JVM refers to the JVM specification, a JVM implementation, or a JVM instance. The specification defines what is required in a conformant implementation. An implementation is a compiler and runtime that meets the specification and can compile and execute JVM programs. A JVM instance is a JVM process executing a single program. In this section we primarily discuss the JVM specification.

The class file format is the interface of the JVM from a programming language perspective. Any language that conforms to this interface can run on the JVM. Class files contain JVM instructions or bytecodes along with the data necessary to execute the instructions. Each class file contains a description of a class and the methods defined in the class. The general format is presented in Figure 4.1 [37].

Before any code from a JVM class can be executed, the JVM must load, link, and initialize the class.

Loading a class is finding the binary representation of a class and creating

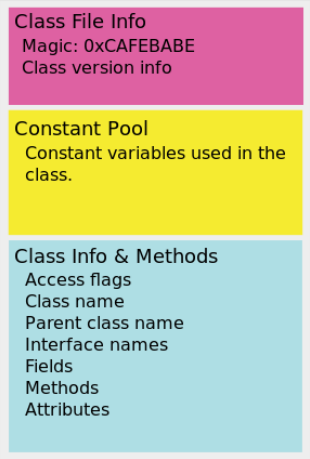


Figure 4.1: Class file

a corresponding internal JVM representation. A JVM class does not need to be stored as a file, but can also be loaded over the network or generated on the fly as in Clojure dynamic compilation.

Linking a class is adding the class to the current state of the JVM runtime. The linking process also verifies that the representation of the class or interface is structurally correct and prepares static fields by creating them and setting them to default values. The linking step may also resolve symbolic references to the actual classes or interfaces that contain them.

Initialization is executing the class initialization method of the class so that the class is ready to be used. In class files this class initialization method is named "clinit".

Classes are initialized only when needed. A class is initialized at JVM start up if it is the main specified class. Classes are also initialized when they are referenced using JVM instructions like "new" or static method invocations, or when a sub-class is initialized. When a class needs to be initialized, it must first be loaded and linked. The JVM specification allows loading and linking to occur at any time before initialization, leaving room for implementers to optimize the process.

The JVM class file structure is designed around the needs of the object oriented language Java. For this reason the mapping between Java constructs and JVM constructs is fairly straightforward. A Java class, for instance, maps directly to a single JVM class file. For other JVM languages such as the functional Clojure language the mapping is less straightforward.

4.2.2 Dalvik virtual machine

The Dalvik virtual machine is a virtual machine developed by Google for executing Android apps. Android apps are typically written in Java, which is compiled to JVM bytecode. Dalvik can execute JVM bytecode by converting JVM bytecode in class files to Dalvik Executable files. Dalvik Executable files, which have a ".dex" extension, are comparable to JVM class files but have changes designed for the mobile environment. In particular Dalvik is optimized for a low memory footprint and fast program startup times [15].

Dalvik achieves a lower memory footprint by sharing constants between classes and merging many JVM class files into typically a single ".dex" file. In standard Java, each class in the source code is compiled to a single JVM class file. A large portion of the class file may be devoted to the constant pool, which contains all literal constants used within the class. When the Dalvik conversion tool *dx* is used to convert JVM class files to ".dex" files, all class files for a single application are converted into a single Dex file and their constant pools are merged. Duplicated constants appear only once in the merged constant pools. This can result in files of less than half the size of the corresponding JVM files [15]. Memory use is also reduced because the Dalvik virtual machine needs to load and keep in memory fewer bytes.

Reducing the number of bytes to load can also decrease app startup time, but the primary way Dalvik targets app startup time is through the Zygote process. The Zygote is a virtual machine process started when an Android system is booted. When it starts, it loads a Dalvik VM instance with partially initialized core library classes.

For security reasons each Android app is run in its own virtual machine instance, which is requested from the Zygote process. The Zygote process spawns new Dalvik instances by forking the initial Dalvik VM process. In the Linux kernel, forking a process does not actually copy the memory from the old process to the new process. Instead it uses a "copy on write" strategy where memory is copied only when it is written to. For these core libraries, the majority of the time they are read from and not written to, so the memory can be largely shared between processes [15].

Thus new processes start quickly because they are forked from a pre-loaded process and can share use of existing libraries that may be already loaded into memory. Sharing memory for core libraries is also another way in which Dalvik reduces application memory footprint.

4.2.3 Android runtime

With the Android 5.0 release Google replaced the Dalvik virtual machine with the Android runtime (ART). ART is a virtual machine runtime with two important changes compared to Dalvik: ahead-of-time (AOT) compilation and improved garbage collection [24].

Dalvik is a Just-in-Time (JIT) compiler. It compiles Dalvik bytecode to machine code dynamically during run time when needed. ART, by contrast, compiles app code ahead of time. App code is compiled to machine code when the app is first installed. This is designed to improve the memory and speed performance of Android apps.

The second major change with ART is the way garbage collection is handled. Garbage collection is the process of reclaiming app memory which is no longer being used. Dalvik garbage collection in certain cases would freeze apps for relatively long periods while garbage collection was happening. ART improves garbage collection by reducing the number of pauses and keeping the pauses shorter [24].

From a user or developer perspective the change from Dalvik to ART is almost transparent. ART apps take a longer amount of time to install and are expected to perform better.

4.3 Clojure compilation and runtime

4.3.1 Compilation

Clojure code can be run on a number of platforms, including the Microsoft CLI, C, and JavaScript through the Clojure-like ClojureScript language. The most popular and best-supported platform for Clojure, however, is the Java Virtual Machine. Clojure code can be either Ahead of Time (AOT) compiled or dynamically compiled when needed. In the former case the code is compiled to Java class files that can be read by the JVM. In the latter case source code is compiled on the fly into an intermediate Java bytecode representation in memory which is not written to disk. Clojure code is never interpreted, although dynamic compilation makes this a subtle distinction from a developer perspective. All Clojure code is compiled to JVM bytecode before execution, while dynamic compilation and evaluation provide much of the flexibility of interpretation without the performance penalty.

The Clojure compiler works by sequentially reading and evaluating each Clojure source code form in turn. For each form, the compilation executes four stages: reading, macro expansion, analysis, and emission. Reading takes

in a text form and converts it to a Clojure data structure. In practice this means a Java object corresponding to a Clojure data structure such as a list.

Macro expansion expands any macros in the source code into regular Clojure forms. After this stage, with possible recursive calls for further macro expansion, the resulting form contains Clojure forms free of macros.

Analysis and emission occur in the same phase in the standard Clojure JVM compiler. Analysis parses the Clojure data structure into the language expressions that it represents. Emission occurs as a side effect of analysis. This is where dynamic class generation and loading occurs. When the Clojure compiler encounters a form that needs to be compiled to a new class file, it generates the new class. If Ahead of Time (AOT) compilation is being used the generated class is saved to a class file on the file system. If the form is just being evaluated, it is evaluated directly in memory without saving the generated classes to disk.

Most of the standard Clojure compiler is written in Java. The main exceptions to this are the Clojure language core functions, which are mostly written in Clojure. It is not uncommon, however, for core functions to defer directly to Java libraries in order to perform their tasks.

4.3.2 Compilation for Android

Clojure compilation for Android works in the same way as Java compilation for Android. Clojure source, including the Clojure runtime, is compiled ahead of time to JVM bytecode. The bytecode is then converted and packaged into an Android application just as with Java. The process is shown in Figure 4.2.

Figure 4.2: Android Build Process

One fundamental limitation of the Android platform for Clojure code is that dynamic class loading is not supported by Dalvik. This means that all Clojure code must be compiled ahead of time and the dynamic development features such as runtime evaluation of arbitrary code described in Section 3.2 are not available in the same way. This also limits the utility of the dynamic binding of vars and namespaces described in Section 3.2. One important use of dynamic var and namespace binding is for dynamic development at the REPL, which is not possible in the same way in Clojure on Android.

4.3.3 Runtime

A programming language *runtime* is the environment in which the developer's code executes. The JVM provides a runtime in the form of the virtual machine itself. As part of this runtime the JVM provides an interface to access underlying hardware features, a standard set of Java libraries, a garbage collector, and features such as the ability to dynamically load and execute Java code.

Clojure has its own runtime. A Clojure program is executed within the Clojure runtime, which in turn executes on the JVM runtime. The dynamic features described in Section 3.2 are available through the Clojure runtime. This includes dynamic loading and execution of Clojure source code, generation of Java classes and objects, and redefinition of vars and namespaces.

All of the functions defined in the Clojure language also come from the Clojure runtime. Clojure's core functionality, including most of the basic functions defined in the language, is defined within the core namespace "clojure.core". The core namespace is created and loaded like any other namespace in Clojure. All other namespaces depend on the core namespace in order to use basic Clojure functions. The core functions are accessed through vars defined in the core namespace. In order for any of these functions to be used, they need to first be loaded. The process by which they are loaded is described in the following section.

The Clojure language and runtime in its entirety is distributed as a single JAR file. Java programs and libraries are typically distributed packaged as JAR files. The Clojure JAR file can be treated as an ordinary Java package and can be loaded and used by the JVM like any Java package. Clojure functionality is available by including this package in other Java bytecode.

4.3.4 Startup process

A Clojure program is typically started in one of three ways:

1. By loading a Clojure REPL and executing Clojure code in the REPL.
2. By manually calling the Clojure language JAR and passing it a Clojure source file to execute.
3. By AOT-compiling Clojure code into JVM bytecode and executing the main class in the same way as a Java program.

The first two options dynamically load Clojure source code, compile it to Java bytecode, and execute the resulting bytecode. In the third option

Clojure code is first compiled and then executed in two completely separate stages.

Production code will often be executed as AOT-compiled code for performance discussed in Section 4.3.2. For this reason the discussion of the Clojure startup process focuses on AOT compilation, although much of the process is the same for other ways of execution.

In broad terms a Clojure program is run by loading the main Clojure namespace and using functions from this namespace to load and execute one's own code. The main Clojure namespace is named "clojure.core". This namespace contains all of the functions defined by the Clojure language. Except for a few critical functions, these functions are written in Clojure. This Clojure core namespace is compiled like any other Clojure namespace. Unlike every other Clojure namespace, however, the core namespace is included with every Clojure program and is loaded before any user Clojure code.

Clojure source files typically begin by declaring a new namespace with a call to the namespace macro "ns". The prototypical AOT-compiled Clojure Hello World program could be defined as in Figure 4.3.

```
(ns hello.core
    (:gen-class))

(defn -main [& args]
    (println "Hello world"))
```

Figure 4.3: Clojure Hello World

This example defines a single function "-main", which is defined within the namespace "hello.core" and prints the text "Hello world" when called. The "gen-class" option on the "ns" macro tells the compiler to create generate the bytecode compatible with a Java class of the same name. The program can then be run like any other Java program by executing the "-main" function or "main" method on the class file containing the "hello.core" class.

The macros "defn" and "ns" and the function "println" are all defined within the Clojure core namespace. Clojure macros are expanded at compile time into macro-free Clojure code, so the bytecode generated from this example corresponds to the macro-expanded Clojure code in Figure 4.4.

Expanding the Clojure macros reveals the underlying imperative nature of Clojure programs. A Clojure program is compiled by sequentially evaluating each form in turn, converting them to bytecode in the process, and saving the resulting bytecode. Pre-compiled Clojure code is executed similarly by sequentially stepping through the equivalent bytecode of each original source

```
(do
  (clojure.core/in-ns 'hello.core)
  ((fn*
     loading__4910__auto__
     ([]
       (. clojure.lang.Var
        (clojure.core/pushThreadBindings
          {clojure.lang.Compiler/LOADER
           (. (. loading__4910__auto__ getClass) getClassLoader)}))
       (try
         nil
         (clojure.core/refer 'clojure.core)
         (finally
           (. clojure.lang.Var (clojure.core/popThreadBindings)))))))
  (if (. 'hello.core equals 'clojure.core)
    nil
    (do
      (. clojure.lang.LockingTransaction
       (clojure.core/runInTransaction
         (fn*
           ([]
             (clojure.core/commute
               @#'clojure.core/*loaded-libs*
               clojure.core/conj
               'hello.core)))))
      nil)))

(def -main
    (fn* ([& args]
       (println "Hello world"))))
```

Figure 4.4: Clojure Hello World with expanded macros

code form in turn.

A Clojure namespace is compiled into a namespace initializer class file and a class file for each var and Java class defined in the namespace. The "gen-class" option also creates an additional wrapper class compatible with a Java class of the same name.

The expanded example of Figure 4.4 compiles to five classes: a namespace initialization class, a namespace wrapper class, a "-main" var class, and two classes for the two anonymous functions in the "ns" macro (defined via "fn*"). The program is executed by calling the "main" method of the namespace wrapper class.

In order for any Clojure namespace or function to be used, it must first be loaded via a method defined in the Clojure runtime class "clojure.lang.RT". In the initialization of this Clojure runtime class, the Clojure core namespace is loaded and initialized. This means that one of the very first things any Clojure program does is load the "clojure.core" namespace.

The core namespace is defined primarily in the source file "core.clj", which is compiled to an initialization class file "core__init.class" and a large number

of class files corresponding to functions defined in the namespace. This is represented in Figure 4.5.

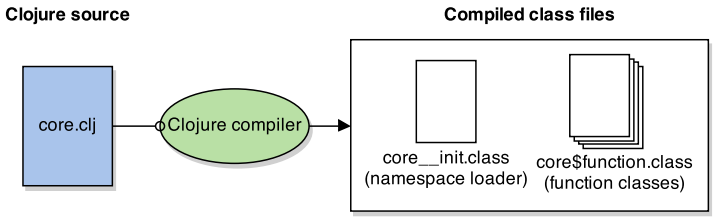


Figure 4.5: Clojure core compilation

The core namespace is loaded by loading the "core__init.class" file and executing its static initializer. The decompiled bytecode for the core namespace static initializer is represented in Figure 4.6. This bytecode is exactly the same for every single Clojure program.

```
static {
    // Create Vars and metadata
    __init0();
    __init1();
    __init2();
    __init3();
        // ...intermediate lines omitted
    __init23();

    Compiler.pushNSandLoader(Class.forName("clojure.core__init").
        getClassLoader());
    try{
      // Assign Vars and metadata, load external functions
      load();

      Var.popThreadBindings();
    }
    finally
    {
      Var.popThreadBindings();
      throw finally;
    }
}
```

Figure 4.6: Clojure core static initializer

The static initializer executes a number of "init" methods and then calls the "load" method within a certain context.

The "init" methods create vars and metadata corresponding to each function defined in the core namespace. The "load" method then adds those vars to the namespace object for the core namespace, adds the metadata to the

vars, and points the vars to a new instance of the corresponding function class.

For example, the core namespace defines a function "cons". The function itself is compiled to a class file named "core$cons.class". Loading the core namespace creates a var in the core namespace object pointing to an instance of this class.

The end result of loading the core namespace and executing the core namespace initializer "core__init.class" is the creation of a "clojure.core" namespace object with core function mappings. For each function in the core namespace, the object contains a var pointing to the class instance which implements the function. This mapping is represented in Figure 4.7.

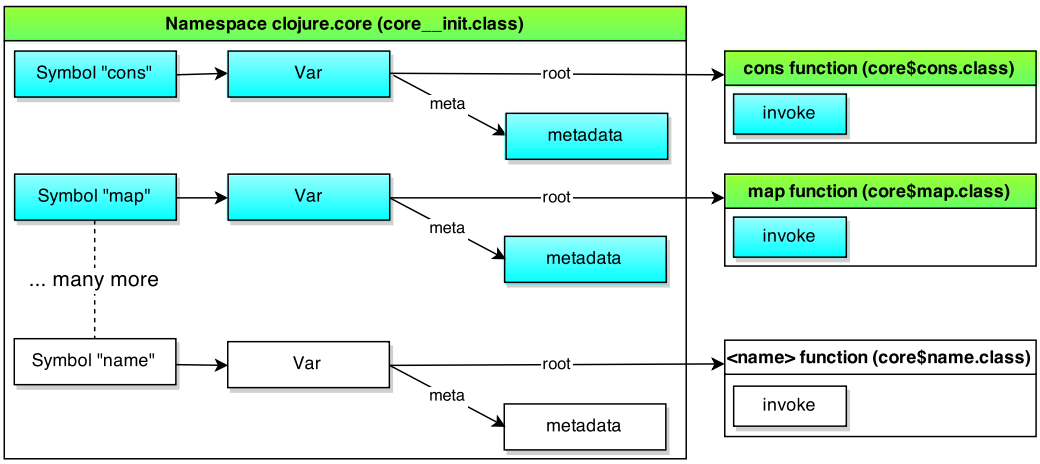


Figure 4.7: Core namespace vars

A core function can then be invoked in the user's code with JVM bytecode corresponding to the Java code of Figure 4.8, using the core function "cons" as an example.

```
RT.var("clojure.core", "cons").getRawRoot().invoke(args);
```

Figure 4.8: Fetching and invoking a var

The call to "RT.var" first looks up the core namespace by name to get its namespace object. It then fetches the var corresponding the the "cons" function and returns it. An instance of the class corresponding to the function itself is obtained with "getRawRoot" and finally the function can be invoked by calling "invoke".

Each invocation of a var, such as a Clojure function, requires two levels of indirection. First the namespace object is fetched, then the var is fetched from the namespace, then the function itself is fetched from the var. This indirection is necessary because both namespaces and vars are mutable at run time. They can be redefined at any point to point to other namespaces and vars, and all code that uses the given namespace or var will then use the updated value. This feature is obtained only because namespaces and vars are looked up explicitly on every call.

To support this dynamic behavior, the Clojure compiler loads all core namespace functions at the beginning of every Clojure program. This has a cost on startup performance, as discussed in Chapter 6 and Chapter 7.

4.4 Lean Clojure

The Clojure startup behavior discussed in the previous section has associated cost in the start time of Clojure programs. For this and other performance reasons over the summer of 2014 two alternative Clojure compiler implementations were developed. The compiler projects are named Skummet and Oxcart and are designed to trade aspects of Clojure's dynamic behavior for improved startup and execution performance.

In this section we first discuss the shared goals of the projects and the intended changes. We then describe individually the results of the Skummet and Oxcart project.

4.4.1 Lean Clojure changes

The basic premise of both lean compiler projects is to trade dynamic features of Clojure for performance. In particular, the projects statically compile Clojure vars and namespaces and eliminate dynamic code evaluation in order to reduce Clojure memory consumption and startup times and improve execution performance.

Static var and namespace binding

Both compilers bind vars and namespaces statically instead of dynamically. The Skummet compiler allows specific vars to be marked to be compiled in the normal dynamic way. The Oxcart compiler compiles all vars statically.

As discussed in Section 3.2, a Clojure var is a pointer to a value such as an integer or function. Most functions in a Clojure program are defined using vars. The vars are dynamic, which mean their values can be updated

at run time and the changes are visible within the same thread without recompilation. A namespace is a mapping of var names to vars.

With static compilation in the lean compilers, the vars can only be defined once at compile time and are fixed at run time. One implication of this is that lean compilation cannot be used with the REPL-driven, dynamic style of development discussed in Section 3.2. With lean-compiled code if a var is changed, all of the code that depends on the var must be recompiled for the changes to be visible.

Static var and namespace compilation is expected to improve Clojure startup times, memory performance, and execution speed. Statically compiled vars and namespaces can be compiled more compactly than dynamic vars and namespaces. Where the standard compiler compiles each namespace and var to its own class file, static compilation would allow vars to be statically compiled within namespaces. This would overall reduce the amount of work needed to set up namespaces and their functions, variables, and values and reduce the amount of bytecode needed to be loaded. This in turn is expected to reduce Clojure program startup times.

This is also expected to improve memory performance and execution speed by reducing indirection. As discussed in Section 3.2, most function calls in Clojure result in two indirect lookups. First the var must be fetched from the namespace, then the function is fetched from the var, then the function is invoked. Under static compilation the function can simply be invoked directly. This is expected to improve performance by reducing the amount of work and making it easier for the underlying virtual machine to optimize function calls.

Removal of dynamic code evaluation

The lean compilers also explicitly remove support for run time evaluation of arbitrary code. As discussed in Section 3.2, Clojure includes support for run time evaluation of arbitrary code via functions such as *eval*. This severely limits the scope of compiler optimizations. Language features or user code that are unused at compile time must be included in the compiled code because the compiler cannot know what will be evaluated at run time. In addition the compiler itself must be included in the compiled code.

Removing dynamic code evaluation allows code that is unused at compile time to be removed. This should reduce Clojure startup times, because the Clojure runtime needs to set up fewer namespace vars and values on start. In addition removing the code supporting run time evaluation from the compiled code slightly reduces the amount of functionality that needs to be loaded. Both of these changes also correspondingly reduce the sizes of

compiled program packages.

ClojureScript, a dialect of Clojure that compiles to JavaScript, made a similar decision to not support dynamic code evaluation [18]. Removing dynamic evaluation allows the ClojureScript compiler to aggressively remove unused code in order to reduce the resulting code size to a minimum. This is important for ClojureScript because it compiles to JavaScript and larger file sizes increase web site loading times.

4.4.2 Skummet compiler

The Skummet lean compiler is an ahead of time compiler based on the standard JVM Clojure compiler [55]. It was created by Alexander Yakushev in a Google Summer of Code project in the summer of 2014. The Skummet project focuses on using static compilation to improve Clojure performance while preserving as much compatibility with standard Clojure as possible.

Specifically the Skummet compiler "lean compiles" or statically compiles vars, skips macro emission, and removes unused metadata. Lean compilation of vars removes the indirection discussed in Section 3.2. In standard Clojure, the *cons* function is typically called as in the following way:

```
RT.var("clojure.core", "cons").getRawRoot().invoke(args);
```

Under the Skummet compiler the same call is made in this way:

```
clojure.core$cons.invoke(args);
```

Instead of fetching the namespace and var dynamically, both of them are directly referenced. Lean var compilation can also be toggled on or off for specific vars to allow dynamic vars to still be used where necessary.

In addition to lean compilation of vars, the Skummet compiler skips emission of macros, which are only used at compile time. It also improves the removal of unused metadata associated with vars.

The resulting compiler project is able to compile the Clojure runtime and arbitrary Clojure programs. The performance results of the project are analyzed in Chapters 6 and 7.

4.4.3 Oxcart compiler

The second lean compiler project, Oxcart, was created by Reid McKenzie based on the Clojure in Clojure project [38]. The Clojure in Clojure project

aims to replace part or all of the Clojure and ClojureScript compilers written in Java and JavaScript with Clojure equivalents. Oxcart was developed as a Google Summer of Code project over the summer of 2014 with similar goals as the Skummet compiler.

Specifically the Oxcart compiler had goals of eliminating indirect function calls via vars and avoiding compiling Clojure functions to individual classes in most cases to avoid the class overhead [39]. Additional goals for Oxcart include "tree shaking" to remove vars that are not used at runtime from compile program code and inlining of functions [38].

As of the end of the Google of Code project in August 2014 the Oxcart compiler is able to compile a limited subset of Clojure programs. The compiler statically compiles vars, performs reach analysis to remove unused dependencies at compile time, and removes support for dynamic compilation and evaluation [40].

The Oxcart compiler is not able to compile the Clojure runtime, however, nor many typical Clojure programs. This is due to lack of time to complete the necessary features and not necessarily a fundamental limitation. In particular Oxcart programs cannot extend Java classes. This renders many comparisons on Android meaningless, as typical Android app development depends heavily on extending Android library classes written in Java.

Chapter 5

Related Work

The Clojure language has not been studied extensively in the literature. The author is aware of only two studies of Clojure performance, which are described in Section 5.1. However, as Clojure is a JVM language, benchmarking Clojure performance touches on many of the same issues as other JVM languages. These are discussed in Section 5.2. Android also introduces its own challenges for benchmarking as described in Section 5.3. Finally, studies of the performance of the Java language itself demonstrate many of the challenges of benchmarking virtual machine programs as is discussed in Section 5.4.

5.1 Clojure

Clojure language performance has received little study and there are no established benchmark suites for Clojure. The closest to a benchmark suite available for Clojure is the Computer Language Benchmarks Game [20]. The Computer Language Benchmarks Game provides implementations of specific algorithmic benchmarks in numerous languages including Clojure. A set of these benchmarks for Clojure is maintained by Andy Fingerhut [17].

The author is aware of only two studies of note that analyze Clojure performance. Both of them rely on benchmark programs from the Computer Language Benchmarks Game. Sarimbekov et al. [51] developed a suite of metrics for characterizing dynamic JVM languages and a toolchain for collecting the metrics. The metrics exercise the differences between Java and non-Java JVM languages and include metrics related to immutability and object lifetimes. In [50] they use the metrics and toolchain to characterize and compare the behavior of programs written in Clojure, Python, Ruby, and Java. The study uses ten programs taken from the Computer Language

Benchmarks Game, supplemented by a real-world program for each language. Among other results, they found that 77.5% of Clojure classes are immutable compared to 58.8% of Java classes. Clojure also performed the largest percentage of unnecessary zeroing of the studied languages. Unnecessary zeroing occurs when a field is unnecessarily set to the value that it already defaults to.

Li, White, and Singer perform similar experiments to analyze the differences between the dynamic behavior of JVM languages including Java, Clojure, JRuby, Jython, and Scala [36]. As in [50], Li et al. use benchmark programs from the Computer Language Benchmarks Game and additional real-world applications written in each language. They used both static and dynamic analysis to collect metrics such as the percent of Java code executed, method hotness, and objects allocated. They found that Clojure objects have longer lifetimes than Java objects and these objects tend to be smaller. Clojure used a large number of slower boxed primitives, even though faster direct primitives can be used in Clojure. Scala, by contrast, requires the use of boxed primitives but at run time used much fewer boxed primitives than Clojure. They also found that non-Java JVM languages including Clojure rely heavily on Java code for significant parts of their language runtimes and libraries. This implies that Clojure and Clojure on Android performance also depend heavily on the underlying Java library implementations.

Both the Sarimbekov et al. [50] and Li, White, and Singer [36] studies use relatively specific metrics to characterize language performance. This is helpful for optimizing programming languages but does not provide a comparison of overall language performance. The results of the Computer Language Benchmarks Game itself suggest that Clojure execution and memory performance will be within a factor of four of Java performance, with typically values within a factor of two [20]. Notably lacking for Clojure is a set of macrobenchmarks such as the Da Capo benchmark suite for Java [52] and Scala [53].

5.2 JVM languages

There are many languages other than Clojure that compile to JVM-compatible bytecode. Some of the more popular JVM languages are Scala [47], Groovy [48], and the JVM implementations of Ruby [3] and Python [4].

While a statically-typed language, Scala is similar in many ways to Clojure with its emphasis on immutable data structures and higher-order functions. Besides the Li et al. [36] study mentioned previously, a few studies compare the performance of the Scala language with Java and other lan-

guages. Sewe, Mezini, Sarimbekov, and Binder designed a benchmark suite for Scala comparable to the Da Capo benchmark suite for Java [52]. This benchmark suite is a set of real world programs written in Scala and is designed to characterize the performance of Scala programs. They subsequently use this suite to compare memory behavior of Scala and Java programs [53]. Among other findings, they found that Scala generally has shorter-lived and smaller objects than Java and a larger proportion of immutable objects.

Hundt in [30] compare C++, Java, Scala, and Go performance on a single, well-defined benchmark utilizing higher level data structures and several different algorithms. Here Scala used about half as much memory as Java and took between 1/6 and 1 times as long to execute. Binary package sizes for Scala were between twice and three times as large as Java packages. They also note the difficulties in characterizing the virtual machine languages Java and Scala, for which garbage collection parameter settings have a large effect on benchmark performance.

5.3 Android

There are few studies of JVM language performance on Android, but several comparisons of benchmark suites to actual Android app performance.

Nurminen and Denti examine the performance of Scala on the Android operating system and found that it used more energy and less memory than Java while having a larger package size [13].

Gutierrez et. al in [26] studied the characteristics of Android applications and compared them to SPEC CPU2006 benchmarks. They found that their interactive Android applications spent a large portion of time in shared libraries and some fraction in operating system code while the SPEC CPU2006 benchmarks spent almost all of their time only in user code. This suggests that the SPEC CPU2006 benchmarks poorly model interactive Android apps.

In [46] Pandiyan, Lee, and Wu analyze the performance of Android applications at the microarchitectural level. They note that smart phones are typically used for short-term actions followed by long idle periods, that computation can be off-loaded into the cloud, and that applications need to be interactive to handle bursts of data in short periods.

In a study comparing Dalvik and JVM performance [45], Oh, Kim, Choi, and Moon found similarly that real Android apps spend more time running kernel and library code than user code. This was not the case for the comparison EEMBCC embedded Java benchmarks, which spent most of their time executing benchmark code. They further remark that actual Android apps tend to execute methods only hundreds of thousands of times per second

while benchmark methods were called millions of times per second.

The aforementioned studies [26], [46], and [45] all suggest that current benchmark suites are unsuitable for modeling modern interactive Android applications. The studies [46] and [45] also show that mobile applications depend heavily on operating system and library functions. In addition, about one quarter of Clojure language instructions are actually Java code [36]. The combination of these factors suggests that existing benchmark studies are likely to poorly represent actual Clojure applications and that Clojure run time performance depends significantly on underlying Java, virtual machine, and operating system library performance.

5.4 Java

Commonly used benchmarks for evaluating Java performance include SPECjvm98 [6], DaCapo [8], SPECjvm2008 [1], and Java Grande [12]. SPECjvm98 is a set of seven benchmarks for computing performance metrics, five of which are derived from real programs. The DaCapo benchmarks were developed in response to issues found with the SPECjvm98 benchmarks. In particular, they found that typical Java programs had different memory behavior than the SPECjvm98 programs. They provide larger and more complex benchmarks for general purpose Java benchmarking. SPECjvm2008 updates the SPECjvm98 benchmarks with support for multithreading and server systems and with additional startup benchmarks. The SPECjvm benchmarks explicitly have low dependence on file and network I/O. The Java Grande benchmarks, by contrast, were developed to study the performance of Java programs that use large amounts of processing power, I/O, network bandwidth or memory.

Computer system performance benchmarking is notoriously difficult and virtual machines add additional challenges. Time may be spent within application code, system libraries, the underlying runtime, dynamic compilation, native libraries, or operating system services. Performance can vary markedly based on the hardware, virtual machine, system parameters, and workloads being executed. Mytkowicz, Diwan, Hauswirth, and Sweeney found measurement bias to be common and significant, and that introducing presumably insignificant changes to an experimental setup can markedly change the results [42]. Benchmarking virtual machine programs adds additional challenges, with additional non-determinism introduced by just-in-time compilation behavior, thread scheduling, and garbage collection [21].

Existing benchmarks designed to overcome many of these problems can still be difficult to apply and interpret. Zaparanuks and Hauswirth in [56]

found that the SPECjvm98 and DaCapo benchmarks do not accurately represent the workloads used by interactive Java programs. The DaCapo benchmarks, for instance, explicitly exclude GUI programs because of the complexities of benchmarking. Zaparanuks and Hauswirth found that actual GUI programs behave quite differently than the DaCapo benchmarks and thus the DaCapo benchmarks are poor indicators of interactive program performance.

Eeckhout, Georges, and Bosschere in [14] found that for the SPECjvm98 benchmarks, runs with small input sizes were mostly dependent on the virtual machine while runs with larger inputs depended either on the virtual machine or the benchmark program, depending on the program. Pinpointing the source of performance differences requires differentiating between these factors.

Chapter 6

Clojure Startup Performance

As discussed in Section 5, there are very few studies of Clojure performance and none that the author is aware of that characterize Clojure performance on Android. In this chapter we benchmark and profile the startup process of Clojure on the desktop and Android. The following chapter, Chapter 7, presents further experiments focused on run time performance of Clojure on Android across a wider variety benchmarks.

6.1 Experimental setup

6.1.1 Goal

The startup time of Clojure on Android apps is recognized within the Clojure community as an issue. However, there are few reliable measurements available of this startup time and the reasons for slow startup are inadequately understood. The goals of the experiments in this section are to determine the minimum startup time of a Clojure on Android app and identify the main factors contributing to the startup performance.

Minimum startup time is chosen as a target metric because, as discussed in Section 4.3.4, even the most trivial Clojure program loads and prepares the entire Clojure runtime. Any Android apps developed with Clojure must thus consume a minimum amount of time performing this work.

Startup time measurements are also taken with the same programs compiled with the lean Clojure benchmarks in order to gauge their effects on performance. Comparable Java measurements are taken for reference. The startup measurements are repeated on the desktop JVM. The JVM experiments are included for several reasons. First, it allows us to determine whether the problem is specific to Android or if it appears also on the JVM.

Second, the desktop JVM has higher quality profiling tools which allow us to get more detailed information. For instance, neither the Dalvik VM [9] nor the newer Android Run Time supports the Java Virtual Machine Tool Interface, which is used by many profiling tools to inspect the runtime state of the virtual machine. Third, a comparison with Clojure on the desktop can provide insight into the problems that are particular to the Android environment.

6.1.2 Methodology

On Android, we run a benchmark program twenty times without profiling and twenty times with logging of class loading events. On the desktop, we run a benchmark program twenty times each without profiling, with class load event logging, and with the YourKit profiler attached.

Runs without profiling are averaged to estimate the minimum startup time of a Clojure app both on Android and the desktop. Our benchmark program is a trivial program that prints "Hello world". While a Clojure program could be written that performs even less work, it is unlikely that a *useful* Clojure program would do less work. In addition, even this trivial program will load and prepare the entire Clojure runtime. The process of loading the Clojure runtime is the same for every Clojure app as discussed in Chapter 4.

Runs with class load event logging are averaged to estimate the amount of time consumed by various startup tasks. Logging class loading events is useful because the Clojure runtime is loaded in the same sequence on every program execution. By observing when different classes are loaded, we can estimate the time breakdown between the virtual machine, the Clojure runtime, and the main task of the program.

On the desktop we also perform the same number of runs with the YourKit profiler attached. The run times are averaged to estimate the overhead of the profiler. The profiler uses method sampling at 20 ms intervals to estimate the amount of time spent in each method. We use this information to analyze the results and provide a rough breakdown of time spent loading classes overall and loading specific classes.

6.1.3 Test programs

The desktop JVM test programs are shown in Figure 6.1 and 6.2 for Java and Clojure, respectively. On the JVM a minimal program can be simply a class that prints "Hello world" in Java and a namespace that compiles to a class that does the same in Clojure.

On Android, the most trivial "real world" program is still an app. It must be packaged and installed as an APK file, runnable from the app list, and display a main Activity view. This means it has a class that extends "android.app.Activity" and constructs the main view. It is also packaged into an APK using the Android tools as a standard app. This introduces a certain amount of overhead when compared to the JVM programs. When the app is executed, it is fetched, loaded, and run within the context of the Android framework. The Java "Hello world" app is shown in Figure 6.3. The Clojure version in Figure 6.4 is a straightforward translation of the Java version.

```
public class Hello {
    public static void main(String[] args){
        System.out.println("Hello world");
    }
}
```

Figure 6.1: Profiled Java program

```
(ns hello.core
    (:gen-class))

(defn -main [& args]
    (println "Hello world"))
```

Figure 6.2: Profiled Clojure program

```
package com.android.helloworldjava;

import android.app.Activity;
import android.os.Bundle;

public class HelloWorld extends Activity
{
    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }
}
```

Figure 6.3: Java on Android Hello World

```
(ns com.android.helloworldminimal.HelloWorld
  (:gen-class
   :extends android.app.Activity
   :exposes-methods {onCreate superOnCreate})
  (:import [android.app Activity]
           [android.os Bundle]))

(defn -onCreate [this #^android.os.Bundle bundle ]
  (.superOnCreate this bundle)
  (.setContentView this com.android.helloworldminimal.R$layout/main))
```

Figure 6.4: Clojure on Android Hello World

	Version (Desktop)	**Version (Mobile)**
Device model	Lenovo Yoga 2 Pro	Nexus 5
Processor	i7-4500U 1.8 GHz (2 cores)	2.3 GHz Krait 400 (4 cores)
Memory	8 GB RAM	2 GB RAM
Operating System	Ubuntu Linux 13.10 64 bit	Android 4.4.2 (SDK 19)
Virtual Machine	Java HotSpot(TM) 64-Bit Server VM (build 24.51-b03, mixed mode)	Dalvik 4.4.4
Runtime	Java(TM) SE Runtime Environment (build 1.7.0_51-b13)	Dalvik
Java Target	1.7.0	1.6
Clojure	1.6.0	1.6.0

Table 6.1: Experimental Setup

6.1.4 Hardware and software

The desktop JVM experiments are performed on a laptop computer and the mobile experiments on a recent mobile phone. The specific details of the respective setups are presented in Table 6.1.

6.1.5 JVM setup

On the JVM, the programs are executed without profiling, with a simple profiling agent, and with the YourKit profiler. Run times are estimated using the elapsed wall times given by the Linux "time" command-line tool.

The profiling agent prints class load event times. As discussed in Section 4.2.1, loading a class is the first step in preparing the class to be used. Our class loading profiling agent prints the current time stamp whenever a

class is loaded. The code for the profiling agent in its entirety is presented in Figure 6.5. The agent is compiled to an executable JAR and given to the JVM as a command line parameter.

```
import java.lang.instrument.*;
import java.security.*;

public class ClassLoadTracer {
    public static void premain(String agentArgs, Instrumentation inst) {
        final java.io.PrintStream out = System.out;

        inst.addTransformer(new ClassFileTransformer() {
            public byte[] transform(ClassLoader loader, String className,
                Class classBeingRedefined, ProtectionDomain protectionDomain
                    ,
                byte[] classfileBuffer) throws IllegalClassFormatException {

                out.print("" + System.currentTimeMillis() + " "
                    + className + " (loaded by "
                    + loader + ")\n");
                return null; // return class unmodified
            }
        });
    }
}
```

Figure 6.5: Class loading profiler agent

Every time the JVM loads a class it first hands the class off to the profiling agent. The profiling agent gets the current time via "System.currentTimeMillis()", prints it, and returns "null" to indicate the class is returned unmodified. This gives a time stamp for every class loading event which we can use to estimate the time devoted to different startup tasks.

We use class loading events as a way to break down the loading time spent on different tasks. As such we are interested in only a few of the class loading events, which are summarized in Figure 6.2. We refer to time between the start of the program and the loading of the first Clojure class, our class "hello.core", as *JVM loading time*. Up until this point none of our code has been executed but only JVM initialization code. The time between the loading of our main class and the loading of "clojure.core__init" we call *Clojure runtime loading*. Once "clojure.core__init" starts being loaded the basic Clojure runtime is already up and running. The Clojure runtime then loads the core Clojure functions. When this is complete we see our program execution starting when our namespace initialization class, 'hello.core__init', is loaded. Finally the last event we see is the loading of 'java.lang.Shutdown$Lock' to shut down the JVM.

In addition to profiling with a simple agent, we also use a profiler to estimate the overall time spent loading classes. We start the program from the

Class Loading Event	Interpretation
hello.core	JVM loaded
clojure.core__init	Clojure runtime loaded
hello.core__init	Clojure core functions loaded
java.lang.Shutdown$Lock	Program execution complete

Table 6.2: JVM Class Loading Interpretation

command line with the YourKit JVM profiler attached and sample program methods being executed at 20 millisecond intervals. From this sampling we estimate the percentage of time consumed by different methods. As this adds a certain amount of overhead to the run process of the program, these are rough numbers and cannot be considered in absolute terms.

More detailed profiling allows us to estimate how much time was devoted to loading and linking JVM classes and where this time was incurred. We collectively estimate total JVM loading and linking time by seeing how much time was spent in the Java method "java.lang.ClassLoader.loadClass". This is the Java method that loads and links new Java bytecode classes into the running JVM. We further decompose this by looking at where the calls to "loadClass" come from.

6.1.6 Dalvik setup

Setup for Dalvik is similar to setup for the JVM, except that we do not perform detailed profiling. We run the test programs without profiling and with small profiling changes introduced into the Dalvik framework.

We modify the Dalvik framework to print the class name and current time to the Android log whenever a class is to be loaded. This allows us to intuit the proportion of time spent on various tasks in the same way as discussed for the JVM. Program start time is measured beginning when Android prints a start message for the application activity. The loading of our main class is used as indication that the virtual machine has been loaded. Loading of the Clojure core namespace initializer is taken to indicate that the Clojure runtime has loaded. Loading of our program's namespace initializer class file indicates that Clojure core functions have been loaded. The final event is when the Android ActivityManager prints a message indicating the activity has been displayed. These event interpretations are summarized in Figure 6.3.

Time measurements come from the Android log timestamps. Times are logged in milliseconds, have precision only up to tens of milliseconds, and are

Log Message	Interpretation
START HelloWorld	JVM loaded
Load clojure.core__init	Clojure runtime loaded
Load HelloWorld__init	Clojure core functions loaded
Displayed HelloWorld	Program execution complete

Table 6.3: Interpretation of Class Loading

likely accurate to less than that.

6.1.7 Skummet and Oxcart

The exact same Clojure programs are also compiled and run with the lean compilers Skummet and Oxcart on the JVM and Skummet on Dalvik. Oxcart tests are omitted on Dalvik due to lack of support for extending Java classes. Without support for extending Java classes, a complete Android app cannot be implemented using Oxcart alone because an Android app needs to have a main view class that extends "android.app.Activity". See also Section 4.4.3 for discussion of the limitations of the Oxcart project.

The Oxcart tests are performed using Oxcart-compiled code but the standard Clojure compiler, as the Oxcart compiler does not currently support compiling the Clojure runtime.

6.2 Results

The program is executed twenty times without profiling and with class load event logging on the JVM and Dalvik. The results are averaged over the twenty runs. The average run times of all programs run without profiling are presented in Figure 6.6.

6.2.1 Clojure

Figures 6.7 and 6.8 show the breakdown of Clojure startup time on the JVM and Dalvik as described in the experimental setup.

For Clojure on the JVM, we estimate the proportion of time spent loading classes (Figure 6.9) and the largest sources for class loading calls using the YourKit profiler results (Figure 6.10).

The method sampling used to get this breakdown adds a large amount of overhead to the startup process while printing class loading timestamps added little overhead. This can be be seen in the Clojure JVM overhead

diagram in Figure 6.11. The profiling technique used in Dalvik added a smaller amount of overhead, as seen in Figure 6.12.

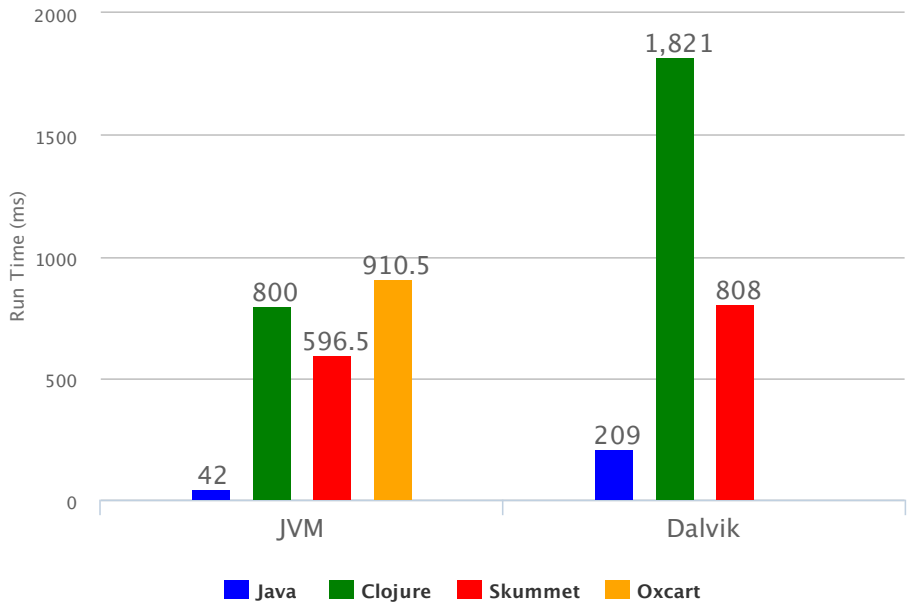


Figure 6.6: Average Run Time for Minimal Java and Clojure Programs

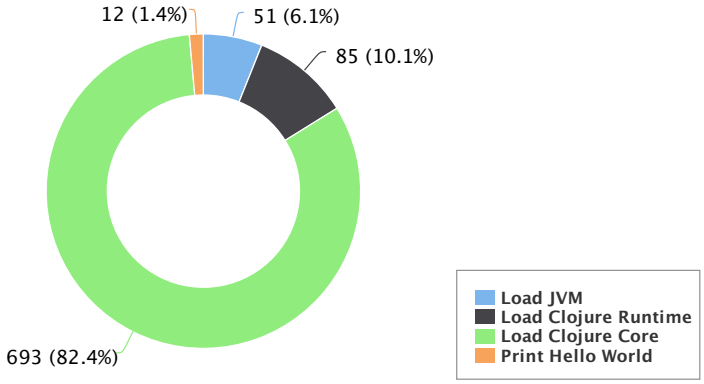


Figure 6.7: Clojure JVM Bootstrapping Times (ms)

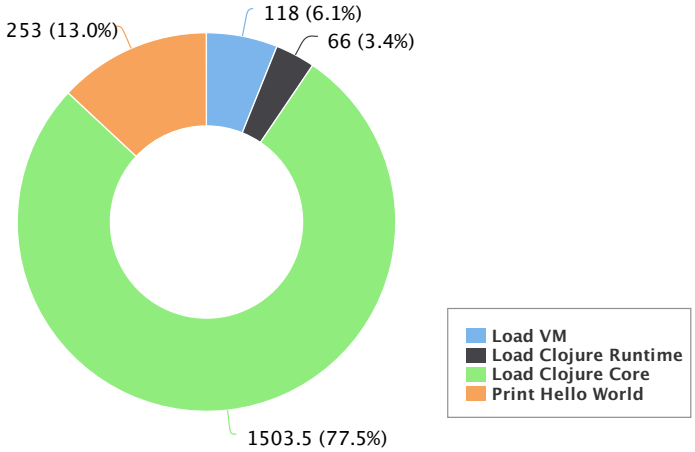


Figure 6.8: Clojure Dalvik Bootstrapping Times (ms)

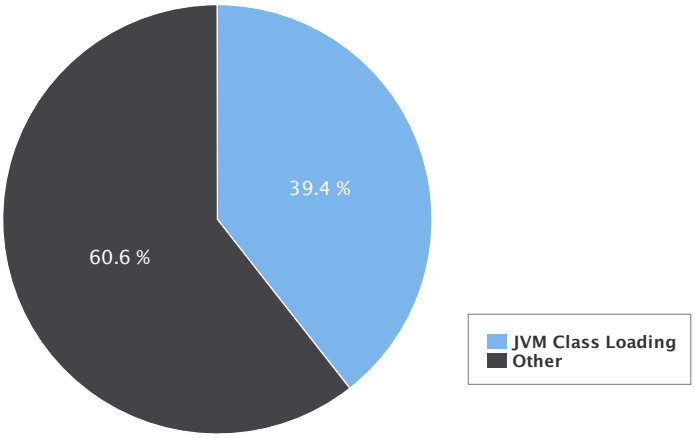


Figure 6.9: Clojure JVM Class Loading vs Total Run Time

6.2.2 Skummet and Oxcart

The bootstrapping breakdowns for Skummet and Oxcart on the JVM and Dalvik are presented in Figures 6.13, 6.14, and 6.15.

On the JVM, printing class loading timestamps introduced about 14% overhead for Skummet and 4% for Oxcart. On Dalvik, printing class loading timestamps introduced about 17% overhead for Skummet app runs.

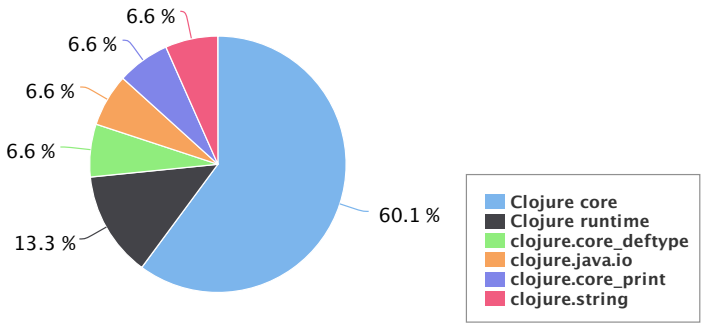


Figure 6.10: Clojure JVM Class Loading Breakdown

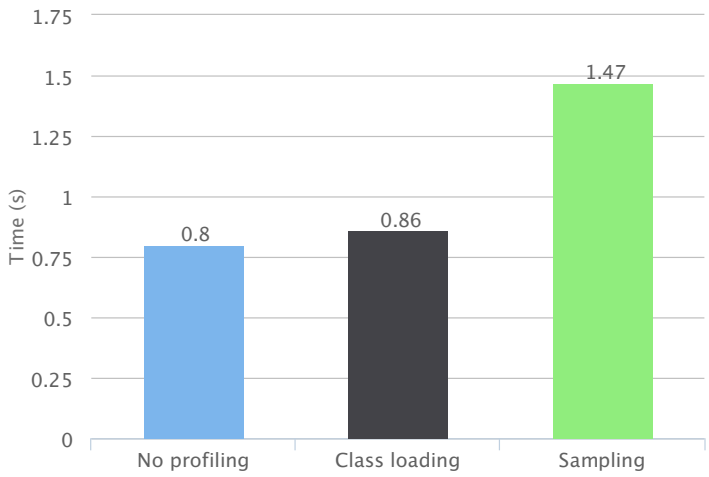


Figure 6.11: Clojure JVM profiling overhead

6.3 Analysis

6.3.1 Clojure

Minimal program execution for Clojure took about 19 and 9 times as long as corresponding Java programs on the JVM and Android Dalvik, respectively (Figure 6.6). On the JVM a minimal Clojure program took nearly a second to execute. On Dalvik this was close to two seconds. In all cases these are

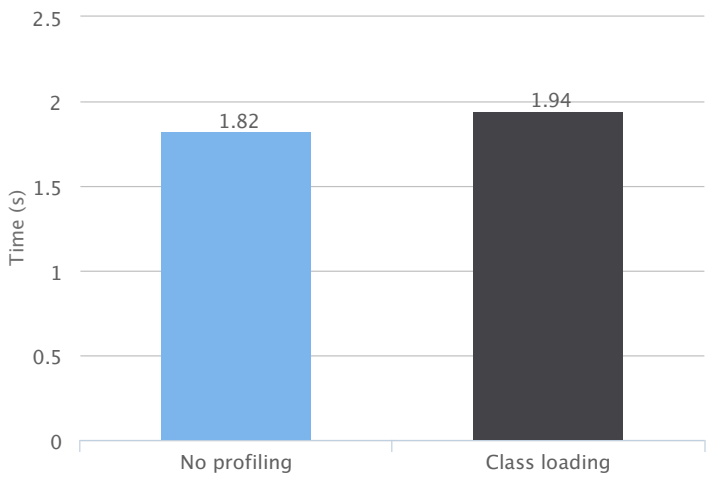


Figure 6.12: Clojure Dalvik profiling overhead

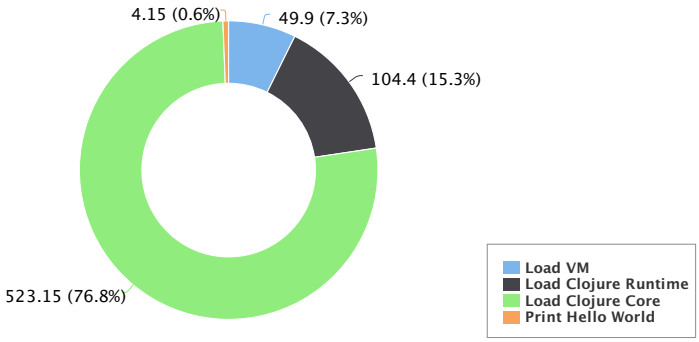


Figure 6.13: Skummet JVM Bootstrapping Times (ms)

minimal programs. Execution times for larger, actual programs are likely to be longer.

On both the JVM and Dalvik the largest proportion of time, about 80% in both cases, was consumed loading the Clojure core library (Figures 6.7 and 6.8). As described in Chapter 4, the Clojure core library defines the functions of the Clojure language and is loaded before any user code is executed.

Loading the virtual machine itself consumes only about 6% of the run time in both environments. This contradicts the seemingly widespread idea that Clojure startup time is long because of JVM startup time. This time is also longer than the total time to execute trivial Java programs, which

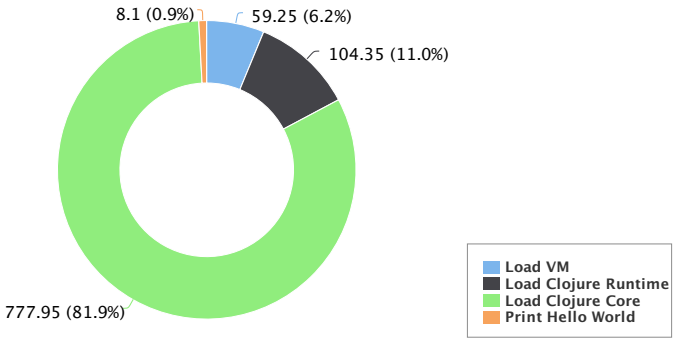


Figure 6.14: Oxcart JVM Bootstrapping Times (ms)

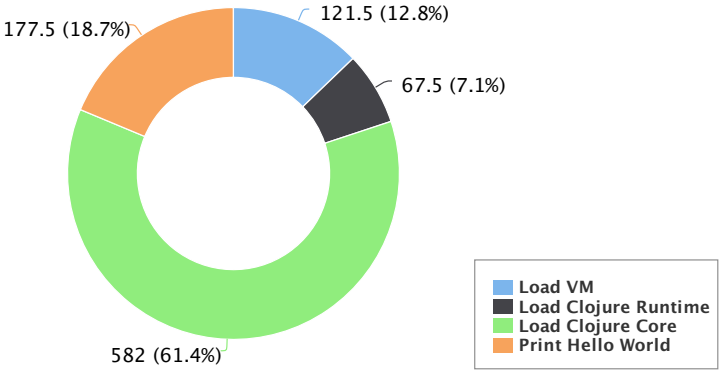


Figure 6.15: Skummet Dalvik Bootstrapping Times (ms)

suggests that executing our larger Clojure programs also affects what we have defined as virtual machine start time. In absolute terms more time is spent here on Dalvik than on the JVM, but the Dalvik program is also doing more work. On Dalvik this also includes the time necessary to fetch and start the Android app through the Android framework.

The time labeled "Load Clojure Runtime" consumes about 10% of JVM execution time and 3% of Dalvik execution time. The actual numbers are very similar, with 85 milliseconds on the JVM and 66 milliseconds on Android. Android may be able to shave off time by virtue of loading fewer class files.

The most notable difference in JVM and Dalvik run times is in the execution of the program itself. This is due primarily to the test setup. The JVM test programs are executed by calling the main class directly. There is very little overhead. The Android test programs are loaded and displayed through the Android framework. They load and display a graphical user interface with a text field, while the JVM test program simply prints text to standard output.

On the desktop JVM about 40% of startup time is consumed by loading and linking program classes when profiling via method sampling. This is shown in Figure 6.9. The majority of these classes, about 60%, are loaded directly when initializing the Clojure core namespace (Figure 6.10). A significant fraction are also loaded during the loading of the Clojure runtime before loading the core namespace.

The remainder of class loading time is divided between four large external namespaces loaded during the core namespace initialization: clojure.core_deftype, clojure.java.io, clojure.core_print, and clojure.string. These provide functions for defining extendable data types and protocols, handling input and output, printing output, and handling strings, respectively.

Printing class loading timestamps only added about 7.5% and 6.6% overhead on the JVM and Dalvik, respectively, as shown in Figures 6.11 and 6.12. Profiling with sampling added much more overhead (about 84%).

6.3.2 Skummet and Oxcart

Skummet significantly outperformed Clojure on both the JVM and Dalvik in these benchmarks, while Oxcart performed slightly worse than the standard compiler on the JVM (Figure 6.6). The programs compiled and run with Skummet take about 75% and 44% as long to execute as their standard Clojure counterparts.

By profiling the Skummet program execution using the YourKit Profiler method tracing it can be seen that the improvement in startup time is almost exclusively due to changes to how the Clojure core namespace is loaded. Where standard Clojure core namespace initialization is broken out into 25 different classes the Skummet runtime uses only three. Even more significantly the Skummet runtime takes about one fourth of the time of the standard Clojure runtime to load the core namespace classes themselves.

The Oxcart-compiled program performed slightly worse than the standard Clojure compiler. This is not a very meaningful difference. Most of the run time is consumed in loading the Clojure runtime, which as discussed earlier is the same Clojure runtime used in the Clojure tests. As it uses the same runtime, the performance can be expected to be very similar for this

trivial program test. A better test would take advantage of the tree-shaking features of the Oxcart compiler and use larger programs. These tests were not performed because Oxcart can compile only a limited subset of Clojure programs.

6.4 Conclusions

On both the desktop and Android Clojure programs take significantly longer to begin executing user code than comparable Java programs. Clojure program startup shows the same pattern on both the JVM and Dalvik. The large majority of time, about 80% in both cases, is consumed loading the core Clojure functions. On the JVM it can be seen that much of this time is spent loading the binary class files themselves before executing any code from the files.

Dramatic improvements are necessary to achieve Java-like startup times. In these experiments the Skummet compiler reduced startup times by about a quarter on the desktop and more than half on Android. This is a large improvement but still leaves a significant startup performance gap between Clojure and Java.

Chapter 7

Clojure Execution Performance

In Section 6 we test and analyze the startup performance of Clojure and the lean Clojure projects on Android Dalvik and the desktop JVM. The experiments determine the minimum amount of time needed to run a Clojure on Android app and profile the startup process. The results show that Clojure apps start much more slowly than standard Java apps, with the Skummet lean Clojure compiler reducing this time significantly.

But startup time is only one aspect in analyzing a programming language's performance. Other important aspects include run time speed, memory usage, and package sizes. The previous experiments also do not tell whether startup time varies significantly for different programs.

In this section we benchmark a selection of programs compiled with Java, Clojure, and Skummet for Android in order to answer some of these questions. The goals of these experiments are to compare the execution performance of Java, Clojure, and Skummet across a sampling of benchmarks and to gain additional insight into the startup performance of Clojure across different programs.

7.1 Experimental setup

7.1.1 Methodology

We created seven different Android apps and benchmark their startup and run times. The benchmarks are executed on two devices, the Nexus 5 phone and the Nexus 7 tablet, using the Dalvik and ART runtimes on each for four sets of executions total. Each benchmark runs as a normal Android app, logs the time when user code from the main activity is first executed, executes a benchmark-specific task, and completes. We execute each benchmark thirty

times and record the *startup* time before user code is executed and the *task* time taken to execute the benchmark task.

The *hello* benchmark is almost identical to the Android "hello world" app from Chapter 6. The only change made is to log the timestamp when user code is first executed for consistency with the other benchmarks.

The *dependencies* benchmark performs trivial data transformation and JSON parsing using the Reactive Extensions libraries and the Transit libraries. The goal of the *dependencies* benchmark is to help see how the startup time of Clojure and Java programs change when larger, real-world dependencies are pulled into the program. The Reactive Extensions and Transit libraries were chosen because they perform useful functions for a client-side application and have libraries for both Clojure and Java. The Reactive Extensions libraries, RxJava and RxClojure, are libraries originally developed by Microsoft for composing asynchronous events [41]. They are commonly used in Android app development to handle the concurrent and asynchronous program flow. Transit is a format and set of libraries for transferring data. It was recently released by Cognitect, the developers of the Clojure language, with libraries and bindings for several languages including Clojure and Java [29].

The remaining five benchmarks are based on programs from the Computer Language Benchmarks Game [20]. As discussed in Section 5.1, the Computer Language Benchmarks game is the only benchmark suite available for Clojure. The Computer Language Benchmarks Game programs each perform a different algorithmic task. Programs in different languages are encouraged to be written in the idiomatic style of the language and using the same algorithm to provide some level of comparability. Even with this restriction, however, there remains a large amount of flexibility in the ways the programs can be written.

We selected five of these benchmark programs from the collection provided by Andy Fingerhut [17]. Each of these five benchmark programs has been written in both Java and Clojure by Andy Fingerhut or his sources. In each case we have selected the newest available version of the program where possible. This is typically the fastest version of the program. There are no other modifications to the task programs themselves except to change their namespaces. The selected programs and their respective workloads are listed in Figure 7.1.

The Computer Language Benchmarks Game programs are wrapped in an Android activity in order to be run as normal apps. For example, the activity wrapper for the *fannkuch-redux* benchmark is shown for Java in Figure 7.1 and for Clojure and Skummet in Figure 7.2.

Benchmark inputs are chosen to increase the app run time while keeping

Benchmark	Workload
hello	Prints hello world
dependencies	Trivial data manipulation via external libraries
binary-trees	Tree depth 9
fannkuch-redux	Sequence length 8
n-body	50000 steps
pidigits	1000 digits of pi
spectral-norm	1000 approximations

Table 7.1: Benchmark Programs

```
package benchmark;

import android.app.Activity;
import android.os.Bundle;
import android.util.Log;

public class MainActivity extends Activity
{
    @Override
    public void onCreate(Bundle savedInstanceState){
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        Log.d("benchmark-fannkuchreduxJava", "onCreate");
        fannkuchredux.main(new String[]{"8"});
    }
}
```

Figure 7.1: Main Activity for fannkuch-redux benchmark

```
(ns benchmark.MainActivity
  (:gen-class
   :extends android.app.Activity
   :exposes-methods {onCreate superOnCreate})
  (:require [benchmark.fannkuchredux])
  (:import [android.app Activity]
           [android.os Bundle]
           [android.util Log]))

(defn -onCreate [this #^android.os.Bundle bundle ]
  (.superOnCreate this bundle)
  (.setContentView this benchmark.R$layout/main)
  (Log/d "benchmark-fannkuchreduxClojure" "onCreate")
  (benchmark.fannkuchredux/-main "8"))
```

Figure 7.2: Clojure on Android Hello World

the entire app run time under ten seconds. On the test devices if an app takes longer than about ten seconds to start, blocking the UI thread during

this time, the process will time out and the app will not run to completion.

The Clojure and Skummet benchmark apps are compiled from the exact same source code. Java apps have different source code that performs the same task using the same general algorithm.

Each benchmark program is executed thirty times in order to compute average startup and task times. Before each set of executions for a given benchmark program the device is restarted. This is to reduce the impact of other processes in memory on the test results. There is a delay of two minutes after restarting to allow all system initialization to complete before running the programs and a short delay between individual program executions.

From each program execution we obtain three timestamps indicating when the program was called to start, when the "onCreate" print statement is reached, and when the program has run to completion. We define the period between the first two events as the *startup time* and the period between the latter two events the *task time*.

7.1.2 Hardware and software

The benchmarks are executed on two different devices, the Nexus 5 phone and Nexus 7 tablet, for two different operating system versions each. This makes a total of four repetitions, which we refer to by device and virtual machine name as Nexus 5 Dalvik, Nexus 5 ART, Nexus 7 Dalvik, and Nexus 7 ART. Hardware and software setup details are listed in Figure 7.2. It should be noted that the ART tests for the two devices use different versions of Android. This is because each device supports only one specific version of Android that is compatible with ART. The Clojure ART tests also use a different version of the Clojure compiler than the Dalvik tests, as this was the only version of Clojure compatible with the ART runtime.

7.2 Results

The benchmarks were executed thirty times on each of the four different environments: Nexus 5 Dalvik, Nexus 5 ART, Nexus 7 Dalvik, and Nexus 7 ART. The averaged run times, divided into startup time and task time, are presented in Figure 7.3.

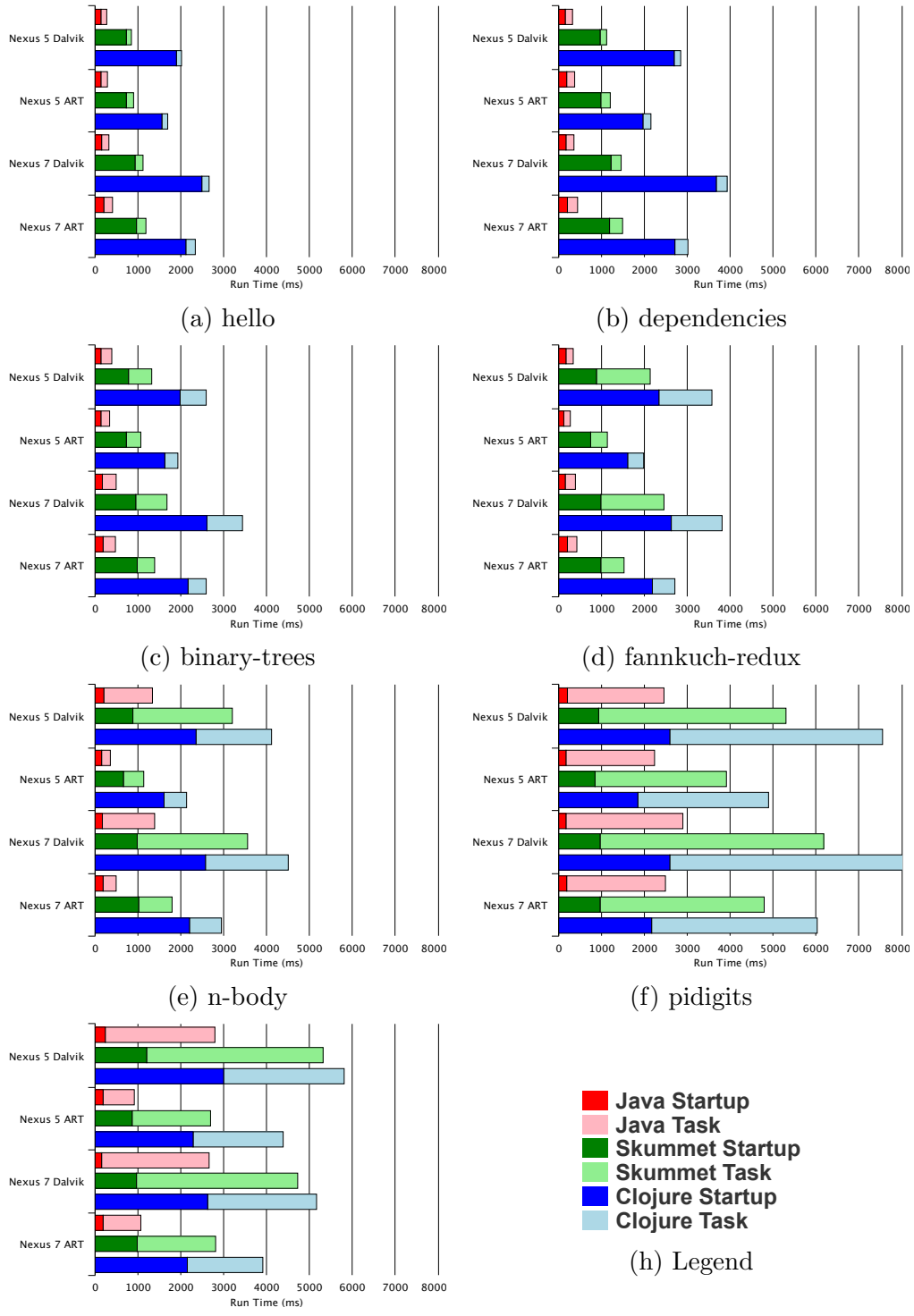
The startup times from the same benchmark runs are presented alone in Figure 7.4 for each platform setup. The sizes of the packaged Android apps for each compiler before installation are shown in Figure 7.3.

	Nexus 5		**Nexus 7**	
Tool	**Dalvik**	**ART**	**Dalvik**	**ART**
Android version	4.4.4	5.0.1	4.4.4	5.0.2
Clojure version	1.6.0	1.7.0-alpha4	1.6.0	1.7.0-alpha4
Skummet version	1.7.0-skummet-SNAPSHOT			
Java version	1.7.0_72			
Android build tools	19.0.1			
Processor	Krait 400 (quad-core)		Krait 300 (quad-core)	
Processor speed	2.26 GHz		1.5 GHz	
RAM	2 GB LPDDR3-1600		2 GB DDR3L	

Table 7.2: Hardware and software

	Java	**Clojure 1.6.0**	**Clojure 1.7.0-alpha**	**Skummet**
hello	12	1352	1515	908
dependencies	1464	2889	3047	2416
binary-trees	13	1352	1520	912
fannkuch-redux	14	1358	1522	912
n-body	14	1360	1524	915
pidigits	13	1357	1519	911
spectral-norm	14	1359	1521	912

Table 7.3: Benchmark App Sizes (KB)



(a) hello

(b) dependencies

(c) binary-trees

(d) fannkuch-redux

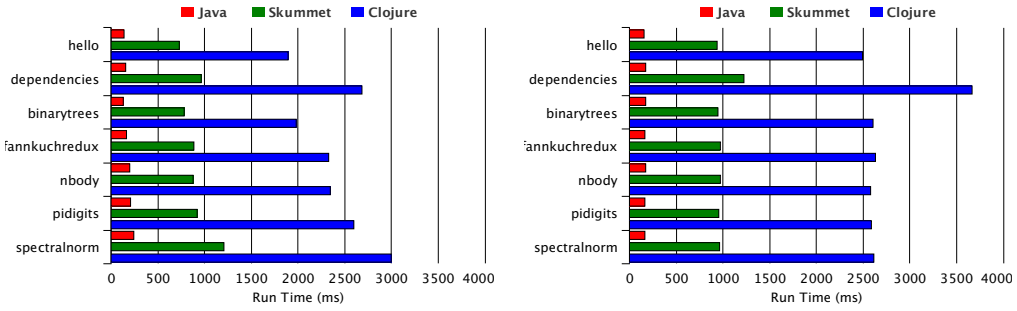
(e) n-body

(f) pidigits

(g) spectral-norm

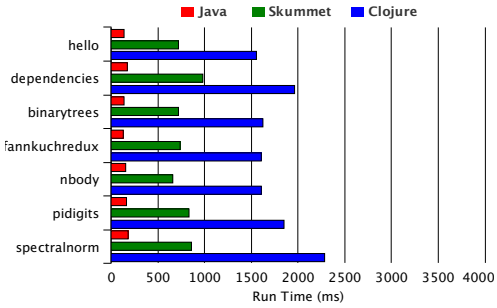
(h) Legend

Figure 7.3: Benchmark results

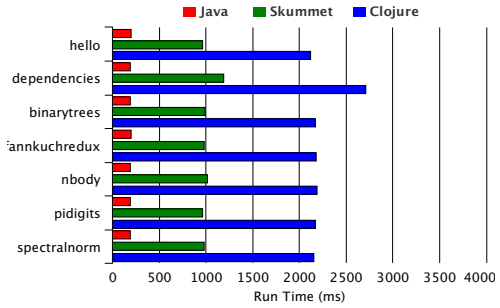


(a) Nexus 5 Dalvik

(b) Nexus 7 Dalvik



(c) Nexus 5 ART



(d) Nexus 7 ART

Figure 7.4: Benchmark startup times

7.3 Analysis

7.3.1 Overall

Java outperformed Clojure and Skummet by a large margin on all tests, with overall run times between 2 and 12 times faster. This is mostly due to long Clojure startup times, as times were closer for longer tests less dependent on startup times.

Skummet overall run times were lower than Clojure across all benchmarks and environments. Skummet run times ranged from just under half as long as Clojure in benchmarks dominated by startup time to values very close to Clojure in benchmarks with longer task times.

The relative performances of the three languages were largely unaffected by different devices and Android versions.

7.3.2 Startup

Two patterns can be seen in the startup time results of Figure 7.4: Skummet startup times are significantly lower than Clojure startup times and Clojure and Skummet startup times scale upwards relatively quickly with more dependencies.

Skummet benchmarks started in about half the time of identical Clojure benchmarks across all test runs. The most dramatic difference can be seen in the benchmark with in most cases the longest startup time, the *dependencies* benchmark. In this benchmark Clojure started in between 2 and 3.5 seconds while Skummet started in around 1 to 1.2 seconds. The reduction in startup time was consistent across devices and Android versions.

The upward scaling of startup times for programs with more dependencies can be seen most clearly in the comparison of the *hello* and *dependencies* benchmark results. The *dependencies* benchmark differs from the *hello* benchmark in including two external library dependencies and performing a trivial task with the libraries. The task time itself is insignificant, as can be seen in Figure 7.3. Startup times, however, increase by 20 to 50% between the *hello* and *dependencies* benchmarks for Clojure and Skummet.

The *dependencies* test increases startup times because it adds additional libraries that are loaded by the Clojure runtime when the program starts. The Clojure runtime performs work in loading all of the code from referenced library namespaces even if only a few functions from the libraries are used. One reason for this is that the Clojure compiler compiles each source code form sequentially in a single pass. If a form is found that loads another namespace, the namespace is immediately loaded and executed. With only a

single pass, the compiler must assume that any referenced dependencies will be used later and there is no opportunity to remove unused dependencies. This is also discussed in Section 4.3.1.

The Skummet compiler works in the same way in this respect. This can be seen in the results, as the Skummet startup times scale upwards in a similar manner to Clojure startup times.

The Java *dependencies* benchmark had slightly higher startup times but the increase was less than for Clojure and Skummet. The Java compiler removes unused class dependencies, but does not remove unused methods from a given class. In this case we are only using a fraction of the methods in the referenced classes, but the whole classes must be loaded. The increase in startup time is less than in Clojure, where entire namespaces, which are typically larger than Java classes, must be loaded if a single entity from the namespace is used.

The Nexus 5 tests saw more variation in startup times, with a similar pattern exhibited by all three compilers. On the Nexus 5 in most cases the *spectral-norm* benchmark ended up take longer than even the *dependencies* benchmark. The reason for this variation is unclear. It may be that the benchmark programs are executed non-linearly on the Nexus 5, with the task execution beginning before the startup time is complete.

7.3.3 Dalvik versus ART

The newer ART runtime in the Android 5.0.1 and 5.0.2 tests generally improved the speed of the benchmarks, although the effect is inconsistent. The *n-body* benchmark, for example, saw reductions in total run times of as much as 73% in the Java Nexus 5 tests. On the other hand other benchmarks, such as the *hello* benchmark, saw varying or negligible changes in run times.

7.3.4 Package size

The sizes of compiled Android packages before installation are shown in Figure 7.3. For all but the *dependencies* benchmark, the program source code itself is trivially small and the package size is dominated by the Clojure runtime and libraries. For these benchmarks the Clojure and Skummet packages are two orders of magnitude larger than their Java equivalents. The Java benchmarks have sizes of 12-14 KB compared to between 0.9 and 1.5 MB for the Skummet and Clojure benchmarks.

The difference in size between Java and Clojure/Skummet benchmarks is almost entirely due to inclusion of the Clojure runtime and core libraries in the Clojure and Skummet packages. Java apps do not have this limitation

because the standard Java and Android framework libraries are already included in Android and are loaded by the system when Android starts. For this reason Java apps that do not have library dependencies or significant binary resources can be tiny.

The *dependencies* benchmark, which includes external libraries, presents a different picture. The Skummet and Clojure benchmarks are between 1.7 and 2.1 times larger than the Java counterpart, but the difference is largely due to the inclusion of the Clojure runtime. Removing this from the package size, the resulting packages are very similar to the Java package size in all cases. This is weak evidence to suggest that larger Clojure and Skummet apps may be similar in size to Java apps. However, this is only a single app comparison with a few comparable library dependencies. Use of tools such as ProGuard could also significantly reduce these sizes.

Excluding the *dependencies* benchmark, Skummet packages were about two thirds the size of corresponding Clojure packages. This is mostly a comparison of their respective runtime and core library sizes, as the benchmarks apps themselves have very little code. In the *dependencies* benchmark the Skummet compiler reduced the total package size by about 16% compared to Clojure 1.6.0.

The Clojure 1.7.0 alpha packages were about 12% larger than their Clojure 1.6.0 counterparts. This in and of itself means very little, as the 1.7.0 compiler is at an alpha stage and will likely have significant changes before being released.

7.4 Conclusions

Java outperformed Clojure and Skummet by a wide margin across the benchmarks, with the gap narrowing for longer benchmarks less dependent on startup time. On longer benchmarks Clojure and Skummet performance was within a factor of two of Java performance.

Clojure startup time exceeded 1.5 seconds across all benchmarks and scaled upwards significantly for a benchmark with a few library dependencies. Skummet programs started in about half the time of Clojure programs across all benchmarks. The resulting time remains four to six times longer than comparable Java programs.

The results suggest that Clojure runtime execution performance can be within at least a factor of two of Java performance, while startup time remains at least four to six times longer even with the changes introduced by Skummet. This points to Clojure startup time being a larger problem than execution time in the Android context.

Chapter 8

Discussion

In this chapter we discuss the current state of Clojure on Android and the lean Clojure projects. Based on our experimental results, we speculate about the feasibility of Clojure on Android and future directions for Clojure on Android development.

8.1 The state of Clojure on Android

The experiments of Section 6 and Section 7 had a minimum time of nearly two seconds to start a Clojure on Android app. The experiments were performed on two relatively recent high-end Android devices, which suggests that longer times would be seen on many other devices.

The *dependency* benchmark in Section 7 shows that this startup time scales upwards when more dependencies are added to an app, with the experimental startup time jumping by nearly a second on the Dalvik tests and by about a half a second on ART runtime tests (Figure 7.4). This suggests that startup times for actual apps, which will be larger than our presented benchmarks, will be much longer.

Based on these results, it seems likely that actual apps would have startup times exceeding three seconds and perhaps much higher. This is an unfeasible length of time for many Android apps. Android apps are used in very flexible situations which often demand quick response times. For instance, a user may need to share information quickly from one app to another and then return to the original app after sharing. This situation occurs when sharing a picture on WhatsApp, for instance, or saving a file to DropBox or Google Drive. In other situations by the time the user navigates to the app and waits for it to load the opportunity has passed. This happens, for instance, when a user needs to snap a quick picture or write a note to themselves.

Slow response times in the context of web sites have been shown to lead to user abandonment of the site and corresponding revenue losses [43].

The startup delay can be managed in part by displaying a splash screen to the user while the rest of the app loads in the background. This does not reduce the delay, but it does provide feedback to the user and would probably make it less likely for the user to close the app before it loads.

One workaround would be to manually load required namespaces in the background in a staggered fashion. This is against Clojure general conventions but would push much of the loading delay further back. This delayed loading can only be performed with the developer's code and libraries, however. The Clojure runtime itself would still need to load and loading this runtime would still consume a minimum of about two seconds based on our experiments.

Future hardware and software improvements will decrease these startup times. It seems unlikely, however, for performance improvements to be significant enough to make Clojure viable in the near future. Supposing the performance doubles every two years according to Moore's Law, it will take at least six years for Clojure startup performance to approach the current startup time of Java apps.

The sampling of benchmarks in Section 7 show Clojure run time performance within a factor of two or less of Java on most of the tests. This is similar to the results seen in the Computer Language Benchmarks Game [20] for Clojure on the JVM. As Clojure on the JVM has achieved a certain level of popularity, this performance tradeoff is clearly reasonable for many developers. However, the performance benchmarks are limited in scope and can only provide weak evidence to point towards Clojure performance in general.

Given the current startup performance as discussed, Clojure is currently not a viable language for most professional Android development. Given the scope of the problem, the situation seems unlikely to change without dramatic changes to the Clojure language itself.

8.2 The state of Lean Clojure on Android

Lean Clojure performance

The Skummet lean Clojure compiler makes significant improvements to Clojure startup times in the experiments presented in Chapter 7. In most of the benchmarks, startup times were reduced at least by half from between 1.5 and 3 seconds to around 0.7 to 0.9 seconds. These run times represent the minimum amount of time to start a Skummet Clojure on Android app. Typ-

ical apps on typical devices are likely to have longer times due to increased code size and library dependencies as discussed in the previous section.

This is a large improvement on Clojure on Android startup times but is insufficient for many development purposes. It remains between about four and six times longer than startup times for corresponding Java apps.

One reasonable goal for Clojure startup time might be to bring it to within a factor of two of Java startup time. This would put it under half of a second in the minimal case. Achieving this goal would require an additional reduction of one half to two thirds of the current Skummet startup time.

One way to achieve this reduction might be to reduce the number of items to load by removing unused dependencies at compile time. Achieving a two thirds reduction would require roughly two thirds of the Clojure runtime to be removed as unused on compilation. There is no experimental evidence to support this, but it does not seem far-fetched.

The Oxcart lean Clojure compiler performs this type of dependency shaking, but the experiments presented in this thesis tell little about its impact on startup performance. The Oxcart compiler is unable to compile the Clojure runtime itself, which consumes most of the startup time of a minimal app, so we were unable to gauge its effect on startup times. The Oxcart compiler is also unable to compile many other programs including the benchmarks used in our Android experiments and was thus excluded from those tests as well.

The execution performance experiments in Chapter 7 provide weak evidence to suggest Skummet steady-state performance is similar or slightly worse than standard Clojure. However, the experiments cover only a narrow range of cases and it seems likely that lean Clojure would outperform standard Clojure at run time with further development. The more statically compiled lean Clojure code would provide many opportunities for performance improvements over the dynamic code in standard Clojure.

For the goal of making Clojure on Android startup time viable for general app development, the Skummet lean Clojure project makes large but insufficient gains. If the lean Clojure projects were taken forward to include features such as dependency shaking, it seems likely the resulting code would start up and execute quickly enough to be used on Android.

Lean Clojure tradeoffs

For reducing Clojure startup time, the lean Clojure projects work. However, the "lean" compilation can only compile source code for a subset of the Clojure language. In particular, as discussed in Section 4.4.1, code that depends on dynamic binding of vars and namespaces or dynamic evaluation of code cannot be lean-compiled.

The Skummet compiler handles this issue by allowing some vars to be lean-compiled while others are compiled normally. Lean compilation cannot be used for dynamic, REPL-driven development (as discussed in Section 3.2), but standard compilation could be used for development and lean compilation for production code.

This approach adds additional complexity to the Clojure compiler and ecosystem. Development and production code would be executed in slightly different ways. Lean-compiled and standard-compiled code would differ in subtle ways. Branching of lean-compiled code and dynamic code would introduce additional complexity into the Clojure ecosystem as well, with some libraries able to be lean-compiled and others dependent on standard compilation.

In exchange for this additional complexity, Clojure would perform faster for a subset of Clojure code. As it currently stands, based on the experimental results of this thesis, the benefits are insufficient to balance out the drawbacks. Clojure on Android even with the current changes is not sufficiently performant. With further development this situation could change.

In essence the lean Clojure compiler forks the Clojure language. Another Clojure language fork is ClojureScript. ClojureScript is a Clojure dialect that compiles to JavaScript. It differs from standard Clojure in a few notable ways. In particular, it optimizes ClojureScript code by removing unused code at compile time via the Google Closure compiler. This dead code removal is only possible because run time evaluation of arbitrary code is also removed.

One key way in which ClojureScript differs from lean Clojure is the environments in which it can be run. ClojureScript compiles to JavaScript, not JVM bytecode, and can only be run in JavaScript environments. Lean Clojure, on the other hand, compiles to JVM bytecode and is run on the JVM in the same environments as Clojure code. The separation of environments with ClojureScript mean that compatibility with standard Clojure is less of a concern as the programs are run in different environments. With lean Clojure this is not the case.

8.3 Future directions

Future directions for Clojure on Android include further performance research, additional lean Clojure changes, and alternative approaches using other frameworks or the ClojureScript dialect.

Performance benchmarking

As discussed in Section 5.1, there is little research about Clojure performance. This thesis analyzes and benchmarks Clojure startup performance and provides some hints toward the runtime performance of Clojure on Android. Another important aspect to study would be the memory performance of Clojure on Android. Computer Language Benchmark Game [20] suggests that Clojure consumes more memory than Java but the issue has not been properly studied. In addition, memory performance may be less important on the desktop than on Android, where background apps will be terminated by the system based in part on memory consumption. An Android app that consumes more memory is thus more likely to have to be reloaded every time it is accessed.

Additional lean Clojure changes

There are at least three different ways in which the lean Clojure compiler projects could be continued to further improve Clojure performance.

The first way is by removing unused dependencies at compile time. The amount of improvement that can be expected from this change has not been tested. The Oxcart compiler removes unused dependencies but unfortunately does not compile a large number of valid Clojure programs and thus cannot be used to test this approach. If the Oxcart work was continued so that it was able to compile more programs and importantly the core Clojure libraries the efficacy of unused dependency removal could be tested.

The second way to improve lean Clojure performance further would be to load vars lazily. As described in Section 4, Clojure namespace bootstrapping sets up a var class to point to every function or var value in the namespace. This work is performed when the namespace is loaded. By default this means the work is performed at the beginning of program execution, as Clojure namespaces typically begin with statements that load dependent namespaces. Loading the Clojure core namespace occurs before a single line of user code is executed.

Many of these vars will never be used or are not used immediately after loading the program. Both vars and namespaces could be loaded lazily. Instead of setting up a namespace at the beginning of a program, the namespace could be initialized only when it is first used. Similarly, vars could be initialized only on first use. This shifts the setup time from the beginning of the program, when namespaces are loaded, to later in program execution. This should significantly improve start times, as not all of the namespaces or vars are needed immediately at program start.

This change could also lead to unexpected variations in run time performance. Currently the loading penalty is paid immediately at the launch of the program. This is simple for the developer to understand, predict, and manage. It is also sensible for server-side apps, for which startup time is not an important factor. With lazy function initialization, the penalty would be paid across the run time of a program. For example, a function may unexpectedly take a long time to execute the first time if it depends on a large number of uninitialized vars or namespaces.

The third way to improve the lean compiler projects is to compile Clojure functions as static methods on namespace classes. Currently in both the standard Clojure compiler and the Skummet compiler, Clojure functions are each compiled to individual class files. In Skummet these functions are referenced from a namespace class using direct, static references. These static references could be replaced by inlining the functions themselves as static methods. This would likely improve startup times by reducing the overall number of classes and amount of bytecode to be loaded at startup.

Alternative approaches

The larger problem of finding a practical, modern language for Android app development remains unresolved. General approaches to this problem include compile-to-JVM languages, platform-independent frameworks, and the Android Native Development Kit (NDK).

Many languages run on the JVM, from more popular languages such as Scala [47], JRuby [3], Jython [4], Groovy [48], and Xtend [19] to less popular ones like Kawa [5], Kotlin [35], Ceylon [49], and Fantom [2]. Scala as a statically-compiled functional language provides many of Clojure's functional programming benefits without suffering from startup time problems. As it currently stands, however, none of these languages has gained significant traction for Android development.

Frameworks such as PhoneGap [31], Cordova [7], and Sencha [33] allow one to develop Android applications using the web technologies of JavaScript, HTML, and CSS. The resulting apps are wrapped as native applications and have access to Android framework features through their specific framework wrappers. Others like Xamarin [34] and Appcelerator Titanium [32] allow native Android app development in C# and JavaScript. In this case the resulting apps are not web apps wrapped as native apps but are in more direct contact with the underlying platform.

Frameworks that allow one to build once and compile for many platforms have many advantages in terms of portability and potential for reducing overall work in porting apps. On the other hand, native APIs need to be

accessed through various wrappers and native performance is more difficult to achieve.

ClojureScript, a dialect of Clojure that compiles to JavaScript, could also be executed on any framework supporting JavaScript. ClojureScript does not suffer from the same startup problem as JVM Clojure. The disadvantage of this approach is the additional layers of abstraction between the programming language and the underlying system. JVM Clojure can inter-operate directly with the standard Java libraries and tools, while ClojureScript would need to be compiled to JavaScript and then access the underlying libraries through some framework.

The Android Native Development Kit (NDK) can also be used to develop Android apps with C or C++ [25]. In certain cases, such as game development, using the NDK can improve the performance of apps. This approach suffers from lack of access to the standard Android Java APIs and is difficult to recommend for general app development.

Chapter 9

Conclusions

Clojure is a promising language for modern Android app development. However, long Clojure on Android app startup times are a recognized problem which has received little attention. In addition, the overall performance of Clojure on Android is poorly understood.

We benchmarked and analyzed Clojure on Android startup and execution performance. Clojure on Android execution performance was found to be comparable to that of the desktop. However, we found that Clojure on Android apps start slowly, even on relatively new devices and on the new Android ART runtime. Minimal startup times exceed 1.5 seconds and scale upwards quickly with larger apps. The issue is closely tied to design choices of the Clojure language and compiler. The Skummet lean Clojure compiler attempts to improve Clojure performance by removing dynamic features. We found that it succeeds in reducing Clojure on Android startup times by about half across all benchmarks.

Based on these results, Clojure on Android is not currently viable for general Android app development and is unlikely to be viable in the near future without significant changes. The Skummet lean Clojure compiler greatly reduces the startup time problem but requires additional improvements to make Clojure on Android practical in the general case. There are also open questions about the tradeoffs of the additional complexity brought by lean Clojure changes.

Future directions for Clojure on Android include further performance research, continuation of the lean Clojure projects, or pursuit of alternative Clojure approaches such as ClojureScript. Further research is needed to estimate the memory performance of Clojure on Android to determine whether Clojure would be viable even with fast startup times. The Skummet lean Clojure project could be continued by removing unused dependencies at compile time, implementing functions as static methods on namespace classes,

or initializing vars and namespaces lazily. Any one of these directions could have an additional large impact on startup times and bring Clojure on Android startup performance to an acceptable range. Finally, another direction for Clojure on Android is to pursue alternative approaches such as use of the ClojureScript language in place of JVM Clojure. ClojureScript compiles to JavaScript and could be run in Android through various third party frameworks. It does not suffer from the same same startup performance issues, but also lacks the easy access to Java tools that JVM Clojure provides. The practicality of this approach is an open question.

The larger question of finding a viable alternative language for Android app development remains open. Clojure on Android is currently not viable for many Android projects even with the implemented lean Clojure changes, but further development of lean Clojure or alternative Clojure approaches could still make Clojure on Android practical.

Bibliography

[1] SPECjvm2008, November 2014. URL `http://www.spec.org/jvm2008/`. 39

[2] Fantom, March 2015. URL `http://fantom.org`. 70

[3] JRuby - the Ruby programming language on the JVM, March 2015. URL `http://jruby.org`. 37, 70

[4] Jython: Python for the Java platform, March 2015. URL `http://www.jython.org`. 37, 70

[5] The Kawa Scheme language, March 2015. URL `http://www.gnu.org/software/kawa`. 70

[6] SPEC JVM98 benchmarks, February 2015. URL `http://www.spec.org/jvm98`. 39

[7] The Apache Software Foundation. Apache Cordova, February 2015. URL `http://cordova.apache.org`. 8, 70

[8] Stephen M Blackburn, Robin Garner, Chris Hoffmann, Asjad M Khang, Kathryn S McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z Guyer, et al. The DaCapo benchmarks: Java benchmarking development and analysis. In *ACM Sigplan Notices*, volume 41, pages 169–190. ACM, 2006. doi: 10.1145/1167473.1167488. 39

[9] Chien-Wei Chang, Chun-Yu Lin, Chung-Ta King, Yi-Fan Chung, and Shau-Yin Tseng. Implementation of JVM Tool Interface on Dalvik virtual machine. In *VLSI Design Automation and Test (VLSI-DAT), 2010 International Symposium on*, pages 143–146. IEEE, 2010. doi: 10.1109/VDAT.2010.5496711. 42

[10] Ben Christensen and Jafar Husain. Reactive programming in the Netflix API with RxJava, Feb 2014. URL `http://techblog.netflix.com/2013/02/rxjava-netflix-api.html`. 14

[11] International Data Corporation. Smartphone OS market share, Q3 2014. *http://www.idc.com/prodserv/smartphone-os-market-share.jsp*, Jan 2015. 10

[12] Charles Daly, Jane Horgan, James Power, and John Waldron. Platform independent dynamic Java Virtual Machine analysis: the Java Grande Forum benchmark suite. In *Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*, pages 106–115. ACM, 2001. 39

[13] Mattia Denti and Jukka K. Nurminen. Performance and energy-efficiency of Scala on mobile devices. In *Next Generation Mobile Apps, Services and Technologies (NGMAST), 2013 Seventh International Conference on*, pages 50–55. IEEE, 2013. doi: 10.1109/NGMAST.2013.18. 38

[14] Lieven Eeckhout, Andy Georges, and Koen De Bosschere. How Java programs interact with virtual machines at the microarchitectural level. In *ACM SIGPLAN Notices*, volume 38, pages 169–186. ACM, 2003. 40

[15] David Ehringer. The Dalvik virtual machine architecture, March 2015. URL `http://davidehringer.com/software/android/The_dalvik_virtual_machine.pdf`. 24

[16] Chas Emerick, Brian Carper, and Christophe Grand. *Clojure Programming*. O'Reilly Media, Inc., 2012. ISBN 9781449335359. 15

[17] Andy Fingerhut. clojure-benchmarks, November 2014. URL `https://github.com/jafingerhut/clojure-benchmarks`. 36, 56

[18] Mike Fogus. Why no eval?, Nov 2014. URL `http://blog.fogus.me/2011/07/29/compiling-clojure-to-javascript-pt-2-why-no-eval`. 34

[19] The Eclipse Foundation. Xtend - modernized Java, March 2015. URL `http://eclipse.org/xtend`. 70

[20] Brent Fulgham and I Gouy. The computer language benchmarks game. *http://shootout.alioth.debian.org*, Nov 2014. 9, 36, 37, 56, 66, 69

[21] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous Java performance evaluation. *ACM SIGPLAN Notices*, 42(10): 57–76, 2007. 39

[22] Google. Processes and threads, November 2014. URL `http://developer.android.com/guide/components/processes-and-threads.html`. 13

[23] Google. Keeping your app responsive, November 2014. URL `https://developer.android.com/training/articles/perf-anr.html`. 13

[24] Google. Introducing ART, Nov 2014. URL `https://source.android.com/devices/tech/dalvik/art.html`. 25

[25] Google. Android NDK, January 2015. URL `https://developer.android.com/tools/sdk/ndk/index.html`. 71

[26] Anthony Gutierrez, Ronald G Dreslinski, Thomas F Wenisch, Trevor Mudge, Ali Saidi, Chris Emmons, and Nigel Paver. Full-system analysis and characterization of interactive smartphone applications. In *Workload Characterization (IISWC), 2011 IEEE International Symposium on*, pages 81–90. IEEE, 2011. 38, 39

[27] Stuart Halloway. *Programming Clojure*. Pragmatic Bookshelf, 2009. ISBN 9781934356333. 15

[28] Rich Hickey. Clojure, February 2015. URL `http://clojure.org`. 15

[29] Rich Hickey. TRANSIT, February 2015. URL `http://blog.cognitect.com/blog/2014/7/22/transit`. 56

[30] Robert Hundt. Loop recognition in C++/Java/Go/Scala. *Proceedings of Scala Days*, 2011, 2011. 38

[31] Adobe Systems Inc. PhoneGap, February 2015. URL `http://phonegap.com`. 8, 70

[32] Appcelerator Inc. Titanium mobile development environment, March 2015. URL `http://www.appcelerator.com/titanium`. 70

[33] Sencha Inc. Sencha, March 2015. URL `http://www.sencha.com`. 70

[34] Xamarin Inc. Xamarin, March 2015. URL `http://xamarin.com/platform`. 70

[35] JetBrains. Kotlin - statically typed programming language targeting the JVM and JavaScript, March 2015. URL `http://kotlinlang.org`. 70

[36] Wing Hang Li, David R White, and Jeremy Singer. JVM-hosted languages: They talk the talk, but do they walk the walk? In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, pages 101–112. ACM, 2013. doi: 10.1145/2500828.2500838. 37, 39

[37] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification*. Addison-Wesley, 2013. ISBN 9780133260441. 22

[38] Reid McKenzie. Project Oxcart, October 2014. URL `https://github.com/oxlang/oxcart`. 34, 35

[39] Reid McKenzie. Oxcart and Clojure, October 2014. URL `http://www.arrdem.com/2014/06/26/oxcart_and_clojure/`. 35

[40] Reid McKenzie. Of oxen, carts and ordering, October 2014. URL `http://www.arrdem.com/2014/08/05/of_oxen,_carts_and_ordering/`. 35

[41] Microsoft. The Reactive Extensions (Rx), February 2015. URL `https://msdn.microsoft.com/en-us/data/gg577609.aspx`. 14, 56

[42] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F Sweeney. Producing wrong data without doing anything obviously wrong! *ACM Sigplan Notices*, 44(3):265–276, 2009. 39

[43] Fiona Fui-Hoon Nah. A study on tolerable waiting time: How long are web users willing to wait? *Behaviour & Information Technology*, 23(3): 153–163, 2004. 11, 66

[44] Jakob Nielsen. *Usability Engineering*. Elsevier, 1994. ISBN 978-0125184069. 11

[45] Hyeong-Seok Oh, Beom-Jun Kim, Hyung-Kyu Choi, and Soo-Mook Moon. Evaluation of Android Dalvik virtual machine. In *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems*, pages 115–124. ACM, 2012. 38, 39

[46] Dhinakaran Pandiyan, Shin-Ying Lee, and Carole-Jean Wu. Performance, energy characterizations and architectural implications of an emerging mobile platform benchmark suite - MobileBench. In *Workload Characterization (IISWC), 2013 IEEE International Symposium on*, pages 133–142. IEEE, 2013. 38, 39

[47] École Polytechnique Fédérale de Lausanne. Scala, February 2015. URL `http://www.scala-lang.org`. 37, 70

[48] The Groovy Project. Groovy - A multi-faceted language for the Java platform, March 2015. URL `http://groovy-lang.org`. 37, 70

[49] Inc. Red Hat. Ceylon, March 2015. URL `http://ceylon-lang.org`. 70

[50] Aibek Sarimbekov, Andrej Podzimek, Lubomir Bulej, Yudi Zheng, Nathan Ricci, and Walter Binder. Characteristics of dynamic JVM languages. In *Proceedings of the 7th ACM workshop on Virtual machines and intermediate languages*, pages 11–20. ACM, 2013. doi: 10.1145/2542142.2542144. 36, 37

[51] Aibek Sarimbekov, Andreas Sewe, Stephen Kell, Yudi Zheng, Walter Binder, Lubomír Bulej, and Danilo Ansaloni. A comprehensive toolchain for workload characterization across JVM languages. In *Proceedings of the 11th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 9–16. ACM, 2013. doi: 10.1145/2462029.2462033. 36

[52] Andreas Sewe, Mira Mezini, Aibek Sarimbekov, and Walter Binder. Da Capo con Scala: Design and analysis of a Scala benchmark suite for the Java Virtual Machine. In *Proceedings of the 26th Conference on Object-Oriented Programming, Systems, Languages and Applications*, OOPSLA '11, pages 657–676, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0940-0. doi: 10.1145/2048066.2048118. 37, 38

[53] Andreas Sewe, Mira Mezini, Aibek Sarimbekov, Danilo Ansaloni, Walter Binder, Nathan Ricci, and Samuel Z. Guyer. new Scala() instanceof Java: A comparison of the memory behaviour of Java and Scala programs. In *Proceedings of the International Symposium on Memory Management*, ISMM '12, pages 97–108, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1350-6. doi: 10.1145/2258996.2259010. 37, 38

[54] James Edward Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*. Elsevier, 2005. ISBN 978-1558609105. 21

[55] Alexander Yakushev. Project Skummet, March 2015. URL `http://clojure-android.info/skummet`. 34

[56] Dmitrijs Zaparanuks and Matthias Hauswirth. Characterizing the design and performance of interactive Java applications. In *Performance*

Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on, pages 23–32. IEEE, 2010. 39