Scala on Android: the-current-state-of-the-art

Chris Howell

March 13, 2016

Abstract

This document features a review of the viability of an alternative programming language as the basis for an Android application. I will present a test case, a review of the development process, an in-depth study of an alternative programming language in comparison to Java and a reflective review detailing my experience and opinion on the subject matter.

Contents

1 Part 1

1.1 Concurrency

Concurrency is a major issue when writing computer programs. Highly concurrent applications execute multiple aspects of the program simultaneously. While this approach does allow for the creation of complex programs with multiple components executing at the same time, this can also cause issues that are hard to reason about and identify the cause of[.3.5].

Concurrency is often confused with parallelism, with this due to the illusion that concurrent applications give of processes executing at the same time. It should be noted that concurrency is multiple processes being executing sequentially at a fast pace to give the illusion that the process are executing at the same time, while parallelism executes the processes at the same time.

An example of a highly concurrent program would be an operating system. An operating system is highly concurrent as it needs to execute multiple aspects of the operating system in rapid succession, with this being achieved by a scheduling policy that allocates CPU time to each running process.

When using an operating system a good scheduling policy is essential because the user needs to believe the illusion that all the components are that they currently have running are executing at the same time. For example, a user reading their emails while listening to a music player should not be able to tell that these tasks are operation at different times.

For example, one common issue caused by concurrency is one thread accessing data that is currently being mutated by a separate thread. Because the data is in the process of being mutated by one thread, when the second thread attempts to also mutate the data unexpected results often occur. This can lead to unexpected program behaviour.

To counter these problems programming languages implement different approaches to ensuring that concurrently executing threads do not cause issues with other threads that are executing. For example, functional programming languages such as Haskell have no mutable state. By creating pure functions, functions that only manipulate variables that are local to them, many of the concurrency issues encountered in imperative programming languages simply do not occur. Imperative programming language such as Java would require a drastic overhaul to remove all mutable data, although it should be noted that Java does support immutable state via the use of final.

One common issue that arises from the use of concurrency is deadlock. Deadlock typically occurs when two process are competing for limited resources. An example of this would be when process A is waiting for process B to finish, while process B is also waiting for process A to finish.

Another common concurrency related issue that arises from the use of concurrency in programs is thread starvation. Thread starvation occurs when one process is taking up all the CPU times while another thread is waiting to be schedule CPU time. An example of thread starvation occurring would be in a GUI when the users presses a button, with nothing happens for a noticeable amount of time and then finally the button event occurs. It should be noted that thread starvation is a possible side effect of using synchronized methods, with this occurring when one thread holds onto a synchronized method for an unusual length of time.

1.2 Explicit

Explicit concurrency is defined by the developer, with this being achieved using threads in Java and C++. Java allows the developer to explicitly define concurrent aspects of a program via the use of threads. The developer can define a new thread and then override the run method to enable them to define some behaviour to be executed by the thread.

Some of the advantages of using explicit concurrency are that CPU time can be better allocated resulting in less time unused and more time executing work. Applications that implement explicit concurrency efficiently have high performance.

Multiple threads can be defined in a program to execute concurrently. One issue that arises from this approach is that as the size of the application grows it can be harder for the developer to reason about how the concurrent aspects of the program need to execute.

When a developer uses explicit concurrency they have to take care to ensure that issues such as deadlock are avoided. One way of achieving this in Java is by implemented synchronized methods. Only one thread can access a synchronized method at any given time. While the current executing thread has controlled of a synchronized method, no other thread can access the same method. This approach reduces the likelihood of shared data being mutated in an unexpected way. One issue that does occur from this approach is thread starvation.

Another approach to avoiding explicit concurrent issues is by declaring a variable final. By declaring a variable to be final it is no longer mutable. This means that threads are capable of retrieving this value, but unable to mutate it to cause unexpected results.

One useful approach to thread-based concurrency that a developer can take is the creation of a thread pool. By creating a thread pool the developer has a better degree of control over how multiple threads are created and executed. By combining thread creation with a loop the developer can create multiple threads without having to explicitly create and run each thread.

The alternative to a thread pool is a thread executor. The thread executor handles the creation and lifespan of the thread and allows the developer to set the number of threads

to create. This is a useful feature as it lets you scale the amount of threads created depending on the number of cores that the current machine has available. By dynamically generating the number of threads created the developer can stop a concurrent program with overwhelming the CPU with a higher number of threads and tasks to execute.

One of the alternatives Java provides to using thread-based concurrency is via the use of future tasks and callable. They allow the developer to define a worker that implements callable so that it must return a value. This approach is arguably better that using raw threads because it creates code that is less blocking when return values are required.

1.3 Implicit

Implicit concurrency differs from explicit in that the programming language compiler handles concurrently executing activities. Generally, when a program features implicitly concurrent processes executing, the developer states what activity they want to concurrently execute and then leaves the lower-level aspects of how to execute the activity up to the compiler/interpreter to handle.

Java 8 is the most recent iteration of Java and implements implicit concurrency into the language via the use of parallel streams. By using parallel streams, a developer can let the compile decide how best to implement the concurrent operations. A parallel stream has a built in sort function for common types, but can also be passed a custom comparator when required.

It should be noted that to truly utilize parallelism the tasks being completed must be appropriate. For example, attempting to sort using a parallel stream is less effective than checking if each number in an array is a prime number. The reason for this is because check if a number is prime in an array doesnt require any other data from the array than the number at that position being checked, with this meaning a parallel stream can rapidly iterate the array checking each number.

For sorting, the parallel stream must firstly break down the array into smaller sections, then sort these sections, put each of these sections back together and then finally sort

One advantage of using implicit concurrency is that the developer spends less time worrying about how best to implement the concurrency and more on the functionality the program is intended to implement.

One disadvantage of using implicit concurrency is that arguably implicit doesn't have the same level of performance as explicitly defined concurrency. When a developer chooses to use explicit concurrency over implicit, they can then define exactly how they want the concurrent processes to operate. A highly skilled developer could potentially write a program with explicit concurrency that performs better than implicit.

2 Part 2

C++ and Java are both object oriented programming languages that share a number of similarities syntactically, but these similarities are mostly superficial and when looking a little deeper at how they operator you can see that they are actually very different programming languages. While they are both considered Object-Oriented programming languages they differ in how they implement the Object-Oriented principles.

It should be noted that neither language is considered a true Object-Oriented programming language, but out of the two languages Java is considered the purer implementation of the Object-Oriented principles.

2.1 Runtime Environment

One key area where the two languages differ significantly is how they are compiled and executed. C++ is compiled down into instructions and can be executed directly from the CPU. Because of this approach the program must be targeted towards a certain O/S when being developed, this greatly reduces the portability of a C++ program. Executing directly on the CPU does allow for skilled developers to create applications that are higher performances than similar Java programs as the programs are built to target a specific type of environment.

Java on the other hand is compiled down into byte code and executed on the JVM. The JVM handles the process of converting the byte code into executable machine code. Because of this it can often involve a rethink of the structure of the program if performance is an issue when porting programmers to the other language.

2.2 Code Portability

This is because porting Java to C++ will require you to write the memory management and error handling code. This can be a time consuming issue in large and complex programs. Alternatively, if you were to port a Java program to C++ this would require you to remove the code relating to memory management JVM handles this explicitly.

Another issue that could arise from porting C++ code to Java or the other way is how the respective language handles object references. For example, in Java if you attempt to create a copy of an object, this results in a reference being created to the object. If you then alter the original object the copy will also be affected. This is because the copy only has a reference to the original object it is not an actual copy of the object.

2.3 Error Handling

Another key difference is the way that both languages handle errors. For example, when trying to access an out of bound element position in a Java array, a null pointer exception

is thrown and the user is informed. This is an example of Java's built-in error handling system.

While in C++ this error doesn't occur unless the user implements the exception throw themselves as C++ has no default null pointer exception. If, for example, the program access a inappropriate array position by default the user is not informed and the program continues to operate. Often this will mean that at some point the program will exhibit strange behavior. The result of this is C++ programs are more likely to encounter hard to trace bugs that might cause strange behaviour in the program or the mutable data to return unexpected results.

2.4 Memory Management

One of the key elements to Java's portability is its use of the Java Virtual Machine. The JVM handles all low level memory management, with this meaning that the memory is allocated implicitly. This is in stark contrast to C++ which requires the user to allocate and de-allocate memory explicitly. Due to the explicit nature of C++'s memory management there is a much greater chance of memory leaks occurring. The JVM has ensured that Java is a far reaching programming language contain on a number of devices, with these ranging from mobile phones to wristwatches.

C++ requires the programmer to allocate and de-allocate the memory explicitly. This does allow for greater fine tuning of the performance of a program, but can result in frustrating and hard to trace errors. For example, the programmer could reallocate the memory utilized by an object that is still referenced by another object. If this does accidentally occur the program will still compile and execute, but will result in an error at runtime.

In contrast to this the JVM includes a built-in garbage collector. This garbage collector reallocates the memory of objects that are no longer accessible. This solution results in a greatly reduced chance of memory leaks, but can arguably result in worse performance than C++ programs. Because the memory is explicitly allocated in C++, this can result in a program that has efficient memory management outperforming Java.

By introducing implicit memory management into the Java programming language they have made it significantly easier for beginners. The implicit approach means that the programmer isnt required to delete objects that are no longer in use, with this being a common cause of memory leaks in C++.

2.5 Approach to object oriented princinples

While they are both considered object-oriented programming languages, of the two languages Java is arguably the true representation of an object-oriented programming language. C++ is an extension of the C programming language that attempts to add object-oriented principles on top of a procedural programming language. In contrast every class

in Java inherits from the Object superclass.

While C++ does allow for the creation of classes in an object-oriented fashion it is by no means mandatory to program in this style. It is perfectly legitimate in C++ to write code in a procedural style in C++ with no class declarations. While Java is only considered an Object-Oriented programming language, C++ is a multi-paradigm programming language in that it facilitates Object-Oriented programming as well as procedural.

2.6 Inheritance

Java only allows you to inherit from one super class, while C++ allows you to inherit from multiple classes. Java does support implementing multiple interfaces into a single class, with this approach still allows you to guarantee a class share methods with more than one other class even if the implementation of those methods varies.

Traditionally in Java an interface was as abstract as it could possibly be and required the developer to write the concrete implementation of each method inside any class that implements the interface, this results in Java only allows single inheritance to reduce the number of errors that might occur from objects inheriting methods of the same name from different classes that have different implementations and outputs.

With the introduction of Java 8 this has now changed and interfaces do allow for concrete method implementation. It should be noted that while they do now allow for the method code to be coded directly into the interface, these interfaces cannot implement global variables for that interface.

Appendices

.1 Benchmark results computer 1

All tests were first run multiple times with a selection of the results combined and the average calculated.

All tests for this section were run on a desktop computer which featured an i5-2500k processor.

.1.1 Benchmark 1000 records

.1.2 Benchmark 10k records

.1.3 Benchmark 100k records

.2 Benchmark results computer 2

All tests for this section were run on a desktop computer which featured an i5-2500k processor.

.3 Benchmark program classes

.3.1 BenchNon Class

```
 //implicit convernsion
implicit def onClickListener(f: (View => Unit)): View.OnClickListener = {
     new View.OnClickListener() {
          override def onClick(v: View) {
        f(v)
        }
     }
     }
```

.3.2 BenchImplicit Class

```
import java.util.ArrayList;
import java.util.Comparator;
import java.util.List;
import java.util.concurrent.TimeUnit;

/**
 * Created by Chris Howell on 10/12/2015.
 *
 * The benchImplicit Class is the benchmark class that implements a parallel stream
 * to test implicit concurrency.
 *
 * This class takes an ArrayList and uses a parallel stream to sort the ArrayList
 * before returning the times.
 */
public class benchImplicit {

    ArrayList arr = new ArrayList<Integer>();
    ArrayList arr2 = new ArrayList();
    public benchImplicit(ArrayList arr){

    this.arr = arr;

    }

    public double test1() {

        long start = System.nanoTime();

            arr.parallelStream().sorted(new BubbleSorter(arr));


        long finish = System.nanoTime();

        double seconds = TimeUnit.MILLISECONDS.convert(finish - start, TimeUnit.NANOSECONDS) / 1000.0;
        return seconds;
    }


    public class BubbleSorter implements Comparator<Integer> {

        public BubbleSorter(ArrayList<Integer> arr){

            sort(arr);

        }

        public int compare(Integer i1, Integer i2) {
           if(i1 < i2) return 01;
           else return 02;
```

```
    }

    List<Integer> sort(List<Integer> arr){
        int temp;
        for( int y=0; y<arr.size()-1; y++ ) {
            for ( int z=y+1; z<arr.size(); z++ ){
                if(compare(arr.get(z),arr.get(y) ) == 1 ) {
                    temp = arr.get(y);
                    arr.set(y,arr.get(z));
                    arr.set(z,temp);
                }
            }
        }

        return arr;
    }
}

}
```

.3.3 BenchExplicit

```
import java.util.*;
import java.util.concurrent.TimeUnit;

/**
 * Created by Chris on 10/12/2015.
 *
 * This class explicitly defines threads to sort smaller sections of the ArrayList. After these
 * smaller sections have been sorted the thread pool then reconnects these segments and sorts
 * the complete ArrayList.
 */
class benchExplicit implements Runnable,sorter {


    ArrayList<Integer> arr;
    List<Integer> arr1;
    List<Integer> arr2;

    boolean release = false;
    public double result;

    public benchExplicit(ArrayList<Integer> arr){

        this.arr = arr;
        int  section = arr.size() / 2;

        arr1 = this.arr.subList(0, section);
        arr2 = this.arr.subList(section,section*2);

    }




    public void run() {
        long start = System.nanoTime();
        Thread thread1 = new Thread(new Runnable(){

            @Override
            public void run() {
                arr1 = sort(arr1);

            }

        });
        thread1.start();
        try {
            thread1.join();
```

```
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

        Thread thread2 = new Thread(new Runnable(){

            @Override
            public void run() {
                arr2 = sort(arr2);

            }

        });
        thread2.start();
        try {
              thread2.join();
        } catch (InterruptedException e) {

            e.printStackTrace();
        }


        List<Integer> temp = new ArrayList<Integer>();
        temp.addAll(arr1);
        temp.addAll(arr2);


        arr = (ArrayList<Integer>) sort(temp);
        long finish = System.nanoTime();
        double seconds = TimeUnit.MILLISECONDS.convert(finish - start, TimeUnit.NANOSECONDS) / 1000.0;

        result = seconds;


    }

}
```

.3.4 benchThreadPool

```
import java.util.ArrayList;
import java.util.List;
import java.util.Random;
import java.util.concurrent.*;

/**
 * Created by chris on 27/02/16.
 *
 * The thread pool class features an inner class which represents the worker
 * thread process for this class. The main thread pool class calculates the
 * number of cores, then generates the appropriate number of threads. Afterwards,
 * it handles the lifetime of a thread while it completes it's operation.
 */
public class benchThreadPool implements sorter {

    ArrayList<List<Integer>> arr = new ArrayList<>();

    boolean release = false;
    int cores = Runtime.getRuntime().availableProcessors();
    List<Integer> sorted = new ArrayList<>();


    public benchThreadPool(ArrayList<Integer> arr1){
        int  section = arr1.size() / cores;

        int increment = 0;
        int y = 2;


        for(int x = 0; x < cores; x++){

            if(x == 0){
                List<Integer> temp = arr1.subList(increment*section,(section));
                arr.add(temp);
                increment++;
            }
            else{
                List<Integer> temp = arr1.subList(increment*section,section*y);
                arr.add(temp);
                increment++;
                y++;
            }

        }


    }
```

```
public double test(){

    long start = System.nanoTime();



    ThreadPoolExecutor executor = (ThreadPoolExecutor) Executors.newFixedThreadPool(cores);

    List<Future<List<Integer>>> resultList = new ArrayList<>();

    for (int i=0; i<executor.getCorePoolSize(); i++)
    {

        worker work = new worker(arr.get(i));
        Future<List<Integer>> result = executor.submit(work);
        resultList.add(result);
    }


    for(Future<List<Integer>> future : resultList)
    {
        try
        {
            //  System.out.println(future.get());
            sorted.addAll(future.get());
            //  future.get().forEach(x -> System.out.println(x));



        }
        catch (InterruptedException | ExecutionException e)
        {
            e.printStackTrace();
        }
    }

    executor.shutdown();

    List<Integer> done = sort(sorted);
    long finish = System.nanoTime();
    double seconds = TimeUnit.MILLISECONDS.convert(finish - start, TimeUnit.NANOSECONDS) / 1000.0;

    return seconds;


}



class worker implements Callable<List<Integer>>
{
```

```
        List<Integer> sorted = new ArrayList<>();
        final List<Integer> arr1 = new ArrayList<>();
        List<Integer> work;
        boolean release = false;


        public worker(List<Integer> work){


            this.work = work;

        }

        @Override
        public List<Integer> call() throws Exception {
            List<Integer> sort = work;

            sorted.addAll(sort(sort));

            return sorted;
        }
    }

}
```

.3.5 Sorter Interface

```
import java.util.List;

/**
 * Created by Chris on 09/03/2016.
 *
 * Interface with default method for sorting a list of integers
 * using bubble sort.
 *
 */
public interface sorter {


       default List<Integer> sort(List<Integer> arr){
           int temp;
           for( int y=0; y<arr.size()-1; y++ ) {


               for ( int z=y+1; z<arr.size(); z++ ){
                   if( arr.get(z) < arr.get(y) ) {
                       temp = arr.get(y);
                       arr.set(y,arr.get(z));
                       arr.set(z,temp);
                   }
               }
           }
           return arr;
   }

}
```

References

[1] J. D. Bovey, M. M. Dodson, The Hausdorff dimension of systems of linear forms *Acta Arithmetica* **45** (1986), 337–358.

[2] J. W. S. Cassels, *An Introduction to Diophantine Approximation*,Cambridge University Press, Cambridge, 1965.

[3] The GAP Group, GAP – Groups, Algorithms, and Programming, Version 4.5.6; 2012. (http://www.gap-system.org)

[4] J. Howie, *Generalised triangle groups of type* $(3, 5, 2)$, http://arxiv.org/abs/1102.2073 (2011).