Improving Java Code Performance

Make your Java/Dalvik VM happier

Agenda

- Who am I
- Java vs optimizing compilers
- Java & Dalvik
- Examples - Do & dont's
- Tooling

Who am I?

(Mobile) Software Engineering Manager at Imagination Technologies

Past:

- Service2Media, Monitise, Microjocs (Ubisoft Barcelona)

Java vs optimizing compilers

Java vs optimizing compilers

- Optimizing compilers produce optimized code for the platform where the code will be executed.

- Java code is supposed to run on many machines & different environments so the java compiler does not assume anything and leaves all optimizations to the JVM.

Java vs optimizing compilers

- For the same exact reason Java bytecode is stack based instead of register based to make it easy to interpret on CPUs with very few registers (although not the case on current mobile devices & desktop computers)

Java vs optimizing compilers

Stack based

```
iload_3
iload_2
iadd
istore_2
```

Register based

```
add-int v2, v3, #1
```

Java vs optimizing compilers

- All optimizations will be done by the JVM when loading or running the code.

- JVM implementations may & will differ from each other. Java specification is not strict on how a JVM should be implemented just on what must be the result

Java vs optimizing compilers

- Some JVM have a JIT compiler which increases dramatically the execution speed.

While the Java code is running, the JVM detects the performance bottlenecks or the parts of code that are most executed (hotpots) and translates them to native code.

Java vs optimizing compilers

Maybe the Java compiler takes the idea of leaving all the optimizations to the JVM too far.

Java vs optimizing compilers - C

```
#include <stdio.h>
int main() {
  int a = 10;
  int b = 1 + 2 + 3 + 4 + 5 + 6 +
  a;


  printf("%d\n", b);
}
```

```
…
movl $31, %esi
call _printf
…
```

* Using gcc & -O2 compiler option

Java vs optimizing compilers - Javac

```
public static void main(String args[]) {
  int a = 10;
  int b = 1 + 2 + 3 + 4 + 5 + 6 + a;

  System.out.println(b);
}
```

```
0: bipush        10
2: istore_1
3: bipush        21
5: iload_1
6: iadd
7: istore_2
...
```

Java vs optimizing compilers - Javac

```
public static void main(String args[]) {
  int a = 10;
  int b = 1 + 2 + 3 + 4 + 5 + a + 6;

  System.out.println(b);
}
```

```
0: bipush        10
2: istore_1
3: bipush        15
5: iload_1
6: iadd
7: bipush        6
9: iadd
10: istore_2

...
```

Java vs optimizing compilers - Javac

```
public static void main(String args[]) {
  int a = 10;
  int b = a + 1 + 2 + 3 + 4 + 5 + 6;

  System.out.println(b);
}
```

```
0: bipush        10
2: istore_1
3: iload_1
4: iconst_1
5: iadd
6: iconst_2
7: iadd
8: iconst_3
9: iadd
10: iconst_4
11: iadd
12: iconst_5
13: iadd
14: bipush        6
16: iadd
17: istore_2
```

Java & Dalvik

Java & Dalvik

- Android devices don't use a standard JVM. They use a Dalvik VM

- Dalvik is register based (instead of stack based) and has a different instruction set than the standard JVM.

Java & Dalvik

- Java classes needs to be converted to dex format in order to run

- Since Android 2.2 Dalvik includes a JIT compiler

Examples - Do & Dont's

Autoboxing

- Autoboxing is transparent for the developer but what are the implications of using it?
- Is it solved in compile time or runtime?
- Java compiler will generate some *extra* code for you...

Autoboxing

```
long total = 0;
for(int i = 0; i < N; i++) {
  total += i;
}
```

```
4: lconst_0
5: lstore_3
6: iconst_0
7: istore 5
9: iload 5
11: ldc #6;
13: if_icmpge 28
16: lload_3
17: iload 5
19: i2l
20: ladd
21: lstore_3
22: iinc 5,1
25: goto 9
```

Autoboxing

```
Long total = 0;
for(Integer i = 0; i < N; i++) {
  total += i;
}
```

```
9: iconst_0
10: invokestatic #4; //Method java/lang/Integer.valueOf:
    (I)Ljava/lang/Integer;
13: astore 4
15: aload 4
17: invokevirtual #5; //Method java/lang/Integer.intValue:()I
20: ldc #6; //int 10000000
22: if_icmpge 65
25: aload_3
26: invokevirtual #7; //Method java/lang/Long.longValue:()J
29: aload 4
31: invokevirtual #5; //Method java/lang/Integer.intValue:()I
34: i2l
35: ladd
36: invokestatic #3; //Method java/lang/Long.valueOf:
    (J)Ljava/lang/Long;
39: astore_3
40: aload 4
42: astore 5
44: aload 4
46: invokevirtual #5; //Method java/lang/Integer.intValue:()I
49: iconst_1
50: iadd
51: invokestatic #4; //Method java/lang/Integer.valueOf:
    (I)Ljava/lang/Integer;
54: dup
55: astore 4
57: astore 6
59: aload 5
61: pop
62: goto 15
```

Autoboxing

- This is what that code is actually doing:

```
Long total = 0;
for(Integer i = Integer.valueOf(0);
            i.intValue() < N;
            i = Integer.valueOf(i.intValue() + 1)) {

  total = Long.valueOf(total.longValue() + (long)i.intValue())
}
```

Sorting

Two easy ways to sort:

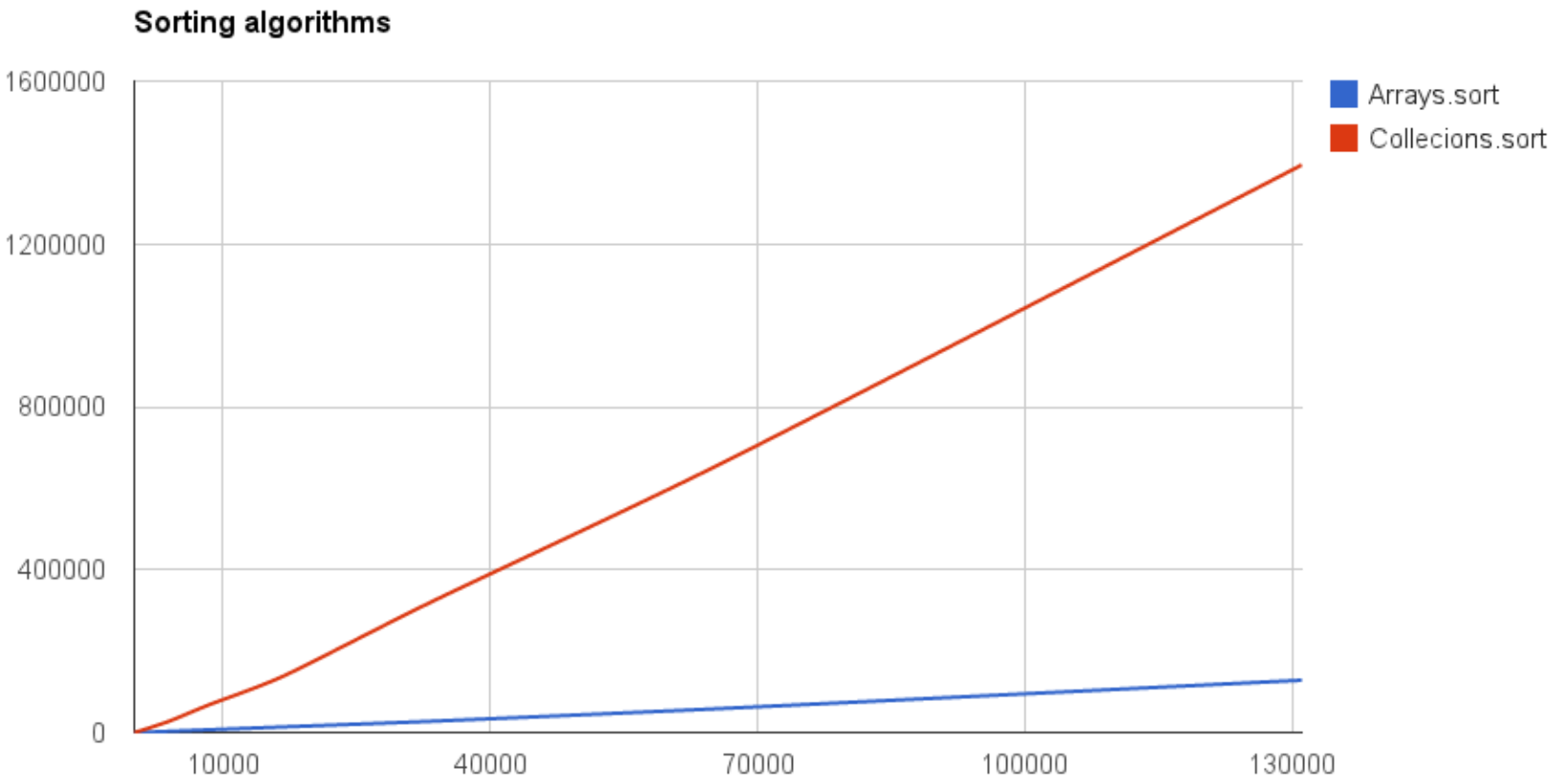
- Arrays.sort(array);
- Collections.sort(list);

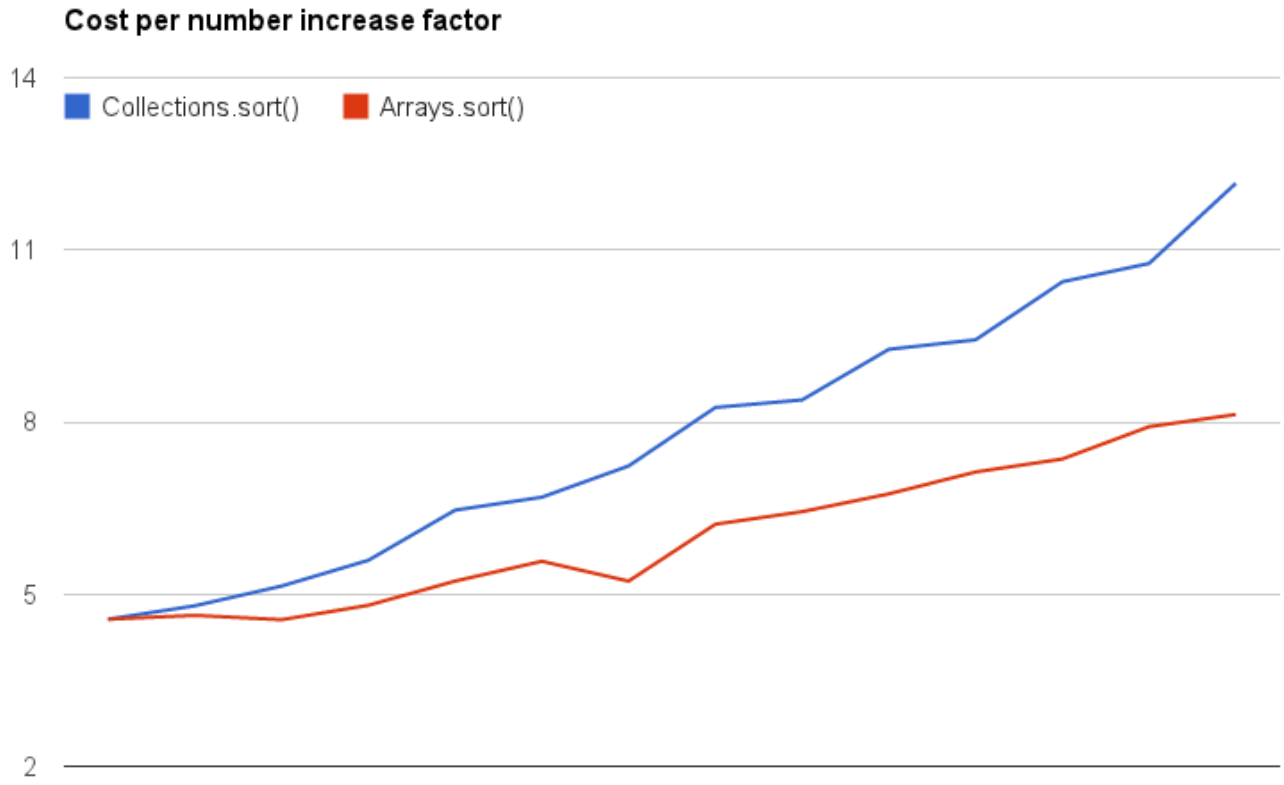
Lets evaluate the performance as both data type & sorting algorithms are different.

Sorting

- Arrays.sort(arrays) uses a modified version of a 3-way quicksort.

- Collections.sort(list) uses a iterative implementation of mergesort (stable)


Sorting algorithms
1600000
1200000
800000
400000
0
10000
40000
70000
100000
130000
Arrays.sort
Collecions.sort


Cost per number increase factor
14
11
8
5
2
Collections.sort()
Arrays.sort()

Loops

Let's evaluate different ways to implement loops and the performance impact they might have

Loops - List

```
ArrayList<Integer> list = new …
static long loopStandardList() {
    long result = 0;
    for(int i = 0; i < list.size(); i++) {
        result += list.get(i);
    }
    return result;
}
```

Loops - List (Java bytecode)

```
7: lload_0
8: getstatic     #26      // Field list:Ljava/util/ArrayList;
11: iload_2
12: invokevirtual #54     // Method java/util/ArrayList.get:(I)Ljava/lang/Object;
15: checkcast     #38     // class java/lang/Integer
18: invokevirtual #58     // Method java/lang/Integer.intValue:()I
21: i2l
22: ladd
23: lstore_0
24: iinc          2, 1
27: iload_2
28: getstatic     #26     // Field list:Ljava/util/ArrayList;
31: invokevirtual #61     // Method java/util/ArrayList.size:()I
34: if_icmplt     7
```

Loops - List (Dalvik bytecode)

```
0003: sget-object v3, Lcom/rrafols/bcndevcon/Loop;.list:Ljava/util/ArrayList;
0005: invoke-virtual {v3}, Ljava/util/ArrayList;.size:()I
0008: move-result v3
0009: if-lt v0, v3, 000c // +0003
000b: return-wide v1
000c: sget-object v3, Lcom/rrafols/bcndevcon/Loop;.list:Ljava/util/ArrayList;
000e: invoke-virtual {v3, v0}, Ljava/util/ArrayList;.get:(I)Ljava/lang/Object;
0011: move-result-object v3
0012: check-cast v3, Ljava/lang/Integer;
0014: invoke-virtual {v3}, Ljava/lang/Integer;.intValue:()I
0017: move-result v3
0018: int-to-long v3, v3
0019: add-long/2addr v1, v3
001a: add-int/lit8 v0, v0, #int 1 // #01
001c: goto 0003 // -0019
```

Loops - foreach

```
ArrayList<Integer> list = new …
static long loopForeachList() {
    long result = 0;
    for(int v : list) {
        result += v;
    }
    return result;
}
```

Loops - foreach (Java bytecode)

```
12: aload_3
13: invokeinterface #70,  1   // InterfaceMethod java/util/Iterator.next:()
18: checkcast     #38         // class java/lang/Integer
21: invokevirtual #58         // Method java/lang/Integer.intValue:()I
24: istore_2
25: lload_0
26: iload_2
27: i2l
28: ladd
29: lstore_0
30: aload_3
31: invokeinterface #76,  1   // InterfaceMethod java/util/Iterator.hasNext:()Z
36: ifne          12
```

Loops - foreach (Dalvik bytecode)

```
0008: invoke-interface {v4}, Ljava/util/Iterator;.hasNext:()Z
000b: move-result v3
000c: if-nez v3, 000f // +0003
000e: return-wide v0
000f: invoke-interface {v4}, Ljava/util/Iterator;.next:()Ljava/lang/Object;
0012: move-result-object v3
0013: check-cast v3, Ljava/lang/Integer;
0015: invoke-virtual {v3}, Ljava/lang/Integer;.intValue:()I
0018: move-result v2
0019: int-to-long v5, v2
001a: add-long/2addr v0, v5
001b: goto 0008 // -0013
```

Loops - Array

```
static int[] array = new ...
static long loopStandardArray() {
    long result = 0;
    for(int i = 0; i < array.length; i++) {
        result += array[i];
    }
    return result;
}
```

Loops - Array (Java bytecode)

```
7: lload_0
8: getstatic     #28                 // Field array:[I
11: iload_2
12: iaload
13: i2l
14: ladd
15: lstore_0
16: iinc          2, 1
19: iload_2
20: getstatic     #28                 // Field array:[I
23: arraylength
24: if_icmplt     7
```

Loops - Array (Dalvik bytecode)

```
0003: sget-object v3, Lcom/rrafols/bcndevcon/Loop;.array:[I
0005: array-length v3, v3
0006: if-lt v0, v3, 0009 // +0003
0008: return-wide v1
0009: sget-object v3, Lcom/rrafols/bcndevcon/Loop;.array:[I
000b: aget v3, v3, v0
000d: int-to-long v3, v3
000e: add-long/2addr v1, v3
000f: add-int/lit8 v0, v0, #int 1 // #01
0011: goto 0003 // -000e
```

Loops - size stored

```
static int[] array = new ...
static long loopStandardArraySizeStored() {
    long result = 0;    int length = array.length;
    for(int i = 0; i < length; i++) {
        result += array[i];
    }
    return result;
}
```

Loops - size stored (Java bytecode)

```
12: lload_0
13: getstatic     #28                 // Field array:[I
16: iload_3
17: iaload
18: i2l
19: ladd
20: lstore_0
21: iinc          3, 1
24: iload_3
25: iload_2
26: if_icmplt     12
```

Loops - size stored(Dalvik bytecode)

```
0006: if-lt v0, v1, 0009 // +0003
0008: return-wide v2
0009: sget-object v4, Lcom/rrafols/bcndevcon/Loop;.array:[I
000b: aget v4, v4, v0
000d: int-to-long v4, v4
000e: add-long/2addr v2, v4
000f: add-int/lit8 v0, v0, #int 1 // #01
0011: goto 0006 // -000b
```

Loops - backwards

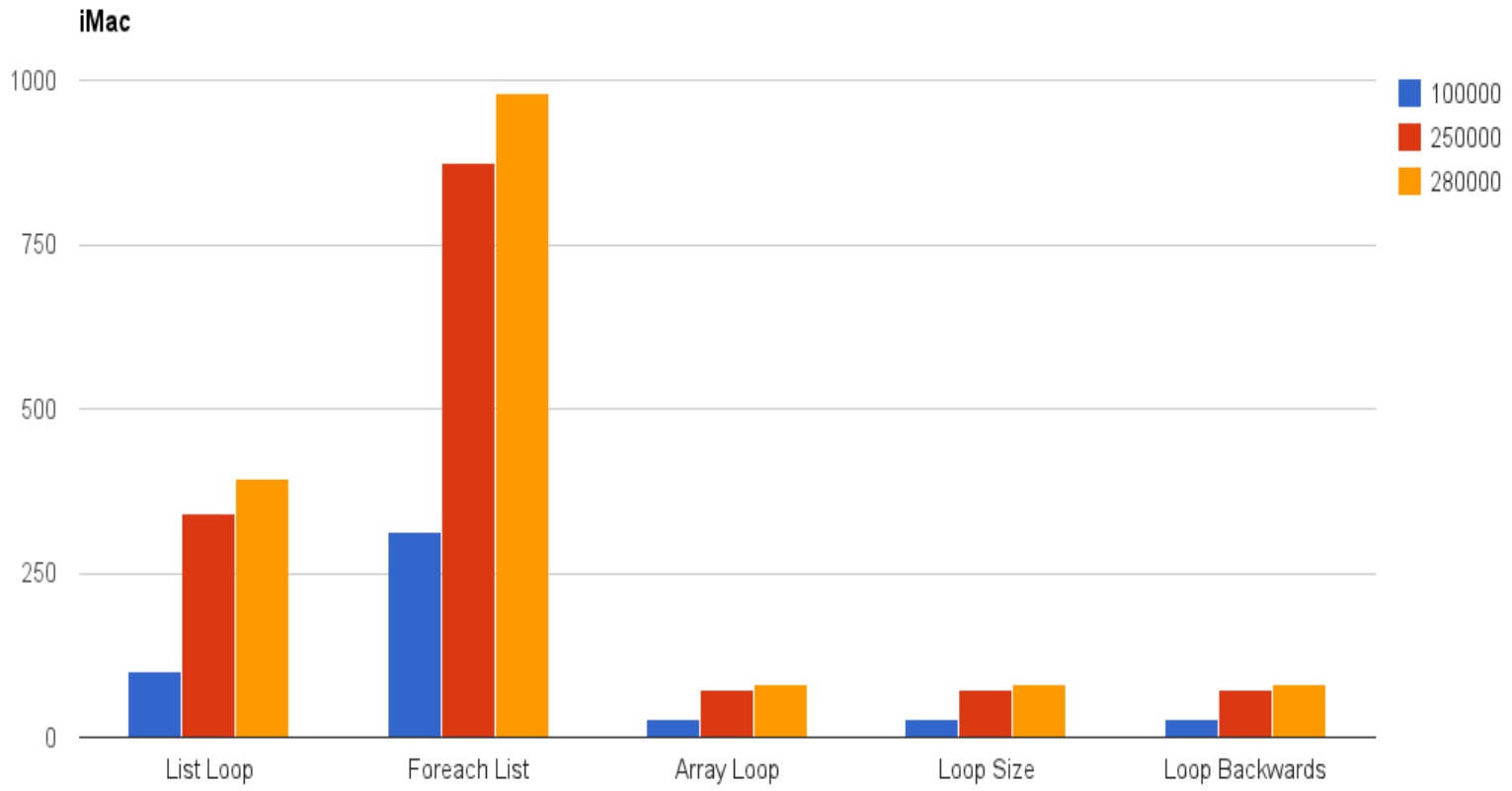
```
static int[] array = new ...
static long loopStandardArrayBackwards() {
  long result = 0;
  for(int i = array.length - 1; i >= 0; i--) {
        result += array[i];
  }
  return result;
}
```

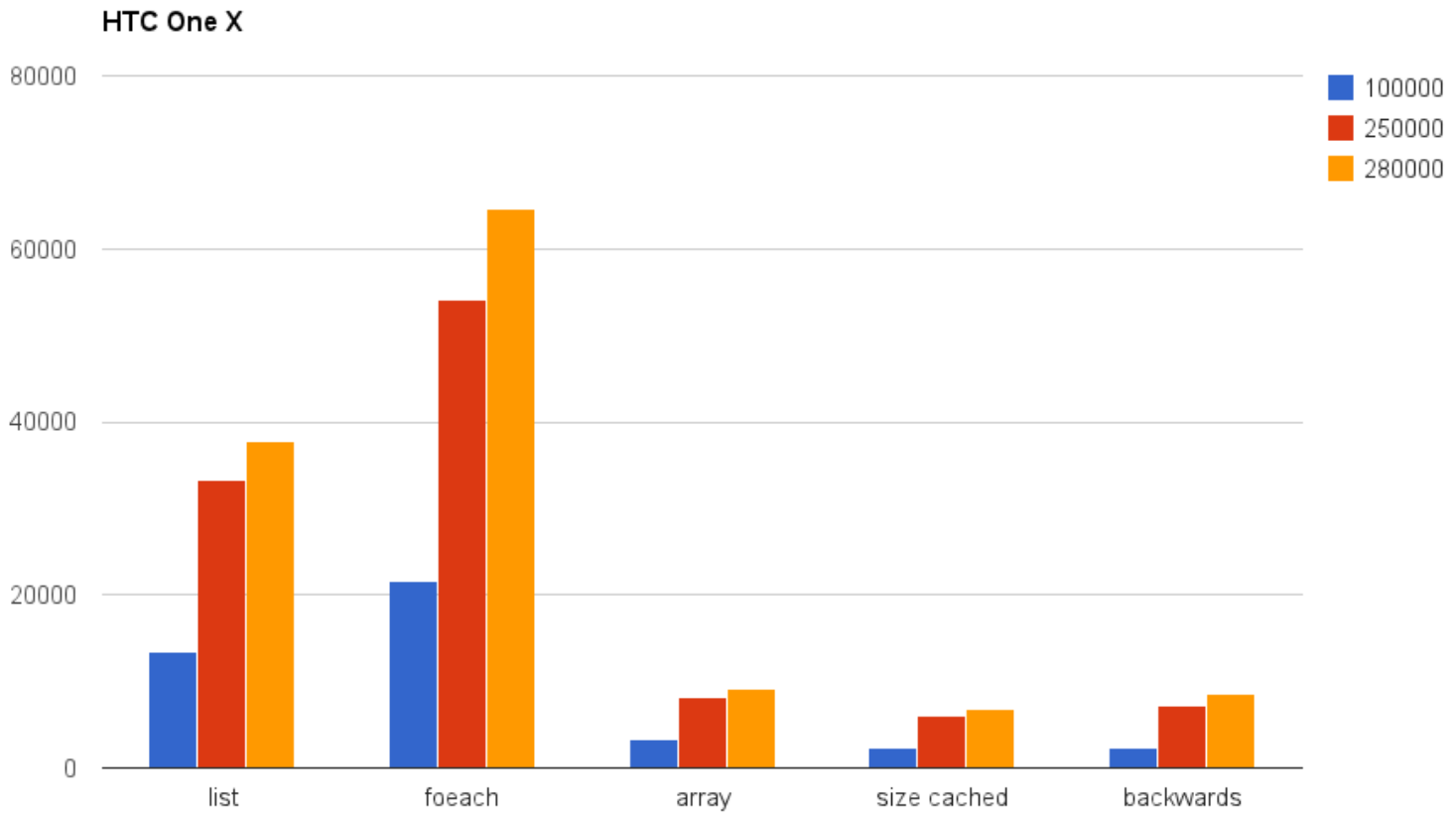
Loops - backwards (Java bytecode)

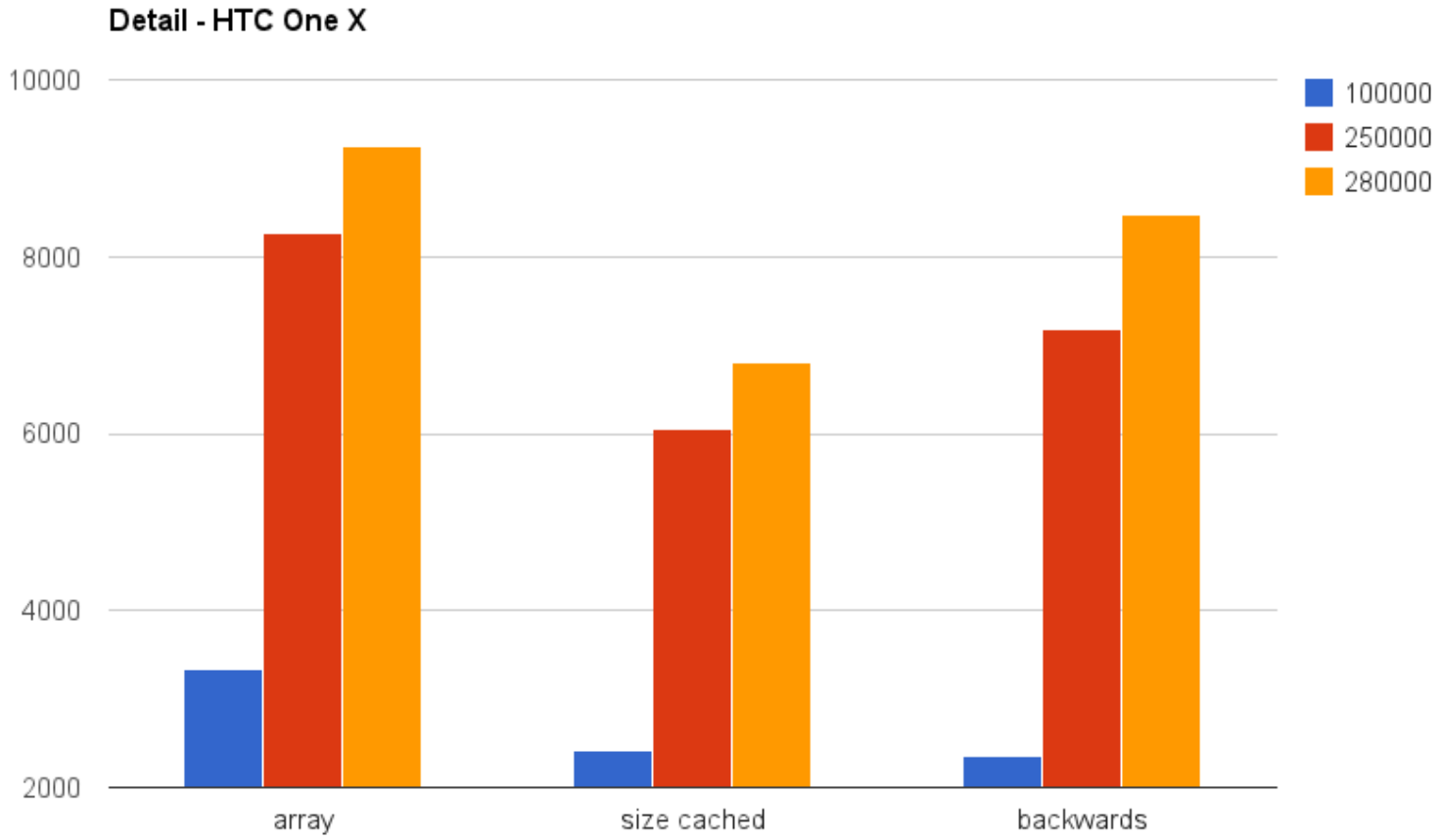
```
12: lload_0
13: getstatic     #28                 // Field array:[I
16: iload_2
17: iaload
18: i2l
19: ladd
20: lstore_0
21: iinc          2, -1
24: iload_2
25: ifge          12
```

Loops - backwards(Dalvik bytecode)

```
0007: if-gez v0, 000a // +0003
0009: return-wide v1
000a: sget-object v3, Lcom/rrafols/bcndevcon/Loop;.array:[I
000c: aget v3, v3, v0
000e: int-to-long v3, v3
000f: add-long/2addr v1, v3
0010: add-int/lit8 v0, v0, #int -1 // #ff
0012: goto 0007 // -000b
```


iMac
1000
750
500
250
0
100000
250000
280000
List Loop
Foreach List
Array Loop
Loop Size
Loop Backwards


HTC One X
80000
60000
40000
20000
0
100000
250000
280000
list
foeach
array
size cached
backwards


Detail - HTC One X
10000
8000
6000
4000
2000
array
size cached
backwards
100000
250000
280000

String concatenation

Java allows us to concatenate Strings using the + operator to make our lifes easier. As what happened before with autoboxing the compiler will generate some additional code.

String concatenation

```
String str = "";
for(int i = 0;  i  < ITERATIONS; i++) {
  str += ANY_OTHER_STRING;
}
```

String concatenation

```
8: new           #26           // class java/lang/StringBuilder
11: dup
12: aload_1
13: invokestatic  #28          // Method java/lang/String.valueOf:
  (Ljava/lang/Object;)Ljava/lang/String;
16: invokespecial #34          // Method java/lang/StringBuilder."<init>":(Ljava/lang/String;)V
19: ldc           #11          // String ANY_OTHER_STRING
21: invokevirtual #37          // Method java/lang/StringBuilder.append:(Ljava/lang/String;)
24: invokevirtual #41          // Method java/lang/StringBuilder.toString:()Ljava/lang/String;
27: astore_1
28: iinc          2, 1
31: iload_2
32: bipush        ITERATIONS
34: if_icmplt     8
```

String concatenation

- It's creating a StringBuilder Object in every iteration of the loop. It's the same as the following code:

```
String str = "";
for(int i = 0;  i  < ITERATIONS; i++) {
    StringBuilder sb = new StringBuilder(String.valueOf(str));
    sb.append(ANY_OTHER_STRING);
    str = sb.toString();
}
```

String concatenation - alternatives

- Use String.concat()
- Concat cost is O(N) + O(M) - (N,M) length of each String
- Concat returns a new String Object.

```
String str = "";
for(int i = 0;  i  < ITERATIONS; i++) {
  str = str.concat(ANY_OTHER_STRING);
}
```

String concatenation - alternatives

- Use StringBuilder
- StringBuffer.append cost is O(M) amortized time (M length of appended String)
- Avoids creation of new objects.

```
StringBuilder sb = new StringBuilder()
for(int i = 0;  i  < ITERATIONS; i++) {
  sb.append(ANY_OTHER_STRING);
}
str = sb.toString();
```

String concatenation - alternatives

- StringBuilder is the not thread safe implementation of StringBuffer.

- No need to add synchronization overhead if it's not going to be used by multiple threads.

Tooling

Tooling - Disassembler

Java

- javap -c <classfile>

Android:

- Dexdump -d <dexfile>
- Smali - https://code.google.com/p/smali/

Tooling - Performance measurement

- Caliper
  - Benchmark library & utils
  - v1.0 -> Annotations

- Vogar
  - Execute caliper in android.
  - Needs caliper 0.5rc-1
  - Doesn't work with latest releases of Caliper (1.0)

Tooling - Obfuscation

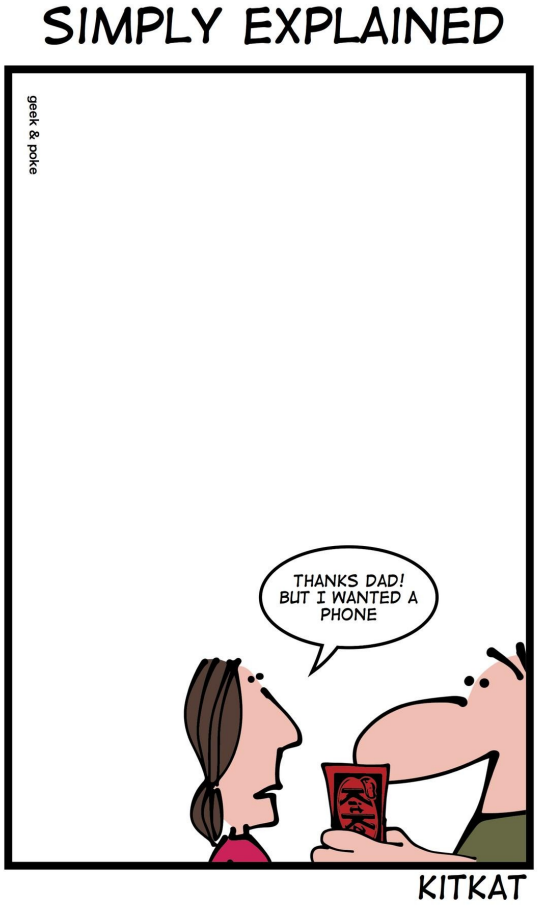
- Proguard & Dexguard
  - Proguard & Dexguard are code obfuscators but they also do a good job on optimizing at bytecode level, reducing overhead, eliminating dead code, inlining methods, …
  - Proguard is free and comes with the android sdk, needs to be enabled by just uncommenting one line in project.properties

Tooling - Performance measurement

- Avoid doing manual performance tests with System.currentTimeMillis() (although gives a good estimate) and multiple tests in one run.

- JIT might be evil!

Do not trust the compiler!



@rrafols

http://blog.rafols.org