EXHIBIT 2-8

UNITED STATES DISTRICT COURT

NORTHERN DISTRICT OF CALIFORNIA

SAN FRANCISCO DIVISION

ORACLE AMERICA, INC.

Plaintiff,

v.

GOOGLE INC.

Defendant.

Case No. CV 10-03561 WHA

SUMMARY AND REPORT OF NOEL POORE

SUBMITTED ON BEHALF OF PLAINTIFF ORACLE AMERICA, INC.

EXHIBIT 2-8

II. BACKGROUND AND QUALIFICATIONS

8. In 1985, I received a BA in Computer Science from Cambridge University, Cambridge, England.

9. I have many years experience of working with system software for mobile devices. From 2001 to 2008 I worked as a senior technical executive for SavaJe Technologies, a company which developed a complete Java platform for smartphones. SavaJe was acquired by Sun, which was later acquired by Oracle. Both in this role and in previous roles I gained a lot of experience of measuring and optimizing the execution time and memory consumption of code for mobile devices.

10. I am currently the lead architect for embedded middleware in JDK8 Embedded. I am also the technical lead of a group which provides Java technology for iOS and Android applications.

11. I am qualified to conduct performance analysis and analyze the results obtained because I have many years experience of developing, performance testing, benchmarking, and optimizing embedded Java software. At SavaJe I both supervised and took part in software performance testing and analysis to maximize execution speed and minimize the memory footprint of the SavaJe OS.

III. PERFORMANCE ANALYSIS OF UNITED STATES PATENT NO. 5,966,702

A. Introduction

12. I performed experiments to estimate the benefits to Android from using the functionality that Oracle accuses of infringing the ’702 patent, including disabling that functionality in Android.

13. The baseline code for these experiments was the Froyo release of Android, pulled from the public Google git repository as follows

```
repo init –u git://android.git.kernel.org/platform/manifest.git –b froyo
repo sync
```

14. Modifications (as detailed below) were made to the source in the local repository to implement the experiments.

B. Methodology – Experiment 1

15. I modified the dx tool to estimate the size of a dex file if duplicate constant pool entries are not removed. Below are the steps I followed:

(1) Download Android 2.2 (Froyo) source code.

(2) Identify the source code that processes Java class files and packages the output as Dalvik class files (also known as dex files).

(3) Modify this source code to prevent the removal of duplicate constant pool entries when multiple Java classes are combined into a single dex file.

(4) Build the modified source code to create a modified dx.jar

(5) Use the modified dx.jar to create dex files for several different jar files. Each jar file was processed multiple times, controlling the duplicate removal process to provide better measurement of the advantage gained by duplicate removal.

The results are shown in the spreadsheet attached as Exhibit B.

C. Methodology – Experiment 2

16. Here, I aimed to estimate the size of a dex file if duplicate constant pool entries are not removed, by extrapolation from Java class sizes. Below are the steps I followed:

(1) I wrote an application to measure the amount of byte code contained in one or more Java class files. This allows a simple calculation of the size of the metadata contained in the class files.

(2) I used the --statistics command line argument to the dx tool to determine the amount of Dalvik code contained in the output dex file.

(3) Making the assumption that in the absence of duplicate removal, the amount of metadata in the amalgamated class file will be the same as the amount of metadata in the separate class files, I performed a simple calculation to estimate the size of the resulting dex file.

(4) The estimated dex file size is
```
Size of Java classes – Java byte code size + Dalvik code size
```

The results are shown in the spreadsheet attached as Exhibit C.

D. Methodology – Experiment 3

17. Here, my goal was to analyze the contents of dexdump files to estimate the amount of space saved by removing duplicate constant pool entries.

(1) I was provided with a set of dexdump files which were produced by running the standard Android dexdump tool on optimized dex ("odex") files taken from an actual Android device. These are attached as Exhibit D.

(2) I wrote a perl script which identifies the method and field definitions and the method, field, type and string references in the dexdump file. I have attached the perl script as Exhibit E. [INTERNAL NOTE – filename: analyze.pl]

(3) The perl script identifies the number of each type of constant pool entry that have been removed by the duplicate removal process, and estimates the memory size of the removed duplicates.

(4) The perl script outputs a number of statistics about the dexdump, including the total amount of space saved by duplicate removal.

(5) The output of the perl script was imported into a spreadsheet.

The results are shown in the spreadsheet attached as Exhibit F.

E. Source Code Modifications – Experiment 1

18. The modifications made to implement experiment 1 are contained in Appendix A.

19. The dex file format differs from the Java classfile format in the sense that instead of a single constant pool, the constant pool is broken up into five sections. As input classes are read, the dx tool accumulates the contents of each class (constants, fields, methods, code etc) into an in-memory data structure. Each input class is processed independently and at this stage there is no identification or removal of duplicate constants. Once all of the input classes have been processed then the five constant pool sections are created by iterating through all of the constants in the in-memory representation of the input classes. Each constant pool section has an associated in-memory data structure where the constants of that kind are accumulated. It is in the accumulation process that the removal of duplicate constant pool entries happens. The in-memory representation of the constant pool sections uses a Java TreeMap data structure, which implements the Map interface. As the Javadoc for the Map interface states (see http://download.oracle.com/javase/1.5.0/docs/api/java/util/Map.html): "A map cannot contain duplicate keys; each key can map to at most one value."

20. The duplicate constant removal is essentially performed as a result of this property of the TreeMap. The existing code checks to see if a constant already exists in the map and returns that as the interned value of the constant if it does. This means that it is not possible

to simply allow duplicate constant pool entries to accumulate without widespread modifications to the way in which the dx tool is architected. To get around this, I forced the creation of duplicate entries and made each duplicate unique by inserting a unique number into the value of the duplicate. For example, the type "Ljava/lang/String;" could be duplicated as "Ljava/lang/String123;". The number is made unique by incrementing it each time a duplicate is created within a specific section of the constant pool. Since this process lengthens the string and therefore inserts extra bytes into the generated dex file, I modified the code to count the number of bytes added so that they could be subtracted from the final file size to get more accurate results.

21. I made this modification to the following classes, each of which represents one of the five sections of the constant pool:

```
com.android.dx.dex.file.FieldIdsSection
com.android.dx.dex.file.MethodIdsSection
com.android.dx.dex.file.ProtoIdsSection
com.android.dx.dex.file.StringIdsSection
com.android.dx.dex.file.TypeIdsSection
```

22. Two points should be noted about this approach. Firstly, the different sections of the constant pool are not completely independent from each other. For example, creating a duplicate type "Ljava/lang/Object2;" causes an additional string constant "L/java/lang/Object2;" to be created as well. If the experiment is carried out in a simplistic way where all five of the constant pool sections are "re-duplicated" at the same time, then because of the renaming, duplicates of duplicates are created, inflating the total number of constant pool entries and thus the final dex file size. The results shown below demonstrate this problem. To avoid it, I modified the code so that I could individually control the duplicate creation for each section of the constant pool. I ran the modified dx tool 5 times for each input jar, creating duplicate entries for only one section of the constant pool each time. The estimated final dex file size is then derived by adding up the additions caused by the duplicate creation for each part of the constant pool.

23. Secondly, consider the fact that the input to the dx tool is a set of compiled Java classes. Within a single class, there will frequently be multiple references to a single constant

pool entry. Within the code of the dx tool, at the point where a constant pool entry is being added to the TreeMap, this context is lost, and so there is a possibility that creating duplicate entries in the way described will create duplicates that do not exist in the input Java class files. Rectifying this would have required significant modifications to the dx source code.

24. These two points mean that the results gained by experiment 1 can be regarded as an "upper bound estimate" of the dex file size in the absence of duplicate constant removal.

25. Experiment 1 was repeated on three different jar files representing popular benchmarking applications, and also on the jar file containing the dx tool itself, as an example of a complex application.

F. Source Code – Experiment 2

26. The code used to calculate the size of the Java byte code contained in one or more Java class files is contained in Appendix B. No modifications were made to Android source code for this experiment.

27. Experiment 2 was repeated on the same jars files as experiment 1, and also on the jar for the dx tool itself, as an example of a larger and more complex application.

G. Source Code – Experiment 3

28. The perl script used to extract information from the dexdump files is contained in Appendix C. No modifications were made to Android source code for this experiment.

29. The perl script takes a set of file names as arguments. Each file is examined in turn.

30. The output of the perl script is data in CSV format. This was chosen to make it easy to import the results into a spreadsheet where calculations could more easily be performed.

31. The basic approach of the perl program is to count the number of times each field, method, type and string constant is used, and then estimate the additional size of the dex file if the constants were duplicated rather than being referenced multiple times.

32. For each file, each line of the file is examined in turn. Lines are examined for patterns indicating class boundaries, field and method declarations, and references to method, field, string and type constants.

33. Note that the dexdump files do not include any information about references to prototype constants. These are allowed for in calculating the size required to store a method constant.

34. Class boundaries are identified in order to make the estimation process more accurate. As mentioned previously in the section "Source Code – Experiment 1", the original Java classes from which the dex file is created do not contain duplicate constant pool entries. To take account of this, the perl script identifies class boundaries in the dexdump file, and only counts a single instance of each constant per class.

35. Field and method declarations each include a reference to the associated field/method constant, and are gathered in hashes called %fieldDefs and %methodDefs. In each case, the field/method name is used as the key of the hash. The value is not used. Since each field and method is only defined once, the existence of the key is all that matters.

36. References to method, field, string and type constants are found by identifying occurrences of the strings "method@nnnn", "field@nnnn", "string@nnnn" and "type@nnnn" respectively. The value of the constant is extracted as well. Four hashes are maintained, one for each type of constant, with the key being the value of the constant and the value the number of occurrences found. As mentioned above, only once occurrence of a constant is counted per class. This is done by remembering the last class number in which a constant was referenced. When a new reference is found, the current class number is compared with the last class number, and the occurrence counted or dropped as appropriate.

37. Symbolic values are defined at the top of the perl program which are used to determine the storage size required in a dex file to store the different kinds of constant. The comments around these definitions indicate how the values were calculated using publicly available Android documents.

38. The dexdump files were created from odex files. These are files that have been optimized from the original dex files. One of the optimizations is to convert many invoke-virtual Dalvik instructions to invoke-virtual-quick instructions. These “quickened” instructions make use of a register and a vtable offset to refer to the target method rather than an offset into the method constant pool. This means that the number of constant pool references is reduced in the odex file. This will reduce the apparent impact of the removal of duplicate references. Thus, the results from this experiment should be regarded as an **under-estimate** of the true benefit derived from duplicate constant removal.

39. Experiment 3 was run on a collection of dexdump files. Each of these files contains a listing of the contents of an application odex file from a Nexus One smartphone.

40. Line 56 of the perl program declares a variable named $showDuplicatesCount. If this is set to be non-zero, the perl script will output information about the number of constants it finds that are referred to more than once. For each kind – method, string, type, and field, it will output the most referenced item along with number of times it is referenced. Using this it can easily be seen that every single odex file contains many constants that are referred to multiple times.

H. Results – Experiment 1

41. The tables below show the results for experiment 1. The rows in each table show the number of entries in each section of the constant pool in the output dex file, the file size of the dex file (corrected for extra bytes added as described above), and the delta from the original dex file size.

IV. PERFORMANCE ANALYSIS OF UNITED STATES PATENT NO. 6,061,520

A. Introduction

54. I performed experiments to estimate the benefits to Android from using the functionality that Oracle accuses of infringing the '520 patent, which including disabling that functionality in Android.

55. The baseline code for these experiments was the Froyo release of Android, pulled from the public Google git repository as follows

```
repo init –u git://android.git.kernel.org/platform/manifest.git –b froyo
repo sync
```

56. Modifications were made to the source in the local repository to implement the experiments.

B. Methodology

57. I modified the dx tool to estimate the size of a dex file if the fill-array-data instructions were not generated. Below are the steps I followed:

(1) Download Android 2.2 (Froyo) source code.

(2) Identify the source code that processes Java class files and packages the output as Dalvik class files (also known as dex files).

(3) Modify this source code to prevent the replacement of static array initialization byte code sequences with the Dalvik fill-array-data instruction.

(4) Build the modified source code to create a modified dx.jar

(5) Use the modified dx.jar to create dex files for several different jar files, recording the size of the output dex file in each case.

The results are shown in the spreadsheet attached as Exhibit G.

C. Source Code Modifications

58. The modifications made to Android source code to implement the experiment are contained in Appendix D.

59. The dx tool translates Java byte code to Dalvik byte code as part of the process of forming the output dex file. As a part of the translation procedure, the code contains a heuristic that attempts to spot Java bytecode sequences that represent the initialization of arrays of

primitive data types. These bytecode sequences are simulated (described as "play execution" in the patent) and the initialized array values accumulated. So, rather than simply translating the Java byte codes that initialize the array to the corresponding Dalvik byte codes, the dx tool replaces the Java byte code array initialization sequence with a fill-array-data instruction that copies the initialized value of the array into place in one instruction. The exact sequence of Dalvik bytecodes involved in array allocation and initialization is shown below.

60. I modified the source code of the dx tool so that by setting an environment variable `NO_ARRAY_OPT`, I could disable the heuristic described above. When the heuristic is disabled, the fill-array-data instruction is not generated, and the original Java bytecode sequence is directly translated into equivalent Dalvik instructions which individually assign constant values to each initialized array element.

61. To run the dx tool with the environment variable set (thus disabling the heuristic) I used a command such as

```
NO_ARRAY_OPT=true ./dx –dex –output=initArrayTest10.dex
initArrayTest10.class
```

62. To study the effect on dex file size for different primitive data types, I created a number of simple Java programs, all of which are similar to the following:

```
public class initArrayTest10 {
    private static int[] array = {
        1, 2, 3, 4, 5, 6, 7, 8, 9, 10
    };

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

The source code of all of the Java programs used for this experiment is shown in Appendix E.

63. Each program consists of a statically initialized array – the data type and number of elements in the array were varied. The contents of the main() method are the same in each case, so the only influence on the dex file size is the number and type of the array elements, and whether or not the array initialization is being optimized.

64. The array initialization in the program shown above translates to the following Java bytecode. There are 44 instructions in this sequence, 40 of which initialize the array:

```
0       bipush 10
2       newarray 10
4       dup
5       iconst_0
6       iconst_1
7       iastore
8       dup
9       iconst_1
10      iconst_2
11      iastore
12      dup
13      iconst_2
14      iconst_3
15      iastore
16      dup
17      iconst_3
18      iconst_4
19      iastore
20      dup
21      iconst_4
22      iconst_5
23      iastore
24      dup
25      iconst_5
26      bipush 6
28      iastore
29      dup
30      bipush 6
32      bipush 7
34      iastore
35      dup
36      bipush 7
38      bipush 8
40      iastore
41      dup
42      bipush 8
44      bipush 9
46      iastore
47      dup
48      bipush 9
50      bipush 10
52      isatore
53      putstatic 5
56      return
```

65. Using the dexdump tool that is part of the Android source distribution, it is possible to see the normal sequence of Dalvik bytecodes generated for the array initialization in the program shown above. Note that there are 5 instructions in this sequence, only 1 of which actually initializes the array:

```
000150:                     |[000150] initArrayTest10.<clinit>:()V
000160: 1300 0a00           |0000: const/16 v0, #int 10 // #a
000164: 2300 0600           |0002: new-array v0, v0, [I // class@0006
000168: 2600 0600 0000      |0004: fill-array-data v0, 0000000a // +00000006
00016e: 6900 0000           |0007: sput-object v0, LinitArrayTest10;.array:[I // field@0000
000172: 0e00                |0009: return-void
000174: 0003 0400 0a00 0000 0100 0000 0200 ... |000a: array-data (24 units)
```

66. When the array initialization heuristic is disabled, the following array initialization code is generated. Note that there are 29 instructions in this sequence, 25 of which are used to initialize the array:

```
000150:                     |[000150] initArrayTest10.<clinit>:()V
000160: 1256                |0000: const/4 v6, #int 5 // #5
000162: 1245                |0001: const/4 v5, #int 4 // #4
000164: 1234                |0002: const/4 v4, #int 3 // #3
000166: 1223                |0003: const/4 v3, #int 2 // #2
000168: 1212                |0004: const/4 v2, #int 1 // #1
00016a: 1300 0a00           |0005: const/16 v0, #int 10 // #a
00016e: 2300 0600           |0007: new-array v0, v0, [I // class@0006
000172: 1201                |0009: const/4 v1, #int 0 // #0
000174: 4b02 0001           |000a: aput v2, v0, v1
000178: 4b03 0002           |000c: aput v3, v0, v2
00017c: 4b04 0003           |000e: aput v4, v0, v3
000180: 4b05 0004           |0010: aput v5, v0, v4
```

```
000184: 4b06 0005          |0012: aput v6, v0, v5
000188: 1261               |0014: const/4 v1, #int 6 // #6
00018a: 4b01 0006          |0015: aput v1, v0, v6
00018e: 1261               |0017: const/4 v1, #int 6 // #6
000190: 1272               |0018: const/4 v2, #int 7 // #7
000192: 4b02 0001          |0019: aput v2, v0, v1
000196: 1271               |001b: const/4 v1, #int 7 // #7
000198: 1302 0800          |001c: const/16 v2, #int 8 // #8
00019c: 4b02 0001          |001e: aput v2, v0, v1
0001a0: 1301 0800          |0020: const/16 v1, #int 8 // #8
0001a4: 1302 0900          |0022: const/16 v2, #int 9 // #9
0001a8: 4b02 0001          |0024: aput v2, v0, v1
0001ac: 1301 0900          |0026: const/16 v1, #int 9 // #9
0001b0: 1302 0a00          |0028: const/16 v2, #int 10 // #a
0001b4: 4b02 0001          |002a: aput v2, v0, v1
0001b8: 6900 0000          |002c: sput-object v0, LinitArrayTest10;.array:[I // field@0000
0001bc: 0e00               |002e: return-void
```

67. Comparing these two Dalvik bytecode sequences, it is clear that the allocation of the array object is not affected by the heuristic – the following two instructions load the array size into a register and allocate the array and are found in both sequences:

```
000160: 1300 0a00          |0000: const/16 v0, #int 10 // #a
000164: 2300 0600          |0002: new-array v0, v0, [I // class@0006
```

68. The assignment of the allocated and initialized array object to the field is also the same, and is done with a single instruction in both cases:

```
00016e: 6900 0000          |0007: sput-object v0, LinitArrayTest10;.array:[I // field@0000
```

69. The difference between the two Dalvik bytecode sequences is the initialization of the array. When the heuristic is enabled, the initialized array values are inserted into the code,

and the fill-array-data instruction is used to copy these initialized array values into the allocated array object. When the heuristic is disabled, each element of the array is initialized individually. This requires that two registers are loaded with the appropriate values for each array element – the element index and the element value. In the example shown here, there is an overlap between the set of element indexes used and the element values, enabling the same register to be used multiple times. The number of instructions required to initialize the array will grow approximately linearly with the array. Typically, it will be three instructions per array element – two constant loads and an aput instruction to actually put the right value into the correct array element. Depending on the data type, this will be a size overhead of 10-14 bytes per array element.

D. Results

70. The results of this experiment are shown in the table below. The "Array type" and "Array size" columns are self-explanatory. The two middle columns show the size in byes of the dex file both with and without the array initialization optimization. The "Bytes added" column shows the number of bytes added to the dex file when the optimization was disabled. The "Array values" column shows the values used for the array initialization. Note that the cost of array initialization without the optimization is greater when larger numbers are used – this is because it takes more Dalvik instructions to load a larger constant number into a register prior to storing the register into an array element to initialize it.

Array type	**Array size**	**Dex with optimization**	**Dex without optimization**	**Bytes added**	**Array values**
-	No array	748	-	-	None
int	[10]	904	932	28	1, 2, … 10
int	[20]	944	1052	108	1, 2, … 20
int	[100]	1268	2016	748	1, 2, … 100
int	[100]	1272	2244	972	2^{31}-1, 2^{31}-2, …
int	[1000]	4868	12816	7948	1, 2, … 1000
short	[10]	888	936	48	1, 2, … 10
short	[20]	908	1056	148	1, 2, … 20
short	[100]	1068	2016	958	1, 2, … 100
short	[100]	1076	2048	968	2^{15}-1, 2^{15}-2, …
double	[8][8]	1436	1632	196	1.0 … 135.0
boolean	[10]	876	916	40	True, false

71. The next table shows the number of bytes contributed to the dex file size by the array. This is approximated by subtracting the size of the dex file with no array defined from the size of the de file including a static array. The "with optimization" column shows the size of the array when the play execution technique is used. The "without optimization" column shows the size when play execution is disabled. The "Ratio" column shows how much the array size grows without the use of play execution.

Array type	**Array size**	**Array size with optimization**	**Array size without optimization**	**Ratio**	**Array values**
-	No array	-	-	-	None
int	[10]	156	184	1.18x	1, 2, … 10
int	[20]	196	304	1.55x	1, 2, … 20
int	[100]	520	1268	2.44x	1, 2, … 100
int	[100]	524	1496	2.85x	2^{31}-1, 2^{31}-2, …
int	[1000]	4120	12068	2.93x	1, 2, … 1000
short	[10]	140	188	1.34x	1, 2, … 10
short	[20]	160	308	1.93x	1, 2, … 20
short	[100]	320	1268	3.96x	1, 2, … 100
short	[100]	328	1300	3.96x	2^{15}-1, 2^{15}-2, …
double	[8][8]	688	884	1.28x	1.0 … 135.0
boolean	[10]	128	168	1.31x	True, false

72. By examining the Dalvik bytecode sequences presented above, it can be seen that the difference in the size of the dex file will not be completely linear with respect to the array size. This is true because the number of registers required to be initialized will depend on the set of values used by the array elements to be initialized, and the number of Dalvik bytecodes to initialize a register depends on the magnitude of the number with which the register is to be initialized. It is clear though that the array initialization optimization through play execution does have a significant impact on both the size of the dex file and the number of Dalvik bytecodes that must be executed to initialize the array.

E. '520 CONCLUSION

73. The use of a heuristic to spot static array initialization, play execution of the array initialization bytecodes and the storing of the contents of the initialized array significantly

reduces the contribution of the array to the size of the dex file, and the number of bytecodes that must be executed to initialize the array.

74. The contribution of an array to the size of a dex file can be as much as four times larger when play execution is not used.

75. The larger the array, the more size advantage is derived. The size advantage grows without limit as the array size is increased.

76. When play execution is used, the number of Dalvik bytecodes that have to be executed to initialize an array is constant. When play execution is not used, the number of bytecodes grows linearly with the number of elements in the array. This increased number of bytecodes would have a detrimental impact on application startup time.

77. For small arrays, for example, the 10 element arrays in this experiment, size difference is small. However, the number of Dalvik bytecode instructions to be executed to perform the array initialization changes dramatically, going from 1 to 25.

78. For larger arrays, the impact on dex file size is much larger, and the number of Dalvik bytecodes to be executed is also correspondingly increased.

I declare under penalty of perjury that the foregoing is true and correct.

Dated: August 6, 2011

Noel Poore

Noel Poore