EXHIBIT 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.

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.

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.

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