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

OPENING EXPERT REPORT OF JOHN C. MITCHELL REGARDING PATENT INFRINGEMENT

SUBMITTED ON BEHALF OF PLAINTIFF ORACLE AMERICA, INC.

compiles a portion of the method ("the function") first virtual machine instruction corresponds to a method.

394. Claim 8-c recites "representing said at least one native machine instruction with a new virtual machine instruction that is executed after the compiling of [the function]."

395. The evidence I discussed to show how Android satisfies claim 1-b and 1-c applies here to show how Android meets the limitation of claim 8-c.

396. Therefore, Android meets the limitations of claim 8.

Inline Infringement

397. There is a second way in which Android satisfies the limitations of claim 1. The dexopt component that runs in the Dalvik virtual machine loads virtual machine instructions into the virtual machine and replaces selected virtual machine instructions with different virtual machine instructions that reference or represent native code to be executed instead of the original virtual machine instructions. (*See, e.g.*, 5/4/2011 McFadden Dep. 154:21-156:7.)

398.

(11/23/2010 Email from Andy McFadden to Android developers (GOOGLE-04-00083078).) I was able to determine that the very long alpha-numeric expression at the bottom of the

McFadden email refers to Android's inline version of String.indexOf() which is in part the subject of the second way Android infringes the '205 patent as shown below:

```
https://android.git.kernel.org/?p=platform%2Fdalvik.git;a=commit;
h=59a434629ba06d4decf7bc88a62ae370a1935f0e

Add inline version of String.indexOf().

author Andy McFadden <fadden@android.com>  Thu, 3 Sep 2009 01:07:23 +0000
(18:07 -0700) committer Andy McFadden <fadden@android.com>  Thu, 3 Sep 2009
19:22:51 +0000 (12:22 -0700) commit 59a434629ba06d4decf7bc88a62ae370a1935f0e
tree 3b53b83996a3f8057f75716d84c74cb7acd23d67 tree | snapshot parent
06f4a19f3c6d6503e936d252fb80d6ca57b29392 commit | diff

Add inline version of String.indexOf().

This provides an inline-native version of String.indexOf(int) and
String.indexOf(int, int), i.e. the functions that work like strchr().
Has a fairly solid impact on specific benchmarks.  Might give a boost to an
app somewhere.

Added some indexOf tests to 020-string.

Added hard-coded field offsets for String.  These are verified during startup.
Improves some of our String micro-benchmarks by ~10%.
```

399. The **preamble of claim 1** recites "In a computer system, a method for increasing the execution speed of virtual machine instructions at runtime, the method comprising."

400. As demonstrated by multiple sources of evidence produced by Google concerning Android, a device that runs Android satisfies this limitation. Devices that run Android satisfy the preamble because any device that runs Android is a system. Android uses the Dalvik virtual machine to execute virtual machine bytecode instructions at runtime. The Dalvik virtual machine – and here dexopt in particular – performs and runs code resulting from certain optimizations to increase the execution speed of virtual machine instructions at runtime.

401. As demonstrated by the McFadden email and the commit log, the infringing feature of Android was added to improve performance – *e.g.*, "Improves some of our String micro-benchmarks by ~10%" is a reference to increasing the execution speed of Android's Dalvik virtual machine, thereby satisfying the preamble of claim 1.

402. Satisfaction of the preamble of claim 1 is also documented in Google's Android website, *e.g.*:

http://android.git.kernel.org/?p=platform/dalvik.git;a=blob;f=docs/dexopt.html:

Dalvik Optimization and Verification With *dexopt*

The Dalvik virtual machine was designed specifically for the Android mobile platform. The target systems have little RAM, store data on slow internal flash memory, and generally have the performance characteristics of decade-old desktop systems. They also run Linux, which provides virtual memory, processes and threads, and UID-based security mechanisms.

The features and limitations caused us to focus on certain goals:

- Class data, notably bytecode, must be shared between multiple processes to minimize total system memory usage.
- The overhead in launching a new app must be minimized to keep the device responsive.
- Storing class data in individual files results in a lot of redundancy, especially with respect to strings. To conserve disk space we need to factor this out.
- Parsing class data fields adds unnecessary overhead during class loading. Accessing data values (e.g. integers and strings) directly as C types is better.
- Bytecode verification is necessary, but slow, so we want to verify as much as possible outside app execution.
- Bytecode optimization (quickened instructions, method pruning) is important for speed and battery life.
- For security reasons, processes may not edit shared code.

The typical VM implementation uncompresses individual classes from a compressed archive and stores them on the heap. This implies a separate copy of each class in every process, and slows application startup because the code must be uncompressed (or at least read off disk in many small pieces). On the other hand, having the bytecode on the local heap makes it easy to rewrite instructions on first use, facilitating a number of different optimizations.

The goals led us to make some fundamental decisions:

- Multiple classes are aggregated into a single "DEX" file.
- DEX files are mapped read-only and shared between processes.
- Byte ordering and word alignment are adjusted to suit the local system.
- Bytecode verification is mandatory for all classes, but we want to "pre-verify" whatever we can.
- Optimizations that require rewriting bytecode must be done ahead of time.
- The consequences of these decisions are explained in the following sections.

….
dexopt

We want to verify and optimize all of the classes in the DEX file. The easiest and safest way to do this is to load all of the classes into the VM and run through them. Anything that fails to load is simply not verified or optimized. Unfortunately, this can cause allocation of some resources that are difficult to release (e.g. loading of native shared libraries), so we don't want to do it in the same virtual machine that we're running applications in.

The solution is to invoke a program called dexopt, which is really just a back door into the VM. It performs an abbreviated VM initialization, loads zero or more DEX files from the bootstrap

class path, and then sets about verifying and optimizing whatever it can from the target DEX. On completion, the process exits, freeing all resources.

It is possible for multiple VMs to want the same DEX file at the same time. File locking is used to ensure that dexopt is only run once.

….

Optimization

Virtual machine interpreters typically perform certain optimizations the first time a piece of code is used. Constant pool references are replaced with pointers to internal data structures, operations that always succeed or always work a certain way are replaced with simpler forms. Some of these require information only available at runtime, others can be inferred statically when certain assumptions are made.

The Dalvik optimizer does the following:

- For virtual method calls, replace the method index with a vtable index.
- For instance field get/put, replace the field index with a byte offset. Also, merge the boolean / byte / char / short variants into a single 32-bit form (less code in the interpreter means more room in the CPU I-cache).
- Replace a handful of high-volume calls, like String.length(), with "inline" replacements. This skips the usual method call overhead, directly switching from the interpreter to a native implementation.
- Prune empty methods. The simplest example is Object.<init>, which does nothing, but must be called whenever any object is allocated. The instruction is replaced with a new version that acts as a no-op unless a debugger is attached.
- Append pre-computed data. For example, the VM wants to have a hash table for lookups on class name. Instead of computing this when the DEX file is loaded, we can compute it now, saving heap space and computation time in every VM where the DEX is loaded.

All of the instruction modifications involve replacing the opcode with one not defined by the Dalvik specification. This allows us to freely mix optimized and unoptimized instructions. The set of optimized instructions, and their exact representation, is tied closely to the VM version.

Most of the optimizations are obvious "wins". The use of raw indices and offsets not only allows us to execute more quickly, we can also skip the initial symbolic resolution. Pre-computation eats up disk space, and so must be done in moderation.

There are a couple of potential sources of trouble with these optimizations. First, vtable indices and byte offsets are subject to change if the VM is updated. Second, if a superclass is in a different DEX, and that other DEX is updated, we need to ensure that our optimized indices and offsets are updated as well. A similar but more subtle problem emerges when user-defined class loaders are employed: the class we actually call may not be the one we expected to call.

These problems are addressed with dependency lists and some limitations on what can be optimized.

(*See also, e.g.*, dalvik\docs\ embedded-vm-control.html#verifier (“The system tries to pre-verify all classes in a DEX file to reduce class load overhead, and performs a series of optimizations to improve runtime performance. Both of these are done by the dexopt command, either in the build

system or by the installer. On a development device, dexopt may be run the first time a DEX file is used and whenever it or one of its dependencies is updated ("just-in-time" optimization and verification).").)

403. Therefore, Android meets the **preamble of claim 1.**

404. **Limitation [1-a] of claim 1** recites "receiving a first virtual machine instruction."

405. Android's running dexopt tool receives a first virtual machine instruction as shown by the Android source code excerpted below. (I note that the code samples are taken from the Froyo version of Android available in December 2010. Since then, the infringing Android DexOptimize.c source code has been split between dalvik\vm\analysis\DexPrepare.c and dalvik\vm\analysis\Optimize.c.)

```
http://android.git.kernel.org/?p=platform/dalvik.git;a=blob;f=vm/analysis/DexOptimize.c
1486 /*
1487 * Run through all classes that were successfully loaded from this DEX
1488 * file and optimize their code sections.
1489 */
1490 static void optimizeLoadedClasses(DexFile* pDexFile)
1491 {
1492 u4 count = pDexFile->pHeader->classDefsSize;
1493 u4 idx;
1494 InlineSub* inlineSubs = NULL;
…
1499 inlineSubs = createInlineSubsTable();
1500
1501 for (idx = 0; idx < count; idx++) {
1502 const DexClassDef* pClassDef;
1503 const char* classDescriptor;
1504 ClassObject* clazz;
1505
1506 pClassDef = dexGetClassDef(pDexFile, idx);
1507 classDescriptor = dexStringByTypeIdx(pDexFile, pClassDef->classIdx);
1508
1509 /* all classes are loaded into the bootstrap class loader */
1510 clazz = dvmLookupClass(classDescriptor, NULL, false);
1511 if (clazz != NULL) {
1512 if ((pClassDef->accessFlags & CLASS_ISPREVERIFIED) == 0 &&
1513 gDvm.dexOptMode == OPTIMIZE_MODE_VERIFIED)
1514 {
1515 LOGV("DexOpt: not optimizing '%s': not verified\n",
1516 classDescriptor);
1517 } else if (clazz->pDvmDex->pDexFile != pDexFile) {
1518 /* shouldn't be here -- verifier should have caught */
1519 LOGD("DexOpt: not optimizing '%s': multiple definitions\n",
1520 classDescriptor);
1521 } else {
1522 optimizeClass(clazz, inlineSubs);
1523
1524 /* set the flag whether or not we actually did anything */
1525 ((DexClassDef*)pClassDef)->accessFlags |=
1526 CLASS_ISOPTIMIZED;
1527 }
1528 } else {
1529 LOGV("DexOpt: not optimizing unavailable class '%s'\n",
1530 classDescriptor);
```

```
1531 }
1532 }
1533
1534 free(inlineSubs);
1535 }
```

At line 1522, dexopt calls `optimizeClass()` for each class in DEX file, passing in a table of inline substitutions, inlineSubs. (*See also, e.g.*, dalvik\vm\analysis\DexPrepare.c and dalvik\vm\analysis\Optimize.c (*e.g.*, dvmCreateInlineSubsTable routines).)

406. The invoked optimizeClass function in turn invokes optimizeMethod on each direct or virtual method in the incoming DEX class at lines 1544-1545 and 1558-1549. (*See also, e.g.*, dalvik\vm\analysis\Optimize.c.)

```
http://android.git.kernel.org/?p=platform/dalvik.git;a=blob;f=vm/analysis/DexOptimize.c
1537 /*
1538 * Optimize the specified class.
1539 */
1540 static void optimizeClass(ClassObject* clazz, const InlineSub* inlineSubs)
1541 {
1542 int i;
1543
1544 for (i = 0; i < clazz->directMethodCount; i++) {
1545 if (!optimizeMethod(&clazz->directMethods[i], inlineSubs))
1546 goto fail;
1547 }
1548 for (i = 0; i < clazz->virtualMethodCount; i++) {
1549 if (!optimizeMethod(&clazz->virtualMethods[i], inlineSubs))
1550 goto fail;
1551 }
1552
1553 return;
…
```

407. The function optimizeMethod optimizes **instructions** in a method, processing one virtual machine instruction at a time, as the Android developers explain by comment shown below:

```
http://android.git.kernel.org/?p=platform/dalvik.git;a=blob;f=vm/analysis/DexOptimize.c
1559 /*
1560 * Optimize instructions in a method.
1561 *
1562 * Returns "true" if all went well, "false" if we bailed out early when
1563 * something failed.
1564 */
1565 static bool optimizeMethod(Method* method, const InlineSub* inlineSubs)
1566 {
1567 u4 insnsSize;
1568 u2* insns;
1569 u2 inst;
1570
1571 if (dvmIsNativeMethod(method) || dvmIsAbstractMethod(method))
1572 return true;
1573
1574 insns = (u2*) method->insns;
1575 assert(insns != NULL);
1576 insnsSize = dvmGetMethodInsnsSize(method);
```

```
1577
1578 while (insnsSize > 0) {
1579 int width;
1580
1581 inst = *insns & 0xff;
1582
1583 switch (inst) {
...
1645 case OP_INVOKE_DIRECT_RANGE:
1646 rewriteExecuteInlineRange(method, insns, METHOD_DIRECT, inlineSubs);
1647 break;
...
1652 case OP_INVOKE_STATIC_RANGE:
1653 rewriteExecuteInlineRange(method, insns, METHOD_STATIC, inlineSubs);
1654 break;
...
1656 default:
1657 // ignore this instruction
1658 ;
1659 }
...
1679 return true;
1680 }
```

At lines 1646 and 1653, Android invokes `rewriteExecuteInlineRange()` for each OP_INVOKE_DIRECT_RANGE or OP_INVOKE_STATIC_RANGE virtual machine instruction. (*See also, e.g.*, dalvik\vm\analysis\Optimize.c.)

408. Therefore, Android meets l**imitation [1-a] of claim 1**.

409. **Limitation [1-b] of claim 1** recites "generating, at runtime, a new virtual machine instruction that represents or references one or more native instructions that can be executed instead of said first virtual machine instruction."

410. The documented descriptions of dexopt discussed above show that dexopt runs at runtime. *E.g.*:

```
http://android.git.kernel.org/?p=platform/dalvik.git;a=blob;f=vm/analysis/DexOptimize.c
1486 /*
1487 * Run through all classes that were successfully loaded from this DEX
1488 * file and optimize their code sections.
1489 */
```

411. *(See also, e.g.*, dalvik\docs\ embedded-vm-control.html#verifier ("The system tries to pre-verify all classes in a DEX file to reduce class load overhead, and performs a series of optimizations to improve runtime performance. Both of these are done by the dexopt command, either in the build system or by the installer. On a development device, dexopt may be run the first time a DEX file is used and whenever it or one of its dependencies is updated ("just-in-time" optimization and verification).");

XIII. CONCLUSION

767. For the foregoing reasons, it is my opinion that Android infringes:

- Claims 11, 12, 15, 17, 22, 27, 29, 38, 39, 40, and 41 of United States Patent No. RE38,104;
- Claims 1, 2, 3, and 8 of United States Patent No. 6,910,205;
- Claims 1, 6, 7, 12, 13, 15, and 16 of United States Patent No. 5,966,702;
- Claims 1, 4, 8, 12, 14, and 20 of United States Patent No. 6,061,520;
- Claims 1, 4, 6, 10, 13, 19, 21, and 22 of United States Patent No. 7,426,720;
- Claims 10 and 11 of United States Patent No. 6,125,447; and
- Claims 13, 14, and 15 of United States Patent No. 6,192,476

It is also my opinion that Google is liable for direct and indirect infringement in the manner described above.

768. For the forgoing reasons, it is my opinion that the patents-in-suit form the basis for consumer demand for Android by developers and end-users.

769. For the forgoing reasons, it is my opinion that once Google decided to adopt the Java execution model in Android, the patents-in-suit became necessary to Android achieving satisfactory performance and security.

Dated: August 8, 2011

John C. Mitchell