GOOGLE'S MOTION TO STRIKE PORTIONS OF THE MITCHELL PATENT REPORT

CIV. NO. CV 10-03561-WHA

Exhibit A

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.

191. Because it would take substantial effort and resources to make significant changes to core Android operating system functions that both outperformed the code that Google already provides and still conform to the CDD and pass the CTS, I would not expect any such changes to be made. To confirm, I reviewed source code for particular Android devices (the Nexus One, Nexus S, HTC Droid Incredible 2, LG Optimus, Samsung Captivate, and Motorola Atrix) that is made available by Android device manufacturers on their open source websites. Versions of source code files, discussed in this report with respect to my infringement analysis, from the OEM websites were compared to the "canonical" Google version from the Android git repository, and there were no changes to the OEM versions that would alter the infringement analysis of Google's canonical version. This supports my conclusion that Android-branded phones are running Google-authored software, at least with respect to the code that causes the infringement.

229. All of the analyses below concerning infringement are to be read together with the material in the claim chart of Exhibit A attached to Oracle's infringement contentions submitted to Google on April 1, 2011. I participated in the analysis and preparation of the Exhibit A chart. I agree with the conclusions of the chart and the evidence supporting those conclusions. While my report contains a narrative-style infringement analysis, this analysis is intended to accompany the additional information supplied in the charts that in some cases provides more details.

230. The infringement evidence illustrated below is exemplary and not exhaustive. The cited examples are taken from Android 2.2, 2.3, and Google's Android websites.[3] My analysis applies to all versions of Android having similar or nearly identical code or documentation, including past and expected future releases. I understand that the publicly released versions of Android before version 2.2 operate as I describe below; I have not been given the opportunity to analyze versions of Android from 3.0 and beyond.

231. In my opinion, Google's Android literally infringes the asserted claims of the '104 patent.

232. The Android code cited below necessarily infringes when it runs because, for example, (1) "when a .dex files arrives on a device it will have symbolic references to methods and fields, but afterwards it might just be a simple, a simple integer vtable offset so that when, for invoking a method, instead of having to do say a string-based lookup, it can just simply index into a vtable" (Google I/O 2008 Video entitled "Dalvik Virtual Machine Internals," presented by Dan Bornstein (Google Android Project), *available at* http://developer.android.com/videos/index.html#v=ptjedOZEXPM ("Dalvik Video")) and (2) "'constant pool' references" are "resolve[d]" "into pointers to VM structs" (*e.g.*,

[3] The cited source code examples are taken from http://android.git.kernel.org/. The citations are shortened and mirror the file paths shown in http://android.git.kernel.org/. For example, "dalvik\vm\native\InternalNative.c" maps to "[platform/dalvik.git] / vm / native / InternalNative.c" (accessible at http://android.git.kernel.org/?p=platform/dalvik.git;a=blob;f=vm/native/InternalNative.c). Google has apparently made modifications to certain source code files since Oracle's Preliminary Infringement Contentions were served on December 2, 2010. As such, file paths may refer to earlier versions of Android than what is immediately available at http://android.git.kernel.org/.

\dalvik\vm\oo\Resolve.h). Moreover, much of the code cited below is executed not only as applications run, but every time a device running Android starts up.

233. For example, Google employee and Android developer Dan Bornstein all but admitted that Android infringes the '104 patent asserted claims when he stated "when a .dex files arrives on a device it will have symbolic references to methods and fields, but afterwards it might just be a simple, a simple integer vtable offset so that when, for invoking a method, instead of having to do say a string-based lookup, it can just simply index into a vtable." (Google I/O 2008 Video entitled "Dalvik Virtual Machine Internals," presented by Dan Bornstein (Google Android Project), *available at* http://developer.android.com/videos/index.html#v=ptjedOZEXPM ("Dalvik Video").)

234. Google Android developers stated in Android code that Android resolves "symbolic references":

```
/*
 * Link (prepare and resolve).  Verification is deferred until later.
 *
 * This converts symbolic references into pointers.  It's independent of
 * the source file format.
     (Comments for "dvmLinkClass" routine in dalvik\vm\oo\Class.c)
```

235. Google Android developers stated in Android code that Android "resolves" references:

```
/*
 * Resolve classes, methods, fields, and strings.
 *
     (Comments for dalvik\vm\oo\Resolve.c)

/*
 * Resolve an instance field reference.
 *
     (Comments for "dvmResolveInstField" routine in dalvik\vm\oo\Resolve.c)
```

236. The substance of the infringement evidence cited in Claim 11 applies to each asserted claim because the evidence is not limited to a particular form of accused infringement. Android infringes in at least two ways, which are both detailed below.

237. The **preamble of claim 11** recites "An apparatus comprising." Devices that run Android or the Android SDK satisfy the preamble because any device that runs Android is an apparatus.

238. **Limitation [11-a] of claim 11** recites "a memory containing intermediate form object code constituted by a set of instructions, certain of said instructions containing one or more symbolic references."

239. As demonstrated by multiple sources of evidence produced by Google concerning Android, a device that runs Android or the Android SDK satisfies this limitation because it has a memory containing intermediate form object code constituted by a set of instructions.

240. Google invited Android developers to program applications in Java. Google's own Android talks and documentation explain that the Java source code must first be compiled into Java bytecode, which in turn gets run through Google's Android dx tool to translate the Java bytecode into Dalvik bytecode (called .dex format where .dex stands for Dalvik Executable). Android's virtual machine – called the Dalvik virtual machine – optimizes the .dex format bytecode into optimized .dex bytecode (referred to as .odex).

241. The Dalvik bytecode contains intermediate object code constituted by a set of instructions. This is because Java source code constitutes a set of instructions. When the Java source code is compiled, the resulting Java bytecode contains intermediate object code constituted by a set of instructions that contain symbolic references. So, too, does a .dex file, which is just a translation of the Java bytecode – albeit representing instructions for the Dalvik virtual machine instruction set rather than the Java virtual machine instruction set – and as such also contains intermediate object code constituted by a set of instructions that contain symbolic references.

242. When the Dalvik bytecode is stored on, saved on, or loaded onto a device, it is contained in memory.

243. This is documented in Google's Android website, *e.g.*:

Android Glossary Definition for ".dex file," *available at* http://developer.android.com/guide/appendix/glossary.html:

> .dex file
> Compiled Android application code file.
> Android programs are compiled into .dex (Dalvik Executable) files, which are in turn zipped into a single .apk file on the device. .dex files can be created by automatically translating compiled applications written in the Java programming language.

Android Basics, entitled "What is Android?," *available at* http://developer.android.com/guide/basics/what-is-android.html:

> **What is Android?**
>
> Android is a software stack for mobile devices that includes an operating system, middleware and key applications. The Android SDK provides the tools and APIs necessary to begin developing applications on the Android platform using the Java programming language.
> …
>
> **Applications**
>
> Android will ship with a set of core applications including an email client, SMS program, calendar, maps, browser, contacts, and others. All applications are written using the Java programming language.
> …
>
> **Android Runtime**
>
> Android includes a set of core libraries that provides most of the functionality available in the core libraries of the Java programming language.
>
> Every Android application runs in its own process, with its own instance of the Dalvik virtual machine. Dalvik has been written so that a device can run multiple VMs efficiently. The Dalvik VM executes files in the Dalvik Executable (.dex) format which is optimized for minimal memory footprint. The VM is register-based, and runs classes compiled by a Java language compiler that have been transformed into the .dex format by the included "dx" tool.
>
> The Dalvik VM relies on the Linux kernel for underlying functionality such as threading and low-level memory management.

See, e.g., dalvik\docs\dexopt.html; *see also* 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.

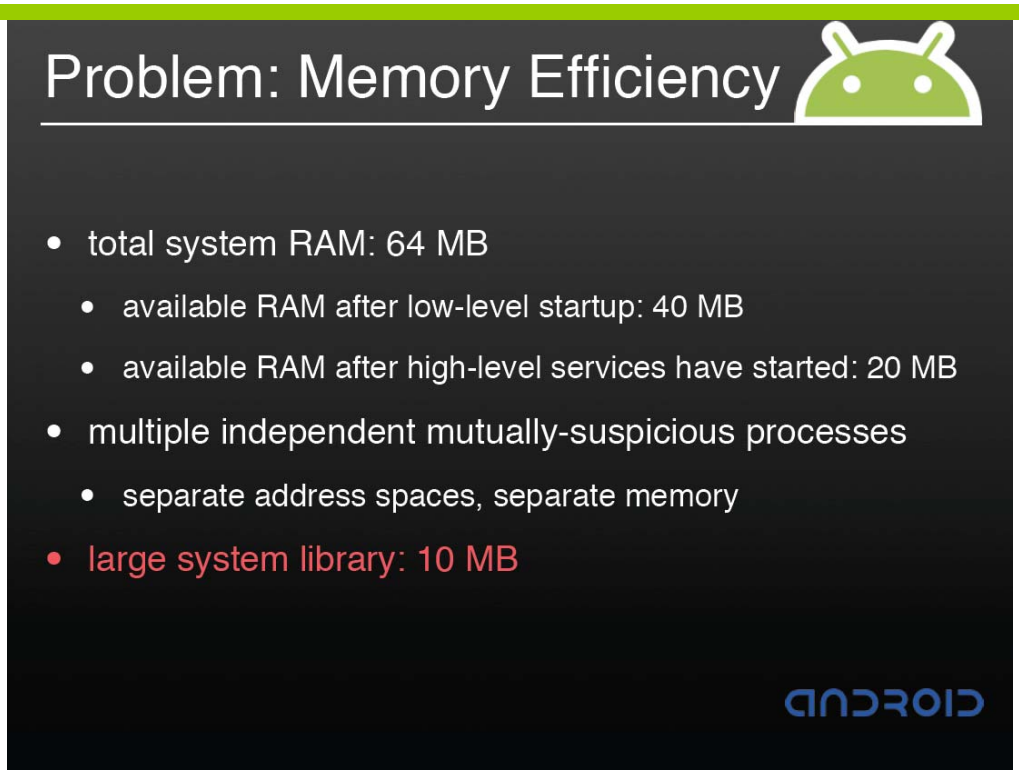
The goals led us to make some fundamental decisions:

- 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.

….

244. Google's Dalvik and .dex creator Dan Bornstein has also made statements demonstrating Android's satisfaction of claim limitation **11-a**, *e.g.*:

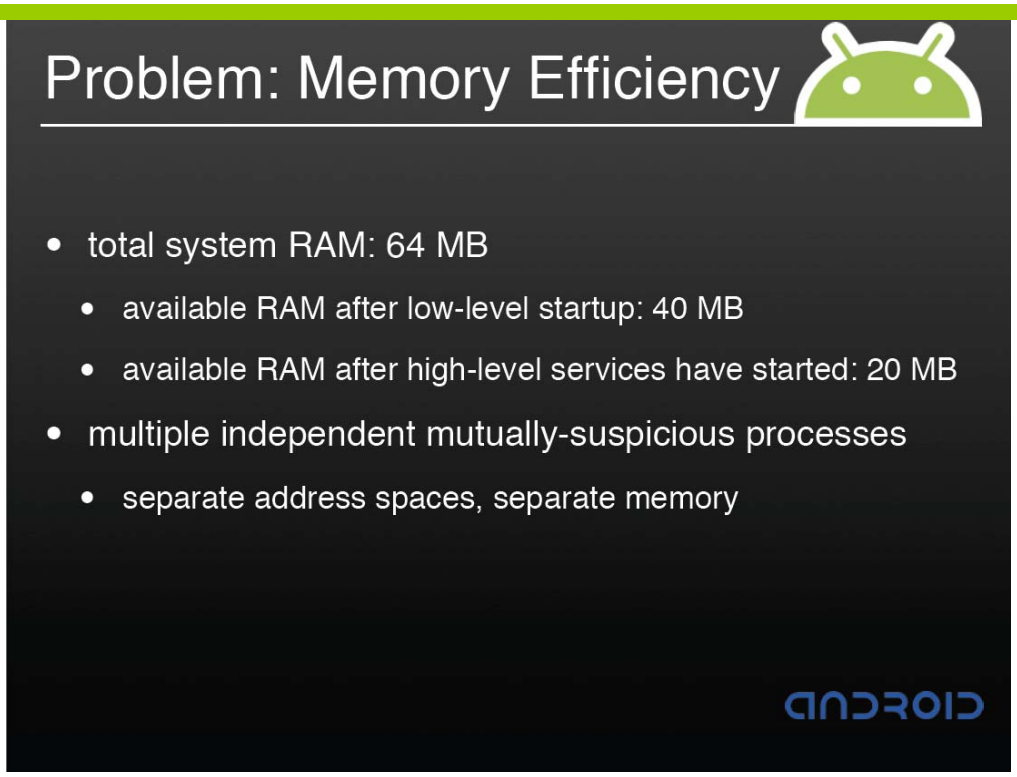
See, e.g., Google I/O 2008 Video entitled "Dalvik Virtual Machine Internals," presented by Dan Bornstein (Google Android Project), *available at* http://developer.android.com/videos/index.html#v=ptjedOZEXPM ("Dalvik Video"):



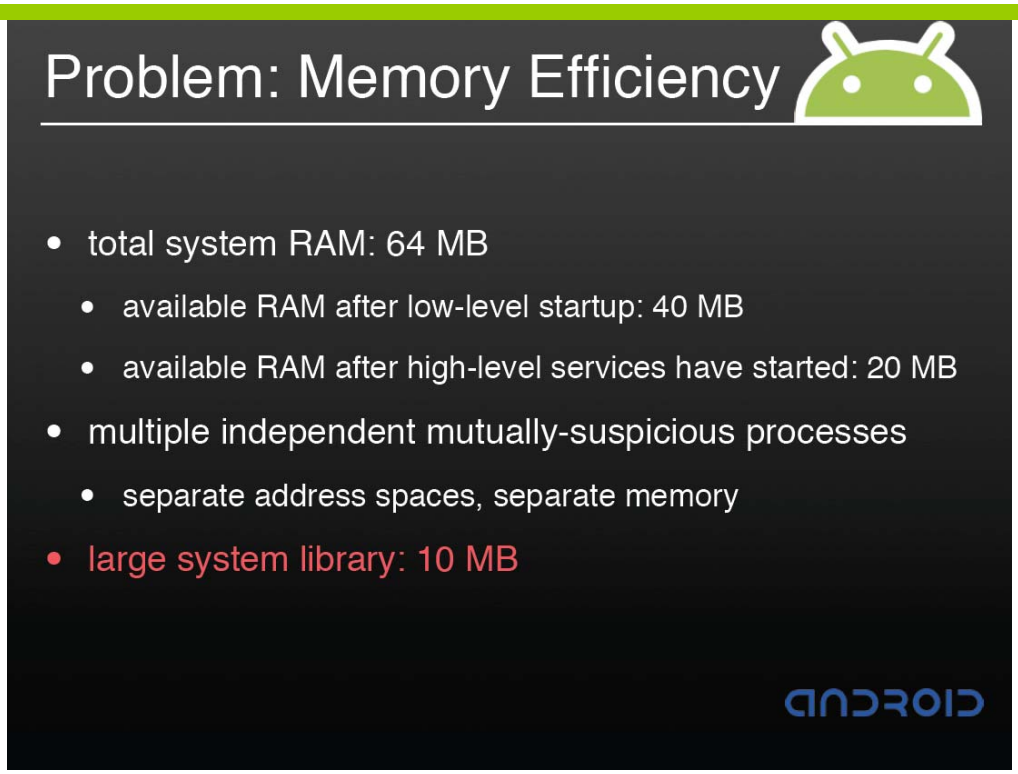
- at 1:22 under "The Big Picture" ("Very briefly, Android is the new platform for mobile devices and it really is the complete stack, includes layers from the OS kernel at the bottom and drivers up through an application framework at the top and it even includes a few applications. You write your applications in the Java programming language and they get translated after compilation into a form that runs on the Dalvik virtual machine.").

- at 2:52 under "What is the Dalvik VM?" ("So the virtual machine, again, is designed based on the constraints of the platform and you can see a few of the key ones. We're assuming, not a particularly powerful CPU, not very much RAM especially by say today's desktop standards. An easy way to think about it is as approximately equivalent to like a late 90s desktop machine with a little more modern operating system, but with one very important constraint.").

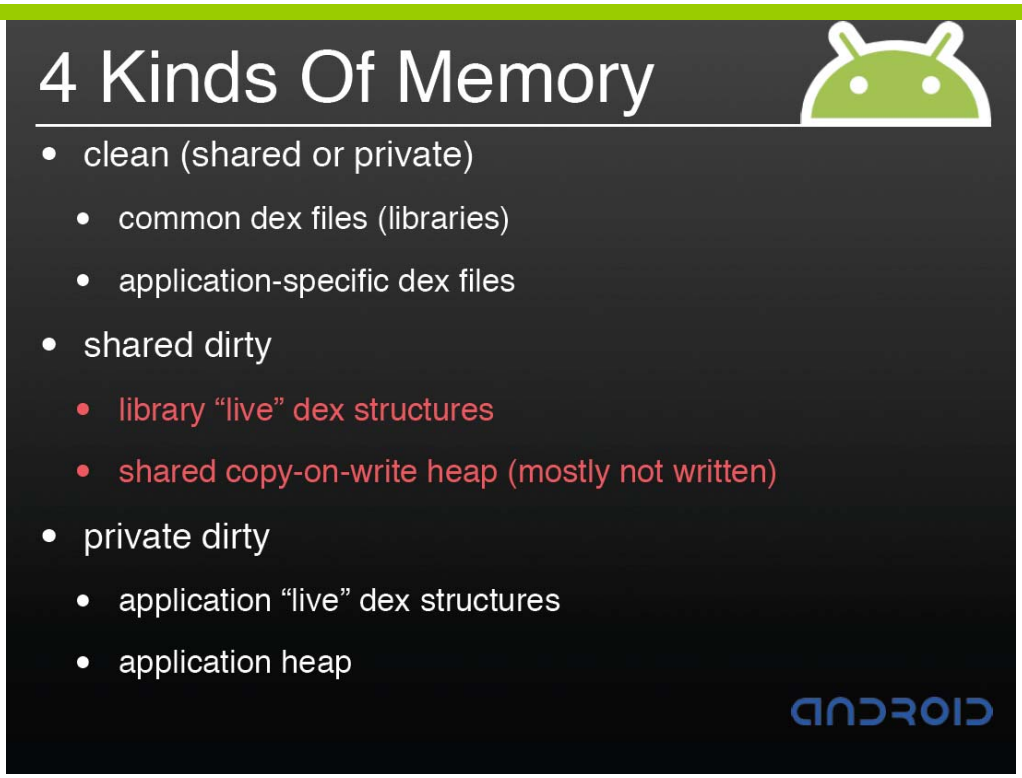
- at 4:06 under "Problem: Memory Efficiency" ("So, in particular this is, this is kind of how a low end Android device is gonna look in terms of, you know, system characteristics. So, you know, once everything is started up on the system we're not really expecting there to be that much memory left for applications and, of course, so we try to make the most of that. But one wrinkle in the works is that our, the Android platform security relies on modern process separation. So each application is running in a separate process. There's a separate address space. It has separate memory and apps are not allowed to interfere with each other at that level and so that means that unless you do something special that 20 megs really isn't gonna go far at all.").



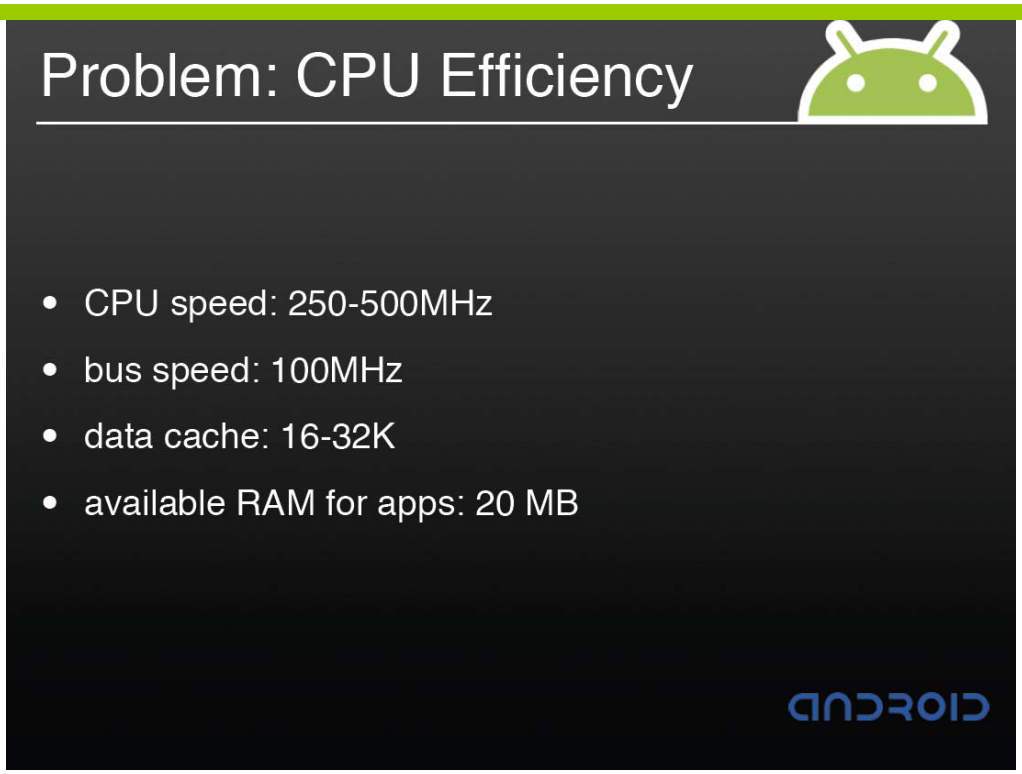
- at 5:05 under "Problem: Memory Efficiency" ("And in addition to this modern platform that, we try to make it, you know, have a rich, have a rich set of APIs for developers to use, we have a fairly large system library. And so again, if you don't do anything special, well, with a 10 meg library, 20 megs left for apps, that really, really doesn't leave much space at all. And I think I had a previous slide, we don't have swap space. So I just wanna emphasize that, so there's no, if you have 64 megs of RAM, you have 64 megs of RAM and that's kind of the size of it. Okay.").



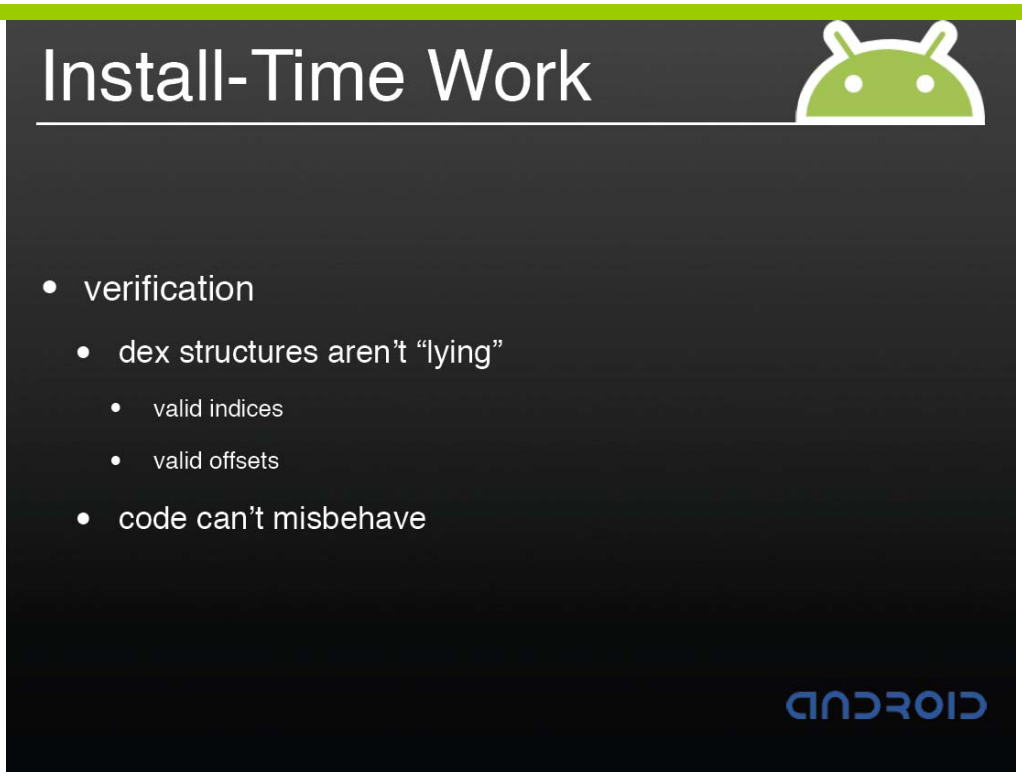
- at 15:38 under "4 Kinds of Memory" ("So our goal, again, is to get as much, as much memory to be mapped clean as possible, but we at least have this out for where we really do have to allocate that we can reduce the cost in terms of the whole system performance.").



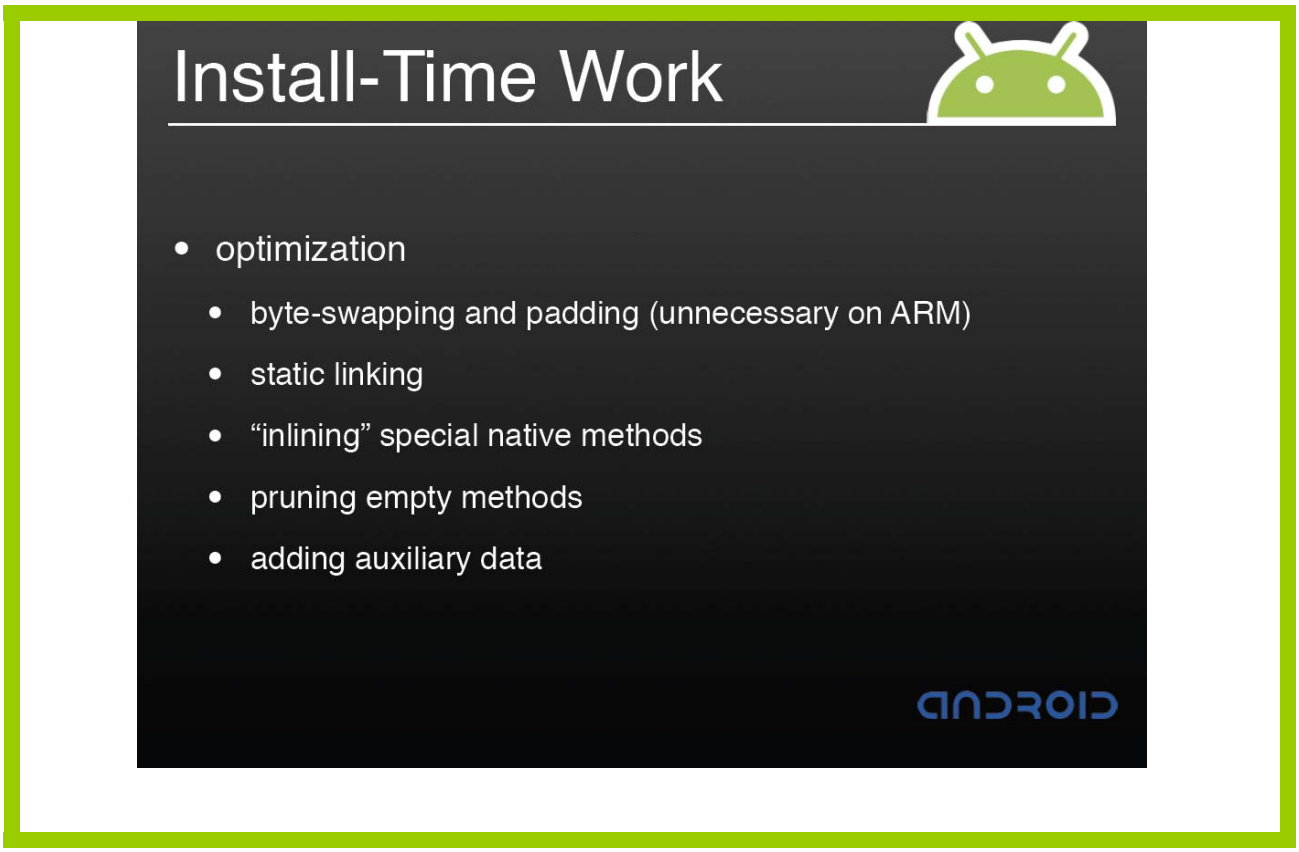
- at 19:07 under "Problem: CPU Efficiency" ("Again, as I said at the beginning, we're running on a platform or expecting to run on a platform that looks like what you might have had on your desktop 10 years ago. And, you know, you can see that it's a fairly slow bus, almost no data cache at all and I just wanna re-emphasize that there's very little RAM for an app, for applications once you consider all of the things that your device is doing, say, as a phone. It has to answer phone calls, it has to be able to take and send SMSs. All of these things are essential services as far as the user is concerned.").



- at 21:54 under "Install-Time Work" ("So, what are we doing to actually be efficient on the platform? So, first of all, when an application gets installed and also when the system itself gets installed, the platform will, the system will do a lot of work up front to avoid doing work at runtime. So one of the major things we do is verification of .dex files and what this means is that as a, as a type safe, reference safe runtime we want to ensure that the code that we're running doesn't violate the constraints of the system. It doesn't violate type safety, it doesn't, it doesn't, it doesn't violate reference safety. And for Android, this is really more about minimizing the app, the impact of bugs in an application as opposed to being a security consideration in and of itself.").



- at 23:35 under "Install-Time Work" ("We do optimization. And, so the first time that a .dex file lands on a device, we do that verification work, we also, we also augment that file, if we have to we will do byte swapping and pad out structures and in addition, we have a bunch of other things that we do such that when it comes time to run, we can run that much faster. So as an example of static linking, before, when a .dex files arrives on a device it will have symbolic references to methods and fields, but afterwards it might just be a simple, a simple integer vtable offset so that when, for invoking a method, instead of having to do say a string-based lookup, it can just simply index into a vtable. And just as another example, you are probably aware that the constructor for java.lang.object has nothing, does nothing inside it and the system can tell. So instead of, instead of actually doing that any time you're constructing an object, we know to avoid just making that call and that actually does make a significant performance impact.").



245. **Limitation [11-b] of claim 11** recites “and a processor configured to execute said instructions containing one or more symbolic references by determining a numerical reference corresponding to said symbolic reference, storing said numerical references, and obtaining data in accordance to said numerical references.”

246. An Android device is configured to run the Dalvik virtual machine upon startup. (In fact, an Android device will run multiple instances of the Dalvik virtual machine.) The code implementing the Dalvik virtual machine configures the hardware processor on the Android device to execute the .dex formatted bytecode or the .odex formatted bytecode as discussed above. As such, Android meets the limitation of “a processor configured to execute said instructions containing one or more symbolic references.”

247. The Android platform contains source code for resolving symbolic references, storing the resulting numeric references, and using the stored numeric references to obtain data. Both the dexopt component and the Dalvik bytecode interpreter resolve symbolic references and

store the resulting numeric references. The bytecode interpreter uses stored numeric references to obtain data.

248. The '104 patent is infringed because Android's dexopt optimizeMethod looks at every instruction in a method and tries to resolve symbolic references and determines, stores, or replaces symbolic references with corresponding numeric reference (*i.e.* Android quickens the instructions where Android can). (*See* 5/4/2011 McFadden Dep. 148:1-5, 153:17-154:17.) For the purpose of illustrating infringement by example, I focus on the OP_IGET (word-sized field fetch) instruction, but Android code handles other sizes of field fetches, field store, static field fetch and store, and various kinds of invocation.

249. The method optimizeMethod excerpted below "[o]ptimize[s] instructions in a method" and "does a single pass through the code, examining each instruction" as the Android developer comments indicate.

250. To be clear, Android Optimize.c line 196 identifies integer field fetch instructions in a switch statement and Android Optimize.c line 201 shows a decision to change the integer field fetch instruction to the corresponding quick instruction. In this process, a symbolic reference is resolved to a numerical reference. Android Optimize.c line 235 shows how instruction is changed by calling rewriteInstField. (The rest of the method handles integer field puts, static field gets and puts, and invoke instructions.)

```
http://android.git.kernel.org/?p=platform/dalvik.git;a=blob;f=vm/analysis/Optimize.c#l156
156 /*
157  * Optimize instructions in a method.
158  *
159  * This does a single pass through the code, examining each instruction.
160  *
161  * This is not expected to fail if the class was successfully verified.
162  * The only significant failure modes occur when an "essential" update fails,
163  * but we can't generally identify those: if we can't look up a field,
164  * we can't know if the field access was supposed to be handled as volatile.
165  *
166  * Instead, we give it our best effort, and hope for the best.  For 100%
167  * reliability, only optimize a class after verification succeeds.
168  */
169 static void optimizeMethod(Method* method, bool essentialOnly)
170 {
171     u4 insnsSize;
172     u2* insns;
173     u2 inst;
174
175     if (!gDvm.optimizing && !essentialOnly) {
176         /* unexpected; will force copy-on-write of a lot of pages */
```

```
177         LOGD("NOTE: doing full bytecode optimization outside dexopt\n");
178     }
179
180     if (dvmIsNativeMethod(method) || dvmIsAbstractMethod(method))
181         return;
182
183     insns = (u2*) method->insns;
184     assert(insns != NULL);
185     insnsSize = dvmGetMethodInsnsSize(method);
186
187     while (insnsSize > 0) {
188         OpCode quickOpc, volatileOpc = OP_NOP;
189         int width;
190         bool notMatched = false;
191
192         inst = *insns & 0xff;
193
194         /* "essential" substitutions, always checked */
195         switch (inst) {
196         case OP_IGET:
197         case OP_IGET_BOOLEAN:
198         case OP_IGET_BYTE:
199         case OP_IGET_CHAR:
200         case OP_IGET_SHORT:
201             quickOpc = OP_IGET_QUICK;
202             if (gDvm.dexOptForSmp)
203                 volatileOpc = OP_IGET_VOLATILE;
204             goto rewrite_inst_field;
205         case OP_IGET_WIDE:
206             quickOpc = OP_IGET_WIDE_QUICK;
207             volatileOpc = OP_IGET_WIDE_VOLATILE;
208             goto rewrite_inst_field;
209         case OP_IGET_OBJECT:
210             quickOpc = OP_IGET_OBJECT_QUICK;
211             if (gDvm.dexOptForSmp)
212                 volatileOpc = OP_IGET_OBJECT_VOLATILE;
213             goto rewrite_inst_field;
214         case OP_IPUT:
215         case OP_IPUT_BOOLEAN:
216         case OP_IPUT_BYTE:
217         case OP_IPUT_CHAR:
218         case OP_IPUT_SHORT:
219             quickOpc = OP_IPUT_QUICK;
220             if (gDvm.dexOptForSmp)
221                 volatileOpc = OP_IPUT_VOLATILE;
222             goto rewrite_inst_field;
223         case OP_IPUT_WIDE:
224             quickOpc = OP_IPUT_WIDE_QUICK;
225             volatileOpc = OP_IPUT_WIDE_VOLATILE;
226             goto rewrite_inst_field;
227         case OP_IPUT_OBJECT:
228             quickOpc = OP_IPUT_OBJECT_QUICK;
229             if (gDvm.dexOptForSmp)
230                 volatileOpc = OP_IPUT_OBJECT_VOLATILE;
231 rewrite_inst_field:
232             if (essentialOnly)
233                 quickOpc = OP_NOP;
234             if (quickOpc != OP_NOP || volatileOpc != OP_NOP)
235                 rewriteInstField(method, insns, quickOpc, volatileOpc);
236             break;
237
238         case OP_SGET_WIDE:
239             volatileOpc = OP_SGET_WIDE_VOLATILE;
240             goto rewrite_static_field;
241         case OP_SPUT_WIDE:
242             volatileOpc = OP_SPUT_WIDE_VOLATILE;
243 rewrite_static_field:
244             rewriteStaticField(method, insns, volatileOpc);
245             break;
246         default:
```

```
247             notMatched = true;
248             break;
249         }
250
251         if (notMatched && gDvm.dexOptForSmp) {
252             /* additional "essential" substitutions for an SMP device */
253             switch (inst) {
254             case OP_SGET:
255             case OP_SGET_BOOLEAN:
256             case OP_SGET_BYTE:
257             case OP_SGET_CHAR:
258             case OP_SGET_SHORT:
259                 volatileOpc = OP_SGET_VOLATILE;
260                 goto rewrite_static_field2;
261             case OP_SGET_OBJECT:
262                 volatileOpc = OP_SGET_OBJECT_VOLATILE;
263                 goto rewrite_static_field2;
264             case OP_SPUT:
265             case OP_SPUT_BOOLEAN:
266             case OP_SPUT_BYTE:
267             case OP_SPUT_CHAR:
268             case OP_SPUT_SHORT:
269                 volatileOpc = OP_SPUT_VOLATILE;
270                 goto rewrite_static_field2;
271             case OP_SPUT_OBJECT:
272                 volatileOpc = OP_SPUT_OBJECT_VOLATILE;
273 rewrite_static_field2:
274                 rewriteStaticField(method, insns, volatileOpc);
275                 notMatched = false;
276                 break;
277             default:
278                 assert(notMatched);
279                 break;
280             }
281         }
282
283         /* non-essential substitutions */
284         if (notMatched && !essentialOnly) {
285             switch (inst) {
286             case OP_INVOKE_VIRTUAL:
287                 if (!rewriteExecuteInline(method, insns, METHOD_VIRTUAL)) {
288                     rewriteVirtualInvoke(method, insns,
289                             OP_INVOKE_VIRTUAL_QUICK);
290                 }
291                 break;
292             case OP_INVOKE_VIRTUAL_RANGE:
293                 if (!rewriteExecuteInlineRange(method, insns, METHOD_VIRTUAL)) {
294                     rewriteVirtualInvoke(method, insns,
295                             OP_INVOKE_VIRTUAL_QUICK_RANGE);
296                 }
297                 break;
298             case OP_INVOKE_SUPER:
299                 rewriteVirtualInvoke(method, insns, OP_INVOKE_SUPER_QUICK);
300                 break;
301             case OP_INVOKE_SUPER_RANGE:
302                 rewriteVirtualInvoke(method, insns, OP_INVOKE_SUPER_QUICK_RANGE);
303                 break;
304
305             case OP_INVOKE_DIRECT:
306                 if (!rewriteExecuteInline(method, insns, METHOD_DIRECT)) {
307                     rewriteEmptyDirectInvoke(method, insns);
308                 }
309                 break;
310             case OP_INVOKE_DIRECT_RANGE:
311                 rewriteExecuteInlineRange(method, insns, METHOD_DIRECT);
312                 break;
313
314             case OP_INVOKE_STATIC:
315                 rewriteExecuteInline(method, insns, METHOD_STATIC);
316                 break;
```

```
317             case OP_INVOKE_STATIC_RANGE:
318                 rewriteExecuteInlineRange(method, insns, METHOD_STATIC);
319                 break;
320
321             default:
322                 /* nothing to do for this instruction */
323                 ;
324             }
325         }
326
327         width = dexGetInstrOrTableWidthAbs(gDvm.instrWidth, insns);
328         assert(width > 0);
329
330         insns += width;
331         insnsSize -= width;
332     }
333
334     assert(insnsSize == 0);
335 }
```

251. The Android developers explain rewriteInstField, called in the code above, as follows:

```
http://android.git.kernel.org/?p=platform/dalvik.git;a=blob;f=vm/analysis/Optimize.c#l620
620 /*
621  * Rewrite an iget/iput instruction.  These all have the form:
622  *   op vA, vB, field@CCCC
623  *
624  * Where vA holds the value, vB holds the object reference, and CCCC is
625  * the field reference constant pool offset.  For a non-volatile field,
626  * we want to replace the opcode with "quickOpc" and replace CCCC with
627  * the byte offset from the start of the object.  For a volatile field,
628  * we just want to replace the opcode with "volatileOpc".
629  *
630  * If "volatileOpc" is OP_NOP we don't check to see if it's a volatile
631  * field.  If "quickOpc" is OP_NOP, and this is a non-volatile field,
632  * we don't do anything.
633  *
634  * "method" is the referring method.
635  */
636 static bool rewriteInstField(Method* method, u2* insns, OpCode quickOpc,
637     OpCode volatileOpc)
638 {
639     ClassObject* clazz = method->clazz;
640     u2 fieldIdx = insns[1];
641     InstField* instField;
642
643     instField = dvmOptResolveInstField(clazz, fieldIdx, NULL);
644     if (instField == NULL) {
645         LOGI("DexOpt: unable to optimize instance field ref "
646              "0x%04x at 0x%02x in %s.%s\n",
647             fieldIdx, (int) (insns - method->insns), clazz->descriptor,
648             method->name);
649         return false;
650     }
651
652     if (instField->byteOffset >= 65536) {
653         LOGI("DexOpt: field offset exceeds 64K (%d)\n", instField->byteOffset);
654         return false;
655     }
656
657     if (volatileOpc != OP_NOP && dvmIsVolatileField(&instField->field)) {
658         updateCode(method, insns, (insns[0] & 0xff00) | (u2) volatileOpc);
659         LOGV("DexOpt: rewrote ifield access %s.%s --> volatile\n",
660             instField->field.clazz->descriptor, instField->field.name);
661     } else if (quickOpc != OP_NOP) {
662         updateCode(method, insns, (insns[0] & 0xff00) | (u2) quickOpc);
```

```
663         updateCode(method, insns+1, (u2) instField->byteOffset);
664         LOGV("DexOpt: rewrote ifield access %s.%s --> %d\n",
665             instField->field.clazz->descriptor, instField->field.name,
666             instField->byteOffset);
667     } else {
668         LOGV("DexOpt: no rewrite of ifield access %s.%s\n",
669             instField->field.clazz->descriptor, instField->field.name);
670     }
671
672     return true;
673 }
```

252. Android `Optimize.c` line 643 resolves the field index in the constant pool to (among other things) a byte offset of the field in the object. Android `Optimize.c` lines 662-663 changes the opcode to the desired quick opcode, and the argument to the quick bytecode to the byte offset of the field.

253. dvmOptResolveInstField can resolve a field. It stores data about the resolved field (including the byte offset of the field) in a table of resolved fields, so the next time it is asked to resolve this field, it can look up the resolved information in the table. (Along the way, it calls dvmOptResolveClass which does the same form of determining, storing, and replacing for classes.)

```
http://android.git.kernel.org/?p=platform/dalvik.git;a=blob;f=vm/analysis/Optimize.c#l1472
472 /*
473  * Alternate version of dvmResolveInstField().
474  *
475  * On failure, returns NULL, and sets *pFailure if pFailure is not NULL.
476  */
477 InstField* dvmOptResolveInstField(ClassObject* referrer, u4 ifieldIdx,
478     VerifyError* pFailure)
479 {
480     DvmDex* pDvmDex = referrer->pDvmDex;
481     InstField* resField;
482
483     resField = (InstField*) dvmDexGetResolvedField(pDvmDex, ifieldIdx);
484     if (resField == NULL) {
485         const DexFieldId* pFieldId;
486         ClassObject* resClass;
487
488         pFieldId = dexGetFieldId(pDvmDex->pDexFile, ifieldIdx);
489
490         /*
491          * Find the field's class.
492          */
493         resClass = dvmOptResolveClass(referrer, pFieldId->classIdx, pFailure);
494         if (resClass == NULL) {
495             //dvmClearOptException(dvmThreadSelf());
496             assert(!dvmCheckException(dvmThreadSelf()));
497             if (pFailure != NULL) { assert(!VERIFY_OK(*pFailure)); }
498             return NULL;
499         }
500
501         resField = (InstField*)dvmFindFieldHier(resClass,
502             dexStringById(pDvmDex->pDexFile, pFieldId->nameIdx),
503             dexStringByTypeIdx(pDvmDex->pDexFile, pFieldId->typeIdx));
504         if (resField == NULL) {
```

```
505             LOGD("DexOpt: couldn't find field %s.%s\n",
506                 resClass->descriptor,
507                 dexStringById(pDvmDex->pDexFile, pFieldId->nameIdx));
508             if (pFailure != NULL)
509                 *pFailure = VERIFY_ERROR_NO_FIELD;
510             return NULL;
511         }
512         if (dvmIsStaticField(&resField->field)) {
513             LOGD("DexOpt: wanted instance, got static for field %s.%s\n",
514                 resClass->descriptor,
515                 dexStringById(pDvmDex->pDexFile, pFieldId->nameIdx));
516             if (pFailure != NULL)
517                 *pFailure = VERIFY_ERROR_CLASS_CHANGE;
518             return NULL;
519         }
520
521         /*
522          * Add it to the resolved table so we're faster on the next lookup.
523          */
524         dvmDexSetResolvedField(pDvmDex, ifieldIdx, (Field*) resField);
525     }
526
527     /* access allowed? */
528     tweakLoader(referrer, resField->field.clazz);
529     bool allowed = dvmCheckFieldAccess(referrer, (Field*)resField);
530     untweakLoader(referrer, resField->field.clazz);
531     if (!allowed) {
532         LOGI("DexOpt: access denied from %s to field %s.%s\n",
533             referrer->descriptor, resField->field.clazz->descriptor,
534             resField->field.name);
535         if (pFailure != NULL)
536             *pFailure = VERIFY_ERROR_ACCESS_FIELD;
537         return NULL;
538     }
539
540     return resField;
541 }
```

254. In the code above, the first time the field has to be resolved, the test at Android `Optimize.c` line 484 fails and the field has to be resolved. Android `Optimize.c` line 493 resolves the class, possibly by finding it in the table of resolved classes, and then finds the field at Android `Optimize.c` line 501. Android `Optimize.c` line 524 saves the result of the resolution so the next time resolving this field will be faster.

255. Android's dvmDexSetResolvedField and dvmDexGetResolvedField use a table indexed by the field index in the constant pool to write and read the resolved field information.

```
http://android.git.kernel.org/?p=platform/dalvik.git;a=blob;f=vm/DvmDex.h#l153
153 INLINE void dvmDexSetResolvedField(DvmDex* pDvmDex, u4 fieldIdx,
154     struct Field* field)
155 {
156     assert(fieldIdx < pDvmDex->pHeader->fieldIdsSize);
157     pDvmDex->pResFields[fieldIdx] = field;
```

```
http://android.git.kernel.org/?p=platform/dalvik.git;a=blob;f=vm/DvmDex.h#l124
124 INLINE struct Field* dvmDexGetResolvedField(const DvmDex* pDvmDex,
125     u4 fieldIdx)
126 {
```

```
127     assert(fieldIdx < pDvmDex->pHeader->fieldIdsSize);
128     return pDvmDex->pResFields[fieldIdx];
129 }
```

256. The calls to updateCode from Android `Optimize.c` lines 662 and 663 are fairly straightforward. One wrinkle is whether the page is read-write or read-only. If the page is read-only, the Android code changes the page to read-write for the write itself.

```
http://android.git.kernel.org/?p=platform/dalvik.git;a=blob;f=vm/analysis/Optimize.c#l337
337 /*
338  * Update a 16-bit code unit in "meth".
339  */
340 static inline void updateCode(const Method* meth, u2* ptr, u2 newVal)
341 {
342     if (gDvm.optimizing) {
343         /* dexopt time, alter the output directly */
344         *ptr = newVal;
345     } else {
346         /* runtime, toggle the page read/write status */
347         dvmDexChangeDex2(meth->clazz->pDvmDex, ptr, newVal);
348     }
349 }
```

```
http://android.git.kernel.org/?p=platform/dalvik.git;a=blob;f=vm/DvmDex.c#l1261
261 /*
262  * Change the 2-byte value at the specified address to a new value.  If the
263  * location already has the new value, do nothing.
264  *
265  * Otherwise works like dvmDexChangeDex1.
266  */
267 bool dvmDexChangeDex2(DvmDex* pDvmDex, u2* addr, u2 newVal)
268 {
269     if (*addr == newVal) {
270         LOGV("+++ value at %p is already 0x%04x\n", addr, newVal);
271         return true;
272     }
273
274     /*
275      * We're not holding this for long, so we don't bother with switching
276      * to VMWAIT.
277      */
278     dvmLockMutex(&pDvmDex->modLock);
279
280     LOGV("+++ change 2byte at %p from 0x%04x to 0x%04x\n", addr, *addr, newVal);
281     if (sysChangeMapAccess(addr, 2, true, &pDvmDex->memMap) != 0) {
282         LOGD("NOTE: DEX page access change (->RW) failed\n");
283         /* expected on files mounted from FAT; keep going (may crash) */
284     }
285
286     *addr = newVal;
287
288     if (sysChangeMapAccess(addr, 2, false, &pDvmDex->memMap) != 0) {
289         LOGD("NOTE: DEX page access change (->RO) failed\n");
290         /* expected on files mounted from FAT; keep going */
291     }
292
293     dvmUnlockMutex(&pDvmDex->modLock);
294
295     return true;
296 }
```

257. The Android virtual machine can execute non-quickened field read and write instructions. (For example, if dexopt did not rewrite these instructions.) For example, the x86 assembler interpreter for the OP_IGET instruction is:

```
http://android.git.kernel.org/?p=platform/dalvik.git;a=blob;f=vm/mterp/out/InterpAsm-
x86.S#l1968
1968 /* ------------------------------ */
1969     .balign 64
1970 .L_OP_IGET: /* 0x52 */
1971 /* File: x86/OP_IGET.S */
1972     /*
1973      * General 32-bit instance field get.
1974      *
1975      * for: iget, iget-object, iget-boolean, iget-byte, iget-char, iget-short
1976      */
1977     /* op vA, vB, field@CCCC */
1978     GET_GLUE(%ecx)
1979     SPILL(rIBASE)                                  # need another reg
1980     movzwl  2(rPC),rIBASE                          # rIBASE<- 0000CCCC
1981     movl    offGlue_methodClassDex(%ecx),%eax      # eax<- DvmDex
1982     movzbl  rINST_HI,%ecx                          # ecx<- BA
1983     sarl    $4,%ecx                               # ecx<- B
1984     movl    offDvmDex_pResFields(%eax),%eax        # eax<- pDvmDex->pResFields
1985     movzbl  rINST_HI,rINST_FULL                    # rINST_FULL<- BA
1986     andb    $0xf,rINST_LO                         # rINST_FULL<- A
1987     GET_VREG(%ecx,%ecx)                            # ecx<- fp[B], the object ptr
1988     movl    (%eax,rIBASE,4),%eax                   # resolved entry
1989     testl   %eax,%eax                              # is resolved entry null?
1990     jne     .LOP_IGET_finish                      # no, already resolved
1991     movl    rIBASE,OUT_ARG1(%esp)                  # needed by dvmResolveInstField
1992     GET_GLUE(rIBASE)
1993     jmp     .LOP_IGET_resolve
```

258. Android `InterpAsm-x86.S` line 1984 tests the slot in the table built by dvmOptResolveInstField; if it's not set, Android branches to .LOP_IGET_resolve to resolve the field index:

```
6980 /* continuation for OP_IGET */
6981
6982
6983 .LOP_IGET_resolve:
6984     EXPORT_PC()
6985     SPILL(rPC)
6986     movl    offGlue_method(rIBASE),rPC             # rPC<- current method
6987     UNSPILL(rIBASE)
6988     movl    offMethod_clazz(rPC),rPC               # rPC<- method->clazz
6989     SPILL_TMP(%ecx)                                # save object pointer across call
6990     movl    rPC,OUT_ARG0(%esp)                     # pass in method->clazz
6991     call    dvmResolveInstField                    #  ... to dvmResolveInstField
6992     UNSPILL_TMP(%ecx)
6993     UNSPILL(rPC)
6994     testl   %eax,%eax                              #  ... which returns InstrField ptr
6995     jne     .LOP_IGET_finish
6996     jmp     common_exceptionThrown
```

259. dvmResolveInstField stores the result of symbolic reference resolution as detailed later after the explanation of how the bytecode interpreter executes quickened bytecodes. As

shown in line 6995, the interpreter jumps to .LOP_IGET_finish after the symbolic reference is resolved to a numeric reference.

```
6998 .LOP_IGET_finish:
6999     /*
7000      * Currently:
7001      *   eax holds resolved field
7002      *   ecx holds object
7003      *   rIBASE is scratch, but needs to be unspilled
7004      *   rINST_FULL holds A
7005      */
7006     movl    offInstField_byteOffset(%eax),%eax   # eax<- byte offset of field
7007     UNSPILL(rIBASE)
7008     testl   %ecx,%ecx                            # object null?
7009     je      common_errNullObject                 # object was null
7010     movl    (%ecx,%eax,1),%ecx                  # ecx<- obj.field (8/16/32 bits)
7011     movl    rINST_FULL,%eax                      # eax<- A
7012     FETCH_INST_WORD(2)
7013     SET_VREG(%ecx,%eax)
7014     ADVANCE_PC(2)
7015     GOTO_NEXT
```

260. Android `InterpAsm-x86.S` line 7006 extracts the field offset from the fieldInst structure (in contrast to the value that would have been written as the operand to the quick instruction at vm/analysis/Optimize.c#l663), and Android `InterpAsm-x86.S` line 7010 fetches the field of the object. Note that there are (at least) two paths to .LOP_IGET_finish: (1) if the field referenced as the operand to the non-quick instruction has been resolved by dexopt (1990), and (2) if the resolution has to be done at during interpretation.

261. If the OP_IGET had been rewritten by dexopt, then the Android execution sequence ends up in the OP_IGET_QUICK branch of the interpreter:

```
http://android.git.kernel.org/?p=platform/dalvik.git;a=blob;f=vm/mterp/out/InterpAsm-
x86.S#l5964
5964 /* ------------------------------ */
5965     .balign 64
5966 .L_OP_IGET_QUICK: /* 0xf2 */
5967 /* File: x86/OP_IGET_QUICK.S */
5968     /* For: iget-quick, iget-object-quick */
5969     /* op vA, vB, offset@CCCC */
5970     movzbl    rINST_HI,%ecx              # ecx<- BA
5971     sarl      $4,%ecx                   # ecx<- B
5972     GET_VREG(%ecx,%ecx)                  # vB (object we're operating on)
5973     movzwl    2(rPC),%eax                # eax<- field byte offset
5974     cmpl      $0,%ecx                   # is object null?
5975     je        common_errNullObject
5976     movl      (%ecx,%eax,1),%eax
5977     movzbl    rINST_HI,%ecx
5978     FETCH_INST_WORD(2)
5979     ADVANCE_PC(2)
5980     andb      $0xf,%cl                  # rINST_FULL<- A
5981     SET_VREG  (%eax,%ecx)                # fp[A]<- result
5982     GOTO_NEXT
```

262. Android `InterpAsm-x86.S` line 5973 gets the offset of the field from the operand to the OP_IGET_QUICK instruction, and Android `InterpAsm-x86.S` line 5973 uses it to read the field from the object. *E.g.*, "movl (%ecx,%eax,1),%eax is the actual field fetch."

263. Therefore, Android meets limitation [11-b] of claim 11 because symbolic references are resolved and stored in dexopt and used to access data in the bytecode interpreter.

264. There is a second way in which Android meets **limitation [11-b] of claim 11**. The Dalvik bytecode interpreter can also resolve symbolic references and store the resulting numeric references (again for later use by the bytecode interpreter). This functionality applies to bytecodes that have not been rewritten by dexopt. One case where dexopt appears inapplicable is that code in the class libraries that are preloaded by *zygote* (see the analysis of the '720 patent) may contain methods that receive objects as part of a method call. If an object passed to the class library comes from a Java class that is defined in the application, then the preloading process does not generally know the layout of the objects of this class. Therefore, it is logical for symbolic reference resolution to fall to the bytecode interpreter. A second example illustrating the benefit of symbolic resolution by the bytecode interpreter arises with applications that are packaged as more than one .dex file, with portions of the application loaded using runtime class loading. Further explanation of this scenario appears in a blog posting called Custom Class Loading in Dalvik (*see* http://m.mobilitybeat.com/site/android-developers-blog/2/; http://android-developers.blogspot.com/2011/07/custom-class-loading-in-dalvik.html).

265. Android's Dalvik virtual machine looks at instructions and resolves symbolic references: Resolve.c functions determine corresponding numerical references, stores those numerical references, and obtain data corresponding to the resolved numerical references. (*See, e.g.*, 5/4/2011 McFadden Dep. 150:7-151:2.) The functions include:

dvmResolveClass
dvmResolveMethod
dvmResolveInterfaceMethod
dvmResolveInstField
dvmResolveStaticField
dvmResolveString

266. As comments from Android developers explain, the functions available in Resolve.c literally serve to resolve classes, methods, fields, and strings:

```
\dalvik\vm\oo\Resolve.c

/*
 * Resolve classes, methods, fields, and strings.
 *
 * According to the VM spec (v2 5.5), classes may be initialized by use
 * of the "new", "getstatic", "putstatic", or "invokestatic" instructions.
 * If we are resolving a static method or static field, we make the
 * initialization check here.
 *
 * (NOTE: the verifier has its own resolve functions, which can be invoked
 * if a class isn't pre-verified.  Those functions must not update the
 * "resolved stuff" tables for static fields and methods, because they do
 * not perform initialization.)
 */
```

267. My best understanding is that "VM spec (v2 5.5)" refers to Sun Microsystems's (now Oracle's) Java virtual machine specification. Of note, Sun Microsystems's Java Virtual Machine Specification, Edition 2, Section 5.5, details:

http://java.sun.com/docs/books/jvms/second_edition/html/ConstantPool.doc.html#77976

5.5 Initialization

Initialization of a class or interface consists of invoking its static initializers (§2.11) and the initializers for static fields (§2.9.2) declared in the class. This process is described in more detail in §2.17.4 and §2.17.5.

A class or interface may be initialized only as a result of:

- The execution of any one of the Java virtual machine instructions *new*, *getstatic*, *putstatic*, or *invokestatic* that references the class or interface. Each of these instructions corresponds to one of the conditions in §2.17.4. All of the previously listed instructions reference a class directly or indirectly through either a field reference or a method reference. Upon execution of a *new* instruction, the referenced class or interface is initialized if it has not been initialized already. Upon execution of a *getstatic, putstatic,* or *invokestatic* instruction, the class or interface that declared the resolved field or method is initialized if it has not been initialized already.
- Invocation of certain reflective methods in the class library (§3.12), for example, in class `Class` or in package `java.lang.reflect`.
- The initialization of one of its subclasses.

- Its designation as the initial class at Java virtual machine start-up (§5.2).

Prior to initialization a class or interface must be linked, that is, verified, prepared, and optionally resolved.

268. This means that the virtual machine cannot initialize a class until it has been resolved. The Java virtual machine abides by this. The evidence indicates that the Dalvik virtual machine is designed to do the same.

269. The function "dvmResolveClass" determines or resolves symbolic references (class names) to numerical references (class indexes, "classIdx") and obtains data in accordance to the numerical reference (the class, "resClass"). (*See also* 5/4/2011 McFadden Dep. 144:3-145:9.) As the Android developers explain, "**We cache a copy of the lookup in the DexFile's "resolved class" table, so future references to "classIdx" are faster.**" This refers to the method performed by Android that infringes the '104 asserted claims.

\dalvik\vm\oo\Resolve.c

```
35 /*
36  * Find the class corresponding to "classIdx", which maps to a class name
37  * string.  It might be in the same DEX file as "referrer", in a different
38  * DEX file, generated by a class loader, or generated by the VM (e.g.
39  * array classes).
40  *
41  * Because the DexTypeId is associated with the referring class' DEX file,
42  * we may have to resolve the same class more than once if it's referred
43  * to from classes in multiple DEX files.  This is a necessary property for
44  * DEX files associated with different class loaders.
45  *
46  * We cache a copy of the lookup in the DexFile's "resolved class" table,
47  * so future references to "classIdx" are faster.
48  *
49  * Note that "referrer" may be in the process of being linked.
50  *
51  * Traditional VMs might do access checks here, but in Dalvik the class
52  * "constant pool" is shared between all classes in the DEX file.  We rely
53  * on the verifier to do the checks for us.
54  *
55  * Does not initialize the class.
56  *
57  * "fromUnverifiedConstant" should only be set if this call is the direct
58  * result of executing a "const-class" or "instance-of" instruction, which
59  * use class constants not resolved by the bytecode verifier.
60  *
```

```
 61  * Returns NULL with an exception raised on failure.
 62  */
 63 ClassObject* dvmResolveClass(const ClassObject* referrer, u4 classIdx,
 64     bool fromUnverifiedConstant)
 65 {
 66     DvmDex* pDvmDex = referrer->pDvmDex;
 67     ClassObject* resClass;
 68     const char* className;
 69
 70     /*
 71      * Check the table first -- this gets called from the other "resolve"
 72      * methods.
 73      */
 74     resClass = dvmDexGetResolvedClass(pDvmDex, classIdx);
 75     if (resClass != NULL)
 76         return resClass;
…
 81     /*
 82      * Class hasn't been loaded yet, or is in the process of being loaded
 83      * and initialized now.  Try to get a copy.  If we find one, put the
 84      * pointer in the DexTypeId.  There isn't a race condition here --
 85      * 32-bit writes are guaranteed atomic on all target platforms.  Worst
 86      * case we have two threads storing the same value.
 87      *
 88      * If this is an array class, we'll generate it here.
 89      */
 90     className = dexStringByTypeIdx(pDvmDex->pDexFile, classIdx);
 91     if (className[0] != '\0' && className[1] == '\0') {
 92         /* primitive type */
 93         resClass = dvmFindPrimitiveClass(className[0]);
 94     } else {
 95         resClass = dvmFindClassNoInit(className, referrer->classLoader);
 96     }
 97
 98     if (resClass != NULL) {
 99         /*
100          * If the referrer was pre-verified, the resolved class must come
101          * from the same DEX or from a bootstrap class.  The pre-verifier
102          * makes assumptions that could be invalidated by a wacky class
103          * loader.  (See the notes at the top of oo/Class.c.)
…
117          */
…
147         /*
148          * Add what we found to the list so we can skip the class search
149          * next time through.
150          *
151          * TODO: should we be doing this when fromUnverifiedConstant==true?
152          * (see comments at top of oo/Class.c)
153          */
154         dvmDexSetResolvedClass(pDvmDex, classIdx, resClass);
155     } else {
…
```

```
160     }
161
162     return resClass;
163 }
164
```

270. As shown in the excerpted code, the function `dvmResolveClass` invokes `dvmDexGetResolvedClass` at line 74 in Resolve.c. The function `dvmDexGetResolvedClass` is defined in DvmDex.h and serves to return the requested ClassObject if it has been "resolved":

```
\dalvik\vm\DvmDex.h

103 /*
104  * Return the requested item if it has been resolved, or NULL if it hasn't.
105  */
…
112 INLINE struct ClassObject* dvmDexGetResolvedClass(const DvmDex* pDvmDex,
113     u4 classIdx)
114 {
115     assert(classIdx < pDvmDex->pHeader->typeIdsSize);
116     return pDvmDex->pResClasses[classIdx];
117 }
…
131 /*
132  * Update the resolved item table.  Resolution always produces the same
133  * result, so we're not worried about atomicity here.
134  */
…
141 INLINE void dvmDexSetResolvedClass(DvmDex* pDvmDex, u4 classIdx,
142     struct ClassObject* clazz)
143 {
144     assert(classIdx < pDvmDex->pHeader->typeIdsSize);
145     pDvmDex->pResClasses[classIdx] = clazz;
146 }
…
```

271. As shown in the Resolve.c excerpted code, the function `dvmResolveClass` invokes `dvmDexSetResolvedClass` at line 154 in Resolve.c. (*See also* 5/4/2011 McFadden Dep. 144:20-145:9.) The DvmDex.h file also has `dvmDexSetResolvedClass` which serves to **store** the resolved symbolic reference after the class name gets resolved for the first time or needs to get resolved again for some reason. (*See also id.* at 146:12-147:5)

272. The function "dvmResolveMethod" determines or resolves symbolic references (method names) to numerical references ("methodRef" or "methodIdx") and obtains data in accordance to the numerical reference (the method, "resMethod"):

\dalvik\vm\oo\Resolve.c

```
166 /*
167  * Find the method corresponding to "methodRef".
168  *
169  * We use "referrer" to find the DexFile with the constant pool that
170  * "methodRef" is an index into.  We also use its class loader.  The method
171  * being resolved may very well be in a different DEX file.
172  *
173  * If this is a static method, we ensure that the method's class is
174  * initialized.
175  */
176 Method* dvmResolveMethod(const ClassObject* referrer, u4 methodIdx,
177     MethodType methodType)
178 {
179     DvmDex* pDvmDex = referrer->pDvmDex;
180     ClassObject* resClass;
181     const DexMethodId* pMethodId;
182     Method* resMethod;
…
188     pMethodId = dexGetMethodId(pDvmDex->pDexFile, methodIdx);
189
190     resClass = dvmResolveClass(referrer, pMethodId->classIdx, false);
…
196     if (dvmIsInterfaceClass(resClass)) {
197         /* method is part of an interface */
198         dvmThrowExceptionWithClassMessage(
199             "Ljava/lang/IncompatibleClassChangeError;",
200             resClass->descriptor);
201         return NULL;
202     }
203
204     const char* name = dexStringById(pDvmDex->pDexFile, pMethodId->nameIdx);
205     DexProto proto;
206     dexProtoSetFromMethodId(&proto, pDvmDex->pDexFile, pMethodId);
207
208     /*
209      * We need to chase up the class hierarchy to find methods defined
210      * in super-classes.  (We only want to check the current class
211      * if we're looking for a constructor; since DIRECT calls are only
212      * for constructors and private methods, we don't want to walk up.)
213      */
214     if (methodType == METHOD_DIRECT) {
215         resMethod = dvmFindDirectMethod(resClass, name, &proto);
216     } else if (methodType == METHOD_STATIC) {
217         resMethod = dvmFindDirectMethodHier(resClass, name, &proto);
218     } else {
```

```
219         resMethod = dvmFindVirtualMethodHier(resClass, name, &proto);
220     }
…
229
230     /* see if this is a pure-abstract method */
231     if (dvmIsAbstractMethod(resMethod) && !dvmIsAbstractClass(resClass)) {
232         dvmThrowException("Ljava/lang/AbstractMethodError;", name);
233         return NULL;
234     }
235
236     /*
237      * If we're the first to resolve this class, we need to initialize
238      * it now.  Only necessary for METHOD_STATIC.
239      */
240     if (methodType == METHOD_STATIC) {
241         if (!dvmIsClassInitialized(resMethod->clazz) &&
242             !dvmInitClass(resMethod->clazz))
243         {
244             assert(dvmCheckException(dvmThreadSelf()));
245             return NULL;
246         } else {
247             assert(!dvmCheckException(dvmThreadSelf()));
248         }
249     } else {
250         /*
251          * Edge case: if the <clinit> for a class creates an instance
252          * of itself, we will call <init> on a class that is still being
253          * initialized by us.
254          */
255         assert(dvmIsClassInitialized(resMethod->clazz) ||
256                dvmIsClassInitializing(resMethod->clazz));
257     }
258
259     /*
260      * If the class has been initialized, add a pointer to our data structure
261      * so we don't have to jump through the hoops again.  If this is a
262      * static method and the defining class is still initializing (i.e. this
263      * thread is executing <clinit>), don't do the store, otherwise other
264      * threads could call the method without waiting for class init to finish.
265      */
266     if (methodType == METHOD_STATIC && !dvmIsClassInitialized(resMethod->clazz))
267     {
…
272     } else {
273         dvmDexSetResolvedMethod(pDvmDex, methodIdx, resMethod);
274     }
275
276     return resMethod;
277 }
```

273. As shown in the Resolve.c excerpted code, the function `dvmResolveMethod` invokes `dexGetMethodId` at line 188 in Resolve.c. The function `dexGetMethodId` is defined in DexFile.h and

serves to return the numerical reference (MethodID with specified index) (shown below), which in turn is used to resolve the class in which the method being resolved belongs as shown in line Resolve.c's 188-190 above:

```
\dalvik\libdex\DexFile.h

 627 /* return the MethodId with the specified index */
 628 DEX_INLINE const DexMethodId* dexGetMethodId(const DexFile* pDexFile, u4 idx) {
 629     assert(idx < pDexFile->pHeader->methodIdsSize);
 630     return &pDexFile->pMethodIds[idx];
 631 }
 632
```

274. Then, `dvmResolveMethod` invokes one of the following three functions in Resolve.c at lines 215-219 to resolve the method:

```
\dalvik\vm\oo\Resolve.c

 215         resMethod = dvmFindDirectMethod(resClass, name, &proto);
 216     } else if (methodType == METHOD_STATIC) {
 217         resMethod = dvmFindDirectMethodHier(resClass, name, &proto);
 218     } else {
 219         resMethod = dvmFindVirtualMethodHier(resClass, name, &proto);
 220     }
```

275. The three functions are shown below, and each serves to resolve a method name to the resolved method:

```
\dalvik\vm\oo\Object.c

 531 /*
 532  * Find a "virtual" method in a class.  If we don't find it, try the
 533  * superclass.
 534  *
 535  * Returns NULL if the method can't be found.  (Does not throw an exception.)
 536  */
 537 Method* dvmFindVirtualMethodHier(const ClassObject* clazz,
 538     const char* methodName, const DexProto* proto)
 539 {
 540     return findMethodInListByProto(clazz, METHOD_VIRTUAL, true, methodName,
 541             proto);
 542 }
 543
...
 570 /*
 571  * Find a "direct" method (static or "<*init>").
 572  *
```

```
 573  * Returns NULL if the method can't be found.  (Does not throw an exception.)
 574  */
 575 Method* dvmFindDirectMethod(const ClassObject* clazz, const char* methodName,
 576     const DexProto* proto)
 577 {
 578     return findMethodInListByProto(clazz, METHOD_DIRECT, false, methodName,
 579             proto);
 580 }
…
582 /*
 583  * Find a "direct" method in a class.  If we don't find it, try the
 584  * superclass.
 585  *
 586  * Returns NULL if the method can't be found.  (Does not throw an exception.)
 587  */
 588 Method* dvmFindDirectMethodHier(const ClassObject* clazz,
 589     const char* methodName, const DexProto* proto)
 590 {
 591     return findMethodInListByProto(clazz, METHOD_DIRECT, true, methodName,
 592             proto);
 593 }
 594
…
```

276. In turn, `findMethodInListByProto` serves to find and return the method being resolved as indicated by Android developer comments and the source code shown below:

```
\dalvik/vm/oo/Object.c

 417 /*
 418  * Look for a match in the given clazz. Returns the match if found
 419  * or NULL if not.
 420  *
 421  * "wantedType" should be METHOD_VIRTUAL or METHOD_DIRECT to indicate the
 422  * list to search through.  If the match can come from either list, use
 423  * MATCH_UNKNOWN to scan both.
 424  */
 425 static Method* findMethodInListByProto(const ClassObject* clazz,
 426     MethodType wantedType, bool isHier, const char* name, const DexProto* proto)
 427 {
 428     while (clazz != NULL) {
 429         int i;
 430
 431         /*
 432          * Check the virtual and/or direct method lists.
 433          */
 434         if (wantedType == METHOD_VIRTUAL || wantedType == METHOD_UNKNOWN) {
 435             for (i = 0; i < clazz->virtualMethodCount; i++) {
 436                 Method* method = &clazz->virtualMethods[i];
 437                 if (dvmCompareNameProtoAndMethod(name, proto, method) == 0) {
 438                     return method;
```

```
439                 }
440             }
441         }
442         if (wantedType == METHOD_DIRECT || wantedType == METHOD_UNKNOWN) {
443             for (i = 0; i < clazz->directMethodCount; i++) {
444                 Method* method = &clazz->directMethods[i];
445                 if (dvmCompareNameProtoAndMethod(name, proto, method) == 0) {
446                     return method;
447                 }
448             }
449         }
450
451         if (! isHier) {
452             break;
453         }
454
455         clazz = clazz->super;
456     }
457
458     return NULL;
459 }
```

277. As shown in the Resolve.c excerpted code, the function `dvmResolveMethod` invokes `dvmDexSetResolvedMethod` at line 273 in Resolve.c. The DvmDex.h file contains the source code for the `dvmDexSetResolvedMethod` function, which serves to **store** the resolved symbolic reference when the method name gets resolved for the first time:

```
\dalvik\vm\DvmDex.h

131 /*
132  * Update the resolved item table.  Resolution always produces the same
133  * result, so we're not worried about atomicity here.
134  */
…
147 INLINE void dvmDexSetResolvedMethod(DvmDex* pDvmDex, u4 methodIdx,
148     struct Method* method)
149 {
150     assert(methodIdx < pDvmDex->pHeader->methodIdsSize);
151     pDvmDex->pResMethods[methodIdx] = method;
152 }
```

278. Likewise, the function "`dvmResolveInterfaceMethod`" determines or resolves symbolic references (interface method names) to numerical references ("pMethodId") and obtains data in accordance to the numerical reference (the method, "resMethod"):

\dalvik\vm\oo\Resolve.c

```
279 /*
280  * Resolve an interface method reference.
281  *
282  * Returns NULL with an exception raised on failure.
283  */
284 Method* dvmResolveInterfaceMethod(const ClassObject* referrer, u4 methodIdx)
285 {
286     DvmDex* pDvmDex = referrer->pDvmDex;
287     ClassObject* resClass;
288     const DexMethodId* pMethodId;
289     Method* resMethod;
290     int i;
…
294     pMethodId = dexGetMethodId(pDvmDex->pDexFile, methodIdx);
295
296     resClass = dvmResolveClass(referrer, pMethodId->classIdx, false);
297     if (resClass == NULL) {
298         /* can't find the class that the method is a part of */
299         assert(dvmCheckException(dvmThreadSelf()));
300         return NULL;
301     }
…
309
310     /*
311      * This is the first time the method has been resolved.  Set it in our
312      * resolved-method structure.  It always resolves to the same thing,
313      * so looking it up and storing it doesn't create a race condition.
314      *
315      * If we scan into the interface's superclass -- which is always
316      * java/lang/Object -- we will catch things like:
317      *   interface I ...
318      *   I myobj = (something that implements I)
319      *   myobj.hashCode()
320      * However, the Method->methodIndex will be an offset into clazz->vtable,
321      * rather than an offset into clazz->iftable.  The invoke-interface
322      * code can test to see if the method returned is abstract or concrete,
323      * and use methodIndex accordingly.  I'm not doing this yet because
324      * (a) we waste time in an unusual case, and (b) we're probably going
325      * to fix it in the DEX optimizer.
326      *
327      * We do need to scan the superinterfaces, in case we're invoking a
328      * superinterface method on an interface reference.  The class in the
329      * DexTypeId is for the static type of the object, not the class in
330      * which the method is first defined.  We have the full, flattened
331      * list in "iftable".
332      */
333     const char* methodName =
334         dexStringById(pDvmDex->pDexFile, pMethodId->nameIdx);
335
336     DexProto proto;
337     dexProtoSetFromMethodId(&proto, pDvmDex->pDexFile, pMethodId);
```

```
…
 341     resMethod = dvmFindVirtualMethod(resClass, methodName, &proto);
…
 368     /*
 369      * The interface class *may* be initialized.  According to VM spec
 370      * v2 2.17.4, the interfaces a class refers to "need not" be initialized
 371      * when the class is initialized.
 372      *
 373      * It isn't necessary for an interface class to be initialized before
 374      * we resolve methods on that interface.
 375      *
 376      * We choose not to do the initialization now.
 377      */
…
 380     /*
 381      * Add a pointer to our data structure so we don't have to jump
 382      * through the hoops again.
 383      *
 384      * As noted above, no need to worry about whether the interface that
 385      * defines the method has been or is currently executing <clinit>.
 386      */
 387     dvmDexSetResolvedMethod(pDvmDex, methodIdx, resMethod);
 388
 389     return resMethod;
 390 }
 391
```

279. As shown in the Resolve.c excerpted code, the function `dvmResolveInterfaceMethod` invokes `dexGetMethodId` at line 294. The function `dexGetMethodId` is defined in DexFile.h and serves to return the numerical reference (MethodID with specified index) (shown below), which in turn is used to resolve the class in which the method being resolved belongs as shown in line Resolve.c's lines 294-296 above:

```
\dalvik\libdex\DexFile.h

 627 /* return the MethodId with the specified index */
 628 DEX_INLINE const DexMethodId* dexGetMethodId(const DexFile* pDexFile, u4 idx) {
 629     assert(idx < pDexFile->pHeader->methodIdsSize);
 630     return &pDexFile->pMethodIds[idx];
 631 }
```

280. Then, `dvmResolveInterfaceMethod` invokes `dvmFindVirtualMethod` at Resolve.c line 341 to resolve the method:

```
\dalvik\vm\oo\Object.c
```

```
504 /*
505  * Find a "virtual" method in a class.
506  *
507  * Does not chase into the superclass.
508  *
509  * Returns NULL if the method can't be found.  (Does not throw an exception.)
510  */
511 Method* dvmFindVirtualMethod(const ClassObject* clazz, const char* methodName,
512     const DexProto* proto)
513 {
514     return findMethodInListByProto(clazz, METHOD_VIRTUAL, false, methodName,
515             proto);
516 }
```

281. In turn, `findMethodInListByProto` is invoked and serves to find and return the method being resolved as indicated by Android developer comments and the source code shown below:

```
\dalvik\vm\oo\Object.c

417 /*
418  * Look for a match in the given clazz. Returns the match if found
419  * or NULL if not.
420  *
421  * "wantedType" should be METHOD_VIRTUAL or METHOD_DIRECT to indicate the
422  * list to search through.  If the match can come from either list, use
423  * MATCH_UNKNOWN to scan both.
424  */
425 static Method* findMethodInListByProto(const ClassObject* clazz,
426     MethodType wantedType, bool isHier, const char* name, const DexProto* proto)
427 {
428     while (clazz != NULL) {
429         int i;
430
431         /*
432          * Check the virtual and/or direct method lists.
433          */
434         if (wantedType == METHOD_VIRTUAL || wantedType == METHOD_UNKNOWN) {
435             for (i = 0; i < clazz->virtualMethodCount; i++) {
436                 Method* method = &clazz->virtualMethods[i];
437                 if (dvmCompareNameProtoAndMethod(name, proto, method) == 0) {
438                     return method;
439                 }
440             }
441         }
442         if (wantedType == METHOD_DIRECT || wantedType == METHOD_UNKNOWN) {
443             for (i = 0; i < clazz->directMethodCount; i++) {
444                 Method* method = &clazz->directMethods[i];
445                 if (dvmCompareNameProtoAndMethod(name, proto, method) == 0) {
446                     return method;
```

```
447                 }
448             }
449         }
450
451         if (! isHier) {
452             break;
453         }
454
455         clazz = clazz->super;
456     }
457
458     return NULL;
459 }
```

282. As shown in the Resolve.c excerpted code, the function `dvmResolveInterfaceMethod` invokes `dvmDexSetResolvedMethod` at line 387. The DvmDex.h file contains the source code for `dvmDexSetResolvedMethod` which serves to **store** the resolved symbolic reference, and which gets called by Resolve.c's `dvmResolveInterfaceMethod` (as shown at line 387) when the method name gets resolved for the first time:

```
\dalvik\vm\DvmDex.h

131 /*
132  * Update the resolved item table.  Resolution always produces the same
133  * result, so we're not worried about atomicity here.
134  */
…
147 INLINE void dvmDexSetResolvedMethod(DvmDex* pDvmDex, u4 methodIdx,
148     struct Method* method)
149 {
150     assert(methodIdx < pDvmDex->pHeader->methodIdsSize);
151     pDvmDex->pResMethods[methodIdx] = method;
152 }
```

283. Next, the function "`dvmResolveInstField`" determines or resolves symbolic references (instance fields) to numerical references ("pFieldId") and obtains data in accordance to the numerical reference (the field, "resField"):

```
\dalvik\vm\oo\Resolve.c

392 /*
393  * Resolve an instance field reference.
394  *
395  * Returns NULL and throws an exception on error (no such field, illegal
396  * access).
397  */
```

```
398 InstField* dvmResolveInstField(const ClassObject* referrer, u4 ifieldIdx)
399 {
400     DvmDex* pDvmDex = referrer->pDvmDex;
401     ClassObject* resClass;
402     const DexFieldId* pFieldId;
403     InstField* resField;
…
408     pFieldId = dexGetFieldId(pDvmDex->pDexFile, ifieldIdx);
409
410     /*
411      * Find the field's class.
412      */
413     resClass = dvmResolveClass(referrer, pFieldId->classIdx, false);
414     if (resClass == NULL) {
415         assert(dvmCheckException(dvmThreadSelf()));
416         return NULL;
417     }
418
419     resField = dvmFindInstanceFieldHier(resClass,
420         dexStringById(pDvmDex->pDexFile, pFieldId->nameIdx),
421         dexStringByTypeIdx(pDvmDex->pDexFile, pFieldId->typeIdx));
422     if (resField == NULL) {
423         dvmThrowException("Ljava/lang/NoSuchFieldError;",
424             dexStringById(pDvmDex->pDexFile, pFieldId->nameIdx));
425         return NULL;
426     }
…
436     /*
437      * The class is initialized (or initializing), the field has been
438      * found.  Add a pointer to our data structure so we don't have to
439      * jump through the hoops again.
440      *
441      * Anything that uses the resolved table entry must have an instance
442      * of the class, so any class init activity has already happened (or
443      * been deliberately bypassed when <clinit> created an instance).
444      * So it's always okay to update the table.
445      */
446     dvmDexSetResolvedField(pDvmDex, ifieldIdx, (Field*)resField);
447     LOGVV("    field %u is %s.%s\n",
448         ifieldIdx, resField->field.clazz->descriptor, resField->field.name);
449
450     return resField;
451 }
```

284. As shown in the excerpted code, the function `dvmResolveInstField` invokes `dexGetFieldId` at line 408 in Resolve.c. The function `dexGetFieldId` is defined in DexFile.h and serves to return the numerical reference (FieldId with the specified index) (shown below), which

in turn is used to resolve the class in which the instance field being resolved belongs as shown in line Resolve.c's 408-413 above:

```
\dalvik\libdex\DexFile.h

 633 /* return the FieldId with the specified index */
 634
DEX_INLINE const DexFieldId* dexGetFieldId(const DexFile* pDexFile, u4 idx) {
 635     assert(idx < pDexFile->pHeader->fieldIdsSize);
 636     return &pDexFile->pFieldIds[idx];
 637 }
```

285. Then, Resolve.c's `dvmResolveInstField` invokes `dvmFindInstanceFieldHier` at line 419 to resolve the instance field:

```
\dalvik\vm\oo\Object.c

  52
  53 /*
  54  * Find a matching field, in this class or a superclass.
  55  *
  56  * Searching through interfaces isn't necessary, because interface fields
  57  * are inherently public/static/final.
  58  *
  59  * Returns NULL if the field can't be found.  (Does not throw an exception.)
  60  */
  61 InstField* dvmFindInstanceFieldHier(const ClassObject* clazz,
  62     const char* fieldName, const char* signature)
  63 {
  64     InstField* pField;
  65
  66     /*
  67      * Search for a match in the current class.
  68      */
  69     pField = dvmFindInstanceField(clazz, fieldName, signature);
  70     if (pField != NULL)
  71         return pField;
  72
  73     if (clazz->super != NULL)
  74         return dvmFindInstanceFieldHier(clazz->super, fieldName, signature);
  75     else
  76         return NULL;
  77 }
```

286. In turn, `dvmFindInstanceField` gets invoked to perform the resolution:

```
\dalvik\vm\oo\Object.c

  22 /*
  23  * Find a matching field, in the current class only.
  24  *
```

```
 * Returns NULL if the field can't be found.  (Does not throw an exception.)
 */
InstField* dvmFindInstanceField(const ClassObject* clazz,
    const char* fieldName, const char* signature)
{
    InstField* pField;
    int i;

    assert(clazz != NULL);

    /*
     * Find a field with a matching name and signature.  The Java programming
     * language does not allow you to have two fields with the same name
     * and different types, but the Java VM spec does allow it, so we can't
     * bail out early when the name matches.
     */
    pField = clazz->ifields;
    for (i = 0; i < clazz->ifieldCount; i++, pField++) {
        if (strcmp(fieldName, pField->field.name) == 0 &&
            strcmp(signature, pField->field.signature) == 0)
        {
            return pField;
        }
    }

    return NULL;
}
```

287. As shown in the Resolve.c excerpted code, the function `dvmResolveInstField` invokes `dvmDexSetResolvedField` at line 446. The DvmDex.h file contains `dvmDexSetResolvedField` which serves to **store** the resolved symbolic reference, and which gets called by Resolve.c's `dvmResolveInstField` (as shown at line 446) when the instance field gets resolved for the first time:

```
\dalvik\vm\DvmDex.h

/*
 * Update the resolved item table.  Resolution always produces the same
 * result, so we're not worried about atomicity here.
 */
…
INLINE void dvmDexSetResolvedField(DvmDex* pDvmDex, u4 fieldIdx,
    struct Field* field)
{
    assert(fieldIdx < pDvmDex->pHeader->fieldIdsSize);
    pDvmDex->pResFields[fieldIdx] = field;
}
```

288. Likewise, the function "dvmResolveStaticField" determines or resolves symbolic references (static field) to numerical references ("pFieldId") and obtains data in accordance to the numerical reference (the method, "resField"):

\dalvik\vm\oo\Resolve.c

```
453 /*
454  * Resolve a static field reference.  The DexFile format doesn't distinguish
455  * between static and instance field references, so the "resolved" pointer
456  * in the Dex struct will have the wrong type.  We trivially cast it here.
457  *
458  * Causes the field's class to be initialized.
459  */
460 StaticField* dvmResolveStaticField(const ClassObject* referrer, u4 sfieldIdx)
461 {
462     DvmDex* pDvmDex = referrer->pDvmDex;
463     ClassObject* resClass;
464     const DexFieldId* pFieldId;
465     StaticField* resField;
466
467     pFieldId = dexGetFieldId(pDvmDex->pDexFile, sfieldIdx);
468
469     /*
470      * Find the field's class.
471      */
472     resClass = dvmResolveClass(referrer, pFieldId->classIdx, false);
473     if (resClass == NULL) {
474         assert(dvmCheckException(dvmThreadSelf()));
475         return NULL;
476     }
477
478     resField = dvmFindStaticFieldHier(resClass,
479                 dexStringById(pDvmDex->pDexFile, pFieldId->nameIdx),
480                 dexStringByTypeIdx(pDvmDex->pDexFile, pFieldId->typeIdx));
481     if (resField == NULL) {
482         dvmThrowException("Ljava/lang/NoSuchFieldError;",
483             dexStringById(pDvmDex->pDexFile, pFieldId->nameIdx));
484         return NULL;
485     }
…
499     /*
500      * If the class has been initialized, add a pointer to our data structure
501      * so we don't have to jump through the hoops again.  If it's still
502      * initializing (i.e. this thread is executing <clinit>), don't do
503      * the store, otherwise other threads could use the field without waiting
504      * for class init to finish.
505      */
506     if (dvmIsClassInitialized(resField->field.clazz)) {
507         dvmDexSetResolvedField(pDvmDex, sfieldIdx, (Field*) resField);
…
515     return resField;
```

```
516 }
```

289. As shown in the Resolve.c excerpted code, dvmResolveStaticField invokes dexGetFieldId at line 467. The function dexGetFieldId is defined in DexFile.h and serves to return the numerical reference (FieldId with the specified index) (shown below), which in turn is used to resolve the class in which the instance field being resolved belongs as shown in line Resolve.c's 467-472 above:

```
\dalvik\libdex\DexFile.h

633 /* return the FieldId with the specified index */
634 DEX_INLINE const DexFieldId* dexGetFieldId(const DexFile* pDexFile, u4 idx) {
635     assert(idx < pDexFile->pHeader->fieldIdsSize);
636     return &pDexFile->pFieldIds[idx];
637 }
```

290. Then, dvmResolveStaticField invokes dvmFindStaticFieldHier in Resolve.c at line 478 to resolve the field:

```
\dalvik\vm\oo\Object.c

110 /*
111  * Find a matching field, in this class or a superclass.
112  *
113  * Returns NULL if the field can't be found.  (Does not throw an exception.)
114  */
115 StaticField* dvmFindStaticFieldHier(const ClassObject* clazz,
116     const char* fieldName, const char* signature)
117 {
118     StaticField* pField;
119
120     /*
121      * Search for a match in the current class.
122      */
123     pField = dvmFindStaticField(clazz, fieldName, signature);
124     if (pField != NULL)
125         return pField;
126
127     /*
128      * See if it's in any of our interfaces.  We don't check interfaces
129      * inherited from the superclass yet.
130      *
131      * (Note the set may have been stripped down because of redundancy with
132      * the superclass; see notes in createIftable.)
133      */
134     int i = 0;
135     if (clazz->super != NULL) {
136         assert(clazz->iftableCount >= clazz->super->iftableCount);
```

```
137         i = clazz->super->iftableCount;
138     }
139     for ( ; i < clazz->iftableCount; i++) {
140         ClassObject* iface = clazz->iftable[i].clazz;
141         pField = dvmFindStaticField(iface, fieldName, signature);
142         if (pField != NULL)
143             return pField;
144     }
145
146     if (clazz->super != NULL)
147         return dvmFindStaticFieldHier(clazz->super, fieldName, signature);
148     else
149         return NULL;
150 }
```

291. In turn, dvmFindStaticField gets invoked to perform the resolution:

```
\dalvik\vm\oo\Object.c

 80 /*
 81  * Find a matching field, in this class or an interface.
 82  *
 83  * Returns NULL if the field can't be found.  (Does not throw an exception.)
 84  */
 85 StaticField* dvmFindStaticField(const ClassObject* clazz,
 86     const char* fieldName, const char* signature)
 87 {
 88     const StaticField* pField;
 89     int i;
 90
 91     assert(clazz != NULL);
 92
 93     /*
 94      * Find a field with a matching name and signature.  As with instance
 95      * fields, the VM allows you to have two fields with the same name so
 96      * long as they have different types.
 97      */
 98     pField = &clazz->sfields[0];
 99     for (i = 0; i < clazz->sfieldCount; i++, pField++) {
100         if (strcmp(fieldName, pField->field.name) == 0 &&
101             strcmp(signature, pField->field.signature) == 0)
102         {
103             return (StaticField*) pField;
104         }
105     }
106
107     return NULL;
108 }
```

292. As shown in the excerpted code, the function dvmResolveStaticField invokes dvmDexSetResolvedField at line 507 in Resolve.c. The DvmDex.h file contains the source code for

dvmDexSetResolvedField which serves to **store** the resolved symbolic reference, and which gets called by Resolve.c's dvmResolveStaticField (as shown at line 507) when the field gets resolved for the first time:

\dalvik\vm\DvmDex.h

```
131 /*
132  * Update the resolved item table.  Resolution always produces the same
133  * result, so we're not worried about atomicity here.
134  */
…
153 INLINE void dvmDexSetResolvedField(DvmDex* pDvmDex, u4 fieldIdx,
154     struct Field* field)
155 {
156     assert(fieldIdx < pDvmDex->pHeader->fieldIdsSize);
157     pDvmDex->pResFields[fieldIdx] = field;
158 }
```

293. As a final example, the function "dvmResolveString" determines or resolves symbolic references (string reference) to numerical references ("stringIdx") and obtains data in accordance to the numerical reference (strObj):

\dalvik\vm\oo\Resolve.c

```
519 /*
520  * Resolve a string reference.
521  *
522  * Finding the string is easy.  We need to return a reference to a
523  * java/lang/String object, not a bunch of characters, which means the
524  * first time we get here we need to create an interned string.
525  */
526 StringObject* dvmResolveString(const ClassObject* referrer, u4 stringIdx)
527 {
528     DvmDex* pDvmDex = referrer->pDvmDex;
529     StringObject* strObj;
530     StringObject* internStrObj;
531     const char* utf8;
532     u4 utf16Size;
…
540     utf8 = dexStringAndSizeById(pDvmDex->pDexFile, stringIdx, &utf16Size);
541     strObj = dvmCreateStringFromCstrAndLength(utf8, utf16Size);
…
548     /*
549      * Add it to the intern list.  The return value is the one in the
550      * intern list, which (due to race conditions) may or may not be
551      * the one we just created.  The intern list is synchronized, so
552      * there will be only one "live" version.
553      *
```

```
554      * By requesting an immortal interned string, we guarantee that
555      * the returned object will never be collected by the GC.
556      *
557      * A NULL return here indicates some sort of hashing failure.
558      */
559     internStrObj = dvmLookupImmortalInternedString(strObj);
560     dvmReleaseTrackedAlloc((Object*) strObj, NULL);
561     strObj = internStrObj;
…
567     /* save a reference so we can go straight to the object next time */
568     dvmDexSetResolvedString(pDvmDex, stringIdx, strObj);
569
570 bail:
571     return strObj;
572 }
```

294. The function `dvmResolveString` invokes `dexStringAndSizeById` at line 540 of Resolve.c, which returns the numerical reference (string_id index):

```
\dalvik\libdex\DexFile.c


283 /* Return the UTF-8 encoded string with the specified string_id index,
284  * also filling in the UTF-16 size (number of 16-bit code points).*/
285 const char* dexStringAndSizeById(const DexFile* pDexFile, u4 idx,
286         u4* utf16Size) {
287     const DexStringId* pStringId = dexGetStringId(pDexFile, idx);
288     const u1* ptr = pDexFile->baseAddr + pStringId->stringDataOff;
289
290     *utf16Size = readUnsignedLeb128(&ptr);
291     return (const char*) ptr;
292 }
```

295. The function `dvmResolveString` then invokes `dvmLookupImmortalInternedString` at line 559 of Resolve.c, which returns the data obtained from resolution:

```
\dalvik\vm\Intern.c

131 /*
132  * Find an entry in the interned string table.
133  *
134  * If the string doesn't already exist, the StringObject is added to
135  * the table.  Otherwise, the existing entry is returned.
136  */
137 StringObject* dvmLookupInternedString(StringObject* strObj)
138 {
139     return lookupInternedString(strObj, false);
140 }
141
```

```
142 /*
143  * Same as dvmLookupInternedString(), but guarantees that the
144  * returned string is a literal.
145  */
146 StringObject* dvmLookupImmortalInternedString(StringObject* strObj)
147 {
148     return lookupInternedString(strObj, true);
149 }
```

296. As shown in the excerpted code in Resolve.c, the function `dvmResolveString` invokes `dvmDexSetResolvedString` at line 568. The DvmDex.h file contains the source code for `dvmDexSetResolvedString` which serves to **store** the resolved symbolic reference, and which gets called by Resolve.c's `dvmResolveString` (as shown at line 568) when the String reference gets resolved for the first time:

```
\dalvik\vm\DvmDex.h

131 /*
132  * Update the resolved item table.  Resolution always produces the same
133  * result, so we're not worried about atomicity here.
134  */
135 INLINE void dvmDexSetResolvedString(DvmDex* pDvmDex, u4 stringIdx,
136     struct StringObject* str)
137 {
138     assert(stringIdx < pDvmDex->pHeader->stringIdsSize);
139     pDvmDex->pResStrings[stringIdx] = str;
140 }
```

297. The Android virtual machine invokes the functions illustrated above when executing instructions. The following documentation provides context for the code examples discussed below:

See also, e.g., Dalvik Virtual Machine, "Porting Dalvik," *available at* http://source.android.com/porting/dalvik.html:

> **Dalvik**
>
> The Dalvik virtual machine is intended to run on a variety of platforms. The baseline system is expected to be a variant of UNIX (Linux, BSD, Mac OS X) running the GNU C compiler. Little-endian CPUs have been exercised the most heavily, but big-endian systems are explicitly supported.
>
> There are two general categories of work: porting to a Linux system with a previously unseen CPU architecture, and porting to a different operating system. This document covers the former.

…

Interpreter

The Dalvik runtime includes two interpreters, labeled "portable" and "fast". The portable interpreter is largely contained within a single C function, and should compile on any system that supports gcc. (If you don't have gcc, you may need to disable the "threaded" execution model, which relies on gcc's "goto table" implementation; look for the THREADED_INTERP define.)

The fast interpreter uses hand-coded assembly fragments. If none are available for the current architecture, the build system will create an interpreter out of C "stubs". The resulting "all stubs" interpreter is quite a bit slower than the portable interpreter, making "fast" something of a misnomer.

The fast interpreter is enabled by default. On platforms without native support, you may want to switch to the portable interpreter. This can be controlled with the dalvik.vm.execution-mode system property. For example, if you:

adb shell "echo dalvik.vm.execution-mode = int:portable >> /data/local.prop"

and reboot, the Android app framework will start the VM with the portable interpreter enabled.

298. For example, the x86 assembler interpreter has the following code showing invocation of the Resolve.c functions discussed above:

http://android.git.kernel.org/?p=platform/dalvik.git;a=blob;f=vm/mterp/out/InterpAsm-x86.S#l1968

```
...
6306 /* This is the less common path, so we'll redo some work
6307    here rather than force spills on the common path */
6308 .LOP_CONST_STRING_resolve:
6309     GET_GLUE(%eax)
6310     movl     %ecx,rINST_FULL           # rINST_FULL<- AA
6311     EXPORT_PC()
6312     movl     offGlue_method(%eax),%eax # eax<- glue->method
6313     movzwl   2(rPC),%ecx               # ecx<- BBBB
6314     movl     offMethod_clazz(%eax),%eax
6315     SPILL(rPC)
6316     movl     %ecx,OUT_ARG1(%esp)
6317     movl     %eax,OUT_ARG0(%esp)
6318     call     dvmResolveString          # go resolve
6319     UNSPILL(rPC)
6320     testl    %eax,%eax                 # failed?
6321     je       common_exceptionThrown
6322     SET_VREG(%eax,rINST_FULL)
6323     FETCH_INST_WORD(2)
6324     ADVANCE_PC(2)
6325     GOTO_NEXT
...
6352 /* This is the less common path, so we'll redo some work
6353    here rather than force spills on the common path */
```

```
6354 .LOP_CONST_CLASS_resolve:
6355     GET_GLUE(%eax)
6356     movl      %ecx,rINST_FULL           # rINST_FULL<- AA
6357     EXPORT_PC()
6358     movl      offGlue_method(%eax),%eax # eax<- glue->method
6359     movl      $1,OUT_ARG2(%esp)        # true
6360     movzwl    2(rPC),%ecx               # ecx<- BBBB
6361     movl      offMethod_clazz(%eax),%eax
6362     SPILL(rPC)
6363     movl      %ecx,OUT_ARG1(%esp)
6364     movl      %eax,OUT_ARG0(%esp)
6365     call      dvmResolveClass           # go resolve
...
7505     /*
7506      * Go resolve the field
7507      */
7508 .LOP_SGET_resolve:
7509     GET_GLUE(%ecx)
7510     movzwl    2(rPC),%eax                       # eax<- field ref BBBB
7511     movl      offGlue_method(%ecx),%ecx         # ecx<- current method
7512     EXPORT_PC()                                 # could throw, need to export
7513     movl      offMethod_clazz(%ecx),%ecx        # ecx<- method->clazz
7514     SPILL(rPC)
7515     movl      %eax,OUT_ARG1(%esp)
7516     movl      %ecx,OUT_ARG0(%esp)
7517     call      dvmResolveStaticField             # eax<- resolved StaticField ptr
7518     UNSPILL(rPC)
7519     testl     %eax,%eax
7520     jne       .LOP_SGET_finish                # success, continue
7521     jmp       common_exceptionThrown            # no, handle exception
...
7843     /* At this point:
7844      * ecx = null (needs to be resolved base method)
7845      * eax = method->clazz
7846     */
7847 .LOP_INVOKE_SUPER_resolve:
7848     SPILL_TMP(%eax)                     # method->clazz
7849     movl    %eax,OUT_ARG0(%esp)         # arg0<- method->clazz
7850     movzwl  2(rPC),%ecx                 # ecx<- BBBB
7851     movl    $METHOD_VIRTUAL,OUT_ARG2(%esp)  # arg2<- resolver method type
7852     movl    %ecx,OUT_ARG1(%esp)         # arg1<- ref
7853     SPILL(rPC)
7854     call    dvmResolveMethod            # eax<- call(clazz, ref, flags)
7855     UNSPILL(rPC)
7856     testl   %eax,%eax                   # got null?
7857     movl    %eax,%ecx                   # ecx<- resolved base method
7858     UNSPILL_TMP(%eax)                   # restore method->clazz
7859     jne     .LOP_INVOKE_SUPER_continue        # good to go - continue
7860     jmp     common_exceptionThrown      # handle exception
...
```

299. For another example, another x86 assembler interpreter has the following code:

```
http://android.git.kernel.org/?p=platform/dalvik.git;a=blob;f=vm/mterp/out/InterpAsm-x86.S

6983 .LOP_IGET_resolve:
6984     EXPORT_PC()
6985     SPILL(rPC)
6986     movl    offGlue_method(rIBASE),rPC            # rPC<- current method
6987     UNSPILL(rIBASE)
6988     movl    offMethod_clazz(rPC),rPC              # rPC<- method->clazz
6989     SPILL_TMP(%ecx)                               # save object pointer across call
6990     movl    rPC,OUT_ARG0(%esp)                    # pass in method->clazz
6991     call    dvmResolveInstField                   #  ... to dvmResolveInstField
6992     UNSPILL_TMP(%ecx)
6993     UNSPILL(rPC)
6994     testl   %eax,%eax                             #  ... which returns InstrField ptr
6995     jne     .LOP_IGET_finish
6996     jmp     common_exceptionThrown
```

300. For another example, the main interpreter entry point and support functions has the following code:

```
\dalvik\vm\interp\Interp.c

 952 /*
 953  * Find the concrete method that corresponds to "methodIdx".  The code in
 954  * "method" is executing invoke-method with "thisClass" as its first argument.
 955  *
 956  * Returns NULL with an exception raised on failure.
 957  */
 958 Method* dvmInterpFindInterfaceMethod(ClassObject* thisClass, u4 methodIdx,
 959     const Method* method, DvmDex* methodClassDex)
 960 {
 961     Method* absMethod;
 962     Method* methodToCall;
 963     int i, vtableIndex;
 964
 965     /*
 966      * Resolve the method.  This gives us the abstract method from the
 967      * interface class declaration.
 968      */
 969     absMethod = dvmDexGetResolvedMethod(methodClassDex, methodIdx);
 970     if (absMethod == NULL) {
 971         absMethod = dvmResolveInterfaceMethod(method->clazz, methodIdx);
 972         if (absMethod == NULL) {
 973             LOGV("+ unknown method\n");
 974             return NULL;
 975         }
 976     }
 977
```

301. For the reasons described above, Android literally satisfies the limitations of Claim 11 and in at least two ways which I have dubbed the "dexopt" (Optimize.c) and "Resolve.c" modes of literal infringement, respectively.

302. Android meets the limitations of **claim 12** in the same ways Android literally infringes claim 11 of the '104 patent.

303. **Claim 12** recites "A computer-readable medium containing instructions for controlling a data processing system to perform a method for interpreting intermediate form object code comprised of instructions, certain of said instructions containing one or more symbolic references, said method comprising the steps of….", where the method that follows comprises the steps of interpreting and resolving. Android devices are data processing systems that include computer-readable mediums such as memories that contain the Android software (including the Dalvik virtual machine), which causes the system to perform the method set forth in claim 12. For the reasons described above and here, Android will perform the steps and substeps method set forth in claim 12 when in operation, and so any computer-readable medium storing a copy of Android will infringe claim 12. Such computer-readable mediums may be found in devices that that store, distribute, or run Android or the Android SDK, including websites, servers, host computers, and mobile devices.

304. Android infringes **limitation [12-a] of claim 12**, which recites "interpreting said instructions in accordance with a program execution control." The Dalvik virtual machine (which runs the routines in Resolve.c to resolve symbolic references) interprets intermediate form object code in accordance with a program execution control (for example, the Android operating system can switch between multiple active Dalvik VM instances).

305. Dexopt is also part of the bytecode interpretation process because it loads classes into the VM and runs over their bytecodes for verification and optimization purposes:

See, e.g., dalvik\docs\dexopt.html; *see also* http://android.git.kernel.org/?p=platform/dalvik.git;a=blob;f=docs/dexopt.html:

….

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).

...

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).").

306. I have covered the reasons Android performs the **resolving step**, **determining substep**, and **storing substep** in my discussion of Android's infringement of claim 11.

307. Android literally infringes **claim 15** in the same way that Android literally infringes claim 11 of the '104 patent.

308. Claim 15 is a method claim that depends on claim 13, whose **preamble** recites "A computer-implemented method for executing instructions, certain of said instructions containing one or more symbolic references, said method comprising the steps of." To the extent the preamble limits the scope of the claim, it is met by Android. Android's Dalvik VM will execute

instructions, some of which contain symbolic references, when it runs. I have covered the reasons Android performs the **resolving step**, **determining substep**, and **storing substep** of claim 13 in my discussion of Android's infringement of claim 11.

309. Next, claim 15 also recites "The method of claim [13], wherein said step of resolving said symbolic reference further comprises the substep of executing said instruction containing said symbolic reference using the stored numerical reference." The source code excerpted below is one example showing that virtual machine instructions containing the resolved symbolic references get executed, thereby showing that Android literally infringes claim 15, *e.g.*:

```
dalvik\vm\mterp\out\InterpAsm-armv5te.S:
/* ------------------------------ */
    .balign 64
.L_OP_NEW_INSTANCE: /* 0x22 */
/* File: armv5te/OP_NEW_INSTANCE.S */
    /*
     * Create a new instance of a class.
     */
    /* new-instance vAA, class@BBBB */
    ldr     r3, [rGLUE, #offGlue_methodClassDex]    @ r3<- pDvmDex
    FETCH(r1, 1)                        @ r1<- BBBB
    ldr     r3, [r3, #offDvmDex_pResClasses]    @ r3<- pDvmDex->pResClasses
    ldr     r0, [r3, r1, lsl #2]        @ r0<- resolved class
    EXPORT_PC()                         @ req'd for init, resolve, alloc
    cmp     r0, #0                      @ already resolved?
    beq     .LOP_NEW_INSTANCE_resolve         @ no, resolve it now
.LOP_NEW_INSTANCE_resolved:   @ r0=class
    ldrb    r1, [r0, #offClassObject_status]    @ r1<- ClassStatus enum
    cmp     r1, #CLASS_INITIALIZED      @ has class been initialized?
    bne     .LOP_NEW_INSTANCE_needinit        @ no, init class now
.LOP_NEW_INSTANCE_initialized: @ r0=class
    mov     r1, #ALLOC_DONT_TRACK       @ flags for alloc call
    bl      dvmAllocObject              @ r0<- new object
    b       .LOP_NEW_INSTANCE_finish          @ continue
….

/* continuation for OP_NEW_INSTANCE */

    .balign 32                          @ minimize cache lines
.LOP_NEW_INSTANCE_finish: @ r0=new object
    mov     r3, rINST, lsr #8           @ r3<- AA
    cmp     r0, #0                      @ failed?
    beq     common_exceptionThrown      @ yes, handle the exception
    FETCH_ADVANCE_INST(2)               @ advance rPC, load rINST
    GET_INST_OPCODE(ip)                 @ extract opcode from rINST
```

```
  SET_VREG(r0, r3)                    @ vAA<- r0
GOTO_OPCODE(ip)                     @ jump to next instruction
```

310. Android literally infringes **claim 17** in the same ways Android literally infringes claim 11 of the '104 patent. Claim 17 is a method claim. The **preamble of claim 17** recites "In a computer system comprising a program, a method for executing said program comprising the steps of." To the extent the preamble limits the scope of the claim, it is met by Android. Android executes programs in a computer system.

311. My discussion and analysis of the **resolving step** above apply equally to the **converting step** of claim 17. I have covered the **resolving substep**, **storing substep**, and **obtaining substep** in my discussion of Android's infringement of claims 11 and 15.

312. Android literally infringes **claim 22** in the same ways Android literally infringes claim 11 of the '104 patent. Once a .dex file (containing intermediate form object code with symbolic data references in certain instructions) is loaded (received) onto an Android device, it is executed without being recompiled. Even versions of Android that include a JIT do not immediately compile code, but instead allow the code to be interpreted until a profiler determines that a portion of the code should be compiled.

313. Android literally infringes **claim 27** in the same ways Android literally infringes claims 11, 15, and 17 of the '104 patent. To the extent the preamble limits the scope of the claim, it is met by Android. Android includes computer-implemented methods for performing the steps described in the claim.

314. Android literally infringes **claim 29** in the same ways Android literally infringes claims 11, 15, and 17 of the '104 patent. To the extent the preamble limits the scope of the claim, it is met by Android. Storage media that contain Android source or binary code, which include Android devices, system images, the Android git repository server, and the Android SDK are computer program products containing instructions for causing a computer to perform the steps described in the claim.

315. Android literally infringes **claim 38** in the same ways Android literally infringes claims 11, 15, and 17 of the '104 patent. To the extent the preamble limits the scope of the claim, it is met by Android. Storage media that contain Android source or binary code, which include Android devices, system images, the Android git repository server, and the Android SDK are computer program products containing instructions for causing a computer to perform the steps described in the claim.

316. Android literally infringes **claim 39** in the same ways Android literally infringes claims 11, 15, and 17 of the '104 patent. To the extent the preamble limits the scope of the claim, it is met by Android. Android includes computer-implemented methods for performing the steps described in the claim.

317. Android literally infringes **claim 40** in the same ways Android literally infringes claims 11, 15, and 17 of the '104 patent. To the extent the preamble limits the scope of the claim, it is met by Android. An Android device includes memory that stores dex and odex files containing intermediate form object code. The Android device runs code implementing the Dalvik virtual machine, thereby configuring the hardware processor on the Android device to execute the .dex formatted bytecode or the .odex formatted bytecode as discussed above.

318. Android literally infringes **claim 41** in the same ways Android literally infringes claims 11, 15, and 17 of the '104 patent. To the extent the preamble limits the scope of the claim, it is met by Android. Storage media that contain Android source or binary code, which include Android devices, system images, the Android git repository server, and the Android SDK are computer program products containing instructions for causing a computer to perform the steps described in the claim.

443. The smaller size of the m-class file (relative to the plurality of class files before processing) results in the classes taking up less space on servers or storage devices, less network or file transfer time to read, less memory when loaded, and faster execution (in part, because shared constants are resolved at most once).

444. Multi-class files further consolidate the loading of required classes instead of loading the classes one by one. Using allocation information, only one dynamic memory allocation is needed instead of multiple allocation operations. This results in less fragmentation, less time spent in the allocator, and less waste of memory space. Because the class files are consolidated in a single multi-class file, only a single transaction is typically needed to perform a network or file system search, to set up a transfer session (e.g., HTTP), and to transfer the entire set of classes. This minimizes pauses in the execution that can result from such transactions and provides for deterministic execution, with no pauses for class loading during a program run. Also, once the multi-class file is loaded and parsed, there is no need for the computer executing the program to remain connected to the source of the classes.

445. I have illustrated these concepts in the animations that were shown or submitted to the Court for the April 6, 2011 Technology Tutorial. If called upon to testify, I intend to use these animations to explain the technology and state of the art involved in the '702 patent.

C. Detailed Infringement Analysis

446. I have compared Google's Android to the elements recited in claims 1, 6, 7, 12, 13, 15, and 16 of the '702 patent.

447. All of the analysis below concerning infringement should be read together with the material in the claim chart of Exhibit C attached to Oracle's infringement contentions submitted to Google on April 1, 2011. I participated in the analysis and preparation of the Exhibit C chart. I agree with the conclusions of those charts and the evidence supporting those conclusions. While my report contains a narrative-style infringement analysis, this analysis is intended to accompany the additional information supplied in the charts that in some cases provides more details.

448. The infringement evidence illustrated below is exemplary and not exhaustive. The cited examples are largely taken from Android 2.2, 2.3, and Google's Android websites. My analysis applies to all versions of Android having similar or nearly identical code or documentation, including past and expected future releases. I understand that the publicly released versions of Android before version 2.2 operate as I describe below; I have not been given the opportunity to analyze versions of Android from 3.0 and beyond.

449. In my opinion, Google's Android literally infringes the asserted claims of the '702 patent. As described in various sources from Google, and found directly in the Android code, Google's Android dx tool pre-processes class files according to claim 1, the process including determining a plurality of duplicated elements in a plurality of class files, forming a shared table comprising said plurality of duplicated elements, removing said duplicated elements from said plurality of class files to obtain a plurality of reduced class files, and forming a multi-class file comprising said plurality of reduced class files and said shared table. The dx tool is distributed as part of the Android SDK. The users of the dx tool include Google and device manufacturers, who use it to build Android system images and applications, as well as application developers, who use the Android SDK to develop applications by converting compiled .class files to Android .dex files. There is no good reason to think that anyone alters the dx tool so that it does not practice the patented method and many reasons to think that nobody alters the dx tool in any significant way.

450. Because the dx tool comprises computer readable program code for pre-processing class files, computer-usable media that contain the dx code make up computer program products that meet all the requirements of claim 7 and its asserted dependent claims. These products include websites, servers, and computers that host the Android SDK for download and computers that have downloaded the Android SDK from Google. The products also include Google's own internal versions of the dx tool that it uses for system builds and application development. Computers that have the dx tool installed, whether as part of the Android SDK or separately, include a processor and a memory. Before the dx tool is run to

create a .dex file, one or more class files are created by running a Java compiler such as that found in the JDK, which Google urges Android SDK users to download. (*See* "Installing the SDK," http://developer.android.com/sdk/installing.html.) When the dx tool runs, is a process configured to form a multi-class file as described below. For these reasons, computers that have the dx tool installed will be reasonably capable of meeting all the requirements of claim 13 and its asserted dependent claims, and will in fact meet all those requirements when the dx tool is used for its intended purpose of converting .class files to a .dex file.

451. In reviewing the Android source code, I have summarized the process the dx tool follows to form .dex files from .class files. The process generally starts with the source code file Main.java in the com.android.dx.command.dexer package (path /src/com/android/dx/command/dexer/Main.java), which contains code to process each input Java class in turn. Specifically, the run method called by main calls processAllFiles, which then calls processOne on each filename (line 220).

```
198     /**
199      * Constructs the output {@link DexFile}, fill it in with all the
200      * specified classes, and populate the resources map if required.
201      *
202      * @return whether processing was successful
203      */
204     private static boolean processAllFiles() {
205         outputDex = new DexFile();
206
207         if (args.jarOutput) {
208             outputResources = new TreeMap<String, byte[]>();
209         }
210
211         if (args.dumpWidth != 0) {
212             outputDex.setDumpWidth(args.dumpWidth);
213         }
214
215         boolean any = false;
216         String[] fileNames = args.fileNames;
217
218         try {
219             for (int i = 0; i < fileNames.length; i++) {
220                 any |= processOne(fileNames[i]);
221             }
222         } catch (StopProcessing ex) {
223             /*
224              * Ignore it and just let the warning/error reporting do
225              * their things.
226              */
227         }
228
229         if (warnings != 0) {
230             DxConsole.err.println(warnings + " warning" +
231                                   ((warnings == 1) ? "" : "s"));
232         }
233
```

```
234         if (errors != 0) {
235             DxConsole.err.println(errors + " error" +
236                     ((errors == 1) ? "" : "s") + "; aborting");
237             return false;
238         }
239
240         if (!(any || args.emptyOk)) {
241             DxConsole.err.println("no classfiles specified");
242             return false;
243         }
244
245         if (args.optimize && args.statistics) {
246             CodeStatistics.dumpStatistics(DxConsole.out);
247         }
248
249         return true;
250     }
251
```

452. The processOne method processFileBytes which leads to method processClass when class files are processed. For each class, processClass creates a ClassDefItem created as the internal representation of the class (see lines 336-337) and puts the resulting information into an object representing the .dex "output file in-progress" (line 338).

```
322     /**
323      * Processes one classfile.
324      *
325      * @param name {@code non-null;} name of the file, clipped such that it
326      * <i>should</i> correspond to the name of the class it contains
327      * @param bytes {@code non-null;} contents of the file
328      * @return whether processing was successful
329      */
330     private static boolean processClass(String name, byte[] bytes) {
331         if (! args.coreLibrary) {
332             checkClassName(name);
333         }
334
335         try {
336             ClassDefItem clazz =
337                 CfTranslator.translate(name, bytes, args.cfOptions);
338             outputDex.add(clazz);
339             return true;
340         } catch (ParseException ex) {
341             DxConsole.err.println("\ntrouble processing:");
342             if (args.debug) {
343                 ex.printStackTrace(DxConsole.err);
344             } else {
345                 ex.printContext(DxConsole.err);
346             }
347         }
348
349         warnings++;
350         return false;
351     }
352
```

453. Each ClassDefItem, created by method CfTranslator.translate defined in source code file, includes the fields, methods and annotations of the class. Each method includes the Dalvik byte code for the method, which is translated from the original Java byte code. In more

detail, file CfTranslator.java, at path dx/src/com/android/dx/dex/cf/CfTranslator.java, defines a static method that turns a bytecode array containing Java classfiles into ClassDefItem instances (comment lines 58-59).

```
57 /**
58  * Static method that turns {@code byte[]}s containing Java
59  * classfiles into {@link ClassDefItem} instances.
60  */
61 public class CfTranslator {
62     /** set to {@code true} to enable development-time debugging code */
63     private static final boolean DEBUG = false;
64
65     /**
66      * This class is uninstantiable.
67      */
68     private CfTranslator() {
69         // This space intentionally left blank.
70     }
71
```

454. At line 82 in CfTranslator.java there is a method translate(), which turns a byte array containing a Java classfile into a corresponding ClassDefItem.

```
72     /**
73      * Takes a {@code byte[]}, interprets it as a Java classfile, and
74      * translates it into a {@link ClassDefItem}.
75      *
76      * @param filePath {@code non-null;} the file path for the class,
77      * excluding any base directory specification
78      * @param bytes {@code non-null;} contents of the file
79      * @param args command-line arguments
80      * @return {@code non-null;} the translated class
81      */
82     public static ClassDefItem translate(String filePath, byte[] bytes,
83             CfOptions args) {
84         try {
85             return translate0(filePath, bytes, args);
86         } catch (RuntimeException ex) {
87             String msg = "...while processing " + filePath;
88             throw ExceptionWithContext.withContext(ex, msg);
89         }
90     }
```

455. The ClassDefItem contains all of the fields, methods, annotations, and constants of the class. At this stage, each class is processed individually, a ClassDefItem being separately created for each input Java class without regard for potential duplication of constant pool entries.

456. Once the ClassDefItem is completely formed, it is added to an object of class DexFile by calling DexFile.add(). This call occurs in line 338 of Main.java.

```
322     /**
323      * Processes one classfile.
324      *
325      * @param name {@code non-null;} name of the file, clipped such that it
326      * <i>should</i> correspond to the name of the class it contains
327      * @param bytes {@code non-null;} contents of the file
328      * @return whether processing was successful
```

```
329      */
330     private static boolean processClass(String name, byte[] bytes) {
331         if (! args.coreLibrary) {
332             checkClassName(name);
333         }
334
335         try {
336             ClassDefItem clazz =
337                 CfTranslator.translate(name, bytes, args.cfOptions);
338             outputDex.add(clazz);
339             return true;
340         } catch (ParseException ex) {
341             DxConsole.err.println("\ntrouble processing:");
342             if (args.debug) {
343                 ex.printStackTrace(DxConsole.err);
344             } else {
345                 ex.printContext(DxConsole.err);
346             }
347         }
348
349         warnings++;
350         return false;
351     }
352
```

457. The source code for DexFile appears in file dx/src/com/android/dx/dex/file/DexFile.java. DexFile has a field classDefs (line 120) of class ClassDefsSection that accumulates the ClassDefItem objects (see lines 142-143).

```
136     /**
137      * Adds a class to this instance. It is illegal to attempt to add more
138      * than one class with the same name.
139      *
140      * @param clazz {@code non-null;} the class to add
141      */
142     public void add(ClassDefItem clazz) {
143         classDefs.add(clazz);
144     }
```

458. After all of the input classes have been processed as described above there is a pass that causes all of the constant pool entries in the ClassDefItem objects to be interned. It is at this stage that the TreeMaps for the constant pool sections are populated and the duplicate removal takes place. For instance, at line 434 of x/src/com/android/dx/command/dexer/Main.java there is a call to outputDex.toDex(). The implementation of toDex is at 196 of DexFile; method toDex() calls toDex0(), which is at line 476 of the same file.

```
185     /**
186      * Returns the contents of this instance as a {@code .dex} file,
187      * in {@code byte[]} form.
188      *
189      * @param humanOut {@code null-ok;} where to write human-oriented output to
190      * @param verbose whether to be verbose when writing human-oriented output
191      * @return {@code non-null;} a {@code .dex} file for this instance
```

```
192      */
193     public byte[] toDex(Writer humanOut, boolean verbose)
194         throws IOException {
195         boolean annotate = (humanOut != null);
196         ByteArrayAnnotatedOutput result = toDex0(annotate, verbose);
197
198         if (annotate) {
199             result.writeAnnotationsTo(humanOut);
200         }
201
202         return result.getArray();
203     }


468     /**
469      * Returns the contents of this instance as a {@code .dex} file,
470      * in a {@link ByteArrayAnnotatedOutput} instance.
471      *
472      * @param annotate whether or not to keep annotations
473      * @param verbose if annotating, whether to be verbose
474      * @return {@code non-null;} a {@code .dex} file for this instance
475      */
476     private ByteArrayAnnotatedOutput toDex0(boolean annotate,
477             boolean verbose) {
478         /*
479          * The following is ordered so that the prepare() calls which
480          * add items happen before the calls to the sections that get
481          * added to.
482          */
483
484         classDefs.prepare();
485         classData.prepare();
486         wordData.prepare();
487         byteData.prepare();
488         methodIds.prepare();
489         fieldIds.prepare();
490         protoIds.prepare();
491         typeLists.prepare();
492         typeIds.prepare();
493         stringIds.prepare();
494         stringData.prepare();
495         header.prepare();
496
497         // Place the sections within the file.
498
499         int count = sections.length;
500         int offset = 0;
501
502         for (int i = 0; i < count; i++) {
503             Section one = sections[i];
504             int placedAt = one.setFileOffset(offset);
505             if (placedAt < offset) {
506                 throw new RuntimeException("bogus placement for section " + i);
507             }
508
509             try {
510                 if (one == map) {
511                     /*
512                      * Inform the map of all the sections, and add it
513                      * to the file. This can only be done after all
514                      * the other items have been sorted and placed.
515                      */
516                     MapItem.addMap(sections, map);
517                     map.prepare();
518                 }
519
520                 if (one instanceof MixedItemSection) {
521                     /*
522                      * Place the items of a MixedItemSection that just
523                      * got placed.
```

```
524                          */
525                         ((MixedItemSection) one).placeItems();
526                     }
527
528                     offset = placedAt + one.writeSize();
529                 } catch (RuntimeException ex) {
530                     throw ExceptionWithContext.withContext(ex,
531                             "...while writing section " + i);
532                 }
533             }
534
```

459. The implementation of toDex0() calls a method called prepare() for the sections of the class file, e.g., lines 484-490 of Dexfile.java above. Considering classDefs.prepare(), the classDefs object is an instance of the ClassDefsSection class (e.g., line 120 of Dexfile). The class ClassDefsSection is a subclass of UniformItemSection, as shown in ClassDefsSection.java. UniformItemSection is defined in UniformItemSection.java as subclass of Section. The prepare() method is defined by Section in line 212 of dx/src/com/android/dx/dex/file/Section.java. The prepare method calls another method prepare0(), which is abstract in Section, and is implemented at line 70 of dx/src/com/android/dx/dex/file/UniformItemSection.java. Method prepare0 iterates through each item in the ClassDefsSection calling a method called addContents() on each item.

```
68      /** {@inheritDoc} */
69      @Override
70      protected final void prepare0() {
71          DexFile file = getFile();
72
73          orderItems();
74
75          for (Item one : items()) {
76              one.addContents(file);
77          }
78      }
```

460. Each kind of item in the class (e.g., method, field, annotation) is represented by a different class, all of which extend the base class Item. For example, a field is represented by an instance of the class EncodedField, found in dx/src/com/android/dx/dex/file/EncodedField.java. The addContents() method of this class is found in line 99 of that file. The addContents method is passed a reference to the DexFile so that it can fetch the FieldIdsSection from the file, and intern the constant for the field by calling the intern() method on FieldIdsSection.

```
 97      /** {@inheritDoc} */
 98      @Override
 99      public void addContents(DexFile file) {
100          FieldIdsSection fieldIds = file.getFieldIds();
101          fieldIds.intern(field);
102      }
```

461. The FieldIdsSection represents the aggregated FieldId constant pool entries for the whole .dex file (i.e. for all of the classes that are being included into the .dex file). The intern() method of class FieldIdsSection, for example, is defined in FieldIdsSection.java at lines 92-113. The intern() method calls FieldIds.put. As shown at the top of that file, e.g., line 45, FieldIds is an instance of class TreeMap. Thus, intern() uses a TreeMap, explained below, to hold the interned field constants. This is the step at which removal of duplicate constants is performed. The same process applies for methods, strings and types, which are represented in the ClassDefItem by other subclasses of Item.

```
/**
 28  * Field refs list section of a {@code .dex} file.
 29  */
 30 public final class FieldIdsSection extends MemberIdsSection {
 31     /**
 32      * {@code non-null;} map from field constants to {@link
 33      * FieldIdItem} instances
 34      */
 35     private final TreeMap<CstFieldRef, FieldIdItem> fieldIds;
 36
 37     /**
 38      * Constructs an instance. The file offset is initially unknown.
 39      *
 40      * @param file {@code non-null;} file that this instance is part of
 41      */
 42     public FieldIdsSection(DexFile file) {
 43         super("field_ids", file);
 44
 45         fieldIds = new TreeMap<CstFieldRef, FieldIdItem>();
 46     }


/**
 93      * Interns an element into this instance.
 94      *
 95      * @param field {@code non-null;} the reference to intern
 96      * @return {@code non-null;} the interned reference
 97      */
 98     public FieldIdItem intern(CstFieldRef field) {
 99         if (field == null) {
100             throw new NullPointerException("field == null");
101         }
102
103         throwIfPrepared();
104
105         FieldIdItem result = fieldIds.get(field);
106
107         if (result == null) {
108             result = new FieldIdItem(field);
109             fieldIds.put(field, result);
110         }
```

```
111
112         return result;
113     }
```

462. TreeMap is part of a java collections framework. Therefore, when the Android SDK is built from source code this portion of the system is based on code provided by the JDK. Documentation for Java 2 Platform SE 5.0 is available at https://download.oracle.com/javase/1.5.0/docs/api/java/util/TreeMap.html. Java Class TreeMap extends the java.util.AbstractMap. Note that for any given key, a TreeMap stores at most one mapping for that key, as explained in the sentence "If the map previously contained a mapping for this key, the old value is replaced."

put

```
public V put(K key,
             V value)
```

Associates the specified value with the specified key in this map. If the map previously contained a mapping for this key, the old value is replaced.

Specified by:
put in interface Map<K,V>

Overrides:
put in class AbstractMap<K,V>

Parameters:
key - key with which the specified value is to be associated.
value - value to be associated with the specified key.

Returns:
previous value associated with specified key, or null if there was no mapping for key. A null return can also indicate that the map previously associated null with the specified key.

Throws:
ClassCastException - key cannot be compared with the keys currently in the map.
NullPointerException - key is null and this map uses natural order, or its comparator does not tolerate null keys.

463. With regard to specific claim elements, the **preamble of claim 1** recites a "method of pre-processing class files." The Android dx tool performs a method of pre-

processing .class files into a Dalvik executable format (.dex) file. For example, the dx tool is described as follows:

> **"dx**
>
> The dx tool lets you generate Android bytecode from .class files. The tool converts target files and/or directories to Dalvik executable format (.dex) files, so that they can run in the Android environment."
>
> Android Developer Tools available at http://developer.android.com/guide/developing/tools/othertools.html

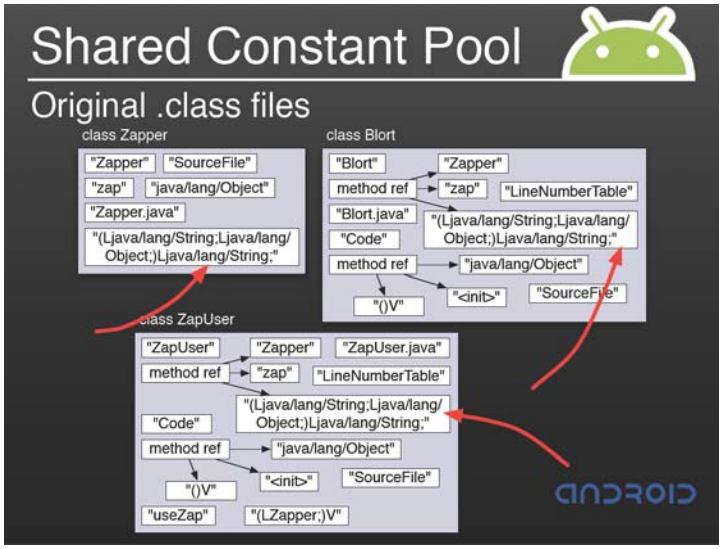
464. The pre-processing of class files performed by the dx tool into a .dex file that can be interpreted by the Dalvik Virtual Machine (Dalvik VM) is further explained in the Dalvik VM video presentation and related presentation from Google I/O 2008, dated 5/29/2008. (*See* Google I/O 2008 Video entitled "Dalvik Virtual Machine Internals," presented by Dan Bornstein, *available at* http://developer.android.com/videos/index.html#v=ptjedOZEXPM ("Dalvik Video"), at 5:45–10:45; *see also* Google I/O 2008 Presentation entitled "Dalvik Virtual Machine Internals" at Slides 11-22, presented by Dan Bornstein, *available at* http://sites.google.com/site/io/dalvik-vm-internals/2008-05-29-Presentation-Of-Dalvik-VM-Internals.pdf?attredirects=0 ("Dalvik Presentation").)

465. Further, the Android source code performs preprocessing of class files, which can be found, for example, in the following packages, which are operable to preprocess class files:

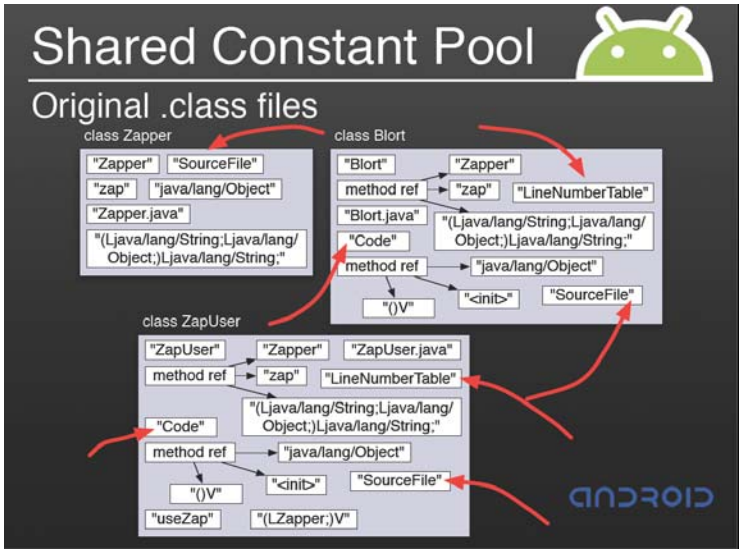
> "Classes for translating Java classfiles into Dalvik classes.
> PACKAGES USED:
> - com.android.dx.cf.code
> - com.android.dx.cf.direct
> - com.android.dx.cf.iface
> - com.android.dx.dex.code
> - com.android.dx.dex.file
> - com.android.dx.rop.code
> - com.android.dx.rop.cst
> - com.android.dx.util"

dalvik\dx\src\com\android\dx\dex\cf\package.html.

466. The **first element of claim 1**, recites "determining a plurality of duplicated elements in a plurality of class files." The Android dx tool determines a plurality of duplicated elements in a plurality of class files as clearly illustrated and explained in the Dalvik Video at 7:50-8:45 and Dalvik Presentation, Slides 18-19. In particular, the Dalvik Presentation shows the determination of a plurality of duplicated elements (e.g., class signatures and string names) in a plurality of class files are determined:



(Dalvik Presentation, Slide 18)
(Shows identification of common class signatures in the class files)



(Dalvik Presentation, Slide 19)
(Shows identification of common string names in the class files)

467. In the Android source code, portions of the above features can be found in the dx.rop.type packages and dx.util package, as follows:

> "Interfaces and implementation of things related to the constant pool.
> PACKAGES USED:
> * com.android.dx.rop.type
> * com.android.dx.util"

dalvik/dx/src/com/android/dx/rop/cst/package.html.

Additionally, the DexFile.java code indicates:

```
440
441     /**
442      * Gets the {@link IndexedItem} corresponding to the given constant,
443      * if it is a constant that has such a correspondence, or return
444      * {@code null} if it isn't such a constant. This will throw
445      * an exception if the given constant <i>should</i> have been found
446      * but wasn't.
447      *
448      * @param cst {@code non-null;} the constant to look up
449      * @return {@code null-ok;} its corresponding item, if it has a corresponding
450      * item, or {@code null} if it's not that sort of constant
451      */
452     /*package*/ IndexedItem findItemOrNull(Constant cst) {
453         IndexedItem item;
454
455         if (cst instanceof CstString) {
456             return stringIds.get(cst);
457         } else if (cst instanceof CstType) {
458             return typeIds.get(cst);
459         } else if (cst instanceof CstBaseMethodRef) {
460             return methodIds.get(cst);
461         } else if (cst instanceof CstFieldRef) {
462             return fieldIds.get(cst);
463         } else {
464             return null;
465         }
466     }
467
468     /**
469      * Returns the contents of this instance as a {@code .dex} file,
470      * in a {@link ByteArrayAnnotatedOutput} instance.
471      *
472      * @param annotate whether or not to keep annotations
473      * @param verbose if annotating, whether to be verbose
474      * @return {@code non-null;} a {@code .dex} file for this instance
475      */
476     private ByteArrayAnnotatedOutput toDex0(boolean annotate,
477             boolean verbose) {
478         /*
479          * The following is ordered so that the prepare() calls which
480          * add items happen before the calls to the sections that get
481          * added to.
482          */
483
484         classDefs.prepare();
485         classData.prepare();
486         wordData.prepare();
487         byteData.prepare();
488         methodIds.prepare();
489         fieldIds.prepare();
490         protoIds.prepare();
```

```
491         typeLists.prepare();
492         typeIds.prepare();
493         stringIds.prepare();
494         stringData.prepare();
495         header.prepare();
496
497         // Place the sections within the file.
498
499         int count = sections.length;
500         int offset = 0;
501
502         for (int i = 0; i < count; i++) {
503             Section one = sections[i];
504             int placedAt = one.setFileOffset(offset);
505             if (placedAt < offset) {
506                 throw new RuntimeException("bogus placement for section " + i);
507             }
508
509             try {
510                 if (one == map) {
511                     /*
512                      * Inform the map of all the sections, and add it
513                      * to the file. This can only be done after all
514                      * the other items have been sorted and placed.
515                      */
516                     MapItem.addMap(sections, map);
517                     map.prepare();
518                 }
519
520                 if (one instanceof MixedItemSection) {
521                     /*
522                      * Place the items of a MixedItemSection that just
523                      * got placed.
524                      */
525                     ((MixedItemSection) one).placeItems();
526                 }
527
528                 offset = placedAt + one.writeSize();
529             } catch (RuntimeException ex) {
530                 throw ExceptionWithContext.withContext(ex,
531                         "...while writing section " + i);
532             }
533         }
534
535         // Write out all the sections.
536
537         fileSize = offset;
538         byte[] barr = new byte[fileSize];
539         ByteArrayAnnotatedOutput out = new ByteArrayAnnotatedOutput(barr);
540
541         if (annotate) {
542             out.enableAnnotations(dumpWidth, verbose);
543         }
544
545         for (int i = 0; i < count; i++) {
546             try {
547                 Section one = sections[i];
548                 int zeroCount = one.getFileOffset() - out.getCursor();
549                 if (zeroCount < 0) {
550                     throw new ExceptionWithContext("excess write of " +
551                             (-zeroCount));
552                 }
553                 out.writeZeroes(one.getFileOffset() - out.getCursor());
554                 one.writeTo(out);
555             } catch (RuntimeException ex) {
556                 ExceptionWithContext ec;
557                 if (ex instanceof ExceptionWithContext) {
558                     ec = (ExceptionWithContext) ex;
559                 } else {
560                     ec = new ExceptionWithContext(ex);
```

```
561                 }
562                 ec.addContext("...while writing section " + i);
563                 throw ec;
564             }
565         }
566
567         if (out.getCursor() != fileSize) {
568             throw new RuntimeException("foreshortened write");
569         }
570
571         // Perform final bookkeeping.
572
573         calcSignature(barr);
574         calcChecksum(barr);
575
576         if (annotate) {
577             wordData.writeIndexAnnotation(out, ItemType.TYPE_CODE_ITEM,
578                     "\nmethod code index:\n\n");
579             getStatistics().writeAnnotation(out);
580             out.finishAnnotating();
581         }
582
583         return out;
584     }
585
586     /**
587      * Generates and returns statistics for all the items in the file.
588      *
589      * @return {@code non-null;} the statistics
590      */
591     public Statistics getStatistics() {
592         Statistics stats = new Statistics();
593
594         for (Section s : sections) {
595             stats.addAll(s);
596         }
597
598         return stats;
599     }
600
601     /**
602      * Calculates the signature for the {@code .dex} file in the
603      * given array, and modify the array to contain it.
604      *
605      * @param bytes {@code non-null;} the bytes of the file
606      */
607     private static void calcSignature(byte[] bytes) {
608         MessageDigest md;
609
610         try {
611             md = MessageDigest.getInstance("SHA-1");
612         } catch (NoSuchAlgorithmException ex) {
613             throw new RuntimeException(ex);
614         }
615
616         md.update(bytes, 32, bytes.length - 32);
617
618         try {
619             int amt = md.digest(bytes, 12, 20);
620             if (amt != 20) {
621                 throw new RuntimeException("unexpected digest write: " + amt +
622                                            " bytes");
623             }
624         } catch (DigestException ex) {
625             throw new RuntimeException(ex);
626         }
627     }
628
629     /**
630      * Calculates the checksum for the {@code .dex} file in the
```

```
631      * given array, and modify the array to contain it.
632      *
633      * @param bytes {@code non-null;} the bytes of the file
634      */
635     private static void calcChecksum(byte[] bytes) {
636         Adler32 a32 = new Adler32();
637
638         a32.update(bytes, 12, bytes.length - 12);
639
640         int sum = (int) a32.getValue();
641
642         bytes[8]  = (byte) sum;
643         bytes[9]  = (byte) (sum >> 8);
644         bytes[10] = (byte) (sum >> 16);
645         bytes[11] = (byte) (sum >> 24);
646     }
647 }
```

dalvik/dx/src/com/android/dx/dex/file/DexFile.java.

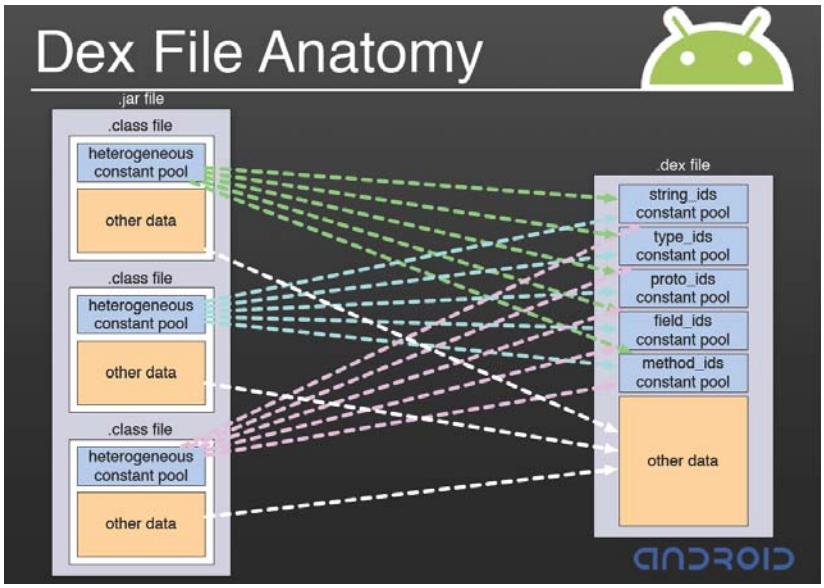
468. With reference to the discussion of how the dx tool uses TreeMap and by way of further explanation, the dx tool determines the duplicated elements (such as strings, field_ids, method_ids, and so forth) in a plurality of class files when it stores them in a TreeMap object, which determines whether the element is a duplicate of one already stored in the TreeMap. The TreeMap will not store more than one copy of a duplicated element. By way of example, the StringIdsSection.java code stores all the strings from the class files into a TreeMap object called "strings," thereby determining the duplicate strings among the class files:

```
35     * {@code non-null;} map from string constants to {@link
36     * StringIdItem} instances
37     */
38    private final TreeMap<CstUtf8, StringIdItem> strings;
...
137
138    public StringIdItem intern(StringIdItem string) {
139        if (string == null) {
140            throw new NullPointerException("string == null");
141        }
142
143        throwIfPrepared();
144
145        CstUtf8 value = string.getValue();
146        StringIdItem already = strings.get(value);
147
148        if (already != null) {
149            return already;
150        }
151
152        strings.put(value, string);
153        return string;
154    }
```

dalvik/dx/src/com/android/dx/dex/file/StringIdsSection.java.

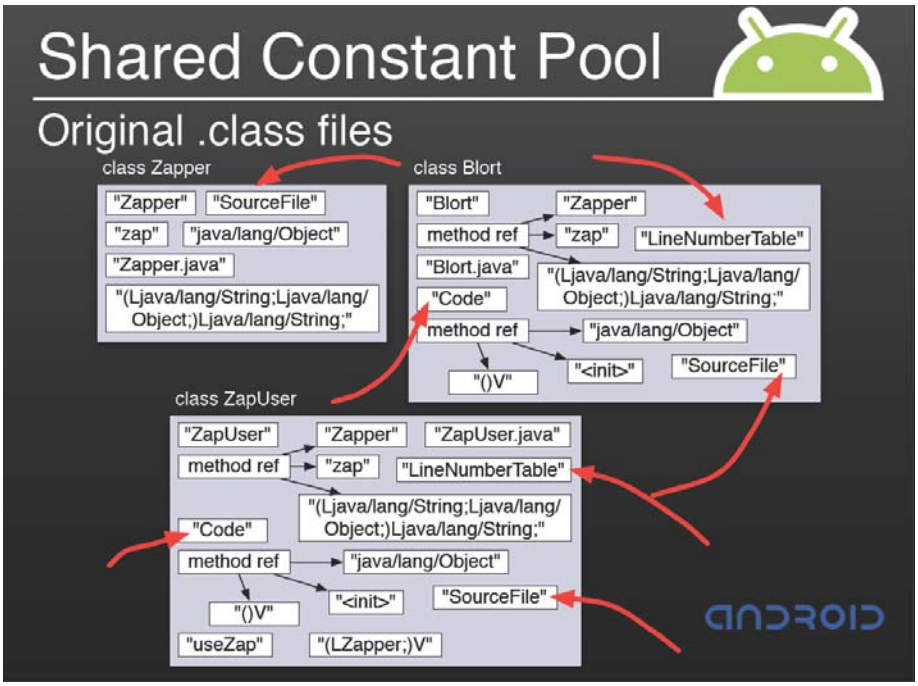
469. The **second element of claim 1**, recites "forming a shared table comprising said plurality of duplicated elements." The Android dx tool forms a shared table of the duplicated elements from the plurality of class files being preprocessed, in the manner discussed above and again here. This process is explained in the Dalvik Video at 7:20–9:25 and Dalvik Presentation, slides 15-20, where the recited shared table includes, *e.g.*, one or more of the "string_ids constant pool," "type_ids constant pool," "proto_ids constant pool," "field_ids constant pool," and "method_ids constant pool."

470. The Dalvik Presentation shows the elements of the class files combining into a shared constant pool (shared tables) in the .dex file.



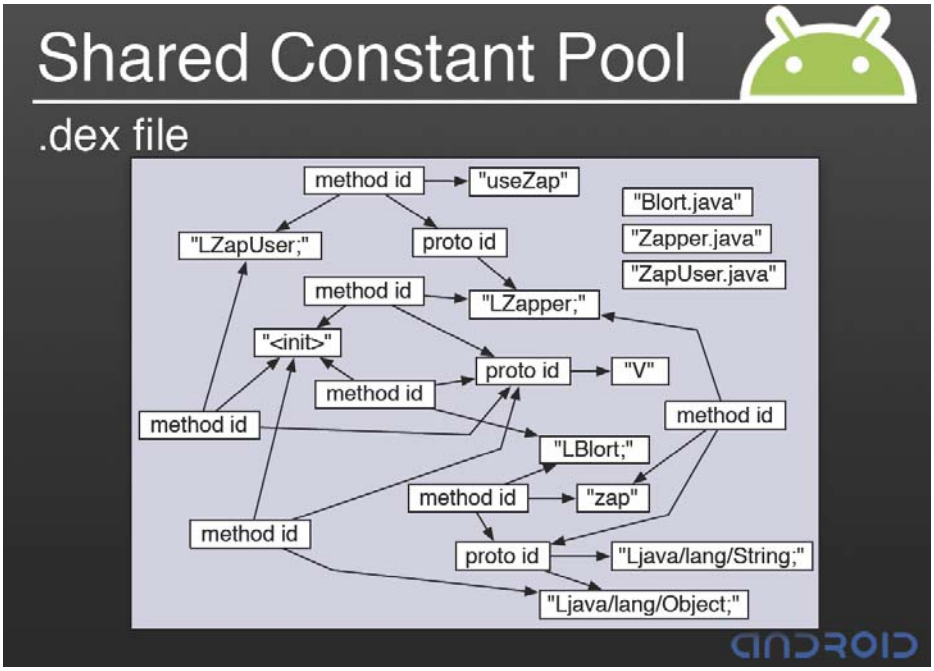
(Dalvik Presentation, Slide 15)

471. In the illustration above, each of "string_ids," "type_ids" and "method_ids" are examples of the shared tables (or, equivalently, a collective shared table). (*See also* 5/16/2011 Bornstein Dep. 91:6-21.) In addition, the discussion of the "Shared Constant Pool" in the Dalvik Video explains that the duplicated elements in the class files are consolidated into the shared constant pool (shared table) of the .dex file. (*See* Dalvik Presentation, Slides 15-21.) For example, slide 19 of the Dalvik Presentation shows the separate class files having duplicated elements.



(Dalvik Presentation, Slide 19)

472. Next, slide 20 of the Dalvik Presentation shows a representation of the class files after being processed into a single .dex file, with the duplicate elements removed; the elements are then stored in a shared constant pool (shared table):



(Dalvik Presentation, Slide 20)

The processes described above and recited by claim 1 can also be found directly in the Android source code. For example, they are described in:

> "Interfaces and implementation of things related to the constant pool.
> PACKAGES USED:
> * com.android.dx.rop.type
> * com.android.dx.util"

dalvik/dx/src/com/android/dx/rop/cst/package.html.

473. In the source code, the dx tool's execution begins at dx/src/com/android/dx/command/dexer/Main.java (http://android.git.kernel.org/?p=platform/dalvik.git;a=blob;f=dx/src/com/android/dx/command/dexer/Main.java). Android developers explain this code as "Main class for the class file translator," where I understand "class file translator" to refer to the dx tool. Through Main.java, the dx tool each input Java class is processed in turn

```
322     /**
323      * Processes one classfile.
324      *
325      * @param name {@code non-null;} name of the file, clipped such that it
326      * <i>should</i> correspond to the name of the class it contains
327      * @param bytes {@code non-null;} contents of the file
328      * @return whether processing was successful
329      */
330     private static boolean processClass(String name, byte[] bytes) {
331         if (! args.coreLibrary) {
332             checkClassName(name);
333         }
334
335         try {
336             ClassDefItem clazz =
337                 CfTranslator.translate(name, bytes, args.cfOptions);
338             outputDex.add(clazz);
339             return true;
340         } catch (ParseException ex) {
341             DxConsole.err.println("\ntrouble processing:");
342             if (args.debug) {
343                 ex.printStackTrace(DxConsole.err);
344             } else {
345                 ex.printContext(DxConsole.err);
346             }
347         }
348
349         warnings++;
350         return false;
351     }
```

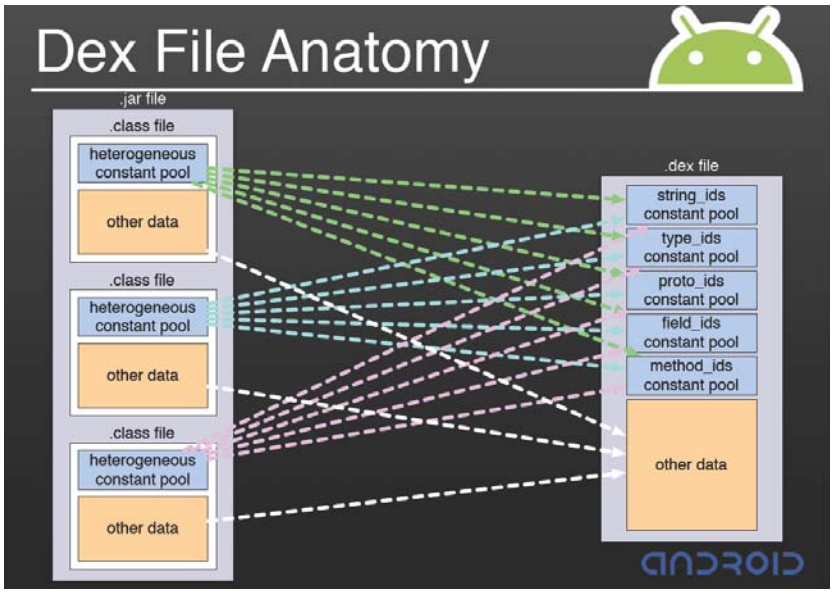
474. As described above, for each input Java class, the class file is processed and a ClassDefItem created as the internal representation of the class. The ClassDefItem includes all fields, methods and annotations of the class. Each method includes the Dalvik byte code for the

method, which is translated from the original Java byte code. When the ClassDefItem is completely formed, it is added to an object of class DexFile by calling DexFile.add(). DexFile has an instance variable of class ClassDefsSection that accumulates the ClassDefItem objects. ClassDefsSection checks that there are no duplicate class names, but does no other processing of the ClassDefItem objects at the time when they are added. After all of the input classes have been processed as above there is a pass that causes all of the constant pool entries in the ClassDefItem objects to be interned to a shared table. It is at this stage that the TreeMaps for the constant pool sections are populated and the duplicate removal takes place, as described above

475. Each of the TreeMaps used to store the elements for the "string_ids constant pool," "type_ids constant pool," "proto_ids constant pool," "field_ids constant pool," and "method_ids constant pool" are a shared table comprising the duplicated elements and become the shared tables comprising the duplicated elements that are ultimately stored in the .dex file, as recited in the claim language.

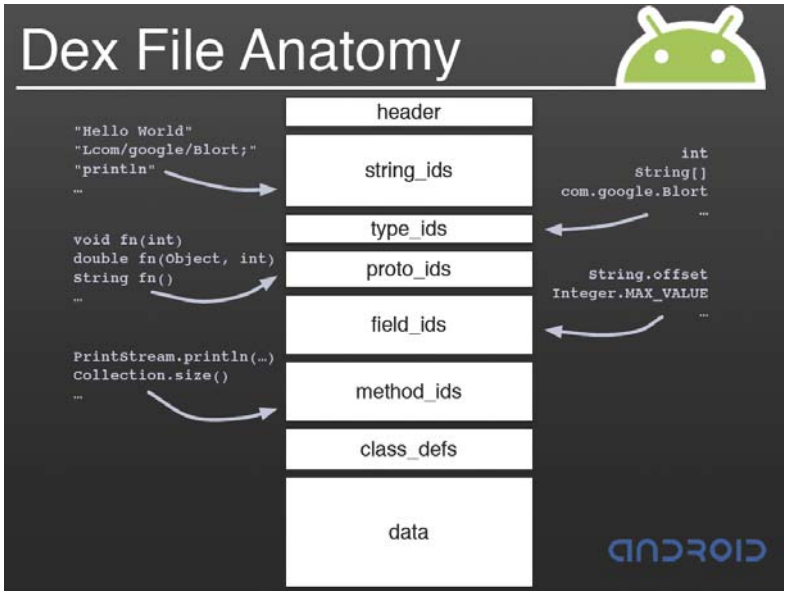
476. The **third element of claim 1**, recites "removing said duplicated elements from said plurality of class files to obtain a plurality of reduced class files." The Android dx tool removes the duplicated elements from the plurality of class files (*e.g.*, as part of the process of forming the .dex file multiple copies of the duplicated elements are not added to the tables that will be used to form the .dex file) and obtains a plurality of reduced class files in the form of the ClassDefSection (the ClassDefSection including a subset of the code and data contained in the class files) in the manner discussed above and again here. In particular, and as described above in the Android code, as class files are processed to form a .dex file, TreeMap is used to identify and remove duplicate elements from ClassDefSection (which generally includes the Dalvik byte code associated with the input class files after the identified duplicates have been removed). Note that removing or interning of the duplicated elements happens at the same time as forming the shared table. This process, and contents of the reduced class file, is clearly explained and illustrated in the Dalvik Video at 7:20–9:25 and Dalvik Presentation, Slides 15-20. The Dalvik Presentation shows the class files combining into a shared constant pool (shared table) in the .dex

file, whereby duplicated elements are removed from the class files when using a subset of the code and data contained in the class files, *i.e.*, the reduced class files, to form the .dex file.



(Dalvik Presentation, Slide 15)

477. The original class files are combined into a single .dex file, which includes a plurality of reduced class files (*i.e.*, a subset of code and data of the class files, with duplicates removed). This is also illustrated in slide 11 of the Dalvik presentation, which shows the anatomy of a .dex file:



(Dalvik Presentation, Slide 11)

478. The information in the ClassDefsSection and the data as shown in the slide form the reduced class files. Consider again the .class file format:

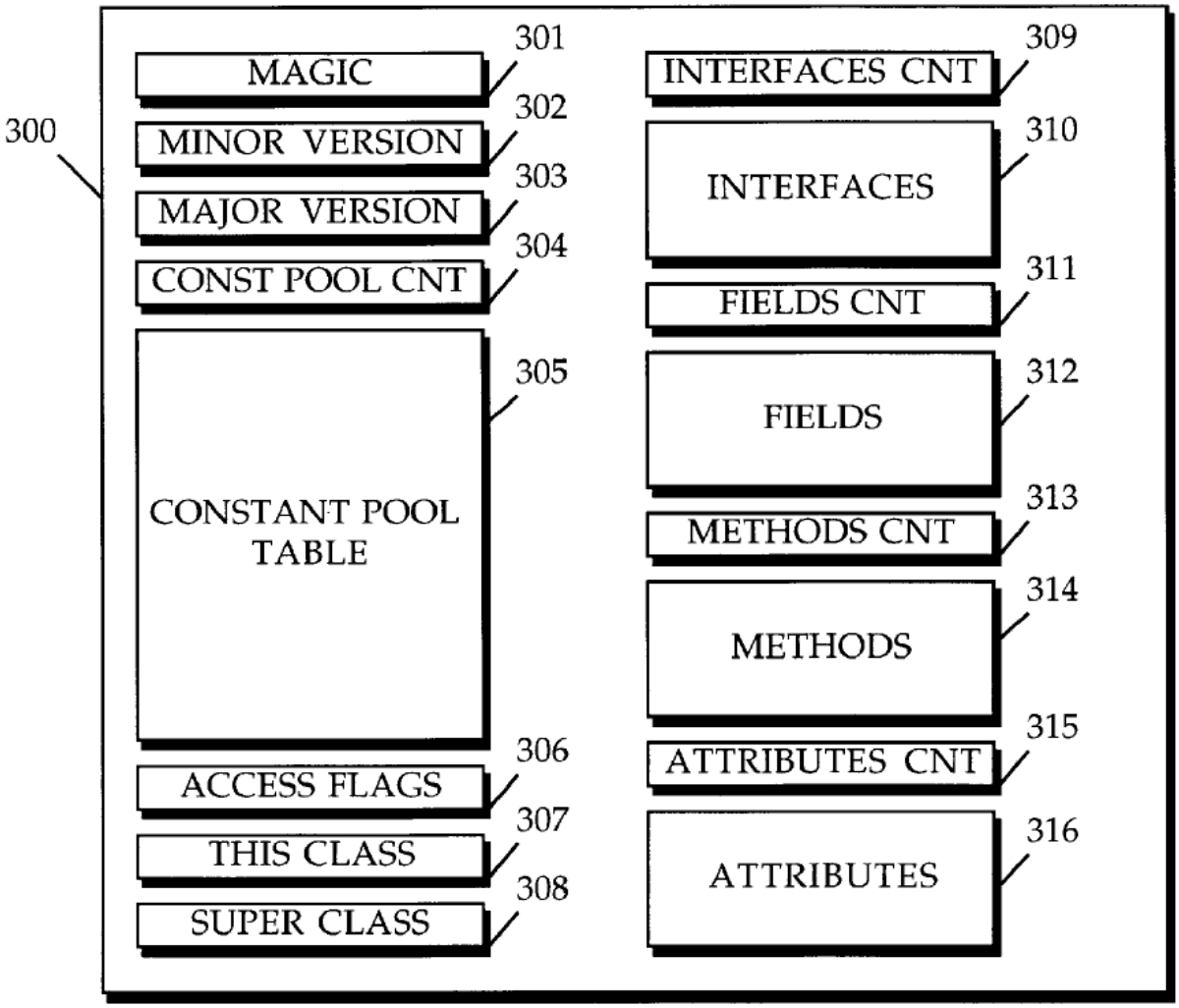


FIGURE 3

479. A .dex file's class_defs section contains information for each class, including the class (this class 307), access flags (306), superclass (308), an index to interface information (310), the source file name, and so on, as can be seen in the ClassDefItem.writeTo method. The .dex file's wordData, typeLists, stringData, byteData, classData sections also contain information for each class in a similar fashion. This is the information that remains after the constant pool

information, which contains one or more duplicated elements, have been removed and stored separately in the shared tables. Put another way, a .dex file comprises stringIds, typeIds, protoIds, fieldIds, methodIds, classDefs, wordData, typeLists, stringData, byteData, and classData sections. The information making up these sections come from the set of class files processed by the dx tool. The stringIds, typeIds, protoIds, fieldIds, methodIds sections are shared tables containing nonduplicated constant pool elements from the class files. The classDefs, wordData, typeLists, stringData, byteData, classData contain remaining information from the original class files (including Dalvik bytecode translated from Java bytecode).

480. The **fourth element of claim 1**, recites "forming a multi-class file comprising said plurality of reduced class files and said shared table." As explained above, the Android dx tool forms a multi-class file—the .dex file—comprising the reduced class files (*e.g.*, the classDefs section mentioned above, which contains a class definition for each class file processed by dx) and a shared constant pool (shared table) (*e.g.*, "string_ids constant pool," "type_ids constant pool," "proto_ids constant pool," "field_ids constant pool," and "method_ids constant pool.") from which duplicated elements have been removed. This process is explained in the Dalvik Video at 7:20–9:25 and Dalvik Presentation, Slides 11 and 15-20. The reduced class files include a subset of the code and data of the original class files, *e.g.*, "class_defs" and "data" illustrated in slide 11 and the "other data" illustrated in slide 15, and the recited shared table includes, e.g., one or more of the "string_ids constant pool," "type_ids constant pool," "proto_ids constant pool," "field_ids constant pool," and "method_ids constant pool."

481. The Dalvik Presentation shows the original class files being combined into a .dex file (multi-class file) comprising the plurality of reduced class files and the shared constant pool (shared table):

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