```
        if (c != null)
        {
             c.moveToFirst();

             while (c.isAfterLast() == false)
             {
                  tasks.add(c.getString(0));
                  c.moveToNext();
             }

             // TIP: Close resources (DONE)
             c.close();
        }

        return tasks;
    }
}
```

Tools

In this section we'll look at two types of tools useful in finding performance bottlenecks—tools that come with the Android SDK, and Unix command-line tools.

The Android SDK ships with the following tools to help us identify any performance issues:

- DDMS
- Traceview
- Lint
- Hierarchy Viewer
- Viewer

The Dalvik Debug Monitor Server (DDMS) is an Android SDK application that works as either a standalone tool or an Eclipse plugin. DDMS does lots of things, including device screen capture and providing a place to find logging output. But it also provides heap analysis, method allocation, and thread monitoring information. The Android SDK also has the Traceview tool for method profiling, `layoutopt` for optimizing your XML layouts, and Hierarchy Viewer for optimizing your UI.

And because Android is basically a Linux shell, we can leverage many of the following command-line Unix tools for performance testing:

- `Top`
- `Dumpsys`
- `Vmstat`
- `Procstats`

In this section we're going to look at how to use those tools to get a quick idea of where your application is spending most of its time.

DDMS

In this section we'll be covering the System Performance, Heap Usage, Threads, and Traceview tools, all of which come as part of DDMS. We'll also look at the Memory Analyzer Tool (MAT), which can be downloaded as part of the Eclipse tool and used to report on how memory is being managed in the Heap.

System Performance

The most basic tool in the DDMS suite is System Performance, which gives a quick snapshot overview of the current CPU load, memory usage, and frame render time, as shown in Figure 3-4. The first sign that you have an underperforming app is when your application is consuming too much CPU or memory.

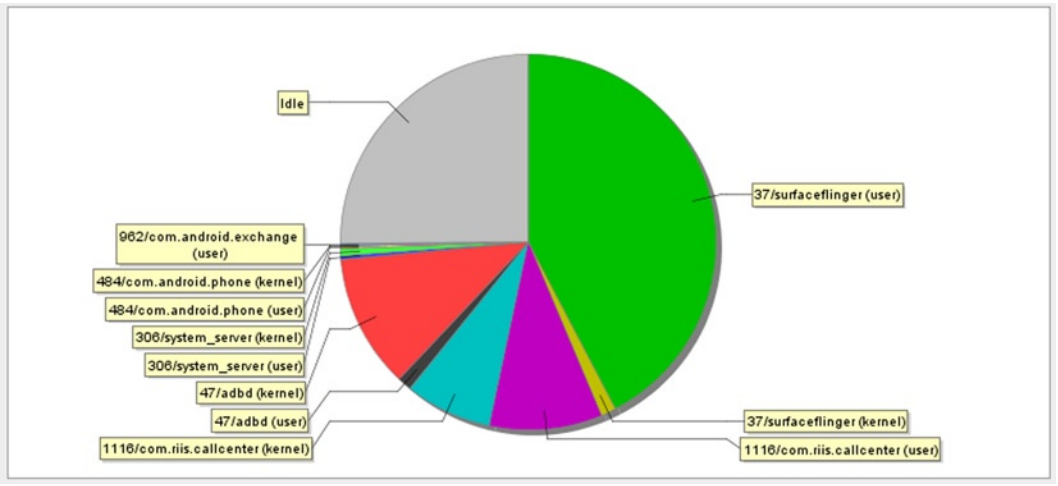

Figure 3-4. The System Performance tool displaying CPU load for CallCenterApp

Heap Usage

DDMS also offers a Heap Usage tool. Take the following steps to view the memory heap, where you can see what objects are being created and if they're being destroyed correctly by the garbage collection. (See Figure 3-5.)

1. In the Devices tab, select the process for which you want to view the heap.
2. Click the Update Heap button to enable heap information for the process.
3. Click Cause GC in the Heap tab to invoke garbage collection, which enables the collection of heap data.
4. When garbage collection completes, you will see a group of object types and the memory that has been allocated for each type.

5. Click an object type in the list to see a bar graph that shows the number of objects allocated for a particular memory size in bytes.

6. Click Cause GC again to refresh the data. Details of the heap are given along with a graph of allocation sizes for a particular allocation type. Watch the overall trend in Heap Size to make sure it doesn't keep growing during the application run.

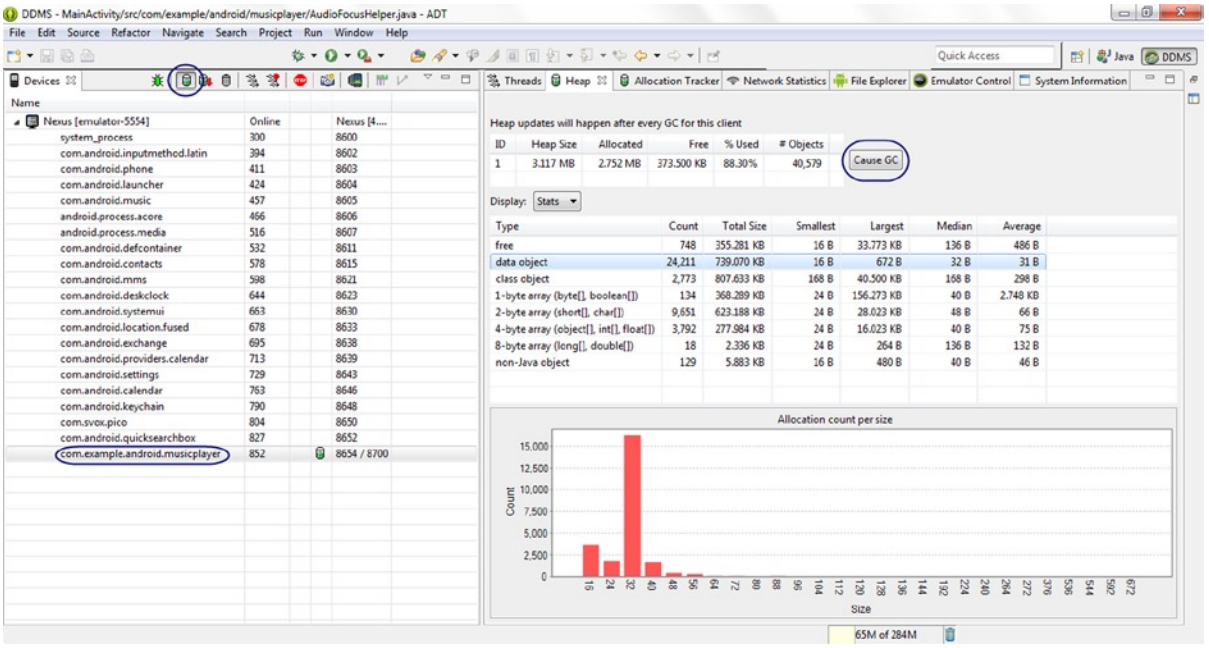

***Figure 3-5.** Viewing the DDMS heap*

Eclipse Memory Analyzer

Eclipse has an integrated Memory Analyzer Tool (MAT) plugin, which you can download and install from http://www.eclipse.org/mat/downloads.php. MAT can help you make sense of the heap output. Now when you dump the heap profile or hprof file (see Figure 3-6), it will be automatically analyzed so you can make some sense of the heap file.

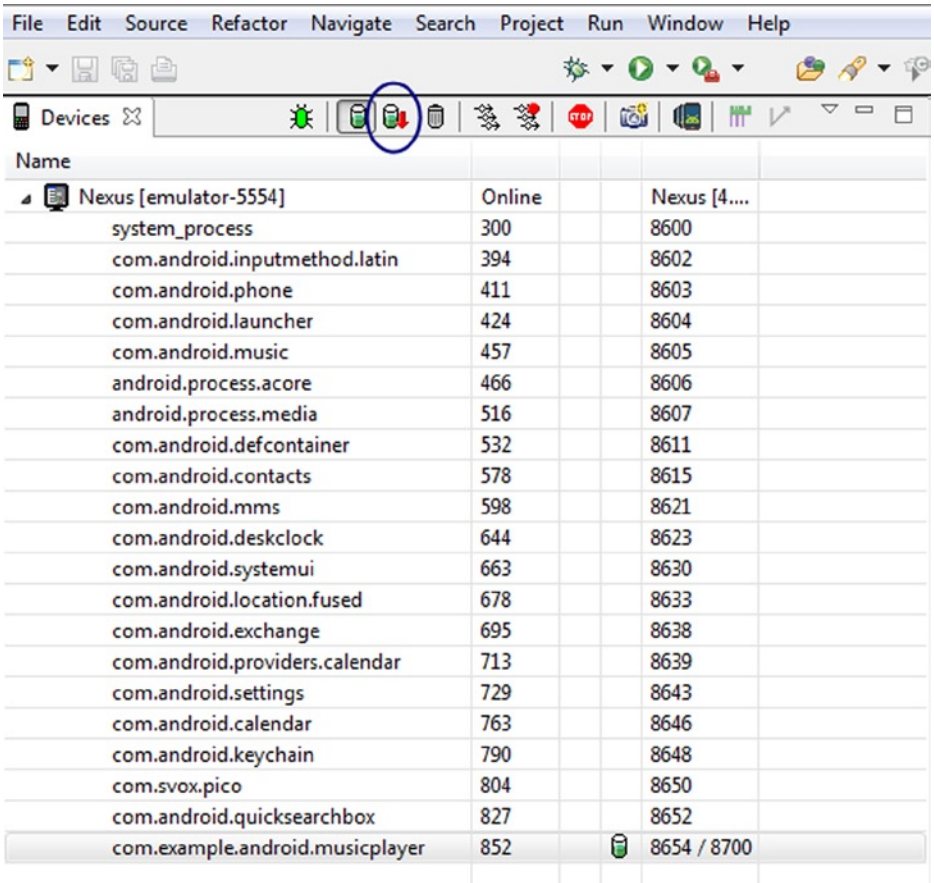

Figure 3-6. *Dumping the hprof file*

MAT provides a number of reports, including a Dominator Tree for the biggest class, a Top Consumers report, and a Leak Suspects report. Figure 3-7 shows Biggest Objects by Retained Size.

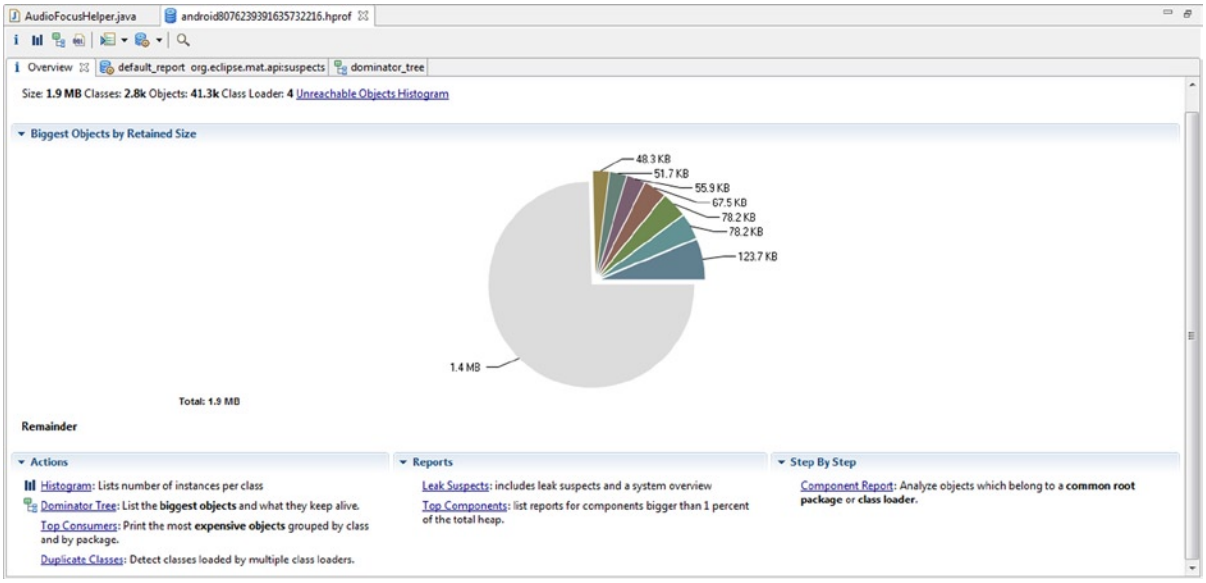

Figure 3-7. Memory Analyzer Tool overview

Memory Allocation

The next level of detail about allocations is shown in the Allocation Tracker view (Figure 3-8). To display it, click Start Tracking, perform an action in the application, and then click Get Allocations. The list presented is in allocation order, with the most recent memory allocation displayed first. Highlighting it will give you a stack trace showing how that allocation was created.

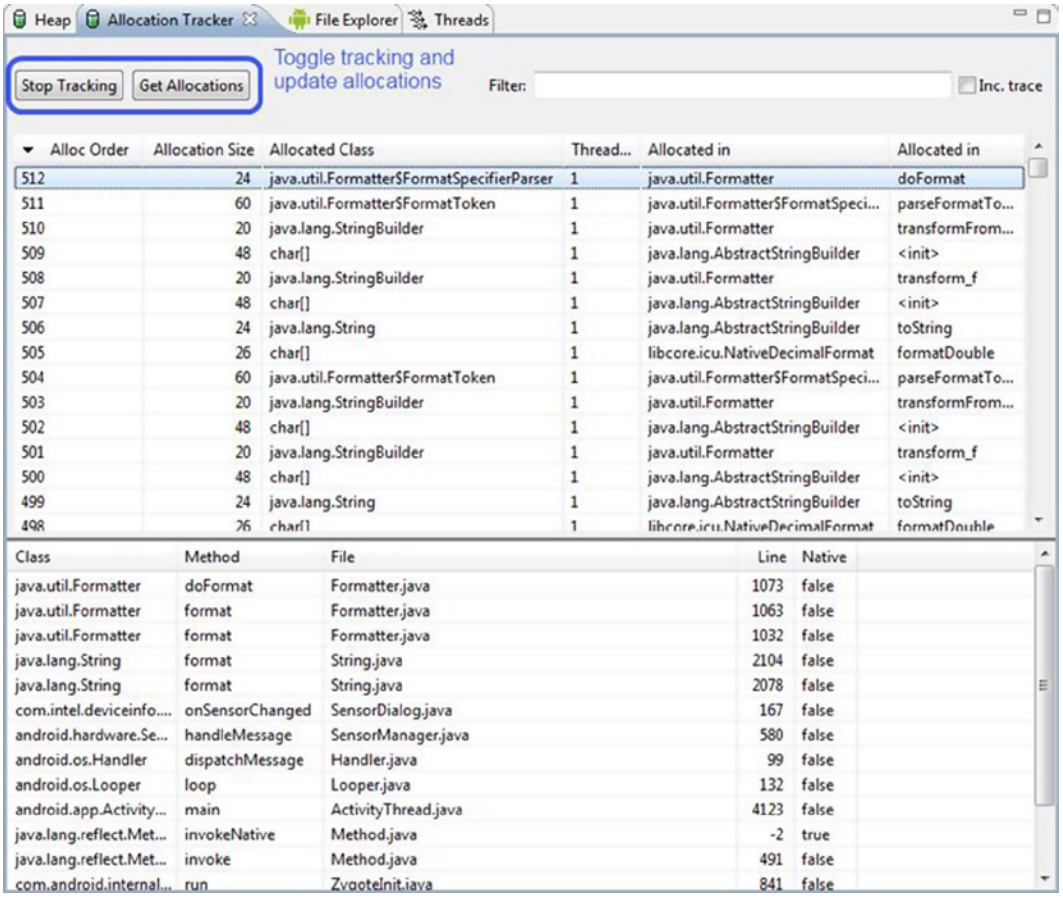

Figure 3-8. Allocation Tracker

Threads

The thread monitor and profiling view in DDMS is useful for applications that manage a lot of threads. To enable it, click the Update Threads icon, shown in Figure 3-9.

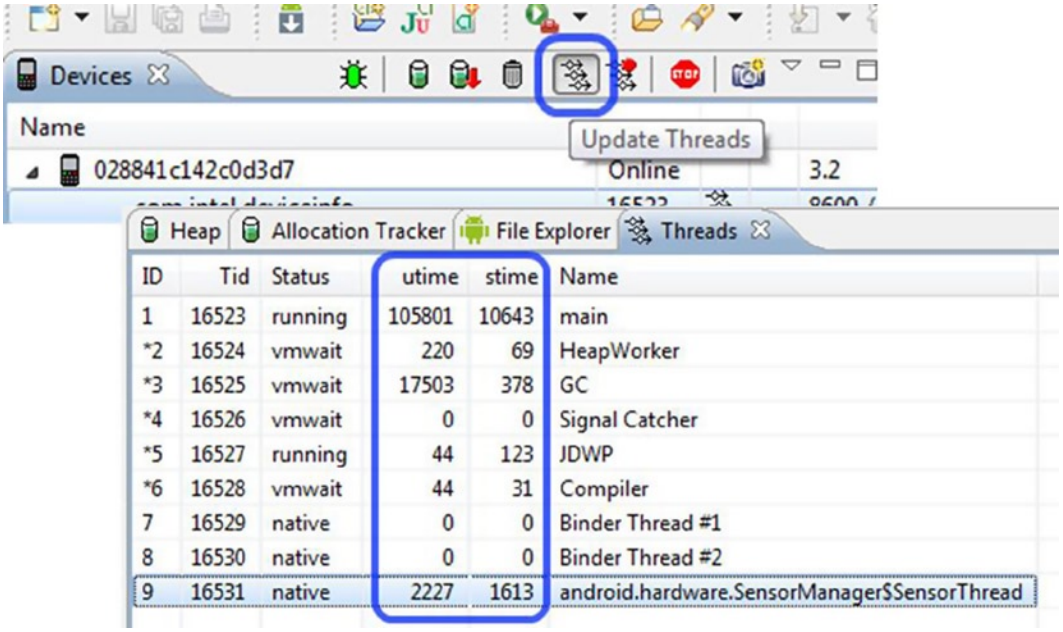

Figure 3-9. *DDMS threads*

The total time spent in a thread running user code (utime) and system code (stime) is measured in what are known as jiffies. A jiffy was originally the time it takes light to travel 1cm, but for Android devices it is the duration of one tick of the system timer interrupt. It varies from device to device but is generally accepted to be about 10ms. The asterisk indicates a daemon thread, and the status Native means the thread is executing native code.

Looking at the sample data in Figure 3-9, it is clear that an unusual amount of time is spent doing GC. A closer look at how the application is handling object creation might be a good idea for improving performance.

Method Profiling

Method Profiling is the tool of choice within DDMS for getting a quick overview of where time is really spent in your application and is the first step in homing in on methods that are taking too much time. With your application running and ideally performing some interesting task that you would like to get more performance data about, take the following steps to use Method Profiling:

1. Click on Start Method Profiling.
2. Click the icon again to stop collection after a couple of seconds.
3. The IDE will automatically launch the Traceview window and allow you to analyze the results from right within the IDE.
4. Click a method call in the bottom pane to create a hierarchy, showing you the current method, the parent(s) that call this method, and then the children methods called from within the selected method (Figure 3-10).

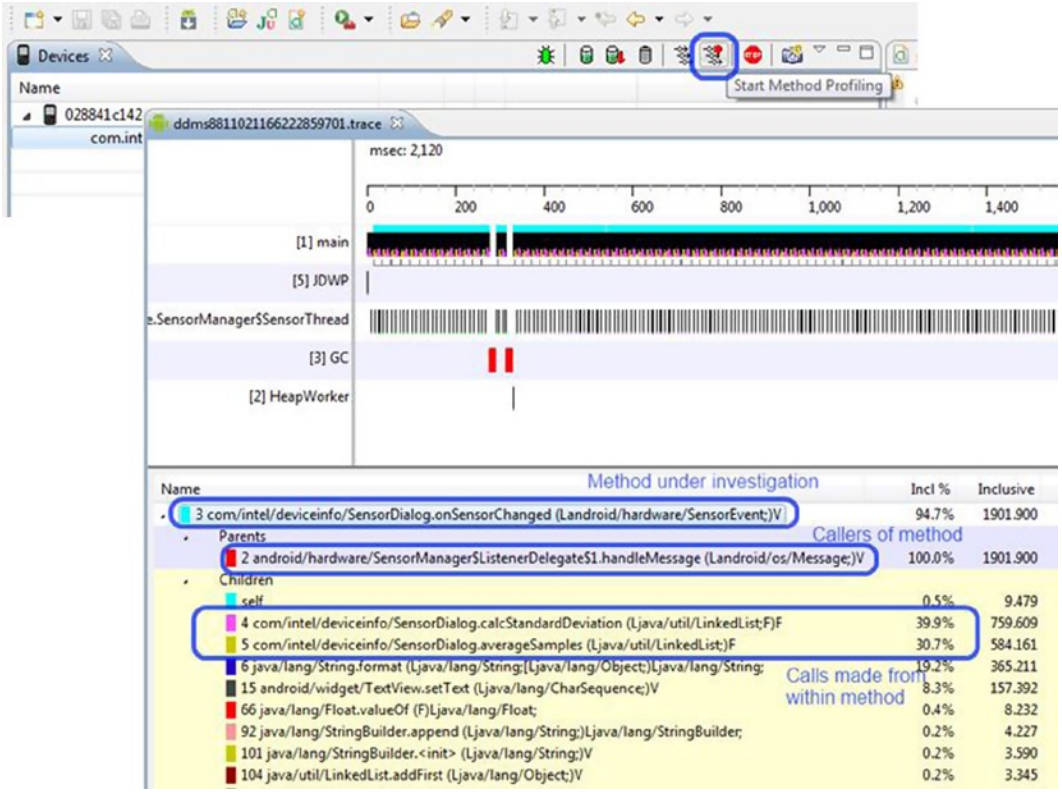

***Figure 3-10.** Method Profiling in DDMS using Traceview*

5. Identify the methods that are taking the most time so you can look at them closer by creating Traceview files, which we'll explore later in this section.

Each method has its parents and children, and the columns are as follows:

Inc % The percentage of the total time spent in the method plus any called methods

Inclusive The amount of time spent in the method plus the time spent in any called methods

Excl % The percentage of the total time spent in the method

Exclusive The amount of time spent in the method

Calls + Recursive The number of calls to this method plus any recursive calls

Time/Call The average amount of time per call

Traceview

Once you've identified the methods to take a closer look at, you can use the command-line version of Traceview with the tracing API for more accurate measurement. Add Debug.startMethodTracing, and Debug.stopMethodTracing around the code you want to profile, as shown in Listing 3-5. Compile your code again and push the APK to your device.

Listing 3-5. startMethodTracing and stopMethodTracing

```
public class ScoresActivity extends ListActivity {
        public void onStart() {
                // start tracing to "/sdcard/scores.trace"
                Debug.startMethodTracing("scores");
                super.onStart();
                // other start up code here
        }

        public void onStop() {
                super.onStop();
                // other shutdown code here
                Debug.stopMethodTracing();
        }

        // Other implementation code
}
```

The trace file can now be pulled off the device and displayed in Traceview using the following commands:

```
adb pull /sdcard/scores.trace scores.before.trace
```

Figure 3-11 shows the results before code optimization.

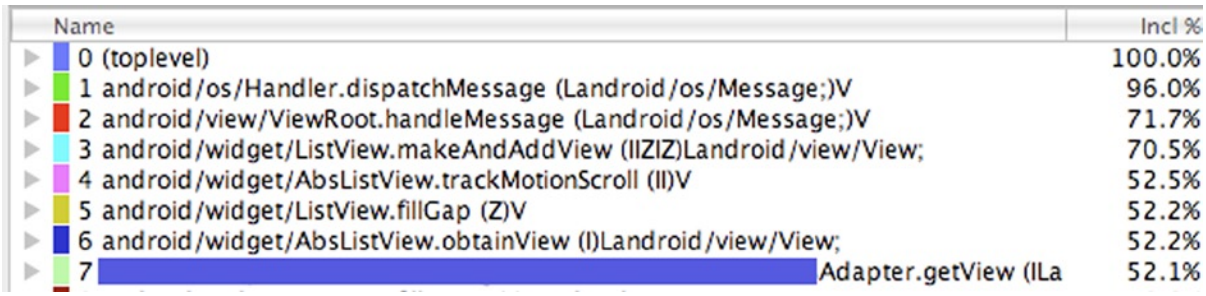

Figure 3-11. *The trace file before optimization*

Optimize the code using some of the suggestions earlier in the chapter and measure again, this time using the following command:

```
adb pull /sdcard/scores.trace scores.after.trace
```

Figure 3-12 shows the results after optimization; the difference is clear.

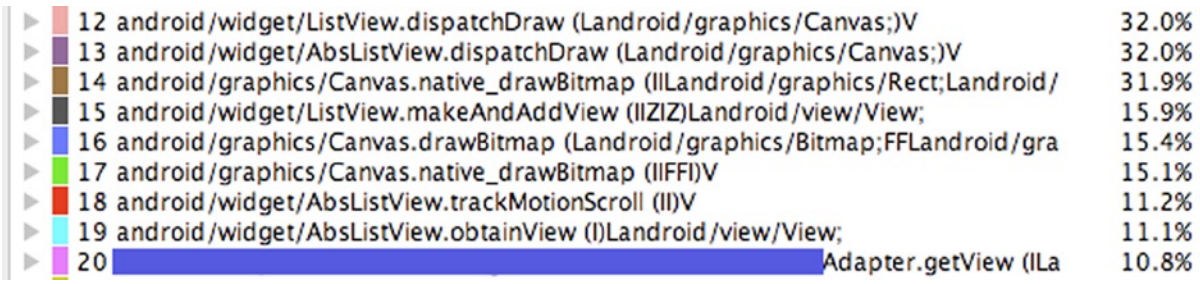

***Figure 3-12.** The trace file after optimization*

Lint

Lint is, like its original Unix namesake, a static code-analysis tool. It replaces the layoutopt tool, which was used to analyze your layout files and point out potential performance issues to get quick performance gains by reorganizing your UI layout. It now does so much more, including the following error-checking categories:

- Correctness
- Correctness:Messages
- Security
- Performance
- Usability:Typography
- Usability:Icons
- Usability
- Accessibility
- Internationalization

If you run the command `lint --list Performance` it will tell you that Lint does the following performance checks, many of which we've already seen in the Android Tips section:

`FloatMath`: Suggests replacing `android.util.FloatMath` calls with `java.lang.Math`.

`FieldGetter`: Suggests replacing use of getters with direct field access within a class.

`InefficientWeight`: Looks for inefficient weight declarations in `LinearLayouts`.

`NestedWeights`: Looks for nested layout weights, which are costly.

`DisableBaselineAlignment`: Looks for `LinearLayouts`, which should set `android:baselineAligned=false`.

`ObsoleteLayoutParam`: Looks for layout params that are not valid for the given parent layout.

`MergeRootFrame`: Checks whether a root *<FrameLayout>* can be replaced with a `<merge>` tag.

`UseCompoundDrawables`: Checks whether the current node can be replaced by a `TextView` using compound drawables.

`UselessParent`: Checks whether a parent layout can be removed.

`UselessLeaf`: Checks whether a leaf layout can be removed.

`TooManyViews`: Checks whether a layout has too many views.

`TooDeepLayout`: Checks whether a layout hierarchy is too deep.

`ViewTag`: Finds potential leaks when using `View.setTag`.

`HandlerLeak`: Ensures that Handler classes do not hold on to a reference to an outer class.

`UnusedResources`: Looks for unused resources.

`UnusedIds`: Looks for unused IDs.

`SecureRandom`: Looks for suspicious usage of the `SecureRandom` class.

`Overdraw`: Looks for overdraw issues (where a view is painted only to be fully painted over).

`UnusedNamespace`: Finds unused namespaces in XML documents.

`DrawAllocation`: Looks for memory allocations within drawing code.

`UseValueOf`: Looks for instances of "new" for wrapper classes, which should use `valueOf` instead.

`UseSparseArrays`: Looks for opportunities to replace `HashMaps` with the more efficient `SparseArray`.

`Wakelock`: Looks for problems with `wakelock` usage.

`Recycle`: Looks for missing `recycle()` calls on resources.

Lint can be run from within Eclipse or on the command line. If you just want to run the performance checks on your project, type `lint --check Performance` *<ProjectName>* on the command line. Listing 3-6 displays the output of this command for the sample application, showing that there are some layouts that need to be better organized.

Listing 3-6. Lint Performance output for the CallCenterApp project

```
Scanning CallCenterV3: ...............................................
Scanning CallCenterV3 (Phase 2): ......................
res\layout\custom_titlebar.xml:6: Warning: Possible overdraw: Root element paints background #004A82
with a theme that also paints a background (inferred theme is @style/CustomTheme) [Overdraw]
    android:background="#004A82"
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~

res\layout\custom_titlebar_with_logout.xml:6: Warning: Possible overdraw: Root element paints
background #004A82 with a theme that also paints a background (inferred theme is @style/CustomTheme)
[Overdraw]
    android:background="#004A82"
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~

res\layout\custom_titlebar_with_settings.xml:6: Warning: Possible overdraw: Root element paints
background #004A82 with a theme that also paints a background (inferred theme is @style/CustomTheme)
[Overdraw]
    android:background="#004A82"
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~

res\layout\login_screen.xml:5: Warning: Possible overdraw: Root element paints background
@drawable/bg_app with a theme that also paints a background (inferred theme is @style/CustomTheme)
[Overdraw]
    android:background="@drawable/bg_app"
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

res\layout\queues_screen.xml:5: Warning: Possible overdraw: Root element paints background
@drawable/bg_app with a theme that also paints a background (inferred theme is @style/CustomTheme)
[Overdraw]
    android:background="@drawable/bg_app"
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

res\layout\settings_screen.xml:5: Warning: Possible overdraw: Root element paints background #1D1D1D
with a theme that also paints a background (inferred theme is @style/CustomTheme) [Overdraw]
    android:background="#1D1D1D"
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~

res\drawable-hdpi\bg_login.9.png: Warning: The resource R.drawable.bg_login appears to be unused
[UnusedResources]
res\drawable-hdpi\btn_ok_xlarge.png: Warning: The resource R.drawable.btn_ok_xlarge appears to be
unused [UnusedResources]
res\drawable-hdpi\no_xlarge.png: Warning: The resource R.drawable.no_xlarge appears to be unused
[UnusedResources]
res\menu\settings_menu.xml: Warning: The resource R.menu.settings_menu appears to be unused
[UnusedResources]
res\values\strings.xml:7: Warning: The resource R.string.loginMessage appears to be unused
[UnusedResources]
    <string name="loginMessage">Enter Your Login Credentials</string>
            ~~~~~~~~~~~~~~~~~~~

res\values\strings.xml:8: Warning: The resource R.string.CSQ_default appears to be unused
[UnusedResources]
    <string name="CSQ_default">Log In</string>
            ~~~~~~~~~~~~~~~~~~

res\values\strings.xml:11: Warning: The resource R.string.default_time appears to be unused
[UnusedResources]
    <string name="default_time">00:00:00</string>
```

```
            ~~~~~~~~~~~~~~~~~~~
res\values\strings.xml:12: Warning: The resource R.string.oldest_in_queue appears to be unused
[UnusedResources]
    <string name="oldest_in_queue">Oldest Call In Queue: </string>
            ~~~~~~~~~~~~~~~~~~~~~~
res\values\strings.xml:16: Warning: The resource R.string.add_to_queue appears to be unused
[UnusedResources]
    <string name="add_to_queue">Add To Queue</string>
            ~~~~~~~~~~~~~~~~~~~
res\layout\login_screen.xml:9: Warning: This LinearLayout view is useless (no children, no
background, no id, no style) [UselessLeaf]
    <LinearLayout
    ^
res\layout\custom_titlebar.xml:10: Warning: This RelativeLayout layout or its LinearLayout parent is
useless; transfer the background attribute to the other view [UselessParent]
    <RelativeLayout
    ^
res\layout\custom_titlebar_with_logout.xml:10: Warning: This RelativeLayout layout or its
LinearLayout parent is useless; transfer the background attribute to the other view [UselessParent]
    <RelativeLayout
    ^
res\layout\custom_titlebar_with_settings.xml:10: Warning: This RelativeLayout layout or its
LinearLayout parent is useless; transfer the background attribute to the other view [UselessParent]
    <RelativeLayout
    ^
res\layout\queue_list_item.xml:13: Warning: This TableRow layout or its TableLayout parent is
possibly useless [UselessParent]
        <TableRow
        ^
res\layout\queue_list_item.xml:45: Warning: This TableRow layout or its TableLayout parent is
possibly useless [UselessParent]
        <TableRow
        ^
res\layout\custom_titlebar.xml:3: Warning: The resource R.id.photo_titlebar appears to be unused
[UnusedIds]
    android:id="@+id/photo_titlebar"
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
res\layout\queue_list_item.xml:7: Warning: The resource R.id.nameTable appears to be unused
[UnusedIds]
        android:id="@+id/nameTable"
        ~~~~~~~~~~~~~~~~~~~~~~~~~~
res\layout\queue_list_item.xml:14: Warning: The resource R.id.tableRow1 appears to be unused
[UnusedIds]
            android:id="@+id/tableRow1"
            ~~~~~~~~~~~~~~~~~~~~~~~~~~
res\layout\queue_list_item.xml:19: Warning: The resource R.id.activeIndicatorDummy appears to be
unused [UnusedIds]
                android:id="@+id/activeIndicatorDummy"
                ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
res\layout\queue_list_item.xml:46: Warning: The resource R.id.tableRow2 appears to be unused
[UnusedIds]
            android:id="@+id/tableRow2"
```

```
            ~~~~~~~~~~~~~~~~~~~~~~~~~~~
res\layout\queue_list_item.xml:62: Warning: The resource R.id.callsInQueueLabel appears to be unused
[UnusedIds]
                android:id="@+id/callsInQueueLabel"
                ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
0 errors, 27 warnings

res\layout\queue_list_item.xml:7: Warning: The resource R.id.nameTable appears to be unused [UnusedIds]
        android:id="@+id/nameTable"
        ~~~~~~~~~~~~~~~~~~~~~~~~~~~
res\layout\queue_list_item.xml:14: Warning: The resource R.id.tableRow1 appears to be unused
[UnusedIds]
            android:id="@+id/tableRow1"
            ~~~~~~~~~~~~~~~~~~~~~~~~~~~
res\layout\queue_list_item.xml:19: Warning: The resource R.id.activeIndicatorDummy appears to be
unused [UnusedIds]
                android:id="@+id/activeIndicatorDummy"
                ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
res\layout\queue_list_item.xml:46: Warning: The resource R.id.tableRow2 appears to be unused
[UnusedIds]
            android:id="@+id/tableRow2"
            ~~~~~~~~~~~~~~~~~~~~~~~~~~~
res\layout\queue_list_item.xml:62: Warning: The resource R.id.callsInQueueLabel appears to be unused
[UnusedIds]
                android:id="@+id/callsInQueueLabel"
                ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
0 errors, 27 warnings
```

Hierarchy Viewer

Another useful tool in debugging performance issues, specifically for layouts, is the Hierarchy Viewer. At its most basic it will show you how long it takes to inflate the layouts. You start Hierarchy Viewer from within Eclipse by adding the perspective; this is similar to the way you would add back DDMS if it ever disappeared.

Hierarchy Viewer first displays a list of devices and emulators; click the name of your app from the list and then click Load View Hierarchy. The Tree View, the Tree Overview, and the Tree Layout will then open, as shown in Figure 3-13. The Tree View shows all the layouts that you defined in your XML files. We talked earlier in this chapter about how nested layouts can be bad for performance, and Tree Overview is a great way to see just how nested your layouts have become and figure out if it's time to merge them into a `RelativeLayout`. Tree View shows how long each layout took to display, so you can identify which views you need to debug and optimize to speed up your UI.

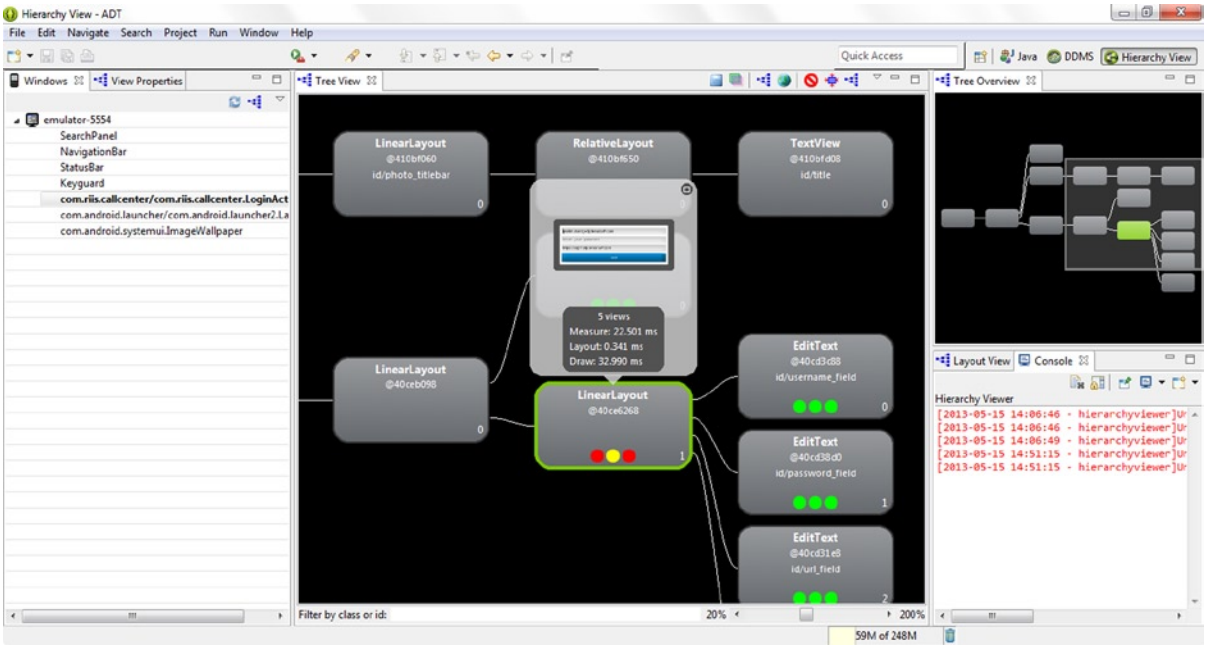

Figure 3-13. *Hierarchy Viewer for CallCenterApp login screen*

In Figure 3-13 we can see that our login view took almost 33ms to display. It also shows what layouts are part of the login view, and by hovering over specific views you can see just how long each took to display.

Hierarchy Viewer also includes a Pixel Perfect tool for designers. We won't be covering that in this book.

Unix Tools

Because Android is built on Linux, we can leverage many of the same shell command tools as Linux for performance testing. The main tools focus on total process load, individual process details, and memory utilization.

Top

The top command will give you an idea of where your app is in relation to all other processes on the device. The higher up the list, the more resources it is consuming. You can log onto the phone using the adb shell command, or you can run the command remotely using adb shell top from your command line. Figure 3-14 shows the results.

```
User 28%, System 12%, IOW 3%, IRQ 0%
User 69 + Nice 22 + Sys 40 + Idle 173 + IOW 12 + IRQ 0 + SIRQ 0 = 316

  PID PR CPU% S  #THR     VSS     RSS PCY UID      Name
  270  0   6% S   96 416256K  62584K  fg system   system_server
 1599  0   6% S    9 293056K  37488K  fg app_110  com.riis.callcenter
  129  0   5% S    9  57644K  10552K  fg system   /system/bin/surfaceflinger
 1586  0   3% R    1   1104K    472K  fg shell    top
31563  0   1% S    1      0K      0K  fg root     kworker/u:0
```

***Figure 3-14.** Output from the* `top` *command*

Dumpsys

Top also gets you the process ID or PID of your application, which you can then use for the dumpsys command, as follows:

```
adb shell dumpsys meminfo 1599
```

Dumpsys will give you information about the memory and heap being used by your application; see Figure 3-15.

```
Applications Memory Usage (kB):
Uptime: 126416263 Realtime: 126416238

** MEMINFO in pid 1599 [com.riis.callcenter] **
                   Pss  Shared  Private     Heap     Heap     Heap
                         Dirty    Dirty     Size    Alloc     Free
                ------  ------  ------   ------   ------   ------
      Native      1113    1540    1076     4232     4168       63
      Dalvik      5794   13084    5544    16035    14912     1123
      Cursor         0       0       0
      Ashmem         0       0       0
   Other dev         4      40       0
    .so mmap       413    2176     280
    .jar mmap        4       0       0
    .apk mmap       55       0       0
    .ttf mmap        3       0       0
    .dex mmap      276       0       0
   Other mmap      543      16      28
      Unknown      640     348     636
        TOTAL     8845   17204    7564    20267    19080     1186

 Objects
               Views:       62         ViewRootImpl:        3
         AppContexts:        2           Activities:        1
              Assets:        3        AssetManagers:        3
        Local Binders:        7        Proxy Binders:       13
     Death Recipients:        1
      OpenSSL Sockets:        0

 SQL
                heap:        0          MEMORY_USED:        0
  PAGECACHE_OVERFLOW:        0          MALLOC_SIZE:        0
```

***Figure 3-15.** Dumpsys Meminfo*

All of the Unix tools mentioned in this section are taking measurements at a point in time. `Procstats` was introduced in Android 4.4 or KitKat to show how much memory and CPU the apps running in the background will consume. Use the command to see the `procstats` output:

```
adb shell dumpsys procstats
```

with the results shown in Figure 3-16.

```
            Home: 0.00%
       (Cached): 67% (54MB-56MB-67MB/44MB-47MB-62MB over 32)
* com.google.android.gms / u0a7:
          TOTAL: 28% (10MB-11MB-11MB/8.0MB-8.1MB-8.2MB over 16)
            Top: 27% (10MB-11MB-11MB/8.0MB-8.1MB-8.2MB over 16)
         Imp Fg: 1.2%
        Service: 0.14%
       Receiver: 0.01%
            Home: 0.00%
    (Last Act): 0.09%
      (Cached): 70% (10MB-11MB-11MB/8.0MB-8.1MB-8.2MB over 15)
* com.google.android.apps.books / u0a26:
          TOTAL: 22% (50MB-80MB-113MB/43MB-63MB-82MB over 12)
            Top: 22% (50MB-80MB-113MB/43MB-63MB-82MB over 12)
        Service: 0.00%
    (Last Act): 4.6% (85MB-85MB-85MB/77MB-77MB-77MB over 2)
      (Cached): 58% (78MB-81MB-82MB/70MB-73MB-74MB over 4)
* com.android.settings / 1000:
          TOTAL: 2.8% (14MB-15MB-16MB/11MB-12MB-13MB over 3)
            Top: 2.8% (14MB-15MB-16MB/11MB-12MB-13MB over 3)
```

***Figure 3-16.** Dumpsys Procstats*

Vmstat

Vmstat allows you to view virtual memory levels on the device; see Figure 3-17. It is a simple Linux command that reports about processes, memory, paging, block IO, traps, and CPU activity. The “b” column shows which processes are blocked. Use the command as follows: adb shell vmstat.