Low RAM Configuration

Introduction

Android now supports devices with 512MB of RAM. This documentation is intended to help OEMs optimize and configure Android 4.4 for low-memory devices. Several of these optimizations are generic enough that they can be applied to previous releases as well.

Android 4.4 platform optimizations

Improved memory management

- Validated memory-saving kernel configurations: Kernel Same-page Merging (KSM), and Swap to ZRAM.
- Kill cached processes if about to be uncached and too large.
- Don't allow large services to put themselves back into A Services (so they can't cause the launcher to be killed).
- Kill processes (even ordinarily unkillable ones such as the current IME) that get too large in idle maintenance.
- Serialize the launch of background services.
- Tuned memory use of low-RAM devices: tighter out-of-memory (OOM) adjustment levels, smaller graphics caches, etc.

Reduced system memory

- Trimmed system_server and SystemUI processes (saved several MBs).
- Preload dex caches in Dalvik (saved several MBs).
- Validated JIT-off option (saves up to 1.5MB per process).
- Reduced per-process font cache overhead.
- Introduced ArrayMap/ArraySet and used extensively in framework as a lighter-footprint replacement for HashMap/HashSet.

Procstats

Added a new Developer Option to show memory state and application memory usage ranked by how often they run and amount of memory consumed.

API

Added a new ActivityManager.isLowRamDevice() to allow applications to detect when running on low memory devices and choose to disable large-RAM features.

Memory tracking

New memtrack HAL to track graphics memory allocations, additional information in dumpsys meminfo, clarified summaries in meminfo (for example reported free RAM includes RAM of cached processes, so that OEMs don't try to optimize the wrong thing).

Build-time configuration

Enable Low Ram Device flag

We are introducing a new API called **`ActivityManager.isLowRamDevice()`** for applications to determine if they should turn off specific memory-intensive features that work poorly on low-memory devices.

For 512MB devices, this API is expected to return **`true`**. It can be enabled by the following system property in the device makefile.

```
PRODUCT_PROPERTY_OVERRIDES += ro.config.low_ram=true
```

Disable JIT

System-wide JIT memory usage is dependent on the number of applications running and the code footprint of those applications. The JIT establishes a maximum translated code cache size and touches the pages within it as needed. JIT costs somewhere between 3M and 6M across a typical running system.

The large apps tend to max out the code cache fairly quickly (which by default has been 1M). On average, JIT cache usage runs somewhere between 100K and 200K bytes per app. Reducing the max size of the cache can help somewhat with memory usage, but if set too low will send the JIT into a thrashing mode. For the really low-memory devices, we recommend the JIT be disabled entirely.

This can be achieved by adding the following line to the product makefile:

```
PRODUCT_PROPERTY_OVERRIDES += dalvik.vm.jit.codecachesize=0
```

Launcher Configs

Ensure the default wallpaper setup on launcher is **not** using live-wallpaper. Low-memory devices should not pre-install any live wallpapers.

Kernel configuration

Tuning kernel/ActivityManager to reduce direct reclaim

Direct reclaim happens when a process or the kernel tries to allocate a page of memory (either directly or due to faulting in a new page) and the kernel has used all available free memory. This requires the kernel to block the allocation while it frees up a page. This in turn often requires disk I/O to flush out a dirty file-backed page or waiting for **`lowmemorykiller`** to kill a process. This can result in extra I/O in any thread, including a UI thread.

To avoid direct reclaim, the kernel has watermarks that trigger **`kswapd`** or background reclaim. This is a thread that tries to free up pages so the next time a real thread allocates it can succeed quickly.

The default threshold to trigger background reclaim is fairly low, around 2MB on a 2GB device and 636KB on a 512MB device. And the kernel reclaims only a few MB of memory in background reclaim. This means any process that quickly allocates more than a few megabytes is going to quickly hit direct reclaim.

Support for a new kernel tunable is added in the android-3.4 kernel branch as patch 92189d47f66c67e5fd92eafaa287e153197a454f ("add extra free kbytes tunable"). Cherry-picking this patch to a device's kernel will allow ActivityManager to tell the kernel to try to keep 3 full-screen 32 bpp buffers of memory free.

These thresholds can be configured via the framework config.xml

```
<!-- Device configuration setting the /proc/sys/vm/extra_free_kbytes tunable
in the kernel (if it exists).  A high value will increase the amount of memory
that the kernel tries to keep free, reducing allocation time and causing the
lowmemorykiller to kill earlier.  A low value allows more memory to be used by
processes but may cause more allocations to block waiting on disk I/O or
lowmemorykiller.  Overrides the default value chosen by ActivityManager based
on screen size.  0 prevents keeping any extra memory over what the kernel keeps
by default.  -1 keeps the default. -->
<integer name="config_extraFreeKbytesAbsolute">-1</integer>


<!-- Device configuration adjusting the /proc/sys/vm/extra_free_kbytes
tunable in the kernel (if it exists).  0 uses the default value chosen by
ActivityManager.  A positive value  will increase the amount of memory that the
kernel tries to keep free, reducing allocation time and causing the
lowmemorykiller to kill earlier.  A negative value allows more memory to be
used by processes but may cause more allocations to block waiting on disk I/O
or lowmemorykiller.  Directly added to the default value chosen by
ActivityManager based on screen size. -->
<integer name="config_extraFreeKbytesAdjust">0</integer>
```

Tuning LowMemoryKiller

ActivityManager configures the thresholds of the LowMemoryKiller to match its expectation of the working set of file-backed pages (cached pages) required to run the processes in each priority level bucket. If a device has high requirements for the working set, for example if the vendor UI requires more memory or if more services have been added, the thresholds can be increased.

The thresholds can be reduced if too much memory is being reserved for file backed pages, so that background processes are being killed long before disk thrashing would occur due to the cache getting too small.

```
<!-- Device configuration setting the minfree tunable in the lowmemorykiller
in the kernel.  A high value will cause the lowmemorykiller to fire earlier,
keeping more memory in the file cache and preventing I/O thrashing, but
allowing fewer processes to stay in memory.  A low value will keep more
processes in memory but may cause thrashing if set too low.  Overrides the
default value chosen by ActivityManager based on screen size and total memory
for the largest lowmemorykiller bucket, and scaled proportionally to the
smaller buckets.  -1 keeps the default. -->
<integer name="config_lowMemoryKillerMinFreeKbytesAbsolute">-1</integer>


<!-- Device configuration adjusting the minfree tunable in the
lowmemorykiller in the kernel.  A high value will cause the lowmemorykiller to
fire earlier, keeping more memory in the file cache and preventing I/O
thrashing, but allowing fewer processes to stay in memory.  A low value will
keep more processes in memory but may cause thrashing if set too low.  Directly
added to the default value chosen by          ActivityManager based on screen
size and total memory for the largest lowmemorykiller bucket, and scaled
proportionally to the smaller buckets. 0 keeps the default. -->
<integer name="config_lowMemoryKillerMinFreeKbytesAdjust">0</integer>
```

KSM (Kernel samepage merging)

KSM is a kernel thread that runs in the background and compares pages in memory that have been marked `MADV_MERGEABLE` by user-space. If two pages are found to be the same, the KSM thread merges them back as a single copy-on-write page of memory.

KSM will save memory over time on a running system, gaining memory duplication at a cost of CPU power, which could have an impact on battery life. You should measure whether the power tradeoff is worth the memory savings you get by enabling KSM.

To test KSM, we recommend looking at long running devices (several hours) and seeing whether KSM makes any noticeable improvement on launch times and rendering times.

To enable KSM, enable `CONFIG_KSM` in the kernel and then add the following lines to your` `init.<device>.rc` file:

```
write /sys/kernel/mm/ksm/pages_to_scan 100
write /sys/kernel/mm/ksm/sleep_millisecs 500
write /sys/kernel/mm/ksm/run 1
```

Once enabled, there are few utilities that will help in the debugging namely : procrank, librank, & ksminfo. These utilities allow you to see which KSM memory is mapped to what process, which processes use the most KSM memory. Once you have found a chunk of memory that looks worth exploring you can use the hat utility if it's a duplicate object on the dalvik heap.

Swap to zRAM

zRAM swap can increase the amount of memory available in the system by compressing memory pages and putting them in a dynamically allocated swap area of memory.

Again, since this is trading off CPU time for a small increase in memory, you should be careful about measuring the performance impact zRAM swap has on your system.

Android handles swap to zRAM at several levels:

- First, the following kernel options must be enabled to use zRAM swap effectively:
  - `CONFIG_SWAP`
  - `CONFIG_CGROUP_MEM_RES_CTLR`
  - `CONFIG_CGROUP_MEM_RES_CTLR_SWAP`
  - `CONFIG_ZRAM`

- Then, you should add a line that looks like this to your fstab:

  ```
  /dev/block/zram0 none swap defaults zramsize=<size in bytes>,swapprio=<swap partition priority>
  ```

  - **`zramsize`** is mandatory and indicates how much uncompressed memory you want the zram area to hold. Compression ratios in the 30-50% range are usually observed.
  - **`swapprio`** is optional and not needed if you don't have more than one swap area.

  You should also be sure to label the associated block device as a swap_block_device in the device-specific sepolicy/file_contexts (https://source.android.com/security/selinux/implement.html) so that it is treated properly by SELinux.

  ```
  /dev/block/zram0 u:object_r:swap_block_device:s0
  ```

- By default, the Linux kernel swaps in 8 pages of memory at a time. When using ZRAM, the incremental cost of reading 1 page at a time is negligible and may help in case the device is under extreme memory pressure. To read only 1 page at a time, add the following to your **`init.rc`**:

  ```
  write /proc/sys/vm/page-cluster 0
  ```

- In your **`init.rc`** after the **`mount_all /fstab.X`** line, add:

  ```
  swapon_all /fstab.X
  ```

- The memory cgroups are automatically configured at boot time if the feature is enabled in kernel.
- If memory cgroups are available, the ActivityManager will mark lower priority threads as being more swappable than other threads. If memory is needed, the Android kernel will start migrating memory pages to zRAM swap, giving a higher priority to those memory pages that have been marked by ActivityManager.

Carveouts, Ion and Contiguous Memory Allocation (CMA)

It is especially important on low memory devices to be mindful about carveouts, especially those that will not always be fully utilized -- for example a carveout for secure video playback. There are several solutions to minimizing the impact of your carveout regions that depend on the exact requirements of your hardware.

If hardware permits discontiguous memory allocations, the ion system heap allows memory allocations from system memory, eliminating the need for a carveout. It also attempts to make large allocations to eliminate TLB pressure on peripherals. If memory regions must be contiguous or confined to a specific address range, the contiguous memory allocator (CMA) can be used.

This creates a carveout that the system can also use of for movable pages. When the region is needed, movable pages will be migrated out of it, allowing the system to use a large carveout for other purposes when it is free. CMA can be used directly or more simply via ion by using the ion cma heap.

Application optimization tips

- Review Managing your App's Memory (http://developer.android.com/training/articles/memory.html) and these past blog posts on the same topic:
  - http://android-developers.blogspot.com/2009/01/avoiding-memory-leaks.html (http://android-developers.blogspot.com/2009/01/avoiding-memory-leaks.html)
  - http://android-developers.blogspot.com/2011/03/memory-analysis-for-android.html (http://android-developers.blogspot.com/2011/03/memory-analysis-for-android.html)
  - http://android-developers.blogspot.com/2009/02/track-memory-allocations.html (http://android-developers.blogspot.com/2009/02/track-memory-allocations.html)
  - http://tools.android.com/recent/lintperformancechecks (http://tools.android.com/recent/lintperformancechecks)
- Check/remove any unused assets from preinstalled apps - development/tools/findunused (should help make the app smaller).
- Use PNG format for assets, especially when they have transparent areas
- If writing native code, use calloc() rather than malloc/memset
- Don't enable code that is writing Parcel data to disk and reading it later.

- Don't subscribe to every package installed, instead use ssp filtering. Add filtering like below:

```
<data android:scheme="package" android:ssp="com.android.pkg1" />
<data android:scheme="package" android:ssp="com.myapp.act1" />
```

Understand the various process states in Android

- SERVICE - SERVICE_RESTARTING
  Applications that are making themselves run in the background for their own reason. Most common problem apps have when they run in the background too much. %duration * pss is probably a good "badness" metric, although this set is so focused that just doing %duration is probably better to focus on the fact that we just don't want them running at all.
- IMPORTANT_FOREGROUND - RECEIVER
  Applications running in the background (not directly interacting with the user) for any reason. These all add memory load to the system. In this case the (%duration * pss) badness value is probably the best ordering of such processes, because many of these will be always running for good reason, and their pss size then is very important as part of their memory load.
- PERSISTENT
  Persistent system processes. Track pss to watch for these processes getting too large.
- TOP
  Process the user is currently interacting with. Again, pss is the important metric here, showing how much memory load the app is creating while in use.
- HOME - CACHED_EMPTY
  All of these processes at the bottom are ones that the system is keeping around in case they are needed again; but they can be freely killed at any time and re-created if needed. These are the basis for how we compute the memory state -- normal, moderate, low, critical is based on how many of these processes the system can keep around. Again the key thing for these processes is the pss; these processes should try to get their memory footprint down as much as possible when they are in this state, to allow for the maximum total number of processes to be kept around. Generally a well behaved app will have a pss footprint that is significantly smaller when in this state than when TOP.
- TOP vs. CACHED_ACTIVITY-CACHED_ACTIVITY_CLIENT
  The difference in pss between when a process is TOP vs. when it is in either of these specific cached states is the best data for seeing how well it is releasing memory when going into the background. Excluding CACHED_EMPTY state makes this data better, since it removes situations when the process has started for some reasons besides doing UI and so will not have to deal with all of the UI overhead it gets when interacting with the user.

Analysis

Analyzing app startup time

Use **`$ adb shell am start`** with the **`-P`** or **`--start-profiler`** option to run the profiler when your app starts. This will start the profiler almost immediately after your process is forked from zygote, before any of your code is loaded into it.

Analyze using bugreports

Now contains various information that can be used for debugging. The services include **`batterystats`**, **`netstats`**, **`procstats`**, and **`usagestats`**. You can find them with lines like this:

```
------ CHECKIN BATTERYSTATS (dumpsys batterystats --checkin) ------
7,0,h,-2558644,97,1946288161,3,2,0,340,4183
7,0,h,-2553041,97,1946288161,3,2,0,340,4183
```

Check for any persistent processes

Reboot the device and check the processes.
Run for a few hours and check the processes again. There should not be any long running processes.

Run longevity tests

Run for longer durations and track the memory of the process. Does it increase? Does it stay constant? Create Canonical use cases and run longevity tests on these scenarios.
