Differences between Desktop and Mobile Operating Systems

SEN YAO, XIAOCHEN HOU, YUJUAN ZHANG, NEIL SINGH, JAI PANDEY
Overview

• Runtime Environments and Process Virtual Machine
• Android Process Management
• Memory Management
• Energy Efficiency
• Multicore Scheduling
Runtime Environments and Process Virtual Machines

Sen Yao
Native Machine Code and Language Virtual Machine

- Both desktop and mobile systems implement both models for program execution

<table>
<thead>
<tr>
<th></th>
<th>Desktop</th>
<th>Mobile</th>
</tr>
</thead>
<tbody>
<tr>
<td>Native machine code</td>
<td>C / C++</td>
<td>Objective-C / Swift</td>
</tr>
<tr>
<td>Language virtual machine</td>
<td>Java virtual machine</td>
<td>Dalvik / Android Runtime (ART)</td>
</tr>
</tbody>
</table>

- Benefits to each execution model?
  - Range of target hardware configurations for an architecture
  - Legal obstacles caused by licensing
  - Architecture-specific optimizations discovered empirically
Architecture Optimizations in Android

- Recall: Android supports a range of low-powered hardware profiles with limited CPU speed and memory

- Stack-based VM: operations executed by popping operands from stack
Architecture Optimizations in Android

- Register-based VM: operations executed by expressly referencing register address where operand resides
Architecture Optimizations in Android

• Register-based VMs have longer instructions per operation but require fewer instructions for completing the same task

• The use of a stack-based architecture in the JVM is thought to achieve hardware independence, by avoiding direct references to registers that may or may not exist

• The use of register-based architectures in Android process VMs is thought to be motivated by reducing power consumption by reducing CPU time in the long term
Memory Optimizations in Android

- The JVM allocates 4 MB heap space to processes on startup, up to a maximum of 512 MB on most systems.

- The Android process VM allocates only 2 MB heap space to processes on startup, up to a maximum of 36 MB.
Memory Optimizations in Android

Visualization of Android memory usage as a result of typical heap size allocation
Memory Optimizations in Android

- JVM can suffer from severe performance degradation if page swapping occurs in conjunction with garbage collection, caused by a LRU page replacement policy

- Android uses paging, but has no swap space; dirty pages can’t be swapped out

- Instead of a LRU page replacement policy, a LRU policy for killing background applications

- .dex format provides “sparse” array data structures that are optimized over hash maps for small arrays
  - There may be gaps in the array when elements are marked for deletion; GC doesn’t immediately take place so “deleted” elements can be recycled
Compiler Optimizations in Android

- The JVM utilizes just-in-time (JIT) compilation, compiling methods as they are invoked; decreased CPU and memory usage at startup is traded for increased usage as the app runs.

- Major change from Dalvik to ART: switching from JIT compilation to ahead-of-time (AOT) compilation, later complemented by JIT, resulting in a hybrid runtime.

- JIT in both the JVM and ART is dependent upon tracking an application’s frequently run “hot methods”; JIT recompiles the app periodically to optimize for hot methods.

- AOT generates architecture-specific machine code; native code reduces power consumption.

- AOT compilation caused extensive “upgrade” time with every Android system update due to recompiling every application; this became an undesirable tradeoff when Android increased the frequency of security updates.

- In ART, JIT recompilation takes place while the system is idle.
Compiler Optimizations in Android

WorkFlow

APK is installed but nothing else

At runtime

Is method compiled?

Yes

Compile Code

AOT Compiled Code

JIT Code Cache

No

Interpreter

Is method warm?

Is method hot?

Creates Profile File

Data Code Cache (Profiles)

Periodically: What methods compiled is saved

Background compilation
Android Process Management

Xiaochen Hou
Components & Applications & Processes

• Components are the essential building blocks of an Android app. Each component is an entry point through which the system or a user can enter your app.

• There are four different types of app components: activities, services, broadcast receivers and content providers.

• Basic idea: one application, one process, one main thread

• When an application component starts and no other components are running, start a new Linux process with a single thread of execution
Process Control Block

- Android process management is similar to that of Linux at a low level
- PCB is managed by standard process management
Zygote

- Zygote is the parent of all App processes

- The Zygote process starts when the system boots and loads common framework code and resources

- Two benefits
  1. Speedup: RAM pages allocated for framework code and resources are shared across all app processes, no need to copy
  2. Memory saving: No matter how many applications are started, the increase in memory usage will be a lot smaller
Application Lifecycle

- As a user navigates through, out of, and back to app, the Activity instances in app transition through different states in the lifecycle

- Callbacks for handling state transitions
Importance Hierarchy

- An application process's lifetime is not directly controlled by the application itself, but rather by "importance hierarchy"

- The most important level found among all the components currently active in the process
Foreground and Visible Process

- **Foreground Process**
  - Process that the user is currently doing, referring to the activity that user is interacting with
  - Only a few such processes in the system
  - Killed if memory is so low that not even these processes can continue to run

- **Visible Process**
  - Doing work that the user is currently aware of, for example, visible to the user on screen but not in the foreground
  - The number of these processes is less bounded than foreground processes
  - Killed when it is required to keep all foreground processes running
Service and Cached Process

• **Service Process**
  ◦ Doing things that the user cares about, such as background network data upload or download
  ◦ Killed when there is not enough memory to retain all foreground and visible process

• **Cached Process**
  ◦ Not currently needed
  ◦ System is free to kill it as desired when memory is needed elsewhere
  ◦ Multiple cached processes always available and regularly kill the oldest ones as needed
Memory Management in Android Operating System

Yujuan Zhang
Android Shared Memory Driver

• Android shared memory driver (Ashmem)

• LINUX use POSIX SHM

• Similar to SHM but with ‘different behavior’

  The Ashmem memory dies when the process dies
  Designating some pages of memory as reclaimable—unpinning a section of memory
  Shrink mapped regions if system is in need of memory
Android Low Memory Killer

- Android low memory killer (LMK)
- Linux out of memory killer (OOM)
- OOM killer sometimes kill high priority process in low memory conditions
- LMK has interface with activity manager which knows priority of processes
- Low memory killer generally kill user space process, where as out of memory killer kill process randomly using some finest complex algorithm but never get killed itself.
- When out of memory killer is invoked it may result in kernel crash.
The Android systems use flash memory, while the standard Linux systems use magnetic drives.

Why NAND flash?

- Dense (more storage per chip) — space constraint
- Read & Writes fast
- Better resistance to kinetic shocks
- Low cost
- A lower memory footprint
File System

• Yet Another Flash File System (YAFFS)
  ◦ The only file system designed for NAND flash

• Log-structured file system—both fast and robust
  ◦ Write fast
  ◦ Easy to recreate from log

• Impressive performance on high capacity NAND flash

• Highly portable
Energy Efficiency in Mobile Operating Systems

Neil Singh
Why It’s Necessary

- Increased frequency and performance come at the consequence of increased power

![Figure 14. Power Performance Trade-offs.](image)
Why It’s Necessary

- You can’t use your phone when its dead.
- SmartPhone producers advertise their battery life compared to competitors.
Why It’s the Operating System’s Job

• OS manages resources and hardware

• OS controls CPU utilization for each application through scheduling
The CPU

- Operating System can change the CPU’s frequency depending on the workload
- Gives peak performance of higher frequency with average energy consumption of lower frequency
- Example: TurboBoost in Intel i7 Mobile [1]
The Scheduler

• Can also schedule based on equal energy consumption [4]

• Rewards energy efficient applications

• Easier to manage energy consumption
The Peripherals

• The operating system can turn off peripherals that are not currently being used
  • Done in iOS [3]

• Can also scale back range of select peripherals such as bluetooth or wifi
Conclusion

• Overall, mobile OS manages energy efficiently by managing:
  • CPU
  • Tasks
  • Peripherals
Citations


Multicore Scheduling in Mobile Operating Systems, for symmetric and asymmetric processing architectures

Based on ARM White Papers and slides by Mike Anderson
Jai Pandey
With the explosive growth of the number of smartphones in the recent years (with the sales surpassing PC sales), the smartphone is has become the most common compute device.

Although mobile processors have advanced a lot over the past decade, both in terms of efficiency and computation power, mobile phones still have energy constraints stricter than the good old PC(or Mac).

Battery technology hasn’t been able to keep up with the advances in the processors.

As we push heavier loads onto our mobile phones, like Virtual Reality or even Video Editing in some cases, energy management has become even more crucial.
Goals for the Section

• Look at multicore scheduling keeping the energy considerations in mind.

• We’ll focus more on the big.LITTLE ARM architecture and how asymmetric compute cores can be leveraged to achieve better performance.

• We’ll gloss over Apple’s implementation of the asymmetric computation cores work in the new A10 Fusion SoC.
Workloads

• There are three basic categories of workloads on a mobile device:

  1. High Intensity Bursty Workloads

  2. Sustained Workloads (with power and thermal limits)

  3. Long-use, low intensity workloads
Energy Considerations for Multi Core Systems

- Operating Voltage
- Operating Frequency
- Number of Cores engaged
Symmetric vs Asymmetric Architectures

- More powerful processors (cores) generally are more power hungry.
- Less powerful processors (cores) are generally more power efficient.
- Symmetric Multiprocessor (Multicore) Architectures have the same processor for each of the multiple processors (cores) the system has.
- Asymmetric Multiprocessor (Multicore) Architectures have different processors for the system. For most of the implementation, some of them are more powerful but less power efficient, while some of them are more power efficient but less powerful.
ARM’s big.LITTLE Architecture

- The system consists of two clusters of processors (cores), big and LITTLE.
- The big cluster consists of cores which are more powerful but less power efficient.
- The LITTLE cluster consists of cores which are less powerful but more power efficient.
Cluster Switch Scheduling

- In the initial implementation of the big.LITTLE architecture, the CPU was cluster scheduled. Either the big or the LITTLE cluster was active, but not both.

- Significant impact on performance during the cluster switch.

- Cache coherency hardware is a must.

- The whole cluster is activate at once, leading to inefficiencies in the case not all the cores are needed.

- The cluster switch approach was the default scheduler in Android 4.2.2

- Android uses cgroup and DVFS mechanisms to help make cluster decisions.

- CPU affinity can be used to override scheduling decisions
Linux Scheduler picks any **One Cluster** at a time, but no combinations. **High Cluster** is picked if at least one High Core is needed, else **Low Cluster**.
Problems with Cluster Switching?

- The Cluster Switch itself is a significant overhead, so the savings aren’t as great as one would like them to be.

- Only one cluster is active at a time. What if we needed all processors(cores)?
In-Kernel Switching or CPU Migration

• The big and LITTLE cores are teamed up into virtual CPUs, one big and one LITTLE core per VCPU.

• Whether the application runs on the big or the LITTLE core is based on CPU load for that task

• Allows a mixture of LITTLE and big cores to run.

• Unbalanced processors?

• IKS patch came out for the 3.10 Linaro Stable Kernel (LSK) and 3.14 Linux mainline

• Problems?
Global Task Scheduler or MP

- Each processor core is active and can be scheduled
- The previous load for the task determines which core the task runs on
- CPU affinity can be used to lock the task to a particular core
<table>
<thead>
<tr>
<th></th>
<th>big.LITTLE IKS</th>
<th>big.LITTLE GTS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Configuration</strong></td>
<td>Pairs of (Cortex-A15 and Cortex-A7) can be inactive or have one core active at a time</td>
<td>Any combination of cores can be available simultaneously</td>
</tr>
<tr>
<td><strong>SoC Cores</strong></td>
<td>Used for SoCs with equal number of Cortex-A15 and Cortex-A7 cores</td>
<td>Can also be used for non-symmetric SoCs - e.g. 4 Cortex-A7 + 1 Cortex-A15</td>
</tr>
<tr>
<td><strong>Kernel Impact</strong></td>
<td>Minimal impact on the Linux kernel (extends existing governors)</td>
<td>More impact on the kernel (Scheduler, process annotation etc.)</td>
</tr>
<tr>
<td><strong>Maximum processing throughput</strong></td>
<td>All Cortex-A15s running simultaneously</td>
<td>All Cortex-A15s and all Cortex-A7s running simultaneously</td>
</tr>
<tr>
<td><strong>Switching</strong></td>
<td>Uses cpufreq framework</td>
<td>Uses scheduler directly, giving ~10% improvement on many ARM benchmarks</td>
</tr>
<tr>
<td><strong>Availability</strong></td>
<td>Now in Linaro monthly builds and LTS-based LSK</td>
<td>Now in Linaro monthly builds and LTS-based LSK</td>
</tr>
<tr>
<td><strong>Upstream (kernel.org)</strong></td>
<td>Underway and expected in Linux 3.11 or 3.12</td>
<td>Underway and expected in next few quarters</td>
</tr>
</tbody>
</table>
Beyond Global Task Scheduler

• Energy Aware Scheduling (EAS)
  • EAS is a set of extensions to the Linux kernel that introduce energy model based decision making for task scheduling and power-performance control. EAS aims to make power-performance control more centralised with the scheduler being the primary driver for power-performance decisions.
  • This contrasts with the current situation where the scheduler, cpufreq and cpuidle tend to step on each other's toes. The goal is to simplify power-performance management with a scheduler-driven policy and a small set of well-defined tunables.
Changes Required for Programmers

• Virtually none for User Space Programmers. However, programmers can make sure that they program in such a way that the programs best utilize the hardware they are running on. For example a highly parallelized workload would run better (more efficient) on hardware that has a larger number of LITTLE cores.

• The schedulers are provided as patches and are completely within the Kernel space, being really easy to abstract.
big.LITTLE’s Performance

On different types of Workloads
High Intensity (Bursty loads)
Sustained Workloads

- The Cortex-A7 cores are run at the maximum frequency, and the Cortex-A15 cores are run as needed. The allocation of work to big and LITTLE cores enables a significant reduction in power, as shown in the next graph.

- The Castle Master game and a similarly high performance Real Racing game, both experience significant power savings relative to the base case where a Cortex-A15 processor runs by itself under DVFS control.
Low Intensity
A Small Overview of the Apple A10 Fusion SoC

- There are two kinds of cores, “Hurricane”, and “Zephyr,” a smaller, lower-power core — are bigger than the competition.

- The A10 Fusion has four CPU cores (two high-performance cores, two low-power cores), but only two can be powered on at any given time, similar to cluster switching.

- The two fast cores run at about 2.34GHz

- The two low-power cores seem to run at about 1.05GHz based on Geekbench 4's CPU clock speed reporting while in Low Power Mode
References


3. Ten Things to Know About big.LITTLE, Brian Jeff. Available: [https://community.arm.com/processors/b/blog/posts/ten-things-to-know-about-big-little](https://community.arm.com/processors/b/blog/posts/ten-things-to-know-about-big-little)

