Improvement of Start-up Time on Linux-Based IVI System

2016/7/13
Panasonic Corporation
Kazuomi KATO
AGENDA

1. Background and Issue
2. Methods of optimizing startup time
3. How to analyze
4. Analysis of start-up time
5. Improvements
6. Summary
7. Future plan
Due to the functionality such as smart phone link function, more and more functions and large-scale software are being required.

Data size is being increased due to the expansion of the display resolution and the realization of multi-screen being equipped.

At the same time, start-up time and quick response of the system are required by the driver much more.

Mobile devices equipping a lot of functions as much as IVI system can quickly start up, therefore quick start-up time of IVI system is required as well.
Start-up time issues of IVI system:

1. Realization of quick start-up time suitable for users who are used to smart phone.

2. In order to put the functions of music player, car navigation into operation as soon as the engine being started, high speed of start-up time of IVI system is necessary.

3. To improve the UX and display resolution, the amount size of data are increased. As a results, reading the data from storage costs much longer.

4. With the change of platform from RTOS to Linux, the start-up time is increased.

5. In the application of vehicle, although large battery is equipped, the issues of dark current has to be considered, when the car not being used for several weeks.
### Methods of optimizing startup time

We adopted the approach to analyze and eliminate the bottleneck.

<table>
<thead>
<tr>
<th>Approach</th>
<th>Typical method</th>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Eliminate the bottleneck and improve processing</td>
<td><strong>Profiling and Analysis</strong>&lt;br&gt;Improving method based on the profiling and analysis.</td>
<td>Fundamentally improvable.</td>
<td>Know-how is required to specify the bottleneck and improve.</td>
</tr>
<tr>
<td>2. Startup is quickly shown as a user's view</td>
<td><strong>CAN wakeup</strong>&lt;br&gt;Method of starting the system in advance by CAN signal such as unlocking the door.</td>
<td>It seems to start up fast when getting into a car.</td>
<td>It depends on OEM requirements.</td>
</tr>
<tr>
<td>3. Fast return without completely stopping the system</td>
<td><strong>Suspend / Resume</strong>&lt;br&gt;Method of saving the current state to DRAM, turning peripheral device power off, waiting and restarting from the state saved at that time.</td>
<td>Fast resume is possible at turning power ON.</td>
<td>The electric power for maintaining DRAM data is consumed.</td>
</tr>
<tr>
<td>4. Carry out power OFF adding to No.3 approach</td>
<td><strong>Snapshot boot</strong>&lt;br&gt;Method of saving the kernel image of a certain point into the storage and booting up from that point when resuming the system.</td>
<td>Electric power is not consumed as compared with No.3.</td>
<td>If boot image size becomes large, the load time from storage will become a neck.</td>
</tr>
</tbody>
</table>
How to analyze

- Divide the system into several phase from power ON to start of the application.
- Since the reasons which cause the bottleneck in different phases are different, therefore, considering the approach to those respectively is necessary.

Overview of each phase during the system start-up

- Loading binary image
- Display opening picture

- Minimum preparation for user space processes
- Make /dev files
- Execute system manager process

Bootloader → Kernel → System init → IVI Service → Application

Startup flow

Starting Linux kernel
- MMU, page, ... initializing
- Device driver initializing

Activate service processes from manager process according to their priority

Bottleneck mainly caused by I/O

Bottleneck mainly caused by CPU
In order to realize the fast boot, use the following as hardware:

- DRAM Self-refresh in part of DDR3
- Connect the high speed eMMC to the SD UHS-I(200MHz) port.
  (because of SoC restriction, eMMC I/F speed is less than device speed)
## Analysis of start-up time in each phase

### Consider the solution of specific bottleneck based on analysis.

### Start-up time in each phase at the beginning (@Cortex A15 1.5GHz x2 SoC)

<table>
<thead>
<tr>
<th>Phase</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bootloader</td>
<td>5.9 sec</td>
</tr>
<tr>
<td>Kernel</td>
<td>1.1 sec</td>
</tr>
<tr>
<td>System init</td>
<td>0.7 sec</td>
</tr>
<tr>
<td>Service</td>
<td>6.0 sec</td>
</tr>
<tr>
<td>App</td>
<td></td>
</tr>
</tbody>
</table>

### Issue of start-up time in each phase

1. **Bootloader**
   - Program loading time from eMMC device occupies.
2. **Kernel**
   - Waiting for device driver initialization occurs.
3. **System init**
   - The CPU usage rate has only been about average 60%.
4. **Service**
   1. Waiting for the process of mounting file system occurs.
   2. Low speed of loading from storage device (eMMC) when starting service.
   3. CPU load increases due to the interruption of eMMC when system started up (2-6secs).

---

**Analyze the method to solve the bottleneck caused by I/O.**

**Analyze the method to solve the bottleneck caused by CPU.**
Improvements for bootloader(1)

The bottleneck is loading time from eMMC to DRAM.

Processing | Time  
---|---
 eMMC -> DRAM transferring procedure (100Mbyte) | 5,000msec  
eMMC initilizing procedure | 200msec  
Debug log output | 200msec  
ROM integrity checking | 65msec  
Primary loader process | 40msec  
...

Before | 20MB/sec  
After | 80MB/sec

Connecting eMMC to SD UHS-1 line

However eMMC bottleneck remains.

For more improvement

Backuped data on DRAM is available
Partial DRAM backup system is necessary.

DRAM backup
  • The backup data on DRAM is reused when turning ignition ON.

Issue:
  • Not allowed to back up whole DRAM area because of the large dark current.

Approach:
  1. Scope for backup is limited to program code.
  2. Program for backup is gathered to the specific memory area.
  3. Only that area is the target area for DRAM backup.

What program should be backuped?
  • Root file system: Executable binaries required on start-up are chosen.
File system available for DRAM backup system should be chosen.

Requirements of DRAM backup for root file system
1. It is operable on RAM.
2. It is not lost by reboot of Linux.
3. No matter happens by reusing backup image of initial state.
4. Not allowed to consume extra RAM.
5. As a security requirement, it is available for dividing the privileges associated with super user into distinct units.

The tmpfs, romfs and cramfs as a candidate don’t meet those requirements.

We adopted PRAMFS (Protected and Persistent RAM Filesystem)

DAX (for direct access) system was experimental and not adopted at that time.
Result: The startup time of bootloader was improved from 5.9 sec to 0.15 sec by using DRAM backup system.

### Improvements for bootloader (4)

<table>
<thead>
<tr>
<th>Processing</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>eMMC → DRAM transferring procedure (100Mbyte)</td>
<td>5,000msec</td>
</tr>
<tr>
<td>eMMC initializing procedure</td>
<td>200msec</td>
</tr>
<tr>
<td>Debug log output</td>
<td>200msec</td>
</tr>
<tr>
<td>ROM integrity checking</td>
<td>65msec</td>
</tr>
<tr>
<td>Primary loader process</td>
<td>40msec</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

eMMC accesses was eliminated.

The log output process for debug is also deleted.

Reused.
Improvement for kernel

Result:
The startup time of kernel was improved from 1.1 sec to 0.8 sec by parallel execution for device driver initialization.

Before
- SDHI initialize: 290msec
- Touch screen: 320msec

After
- Touch screen: 320msec
- SDHI initialize: 290msec

Create a thread for SDHI

0.8 sec
We developed original simple init called “system init”. (Not System V init)

- Making device files and symbolic links
- Mount filesystem
- Setting resource parameter
- Setting network parameter
- Activating a service process

Why originally developed?
- Need the minimum init functionality.
- Standard init process like systemd is not necessary for our product requirements.
Improvements for system init (2)

1. Low CPU usage part
   1-1. Modified code from shell script to native C code - 2.3 sec
   1-2. Changed serial execution to parallel execution - 0.1 sec

2. High CPU usage part - 0.3 sec
   - Optimized the udev event control
1.2 Parallel execution -0.13 sec
Process groups based on each dependencies are parallel executed as the threads.

- rcpvr: initialize GPU driver
- init_fs: mount process
- init_insmode: loading modules
- init_dir: chown/chmod
- init_iptables: firewall settings
- init_command: some commands
- init_udev: udev process
Improvements for system init (4)

2. Optimize the udev event control  
   Devices to be excepted from “udev” should be clarified and processing of "udev" occupied in start-up should be optimized.  

   -0.3 msec

Issue:
1. Thread processing for handling an udev event spends time from 20 msec to 40 msec. About 400 msec occurs.

2. However, the udev operation is necessary for plug and play device in our system.

Approach:
1. The udev which can specify the exceptional events such as the system fixed devices is originally developed.

2. The original udev has only the ability to register the event for controlling.
Improvements for system init (5)

Result:
The startup time of “system init” was improved from 3.0 sec to 0.3 sec

Improved “system init” phase through the approach as below
1. Change to the native C code
2. Parallel execution
3. Optimization for udev event control
Improvement for service processing (1)

**Issue:**
CPU load increases when accessing to eMMC during activating service processes.

**Analysis:**
1. eMMC access after Linux starting is almost loading to not contiguous DRAM area.
2. It was found that since DMA transfer completion interruption was occurring frequently, CPU load is increasing.
Improvement for service processing (2)

**Approach:**
The DMA for eMMC to non-contiguous areas can be transferred by using the function of the hardware as if they were a contiguous area.

---

### After improvement

<table>
<thead>
<tr>
<th>Item</th>
<th>before</th>
<th>after</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DRAM</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Descriptor</td>
<td>off</td>
<td>on</td>
</tr>
<tr>
<td>BounceBuf</td>
<td>Used</td>
<td>Unused</td>
</tr>
<tr>
<td><strong>CPU Usage</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mmcqd DMA driver</td>
<td>26.6 %</td>
<td>9.0 %</td>
</tr>
<tr>
<td><strong>Interruptions</strong> (times/sec)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SDHI DMAC</td>
<td>11,029</td>
<td>2,221</td>
</tr>
<tr>
<td><strong>Read (MB/sec)</strong></td>
<td>57.1 MB/s</td>
<td>71.5 MB/s</td>
</tr>
</tbody>
</table>
As a result of the CPU load decrease of eMMC access, it was shortened from 6.0 sec to 5.2 sec.
1. We have achieved 9.6 seconds reduce for the startup time.
2. Service phase still now occupies 5.2 seconds during startup time.
3. Improvement for each service is still on the way, and other system boot optimization method will be required to meet user’s requests.

1. Before (original code)

- Bootloader: 5.9 sec
- Kernel: 1.1 sec
- System init: 3.0 sec
- Service: 6.0 sec

2. After the improvements

- Boot: 0.15 sec
- Kernel: 0.8 sec
- init: 0.3 sec
- Service: 5.2 sec

**-9.6sec**
Future plan

1. The methods, which are independent of tuning application, as bellow need to be considered to the next IVI system.

- CAN Wakeup : Startup is quickly shown as a user's view
- Suspend and Resume : Since the performance of DRAM and the dark current will be improved, the availability of backup of the whole DRAM increases.
- Snapshot Boot : Because of the speedup of eMMC loading, cold boot from snapshot will become quick.

Device trend for IVI system

<table>
<thead>
<tr>
<th></th>
<th>Current</th>
<th>Next Generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>DRAM I/F</td>
<td>DDR3 1.6GHz</td>
<td>LP-DDR4 3.2GHz</td>
</tr>
<tr>
<td>Dark current (12V) 4GB</td>
<td>18.0 mA</td>
<td>2.2mA</td>
</tr>
<tr>
<td>eMMC read perf.</td>
<td>80MB/s</td>
<td>300MB/s (expected)</td>
</tr>
</tbody>
</table>
Thank you!