Virtualized Dual Android on Top of the VMXL4 Microkernel

Mihai Carabas, Irina Grădinaru, Lucian Mogosanu, Adrian Sendroiu, Ştefan Tică
Computer Science and Engineering Department
University POLITEHNICA of Bucharest
Bucharest, Romania
{mihai.carabas,irina.gradinaru,lucian.mogosanu,adrian.sendroiu,steфан.tica}@cti.pub.ro

Abstract—This paper presents a solution providing two virtualized Android operating systems on top of VMXL4, a performance-oriented microkernel with hypervisor capabilities. The Android instances run paravirtualized in separate containers, facilitating their use as “virtual phones” in enterprise environments, where traditionally employees carry separate phones for private and corporate use. We present the components comprising our solution and discuss device virtualization, then show the results of our efforts.

Keywords—virtualization; embedded; mobile; microkernel; performance management; Android

I. INTRODUCTION

Virtualization is rapidly becoming a requirement on the smartphone market. Research done by Gartner [2] predicts that in the future, Bring Your Own Device (BYOD) strategies will become mandatory in the enterprise. At the same time, companies often impose corporate policies on mobile phones, thus requiring a separation between personal and corporate phones. In this context, virtualization comes as a solution that is both cheap and comfortable for the mobile phone users, who can avoid the need to carry two separate physical phones.

Our solution offers an environment consisting of two virtualized Android operating systems running on top of VMXL4, a microkernel acting as a thin layer between the bare metal hardware and the guest OS. The two Android instances run fully separated, i.e. each on top of its own isolated Linux kernel, allowing complete separation between the personal phone and the corporate phone.

The paper is structured as follows: Section II presents similar virtualization solutions. In Section III we describe the experimental setup that we used to implement our solution. We present its architecture in Section IV and then we discuss device virtualization in Section V. Finally, in Section VI we present the results of our work and in Section VII we talk about future research and development directions for the project.

II. RELATED WORK

We give a summary on other attempts to provide solutions for running a dual virtualized Android on mobile embedded platforms. Andrus et al. [8] describe Cells, an architecture for smartphone virtualization. The solution uses OS-level virtualization to provide isolation between operating system instances and the idea of device namespaces to isolate devices at the kernel level. All these are encompassed into the concept of Virtual Phones (VPs), virtualization of both user and kernel devices dealing mainly with the aspect of securely switching between VPs.

LeVasseur [20] proposes a scheme that reuses the drivers in existing paravirtualized operating systems. Lange et al. [19] present L4Android, a port on Android over the Fiasco.OC microkernel, based on L4Linux [18]. L4Android and L4Linux run paravirtualized on top of the microkernel and L4Re runtime environment, reusing most of the native drivers of the underlying machine. The article deals specifically with a dual Android scenario where certain security needs (isolation of security-critical drivers, corporate and personal Android instances etc.) are met.

Heiser and Leslie [16] discuss the OKL4 Microvisor, a microkernel based on the L4 architecture aimed at providing virtualization for mobile and embedded environments. The design of OKL4 is largely based on the formally verified seL4 microkernel developed by Klein et al. [17]. General Dynamics present a commercial demo [3] of a phone running two Android instances on top of the OKL4 microvisor.

A solution based on VMWare’s virtualization technology is presented by Barr et al. [9]. The Mobile Virtualization Platform (MVP) is a Type 2 hypervisor, providing a host OS that includes the virtualization software, along with a guest OS contained inside the host, similar to the architecture used by VMWare’s desktop solutions. Lightweight paravirtualization is achieved by using a set of drivers in the guest kernel that communicate via shared memory with the devices in the host kernel, while host utilities provide features such as virtual machine checkpointing.

A dual Android demo for both the Pandaboard platform and the Samsung Galaxy Nexus phone was developed by the B Labs [5] community, using their own L4-based hypervisor called CODEZERO.

Traditionally, desktop-oriented virtualization solutions such as Xen or KVM were designed for the x86 architecture. Dall and Nich [12] describe the work made to port KVM

1 http://os.inf.tu-dresden.de/fiasco
on the ARMv5 to ARMv7 architectures, the main challenges consisting of virtualization of the CPU and memory.

III. Experimental Setup

The long-term goal of our solution is deployment in mobile embedded scenarios, such as those where parties employ BYOD policy. This leads to hardware and software requirements which we will describe in the following paragraphs.

The VMXL4 microkernel runs on ARMv5 to ARMv7 hardware architectures, more specifically on XScale PXA and Texas Instruments OMAP System-on-Chips (SoC) [10], [13], [14]. The latest SoCs we support are OMAP4430 and OMAP4460, the former being implemented on Pandaboard, while the latter is part of Pandaboard ES and the Samsung Galaxy Nexus phone. Our ability to run single, paravirtualized instances of Android comes as a result of previous porting work done in the Linux 3.0.8 kernel [11], [22].

Next, we give a detailed description of our choices.

Pandaboard: The Pandaboard is a development board released by Texas Instruments, based on the OMAP4430 and OMAP4460 SoCs. The CPUs are dual-core Cortex A9 running at 1-1.5GHz frequencies and the GPU is PowerVR SGX540. Several other components such as DSPs, Bluetooth/WiFi, MMC/SD or face detection are included on the board [4].

Samsung Galaxy Nexus: Being one of the latest Nexus smartphones supported by Google, the Galaxy Nexus is based on OMAP4460, making it the best choice for our project due to the fact that it reduces the time required to port our software components. In addition to the Pandaboard, the Galaxy Nexus comes with all the peripherals associated with a phone, the most relevant from the point of view of usability being the touchscreen and the baseband module.

Linux 3.0.8 kernel: We used the Linux kernel branch associated with the Android Ice Cream Sandwich, Linux 3.0.8, with two sets of patches: the Android patches adding support for mechanisms such as Binder or the Low Memory Killer, and the patches that enable paravirtualization on top of VMXL4. For the Galaxy Nexus, we applied the same patches on the Maguro branch of the Linux kernel.

Android Ice Cream Sandwich: The Android framework runs on top of the paravirtualized Linux kernel, as described in Section IV. We chose to use the framework provided by the Android Open Source Project (AOSP) [11], the availability of the source code making it easy to adapt Android to our specific needs. For example, one of the two Android instances runs using software rendering, as detailed in Sections V-A and VI.

IV. Architecture

As mentioned in the previous section, the base system we used for running multiple Android Linux kernel instances on top of a single hardware is based on a L4 microkernel from VirtualMetrix [6], also called VMXL4. It isolates each operating system personality in the system in containers called Secure Domains (or shortly SecDom). Basically the microkernel provides abstraction for the CPU and memory.

Due to the fact that it is only an moderator between entities in the system for the CPU and memory, we need a management Secure Domain that takes care for resource management of the system, like a hypervisor (e.g. Xen Hypervisor [7]). In our architecture, we will call this new management Secure Domain SecDom0 (see Figure 1).

SecDom0 is responsible with allocating resources as follows:

- the physical memory
- number of allowed address spaces
- required capabilities
- number of allowed threads

Another feature implemented in the SecDom0 involves restarting one of the containers. In case one of the operating system instances executes a reboot, we need only to restart that instance, not the whole board. SecDom0 would catch the reboot command and would reclaim the resources allocated at boot time.

![Figure 1. Architecture for Dual-Android on top of the VMXL4](image)

The two Linux kernels are running in separate Secure Domains, being isolated from each other (see Figure 1). The first Linux kernel will have the role of an I/O server for various devices (see next sections). Due to the fact that the Linux kernels are isolated, there has to be some kind of communication mechanism between them. In fact there are two:

- IPC (Inter Process Communication) provided by the microkernel
- Shared Memory provided by the SecDom0.

SecDom0 ensures the Shared Memory mechanism by allocating the same physical memory range for both drivers (the server and the client).

In this architecture, the VMXL4 microkernel virtualizes the CPU and the memory and one of the Android SecDoms,
called the I/O server, is responsible for virtualizing all the I/O devices. We discuss virtualization in the following section.

V. VIRTUALIZATION

As described in section IV, our setup consists of two Android instances, each with its own Linux kernel. The first instance is the I/O server, which has direct access to physical devices and acts as an intermediate for the second instance, the Linux client. Thus, all I/O operations that the second Android makes will pass through the first.

Device virtualization is done using the VirtOps framework described in our previous paper [19]. The paper shows how we used VirtOps to virtualize some of the required I/O devices: block I/O, serial console and network. Two more devices, the display and the input subsystem, are the topic of the rest of this section.

A. Display virtualization

In order to avoid having to virtualize the GPU, we use software rendering in the second Android instance. We disabled hardware rendering in the Android framework using the same method as the B Labs [5] Dual Android project. However, the B Labs demo uses a VNC server on Android1 and a VNC client on Android0 in order to display the screen of the second Android. The inherent overhead of the VNC protocol makes this a slow solution, so we tried for a different approach, described next.

By default, the Android subsystem renders graphics in the first framebuffer, accessible via `/dev/graphics/fb0`. In the second Android, `fb0` is managed by a driver based on the virtual framebuffer implementation in the Linux kernel. The only difference is that, instead of being allocated from the kernel’s memory, the virtual framebuffer is mapped in a shared memory segment defined in the build system of the VMXL4 microkernel.

The first Android’s display subsystem also handles the second Android’s screen. We do this by allocating an extra framebuffer in Android0 and mapping it in the same shared memory segment as Android1’s virtual framebuffer (see Figure 2).

The OMAP Display Subsystem (DSS) exposes a configuration interface in Linux’s sysfs. We use this interface to switch the display between the two Android instances. We make use of the overlay objects of the OMAP DSS. An overlay defines a rectangular area of a framebuffer and is used by an overlay manager to redraw only certain parts of the screen at a time and thus make the refresh process more efficient. In our case, we “borrow” one of the OMAP DSS’s overlays and set it to contain the entire extra framebuffer. The switch between the Android instances is done by enabling and disabling overlays via the sysfs entries.

B. Linux Kernel Input Subsystem

The Input subsystem is an abstraction layer between devices (keyboard, mouse, joystick, touchpad, and so on) and input handlers. The input devices capture inputs from the user actions or from other sources and produce input events. The input events go through the Input core and are dispatched to the interested handlers, which can make them available to user space through the standard Unix file interface. The Input core provides a many-to-many mapping between input devices and event handlers.

The input subsystem operation mode is depicted in figure 3. The two drivers classes working together are device and event drivers. The first ones are responsible for communication with input devices (the PS/2 mouse driver for example) and the later for applications interfacing (e.g., mousedev). The Input Core is an efficient, bug-free, reusable code which is the heart of the input subsystem.

1) Input Device Drivers: The input device drivers are responsible for low-level communication with hardware input devices and publishing input events accordingly. When loaded into the kernel, a driver module of an input device usually sets up a probing routine to detect the presence of the types of hardware it is supposed to manage. If successful, the module will invoke the function `input_register_device( ... )` in `include/linux/input.h` which sets up a file representing the physical device as `/dev/input/eventX`. The events are reported (published) in a standard format by calling one or more of the `input_*` functions in `include/linux/input.h`; these include:
   - `input_event( ... )` - general purpose
   - `input_report_key( ... )` - for key down and up events
   - `input_report_abs( ... )` - for position events e.g. from a touchscreen among others.
Finally, when all event publishing is finished, the event processing method calls `input_sync(...)` to flush the event out.

2) Input Event Drivers: The event interfaces exported by the input subsystem have evolved into a standard that many graphical windowing systems understand. Event drivers offer a hardware-independent abstraction to talk to input devices. Event drivers, in tandem with frame buffer drivers, insulate graphical user interfaces (GUIs) from the vagaries of the underlying hardware. The Linux kernel offers a set of standard event drivers such as:

- **mousedev** - for mouse-like devices
- **joydev** - for joysticks controllers
- **evdev** - generic input event driver

Each event packet produced has the format depicted in Listing 1 defined in `include/linux/input.h`.

```c
struct input_event {
    struct timeval time; /* Timestamp */
    __u16 type; /* Event Type */
    __u16 code; /* Event Code */
    __s32 value; /* Event Value */
};
```

Listing 1. The input_event structure

When loaded into the kernel, each event driver module calls the `input_register_handler(...)` function in `include/linux/input.h` which registers a new input handler (part of a event driver) for input devices in the system (registered by input device drivers as depicted in [V-B1]) and attaches it to all input devices that are compatible with the handler. The compatibility is determined by matching the entries of a `struct input_device_id` array provided by the handler with device specific indentifying information. For example, the `evdev` event driver passes the array depicted in Listing 2 which matches all devices.

```c
static const input_device_id evdev_ids[] = {
    {.driver_info = 1 }, /* Match all devices */
}; /* Terminating entry */
```

Listing 2. The evdev device matching ids

By contrast, the `mousedev` driver is looking for specific devices as we can see in Listing 3.

```c
static const input_device_id mousedev_ids[] = {
    {.flags = INPUT_DEVICE_ID_MATCH_EVBIT | INPUT_DEVICE_ID_MATCH_EVBIT |
        INPUT_DEVICE_ID_MATCH_KEYBIT, .evbit = { BIT_MASK(EV_KEY) | BIT_MASK(EV_REL) },
    .keybit = { [BIT_WORD(BTN_LEFT)] = BIT_MASK(BTN_LEFT) },
    .relbit = { BIT_MASK(REL_X) | BIT_MASK(REL_Y) },
}, /* A mouse like device, at least one button, two relative axes */
```

Listing 3. A piece of mousedev device matching ids

3) Input Virtualization: Due to the fact that the Linux Clients have no direct access to the underlying hardware we cannot make use of default input device drivers as they will have no controllers to communicate with. Our solution is based on intercepting the input events issued by the Server’s input device drivers and passing them to each Client. This leads us to a client-server communication model based on the VirtOps framework.

4) Server Side: As described in section [V-B1], the input events are published by the input device drivers by calling a `input_event(...) family function. By inspecting the `input_event(...) function implementation in `drivers/input/input.c` we can see that the `input_pass_event(...) function is eventually called. This function passes the received event first through all filters and then, if the event has not been filtered out, through all open handles. An input event filter is an event driver-like module exposing a filtering routine which returns `true` if the event is filtered or `false` otherwise. This is exactly the functionality that we are looking for and implementing an input event filter suits our purposes of input virtualization. Like the `evdev` event driver, our filter matches all input devices so that each input event is sent to every registered...
Clients using the VirtOps framework. To associate each input event with a specific device some identifying information (such as bustype, vendor, product_id, version) is passed along.

5) Client Side: On the Client side, each input device is emulated by a fake input device driver build using the input device driver template. Unlike a native driver who communicates with the underlying hardware for issuing input events, our fake drivers get their events from the Server through the VirtOps framework and publish them by calling input_event(...) function.

VI. RESULTS

We managed to bring up two Android frameworks, running simultaneously on top of a single hardware, a phone from Samsung: Galaxy Nexus. However, there are several dropdows in this configuration. Mainly it consists of the features available on the second Android (the client). We only virtualized the block device (flash), the network driver, the display and the touchscreen input. Table 4 presents all the features available for the first Android and are missing in the second.

<table>
<thead>
<tr>
<th>Android 0 (I/O Server)</th>
<th>Android 1 (Client)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block device (flash)</td>
<td>Working (hardware) Working (virtual)</td>
</tr>
<tr>
<td>Network</td>
<td>Working (hardware) Working (virtual)</td>
</tr>
<tr>
<td>Display</td>
<td>Working (hardware) Working (virtual)</td>
</tr>
<tr>
<td>Input (touchscreen)</td>
<td>Working (hardware) Working (virtual)</td>
</tr>
<tr>
<td>GPU</td>
<td>Working (hardware) Not-working</td>
</tr>
<tr>
<td>Multimedia system</td>
<td>Working (hardware) Not-working</td>
</tr>
<tr>
<td>Camera</td>
<td>Working (hardware) Not-working</td>
</tr>
<tr>
<td>GSM system</td>
<td>Working (hardware) Not-working</td>
</tr>
</tbody>
</table>

Figure 4. Devices available in the Dual Android setup

The GPU and the multimedia system (needed for HD video playing and camera recording) have proprietary user-space drivers available only in binary format. They only use a bridge in the kernel to communicate with the device. This is the reason virtualizing these devices is very time consuming and almost impossible.

As from a performance point of view, our virtualized Android is running almost twice slower as the native one. Mainly the performance drop is at boot-time, in loading the Android framework. Loading the Linux kernel is almost as fast as the native one and the drop-down comes when starting to initialize the Android framework. User experience isn’t affected very much after booting up the phone.

The Galaxy Nexus has two cores; thus we run each Android on one core. This would be another reason for the difference of performance between native and virtualized Android.

The remarks made until now applies for the first Android (the I/O server). For the second Android we can add the fact that the GPU isn’t available and all the graphic processing is done on the CPU. We had to modify the Android framework in order to enable software rendering, instead of the hardware rendering done by the GPU.

Besides the CPU strip-down in our experimental setup (using one instead of both cores for a single Android), we had to divide the limited memory between the two Androids. Our device had one gigabyte of RAM and we allocated about 600 megabytes for the first Android which had access to all devices and 300 megabytes for the second Android which had limited access to devices and services. We used the rest of the memory for the microkernel and for the management domain, the SecDom0. The problem with this strip-down of the memory is that Android is very memory hungry, especially with some applications like the browser.

In our configuration, in the second Android you cannot load a consistent page because you would rapidly remain out of memory. To be more precise, the Linux kernel plus the minimal Android framework consumes about 220 megabytes of RAM. About 80 megabytes would remain for the other applications. From our tests, with the most heavy pages (full of images) the browser needs about 100 megabytes of RAM. So definitely the memory isn’t enough, but new mobile platforms are available now with 2 gigabytes of RAM, for example Nexus 4.

VII. CONCLUSION AND FURTHER WORK

We presented a solution consisting of two virtual Android phones running on top of the VMXL4 microkernel. We discussed other solutions based on various technologies, including paravirtualization on top of L4 microkernels. We presented the hardware and software environment upon which we implemented our solution. We outlined the architecture of the project and the efforts done to virtualize input and display, in continuation of previous work. Finally, we presented the current status and highlighted the main issues and challenges.

In the future, we plan to stabilize our work and improve the overall performance of the system, such that it leads to an improved user experience. This requires an evaluation of the performance of both the VMXL4 microkernel and the Linux kernel, in particular the components in arch/l4 that make the kernel paravirtualizable.

We plan to port VMXL4 and the L4-enhanced Linux kernel to a Qualcomm-based SoC, such that it can run on newer, more performant phones such as the flagship Google LG Nexus 4. This would give us the ability to obtain better memory and time performance, as the Nexus 4 has 2GB of RAM and four cores.
Another step towards the improvement of user experience would be the virtualization of several other components, such as the GPU, sound, USB or the telephony stack. This is a challenge due to the fact that most of these components are managed by proprietary drivers, which makes it more difficult to achieve performant virtualization.

We also aim to port VirtualMetrix’s Performance Management technology to our scenario, after previous work involving integration of Performance Management with the VMXL4 microkernel.

REFERENCES


