Cells: A Virtual Mobile Smartphone Architecture

Jeremy Andrus, Christoffer Dall, Alexander Van't Hof, Oren Laadan, and Jason Nieh
{jeremya, cdall, alexvh, orenl, nieh}@cs.columbia.edu
Department of Computer Science, Columbia University
Technical Report CUCS-022-11
May 2011

Abstract

Cellphones are increasingly ubiquitous, so much so that many users are inconveniently forced to carry multiple cellphones to accommodate work, personal, and geographic mobility needs. We present *Cells*, a virtualization architecture for enabling multiple virtual smartphones to run simultaneously on the same physical cellphone device in a securely isolated manner. *Cells* introduces a usage model of having one foreground virtual phone and multiple background virtual phones. This model enables a new device namespace mechanism and novel device proxies that integrate with lightweight operating system virtualization to efficiently and securely multiplex phone hardware devices across multiple virtual phones while providing native hardware device performance to all applications. Virtual phone features include fully-accelerated graphics for gaming, complete power management features, and full telephony functionality with separately assignable telephone numbers and caller ID support. We have implemented a *Cells* prototype that supports multiple Android virtual phones on the same phone hardware. Our performance results demonstrate that *Cells* imposes only modest runtime and memory overhead, works seamlessly across multiple hardware devices including Google Nexus 1 and Nexus S phones and NVIDIA tablets, and transparently runs all existing Android applications without any modifications.

1 Introduction

The preferred platform for everyday computing needs is shifting from traditional desktop and laptop computers toward mobile smartphone and tablet devices [5]. Smartphones are becoming an even more important work tool for mobile professionals as they often primarily rely on them for telephone, text messaging, and email communication, Web browsing, contact and calendar management, and news, travel-related, and location-specific information. Many of these same functions as well as the ability to play music, movies, and games also make smartphones a useful personal tool. In fact, hundreds of thousands of smartphone applications are available for users to download and try through various online application stores. The ease with which users can download new software imposes a risk on users as malicious software can easily access sensitive data with the risk of corrupting it or even leaking it to third parties [41]. For this reason, companies often lock down the smartphones they allow to connect to the company network, resulting in many users having to carry separate work and personal phones. Application developers also often carry additional phones for development to avoid having a misbehaving application prototype corrupt their primary phone. Parents sometimes wish they had additional phones when they let their children use their smartphones as entertainment devices to play educational games and end up with unexpected charges due to accidental phone calls or unintended in-application purchases.

To address these problems, a few approaches have started leveraging traditional virtual machine mechanisms to enable two separate and isolated instances of the entire software stack of a smartphone to run on the same ARM hardware phone [25, 3, 6, 14]. These approaches require substantial modifications to both user and kernel levels of the software stack, and paravirtualization is used in all cases because ARM is not virtualizable and proposed ARM virtualization extensions are not yet available in hardware. While virtual machines provide important benefits in the context of desktop and server computers, applying these hardware virtualization techniques to smartphones as done by existing approaches has two crucial drawbacks. First, smartphones are much more resource constrained and running an entire additional operating system (OS) and user-space environment in a virtual machine imposes high overhead and limits the number of instances that can run. Second, smartphones incorporate a plethora of devices that applications expect to be able to use, such as GPUs for providing accelerated graphics. Existing approaches provide no effective mechanism for enabling applications to directly leverage these hardware device features from within virtual machines, severely limiting the performance of such applications and making them unusable on a smartphone.

We present *Cells*, a new, lightweight virtualization architecture for enabling multiple virtual phones (VPs) to run simultaneously on the same physical smartphone hardware with high performance and securely isolated from one another. *Cells* does not require running multiple OS instances, but instead uses lightweight OS virtualization to provide virtual namespaces that can run and securely isolate multiple VPs on a single OS instance.

The isolation between VPs ensures that buggy or malicious applications running in one VP cannot adversely impact other VPs. *Cells* provides a novel file system layout based on unioning to maximize sharing of common read-only code and data across VPs, minimizing memory consumption and enabling additional VPs to be instantiated without much overhead.

Because smartphones have small display form factors and are designed to only allow a single application to be visible at a given time, *Cells* takes advantage of this characteristic to introduce a usage model of having one foreground VP that is displayed and multiple background VPs that are not being displayed at any given time. This simple yet powerful model enables *Cells* to provide a novel kernel-level device namespace mechanism to efficiently and securely multiplex phone hardware devices across multiple VPs while providing native hardware device performance to all applications. *Cells* also provides a user-level device namespace proxy mechanism that offers similar functionality for devices such as the baseband processor that are proprietary and entirely closed source. These hybrid mechanisms ensure that the foreground VP is always given direct access to hardware devices. Background VPs are only given shared access to hardware devices when the foreground VP does not require exclusive access. Visible applications are always running in the foreground VP and those applications can take full advantage of any available hardware support, such as hardware-accelerated graphics. Since foreground applications have direct access to hardware, they perform as fast as when they are running natively.

Another key aspect of *Cells* is that it provides full telephony functionality with separately assignable telephone numbers per VP and caller ID support. *Cells* achieves this by integrating a cloud VoIP service to provide individual telephone numbers for each VP without the need for multiple SIM cards. Incoming and outgoing calls are performed in the standard manner on the smartphone using the cellular network, but are routed through the cloud VoIP service as needed to provide both incoming and outgoing caller ID functionality for each VP. In other words, basic telephony is provided using the standard cellular network and standard Android applications, while a VoIP cloud service enables per VP phone numbers and caller ID features.

We have implemented a *Cells* prototype that supports multiple virtual Android phones on the same mobile device hardware. Each VP can be configured the same or completely different from other VPs, and VPs may be running different versions of Android simultaneously. Our prototype has been tested to work with even the most recent openly available version of Android, version 2.3.3. Our performance results demonstrate that *Cells* imposes almost no runtime overhead and only modest memory overhead on benchmarking applications designed to stress the system. *Cells* works seamlessly across multiple hardware devices including the Google Nexus 1 and Nexus S systems as well as an NVIDIA tablet. It is the first virtualization system that fully supports available hardware devices with native performance including GPUs, sensors, cameras, and touchscreens, and transparently runs all Android applications in VPs without any modifications.

2 Usage Model

Cells runs multiple VPs on a single hardware phone. Each VP runs a full standard Android system capable of making phone calls, running standard applications, using data connections, interacting through the touch screen, utilizing the accelerometer and everything else that a user can normally do on the available hardware. Each VP is completely isolated from other VPs and cannot inspect, tamper with, or otherwise access any other VP.

Given the limited size of smartphone screens and the ways in which smartphones are used, *Cells* only allows a single VP to be displayed at any time, referred to as the foreground VP. All other running instances are referred to as background VPs. Background VPs are still running on the system and are capable of receiving system events and performing tasks, but will simply run in the background and not render content on the screen. A user can easily switch between VPs by selecting one of the background VPs to become the foreground one. This can be done using a custom key-combination to cycle through the set of running VPs, or by swiping up and down on the home screen of a VP. Each VP also has a simple application running in it that can be launched to see a list of available VPs, the selection of which switches the respective VP to the foreground. The system can also force a new VP to become the foreground VP as a result of an event, such as an incoming call or text message. For security and convenience reasons, a no-auto-switch can be set to prevent background VPs from being switched to the foreground without explicit user action, depending the type of events. As discussed in Section 3, the foreground-background VP usage model is fundamental to the *Cells* virtualization architecture.

VPs are created and configured on a PC and downloaded to a phone via USB. A VP can be deleted by the user but its configuration is password protected and can only be changed from a PC given the appropriate credentials. For example, a user can create a VP and can decide to later change various options regarding how the VP is run and what devices it can access. On the other hand, an IT administrator can also create a VP that users can download or remove from their phones, but cannot be reconfigured by users. This is useful for companies that may want to distribute locked down VPs.

Each VP can be configured to have different access rights for different devices. For each device, a VP can be configured to have no access, shared access, or exclusive access. Some settings may not be available on certain devices; shared access is for example not available for the framebuffer, since only a single VP is displayed at any time. These per device access settings provide a highly flexible security model and can therefore be used to accommodate a wide range of security policies.

No access means that applications running in the VP cannot access the given device at any time. For example, a VP with no access to the GPS sensor would never be able to track GPS location, despite any user acceptances of application requests to allow location tracking. Users often acquiesce to such privacy invasions because an application will not work without such consent even if the application has no need for such information. By using the no access option, *Cells* enables users to run such applications without compromising privacy.

Shared access to a device means that when the given VP is running in the foreground, other background VPs can access the device at the same time. For example, a foreground VP with shared access to the audio device would allow a background VP with shared access to play music in the background. Exclusive access means that when a given VP is running in the foreground, other background VPs are not allowed to access the device. For example, a foreground VP with exclusive access to the microphone would not allow background VPs to access the microphone, thus preventing applications running in background VPs from eavesdropping on conversations or leaking information. This kind of functionality is essential for supporting secure VPs. Exclusive access may be used in conjunction with the no-auto-switch to ensure that events cannot cause a background VP to move to the foreground and gain access to devices as a means to circumvent the exclusive access rights of another VP.

3 System Architecture

Figure 1 provides an overview of the *Cells* system architecture. We will describe *Cells* using Android since our prototype is based on it. Each VP runs a full stock Android user-space environment. *Cells* leverages lightweight OS virtualization [27, 26, 18, 16] to isolate VPs from one another. *Cells* uses a single operating system kernel across all VPs that virtualizes identifiers, kernel interfaces and hardware resources in such a way that several execution environments can exist side-by-side in virtual OS sandboxes. Each VP has its own private virtual namespace so that VPs can run concurrently and use the same OS resource names inside their respective namespaces, yet be isolated from and not conflict with each other. This is done by transparently remapping OS resource identifiers to virtual ones that are used by processes within each VP. File system paths, process identifiers (PIDs), IPC identifiers, network interface names, and user names (UIDs) must all be virtualized to prevent conflicts and ensure that processes running in a VP cannot see processes in other VPs. The Linux kernel, used by Android, provides virtualization for these identifiers through namespaces [4]. For example, the file system is virtualized using mount namespaces, which allow different independent views on the file system and provide secure private file system jails for VPs.

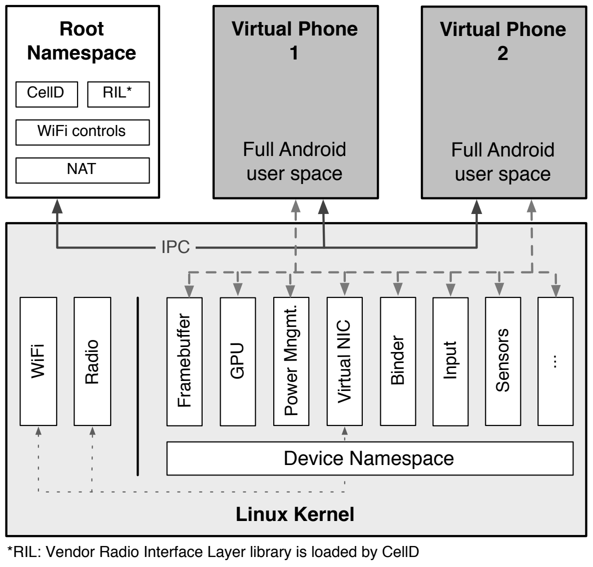


Figure 1: Overview of *Cells* architecture

However, basic OS virtualization is not sufficient to run a complete Android user-space environment. Virtualization mechanisms have primarily been used in headless server environments which support relatively few device interactions, such as networking and storage, which can already be virtualized in commodity OSes such as Linux. However, smartphone applications expect to be able to interact with a plethora of hardware devices, many of which are physically not designed to be multiplexed, and OS device virtualization support is nonexistent for these devices. For Android, at least the devices listed in Table 1 must be fully supported, which include both hardware devices and pseudo devices unique to the Android environment. Three requirements for supporting devices must be met: (1) support exclusive or shared access across VPs, (2) never leak sensitive information between VPs, and (3) prevent malicious applications in a VP from interfering with device usage by other VPs.

To accomplish this, we introduce *device namespaces*, a new kernel-level abstraction that provides secure isolation and efficient hardware resource multiplexing in a manner that is completely transparent to applications. Figure 1 shows how device namespaces are implemented

Device	Description
Alarm*	RTC-based alarms
Audio	Audio output (speakers, headset)
Binder*	IPC framework
Bluetooth	Short range communication
Camera	Graphics input: stills and video
Frame Buffer	Graphics output
GPU	Graphical processing unit
Input	Touch-screen and input buttons
Leds	Backlight and indicator leds
Logger*	Light-weight RAM log driver
LMK*	Low memory killer
Microphone	Audio input (microphone)
Pmem*	Contiguous physical memory
Power*	Power management framework
Sensors	Accelerometer, GPS

Table 1: Android devices
*custom Google drivers

within the overall *Cells* architecture. Unlike PID namespaces in the Linux kernel which virtualize PID identifiers, a device namespace does not virtualize identifiers, but is designed to be used by individual device drivers to tag data structures and to register callback functions. The callback functions can be called when a device namespace changes state. Each VP is given its own unique device namespace for its device usage. *Cells* then leverages its foreground-background VP usage model to register callback functions that are called when the VP changes between foreground and background state. This enables devices to be aware of the VP state and change how it responds to VPs depending on whether it is visible to the user and therefore the foreground VP, or not visible to the user and therefore one of potentially multiple background VPs. The usage model is crucial for enabling *Cells* to virtualize devices efficiently and cleanly.

Cells virtualize existing kernel hardware devices with near zero performance overhead based on three ways for implementing device namespace functionality. One way is to create a device driver wrapper by creating a new device driver for a virtual device, which then multiplexes access and communicates on behalf of applications to the real device driver. The wrapper typically passes through all requests from the foreground VP, and changes the device state and access to the device when a new VP becomes the foreground VP. For example, *Cells* uses a device driver wrapper to virtualize the framebuffer as described in Section 4.1.

A second way is to modify a device subsystem to be aware of device namespaces. For example, the input device subsystem in Linux handles various Android input devices such as touchscreen, navigation wheel, compass, proximity sensor, light sensor, headset input controls, and input buttons. The input subsystem consists of the input core, device drivers, and event handlers, the latter being responsible for passing input events to user-space. By default in Linux, input events are sent to any process that is listening for them, but this does not provide the isolation needed for supporting VPs. To enable the input subsystem to use device namespaces, *Cells* only has to modify the event handlers so that, for each process listening for input events, it first checks if the corresponding device namespace is in the foreground. If it is not, the event is not raised to that specific process. The implementation is simple, and no changes are required to device drivers or the input core. As another example, virtualization of the power management subsystem is described in Section 5.

A third way is to modify a device driver to be aware of device namespaces. For example, Android includes a number of custom pseudo drivers, such as the Binder IPC mechanism. To provide isolation among VPs, *Cells* needs to ensure that under no circumstances can a process in one VP gain access to Binder instances in another VP. This is simply done by modifying the Binder driver so that instead of allowing Binder data structures to reference a single global list of all processes, they first check the device namespace of the calling process and can only reference processes associated with that same device namespace. The namespace context is only initialized when the Binder device file is first opened resulting in almost no overhead for future accesses. While the device driver itself needs to be modified, pseudo device drivers are not hardware-specific and thus changes only need to be made once for all hardware platforms.

In some cases, it may be necessary to modify a hardware-specific device driver to make it aware of device namespaces. For most devices, this is straightforward to do and just involves duplicating necessary driver state on device namespace creation and tagging the data describing that state with the device namespace. Even this can be avoided if the device driver provides some basic capabilities as described in Section 4.2, which discusses GPU virtualization.

In addition to the kernel device namespace abstraction, *Cells* also provides a user-level device namespace proxy mechanism that offers similar functionality for devices such as the baseband processor that are proprietary and entirely closed source. Sections 6 and 7 describe how this user-level proxy approach is used to virtualize telephony and wireless network configuration. *Cells* uses this hybrid combination to virtualize devices to take advantage of the benefits of each mechanism. The kernel-level mechanism provides transparency and performance. The user-level mechanism provides greater portability and a high degree of transparency when the user-space environment provides appropriate interfaces that can be leveraged for virtualization. In the case of proprietary devices with completely closed software stacks, having such an interface is an inherent necessity.

In addition to VPs and device namespaces, Figure 1 also shows the root namespace used by *Cells* to manage VP instances. *Cells* works by booting a minimal init environment in a root namespace which is not visible to the user of a VP and is used to manage individual VPs. The root namespace is considered part of the trusted computing base and processes in the root namespace have full access to the entire file system. The init environment starts a custom process, `CellD`, which manages the starting and switching of VPs between operating in the background or foreground. The device namespaces export an interface to the root namespace through the `/proc` filesystem, which is used to switch the foreground VP and set access permissions for devices. `CellD` also coordinates the configuration of telephony and wireless networking as discussed in Sections 6 and 7. To start a new VP, `CellD` mounts the VP filesystem then clones itself into a new process with separate namespaces and starts the VP's init process to boot up the Android user-space environment. `CellD` also sets up the limited set of IPC sockets accessible to processes in the VP for communicating with the root namespace. These IPC sockets are the only ones that can be used for communicating with the root namespace; all other IPC sockets are internal to the respective VP. To further ensure isolation, `CellD` removes the capability to create device nodes for processes inside the VP to prevent processes from gaining direct access to Linux devices and to outside their confined environment, e.g. by re-mounting block devices. *Cells* also leverages existing Linux kernel frameworks for resource control to prevent resource starvation from a single VP [15].

To enable multiple VPs running the same Android environment to share code and reduce memory usage, each VP is given read-only access to the same base Android file system. To provide a read-write file system view for a VP, file system unioning [38] is used to join the read-only base Android file system with a writable file system layer by stacking the latter on top of the former. This creates a unioned view of the two: files system objects, namely files and directories, from the writable layer are always visible, while objects from the read-only layer are only visible if no corresponding object exists in the other layer. When a new VP is started, *Cells* also enables Linux Kernel Samepage Merging (KSM) for a short time to further reduce memory usage by finding anonymous memory pages used by the base Android system that have the same contents, then arranging for one copy to be shared among the various VPs [36]. *Cells* also leverages the unique Android low memory killer technology to increase the total number of VPs it is possible to run on a device without sacrificing functionality. This Linux kernel driver is a more flexible replacement for the standard Linux OOM killer. Instead of randomly choosing processes to kill when the system runs out of RAM, the Android low memory killer uses heuristics to prioritize background and inactive processes consuming large amounts of RAM. Android starts these processes purely as an optimization to reduce application startup-time, so these processes can be killed and restarted without any loss of functionality. Critical system processes are never chosen to be killed, and if the user requires the services of a background process which was killed, the process is simply restarted (possibly killing other unused background processes).

4 Graphics

Starting with graphics, we now describe in detail the virtualization of several key Android devices. The display and its graphics hardware is one of the most important devices in modern smartphones. Applications expect to be able to take full advantage of any hardware display acceleration or GPU available on the smartphone. Android relies on a standard Linux framebuffer which provides an abstraction to a physical display, including screen memory, memory dedicated to and controlled exclusively by the display device. The framebuffer allows applications to map screen memory, and the GPU hardware also maps screen memory, so that it can be written to directly for performance reasons. The memory-mapped use of screen memory and the performance requirements of the graphics subsystem make it one of the more challenging devices to virtualize in a mobile phone.

4.1 Framebuffer

To virtualize the framebuffer to support multiple VPs, *Cells* leverages device namespaces and its foreground-background usage model to provide a multiplexing framebuffer, a new framebuffer driver called `mux_fb` that serves as a device driver wrapper for the hardware framebuffer. `mux_fb` is registered as a standard framebuffer device and multiplexes access to a single physical framebuffer device. The foreground VP is given exclusive access to the screen memory and display hardware while each background VP maintains virtual hardware state and renders any output to a virtual screen memory buffer in system RAM, referred to as the backing buffer. VP access to the `mux_fb` driver is isolated through its device namespace, such that a unique virtual device state and backing buffer is associated with each VP. `mux_fb` currently supports multiplexing a single physical frame buffer device, but more complicated multiplexing schemes involving multiple physical devices could be accomplished in a similar manner.

In Linux, the basic framebuffer usage pattern involves three types of accesses: `mmaps`, standard control `ioctls`, and custom `ioctls`. When a process `mmaps` an open framebuffer device file, the framebuffer driver is expected to map its associated screen memory into the process' address space allowing the process to render directly on the display. A process controls and configures the framebuffer hardware state through a set of standard control `ioctls` defined by the Linux framebuffer interface, which, for example, can change the pixel format. Each framebuffer device may also define custom `ioctls` which can be used to perform accelerated drawing or rendering operations.

Cells passes all accesses to the `mux_fb` device from the foreground VP directly to the hardware back end. This includes control `ioctls` as well as custom `ioctls`, allowing applications to take full advantage of any custom `ioctls` implemented by the physical device driver that may be used, for example, for accelerated graphics. When an application running in the foreground VP `mmaps` an open `mux_fb` device, the `mux_fb` driver simply maps the physical screen memory provided by the hardware back end. This creates the same zero-overhead pass-through to the screen memory as on native systems.

Cells does not pass any accesses to the `mux_fb` driver from background VPs to the hardware back end, ensuring that the foreground VP has exclusive hardware access. Standard control `ioctls` are applied to virtual hardware state maintained in RAM. Custom `ioctls`, by definition, perform non-standard functions such as graphics acceleration or memory allocation, and therefore accesses to these functions from background VPs must be at least partially handled by the same kernel driver which defined them. Instead of passing the `ioctl` to the hardware driver, *Cells* uses a new notification API that allows the original driver to appropriately virtualize the access. If the new driver does not register for this new notification then *Cells* will return an error code when the custom `ioctl` is called from a background VP. In our experience, returning an error code was sufficient for all systems tested except an NVDIA development tablet. When an application running in a background VP `mmaps` the framebuffer device, the `mux_fb` driver will map its backing buffer into the process' virtual address space.

Switching the display from a foreground VP to a background VP is accomplished in four steps, all of which must occur before any additional framebuffer operations are performed: (1) screen memory remapping, (2) screen memory deep copy, (3) hardware state synchronization, and (4) GPU coordination. Screen memory remapping is done by altering the page table entries for each process which has mapped framebuffer screen memory to redirect virtual addresses in each process to different physical locations. Processes running in the VP which is to be moved into the background have their virtual addresses remapped to backing memory in system RAM, and processes running in the VP which is to become the foreground VP have their virtual addresses remapped to the physical screen memory. Screen memory deep copy is done by copying the contents of the screen memory into the previous foreground VP's backing buffer and copying the contents of the new foreground VP's backing buffer into screen memory. This copy is not strictly necessary if the new foreground VP completely redraws the screen. Hardware state synchronization is done by saving the current hardware state into the virtual state of the previous foreground VP and passing the virtual hardware state in the previous background VP into the hardware. Because the display device only uses the current hardware state to output the screen memory, there is no need to correlate particular drawing updates with individual standard control `ioctls`; only the accumulated virtual hardware state is needed. GPU coordination involves notifying the GPU of the memory address switch so that it can update any internal graphics memory mappings. This is discussed in more detail in Section 4.2.

To better scale the *Cells* framebuffer virtualization, the backing buffer in system RAM could be reduced to a single memory page which is mapped into the entire screen memory address region of background VPs. This optimization not only saves memory, but also eliminates the need for the screen memory deep copy. However, it does require the VP's user-space environment to redraw the entire screen when it becomes the foreground VP. The redraw overhead is minimal. Android conveniently provides this redraw functionality through the `fbearlysuspend` driver discussed in Section 5.1.

4.2 GPU

Cells virtualizes the GPU by leveraging the GPU's independent graphics contexts [28] together with the framebuffer virtualization of screen memory described in Section 4.1. Each VP is given direct pass-through access to the GPU device. Because each process which uses the GPU executes graphics commands in its own context, processes are already isolated from another and there is no need for further VP GPU isolation. The key challenge is that each VP requires framebuffer screen memory on which to compose the final scene to be displayed, and in general the GPU driver will request and use this memory from within the OS kernel.

To address this problem, *Cells* again leverages its foreground-background usage model to provide a virtualization solution similar to how framebuffer screen memory remapping is done. The foreground VP will use the GPU to render directly into screen memory. Background VPs which use the GPU will render into their respective backing buffers. When the foreground VP is switched, the GPU driver must locate all GPU addresses which are mapped to the physical screen memory as well as the new foreground VP's backing buffer in system RAM. It must then remap those GPU addresses to point to the new backing buffer and the physical screen memory, respectively. To accomplish this remapping, *Cells* provides a callback interface from the `mux_fb` driver which provides source and destination physical addresses on each foreground VP switch. Implementing this virtualization technique for the Nexus 1 and Nexus S phones required adding 329 out of 5,550 lines of code to the *Adreno* GPU driver, which is used in the Nexus 1, and adding 835 out of 48,042 lines of code to the *PowerVR* GPU driver, which is used in the Nexus S.

While this technique necessitates a certain level of access to the GPU driver, it does not preclude the possibility of using a proprietary driver so long as it exposes three basic capabilities. First, it should provide the ability to remap GPU linear addresses to specified physical addresses as required by the virtualization mechanism. Second, it should provide the ability to safely reinitialize the GPU device or ignore re-initialization attempts, as each VP running a stock Android instance will attempt to do GPU initialization on startup. Third, it should provide the ability to ignore device power management and other non-graphics related hardware state updates, to make it possible to ignore such updates from an Android instance running in a background VP. Some of these capabilities were already available on the *Adreno* GPU driver, but not all. The small number of lines of code added to the *Adreno* and *PowerVR* GPU drivers were precisely to add these three capabilities.

While most modern GPUs include an MMU, there are some devices which require memory used by the GPU to be physically contiguous. For example, the *Adreno* GPU can selectively disable the use of the MMU. For *Cells* GPU virtualization to work under these conditions, the backing memory in system RAM must be physically contiguous. This can be done by allocating the backing memory either with `kmalloc`, or using an alternate physical memory allocator such as Google's `pmem` driver, Samsung's `s3c_mem` driver or NVIDIA's `nvmap` driver.

5 Power Management

To provide *Cells* users the same power management experience that they have when using non-virtualized phones, we apply two simple virtualization principles: (1) background VPs should not be able to put the device into a low power mode, and (2) background VPs should not prevent the foreground VP from putting the device into a low power mode. We apply these principles to Android's custom power management, which is based on the premise that a mobile phone's preferred state should be suspended. Android introduces three interfaces which attempt to extend the battery life of mobile devices through extremely aggressive power management: *early suspend*,*fbearlysuspend*, and *wake locks*, also known as suspend blockers.

The *early suspend* framework is an ordered callback interface which allows drivers to receive notifications just before the device is suspended and after it resumes. *Cells* virtualizes the early suspend framework by simply disallowing background VPs from initiating a suspend operation. The remaining two Android-specific power management interfaces present more unique challenges and offer insights into aggressive power management virtualization.

5.1 Frame Buffer Early Suspend

The *fbearlysuspend* driver exports display device suspend and resume state into user-space. This allows user space to stop all processes which use the display when the display is powered off and redraw the screen after the display is powered on. Power is saved because the overall device workload is lower, and devices such as the GPU may be powered down or made quiescent. Android implements this functionality with two `sysfs` files, `/sys/power/wait_for_fb_sleep` and `/sys/power/wait_for_fb_wake`. A user process can open and read from one of these files, and the read will block until the frame buffer device is either asleep or awake, respectively.

Cells virtualizes *fbearlysuspend* by making the *fbearlysuspend* driver namespace-aware, leveraging the device namespace and foreground-background usage model. Reads from the foreground VP function exactly as a non-virtualized system while reads from a background VP always report the device as sleeping. When the foreground VP switches, all processes which are blocked on either of the two files are unblocked, and the return values from the read calls are based on the new state of the VP in which the process is running. Processes in the new foreground VP will see the display as awake (on), processes in the formerly foreground VP will see the display as asleep, and processes which are running in background VPs that remain in the background will continue to see the display as asleep. This forces background VPs to pause drawing or rendering, resulting in three benefits. First, it reduces the overall system load by reducing the number of processes which are using the hardware drawing resources. Second, it increases throughput in the foreground VP by ensuring that its processes are the only ones using the drawing resources. Third, it reduces power consumption and minimizes the power footprint of multiple VPs running simultaneously on a mobile phone.

5.2 Wake Locks

Wake locks are a controversial [39] Android power management feature. A *wake lock* is a special kind of kernel reference counter with two states: *active* and *inactive*. When a wake lock is "locked" its state is changed to active; conversely when the lock is "unlocked," its state is changed to inactive. A wake lock can be locked multiple times, but only requires a single unlock to put the lock into the inactive state. The Android system will not enter either suspend, or any other low power mode until all wake locks are in the inactive state.

To further complicate the matter, wake locks can be created statically at compile time, dynamically by kernel drivers or dynamically by user-space. Wake locks can also be locked and unlocked from user context, kernel context (from work queues), and interrupt context (from IRQ handlers) in a completely orthogonal manner. This creates an extremely distributed power management paradigm, and makes determining the VP to which a wake lock belongs when it is locked or unlocked a non-trivial task. Consequently, preventing a background VP from suspending the device or interfering with the foreground VP's ability to suspend the device becomes a challenging virtualization problem.

Cells virtualizes wake locks by leveraging the device namespace and foreground-background usage model to maintain both the kernel and user-space wake lock interfaces as well as adhere to the two virtualization principles specified above. The solution is predicated on three assumptions. First, all kernel and interrupt context locking and unlocking coordination as well as any user-level locking and unlocking coordination in the trusted root namespace were correct and appropriate before virtualization. Second, we "trust" the kernel and its drivers; when lock or unlock is called from interrupt context, we perform the operation unconditionally. Third, the foreground VP maintains full control of the hardware.

Under these assumptions, *Cells* solves the wake lock virtualization problem by allowing multiple device namespaces to lock and unlock the same wake lock orthogonally, and only starting power management operations based on the set of locks associated with the foreground VP. The solution can be summarized in the following set of rules:

1. When a wake lock is locked, keep a namespace "token" associated with the lock which indicates the context in which the lock was taken. A wake lock token may contain references to multiple namespaces if the lock was taken from those namespaces.
2. When a wake lock is unlocked from user context, remove the associated namespace token.
3. When a wake lock is unlocked from interrupt context or the root namespace, remove *all* lock tokens. This follows from the second assumption.
4. After a user context lock or unlock operation, adjust any suspend timeout value based only on the locks active in the current device namespace.
5. After a root namespace lock or unlock operation, adjust the suspend timeout based on the foreground VP's device namespace.
6. When the foreground device namespace changes, reset the suspend timeout based on the locks acquired in the new namespace. This involves keeping per-namespace timeout values with each token associated with a given wake lock.

One additional piece of infrastructure was necessary to implement the *Cells* wake lock virtualization mechanism. The set of rules given above implicitly assumes that, aside from interrupt context, the lock and unlock functions are aware of the device namespace in which the operation is being performed. While this is true for operations started from user context, it is not the case for operations performed from kernel work queues. To address this issue, we introduced a mechanism which forces a kernel work queue into a specific device namespace.

6 Telephony

Cells provides each VP with its own separate telephony functionality so that it can have its own call logs, list of recently dialed and received telephone numbers, and separate phone number. We first describe how *Cells* virtualizes the radio stack to provide telephony isolation among VPs, then we discuss how multiple phone numbers can be provided on a single hardware phone.

6.1 RIL Proxy

The Android telephony subsystem is designed to be easily ported by phone vendors to their specific hardware devices. The Android phone application uses a set of Java libraries and services that handle the telephony state and settings, such as displaying current radio strength in the status bar and menus to select, for instance, roaming options. The phone application, the libraries and the services all communicate via Binder IPC with the Radio Interface Layer (RIL) Daemon (RilD). RilD dynamically links with a library provided by the phone hardware vendor which in turn communicates with kernel drivers and the radio baseband system. The left side of Figure 2 shows a representation of the standard Android telephony system.

The entire radio baseband system is proprietary and closed source, starting from the user-level RIL vendor library down to the physically separate hardware baseband processor. Details of the vendor library implementation and its communication with the baseband are well-guarded secrets. Each hardware phone vendor provides its own proprietary radio stack. Since the stack is a complete black box, it would be difficult if not impossible to intercept, replicate, or virtualize any aspect of this system in the kernel without direct hardware vendor support. Furthermore, the vendor library is designed to be used by only a single RilD and the radio stack as a whole is not designed to be multiplexed.

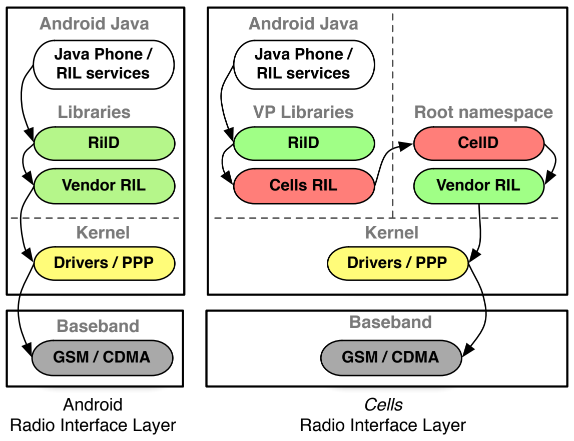


Figure 2: Android Radio Interface Layer for *Cells*

As a result of these constraints, *Cells* virtualizes telephony using a different user-space approach designed to work transparently with the black box radio stack. Each VP has the standard Android telephony Java libraries and services and its own stock RilD, but rather than having RilD communicate directly with the hardware vendor provided RIL library, *Cells* provides its own proxy RIL library in each VP. The proxy RIL library is loaded by RilD in each VP and connects to CellD running in the root namespace. CellD in turn communicates with the vendor library to use the proprietary radio stack. Since there can be only one radio stack, CellD loads the vendor RIL library on system startup and multiplexes access to it. The right side of Figure 2 shows a representation of the *Cells* Android telephony system. We refer to the proxy RIL library together with CellD as the *RIL proxy*. This approach has three key advantages. First, no hardware vendor support is required since it treats the radio stack as a black box. Second, it works with a stock Android environment by leveraging the fact that Android already does not provide its own RIL library but instead relies on it being supplied by the system on which it will be used. Third, it operates at a well-defined interface, making it possible to understand exactly how communication is done between RilD and the RIL library it uses.

Cells leverages its foreground-background VP model to enable the necessary multiplexing of the radio stack. Since the user can only make calls from the foreground VP, because only its user interface is displayed, CellD allows only the foreground VP to make calls. All other forms of multiplexing are done in response to incoming requests from the radio stack through CellD. CellD operates in the same manner as Android's RilD in how it interfaces with the vendor RIL library, and can therefore provide all of the standard call multiplexing available in Android for handling incoming calls. For example, to place the current call in the foreground VP on hold while answering another call to a background VP, CellD issues the same set of standard GSM commands that RilD would issue to the radio stack.

The RIL proxy needs to support the two classes of function calls defined by the RIL, *solicited calls* which pass from RilD to the RIL library, and *unsolicited calls* which pass from the RIL library to RilD. The interface is relatively simple, as there are only four defined solicited function calls and two defined unsolicited function calls, though there are a number of possible arguments. Both the solicited requests and the responses carry structured data in their arguments. The structured data can contain pointers to nested data structures and arrays of pointers. The main complexity in implementing the RIL proxy is dealing with the implementation assumption in Android that the RIL vendor library is normally loaded in the RilD process so that pointers can be passed between the RIL library and RilD without further processing. In *Cells*, the RIL vendor library is loaded in the CellD process but not the RilD process. Instead, the RIL proxy passes the arguments over a standard Unix Domain socket so all data must be thoroughly packed and unpacked on either side.

The basic functionality of the RIL proxy is to pass requests sent from within a VP unmodified to the vendor RIL library and to forward unsolicited calls from the vendor RIL library unmodified to RilD inside a VP. To support the *Cells* security model, the CellD component of the proxy will filter requests as needed to disable telephony functionality for VPs that are configured not to have telephony access. However, even in the absence of such VP configurations, some solicited requests must be filtered from background VPs and some calls require special handling to properly support our foreground-background model and provide working telephony isolation functionality. Filtering and special handling are done by the `CellD` component of the proxy for security reasons. The commands that require filtering or special handling can be loosely categorized as those involving the foreground VP, initialization, signal strength, and incoming calls.

Dial Request represents outgoing calls, *Set Screen State* is used to suppress certain notifications like signal strength, and *Set Radio State* is used to turn the radio on or off. These operations are only allowed from the foreground VP and filtered from all background VPs to ensure that the only the foreground VP can make calls and background VPs do not interfere with that functionality. These restrictions are implemented in CellD by simply examining the requests from each VP and returning an error code to background VPs.

SIM I/O requests are used amongst other things to initialize the radio when turning on the device or turning off airplane mode, and to query SIM information such as the *IMSI*. When a second VP starts up, it will try to re-initialize the radio, which is problematic as the radio is already initialized. CellD addresses this problem by keeping a log of *SIM I/O* requests and corresponding responses, including recorded data from when the foreground VP initialized the radio. This sequence is replayed in order upon requests from background VPs. When the radio is turned off, the log is cleared, and the first foreground VP to turn on the radio will be allowed to do so, causing CellD to start recording a new log. Since a VP can also inquire about the *radio state*, and this is inherently tied to the initialization process, CellD also records the radio state between each *SIM I/O* operation and replays these states for background VPs. The result is that background VPs successfully start up, capable of receiving phone calls and displaying the network status.

Signal Strength is an unsolicited notification about the current signal strength generated by the vendor library and sent to RilD. In *Cells*, this notification is received by CellD and forwarded across the sockets to the RIL proxy library and thereby RilD inside a VP. Since all VPs with access to telephony functionality should know the signal strength, this is sent unmodified to all such VPs, with one important exception. During initialization, the VP cannot be notified of a signal strength, since that would indicate an already initialized radio, which is unexpected from the point of view of the starting VP.

Call State Changed, *Call Ring* and *Get Current Calls* are all related to incoming calls. When an incoming call occurs, the vendor library will first send a *Call State Changed* notification, followed by a number of *Call Ring* notifications for as long as the call is pending. CellD inspects each notification and decides to which VP it should forward the notification, but this is complicated by the fact that these notifications are not associated with a phone number. Therefore, CellD holds these notifications and issues a *Get Current Calls*, mirroring functionality typically done by RilD, to receive a list of all incoming and active calls and determines the VP that is being called based on tagging information in the incoming caller ID as discussed in Section 6.2. Once the VP has been determined, CellD replays the notifications to that VP. Any VP can issue a *Get Current Calls* request, in which case CellD inspects the data returned from the vendor library and allows only calls made to the requesting VP or calls initiated from the requesting VP to exist in the returned list.

6.2 Multiple Phone Numbers

While some hardware phones support multiple SIM cards, which makes supporting multiple phone numbers straightforward, most phones do not provide this feature. Since mobile network operators do not generally offer multiple phone numbers per SIM card or CDMA phone, we offer an alternative system to provide a distinct phone number for each VP on existing unmodified single SIM card phones which dominate the marketplace. Our approach is based on pairing *Cells* with a VoIP cloud service that enables telephony with the standard cellular network and standard Android applications, but with separate phone numbers.

The *Cells* VoIP cloud service consists of a VoIP server which registers a pool of subscriber numbers. Numbers from this pool are paired with the actual mobile phone number of the user's hardware phone and when those numbers are dialed, the VoIP server receives those calls. The VoIP server then calls the user's actual phone number, but replaces the caller ID with the one that called the VoIP server. The VoIP server further appends a digit to the caller ID, which is used to designated the VP to which the call should be delivered. When CellD receives the incoming call list, it checks the last digit of the caller ID and chooses a VP based on that digit. *Cells* allows users to configure which VP should handle which digit through the cloud service interface. CellD strips the appended digit before forwarding the information to the receiving VP, causing the caller ID to be correctly presented to the user. If the VP is not available, the VoIP cloud service will direct the incoming call to the server-provided voice mail. We currently use a single digit scheme supporting a maximum of ten selectable VPs, which should be more than sufficient for any user. It is certainly possible to spoof caller ID as is currently the case on existing hardware phones; in the worst case, this would simply appear to be a case of dialing the wrong phone number.

Our VoIP cloud service is implemented using an Asterisk server [2] as it provides unique functionality not available through other commercial voice services. For example, Google Voice cannot provide this functionality because, although it can forward multiple phone numbers to the same land line, it does not provide this capability for mobile phone numbers [10].

The caller ID of outgoing calls should also be replaced with the phone number of the VP that actually makes the outgoing call instead of the hardware phone's actual mobile phone number. Unfortunately, the GSM standard does not have any facility to *change* the caller ID, only to either enable or disable showing the caller ID. Therefore, if the VP is configured to display outgoing caller IDs, *Cells* ensures that they are correctly sent by routing those calls through the VoIP server. CellD intercepts the *Dial Request*, dials the VoIP number associated with the dialing VP, and passes the actual number to be dialed via DTMF tones. The VoIP server interprets the tones, dials the requested number, and connects the call.

7 Networking

Mobile phones are most commonly equipped with an IEEE 802.11 wireless LAN adapter and cellular data connectivity through either a GSM or CDMA network. Each VP that has network access must be able to use either WLAN or mobile data depending on what is available to the user at any given location. At the same time, each VP must be completely isolated from other VPs. *Cells* takes a hybrid approach to virtualize networking and to provide necessary isolation and functionality, including virtualizing core network resources and uniquely virtualizing wireless configuration management.

Cells leverages previous work [32, 33] in virtualizing core network resources such as network adapters, routing tables, IP addresses, and port numbers, which has been largely built-in to recent versions of the Linux kernel in the form of network namespaces [4]. Virtual identifiers are provided in VPs for all network resources, which are then translated into real physical identifiers. Real network devices representing the WLAN or cellular data connection are not visible within a VP. Instead, a virtual Ethernet pair is setup from the root namespace where one end of the Ethernet pair is present inside a VP and the other end in the root namespace. The kernel is then configured to perform Network Address Translation (NAT) between the active public interface (either WLAN or cellular data) and the VP end of an Ethernet pair. Each VP is then free to bind to any socket address and port without conflicting with other VPs. *Cells* uses NAT as opposed to bridged networking since bridging is not supported on cellular data connections and is also not guaranteed to work on WLAN connections. Note that since each VP has its own virtualized network resources, network security mechanisms are isolated among VPs. For example, VPN access to a corporate network from one VP cannot be used by another VP.

However, virtualizing core network resources is not enough in the context of mobile phones, which have WLAN and cellular data connections that introduce a new challenge. These types of connections require configuration by the user, and that configuration changes as the user moves around with the mobile phone. There exists little if any support for virtualizing WLAN or cellular data configuration. Current best practice is embodied in desktop virtualization products such as VMware Workstation [35] which create a virtual wired Ethernet adapter inside a virtual machine but leave the configuration on host system. This model does not work on a mobile phone where no such host system is available and a VP is the primary system used by the user. VPs rely heavily on network status notifications reflecting a network configuration that can frequently change, making it essential for wireless configuration and status notifications to be virtualized and made available to each VP.

Virtualizing wireless configuration management requires different mechanisms from virtualizing core network resources. Configuration management is highly device-specific in terms of the exact operations that are used for configuration. A user-level library called `wpa_supplicant` with support for a large number of devices is typically used to issue various `ioctls` and netlink socket options that are unique to each device. Unlike virtualizing core network resources which are general and well-defined, virtualizing wireless configuration in the kernel would involve essentially emulating the device-specific understanding of configuration management, which is error-prone and complicated.

To address this problem, *Cells* leverages its foreground-background VP model to decouple wireless configuration from the actual network interfaces. A configuration proxy is introduced to replace the user-level WLAN configuration library and RIL libraries inside each VP. The proxy communicates with CellD running in the root namespace, which in turn communicates with the user-level library for configuring the Wi-Fi or cellular data connections. Assuming the default case when all VPs are allowed network access, `CellD` basically forwards all configuration requests from the foreground VP proxy to the user-level library, but ignores configuration requests from background VP proxies that would adversely affect the foreground VP's network access. This approach is minimally intrusive since user-space phone environments, such as Android, are already designed to run on multiple hardware platforms and therefore cleanly interfaces with user-space configuration libraries.

For virtualizing Wi-Fi configuration management, *Cells* replaces `wpa_supplicant` inside each VP with a thin Wi-Fi proxy. It is simple to virtualize `wpa_supplicant`, as Android communicates with it over a well-defined socket interface. The Wi-Fi proxy communicates with CellD running in the root namespace, which in turn starts and communicates with `wpa_supplicant` as needed on behalf of individual VPs. The protocol used by the Wi-Fi proxy and CellD is quite simple, as the standard interface to `wpa_supplicant` consists of only eight function calls each with text-based arguments. The protocol just sends the function number, a length of the following message, and the message data itself. Replies are similar, but also contain an integer return value in addition to data. CellD ensures that background VPs cannot interfere with the operation of the foreground VP. For instance, if a foreground VP is connected to a Wi-Fi network and a background VP attempts to disable the Wi-Fi access, the request from the background VP is ignored. At the same time, inquiries that do not change state or divulge sensitive information sent from background VPs, such as requesting the current signal strength, are typically processed since applications such as email clients inside background VPs may use this information to decide whether to check for new email.

For virtualizing cellular data connection management, *Cells* replaces the RIL vendor library as described in Section 6, which is responsible for establishing cellular data connections. As in the case of Wi-Fi, CellD ensures that background VPs cannot interfere with the operation of the foreground VP. Similarly, innocuous inquiries from background VPs that have network access, such as the status of the data connection (connected, Edge, 3G, HSPDA, etc.) or signal strength, are processed and reported back to the VPs.

8 Experimental Results

We have implemented an Android *Cells* prototype and demonstrated its functionality across different Android devices, including Google Nexus 1 [8] and Nexus S [9] phones, and NVIDIA Ventana development tablets [23]. In running multiple VPs on a phone or tablet, there is no user noticeable performance difference between running in a VP and running natively on the phone. For example, while running 4 VPs on a Nexus 1 device, we simultaneously played the popular game *Angry Birds* [31] in one VP, raced around a dirt track in the *Reckless Racing* [29] game on a second VP, crunched some numbers in a spreadsheet using the *Office Suite Pro* [21] application in a third VP, and listened to some music using the Android music player in the fourth VP. Using *Cells* we were able to deliver native 3D acceleration to both game instances while seamlessly switch between and interacting with all four running VPs.

8.1 Methodology

We further quantitatively measured the performance of our unoptimized prototype running a wide range of applications in multiple VPs. Most of our measurements were obtained using Nexus 1 (Qualcomm 1GHz QSD8250 + Adreno 200 GPU, 512 MB RAM) and Nexus S (Samsung Hummingbird 1GHz Cortex A8 + PowerVR GPU, 512 MB RAM) phones. The Nexus 1 uses an SD card for storage for some of the applications; we used a *Patriot Memory* class 10 16 GB SD card. Also, due to space constraints on the Nexus 1 flash device, all Android system files for all *Cells* configurations were stored on, and run from, the SD card. We also obtained some measurements using an NVIDIA Ventana development tablet (NVIDIA Tegra 2 dual 1GHz ARM Cortex A9 + GeForce GPU, 1 GB of RAM).

The *Cells* implementation used for our measurements was based on then Android Open Source Project (AOSP) version 2.3.3. Aufs version 2.1 was used for file system unioning [24]. A single read-only branch of a union file system was used as the `/system` and `/data` partitions of each VP. This saves megabytes of file system cache while maintaining isolation between VPs through separate writeable branches i.e. when one VP modified a file in the read-only branch, the modification would be stored in its own private write branch of the file system. The implementation enables the Linux KSM driver for a period of time when a VP is booted. To maximize the benefit of KSM, `CellD` made a custom system call which added all memory pages from all processes to the set of pages KSM attempts to merge. While this potentially maximizes shared pages, the processing overhead required to hash and check all memory pages from all processes quickly outweighs the benefit. Therefore, we monitored the KSM statistics through the procfs interface and disabled shared page merging after the page merging rate dropped below a pre-determined threshold.

We present measurements along three dimensions of performance: runtime overhead, power consumption, and memory usage. To measure runtime overhead, we compared the performance of various applications running with *Cells* versus running the applications on the latest manufacturer stock image available for the respective mobile devices. We measured the performance of *Cells* when running 1 VP (1-VP), 2 VPs (2-VP), 3 VPs (3-VP), 4 VPs (4-VP), and 5 VPs (5-VP), each with a fully booted Android environment running all applications and system services available in such an environment. Since AOSP v2.3.3 was used as the origin for the *Cells* Android version used in our experiments, we also measured the performance of a baseline system (Baseline) created by downloading AOSP v2.3.3 source, compiling, and installing it without modification.

We measured runtime overhead in two scenarios, one with a benchmark application designed to stress some aspect of the system, and the other with the same application running, but simultaneously with an additional background workload. For *Cells*, the benchmark application was always run in the foreground VP and if a background workload was used, it was run in a background VP when multiple VPs were used. For the benchmark application, we ran one of six Android applications designed to measure different aspects of performance: CPU using Linpack for Android v1.1.7; 2D graphics and file I/O using Quadrant Advanced Edition v1.1.1 (2D, I/O); 3D graphics using Neocore by Qualcomm; web browsing using the popular SunSpider v0.9.1 JavaScript benchmark; and networking using the `wget` module in a cross-compiled version of BusyBox v1.8.1 to download a single 400 MB file from a dedicated Samsung *nb30* laptop. The laptop was running Windows 7, providing a WPA wireless access point via its Atheros AR9285 chipset and built-in Windows 7 SoftAP [20] functionality, and serving up the file through the HFS [19] file server v2.2f. To minimize network variability, a location with minimal external Wi-Fi network interference was chosen. Each Android phone was used from this same location for all experiments, and was connected to the same 802.11g laptop access point. For the background workload, we played a music file from local storage in a loop using the standard Android music player. All results were normalized against the performance of the manufacturer's stock configuration without the background workload.

To measure power consumption, we compared the power consumption of the latest manufacturer stock image available for the respective mobile devices against that of Baseline and *Cells* in 1-VP, 2-VP, 3-VP, 4-VP, and 5-VP configurations. We measured two different scenarios. In the first power scenario, the device configuration under test was fully booted, i.e. all VPs started up and KSM had stopped merging pages, and then the device was left idle for 12 hours. During the idle period, the device would normally enter a low power state, preventing intermediate measurements. However, occasionally the device would wake up to service timers and Android system alarms, and during this time we would take a measurement of the remaining battery capacity. At the end of 12 hours we took additional measurements of capacity. In the second power scenario, the device configuration under test was fully booted, then the Android music player was started. In multiple VP configurations, the music player ran in the foreground VP in order to prevent the device from entering a low power state. The music player was set to repeat the same song continuously and did so for four hours. During this time we sampled the remaining battery capacity every 10 seconds. To measure power consumption due to *Cells* and avoid having those measurements completely eclipsed by WiFi, cellular, and display power consumption, we disabled Wi-Fi and cellular communication, and turned off the display back-light for these experiments.

To measure memory usage, we recorded the amount of memory used for the Baseline and *Cells* in 1-VP, 2-VP, 3-VP, 4-VP, and 5-VP configurations. We measured four different scenarios. First, we ran a full Android environment without launching any additional applications other than those that are launched by default on system bootup (NoApps). Second, we ran the first scenario plus the Android web browser (Browser). Third, we ran the second scenario plus the Android email client (Browser+Email). Finally, we ran the third scenario plus the Android calendar application. In each scenario, an instance of every application was running in all background VPs as well as the foreground VP. We also performed the same measurements of memory usage on the NVIDIA tablet while scaling up to 9 and 10 VPs (9-VP and 10-VP). This illustrates the scalability of the system on more powerful tablet devices.

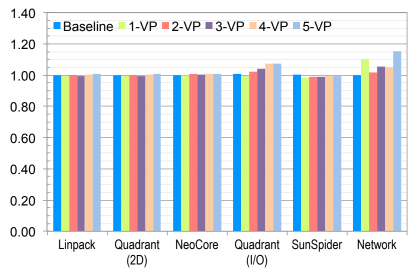
8.2 Measurements

Figures 3a to 3i show the measurement results. These are the first measurements that we are aware of for running multiple Android instances on a single phone or tablet. In all experiments, Baseline and stock measurements were within 1% of each other, so only Baseline results are shown.

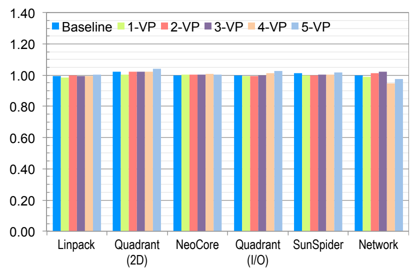
Figures 3a and 3b show the runtime overhead on the Nexus 1 and Nexus S, respectively, for each of the benchmark applications with no additional background workload. *Cells* runtime overhead was small in all cases, even with up to 5 VPs running at the same time. *Cells* incurs less than than 1% overhead in all cases on the Nexus 1 except for Network and Quadrant I/O, and less than 4% overhead in all cases on the Nexus S. The Quadrant 2D and Neocore measurements show that *Cells* is the first system that can deliver fully-accelerated graphics performance in virtual mobile devices. Quadrant I/O on the Nexus 1 has less than 7% overhead in all cases, though the 4-VP and 5-VP measurements have more overhead than the configurations with fewer VPs. This is likely due to the use of the slower SD card on the Nexus 1 for this benchmark instead of internal flash memory on the Nexus S coupled with the presence of I/O system background processes running in each VP.

The Network runtime overhead measurements show the highest overhead on the Nexus 1 and the least overhead on the Nexus S. The measurements shown are averaged across ten experiments per configuration. The differences here are not reflective of any significant differences in performance as much as the fact that the results of this benchmark were highly variable; the variance in the results for any one configuration was much higher than any differences across configurations. Despite our best efforts, the variability was extremely difficult to control. The Wi-Fi connection would occasionally drop out, was subject to varying interference levels based on time of day, and was sensitive to the distance and orientation between the device and the laptop providing the Wi-Fi hot spot. While more extensive testing in a tightly controlled environment would provide increasingly stable numbers, any overhead introduced by *Cells* was consistently below Wi-Fi variability levels observed on the manufacturer's stock system and would not be noticeable by a user.

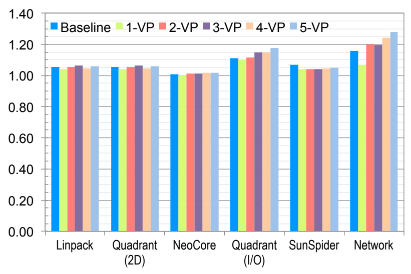
Figures 3c and 3d show the runtime overhead on the Nexus 1 and Nexus S, respectively, for each of the benchmark applications while running the additional background music player workload. Note that all results are



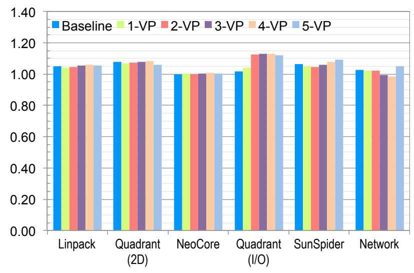
(a) Normalized Nexus 1 results



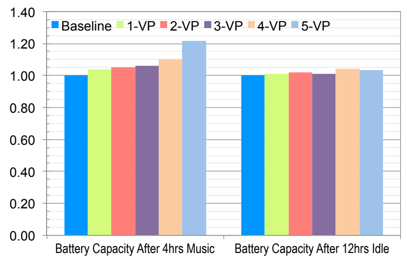
(b) Normalized Nexus S results



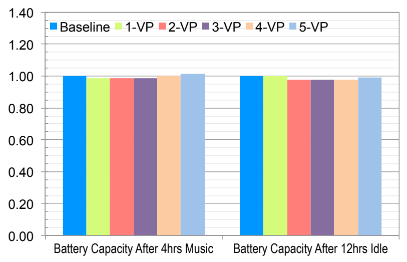
(c) Normalized Nexus 1 + music results



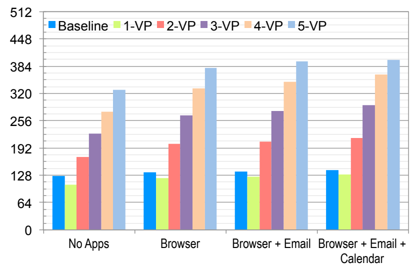
(d) Normalized Nexus S + music results



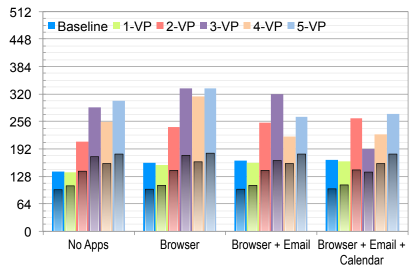
(e) Normalized Nexus 1 power usage



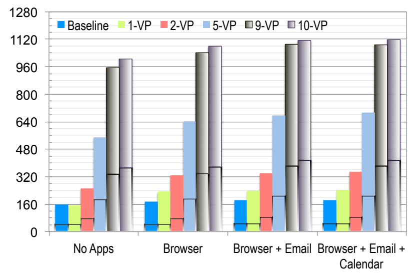
(f) Normalized Nexus S power usage



(g) Nexus 1 Memory usage in MB



(h) Nexus S Memory usage in MB with total number of processes overlayed



(i) NVIDIA Tegra total memory usage in MB (graphics memory below separator)

Figure 3: Experimental results

normalized to the performance of the stock system running the first scenario without a background workload to show the overhead introduced by the background workload. As would be expected, there is some additional overhead relative to a stock system not running a background workload, though the amount of overhead varies across applications. Relative to a stock system, Neocore has the least overhead and has almost the same overhead as without the background workload, because it primarily uses the GPU for 3D rendering, which is not used by the music player. Linpack, Quadrant 2D, and SunSpider incur some additional overhead compared to running without the background music player workload, reflecting the additional CPU overhead of running the music player at the same time. Network runtime overhead while running an additional background workload showed the same level of variability in measurement results as the benchmarks run without a background workload. *Cells*'s network performance overhead is modest, as the variance in the results for any one configuration still exceeded the difference across configurations. Quadrant I/O overhead was the highest among the benchmark applications, reflecting the expected I/O contention between the I/O benchmark and the music player.

Comparing to the Baseline configuration with an additional background workload, *Cells* overhead remains small in all cases. It incurs less than 1% overhead in all cases on the Nexus 1 except for Network and Quadrant I/O, and less than 4% overhead in all cases on the Nexus S except for Quadrant I/O, although the majority of benchmark results on the Nexus S show nearly zero overhead. Quadrant I/O on the Nexus 1, while running an additional background workload, incurs a maximum overhead of 7% relative to Baseline performance. Quadrant I/O on the Nexus S has less than 2% overhead for the 1-VP configuration when compared to the Baseline configuration. However, configurations with more than 1 VP show an overhead of 10% relative to the Baseline. This is because the Baseline results on the Nexus S were better than the Nexus 1 reflecting better I/O performance. This higher absolute performance of the Nexus S accentuates the virtualization overhead due to running multiple VPs.

Figures 3e and 3f show power consumption on the Nexus 1 and Nexus S, respectively, both while playing music with the standard Android music player for 4 hours continuously, and while letting the phone sit idle for 12 hours in a low-power state. In both scenarios, the background VPs were the same as the foreground VP except that in the second scenario the music player was not running in the background VPs. Note that the graph presents normalized results, not absolute percentage difference in battery capacity usage, so lower numbers are better.

The power consumption attributable to *Cells* during the 4 hours of playing music on the Nexus 1 increased while running more VPs, which involved running more processes on the system. We found that while playing music, *Cells* created a higher power supply load variation due to the scheduling and running of a much larger set of processes and threads. The nonlinearity in how this variation affects power consumption resulted in the 4-6% overhead in battery usage for 1-VP through 3-VP, and the 10-20% overhead for 4-VP and 5-VP experiments. Nexus 1 power consumption after 12 hours of sitting idle was within 2% of Baseline. Note that when the device sat idle, the Android wake lock system would aggressively put the device in a low-power mode where the CPU was completely powered down. The idle power consumption results hold even when background cells are running applications which would normally hold wake locks to prevent the device from sleeping such as a game like *Angry Birds* or the Android music player. This shows that the *Cells*' wake lock system virtualization makes efficient use of precious battery resources.

Nexus S power measurements showed no measurable increase in power consumption due to *Cells* for either playing music for 4 hours or letting the phone idle for 12 hours. Because the Nexus S is a newer device, the better power management performance may be reflective of what could be expected for running *Cells* on newer hardware.

Figures 3g, 3h, and 3i show memory usage on the Nexus 1, Nexus S, and NVIDIA tablet, respectively. These results show that by leveraging the KSM driver and file system unioning, *Cells* requires incrementally less memory to start each additional VP compared to running the first VP. Furthermore, the 1-VP configuration uses less memory than the Baseline configuration, also because of the use of the KSM driver. In other words, the memory used increases linearly with the number of VPs running, but at a rate that is much less than the amount of memory required for the Baseline configuration.

Figure 3g shows memory usage on the Nexus 1 for all six configurations with different combinations of pre-installed and commonly used Android applications running in each VP. In each configuration, an instance of every application listed in the figure is running in all background VPs as well as the foreground VP. Leveraging the Linux KSM driver, *Cells* uses approximately 20% less memory for 1-VP than Baseline. The memory cost for *Cells* to start each additional VP is approximately 55 MB, which is roughly 40% of the memory used by the Baseline Android system and roughly 50% of the memory used to start the first VP. The reduced memory usage of additional VPs is due to *Cells*'s use of file system unioning to share common code and data as well as KSM, providing improved scalability on memory-constrained phones.

Figure 3h shows memory usage on the Nexus S under the same conditions described above. The memory cost of starting a cell on the Nexus S is roughly 70 MB. This is higher than the Nexus 1 due to increased heap usage of Android base applications and system support libraries. In addition, the Nexus S device contains several hardware acceleration components which require dedicated regions of memory. These regions can be multiplexed across VPs, but reduce the total available system memory for use by applications for general use. As a result, although the Nexus 1 and Nexus S have the same amount of RAM, the RAM available for general use on the Nexus S is about 350 MB versus 400 MB for the Nexus 1. Thus, after starting the 4^{th} VP, and after starting the browser and the email client in each of 3 VPs, the Android low memory killer kernel driver begins to kill background processes to free system memory for new applications. While this allows us to start and interact with 5 VPs on the Nexus S, it can increase application startup time, and invalidates a direct memory usage comparison to the Nexus 1. To better illustrate the effects of the low memory killer, the total number of processes running at the time of measurement is indicated by the black bar overlaid on each measurement point in Figure 3h. Note that the set of processes in each measurement may be different, so the number of processes and amount of memory used are not strictly correlated.

Figure 3i shows memory usage on the NVIDIA tablet. We were able to start and use 10 VPs on the NVIDIA Ventana tablet development kit. Each VP ran a complete and standard Android software suite, and each had full access to the NVIDIA GeForce GPU and well as other hardware devices present in the dev-kit such as a forward-facing camera, a back-facing stereo camera pair, an accelerometer and a 1080p video decoder. To show the linear increase of both system and graphics memory, the portion of the each column below the horizontal black bar represents the amount of dedicated graphics memory requested in each configuration. Memory required to start a single VP is approximately equivalent to the Baseline configuration, and the memory cost for *Cells* to start each additional VP is roughly 65 MB, which is less than 60% of the memory required for the Baseline system, and also less than the amount of memory required to start additional VPs on the Nexus S.

Note that the NVIDIA graphics memory allocator kernel driver, `nvmap`, can over-commit RAM resources by only physically allocating memory blocks which are actually used. This driver also provides a mechanism similar to the low memory killer which can kill processes in order to reclaim graphics memory. Also note that the apparent memory cost of starting the 10^{th} cell is roughly 14 MB of system RAM. In the 10-VP configuration, the physical memory of the tablet device is exhausted and the Android low memory killer is invoked to kill background processes and free memory.

9 Related Work

Virtualization on embedded and mobile devices is a relatively new area. OKL4 Microvisor [25] is a bare metal hypervisor based on the L4 microkernel [12]. A bare metal hypervisor offers the benefit of a smaller trusted computing base, but the disadvantage of having to provide device support and emulation, an onerous requirement for smartphones which provide increasingly diverse hardware devices. We are not aware of any OKL4 implementations that run Android on any phones other than the HTC G1, which is no longer sold. Another commercial bare-metal hypervisor solutions is VLX from Red Bend [30]. VMware MVP [3] is a hosted virtualization solution for Android. It can therefore leverage Android device support and runs on more recent hardware such as the Google Nexus 1. However, its trusted computing base is larger as it includes both the Android user-space environment and host Linux OS. Xen for ARM [14] and KVM/ARM [6] are open-source virtualization solutions for ARM, but are both incomplete with respect to device support. All of these approaches require paravirtualization and need to run an entire OS instance in each VM, which adds to both memory and CPU overhead. This can significantly limit scalability and performance on resource constrained phones. For example, VMware MVP is targeted for running just one VM to encapsulate an Android virtual work phone on an Android host personal phone. INTEGRITY from Green Hills [11] provides ARM virtualization specifically targeted at real-time applications.

Cells's OS virtualization approach provides several advantages over these hardware virtualization approaches on resource constrained phones. First, OS virtualization is more lightweight and introduces less overhead. Second, only a single OS instance is run to support multiple VPs as opposed to needing to run several OS instances on the same hardware, one per virtual machine plus an additional host instance for hosted virtualization. Attempts have been made to run a heavily modified Android in a VM without the OS instance [13], but they lack support for most applications and are problematic to maintain. Third, OS virtualization is supported in existing commodity OSes such as Linux, enabling *Cells* to leverage existing investments in commodity software as opposed to building and maintaining a separate, complex hypervisor platform. Fourth, by running the same commodity OS already shipped with the hardware, we can leverage already available device support instead of needing to rewrite our own with a bare metal hypervisor.

A potential disadvantage of relying on the OS is that the trusted computing base necessary for ensuring security is potentially larger than a bare metal hypervisor. We believe the benefits in ease of deployment from leveraging existing OS infrastructure are worth this tradeoff. Furthermore, existing non-virtualized smartphones rely on the same OS as the trusted computing base, so the assumption of trusting the OS for security is no worse than existing non-virtualized smartphones and hosted hardware virtualization approaches such as that used by VMware. Another potential disadvantage of an OS virtualization approach is that applications in VPs are expected to run on the same kind of OS; VPs cannot run Apple iOS on an Android system. However, any attempt at running a different OS using hardware virtualization would first need to overcome licensing restrictions and device compatibility issues that would prevent popular smartphone OSs such as iOS from being run on non-Apple hardware and hypervisors from being run on Apple hardware.

User-level approaches have also been proposed to support separate work and personal virtual phones on the same hardware [1]. This is done by providing an Android work phone application that also supports other custom work-related functions such as email. Alternative user-level approaches supply a secure SDK on which applications can be developed [37]. While the advantage of such solutions is portability, a fundamental limitation of this approach is the inability to run standard applications in the virtual phone and a weaker security model.

Efficient device virtualization has been a difficult problem on user-centric systems such as desktops and phones that must support a plethora of devices. Most approaches require emulation of hardware devices, imposing high overhead [40]. Dedicating a device to a VM can enable low overhead pass through operation, but then does not allow the device to be used by other VMs [22]. Mechanisms to bypass the kernel or VMM for network I/O have been proposed to reduce overhead [17], but require specialized hardware support used in high-speed network interfaces and not present on most user-centric systems, including phones. GPU devices are perhaps the most difficult to virtualize. For example, VMware MVP simply cannot run graphics applications such as games within a VM with reasonable performance [34]. Front-end GPU approaches require user-level modifications and have high overhead, while back-end GPU approaches have had inherent problems with multiplexing [7]. In contrast, *Cells* takes advantage of the usage model of phones to provide and use a new device namespace abstraction that can transparently virtualize devices while maintaining native or near native device performance for applications across a wide range of devices, including GPU devices

10 Conclusions

We have designed, implemented, and evaluated *Cells*, the first operating system virtualization solution for Android mobile phones. Mobile phones have a different usage model than traditional computers. We use this observation to provide new device virtualization mechanisms, device namespaces and device namespace proxies, that leverage a foreground-background usage model to isolate and multiplex phone devices with near zero overhead. Device namespaces provide a kernel-level abstraction that is used to virtualize critical hardware devices such as the framebuffer and GPU while providing fully-accelerated graphics performance. Device namespaces are also used to virtualize Android's complicated power management framework, resulting in almost no extra power consumption for *Cells* compared to stock Android. Device namespace proxies provide a user-level mechanism to virtualize closed and proprietary device infrastructure, such as the telephony radio stack, with only minimal configuration changes to the Android user-space environment. *Cells* further provides each virtual phone complete use of the standard cellular phone network with its own phone number and incoming and outgoing caller ID support through the use of a VoIP cloud service.

We have implemented a *Cells* prototype that runs on the latest versions of Android, the most recent Google phone hardware, including both the Nexus 1 and Nexus S, and an NVIDIA Tegra powered tablet device. The system is no less secure than a standard unmodified Android system and can use virtual mobile devices to run standard unmodified Android applications downloadable from the Android market. Scalability tests on the NVIDIA tablet device show that *Cells* makes efficient use of device memory via KSM and scales to multiple VPs with a per-VP memory cost nearly half that of an unmodified Android device. It also shows that *Cells* can leverage the Android low memory killer to start more VPs than the physical memory would otherwise allow. Performance results across a wide-range of applications running in up to 5 virtual phones on the same Nexus 1 and Nexus S hardware show that *Cells* incurs near-zero performance overhead, and human UI testing reveals no visible performance degradation in any of the benchmark configurations.

11 Acknowledgments

Qi Ding and Charles Hastings helped with running benchmarks to obtain many of the measurements in this paper. Kevin DeGraaf setup our Asterisk VoIP service. This work was supported in part by NSF grants CNS-1018355, CNS-0914845, CNS-0905246, AFOSR MURI grant FA9550-07-1-0527, and a Google Research Award.

References

[1] Enterproid. `http://www.enterproid.com`.

[2] Asterisk - The Open Source Telephony Projects. `http://www.asterisk.org`.

[3] K. Barr, P. Bungale, S. Deasy, V. Gyuris, P. Hung, C. Newell, H. Tuch, and B. Zoppis. The VMware mobile virtualization platform: is that a hypervisor in your pocket? *ACM SIGOPS Operating Systems Review*, 44(4):124–135, 2010.

[4] S. Bhattiprolu, E. W. Biederman, S. Hallyn, and D. Lezcano. Virtual servers and checkpoint/restart in mainstream linux. *SIGOPS Oper. Syst. Rev.*, 42:104–113, July 2008.

[5] CNN. Industry first: Smartphones pass PCs in sales. `http://tech.fortune.cnn.com/2011/02/07/idc-smartphone-shipment-numbers-passed-pc-in-q4-2010`.

[6] C. Dall and J. Nieh. KVM for ARM. In *OLS 2010: Proceedings of the Linux Symposium*, pages 45–56, 2010.

[7] M. Dowty and J. Sugerman. GPU Virtualization on VMware's Hosted I/O Architecture. *USENIX Workshop on I/O Virtualization*, 2008. `http://www.usenix.org/event/wiov08/tech/full_papers/dowty/dowty.pdf`.

[8] Google. Nexus One - Google Phone Gallery, May 2011. `http://www.google.com/phone/detail/nexus-one`.

[9] Google. Nexus S - Google Phone Gallery, May 2011. `http://www.google.com/phone/detail/nexus-s`.

[10] Google Inc. Google Voice, February 2011. `http://www.google.com/googlevoice/about.html`.

[11] Green Hills Software. Integrity secure virtualization. `http://www.ghs.com`.

[12] G. Heiser and B. Leslie. The okl4 microvisor: convergence point of microkernels and hypervisors. In *Proceedings of the first ACM asia-pacific workshop on Workshop on systems*, APSys '10, pages 19–24, 2010.

[13] M. Hills. Android on OKL4. http://www.ertos.nicta.com.au/software/androidokl4/.

[14] J. Hwang, S. Suh, S. Heo, C. Park, J. Ryu, S. Park, and C. Kim. Xen on ARM: System Virtualization using Xen Hypervisor for ARM-based Secure Mobile Phones. *Consumer Communications and Networking Conference, 2008. CCNC 2008. 5th IEEE*, pages 257–261, 2008.

[15] K. Kolyshkin. Recent advances in the Linux kernel resource management. http://www.cse.wustl.edu/~lu/control-tutorials/im09/slides/virtualization.pdf.

[16] Linux-VServer. http://www.linux-vserver.org.

[17] J. Liu, W. Huang, B. Abali, and D. Panda. High performance VMM-bypass I/O in Virtual Machines. *Proceedings of the annual conference on USENIX*, 6, 2006.

[18] lxc Linux Containers. http://lxc.sourceforge.net.

[19] M. Melina. HFS ~ HTTP File Server. http://www.rejetto.com/hfs/.

[20] Microsoft. About the Wireless Hosted Network (Windows). http://msdn.microsoft.com/en-us/library/dd815243(v=vs.85).aspx.

[21] Mobile Systems. Office Suite Pro (Trial) – Android Market. https://market.android.com/details?id=com.mobisystems.editor.office_with_reg.

[22] NVIDIA Corporation. NVIDIA SLI MultiOS, February 2011. http://www.nvidia.com/object/sli_multi_os.html.

[23] NVIDIA Corporation. Ventana Development Kit | Tegra Development Zone, April 2011. http://developer.nvidia.com/tegra/devkit-ventana.

[24] J. R. Okajima. aufs. http://aufs.sourceforge.net/aufs2/man.html.

[25] Open Kernel Labs. OKL4 Microvisor, March 2011. http://www.ok-labs.com/products/okl4-microvisor.

[26] OpenVZ. http://wiki.openvz.org/Main_Page.

[27] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The design and implementation of Zap: a system for migrating computing environments. *SIGOPS Oper. Syst. Rev.*, 36:361–376, December 2002.

[28] J. Owens. GPU architecture overview. In *ACM SIGGRAPH 2007 courses*, SIGGRAPH '07, 2007.

[29] polarbit. Reckless Racing – Android Market. https://market.android.com/details?id=com.polarbit.RecklessRacing.

[30] Red Bend Software. VLX Mobile Virtualization. http://www.redbend.com.

[31] Rovio Mobile Ltd. Angry Birds – Android Market. https://market.android.com/details?id=com.rovio.angrybirds.

[32] G. Su. *MOVE: Mobility with Persistent Network Connections*. PhD thesis, Columbia University, 2004.

[33] J. Sugerman, G. Venkitachalam, and B. Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 1–14, 2001.

[34] H. Tuch. Personal communication, December 2010.

[35] VMware Inc. VMware Workstation. http://www.vmware.com/products/workstation/.

[36] C. A. Waldspurger. Memory resource management in VMware ESX server. *SIGOPS Oper. Syst. Rev.*, 36:181–194, December 2002.

[37] WorkLight, Inc. WorkLight Mobile Platform. http://www.worklight.com.

[38] C. P. Wright, J. Dave, P. Gupta, H. Krishnan, D. P. Quigley, E. Zadok, and M. N. Zubair. Versatility and Unix semantics in namespace unification. *Trans. Storage*, 2:74–105, February 2006.

[39] R. J. Wysocki. Technical Background of the Android Suspend Blockers Controversy. http://lwn.net/images/pdf/suspend_blockers.pdf.

[40] Xen Project. Architecture for Split Drivers Within Xen, March 2011. `http://wiki.xensource.com/xenwiki/XenSplitDrivers`.

[41] ZDNet. Stolen apps that root Android, steal data and open backdoors available for download from Google Market. `http://zd.net/gGUhOo`.