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

ABSTRACT

Smartphones are increasingly ubiquitous, and many users carry multiple phones 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 in an isolated, secure 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 multiplex phone hardware across multiple virtual phones while providing native hardware device performance. *Cells* virtual phone features include fully accelerated 3D graphics, complete power management features, and full telephony functionality with separately assignable telephone numbers and caller ID support. We have implemented a prototype of *Cells* that supports multiple Android virtual phones on the same phone. 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 transparently runs Android applications at native speed without any modifications.

Categories and Subject Descriptors

C.0 [**Computer Systems Organization**]: General–System architectures; D.4.6 [**Operating Systems**]: Security and Protection; D.4.7 [**Operating Systems**]: Organization and Design; D.4.8 [**Operating Systems**]: Performance; H.5.2 [**Information Interfaces and Presentation**]: User Interfaces–User-centered design; I.3.4 [**Computer Graphics**]: Graphics Utilities–Virtual device interfaces

General Terms

Design, Experimentation, Measurement, Performance, Security

Keywords

Android, Smartphones, Virtualization

1. INTRODUCTION

The preferred platform for a user's everyday computing needs is shifting from traditional desktop and laptop computers toward mobile smartphone and tablet devices [4]. Smartphones are becoming an increasingly important work tool for professionals who rely on them for telephone, text messaging, email, Web browsing, contact and calendar management, news, and location-specific information. 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 of downloading 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 [35]. For this reason, smartphones given to employees for work use are often locked down resulting in many users having to carry separate work and personal phones. Application developers also carry additional phones for development to avoid having a misbehaving application prototype corrupt their primary phone. Parents sometimes wish they had additional phones when their children use the parent's smartphone for entertainment and end up with unexpected charges due to accidental phone calls or unintended in-app purchases.

Virtual machine (VM) mechanisms have been proposed that enable two separate and isolated instances of a smartphone software stack to run on the same ARM hardware [2, 5, 13, 22]. These approaches require substantial modifications to both user and kernel levels of the software stack. Paravirtualization is used in all cases since ARM is not virtualizable and proposed ARM virtualization extensions are not yet available in hardware. While VMs are useful for desktop and server computers, applying these hardware virtualization techniques to smartphones has two crucial drawbacks. First, smartphones are more resource constrained, and running an entire additional operating system (OS) and user space environment in a VM imposes high overhead and limits the number of instances that can run. Slow system responsiveness is less acceptable on a smartphone than on a desktop computer since smartphones are often used for just a few minutes or even seconds at a time. Second, smartphones incorporate a plethora of devices that applications expect to be able to use, such as GPS, cameras, and GPUs.

Existing approaches provide no effective mechanism to enable applications to directly leverage these hardware device features from within VMs, severely limiting the overall system performance and making existing approaches unusable on a smartphone.

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

Cells takes advantage of the small display form factors of smartphones, which display only a single application at a time, and introduces a usage model having one foreground VP that is displayed and multiple background VPs that are not displayed at any given time. This simple yet powerful model enables *Cells* to provide novel kernel-level and user-level device namespace mechanisms to efficiently multiplex hardware devices across multiple VPs, including proprietary or opaque hardware such as the baseband processor, while maintaining native hardware performance. The foreground VP is always given direct access to hardware devices. Background VPs are 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 feature, such as hardware-accelerated graphics. Since foreground applications have direct access to hardware, they perform as fast as when they are running natively.

Cells uses a VoIP service to provide individual telephone numbers for each VP without the need for multiple SIM cards. Incoming and outgoing calls use the cellular network, not VoIP, and are routed through the VoIP service as needed to provide both incoming and outgoing caller ID functionality for each VP. *Cells* uses this combination of a VoIP server and the cellular network to allow users to make and receive calls using their standard cell phone service, while maintaining per-VP phone number and caller ID features.

We have implemented a *Cells* prototype that supports multiple virtual Android phones on the same mobile device. Each VP can be configured the same or completely different from other VPs. The prototype has been tested to work with multiple versions of Android, including the most recent open-source version, version 2.3.4. It works seamlessly across multiple hardware devices, including Google Nexus 1 and Nexus S phones. Our experimental results running real Android applications in up to five VPs on Nexus 1 and Nexus S phones demonstrate that *Cells* imposes almost no runtime overhead and only modest memory overhead. *Cells* scales to support far more phone instances on the same hardware than VM-based approaches. *Cells* 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.

We present the design and implementation of *Cells*. Section 2 describes the Cells usage model. Section 3 provides an overview of the system architecture. Sections 4 and 5 describe graphics and power management virtualization, respectively, using kernel device namespaces. Sections 6 and 7 describe telephony and wireless network virtualization, respectively, using user-level device namespaces. Section 8 presents experimental results. Section 9 discusses related work. Finally, we present some concluding remarks.

2. USAGE MODEL

Cells runs multiple VPs on a single hardware phone. Each VP runs a standard Android environment capable of making phone calls, running unmodified Android applications, using data connections, interacting through the touch screen, utilizing the accelerometer, and everything else that a user can normally do on the 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, the foreground VP, to be displayed at any time. We refer to all other VPs that are running but not displayed as, background VPs. Background VPs are still running on the system in the background and are capable of receiving system events and performing tasks, but do not render content on the screen. A user can easily switch among 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 an application that can be launched to see a list of available VPs, and to switch any of these to the foreground. The system can 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, preventing background VPs from stealing input focus or device data. An auto-lock can be enabled forcing a user to unlock a VP using a passcode or gesture when it transitions from background to foreground. Section 3 discusses how the foreground-background 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, IT administrators can also create VPs 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 that can 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, VPs with no access to the GPS sensor would never be able to track 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 IT administrators to create VPs that allow users to install and run such applications without compromising privacy.

Shared access means that when a 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.

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, 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.

In addition to device access rights, *Cells* leverages existing OS virtualization technology to prevent privilege escalation attacks in one VP from compromising the entire device. Both user credentials and process IDs are isolated between VPs; the root user in one VP has no relation to the root user in any other VP.

3. SYSTEM ARCHITECTURE

Figure 1 provides an overview of the *Cells* system architecture. We describe *Cells* using Android since our prototype is based on it. Each VP runs a stock Android user space environment. *Cells* leverages lightweight OS virtualization [3, 23] to isolate VPs from one another. *Cells* uses a single OS kernel across all VPs that virtualizes identifiers, kernel interfaces, and hardware resources such 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

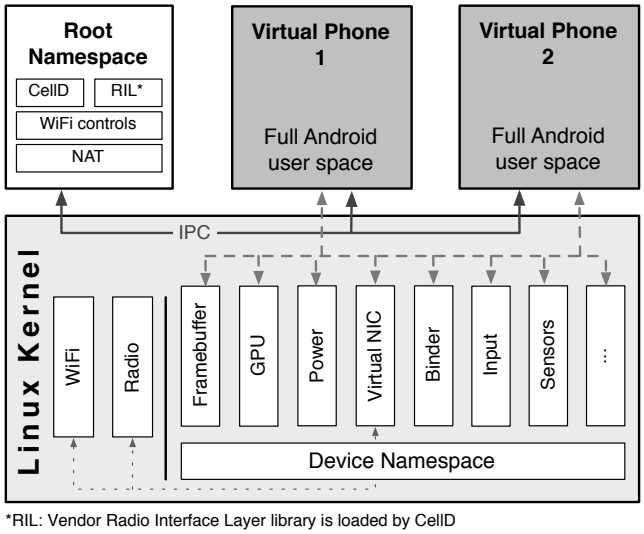


Figure 1: Overview of *Cells* architecture

in one VP cannot see processes in other VPs. The Linux kernel, including the version used by Android, provides virtualization for these identifiers through namespaces [3]. For example: the file system (FS) is virtualized using mount namespaces that allow different independent views of the FS and provide isolated private FS jails for VPs [16].

However, basic OS virtualization is insufficient to run a complete smartphone user space environment. Virtualization mechanisms have primarily been used in headless server environments with relatively few devices, such as networking and storage, which can already be virtualized in commodity OSes such as Linux. Smartphone applications, however, expect to be able to interact with a plethora of hardware devices, many of which are physically not designed to be multiplexed. OS device virtualization support is non-existent 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.

Cells meets all three requirements in the tightly integrated, and often proprietary, smartphone ecosystem. It does so by integrating novel kernel-level and user-level device virtualization methods to present a complete virtual smartphone OS environment. Kernel-level mechanisms provide transparency and performance. User-level mechanisms provide portability and transparency when the user space environment provides interfaces that can be leveraged for virtualization. For proprietary devices with completely closed software stacks, user-level virtualization is necessary.

3.1 Kernel-Level Device Virtualization

Cells introduces a new kernel-level mechanism, *device namespaces*, that provides isolation and efficient hardware resource multiplexing in a manner that is completely transparent to applications. Figure 1 shows how device names-

Device	Description
Alarm*	RTC-based alarms
Audio	Audio I/O (speakers, microphone)
Binder*	IPC framework
Bluetooth	Short range communication
Camera	Video and still-frame input
Framebuffer	Display output
GPU	Graphics processing unit
Input	Touchscreen and input buttons
LEDs	Backlight and indicator LEDs
Logger*	Lightweight RAM log driver
LMK*	Low memory killer
Network	Wi-Fi and Cellular data
Pmem*	Contiguous physical memory
Power*	Power management framework
Radio	Cellular phone (GSM, CDMA)
Sensors	Accelerometer, GPS

Table 1: Android devices
*custom Google drivers

paces are implemented within the overall *Cells* architecture. Unlike PID or UID namespaces in the Linux kernel, which virtualize process identifiers, a device namespace does not virtualize identifiers. It is designed to be used by individual device drivers or kernel subsystems to tag data structures and to register callback functions. Callback functions are called when a device namespace changes state. Each VP uses a unique device namespace for device interaction. *Cells* 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 they respond to a VP 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 virtualizes existing kernel interfaces based on three methods of implementing device namespace functionality. The first method is to create a device driver wrapper using a new device driver for a virtual device. The wrapper device 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 updates device state and access to the device when a new VP becomes the foreground VP. For example, *Cells* use a device driver wrapper to virtualize the framebuffer as described in Section 4.1.

The second method is to modify a device subsystem to be aware of device namespaces. For example, the input device subsystem in Linux handles various devices such as the touchscreen, navigation wheel, compass, GPS, 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, event handlers first check 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.

The third method is to modify a device driver to be aware of device namespaces. For example, Android includes a number of custom pseudo drivers which are not part of an existing kernel subsystem, 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 done by modifying the Binder driver so that instead of allowing Binder data structures to reference a single global list of all processes, they reference device namespace isolated lists and only allow communication between processes associated with the same device namespace. A Binder device 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, however, it may be necessary to modify a hardware-specific device driver to make it aware of device namespaces. For most devices, this is straightforward and involves duplicating necessary driver state upon 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.

3.2 User-Level Device Virtualization

In addition to kernel-level device namespace mechanisms, *Cells* introduces 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. *Cells* also uses this mechanism to virtualize device configuration, such as Wi-Fi, which occurs in user space. Sections 6 and 7 describe how this user-level proxy approach is used to virtualize telephony and wireless network configuration.

Figure 1 shows the relationship between VPs, kernel-level device namespaces, and user-level device namespace proxies which are contained in a *root namespace*. *Cells* works by booting a minimal init environment in a root namespace which is not visible to any 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. Kernel device namespaces export an interface to the root namespace through the `/proc` filesystem that is used to switch the foreground VP and set access permissions for devices. CellD also coordinates user space virtualization mechanisms such as the configuration of telephony and wireless networking.

To start a new VP, CellD mounts the VP filesystem, clones itself into a new process with separate namespaces, and

starts the VP's init process to boot up the 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. *Cells* also leverages existing Linux kernel frameworks for resource control to prevent resource starvation from a single VP [15].

3.3 Scalability and Security

Cells uses three scalability techniques to enable multiple VPs running the same Android environment to share code and reduce memory usage. First, the same base file system is shared read-only among VPs. To provide a read-write file system view for a VP, file system unioning [32] is used to join the read-only base 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: file 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. Second, when a new VP is started, *Cells* enables Linux Kernel Samepage Merging (KSM) for a short time to further reduce memory usage by finding anonymous memory pages used by the user space environment that have the same contents, then arranging for one copy to be shared among the various VPs [30]. Third, *Cells* leverages the Android low memory killer to increase the total number of VPs it is possible to run on a device without sacrificing functionality. The Android low memory killer kills 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.

Cells uses four techniques to isolate all VPs from the root namespace and from one another, thereby securing both system and individual VP data from malicious reads or writes. First, user credentials, virtualized through UID namespaces, isolate the root user in one VP from the root user in the root namespace or any other VP. Second, kernel-level device namespaces isolate device access and associated data; no data or device state may be accessed outside a VP's device namespace. Third, mount namespaces provide a unique and separate FS view for each VP; no files belonging to one VP may be accessed by another VP. Fourth, CellD removes the capability to create device nodes inside a VP, preventing processes from gaining direct access to Linux devices or outside their environment, e.g., by re-mounting block devices. These isolation techniques secure *Cells* system data from each VP, and individual VP data from other VPs. For example, a privilege escalation or root attack compromising one VP has no access to the root namespace or any other VP, and cannot use device node creation or super-user access to read or write data in any other VP.

4. GRAPHICS

The display and its graphics hardware is one of the most important devices in smartphones. Applications expect to take full advantage of any hardware display acceleration or GPU available on the smartphone. Android relies on a standard Linux framebuffer (*FB*) which provides an abstraction to a physical display, including screen memory, memory dedicated to and controlled exclusively by the display device. For performance reasons, screen memory is mapped and written to directly both by processes and GPU hardware. The direct memory mapping and the performance requirements of the graphics subsystem present new challenges for virtualizing mobile devices.

4.1 Framebuffer

To virtualize *FB* access in multiple VPs, *Cells* leverages the kernel-level device namespace and its foreground-background usage model in a new multiplexing *FB* device driver, `mux_fb`. The `mux_fb` driver registers as a standard *FB* device and multiplexes access to a single physical 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 *FB* usage pattern involves three types of accesses: `mmaps`, standard control `ioctls`, and custom `ioctls`. When a process `mmaps` an open *FB* device file, the 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 *FB* hardware state through a set of standard control `ioctls` defined by the Linux framebuffer interface which can, for example, change the pixel format. Each *FB* 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. 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 used, for example, to accelerate rendering. 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 driver does not register for this

new notification, *Cells* either returns an error code, or blocks the calling process when the custom `ioctl` is called from a background VP. Returning an error code was sufficient for both the Nexus 1 and Nexus S systems. 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 *FB* 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 *FB* screen memory to redirect virtual addresses in each process to new 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 have their virtual addresses remapped to physical screen memory. The 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 then setting the current hardware state to the new foreground VP's virtual hardware state. 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, discussed in section 4.2, involves notifying the GPU of the memory address switch so that it can update any internal graphics memory mappings.

To better scale the *Cells FB* 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. Redraw overhead is minimal, and Android conveniently provides this 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 together with the *FB* 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 each other and there is no need for further VP GPU isolation. The key challenge is that each VP requires *FB* screen memory on which to compose the final scene to be displayed, and in general the GPU driver can request and use this memory from within the OS kernel.

Cells solves this problem by leveraging its foreground-background usage model to provide a virtualization solution similar to *FB* screen memory remapping. The foreground VP will use the GPU to render directly into screen memory, but background VPs, which use the GPU, will render into their respective backing buffers. When the foreground VP changes, the GPU driver locates all GPU addresses which are mapped to the physical screen memory as well as the background VP's backing buffer in system RAM. It must then remap those GPU addresses to point to the new backing buffer and to 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.

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 user space configuration will attempt to initialize the GPU on startup. Third, it should provide the ability to ignore device power management and other non-graphics related hardware state updates, making it possible to ignore such events from a user space instance running in a background VP. Some of these capabilities were already available on the *Adreno* GPU driver, used in the Nexus 1, but not all. We added a small number of lines of code to the *Adreno* GPU driver and *PowerVR* GPU driver, used in the Nexus S, to implement 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 or Samsung's `s3c_mem` driver.

5. POWER MANAGEMENT

To provide *Cells* users the same power management experience as 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 [33].

The *early suspend* subsystem is an ordered callback interface allowing drivers to receive notifications just before a device is suspended and after it resumes. *Cells* virtualizes this subsystem by disallowing background VPs from initiating sus-

pend operations. The remaining two Android-specific power management interfaces present 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 block all processes using the display while the display is powered off, and redraw the screen after the display is powered on. Power is saved since 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, `wait_for_fb_sleep` and `wait_for_fb_wake`. When a user process opens and reads from one of these files, the read blocks until the framebuffer device is either asleep or awake, respectively.

Cells virtualizes fbearlysuspend by making it namespace aware, leveraging the kernel-level device namespace and foreground-background usage model. In the foreground VP, reads function exactly as a non-virtualized system. Reads from a background VP always report the device as sleeping. When the foreground VP switches, all processes in all VPs 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 see the display as awake, processes in the formerly foreground VP see the display as asleep, and processes running in background VPs that remain in the background continue to see the display as asleep. This forces background VPs to pause drawing or rendering which reduces overall system load by reducing the number of processes using hardware drawing resources, and increases graphics throughput in the foreground VP by ensuring that its processes have exclusive access to the hardware.

5.2 Wake Locks

Wake locks are a special kind of OS kernel reference counter with two states: *active* and *inactive*. When a wake lock is "locked", its state is changed to active; when "unlocked," its state is changed to inactive. A wake lock can be locked multiple times, but only requires a single unlock to put it into the inactive state. The Android system will not enter suspend, or low power mode, until all wake locks are inactive. When all locks are inactive, a suspend timer is started. If it completes without an intervening lock then the device is powered down.

Wake locks in a background VP interfering with the foreground VP's ability to suspend the device coupled with their distributed use and initialization make wake locks a challenging virtualization problem. Wake locks can be created statically at compile time or dynamically by kernel drivers or user space. They can also be locked and unlocked from user context, kernel context (work queues), and interrupt context (IRQ handlers) independently, making determination of the VP to which a wake lock belongs a non-trivial task.

Cells leverages the kernel-level device namespace and foreground-background usage model to maintain both kernel and user space wake lock interfaces while adhering to the two virtualization principles specified above. The solution is predicated on three assumptions. First, all lock and unlock coordination in the trusted root namespace was 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* virtualizes Android wake locks by allowing multiple device namespaces to independently lock and unlock the same wake lock. Power management operations are initiated based on the state of the set of locks associated with the foreground VP. The solution comprises the following set of rules:

1. When a wake lock is locked, a namespace "token" is associated with the lock indicating 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, adjust any suspend timeout value based only on locks acquired in the current device namespace.
5. After a root namespace lock or unlock, adjust the suspend timeout based on the foreground VP's device namespace.
6. When the foreground VP changes, reset the suspend timeout based on locks acquired in the newly active namespace. This requires per-namespace bookkeeping of suspend timeout values.

One additional mechanism was necessary to implement the *Cells* wake lock virtualization. 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 executes a kernel work queue in a specific device namespace.

6. TELEPHONY

Cells provides each VP with separate telephony functionality enabling per-VP call logs, and independent phone numbers. 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 physical phone using the standard carrier voice network and a single SIM.

6.1 RIL Proxy

The Android telephony subsystem is designed to be easily ported by phone vendors to different 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 selection of different roaming options. The phone application, the libraries and the services all communicate via Binder

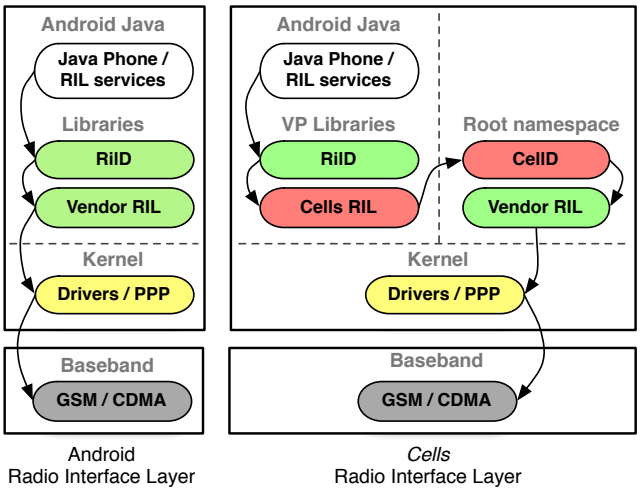


Figure 2: *Cells* Radio Interface Layer

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 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.

As a result of these constraints, *Cells* virtualizes telephony using our user-level device namespace proxy in a solution 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 then communicates with the hardware 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. We refer to the proxy RIL library together with CellD as the *RIL proxy*. The right side of Figure 2 shows the *Cells* Android telephony system, which has three key features. 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 since Android 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.

Call	Class	Category
Dial Request	Solicited	Foreground
Set Screen State	Solicited	
Set Radio State	Solicited	
SIM I/O	Solicited	Initialization
Signal Strength	Unsolicited	Radio Info
Call State Changed	Unsolicited	Phone Calls
Call Ring	Unsolicited	
Get Current Calls	Solicited	

Table 2: Filtered RIL commands

Cells leverages its foreground-background 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 uses the vendor RIL library in the same manner as Android's RilD, 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 an incoming call to a background VP, CellD issues the same set of standard GSM commands RilD would have used.

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. In *Cells*, the RIL vendor library is loaded in the CellD process instead of the RilD process and 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 to RilD inside a VP. CellD filters 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 isolated telephony. The commands that require filtering or special handling are shown in Table 2 and can be categorized as those involving the foreground VP, initialization, radio info, and phone calls.

Foreground commands are allowed only from the foreground VP. The *Dial Request* command 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. *Set Screen State* is filtered from background VPs by only changing a per-VP variable in CellD that suppresses notifications to the issuing background VP accordingly. *Dial Request* and *Set Radio State* are filtered from all background VPs by returning an error code to the calling background VP. This ensures that background VPs do not interfere with the foreground VP's exclusive ability to place calls.

Initialization commands are run once on behalf of the first foreground VP to call them. The *SIM I/O* command is used to communicate directly with the SIM card, and is called during radio initialization (when turning on the device or turning off airplane mode), and when querying SIM information such as the *IMSI*. The first time a VP performs a *SIM I/O* command, CellD records an ordered log of commands, associated data, and corresponding responses. This log is used to replay responses from the vendor RIL library when other VPs attempt *SIM I/O* commands. 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. CellD also records the radio state between each *SIM I/O* command to properly replay the state transitions.

Radio Info commands are innocuous and are broadcast to all VPs. *Signal Strength* is an unsolicited notification about the current signal strength generated by the vendor library. CellD re-broadcasts this information to all VPs with one exception. During initialization, a VP cannot be notified of the signal strength since that would indicate an already initialized radio and generate errors in the initializing VP.

The *Phone Call* commands, *Call State Changed*, *Call Ring*, and *Get Current Calls*, notify a VP of incoming calls and call state changes. When an incoming call occurs, a *Call State Changed* notification is sent, followed by a number of *Call Ring* notifications for as long as the call is pending. CellD inspects each notification and determines the VP to which it should forward the notification. However, this is somewhat complicated since neither notification is associated with a phone number. Therefore, CellD queues these notifications and issues a *Get Current Calls* command, mirroring the functionality of RilD, to receive a list of all incoming and active calls. Using tagging information encoded in the caller ID as discussed in Section 6.2, CellD determines the target VP and passes the queued notifications into the appropriate VP. When a VP issues a *Get Current Calls* request, CellD intercepts the data returned from the vendor library and only returns data from calls directed to, or initiated from the requesting VP.

CellD's architecture supports a highly configurable implementation, and there are many valid security configuration scenarios. For example, if the user switches the foreground VP during a call, CellD can either drop the call and switch to the new VP, keep the call alive and switch to a new VP (handling the active call in a background VP, or, deny switching to a new VP until the call is ended by the user. Under all configurations, *Cells* provides strict isolation between every VP by not allowing any information pertaining to a specific VP to be revealed to another VP including incoming and outgoing call information and the phone call voice data.

6.2 Multiple Phone Numbers

While some smartphones 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 market. Our approach is based on pairing *Cells* with a VoIP service that enables telephony with the standard cellular voice network and standard Android applications, but with separate phone numbers.

The *Cells* VoIP service consists of a VoIP server which registers a pool of subscriber numbers and pairs each of them with the carrier provided number associated with a user's SIM. The VoIP server receives incoming calls, forwards them to a user's actual phone number using the standard cellular voice network, and passes the incoming caller ID to the user's phone appending a digit denoting 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 VoIP service interface. CellD strips the appended digit before forwarding call information to the receiving VP resulting in correctly displayed caller IDs within the VP. If the VP is not available, the VoIP service will direct the incoming call to a 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. While it is certainly possible to spoof caller ID, in the worst case, this would simply appear to be a case of dialing the wrong phone number. Our VoIP service is currently implemented using an Asterisk [1] server as it provides unique functionality not available through other commercial voice services. For example, although Google Voice can forward multiple phone numbers to the same land line, it does not provide this capability for mobile phone numbers, and does not provide arbitrary control over outgoing caller ID [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 mobile 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 service subscriber 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 devices are most commonly equipped with an IEEE 802.11 wireless LAN (WLAN) 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 cellular 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* inte-

grates both kernel and user-level virtualization to provide necessary isolation and functionality, including core network resource virtualization and a unique wireless configuration management virtualization.

Cells leverages previous kernel-level work [27,28] that virtualizes core network resources such as IP addresses, network adapters, routing tables, and port numbers. This functionality has been largely built in to recent versions of the Linux kernel in the form of network namespaces [3]. Virtual identifiers are provided in VPs for all network resources, which are then translated into 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 is present inside a VP and the other end is 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, WLAN and cellular data connections use device-specific, user-level configuration which requires support outside the scope of existing core network virtualization. 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 [29] which create a virtual wired Ethernet adapter inside a virtual machine but leave the configuration on the host system. This model does not work on a mobile device 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. 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 emulating the device-specific understanding of configuration management which is error-prone, complicated, and difficult to maintain.

To address this problem, *Cells* leverages the user-level device namespace proxy and the foreground-background 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 communicates with the user-level library for configuring WLAN or cellular data connections. In the default case where all VPs are allowed network access, CellD forwards all configuration requests from the foreground VP proxy to the user-level library, and 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 interface with user space configuration libraries.

To virtualize Wi-Fi configuration management, *Cells* replaces `wpa_supplicant` inside each VP with a thin Wi-Fi proxy. The well-defined socket interface used by `wpa_supplicant` is simple to virtualize. 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 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 the foreground VP is connected to a Wi-Fi network and a background VP requests to disable the Wi-Fi access, the request is ignored. At the same time, inquiries sent from background VPs that do not change state or divulge sensitive information, such as requesting the current signal strength, are processed since applications such as email clients inside background VPs may use this information when checking for new email.

For virtualizing cellular data connection management, *Cells* replaces the RIL vendor library as described in Section 6, which is also responsible for establishing cellular data connections. As with Wi-Fi, CellD ensures that background VPs cannot interfere with the operation of the foreground VP. For instance, a background VP cannot change the data roaming options causing the foreground VP to either lose data connectivity or inadvertently use the data connection. Cellular data is configured independently from the Wi-Fi connection and VPs can also be configured to completely disallow data connections. Innocuous inquiries from background VPs with network access, such as the status of the data connection (Edge, 3G, HSPDA, etc.) or signal strength, are processed and reported back to the VPs.

8. EXPERIMENTAL RESULTS

We have implemented a *Cells* prototype using Android and demonstrated its complete functionality across different Android devices, including the Google Nexus 1 [8] and Nexus S [9] phones. The prototype has been tested to work with multiple versions of Android, including the most recent open-source version, version 2.3.4. In UI testing while running multiple VPs on a phone, 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 Nexus 1 device, we simultaneously played the popular game *Angry Birds* [26] in one VP, raced around a dirt track in the *Reckless Racing* [24] game on a second VP, crunched some numbers in a spreadsheet using the *Office Suite Pro* [19] 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 in-

stances while seamlessly switching 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. Our measurements were obtained using a Nexus 1 (Qualcomm 1 GHz QSD8250, Adreno 200 GPU, 512 MB RAM) and Nexus S (Samsung Hummingbird 1 GHz 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. 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.

The *Cells* implementation used for our measurements was based on the Android Open Source Project (AOSP) version 2.3.3, the most recent version available at the time our measurements were taken. Aufs version 2.1 was used for file system unioning [21]. 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 writable branches. When one VP modified a file in the read-only branch, the modification is 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 uses a custom system call which adds 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, CellD monitors the KSM statistics through the procfs interface and disables shared page merging after the merge rate drops 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 (Android 2.3.3 build GRI40). 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 system origin in our experiments, we also measured the performance of a baseline system (Baseline) created by compiling the AOSP v2.3.3 source and installing it unmodified.

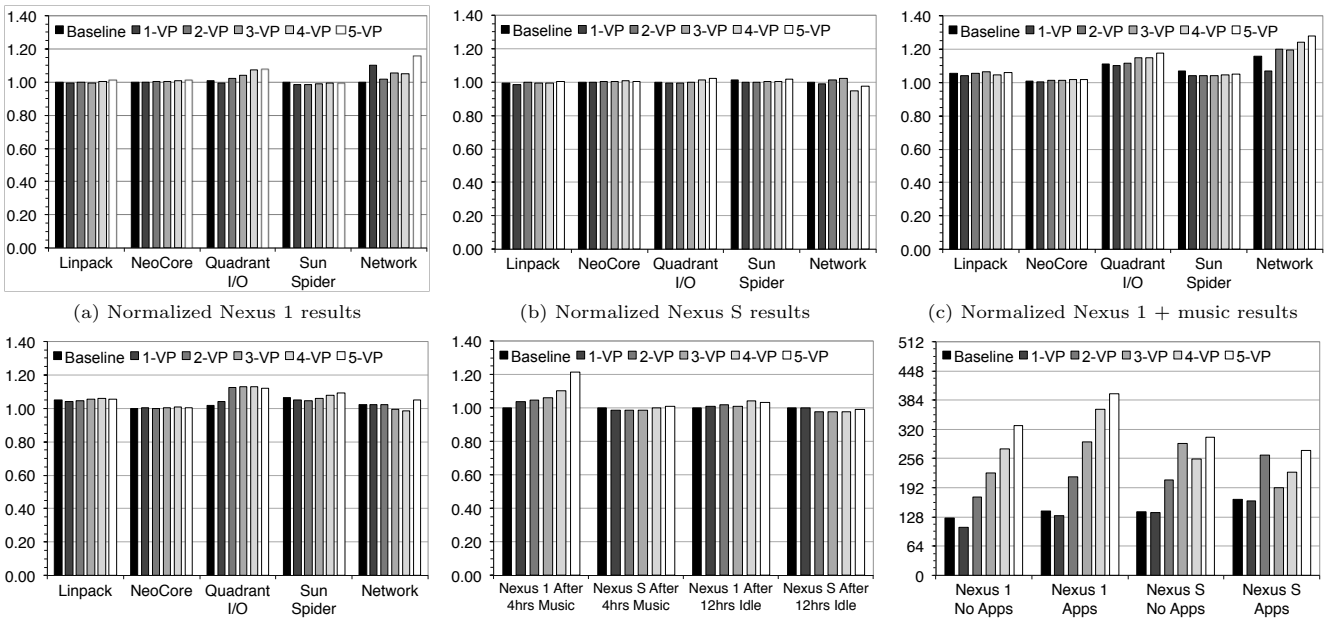
We measured runtime overhead in two scenarios, one with a benchmark application designed to stress some aspect of the system, and the other with the same application running, but simultaneously with an additional background workload. The benchmark application was always run in the foreground VP and if a background workload was used, it was run in a single 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; file I/O using Quadrant Advanced Edition v1.1.1; 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 (1.66 GHz Intel Atom N450, Intel GMA 3150 GPU, 1 GB RAM). The laptop was running Windows 7, providing a WPA wireless access point via its Atheros AR9285 chipset and built-in Windows 7 SoftAP [18] functionality, and serving up the file through the HFS [11] file server v2.2f. To minimize network variability, a location with minimal external Wi-Fi network interference was chosen. Each experiment was performed from this same location with the phone connected to the same 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 power scenarios. In the first scenario, the device configuration under test was fully booted, all VPs started up and KSM had stopped merging pages, then the Android music player was started. In multiple VP configurations, the music player ran in the foreground VP, preventing the device from entering a low power state. The music player repeated the same song continuously for four hours. During this time we sampled the remaining battery capacity every 10 seconds. In the second power scenario, the device configuration under test was fully booted, 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. To measure power consumption due to *Cells* and avoid having those measurements completely eclipsed by Wi-Fi, cellular, and display power consumption, we disabled Wi-Fi and cellular communication, and turned off the display backlight 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 two different memory scenarios. First, we ran a full Android environment without launching any additional applications other than those that are launched by default on system bootup (No Apps). Second, we ran the first scenario plus the Android Web browser, the Android email client, and the Android calendar application (Apps). In both scenarios, an instance of every application was running in all background VPs as well as the foreground VP.

8.2 Measurements

Figures 3a to 3f show measurement results. These are the first measurements we are aware of for running multiple An-



(a) Normalized Nexus 1 results

(b) Normalized Nexus S results

(c) Normalized Nexus 1 + music results

(d) Normalized Nexus S + music results

(e) Normalized battery capacity

(f) Memory usage in MB

Figure 3: Experimental results

droid instances on a single phone. 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 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 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. While testing in a more tightly controlled environment would provide more stable numbers, any overhead introduced by *Cells* was consistently below Wi-Fi variability levels observed on the manufacturer's stock system and should 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. All results are 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 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 and SunSpider incur some additional overhead compared to running without the background 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* 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 Base-

line due to higher I/O performance in the Nexus S baseline compared to the Nexus 1. The higher absolute performance of the Nexus S accentuates the virtualization overhead of running multiple VPs.

Figure 3e shows power consumption on the Nexus 1 and Nexus S, 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 scheduling and running more processes and threads on the system and resulted in a higher power supply load variation. 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. In contrast, the Nexus S showed no measurable increase in power consumption during the 4 hours of playing music, though the the noisy measurements had some slight variation. Because the Nexus S is a newer device, the better power management may be reflective of what could be expected when running *Cells* on newer hardware.

Nexus 1 power consumption after 12 hours of sitting idle was within 2% of Baseline. Similarly, Nexus S measurements showed no measurable increase in power consumption due to *Cells* after the 12 hour idle period. 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 VPs 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 virtualization makes efficient use of battery resources.

Figure 3f shows memory usage on the Nexus 1 and Nexus S. 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 due to the use of the KSM driver. *Cells* device memory use increases linearly with the number of VPs running, but at a rate much less than the amount of memory required for the Baseline.

The Nexus 1 memory usage is reported for both memory scenarios, No Apps and Apps, across all six configurations. The No Apps measurements were taken after booting each VP and waiting until CellD disabled the KSM driver. The Apps measurements were taken after starting an instance of the Android Web browser, email client, and calendar program in each running VP. Leveraging the Linux KSM driver, *Cells* uses approximately 20% less memory for 1-VP than Baseline in the No Apps scenario. The No Apps measurements show that 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*' use of file system unioning to share common code and data as well as KSM, providing improved scalability on memory-constrained phones.

As expected, the No Apps scenario uses less memory than the Apps scenario. Starting all three applications in the 1-VP Apps scenario consumes 24 MB. This memory scales linearly with the number of VPs because we disable the KSM driver before starting the applications. It may be possible to reduce the memory used when running the same application in all VPs by periodically enabling the KSM driver, however application heap usage would limit the benefit. For example, while *Cells* uses 20% less memory for 1-VP than Baseline in the No Apps scenario, this savings decreases in the Apps scenario because of application heap memory usage.

The Nexus S memory usage is reported under the same conditions described above for the Nexus 1. The memory cost of starting a VP on the Nexus S is roughly 70 MB. This is higher than the Nexus 1 due to increased heap usage by Android base applications and system support libraries. The memory cost of starting all three apps in the 1-VP Apps scenario is approximately the same as the Nexus 1, and also scales linearly with the number of running VPs.

However, the total memory usage for the Nexus S shown in Figure 3f does not continue to increase with the number of running VPs. This is due to the more limited available RAM on the Nexus S and the Android low memory killer. The Nexus S 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 general use by applications. 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 in the No Apps scenario, and after starting the 3^{rd} VP in the Apps scenario, the Android low memory killer begins to kill background processes to free system memory for new applications. While this allowed us to start and interact with 5 VPs on the Nexus S, it also slightly increased application startup time.

9. RELATED WORK

Virtualization on embedded and mobile devices is a relatively new area. Bare-metal hypervisors such as OKL4 Microvisor [22] and Red Bend's VLX [25] offer the potential 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. For example, we are not aware of any OKL4 implementations that run Android on any phones other than the dated HTC G1. A hosted virtualization solution such as VMware MVP [2] can leverage Android device support to more easily run on recent hardware, but its trusted computing base is larger as it includes both the Android user space environment and host Linux OS. Xen for ARM [13] and KVM/ARM [5] are open-source

virtualization solutions for ARM, but are both incomplete with respect to device support. All of these approaches require paravirtualization and require an entire OS instance in each VM adding to both memory and CPU overhead. This can significantly limit scalability and performance on resource constrained phones. For example, VMware MVP is targeted to run just one VM to encapsulate an Android virtual work phone on an Android host personal phone.

Cells' OS virtualization approach provides several advantages over existing hardware virtualization approaches on smartphones. First, it 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 VM 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 [12], 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.

Cells has two potential disadvantages versus hardware virtualization. First, the TCB necessary for ensuring security is potentially larger than a bare metal hypervisor, though no worse than hosted virtualization. We believe the benefits in ease of deployment from leveraging existing OS infrastructure are worth this tradeoff. Second, applications in VPs are expected to run on the same OS, for example VPs cannot run Apple iOS on an Android system. However, running a different OS using hardware virtualization would first need to overcome licensing restrictions and device compatibility issues that would prevent popular smartphone OSes 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 phone environments on the same mobile hardware. This is done by providing either an Android work phone application [7] that also supports other custom work-related functions such as email, or a secure SDK on which applications can be developed [31]. While such solutions are easier to deploy, they suffer from the inability to run standard Android applications and a weaker security model.

Efficient device virtualization is 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 [34]. 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 [20]. Bypass mechanisms for network I/O have been proposed to reduce overhead [17], but require specialized hardware support used in high-speed network interfaces 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 [VMware, personal communication]. There are two basic GPU virtualization techniques, API forwarding and back-end virtualization [6]. API forwarding adds substantial complexity and overhead to the TCB, and is problematic due to vendor-specific graphics extensions [14]. Back-end virtualization in a type-1 hypervisor offers the potential for transparency and speed, but unfortunately most graphics vendors keep details of their hardware trade-secret precluding any use of this virtualization method. In contrast, *Cells* leverages existing GPU graphics context isolation and takes advantage of the usage model of mobile devices to create a new device namespace abstraction that transparently virtualizes devices while maintaining native or near native device performance across a wide range of devices including GPU devices.

10. CONCLUSIONS

We have designed, implemented, and evaluated *Cells*, the first OS virtualization solution for mobile devices. Mobile devices 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. 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. *Cells* proxy libraries 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 the latest open-source version of Android on the most recent Google phone hardware, including both the Nexus 1 and Nexus S. The system can use virtual mobile devices to run standard unmodified Android applications downloadable from the Android market. Applications running inside VPs have full access to all hardware devices, providing the same user experience as applications running on a native phone. Performance results across a wide-range of applications running in up to 5 VPs 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. Philip Levis provided helpful comments on earlier drafts of this paper. 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.

12. REFERENCES

[1] Asterisk. http://www.asterisk.org.

[2] 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*, 44:124–135, Dec. 2010.

[3] S. Bhattiprolu, E. W. Biederman, S. Hallyn, and D. Lezcano. Virtual Servers and Checkpoint/Restart in Mainstream Linux. *ACM SIGOPS Operating Systems Review*, 42:104–113, July 2008.

[4] 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.

[5] C. Dall and J. Nieh. KVM for ARM. In *Proceedings of the Ottawa Linux Symposium*, Ottawa, Canada, June 2010.

[6] M. Dowty and J. Sugerman. GPU Virtualization on VMware's Hosted I/O Architecture. *ACM SIGOPS Operating Systems Review*, 43:73–82, July 2009.

[7] Enterproid, Inc. http://www.enterproid.com.

[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, Feb. 2011. http://www.google.com/googlevoice/about.html.

[11] HFS ~ HTTP File Server. http://www.rejetto.com/hfs/.

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

[13] 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. In *Proceedings of the* 5^{th} *Consumer Communications and Newtork Conference*, Las Vegas, NV, Jan. 2008.

[14] Khronos Group. OpenGL Extensions – OpenGL.org. http://www.opengl.org/wiki/OpenGL_Extensions.

[15] K. Kolyshkin. Recent Advances in the Linux Kernel Resource Management. http://www.cse.wustl.edu/~lu/control-tutorials/im09/slides/virtualization.pdf.

[16] O. Laadan, R. Baratto, D. Phung, S. Potter, and J. Nieh. DejaView: A Personal Virtual Computer Recorder. In *Proceedings of the* 21^{st} *Symposium on Operating Systems Principles*, Stevenson, WA, Oct. 2007.

[17] J. Liu, W. Huang, B. Abali, and D. K. Panda. High Performance VMM-bypass I/O in Virtual Machines. In *Proceedings of the 2006 USENIX Annual Technical Conference*, Boston, MA, June 2006.

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

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

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

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

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

[23] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The Design and Implementation of Zap: a System for Migrating Computing Environments. In *Proceedings of the* 5^{th} *Symposium on Operating Systems Design and Implementation*, Boston, MA, Dec. 2002.

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

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

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

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

[28] J. Sugerman, G. Venkitachalam, and B. Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *Proceedings of the 2001 USENIX Annual Technical Conference*, Boston, MA, June 2001.

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

[30] C. A. Waldspurger. Memory Resource Management in VMware ESX Server. In *Proceedings of the* 5^{th} *Symposium on Operating Systems Design and Implementation*, Boston, MA, Dec. 2002.

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

[32] 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. *ACM Transactions on Storage (TOS)*, 2:74–105, Feb. 2006.

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

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

[35] ZDNet. Stolen Apps that Root Android, Steal Data and Open Backdoors Available for Download from Google Market. http://zd.net/gGUhOo.