# A Virtualization Solution for BYOD with Dynamic Platform Context Switching

THIS ARTICLE CONSIDERS AN EMBEDDED VIRTUALIZATION PLATFORM DESIGN FOR A BRING YOUR OWN DEVICE (BYOD) SCENARIO, IN WHICH A COMPANY'S PRIVILEGED INFORMATION MUST BE ISOLATED FROM UNAUTHORIZED EMPLOYEE ACCESS. THE AUTHORS PROPOSE THE VNATIVE ARCHITECTURE DESIGN FOR COMPLETE ISOLATION AMONG DIFFERENT USAGE SCENARIOS. VNATIVE IS SUITABLE FOR BYOD TO FREEZE OTHER UNAUTHENTIC APPLICATIONS AND IS CONSISTENT WITH THE POWER CONSUMPTION AND LIMITED DISPLAY SIZE OF MOBILE DEVICES.

YaoZu Dong
Shanghai Jiao Tong University

JunJie Mao
Tsinghua University

HaiBing Guan
Jian Li
Shanghai Jiao Tong University

Yu Chen
Tsinghua University

Mobile devices have become the most common daily computing and communication devices, and they are continuously increasing in both computing capacity and popularity. Bring Your Own Device (BYOD),[1] also known as Bring Your Own Technology, is a largely generational phenomenon for business and education settings, in which people are allowed to bring personally owned mobile devices (laptops, tablets, and smartphones) to their workplace or education settings, with access authority to privileged information and applications. To this end, BYOD must "shrink" the functionalities of multiple devices into one, and virtualization technology naturally provides a solution. Virtualization can consolidate different applications or use cases in isolated virtual machines (VMs) with either the same or different operating systems. Traditional server virtualization solutions focus on providing high-performance computing for enormous and concurrent user requests with quick responsiveness[2] and flexible management.[3] Paravirtualization closes the virtualization gap of x86 architectures such as Xen.[4,5] To improve performance, researchers also invented additional hardware assistances, such as 2D page table and single-root I/O virtualization technologies.[6,7] Server virtualization technology is considered unsuitable for mobile devices because of their deficiency in lightweight overhead and efficient power consumption.[8]

Mobile use cases differ from server use cases because they are more concerned about user experience and power efficiency. BYOD in particular requires hardened isolated environments for VMs; for example, the companies need the professionals to stop connecting to a public network when accessing privileged information. Moreover, it's

Published by the IEEE Computer Society

better to concentrate resources to a working VM in consideration of system performance, user experience, and power consumption. At the same time, multitasking a single VM can easily fully occupy a mobile device's upper limit of display capacity. These requirements motivated us to invent an alternative architecture rather than running multiple VMs equally, as proposed elsewhere.[9]

Industry and the open source community offer various mobile virtualization solutions. However, these focus on hosting multiple operating systems or applications to run simultaneously and symmetrically, and none of them is dedicated for a BYOD use case. For example, OKL4 Microvisor supports paravirtualized VMs to support mobile virtualization, componentization, and security.[10] But OKL4 Microvisor suffers from the extra cost of processing context switches. L4-Android provides device functions to other VMs via abstract interfaces.[11] In Bromium Microvisor, each application is contained inside a single isolated VM. In the open source community, kernel-based VM (KVM) on an Advanced RISC Machine (ARM) has been available since Linux 3.9, and Xen on ARM is released in Xen 4.3. On the other hand, the OS can also be modified to provide multiple isolated execution environments, such as container technology. For example, Cells introduces a usage model with one foreground virtual phone and multiple background virtual phones, and implements multiple Android virtual phones on the same device.[12] However, the OS container shares the OS kernel among multiple execution environments, and therefore subjects the execution environment's security to the OS kernel's security. At the same time, it cannot host applications from different operating systems simultaneously.

For the BYOD use case, we propose vNative, a novel mobile virtualization solution that exploits mobile computing use cases to multiplex the underlying mobile platform efficiently between two or more virtual machines. vNative hosts VMs with two different runtime statuses, among which only one VM executes in the foreground with the exclusive priority to access the entire platform's resources. Other VMs in the background are idle with dedicated and isolated memory and storage space. These background VMs are ready to switch forward on demand with a full platform context saving-and-restoring scheme, and this full context switch is implemented to be quick enough to keep them always reactive. We designed this solution on the Xen virtualization platform,[5] which can consolidate all mainstream (embedded) operation systems. Our experimental results show that vNative achieves 98.6 percent of native runtime performance with a power overhead of less than 6.5 percent. We evaluated the time cost for a VM switch procedure to be about 376 ms to keep the background VMs' responsiveness.

## vNative design

The vNative design comprises an architecture, guest VM modules, hypervisor modules, and a VM context switching procedure. Essentially, the vNative design is a virtual machine runtime status control scheme with a novel VM context switching procedure, which needs a series of additional modules implemented in guest VM and hypervisor.

### Architecture

vNative classifies VMs into a foreground VM (FVM) and background VMs (BVMs). The FVM is the only VM that handles user requests. It has direct access to devices on the platform and thus has full control over them using the native drivers without modification. The FVM also sends heartbeat messages periodically to the hypervisor. There can be multiple BVMs, on the other hand, which concede the hardware access and do not consume CPU cycles. However, BVMs are kept interactive to the client response through an expeditious context switching scheme from background to foreground.

Figure 1 illustrates vNative's components. The hypervisor's novel components include a VM switcher, a heartbeat receiver, I/O filtering, and virtual memory management unit (vMMU). Each guest VM contains a heartbeat sender, a switch agency, a platform context switch (PCS) driver, and a suite of native device drivers for dedicated mobile platform hardware. Each guest VM is identically labeled, and only one VM can be in FVM status.

wireless
approach

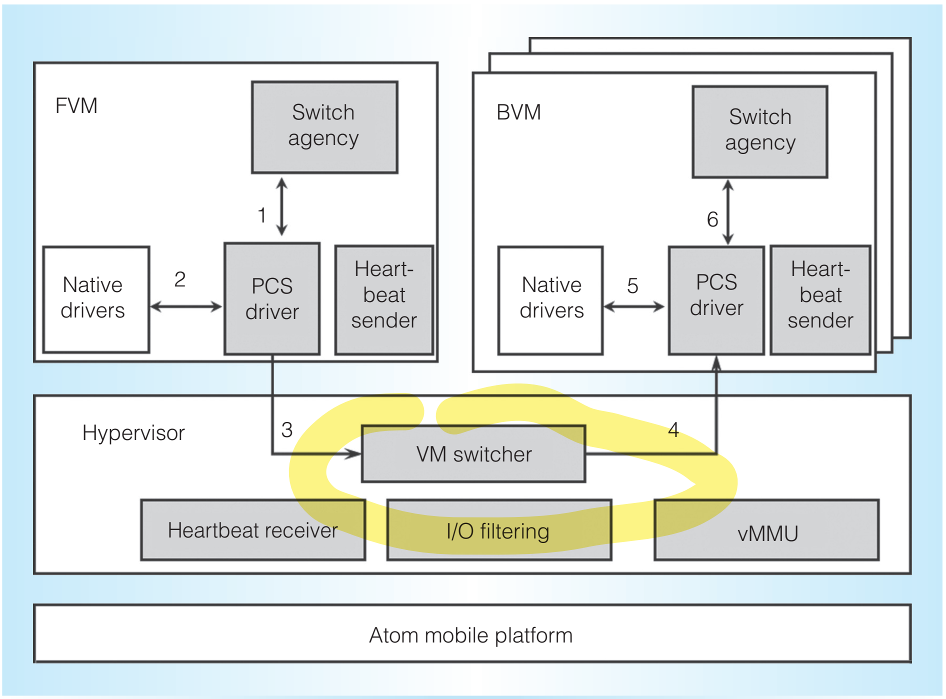

Figure 1. vNative architecture and context switching scheme. The hypervisor contains a virtual machine (VM) switcher, a heartbeat receiver, I/O filtering, and a virtual memory management unit (vMMU). vNative realizes the context switching through the cooperation between the VM switcher in the hypervisor and switch agency in guest VMs. The hypervisor manages the virtualized environment through the heartbeat scheme, vMMU, and I/O filtering. (BVM: background VM; FVM: foreground VM.)

### Guest VM modules

vNative introduces multiple modules in each guest VM:

- A heartbeat sender in the FVM periodically sends heartbeat messages to the hypervisor so that the hypervisor can detect FVM stalls when they occur.
- Native drivers are active in FVM to control devices on the platform because vNative lets the FVM access devices directly.
- For the context save process, the PCS driver reuses the OS's device suspension features to ask native device drivers to save device states to memory and then bring the devices to the initial (likely stateless) state. The context restore process works in the opposite manner.
- Switch agency provides an interface to implement a temporary VM switch scheme through the PCS driver, so that BVMs seem alive by moving to the foreground to handle the pending events (such as receiving messages and other notifications) and switching to the background immediately after completion, by means of expeditious context switching. Note that the temporary VM switch is off by default because of the BYOD practical running policy, and it must be initiated discreetly in accordance with the security allowance and power constraint.

These modules are implemented in the guest VM without difference for BVM and FVM, but only the native drivers module of FVM is active at a given time point.

### vNative hypervisor modules

vNative introduces the following additional modules to the hypervisor:

- *VM switcher:* The PCS driver module in the guest VM can self-support platform context save and restore, so the VM switcher in the hypervisor is primarily responsible for switching CPU states. During a VM switch, the CPU states of the running VM (FVM) are saved to memory, and the states of the new VM (one BVM) are restored.
- *vMMU with identical memory mapping:* The vMMU in vNative is responsible for mapping the pseudo-physical to the physical address, so that FVM can execute the native drivers with direct access. This is because some device drivers require physically continuous memory blocks for direct memory access (DMA) transmission, and others directly use the device host information. To address these issues, the pseudo-physical addresses are identically mapped so that physical continuity is always ensured and direct accesses to device host information from the guest driver cannot break the system execution.
- *I/O filtering for device access management:* vNative reuses Xen's virtual Advanced Programmable Interrupt Controller (APIC) solution for generating timer events to guest VMs, and intercepts storage accesses from guest VMs for storage partition. The hypervisor first receives interrupts from devices. Xen handles timer interrupts for time management and virtual timer interrupt generation, and device interrupts are forwarded to the FVM.
- *Heartbeat receiver module:* This module monitors the heartbeat messages sent from the FVM. When the message is absent for a configured time threshold, the FVM is considered stalled (and thus incapable of handling user requests). Then, a forceful VM switch resets the platform device states using the function-level reset feature of devices and resumes the next BVM. Unresponsive VMs switched to the background will reboot automatically when they are brought to the foreground again.

Overall, the general functionality of the vNative hypervisor is responsible for managing the platform resources and the runtime status for the hosted VMs.

### VM context switching procedure

vNative consolidates VMs with one FVM and multiple BVMs. Figure 1 shows the context switching procedure in vNative; the FVM is switched backward as a BVM, and one BVM is switched as the front end. The steps are as follows:

- Step 1: The switch agency informs the PCS driver in the FVM to save device context states and resets the devices to their initial states.
- Step 2: The PCS driver notifies each device driver to enter the suspend state by reusing the device suspend process. The drivers will then save the device context to memory. They will also bring the devices to their initial (likely stateless) states, using the PCI Express function level reset (FLR) feature, for example, so that one VM's device states will not leak to other VMs.
- Step 3: After saving the device context, the PCS driver in the FVM uses a hypercall to inform the hypervisor to perform a cooperative VM switch. This will save the (CPU) state of the previous VM and resume the (CPU) state of a BVM.

The restore procedure steps are similar to those of the save process, but executed in reverse order (see steps 4 through 6 in Figure 1).

## vNative implementation

Here, we introduce the implementation of vNative in memory management, disk access, and power management, along with the reliability isolation between the FVM and BVMs.

### Memory and storage isolation

vNative adopts static partitioning for providing exclusive access among the VMs for memory and storage space. Each VM is assigned one large contiguous chunk of memory pages. In vNative, VMs reside in memory once they boot up, and this is essential to provide a quick response time to VM switching by avoiding the cost of memory swapping.

vNative employs the logical block address (LBA) remapping mechanism to partition the storage securely and efficiently. Each VM must have its own storage that is inaccessible to other VMs. Because the Atom platform adopts an embedded multimedia card (eMMC) as storage with block-based accesses only, the storage blocks' addresses are represented by the LBA, which is identified by the argument of data transfer commands. vNative traps guest access of the storage device, checks command operation code, and remaps the guest LBA to the physical LBA with an additional LBA remapping for data transfer commands. The hypervisor will further ensure that a VM's accesses are limited to the VM's own partition.

Although we applied it to eMMC only, LBA remapping is generic enough to be used on any block-oriented storage interface. Moreover, vNative exploits hardware isolated memory regions (IMRs) in the Atom system-on-chip (SoC) platform to ensure one VM will never gain access to the other VMs' memory via DMA. Guest operating systems can also be enhanced to adopt memory ballooning technology if the guest has a large amount of free memory pages to donate at a time. This enhancement efficiently supports memory sharing by distinguishing the usage of physical memory into DMA pages, which requires identical memory mapping, and non-DMA pages.

## Power management

We introduce power management (PM) policies of vNative in terms of the device power state and CPU frequency and state controls.

- *Device PM:* vNative can use the guest power management policy, because—like the native platform—the FVM has exclusive access to all hardware devices. vNative guest Android OS employs the dynamic power management mechanism and makes the driver automatically enter a low-power state (such as D3) through the device PM control mechanism once the device is idle.
- *CPU frequency control PM:* vNative relies on the Android OS frequency manager to manage the processor frequency, by directly forwarding the corresponding machine state register (MSR) access right to the guest OS. For example, the Android OS uses MSRs APERF and MPERF to collect information on CPU usage and control the frequency according to the OS policy.
- *CPU state control PM:* In vNative, all VMs are run in ring 1 and are not privileged to execute the instructions. vNative exposes the full hardware C-state control capacity to the FVM, in reusing the well-tuned OS C-state management features. This lets vNative achieve close to native power efficiency.

vNative implements power management policies to make the platform enter a low-power consumption state adequately.

## VM switch integrity and acceleration

A preemptive VM switch mechanism could require the hypervisor to save and restore the platform context at an arbitrary point, which requires complicated device knowledge by the hypervisor and a large engineering effort to forcefully save and restore the platform context. To ensure the state integrity, vNative implements a simplified but efficient cooperative VM switch mechanism—that is, the VM switch's integrity is ensured by the VMs themselves in a safe and convenient point by the device suspend process. That is, the VM, which initiates a VM switch, should ensure that a switch happens at a safe point and reuse the native device driver for platform context save and restore.

vNative further optimizes the VM switch process by replacing power operations inherited from device suspend and resume processes with function-level resettings. The device suspend and resume processes are adequate for device state saving and restoring, but they take about 1 second to complete because they must wait for hardware to finish

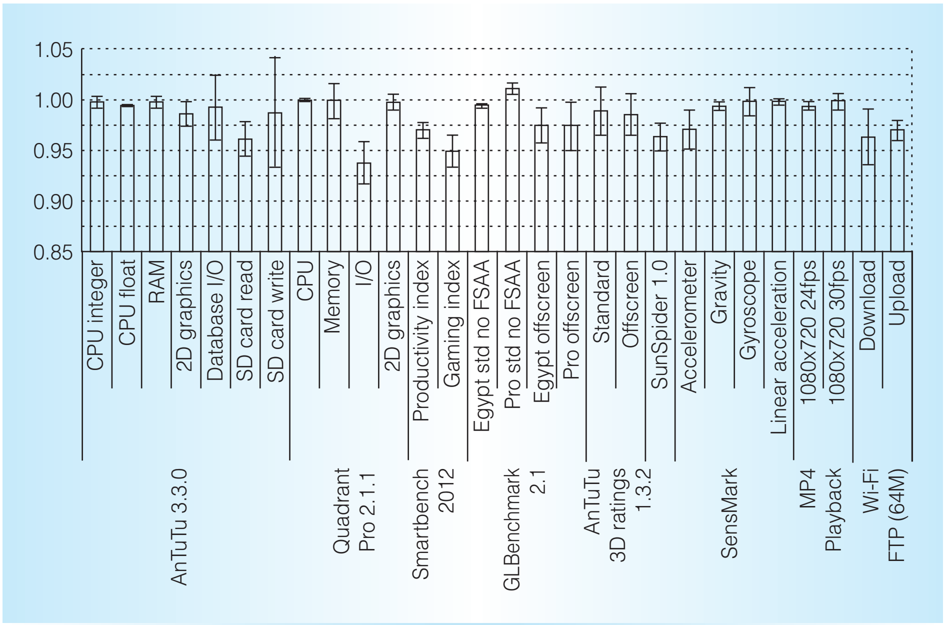

Figure 2. Normalized performance of vNative in comparison with native performance. Various benchmarks are used to evaluate different performance aspects of vNative, including CPU integer/float computing, memory and I/O throughput, graphics processing, video playing, and FTP transmission performances.

powering operations on or off. We remove the time-consuming power operations to reduce VM switch time and add function-level resettings to prevent device configuration detail leakage, which the power operations had previously handled. For devices that do not support function-level resetting, we manually reset the registers to their initial values.

## Evalution

We evaluated the runtime performance, power consumption, and switching time cost of the vNative prototype compared with a native Android system with smartphone benchmarks under different use cases. We call these environments *native* and *vNative*, respectively.

### Evaluation testbed configuration

The testbed is on the Intel Z2460 Medfield smartphone platform. The platform is based on the Atom Z2460 SoC model, which contains a single-core 1.6 GHz processor with hyperthreading, 1 Gbyte of low-power double data rate 2 (DDR2), 400 MHz of memory, and a PowerVR SGX 540 GPU. A 16-Gbyte eMMC serves as the storage device. In addition, the platform has a 4.03-inch display covered with Atmel maXTouch 224 touchscreen and various sensors, including an accelerometer, a gyroscope, a magnetometer, an ambient light sensor, and a proximity sensor.

vNative's implementation is based on a modified Xen hypervisor and two Android 4.2 instances run as paravirtualized and privileged VMs. Each VM is assigned two virtual CPUs pinned to the corresponding physical CPU thread, along with 430 Mbytes memory and 7.42 Gbytes of eMMC storage. In native, a single Android instance uses the same amount of memory and eMMC storage, which is evaluated to compare fairly with the performance of a VM in vNative.

We evaluated the overhead of vNative in a series of benchmark scenarios, compared with native performance as the normalized factor.

We used various smartphone benchmarks to comprehensively evaluate the runtime throughput performance of vNative, including AnTuTu 3.3.0, Quadrant Pro 2.1.1, Smartbench 2012, GLBenchmark 2.1, AnTuTu 3D Ratings 1.3.2, SunSpider, SensMark, MP4 Playback, and Wi-Fi FTP Transmission. The tested results show their functionalities, and a detailed introduction of these benchmarks can be retrieved from Google Play.

## Native vs. vNative performance

We used benchmark scores, video playback frames per second (fps), and file transmission time as metrics of runtime performance. Each benchmark or test case has been repeated four times to measure the average performance along with its variation. Figure 2 shows the performance results of the vNative with the 95 percent confidence intervals for the statistic interval estimates, and the results are illustrated with normalized values to the average performance of native environment. Therefore, we can merge all results in a single figure to show vNative's overhead. Because vNative's overhead is small, we plot the *y*-axis starting with 0.85 in Figure 2.

*CPU.* In Figure 2, CPU performance of vNative achieves 99.80 percent of native performance in AnTuTu integer tests, 99.43 percent in AnTuTu floating-point tests, and 99.97 percent in Quadrant CPU tests. The difference of CPU performance between native and vNative is less than 0.6 percent. The confidence intervals of both tests are no more than 0.6 percent, which indicates a better precision compared to results of graphics and eMMC storage. This is reasonable because processors are assigned to the vNative VMs exclusively, and the interrupt rate of CPU-intensive workloads is lower than I/O-intensive ones.

*Memory.* vNative guest achieves 99.80 percent and 99.88 percent of native performance in the AnTuTu RAM and Quadrant memory tests, respectively. This means that vNative's memory performance overhead is trivial for maintaining a direct page table because page table operations are not frequent in the tests.

*eMMC access.* The AnTuTu benchmark test indicates that, compared to native, vNative achieves 99.30 percent in database I/O performance, 96.11 percent in SD card reading, and 98.76 percent in SD card writing. (AnTuTu refers to "external storage" as an SD card, whereas in our platform it is actually an eMMC). Quadrant I/O tests show that vNative achieves 93.77 percent of native performance. At the same time, variance of the results is notably higher in these tests compared to the CPU and memory tests. This unpredictable I/O performance is introduced by the low random write performance of the eMMC than can be observed in both native and vNative.

*Multimedia.* Concerning 2D graphics, vNative achieves 98.62 percent of native performance in AnTuTu 2D tests and 99.77 percent in Quadrant 2D tests. For 3D graphics performance, vNative's performance is 99.45 percent and 101.08 percent of the native in GLBenchmark Egypt and Pro tests in standard rendering mode without Fast Approximate Anti-Aliasing (FSAA), and 97.45 percent and 97.42 percent in GLBenchmark Egypt and Pro tests in offscreen mode. At the same time, vNative achieves 98.92 percent and 98.57 percent native performance in AnTuTu 3D Ratings standard and offscreen mode tests. Because vNative achieves native performance in all of the 3D standard mode tests, we ascribe vNative's performance advantage over native in the GLBenchmark Pro standard test to measurement variance. As for MP4 video playing, vNative's average fps is 99.37 percent and 99.83 percent compared to the native for 24-fps and 30-fps movies, respectively.

*Wi-Fi.* We evaluate the data transmission time to download or upload a 64-Mbyte file, where the average vNative data transmission rate achieves 96.33 percent when downloading and 96.99 percent when uploading, compared to the native. Variance in file downloading time is larger than uploading time because it involves eMMC writing, which suffers from strong random performance because of the eMMC writing attributes.

*Others.* The other benchmarks we used include SunSpider 1.0 on JavaScript, SensMark on sensor event accuracy, and Smartbench on overall system performance. vNative scores 96.33 percent of the native in

SunSpider. In SensMark, vNative scores 97.10 percent for accelerometer, 99.36 percent for gravity sensor, 99.83 percent for gyroscope, and 99.82 percent for linear acceleration, when normalized to the native performance. Furthermore, vNative achieves 96.96 percent in the Smartbench Productivity Index and 94.90 percent in the Smartbench Gaming Index.

In summary, vNative's runtime performance overhead is no more than 6.3 percent (obtained in eMMC benchmark testing), and its average overhead is only approximately 1.4 percent compared with the native performance (vNative achieves 98.6 percent performance of native overall).

## Power efficiency

To illustrate vNative's power efficiency, we establish five mobile device daily operation scenarios to compare the performance between the native and vNative environments:

1. Standby in flight mode with cellular and Wi-Fi connections disabled.
2. Standby with only Wi-Fi connection active.
3. MP3 playback, wherein a 320-Kbits per second MP3 is played with headset on and screen off.
4. MP4 playback with headset and screen on for displaying a 480 × 360 pixel, 60-fps MP4, and audio is played through the headset.
5. Game playing with Rovio's Angry Birds game. The headset is unplugged and sound effects are played through the speakerphone.

We used a Monsoon power monitor to measure the practical platform power consumption. Figure 3 shows the results. Under the two standby scenarios (scenarios 1 and 2), vNative consumes about 14.00 and 38.78 mW, whereas native uses 14.25 and 35.66 mW, respectively. In addition, vNative and native consume, respectively, 155.74 and 156.20 mW in scenario 3; 1,091.99 and 1,043.70 mW in scenario 4; and 1,173.79 and 1,101.37 mW in scenario 5. The experiments indicate that the highest power overhead of vNative is 8.7 percent, obtained in scenario 2; vNative even consumes less power than native in scenario 1, and no more than 6.5 percent under all other scenarios.

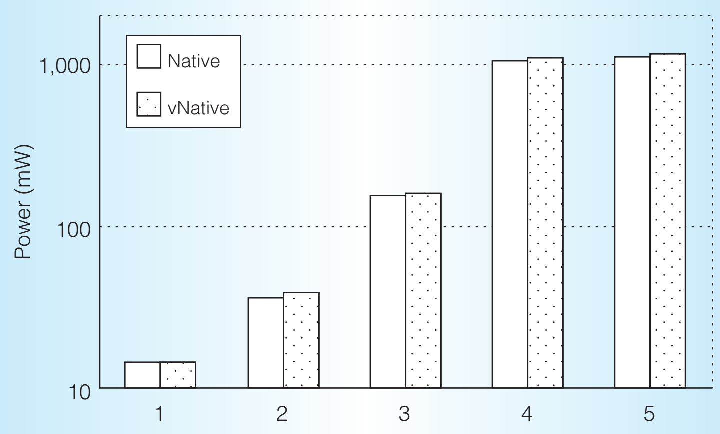

Figure 3. Platform power consumption. vNative incurs a light overhead in power consumption, because it presents the exactly identical native platform to FVM so that FVM can effectively reuse the native OS power management policy.

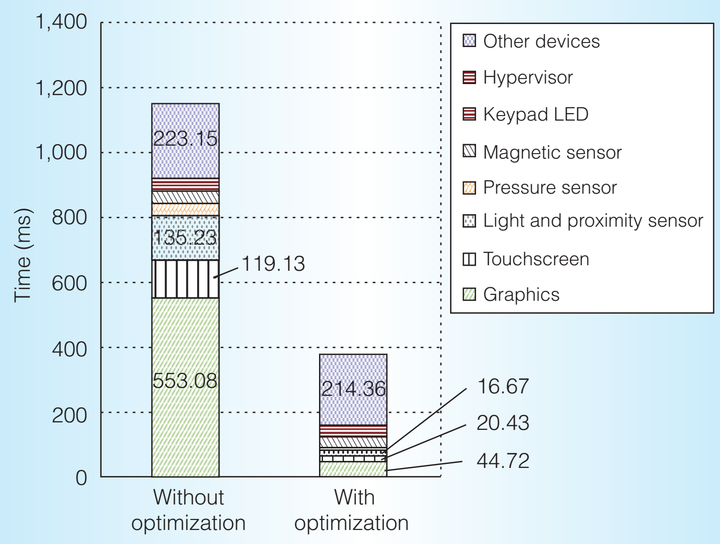

Figure 4. VM switching cost before and after removing unnecessary power operations. The optimization can significantly shorten the VM context switching time from about 1,149 ms to 376 ms, wherein 473 ms of speedup comes from VM switching acceleration of device suspend and resume process.

## VM switching time cost

The switching cost is the time spent switching a BVM forward to be the FVM to interact with users. We obtained it by reading and printing values of the time-stamp counter (TSC) at the beginning and end of a VM switch, to show the dominant time consumers in the process. We also collected the time spent in each device's suspend and resume subroutine. Figure 4 shows the switching cost of a VM switch with and without acceleration.

The time costs in Figure 4 are divided into eight parts, seven of which are consumed by device suspend and resume subroutines. The graphics subsystem originally takes 553.08 ms; the touchscreen and light/proximity sensor require 119.13 and 135.23 ms, respectively. After applying the optimization, the time needed for those devices to complete the suspend and resume processes is reduced to 80 ms in total (44.72 + 20.43 + 16.67 ms), and the time cost of a switch decreases to about 376 ms, dominated by the "other devices" part, which comprises suspending and resuming about 350 devices, including the battery, eMMC storage, general-purpose I/O, serial ports, and buttons.

vNative expeditious context switch reduces the VM switch time to 375.72 ms from 1149.58 ms of the traditional scheme. In other words, because more than 90 percent of the time is spent in suspending and resuming devices, and removing unnecessary power operations, the time cost of a VM switch can be reduced by 67 percent.

We can amend vNative's VM context switching procedure to improve user experience according to the practical use cases, which we do not detail in this article. Concretely, vNative can sequence the devices' save-and-restore order on the basis of their impact on user experience. That is, the display device, such as the graphics device, should be sequenced at the first position so that the user can have a vision of switched VM foremost. The secondary position can be the human input devices, such as the touchscreen and key button. vNative can take the lazy device state save-and-restore mechanism to reduce the latency of VM switches, because performing a save-and-restore of the entire platform state can be completed much more quickly, thereby improving the user experience. Moreover, vNative's switch agency can provide more pervasive functionalities to enable BVM to adequately respond to external events, such as healthcare notifications or incoming calls or messages.

MICRO

## References

1. K.W. Miller, J. Voas, and G.F. Hurlburt, "BYOD: Security and Privacy Considerations," *IT Professional*, vol. 14, no. 5, 2012, pp. 53–55.
2. H. Lv et al., "Virtualization Challenges: A View from Server Consolidation Perspective," *ACM SIGPLAN Notices*, vol. 47, no. 7, pp. 15–26.
3. H. Liu et al., "Live Virtual Machine Migration via Asynchronous Replication and State Synchronization," *IEEE Trans. Parallel and Distributed Systems*, vol. 22, no. 12, 2011, pp. 1986–1999.
4. P. Barham et al., "Xen and the Art of Virtualization," *Proc. 19th ACM Symp. Operating Systems Principles* (SOSP 03), 2003, pp. 164–177.
5. D. Gupta et al., "Difference Engine: Harnessing Memory Redundancy in Virtual Machines," *Comm. ACM*, vol. 53, no. 10, 2010, pp. 85–93.
6. Y. Dong et al., "High Performance Network Virtualization with SR-IOV," *J. Parallel and Distributed Computing*, vol. 72, no. 11, 2012, pp. 1471–1480.
7. K. Adams and O. Agesen, "A Comparison of Software and Hardware Techniques for x86 Virtualization," *Operating Systems Rev.*, vol. 40, no. 5, 2006, pp. 2–13.
8. H.A. Lagar-Cavilla et al., "VMM-Independent Graphics Acceleration," *Proc. 3rd Int'l Conf. Virtual Execution Environments* (VEE 07), 2007, pp. 33–43.
9. J. Andrus et al., "Cells: A Virtual Mobile Smartphone Architecture," *Proc. 23rd ACM Symp. Operating Systems Principles*, 2011, pp. 173–187.
10. G. Heiser and B. Leslie, "The OKL4 Microvisor: Convergence Point of Microkernels and Hypervisors," *Proc. 1st Asia-Pacific Workshop Systems* (APSys 10), 2010, pp. 19–24.
11. M. Lange et al., "L4Aandroid: A Generic Operating System Framework for Secure

Smartphones," *Proc. 1st ACM Workshop Security and Privacy in Smartphones and Mobile Devices*, 2011, pp. 39–50.

12. J. Andrus et al., "Cells: A Virtual Mobile Smartphone Architecture," *Proc. 23rd ACM Symp. Operating Systems Principles* (SOSP 11), 2011, pp. 173–187.

**YaoZu Dong** is a PhD candidate in the Department of Computer Science and Engineering at Shanghai Jiao Tong University and a software architect in the Intel Open Source Technology Center. His research interest focuses on architecture and systems, including virtualization, operating systems, and distributed and parallel computing. Dong has a master's degree in electronic engineering from Shanghai Jiao Tong University. Contact him at dongyaozu@gmail.com.

**JunJie Mao** is a PhD candidate in the Department of Computer Science and Technology at Tsinghua University. His research interests include virtualization architecture, static bug detection in operating systems, and dynamic failure diagnosis in large-scale software systems. Mao has a BS in computer science and technology from Tsinghua University. Contact him at eternal.n08@gmail.com.

**HaiBing Guan** is a professor in the Department of Computer Science and Engineering at Shanghai Jiao Tong University. His research interests include computer architecture, compiling, virtualization, and hardware–software codesign. Guan has a PhD in computer science from Tongji University. He is a member of IEEE and the ACM. Contact him at hbguan@sjtu.edu.cn.

**Jian Li** is an associate professor in the School of Software at Shanghai Jiao Tong University. His research interests include embedded systems and virtualization, real-time scheduling theory, network protocol design, and quality of service. Li has a PhD in computer science from the Institut National Polytechnique de Lorraine. He is a member of IEEE and the ACM. Contact him at li-jian@sjtu.edu.cn.

**Yu Chen** is an associate professor in the Department of Computer Science and Technology at Tsinghua University. His research focuses on operating systems, including virtualization, optimization for multicore architecture, system security, and distributed and parallel computing. Chen has a PhD in computer science from the National University of Defense Technology. Contact him at yuchen@mail.tsinghua.edu.cn.