MobiXen: Porting Xen on Android Devices for Mobile Virtualization

Yaozu Dong[1,2], Jianguo Yao[1], Haibing Guan[1]
[1]*Shanghai Jiao Tong University*
Email: {jianguo.yao, hbguan}@sjtu.edu.cn

Ananth Krishna R[2], Yunhong Jiang[2]
[2]*Intel Asia-Pacific Research and Development Ltd.*
Email: {eddie.dong, ananth.krishna.r, yunhong.jiang}@intel.com

***Abstract*—The mobile virtualization technology provides a feasible way to improve the manageability and security for embedded systems. This paper presents an architecture named MobiXen to address these challenges. In the MobiXen, both Xen's physical memory space and virtual address space are shrunk as much as possible and thus Android owns more memory resource; optimizations are developed to reduce the virtualization overhead when Android is accessing system resources; new policies are implemented to achieve low suspend/resume latency. With these work adopted, MobiXen is customized as a high efficient mobile hypervisor. Detailed implementations shows that, most of the performance degradation brought by MobiXen is less than 3%, which is imperceptible by end users.**

***Index Terms*—Mobile Virtualization, Xen, Android, hypervisor**

1. Introduction

Like viruses that can infect PCs, there are also various security issues that can affect mobile devices, and pose an extraordinary threat to users' critical and sensitive data [1]. Meanwhile, with the popularity of BYOD and mobile officing, a lot of business-related content is also stored in these mobile devices, such as emails, confidential conversations, technical documents, etc. The security issues in respect to these business data are even more critical for today's smartphones and tablets [2]. How to protect these critical resources on mobile devices, whether personal or business, remains an open problem for both academics and industry.

To address the problem, we intend to follow the direction of Overshadow [3], [4], which uses virtualization technology to protect the privacy and integrity of application data, even in the event of a total OS compromise. First, a hypervisor sitting behind the OS is widely used to provide additional security, protecting applications from a tampered OS or providing a secure execution environment for the applications without OS intervention, such as McAfee DeepSafe[1], and providing a snapshot of the OS state for integrity measurement, migration, malware detection, correctness validation and other purposes [5]. It is also widely used to manage the OSs, such as in dynamic OS patching [6]. Second, the promise of virtualization technology is to centralize applications to make them easier to manage and aid provision, while stretching hardware resources and keeping nagging software conflicts to a minimum in the bargain.

In this paper, we propose an architecture named MobiXen which ports Xen on Android Devices for mobile virtualization[2]. In this architecture, Android is chosen as the platform as it is expected to own over 80% of the smartphone OS market by the end of 2014, and the open source hypervisor is used for the reason that our solution can be easily adopted by various mobile system designers for their follow-up research and product. In the meantime, we choose Xen, type-I open source hypervisor, for the merits of the potential opportunity to minimize the trusted computing base (TCB) of hypervisor comparing with other source hypervisor, such as the type-II hypervisor KVM. The contributions of this paper are summarized as follows: 1)We propose MobiXen, which can support sophisticated manageability and security for mobile applications.2)MobiXen extends Xen to support mobile use cases including small memory footprint and fast system reboot. 3)MobiXen improves page cache allocation and thread switching by utilizing free GDT entries. 4) We give comprehensive measurements with a variety of mobile benchmarks including CPU, memory, and multimedia for the proposed MobiXen.

2. Overview of MobiXen

To extend Xen to support mobile use cases, we propose MobiXen, in which Android acts as domain 0 and is the only domain running on top of Xen. Therefore the users' expectation towards this special Android domain 0 is different from traditional solutions. For example, it has high performance, low power consumption and stability. MobiXen runs Android as the guest domain and only domain with direct access to the hardware resources like Dom0 in traditional Xen architecture. In the meantime, MobiXen runs Android as a 32-bit Xen paravirtualized virtual machine to take advantage of paravirtualization in a less powerful small core platform. The general Android on MobiXen system architecture is presented in Fig. 1. In this architecture, Xen

978-3-9815370-8-6/17/$31.00

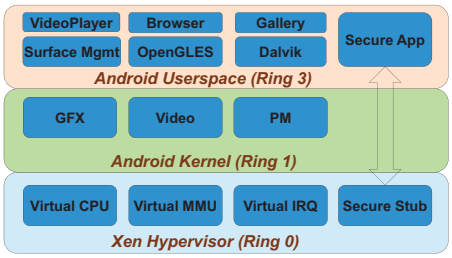

Figure 1. Android on MobiXen System Architecture.

is located in system ring 0 and Android in ring 1 and ring 3. Xen presents virtual CPU, memory and interruption to the Android guest, but for hardware devices, like GPS, WiFi and telephoning, Android directly drives them to get rid of the virtualization cost and achieves high performance.

MobiXen runs secure applications together with traditional but might-be-not secure applications in Android guest. MobiXen creates a trusted environment for the secure applications such as the virus scan software, and secure components of an application such as the encryption/decryption algorithm of an e-payment application.

3. Optimizations of MobiXen

User experience is one of the most important indicators for end users that judge whether a mobile device is easy to use. To facilitate the new mobile usage case, several optimizations needs to be applied to MobiXen.

3.1. Improving I/O performance

When running the Quadrant[7] I/O benchmark in Xen-enabled Android, its score varies on a wide range. Among hundreds of tests, only a few times could the I/O score get near native performance, while for most of the time, it showed about a 30% downgrade compared to the native Android score. For Quadrant I/O issues, the Ftrace profiling results show that there is a burst of page cache allocation operations happened when running the benchmark. On native Android, such page table modification overhead is very small since it is only a memory write operation. However on Xen-enabled Android, the overhead is amplified significantly because Android's page table is write protected, and each time when Android tries to modify its page table, it involves a page fault, instruction emulating and complex page structure handling. Besides, Xen needs to audit such operations for security and correctness checks. Compared with a single memory write cost on native Android, this cost on Xen-enabled Android is much higher, which might be as many as several thousands of instruction cycles.

Since the Quadrant I/O performance downgrade case in Xen-enabled Android only happens when the page cache memory is allocated in high memory zone, one straightforward idea is to force such page cache allocation in the memory normal zone, so that the memories have pre-setup 1:1 mappings in kernel to avoid the possibility of frequent kmap() operations. This solution introduces an immediate improvement for the Quadrant I/O benchmark on Xen-enabled Android, making the performance score stable and near native Android. Putting system page cache in low memory could greatly enhance the I/O related benchmarks. However this solution also has its side effects, which might put the low memory in short supply for other components. To minimize this impact, following memory related optimizations are needed to let Android own more virtual address space and physical memory.

3.2. Accelerating thread switch

How to minimize the thread-local storage (TLS) switching cost is a key point in improving the thread switching efficiency. In the current x86 architecture, the biggest GDT entry number that a system can use is 8192. However in reality, only less than 32 entries are used in the real Linux implementation. One idea for improvement is to leverage these unused GDT entries, assigning different threads to hold each TLS. When a new thread is created, a new group of GDT TLS entries will be allocated, and it will be loaded when it is running on a certain CPU. When these threads are exited, the related GDT TLS entry slots will also be freed. Suppose there are two threads created with the above rules, one thread using the GDT no. 33, 34 and 35 entries to hold TLS, while another thread uses GDT no. 36, 37 and 38 for its own TLS. When scheduling the two threads in the same CPU, GDT TLS entries do not need to be switched since they are located in different GDT entries. However, although the total number of GDT entries is more than 8000, they may still be used up if plenty of threads are created. In this case, all the threads created later will fall back to the original scenario that uses GDT 6, 7 and 8 entries and will perform TLS switch on thread scheduling.

By eliminating most of the TLS switch, this optimization can reduce the thread switching cost a lot, especially when Xen is enabled. This optimization is a typical use case to change spaces and resources with time in certain workloads that are sensitive to execution efficiency.

3.3. Reduce system suspend/resume latency

Android has followed Linux in implementing its suspend/resume process. The latency of resuming from suspended state is a key indicator for evaluating user experience for mobile devices. For example, Microsoft's connected standby requires that the latency must be less than 500ms [8]. Therefore how to achieve a high efficiency suspend/resume process becomes a big challenge for Xen-based Android. A new S3 suspend/resume method is proposed to address this issue. For these non-boot CPUs, they will not really go to offline state; instead Xen will put them into a deep C6 state with all context saved. After an S3 resuming, non-boot CPUs are woken up by boot CPUs through generation of general IPIs or touch monitor variables, and then they continue to run from the original code path. From the CPU perspective, it looks like it is paused after one instruction, and S3 suspend

and resume become transparent to all non-boot CPUs. CPU0 (boot CPU) will call the disable_non-boot_cpus() function to suspend all the other CPUs (CPU1-CPU3). Here is the suspend process:

1) For each non-boot CPU shutting down process, Android will issue a vCPU down hypercall into Xen.
2) Xen will bring down its related vCPU, and issue an mwait instruction to halt the current physical CPU.
3) When all non-boot CPUs are suspended, CPU0 will call mwait hypercall into Xen, and Xen will issue a real mwait instruction to halt physical CPU0.
4) When all CPUs enter the deep sleep mode, the whole system is suspended.

The resume process is interpreted as follows:

1) When the system receives a wakeup notification (e.g. user presses the power key), CPU0 will wake up first.
2) CPU0 will issue IPI or touch monitor variable to all non-boot CPUs (CPU1CPU3). Thus these non-boot CPUs will leave the deep sleep state.
3) In the resume process, CPU0 will call enable_nonboot_cpus() function to bring up all non-boot CPUs.
4) Each non-boot CPU will issue vCPU up hypercall into Xen.
5) Xen brings up all the vCPUs and the whole system is resumed.

In this way, S3 resuming latency can be reduced from 700ms to 330ms, and we get 2x faster response from a user experience point of view.

3.4. Reduce Memory Footprint

In the current architecture, Xen and Android Dom0 live in the same virtual address space, so the 168M memory used by Xen is borrowed from Android virtual address space. Besides, the 168M size is designed for generic Xen, thus some areas are larger than what the mobile virtualization scenario really needs. There is much room for shrinking of Xen's virtual address space to let Android Dom0 own more. For example, the target device's system memory is 1GB, and 6MB is enough for its frame-info table, and 1MB is enough for the machine-to-physical translation table. By customizing such redundant address spaces according to the specific hardware requirements, the virtual address footprint for Xen hypervisor could be finally reduced to 32MB. This would allow Android Dom0 to own more normal zone memory. Table 1 shows the virtual memory usage improvement from such customization. After the memory customization for a certain configured device, the total physical memory occupied in Xen is only 10MB.

Another method of allowing Android to own more normal zone memory is to reduce the vmalloc size. Since the vmalloc area belongs to the system's high memory zone, the larger the vmalloc area is, the smaller the normal memory zone that Android Dom0 can own. Reducing the vmalloc size to an acceptable size could also mitigate the burden of the normal memory zone. Since these optimizations change the system behavior, a lot of system-wide measurements were performed to verify that it does not bring negative impacts to the whole system.

TABLE 1. VIRTUAL ADDRESS USAGE COMPARISON.

Virtual Address Usages	w/o	w/
Direct-map area	12 MB	4 MB
I/O remapping area	4 MB	4 MB
Per-domain mapping	8 MB	8 MB
Shadow linear page table	8 MB	0 MB
Guest linear page table	8 MB	8 MB
M2P translation table (both r/o and r/w)	32 MB	2 MB
Frame table	96 MB	6 MB
Total	168 MB	32 MB

4. Experiments

The experimental prototype was implemented in an Intel Clovertrail+ device, which is equipped with a dual-core 2.0GHz SoC together with 1GB memory. In this prototype, all Android's functionalities were verified to work smoothly in the virtualized environment, including telephoning, text messaging, WiFi, 3G, GPS, web browsing, 3D gaming, video/audio playback, etc. For performance, there is no obvious degradation between virtualized Android and native Android. Besides, the system stability for Android on MobiXen is quite good; it is even qualified for product usage.

4.1. Configuration

To have a better understanding of the performance characteristics after introducing the optimized virtualization technology in mobile devices, a full round of measurements was performed for Android on MobiXen and Android native scenarios and a comparison was made between the two sets of data. The measurement is based on the Intel Clovertrail+ smartphone that is equipped with 2GHz CPU, 1GB memory and a PowerVR GPU.

The implementation used for measurement is based on Xen version 4.1 and Android Open Source Project version 4.2.2. Both Android and Xen work under 32-bit mode. For performance measurement, the benchmarks are mainly divided into 3 categories: CPU, memory and graphics.

4.2. Performance comparisons

Figure 2 shows the CPU performance comparison between Android on MobiXen and native Android. By pinning all vCPUs into physical CPUs, most of the CPU benchmarks can gain more than 99% of the performance of native Android. Even for EEMBC and SmartBench 2012 productivity, the performance downgrade introduced by Xen hypervisor is less than 4%.

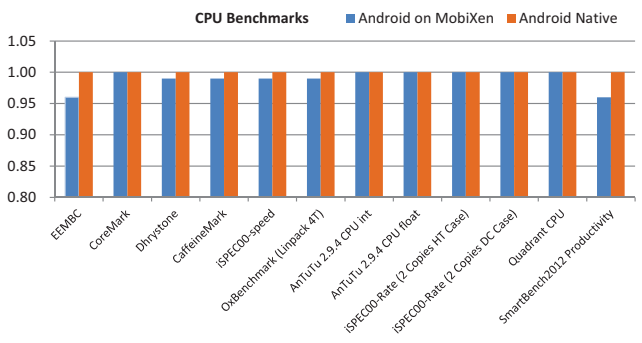

Figure 2. CPU Benchmark Results.

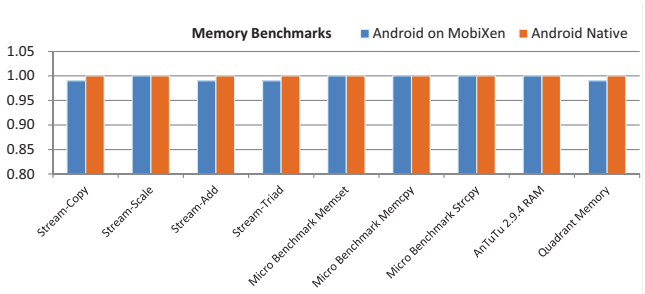

Figure 3. Memory Benchmark Results.

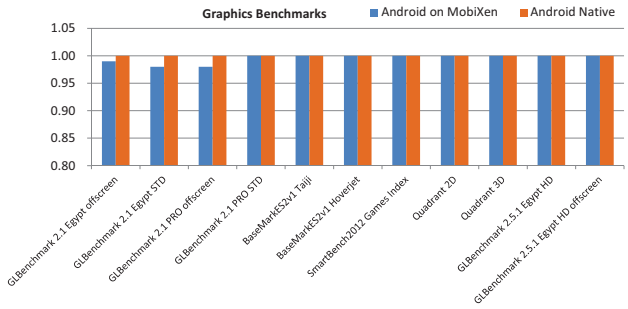

Figure 4. Graphics Benchmark Results.

Figure 3 shows the memory benchmarks comparison between Android on MobiXen and native Android. Since Android on MobiXen uses the directed page table mechanism, when the memory test application is running, all its page tables have been set up, so there will be very few memory-related hypercalls triggering in runtime. The result shows that all memory benchmarks running in virtualized Android can gain more than 99% of the effectiveness of the native Android case.

Figure 4 shows the graphic performance comparison between Android on MobiXen and native Android. In this solution, graphic devices are directly assigned to Android, and swiotlb cost is even eliminated by using 1:1 mapping between Android memory and host physical memory. Therefore Android running on Xen could gain more than 97% of the performance of Android native in all graphics test cases.

5. Conclusions

In this architecture, OEMs and 3rd party developers could utilize the hypervisor capability to design various policies to protect sensitive code and data. After studying the differences between server virtualization and mobile virtualization, various optimizations have been adopted to minimize the virtualization overhead and customize Xen to be a highly efficient mobile hypervisor. As for the real user experience, the whole system runs very smoothly and users are even not aware that the Android is running in a virtualized environment. Detailed measurement data shows that most of the performance impact brought by Xen is less than 3%, which is not noticeable to end users.

Acknowledgments

Dongxiao Xu helped with development, implemented MobiXen, and assisted us with hardware bring up. This work was supported in part by the Program for NSFC (No.61525204, 61572322), the STCSM project (No.16QA1402200), Aeronautical Science Foundation of China (No. 20145557010) and the Shanghai Aerospace Science and Technology Innovation Fund.

References

[1] J. Ren, Y. Qi, Y. Dai, X. Wang, and Y. Shi, "Appsec: A safe execution environment for security sensitive applications," in *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '15, 2015, pp. 187–199.

[2] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel, "Inktag: Secure applications on an untrusted operating system," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '13. New York, NY, USA: ACM, 2013, pp. 265–278. [Online]. Available: http://doi.acm.org/10.1145/2451116.2451146

[3] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports, "Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. New York, NY, USA: ACM, 2008, pp. 2–13.

[4] A. Baumann, M. Peinado, and G. Hunt, "Shielding applications from an untrusted cloud with haven," in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Broomfield, CO, Oct. 2014, pp. 267–283.

[5] A. Srivastava, H. Raj, J. Giffin, and P. England, "Trusted vm snapshots in untrusted cloud infrastructures," *In proceedings of 15th International Symposium On Research in Attacks, Intrusions and Defenses (RAID) Symposium*, 2012.

[6] H. Chen, R. Chen, F. Zhang, B. Zang, and P.-C. Yew, "Live updating operating systems using virtualization," *In proceedings of the 2nd International Conference on Virtual Execution Environments(VEE)*, pp. 35–44, June 2006.

[7] Xen, "Xen power management," http://wiki.xen.org/wiki/Xen_power_management.

[8] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. Martins, A. V. Anderson, S. M. Bennett, A. Kagi, F. H. Leung, and L. Smith, "Intel virtualization technology," *In proceedings of IEEE Computer Society Press*, pp. 48–56, May 2005.