Remote Graphical Processing for Dual Display of RTOS and GPOS on an Embedded Hypervisor

Hyunwoo Joe, Dongwook Kang, Jin-Ah Shin, Vincent Dupre, Soo-Young Kim, Taeho Kim and Chaedeok Lim
Embedded SW Research Department, Electronics and Telecommunications Research Institute (ETRI)
Daejeon, Republic of Korea
{hwjoe, dkang, jashin, vdupre, sykim, taehokim, cdlim}@etri.re.kr

***Abstract*— In this paper, we introduce a remote graphics library framework based on inter-virtual-machines communication in an embedded hypervisor. With this framework, there are no causality conflicts when multiple guest operating systems share one GPU. We adopted API remoting for GPU virtualization because it has relatively small overhead when connected with OpenGL ES standard library on embedded hypervisors. Interferences from the hypervisor during synchronization between front-end and back-end can be reduced by inter-VM commutation. To make improvement on size, weight and power for embedded systems, we opted for displaying both guest operating systems on a single display panel. The presented framework is applied to a real-world embedded hypervisor used for safety-critical systems. Our implementation runs an automotive digital instrument cluster on a real-time guest operating system and an in-vehicle infotainment application on a general purpose guest operating system within the hypervisor. We found it feasible for an embedded hypervisor to provide GPU service to heterogeneous industrial guest operating systems on a single hardware platform.**

Keywords*— *GPU virtualization, dual display, embedded hypervisor, embedded virtualization, API remoting, OpenGL ES, Inter-VM communication

I. INTRODUCTION

In recent years, the widespread use of smartphones and tablet PCs has spurred even industrial systems to embrace graphics processing unit (GPU) [1][2]. Demand has risen for high-performance multimedia graphical applications to run with real-time applications even though the conventional role of embedded system software for industry applications primarily has been mechanical control. Some research in industrial domain claims that a real-time operating system (RTOS) and a general purpose operating system (GPOS) need to share a single piece of embedded hardware [3][4]. For this sort of systems, virtualization technology can be one of the most practical options available.

When an embedded system is virtualized, it generally hosts one RTOS guest and one GPOS guest [5] and its hypervisor manages a control system in real time and supports execution of heavy multimedia applications simultaneously. However, the majority of research for embedded virtualization has focused on virtual machine manager, the purpose of which is managing system resources to keep the system robust [6][7][8].

978-1-4673-7929-8/15/$31.00 ©2015 IEEE

There are only a few examples of virtualization for embedded multimedia IO devices, but GPU virtualization in server markets has been actively researched in various contexts such as I/O pass-through [9], para-virtualization, full virtualization [10] and API remoting for graphics library or device drivers [11]. Despite its markedly small overhead, I/O pass-through is not used in our work since GPUs in embedded hardware do not support virtualization. According to Suzuki [10], the performance of para-virtualization is slower by two to three times whereas full-virtualization exhibits a different scale of overhead due to increased memory-mapped I/O operations. API remoting is also another well-known method for its relatively small overhead than full and para virtualization, but a few lines of source has to be appended to both the host and guest ends for deployment. It is similar to remote procedure call (RPC) so actions can be performed remotely via the graphic software stack such as OpenGL ES or at the device driver.

In this paper, we present a framework for remote graphics library using inter-VM communication for virtualized embedded systems. Commands can be processed without inducing conflicts between guests when multiple guest operating systems share one GPU in this framework. We use API remoting with optimization techniques since it has smaller overhead than full and para virtualization. Furthermore, there are no GPU devices that has hardware assisted virtualization feature in embedded systems. We choose OpenGL ES for API remoting because it is the most widely used platform for multimedia applications in embedded systems. Hence our framework can be readily used in industrial systems

API remoting generally carries communication overhead when the front-end relays APIs to the back-end library. Such overhead is mainly caused by hypervisor invasion between the front-end and the back-end. In order to make an improvement on this overhead, our implementation incorporates transparent shared block policy between the front-end and the back-end so that communication activities between VMs and interference from hypervisor could incur reduced overhead. For user's convenience, dual display is supported and each guest operating system can render graphics output without considering the other OS at all.

With the presented framework, operating systems without a GPU device driver can access OpenGL ES and GPU simply by adding our library. In addition, the framework also makes it possible to display control information and multimedia

contents side-by-side, so it is a useful enhancement to most virtualization platforms used in industrial settings, e.g., automobile, factory automation, avionics, and military embedded systems. Our framework includes a virtualized remote OpenGL ES library and a dual display module. We implemented and tested the framework on Qplus-Hyper [12], a hypervisor tailored for safety-critical embedded systems. The hardware platform for our implementation is an embedded development board housing Exynos5250 [13]. For killer applications that need real-time responsiveness and high-performance for graphics, the presented framework runs an automotive digital instrument cluster above its RTOS layer and in-vehicle infotainment (IVI) on top of the GPOS layer. The graphic output of both guests uses the dual display module. In this work, we explore GPU virtualization for embedded hypervisor and observe that it is possible to host heterogeneous industrial systems on a single hardware platform.

This paper is organized as follows. Section II introduces Qplus-Hyper, the hypervisor used in this work. In Section III, the proposed virtualized remote graphics processing is explained. Section IV demonstrates our implementation. The paper is concluded in Section V with a summary of on-going studies with directions for future work.

II. Qplus-Hyper: Embedded Hypervisor

Qplus-Hyper is a hypervisor that can reliably host an RTOS and a GPOS to run on a single embedded system side by side simultaneously. It is a TYPE I hypervisor in ARMv7, which uses virtualization extension, that attempted to minimize virtualization overhead. Qplus-Hyper is written with fewer than 10K lines of code for formal verification and certification, both of which verify the reliability of the embedded hypervisor. To achieve this, I/O is processed through IOVM, which is a micro-kernel-type virtual machine. Health monitoring and snapshot restoration are supported to quickly respond to faults in guest operating systems. As shown in Fig. 1, this hypervisor was implemented by following the Automotive-SPICE [14] development process.

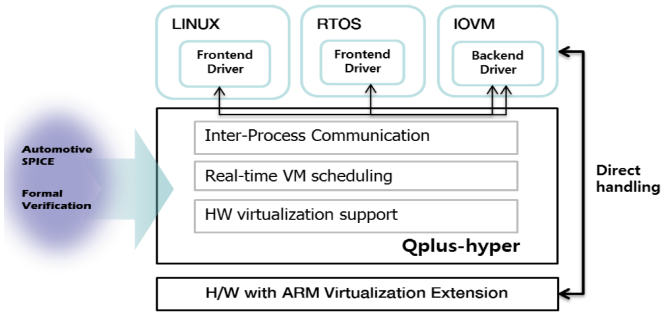

Fig 1. Micro-kernel architecture and hybrid virtualization with safe and reliable development process

III. Virtualized Remote Graphic Processing Framework

Overhead needs to be kept to minimum when virtualizing GPU for embedded environment because an embedded hypervisor usually hosts an RTOS guest. I/O pass-through is known to have the lowest overhead among the GPU virtualization methods. However, we resorted to API remoting instead because no GPU for embedded system supports I/O pass-through to the best of our knowledge and API remoting also has very low overhead in the software techniques. In API remoting, API calls of graphics software stack are intercepted for processing so that it can have only small overhead because the intercepted API calls come from graphics stack API, which lies in a level above para-virtualization and device drivers. On the other hand, this approach would be inflexible and cumbersome for server or desktop settings because the API remoting module needs to be modified for each graphics library such as OpenGL or DirectX. Fortunately for embedded systems, most multimedia contents follow one standard of OpenGL ES. It is drafted and maintained by the Khronos Group, a non-profit organization dedicated to creating open standard APIs to enable the authoring and playback of rich media on a wide variety of platforms and devices. It has industry-leaders as members such as Google, Pixar, IBM, Sony, and Nvidia [15]. Thus, we applied API remoting to OpenGL ES standard library for GPU virtualization to be shared by multiple guest operating systems. In fact, performing remote actions needs copying or transferring API blocks between the front-end and the back-end; this technique is similar to the structure of RPC. In a virtualization layers, a hypervisor should intrude into the front-end and the back-end to make a copy of shared data. Therefore, we incorporated a transparent shared block policy between them in an attempt to reduce communication overhead. This approach enables use of OpenGL ES API even in real-time operating systems that do not support OpenGL ES library or have a GPU driver, hence a real-time OS or a bare-metal system could utilize GPU just by including our library. The following discusses our virtualized library and dual display in detail.

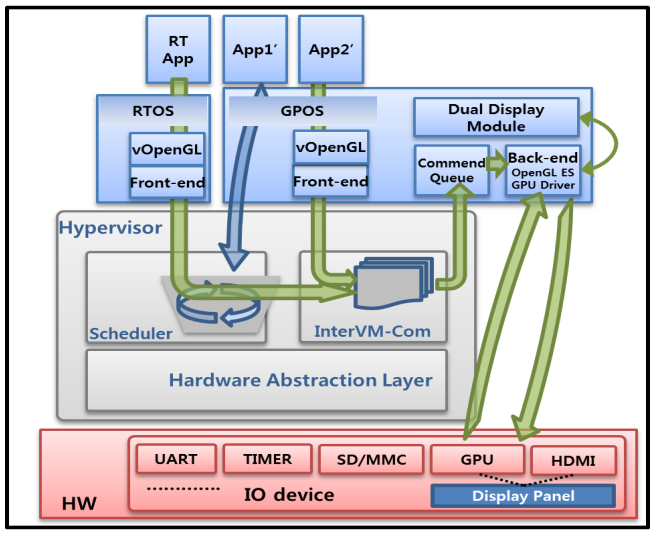

Fig 2. Remote graphic processing framework based on inter-VM communication

A. Virtualized Remote OpenGL ES

Fig. 2 is the overall architecture of our remote graphic processing framework based on inter-VM communication. The hypervisor assumes that an RTOS and a GPOS are installed in the system as guests. The virtualized OpenGL ES consists of front-end library on the guest and back-end library on the I/O

device virtual machine (IOVM), which is a virtual machine similar to Domain 0 of XEN. In the current implementation instead, the back-end part with the dual display module is simply located on the GPOS. The primary purpose of the front/back-ends in the virtualized OpenGL ES library is to transmit and receive API calls, and overhead arises during communication activities between them. We use the transparent shared block policy and polling mechanism to make this communication overhead as small as possible. When a multimedia application on a guest OS calls OpenGL ES APIs, the front-end and back-end libraries create a grant table in the shared memory block within the hypervisor to enable communication. At first, the shared memory block is accessible only through the triggering guest. API calls to the wrapping OpenGL ES library are stored in the shared block until an application makes an *eglSwapbuffer* API call or until a command is sent that requires a return value. When an *eglSwapBuffer* API call is made at the front-end or when an API call requiring a return value is requested, the block for storing the API calls adjusts the access permission of the block so that it could be accessed by the back-end in IOVM. The command queue manager directly points to the memory address of that block. Then the back-end module decodes the wrapping API and makes a request to the GPU for command through the OpenGL ES library.

The first version of our implementation relied on hypervisor-based interrupts to exchange messages between the front-end and the back-end. However, using interrupts involves the hypervisor to perform certain traps and this becomes overhead. If the GPOS consists of multiple vCPUs, then scheduling overhead would arise as well. For these reasons, we added a permission block so that memory blocks could be shared between the front-end and the back-end without involving the hypervisor. The command queue module checks if the requested shared block is open to access and points to the API call address as necessary. This is similar to the structure where two threads poll each other for communication synchronization without a virtualization layer in order to reduce the interferences. In our approach nevertheless, delays arise between the point at which the front-end first requests a GL command and the point at which the back-end processes this request. After the first delay, overhead for communication becomes almost negligible.

While running a GL command, the return value from the back-end sometimes needs to be sent back to the front-end. In such a case, the front-end and the back-end need to be synchronized again, which is negatively affects performance. We attempt an optimistic scheme to reduce this overhead in the following way. When the front-end makes a request for a return value in an API call, a predefined value is returned instead. If the front-end makes an API call using the returned value, the back-end traces back the previous APIs by means of parameters and the actual return value is located. Unfortunately, this scheme is not a complete solution. For example, this method is not applicable to a case in which the coordinates of a specific pixel is obtained from GL texture or surface because GL is not processed at the front-end. In a later version thus, we plan to add a history manager to as an improved solution. After a wrapping API is processed at the back-end, the result will be stored in the history log with special rules. Then we will devise a table shared between the back-end and the front-end to backtrace history and perform rollback as necessary.

B. Dual Display

One purpose for virtualized embedded system is making an improvement on SWaP (size, weight, and power). Hence, we opted for displaying both guest operating systems on single screen by means of frame buffer object (FBO) rather than assigning a physically separated I/O display to each guest OS.

Fig. 3 shows an overview of this implementation. When the dual display module runs, command queues and a texture ID table are constructed for a simple separate UI. IOVM holds one command queue for each guest operating system. The queues are used for scheduling so that commands can be sequentially executed at the GPU. Each time IOVM receives an *eglSwapBuffer* API call from the front-end, FBO is updated by the dual display module. In this way, GPU requests by each guest operating system can stay unmixed and no influence can be made to each allocated region on screen even when one of the guest operating systems encounters faults or terminates unexpectedly.

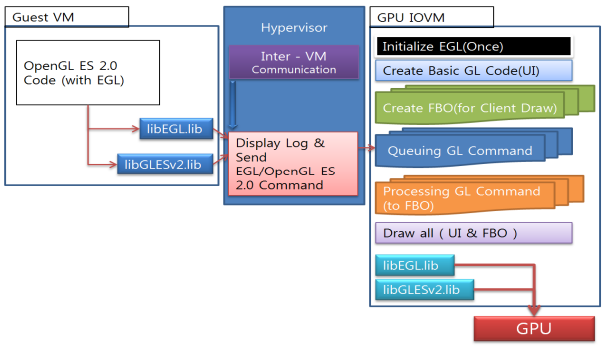

Fig 3. Dual display processing flow using FBO (Frame Buffer Object)

IV. DEMONSTRATION

We demonstrate GPU sharing between virtual machines on an embedded hypervisor. Our implementation functions as an in-vehicle infotainment system that requires an RTOS control and GPOS performance. On top of Qplus-AIR [16], which is an RTOS certifiable package for DO-178B Level A Avionics [17], we implement an RTOS certifiable package for DO-178B Level-A Avionics [17], an automotive digital instrument cluster connected with the accelerator and the brake. For high-performance multimedia application to run on our GPOS, a linux-compatible navigation application with 3D view is used. The guest operating systems are laid on top of Qplus-Hyper as discussed in section II. For hardware, Arndale board [18] is used. It has Exynos5250, which has Cortex-A15 CPU and Mali-T604 GPU built in it. Fig. 4 shows a screenshot of the implemented system. The screen resolution is 800x600 pixels with the left screen assigned with the digital instrument cluster of the Qplus-AIR guest operating system and the right screen used by the 3D navigation application running on linux. Although Qplus-AIR does not support the OpenGL ES library yet, the virtualized remote OpenGL ES library can be installed

and a multimedia application using the library can be compiled and executed on it. Our implementation shows that an RTOS and a GPOS could share a GPU successfully and the subjective experience of their performance is almost the same as running them in the native environments. The performance of the digital instrument cluster on the RTOS remains unaffected even when the 3D navigation application on the GPOS is abruptly terminated.

In our future work, our research will focus on GPU sharing scheduling between multiple guests. The performance can be measured with frame per seconds (FPS) in the navigation application and the response time of the digital instrument cluster by an accelerator and a brake.

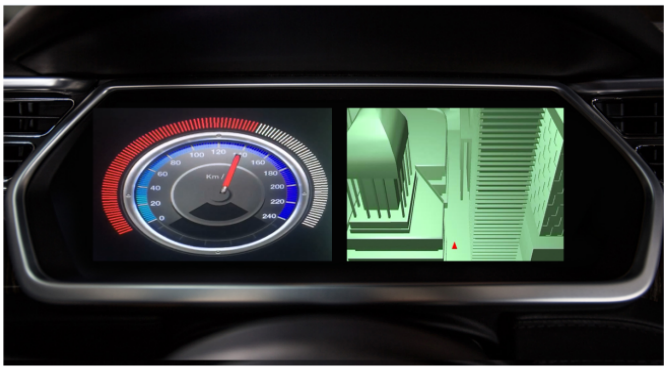

Fig 4. The Demonstration of dual display (left: digital cluster on RTOS, right: 3D navigation on GPOS)

V. Discussions and Future Work

In this work, we explored how to virtualize GPU on a hypervisor for embedded system. Prior to the work, we had anticipated that the performance of virtualized GPU on existing embedded platform would be inadequate for practical use. In fact, GPU devices for embedded systems do not have hardware assisted virtualization features yet. Moreover, software techniques for GPU virtualization in embedded systems had not been fully developed to smoothly run 3D multimedia applications. Contrary to our expectation, we found it is feasible for an embedded hypervisor to provide practical GPU virtualization service to guest operating systems. Thus, our framework can be applied to practical embedded systems when a GPU without hardware assisted virtualization feature. If our proposed framework is supplemented with an optimized GPU scheduler and a fault tolerance module for RTOS, our framework would be more useful even in the safety-critical systems.

The current work is the first prototype developed for proof-of-concept. In the future, we will extend it by improving on support for RTOS and GPOS. For real-time performance, we will work on response time between the accelerator / brake and the digital instrument cluster. In the context of GPOS, we will seek to improve frame per seconds (FPS) of 3D contents. At the moment, our research is focused on a dual OS system scheduling policy in the context of improving GPU virtualization. The synchronization method for inter-VM communication is also being explored. Later in our research, we will further develop our work so that the virtualized GPU module is accessible by the embedded hypervisor and commercial multimedia applications could run smoothly in a virtualized environment.

Acknowledgment

RTst(c). worked on the initial grant table module. REAKOSYS INC. developed multimedia applications and the networked library on linux. Lee provided helpful advices on earlier draft of this research. This work was partly supported by the ICT R&D program of MSIP/IITP[R0101-15-0081, Research and Development of Dual Operating System Architecture with High-Reliable RTOS and High-Performance OS] and [B0101-15-0663, Safety-critical Distributed Modular SW Platform].

References

[1] L. Scholz, M. Beckmann, P. Bartsch. “In-vehicle access of mobile device functions”, U.S. Patent Application 13/834,593, 2013.

[2] Car Connectivity Consortium. MirrorLink. [Online]. Available: http://www.mirrorlink.com (accessed on May 2015)

[3] Intel Corporation (white paper) “Applying multi-core and virtualization to industrial and safety-related application” (2009), http://download.intel.com/platforms/applied/indpc/321410.pdf, (accessed on May 2015)

[4] G. Heiser, “Virtualizing embedded systems: why bother?” in Proceedings of the 48th Design Automation Conference, 2011, pp. 901-905

[5] G. Heiser, "The role of virtualization in embedded systems." Proceedings of the 1st workshop on Isolation and integration in embedded systems. ACM, 2008, pp.11-16

[6] C. Dall and J. Nieh, “Kvm/arm: The design and implementation of the linux arm hypervisor”, Proceedings of the 19th international conference on Architectural support for programming languages and operating systems. ACM, 2014, pp.333-348

[7] J. Y. Hwang, S. B. Suh, S. K. Heo, C. J. Park, J. M. Ryu, S. Y. Park, C. R. Kim, “Xen on ARM: System virtualization using Xen hypervisor for ARM-based secure mobile phones” In Consumer Communications and Networking Conference, 2008, pp.257-261

[8] I. Kolchin, S. Filippov “Bare–metal microhypervisor prototype and measurement of real–time characteristics” International Journal of Embedded Systems Vol.7 No.1 2015, pp.1-10.

[9] M. Dowty, J. Sugerman, “GPU virtualization on VMware's hosted I/O architecture”, ACM SIGOPS Operating Systems Review Vol.43 No.3 , 2009, pp 73-82

[10] Y. Suzuki, S. Kato, H. Yamada, K. Kono, “GPUvm: why not virtualizing GPUs at the hypervisor?”, In Proceedings of USENIX Annual Technical Conference. USENIX, 2014, pp.109-120

[11] S. Jang, W. Choi and W. Kim, "Client Rendering Method for Desktop Virtualization Services” ETRI Journal Vol.35.No.2, 2013, pp 348-351.

[12] T. Kim, D. Kang, S. Kim, J. Shin, D. Lim and V. Dupre, “ Qplus-hyper : A hypervisor for safty-criticla systems”, The 9th International Symposium on Embedded Technology (ISET) 2014

[13] Exynos5250 spec, Samsung Electronics http://www.samsung.com (accessed on May 2015)

[14] Automotive, S. I. G. "Automotive SPICE Process Assessment Model." The Procurement Forum. 2005

[15] Khronos Group, http://khronos.org (accessed on May 2015)

[16] T. Kim, D. Son, C. Shin, S. Park, D. Lim, H. Lee, B. Gim, and C. Lim, “Qplus-AIR: A DO-178B Certifiable ARINC 653 RTOS” The 8th International Symposium on Embedded Technology (ISET) 2013

[17] Johnson, Leslie A. "DO-178B, Software considerations in airborne systems and equipment certification." Crosstalk, October 1998.

[18] Arndale Board, http://www.arndaleboard.org/ (accessed on May 2015)