VADI: GPU Virtualization for an Automotive Platform

Chiyoung Lee, Se-Won Kim, and Chuck Yoo, *Member, IEEE*

***Abstract*—Modern vehicles are evolving with more electronic components than ever before (In this paper, "vehicle" means "automotive vehicle." It is also equal to "car.") One notable example is graphical processing unit (GPU), which is a key component to implement a digital cluster. To implement the digital cluster that displays all the meters (such as speed and fuel gauge) together with infotainment services (such as navigator and browser), the GPU needs to be virtualized; however, GPU virtualization for the digital cluster has not been addressed yet. This paper presents a Virtualized Automotive DIsplay (VADI) system to virtualize a GPU and its attached display device. VADI manages two execution domains: one for the automotive control software and the other for the in-vehicle infotainment (IVI) software. Through GPU virtualization, VADI provides GPU rendering to both execution domains, and it simultaneously displays their images on a digital cluster. In addition, VADI isolates GPU from the IVI software in order to protect it from potential failures of the IVI software. We implement VADI with Vivante GC2000 GPU and perform experiments to ensure requirements of International Standard Organization (ISO) safety standards. The results show that VADI guarantees 30 frames per second (fps), which is the minimum frame rate for digital cluster mandated by ISO safety standards even with the failure of the IVI software. It also achieves 60 fps in a synthetic workload.**

***Index Terms*—Automotive virtualization, device isolation, driver information systems, graphical processing unit (GPU) virtualization, road vehicle software.**

I. Introduction

IN MODERN vehicles, an analog cluster is being replaced with a digital cluster that consists of graphical processing unit (GPU) and display device. It is expected that the digital cluster displays all the meters (such as speed and fuel gauge) together with infotainment services (such as navigator and browser). These meters gather data from the control software running on a vehicle, and infotainment services are managed by the in-vehicle infotainment (IVI) software that allows interaction with the driver of the vehicle. This means that the digital cluster needs to be managed by both the control software and IVI software. Therefore, the GPU and display device for the digital cluster must be "properly" shared.

Because IVI software allows external inputs, the digital cluster needs to be protected from potential failures and attacks by IVI software. For example, the digital cluster may display wrong values due to some mysterious bugs in the IVI software or malware downloaded externally may conceal the current status of a vehicle such as refuel sign and speed by overlapping other images on them, which can compromise the safety of the vehicles. Therefore, the GPU and display device for the vehicle must be isolated from failures and attacks of the unreliable IVI software for safety.

Virtualization is a technology that consolidates automotive control software and IVI software [1]. There are a few research studies that actually implement virtualization for vehicles. Nautilus [2] is a virtualization platform based on the Xen hypervisor [3] to support reliable, secure, and well-optimized IVI software. This platform focuses on improving the reliability of Android for the IVI software via sandboxing. Mentor embedded hypervisor [4] is a type 1 hypervisor with a small footprint, and is based on TrustZone technology [5], [6]. This hypervisor is designed to consolidate and isolate critical and noncritical functions, and reuse existing software. SafeG [7] is a dual-operating system (OS) monitor based on TrustZone technology. This monitor supports reliable and stable operation of a real-time operating system (RTOS) in a consolidated environment. However, these virtualization solutions do not address how to virtualize GPU for a digital cluster.

Previous research for GPU virtualization have been proposed to support graphic processing using GPU on the server virtualization environments. There are two approaches: one is to allocate the GPU device to a virtual machine [8], and the other is to implement a virtual GPU device using device emulation [9]. The first approach (i.e., allocating the GPU device) only allows one virtual machine to use GPU device. On the other hand, the virtual GPU device mechanism is complex because all features of the GPU device must be implemented via software. Moreover, the implementation of the virtual GPU device requires significant development efforts and costs, because the GPU vendors do not open the details of a GPU architecture or a device driver code. To overcome these disadvantages, VMGL [10] and Blink [11] propose the API redirection that intercepts the graphic library functions. However, these researchers focus on desktops and servers and do not consider features of digital cluster that require both the minimum frame rate and protection

Manuscript received June 10, 2015; revised September 05, 2015; accepted December 01, 2015. Date of publication December 17, 2015; date of current version February 02, 2016. This work was supported in part by the Institute for Information and Communications Technology Promotion (IITP) Grant funded by the Korea Government (MSIP): (Grant B0126-15-1046, Research of Network Virtualization Platform and Service for SDN 2.0 Realization) and [Grant R0126-15-1066, (SW Starlab) Next generation cloud infrasoftware toward the guarantee of performance and security SLA], and in part by the National Research Foundation of Korea (NRF) Grant funded by the Korean Government (MEST) (Grant 2010-0029180) with KREONET. Paper no. TII-15-0912.

The authors are with the Department of Computer Science and Engineering, Korea University, Seoul 136-701, South Korea (e-mail: cylee@os.korea.ac.kr; swkim@os.korea.ac.kr; chuckyoo@os.korea.ac.kr).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TII.2015.2509441

of the control software and GPU device. Therefore, the existing GPU virtualization mechanism cannot be applied to vehicles.

GPU virtualization for a vehicle must resolve three main issues. The first issue is consideration of the International Standard Organization (ISO) safety standards [12]–[15]. This issue is not only the motivation of this paper, but also a contribution to be provided by our mechanism. The ISO safety standards define and encourage safety rules for all automotive softwares to prevent accidents by software failures. Most of the countries regulate the development of automotive software through the ISO safety standards because these standards provide the minimum requirements to control vehicles and protect humans from unexpected accidents. However, the existing GPU virtualization mechanisms do not refer to the ISO safety standards. Thus, a new GPU virtualization mechanism for vehicles that consider the safety standards is required.

The second issue is the isolation of the GPU device. The GPU in a vehicle is exposed to potential failures caused by software faults or attacks because the unreliable IVI software can access the GPU device. If the software causes the failure of the GPU device and disturbs the GPU use of the control software, this can imperil the safety of the vehicle. Therefore, the GPU virtualization must isolate the GPU device from the unreliable IVI software.

The third issue is to provide reliable communication between the execution domains sharing a GPU device. This communication is responsible to deliver graphical commands of software to the GPU device. If malware transmits corrupt data to the GPU device, the possibility of unexpected actions increases, which can cause accidents. Therefore, the communication mechanism must prevent unreliable data exchanges by malware.

In addition, the manufacturer dependency and compatibility are required in the GPU virtualization for vehicles. The GPU virtualization that depends on the specific GPU device is not suitable for vehicles because modern vehicles consist of many hardware devices and system boards produced by various manufacturers. Therefore, the GPU virtualization must be independent from the GPU manufacturers in order to run with various GPUs. This compatibility allows us to run the legacy graphic software on the virtualized system without any modifications. The graphic software works closely with other legacy software. If the legacy software is modified because of GPU virtualization, difficulties may arise. Thus, the GPU virtualization must support the legacy graphic software without modifications. This paper resolves the dependency and compatibility through library-level virtualization of the GPU.

This paper proposes the Virtualized Automotive DIsplay (VADI) system that consolidates and isolates GPU for a digital cluster. VADI is a GPU virtualization technique based on a lightweight virtualized automotive platform called secure automotive software platform (SASP) [16] that virtualizes resources such as CPU, memory, and interrupts to the control software and the IVI software. On top of SASP, VADI isolates a GPU device from the IVI software by implementing indirect access mechanism for the GPU device (the second issue above). This does not allow the IVI software to change values of the GPU registers and memory.

Also, VADI provides a pseudodevice for reliable communication (the third issue) between the control software and IVI software based on a shared memory, called V-Bridge. This is achieved by hiding the shared memory from a graphic software and wrapping access functions for the share memory.

For GPU virtualization independent from the manufacturers and application compatibility, VADI is implemented at the user library level. This removes the manufacturer dependency of the GPU and resolves the closed GPU driver code issue, which means that VADI can easily migrate to a different GPU. Moreover, VADI does not require the modification of the legacy graphic software; it can execute the existing software without any changes to the code. This is because all the functions of VADI have the same names and parameters as the original graphic library functions. These functions are based on embedded graphic libraries such as OpenGL ES [17] and EGL [18].

These features such as the GPU isolation, reliable communication, independency, and compatibility make VADI satisfy the ISO safety standards (the first issue). VADI provides a short interrupt processing time and indirect device access, which fulfills the time requirement and integrity requirement of the ISO safety standards. In addition, VADI supports a degraded operation mode, which guarantees the GPU use of the control software even with unrecoverable errors of the IVI software.

This paper is structured as follows. Section II describes automotive virtualization platforms, SASP that VADI is based on, and safety requirements driven from the ISO safety standards. Section III presents the design of VADI and Section IV describes our implementation details. In Section V, we show the performance results of VADI. Section VI discusses previous research for GPU virtualization. Finally, this paper is concluded with a summary and future work in Section VII.

II. Background

A. Automotive Virtualization Platform

The trend of modern vehicles is the integration of various technologies to provide improved convenience to a driver and passengers of vehicles [19]–[21]. The advent of new IVI software leads to the increased number of electronics control units (ECUs) and the higher possibility of software errors. Thus, many vendors of vehicles try to reduce the number of ECUs.

Virtualization can aid in the reduction of ECUs because it allows multiple OSs to be consolidated simultaneously in a single physical machine. In the virtualization technology, software at the bottom of the system software stack, referred to as hypervisor or virtual machine monitor (VMM), abstracts all computing resources such as CPU and memory. Then, the hypervisor creates multiple logical resources from the abstracted resources. For example, CPU is allocated to each OS by a scheduler and memory is individually partitioned for OSs and their applications. By using these logical resources, virtualization technology can provide multiple logical machines, which are matched to ECUs, on a physical machine. Therefore,

various virtualization platforms for a vehicle have been proposed to consolidate several ECUs and isolate the control software from the IVI software.

The automotive virtualization platforms are classified according to their virtualization technologies. There are two types: hypervisor virtualization and physical virtualization. The hypervisor virtualization uses software such as a hypervisor or VMM. The hypervisor has the highest privilege such as ring zero and controls OS on execution domains (also called virtual machines). For the isolation among virtual machines, the hypervisor enforces mutual access for memory and peripheral devices and determines when to allocate CPU cores to each virtual machine. The well-known hypervisors are Xen, VMware [22], and KVM [23] for desktop and server computers. However, the hypervisor virtualization causes performance degradation because the hypervisor has to monitor virtual machine execution with interception of privileged instructions and interrupt distribution.

To resolve the performance disadvantage of the hypervisor virtualization, the latest research [24]–[26] utilizes the recent hardware-assisted virtualization technologies such as ARM virtualization extension [27]. However, the hardware-assisted hypervisor is only supported in the latest processor such as CortexA15 because it requires a special hardware for virtualization such as two-stage memory management unit (MMU).

The physical virtualization is a technology that uses physically partitioned hardware. This resolves the performance disadvantage of the hypervisor virtualization. Mentor embedded hypervisor and SafeG allow direct access of peripheral devices and schedule ownership of the device. However, this device access mechanism cannot address device failures. For example, if a virtual machine with the ownership for a device is halted, another virtual machine cannot access the device. In addition, the device access mechanisms of mentor embedded hypervisor and SafeG are vulnerable to the device driver failure by corrupt values because they allow both virtual machines to directly write the values on registers and memory for the device.

To resolve the device failure problem and provide improved device isolation, SASP [16] has been proposed. SASP is a virtualization platform to support secure device access that is explained briefly in Section II-B.

B. Secure Automotive Software Platform

SASP[1] is a lightweight virtualization platform based on TrustZone that consolidates the control software and the IVI software and guarantees secure device access for the consolidated automotive software.

TrustZone [5], [6] is the security enhancement System on Chip (SoC) technology proposed in ARM cooperation. It was originally proposed to protect the software of a trusted domain (secure world) from another domain (normal world). For protection, each domain of TrustZone has individual CPU registers and memory that are physically separated from those of the opposite domain, and provides asymmetric privileged access rights. For example, the secure world can access the memory and peripheral devices for the normal world, whereas the inverse case is not allowed. In addition, TrustZone provides two types of interrupts with priority, FIQ and IRQ, to prevent interrupts of the secure world from being delayed by interrupts of the normal world.

SASP uses the features of TrustZone to simultaneously run an RTOS such as AUTOSAR [28] with control software and a general-purpose OS (GPOS) such as Linux with the IVI software. SASP has two components: Para-TrustZone OS and V-Monitor. Para-TrustZone OS means OS running in each world, including RTOS and GPOS. Each OS is modified to communicate with V-Monitor for interrupt distribution, the explicit monitor call toward V-Monitor, and the management of shared memory. V-Monitor is responsible for the admission control of the Para-TrustZone OS and the allocation of hardware resources. It comprises the Guest OS Manager, Interrupt Router, Buffer Manager, and Inter-Processor Interrupt (IPI) Stub. The Guest OS Manager boots up RTOS and GPOS and manages hardware resources. For instance, it dedicates a CPU core to the software of each world or schedules both worlds on one core. The Interrupt Router distributes device interrupts to each world using FIQ and IRQ. The Buffer Manager creates and manages shared memory between RTOS and GPOS. The IPI Stub is used for communication between RTOS and GPOS by IPI. SASP was evaluated to show that the performance degradation is within 1% when performing arithmetic operations and within 5% in system call operations. These results imply that SASP can consolidate the control software and IVI software with a low virtualization overhead.

However, the device access mechanism of SASP is not suitable in graphic processing environment that must transmit parameters with various sizes because it only supports small data of the maximum 40 kB. The size of GPU data is various from, e.g., glTexImage2D() for texturing an object requires several megabytes of memory space for a texture image with large number of pixels, whereas glViewport() for setting location of a frame only uses several bytes to store coordinate values (x and y) and size values (height and width). Therefore, the device access mechanism must be extended to support the digital cluster.

This paper presents the design and implementation of GPU virtualization (called VADI) on the top of SASP. Then, it shows the performance results of VADI that processes graphic commands using the real GPU device and displays rendered images on the real display device. Moreover, VADI achieves consolidation to simultaneously display images of control software and infotainment services on a digital cluster, and isolation to protect the GPU and control software from the unreliable IVI software.

C. ISO Safety Standards

The ISO automotive safety standards are the international software standards for safety of vehicles that are established by the ISO. ISO standards related to the digital cluster are ISO 11428 [12], 15005 [13], 15408-2 [14], and 16951 [15].

[1]For the details of SASP, please refer to [16].

ISO 11428 defines ergonomic requirements for the visual danger signals, and ISO 15005 provides ergonomic requirements for safe and effective control while the vehicle is in motion. ISO 15408-2 defines security techniques for IT systems and ISO 16951 provides priority of on-board messages presented to drivers.

The ISO automotive safety standards provide processing and rendering time constraints for real-time properties such as real-time image processing [29] and time-triggered CAN [30]. The processing time constraint in ISO 15005 defines the maximum processing time for input event handling. For example, an input event of the touch screen device must be processed in 250 ms. The rendering time constraint means that the display system requires a minimum number of frames[2] per second (fps). If the system draws the images slowly, it is difficult to deliver the automotive status information to a driver immediately. On the contrary, if the system frequently updates the images on the digital cluster, it obstructs the driver's view. Thus, the GPU virtualization for the digital cluster has to keep the time constraint.

In addition, ISO 15005, 15408-2, and 16951 encourage the processing and drawing images based on their priorities. These requirements prevent to be overlapped the images by other images that are drawn later. The priority is determined by safety, urgency, and criticality of the software. If a map hides speedometer on the digital cluster, it should cause an accident. Thus, the GPU virtualization must prevent image overlapping by priorities.

According to ISO 15408-2, all internal communications of the vehicle have to provide data integrity during the transmission. The data integrity is to prevent changes of data with intention and without intention. The data integrity is a big challenge because the virtualized GPU is shared between the control software and IVI software and both software can change the GPU data. ISO 15408-2 also restricts use of the direct device access to prevent unexpected device errors. Thus, the GPU virtualization must support data integrity through the indirect access.

Finally, ISO 15408-2 defines the system integrity, which requires the vehicle to continuously operate in the degraded mode when unrecoverable errors are detected. The system integrity can be attained by preventing an abrupt halt of the entire automotive system. Thus, GPU virtualization should support the degraded mode of operation when the IVI software halts. This paper assumes that the control software is verified to be trustworthy by automotive industries [31], [32].

III. Design

This section explains the design of VADI as follows. First, VADI shares the GPU device between two execution domains and isolates the GPU device from the IVI software. This architecture virtualizes the GPU and display device to allow both of the control software and IVI software to use the devices for displaying their rendered images. Second, VADI prevents the GPU failure induced by errors of IVI software. This guarantees that control software continuously uses the GPU and display device without any error. Third, VADI preserves requirements of the ISO safety standards. It maintains the minimum frame rate even when simultaneously runs the control software and IVI software. In addition, it preferentially displays images of the control software with higher priority, and prevents unapproved data changes in communication.

[2]"Frames" means the unique consecutive images produced by an imaging device such as GPU.

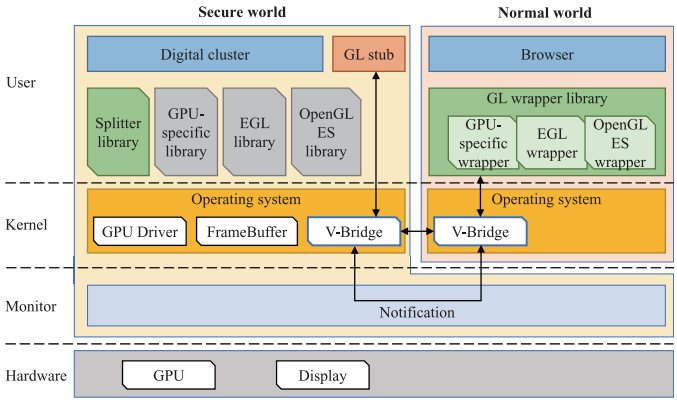


Fig. 1. GPU virtualization architecture on the automotive virtual platform.

A. VADI Architecture

VADI architecture is designed on the SASP to share and isolate the GPU. Along with SASP, VADI consists of two execution domains, called secure world and normal world. The secure world is responsible for running the automotive control software and the normal world runs the IVI software. The secure world uses a dedicated CPU core to reduce performance degradation of control software occurred by CPU scheduling. Moreover, interrupts for each world are partitioned as FIQ and IRQ. The FIQ is used to deliver interrupts to the secure world and the IRQ is used for the normal world.

Fig. 1 shows the VADI architecture. In this figure, the components of VADI are the GL wrapper library, V-Bridge, GL stub, and Splitter library that are modulated to well expand the VADI to a system with another purpose, such as general-purpose graphic processing unit (GPGPU). The other components and monitor in the secure world are parts of the SASP framework. To explain how components of VADI work, this paper assumes that digital cluster runs in the secure world and web browser is in the normal world. The GPU in VADI is attached to the secure world so that the browser indirectly accesses to the GPU using the components of VADI in order to render the browser images.

The GL wrapper library interprets graphic commands of the normal world and packetizes them into individual GPU request messages in order to deliver them to the GPU. To support the interpreting and packetizing, the GL wrapper library implements wrapper functions of three libraries, which are OpenGL ES, EGL, and GPU-specific library. OpenGL ES library is a standard API for three-dimensional (3-D) graphic rendering for embedded systems. EGL library provides common interfaces among rendering APIs of OpenGL ES. GPU-specific library

defines APIs to create a window frame, get a device descriptor for the screen, and get a device status such as resolution.

The V-Bridge of VADI is a pseudodevice driver to transmit the packetized graphic commands of the normal world to the secure world and receive the return values from the secure world. For the transmission of the commands and returns, the V-Bridge uses three buffers that are based on a shared memory to reduce the copy overhead between the secure world and normal world. Details of the buffers will be described in Section IV. The V-Bridge is located in both worlds because it provides bidirectional transmission for the request and response. This paper calls the V-Bridge in the secure world and the normal world as the V-Bridge secure and the V-Bride normal, respectively.

The GL stub is an application that runs in the secure world. It reads the GPU request message from the V-Bridge secure and parses the message to identify the function ID and parameter data. Then, it calls the original OpenGL ES, EGL, and GPU-specific functions of the secure world to process the extracted graphic commands.

The Splitter library of the secure world is responsible to divide the display device for both worlds. For the partition of the display device, the Splitter library separates a buffer for displaying images, which is called frame buffer, into two parts. One part of the frame buffer is allocated to the secure world, and another part is used for the normal world. This prevents the images of the secure world to be overlapped by the images of the normal world.

B. Advantages of VADI

By the components mentioned in Fig. 1, VADI allows the GPU device to be shared between the secure world and normal world. Graphics requests from the secure world go through the graphic libraries, GPU driver, and the GPU, similar to the processing route of the unmodified system. Rendered images by the GPU are stored in the partitioned frame buffer for the secure world. Then, the direct memory access (DMA) controller delivers the rendered images from the frame buffer to the display device.

On the other hand, for the normal world, VADI interprets GPU commands of the normal world and transmits them to VADI components of the secure world. This provides the GPU and display devices for the normal world without real devices. When any software of the normal world calls a graphic function, the wrapper library creates a GPU request message, which includes the identifier of the called graphic function, and conveys the message to the V-Bridge normal. The V-Bridge normal stores the GPU request message into the shared memory and notifies the secure world. When the V-Bridge secure receives the notification, it reads the GPU request message from the shared memory and delivers the message to the GL stub. The GL stub searches the proper stub function using the function identifier and calls the original graphic functions to perform the graphic command. The rendered image of the normal world is stored in the partitioned frame buffer for the normal world. The delivery sequence to the display device is same to the secure world.

Furthermore, VADI supports GPU isolation that separates the GPU device from an unreliable IVI software. As mentioned in Section II, VADI only considers failures of the IVI software because it is difficult to verify them created by various third-party industries. In VADI, the IVI software of the normal world indirectly accesses the GPU and display devices by the transmission of GPU request messages, as required by the ISO 15408-2. This means that the IVI software of the normal world cannot access the registers of the GPU or the frame buffer for the secure world. Thus, the secure world is protected from the errors of the OpenGL requests transmitted from the IVI software because they only manipulate data in the frame buffer allocated to the normal world.

Even, the secure world continues to use the GPU and display devices when the OS of the normal world is halted by a serious error. The reasons are the separated GPU processing routes for each world and the GPU and display devices attached to the secure world. This supports the system integrity mentioned in the ISO 15408-2.

The separated GPU processing routes and the higher priority of the FIQ also make VADI provide the priority-based processing and rendering that is mentioned in the ISO 15005, 15408-2, and 16951. VADI preferentially transfers a GPU requests of the secure world to the GPU device because the GPU processing route for the secure world is shorter than the normal world. In addition, when the device interrupt is generated by the display device, VADI preferentially processes the interrupts for the control software in the secure world due to the high priority of the FIQ. The image overlapping problem is also resolved by the Splitter library of VADI.

Finally, VADI maintains data integrity in the message transmission through the V-Bridge. The V-Bridge is implemented in the kernel and wraps all operations for the shared memory in several system call functions such as vb_write() and vb_read(). Because these system call functions hide the location of the shared memory and prevent direct access for the shared memory, a malicious software cannot read or modify data in the shared memory.

IV. Implementation

A. GL Wrapper Library

Fig. 2 shows the structure of the GL wrapper library. The GL wrapper library redefines all functions of the OpenGL ES library to deliver the GPU request messages of the normal world to the V-Bridge normal. It does not require modification of existing graphic software because the redefined functions of the GL wrapper library have the same function name and parameters as the original OpenGL ES functions. The message consists of function ID, return flag, large param length, parameters, and large parameters. The function ID, return flag, and large param length compose a common header for the request message. The return flag indicates whether the function requires return values or not. The large param length is a variable that holds the length of the parameter data stored in a specific buffer, which is called large param buffer. This length variable is written when the total size of parameters for the requested graphic function is larger

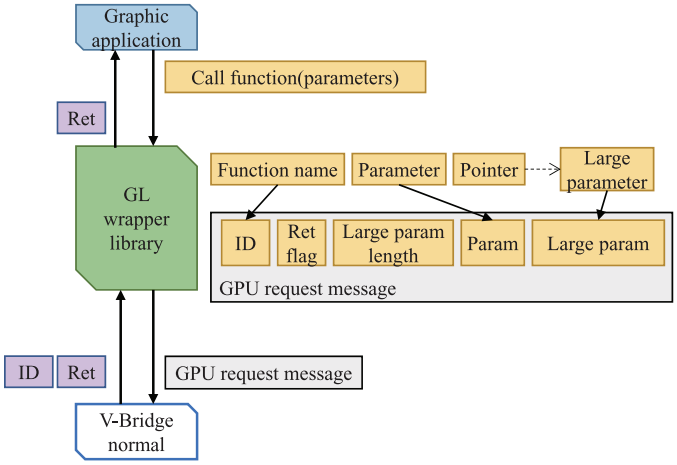


Fig. 2. Implementation of the GL wrapper library.

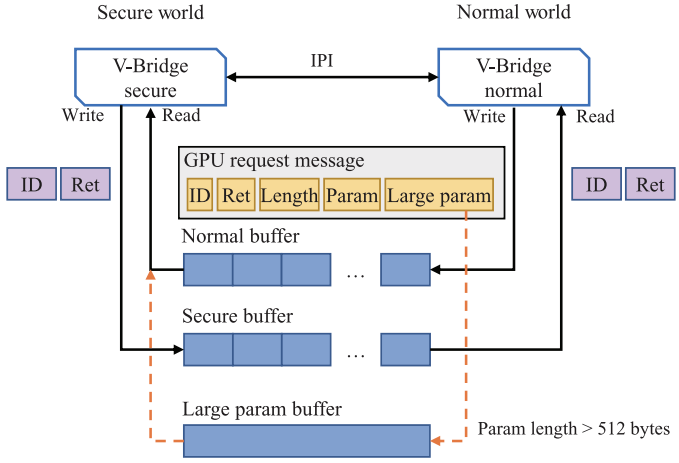


Fig. 3. Implementation of the V-Bridge and Transmission Buffer based on shared memory.

than 512 bytes. If the size is smaller than 512 bytes, the large param length would be zero. The parameter data part consists of the contiguous memory spaces with various sizes and types because each function requires different parameter types, sizes, and counts.

The sequence of the GL wrapper library is as follows. When a wrapper function is called, it stores its own function ID and sets the return flag if a return value is required. Then, it determines which parameter data are a pointer type. If the type is a value, the wrapper function stores the value to the parameter part of the request message. Otherwise, the wrapper function accesses the value using the pointer and tries to copy it into the message. If the size of the parameter is larger than 512 bytes, the wrapper function stores it in the large param buffer and writes the size to the large param length of the request message. After packetizing the request message, the wrapper function transmits the message to the V-Bridge.

B. V-Bridge

Fig. 3 describes the structure of the V-Bridge and data buffers. There are three types of data buffers: the normal buffer, secure buffer, and large param buffer. The normal buffer and secure buffer have independent ring structures of 4 kB that consists of contiguous 512-byte blocks, respectively. The normal buffer is created and managed by the V-Bridge normal. It stores GPU request messages from the normal world. The V-Bridge secure reads the request messages and delivers them to the GL stub. On the other hand, the V-Bridge secure creates and manages the secure buffer that stores the return values given from the GL stub. Because the secure buffer is designed to transmit return values, it is used only when the return flag of the GPU request message is set. On response, the V-Bridge normal reads the function ID and return values from the secure buffer and delivers them to the GL wrapper library.

The large param buffer stores parameters with a large size like a texturing image. This buffer is configured with 1 MB of memory, whereas the secure buffer and normal buffer consist of 512-byte blocks. The V-Bridge divides parameter data into several pieces when the data size is larger than 1 MB in order to store them in the large param buffer. Then, the V-Bridge sequentially transmits these pieces using synchronous read and write operations. When the V-Bridge requires to use the large param buffer for large data, it stores parameter data in the large param buffer and places message headers such as the function ID, return flag, and large param length in the secure buffer or normal buffer. This separation of the header and parameter data reduces the number of access to the large param buffer because the V-Bridge only uses the message header to ascertains whether data in large param buffer are valuable for the current graphic function or not. For example, when glViewport() is called, the V-Bridge secure does not access the large param buffer because the value of the large param length in the message header is zero. Otherwise, functions such as glTexImage2D(), which require parameters that are larger than 512 bytes, set the large param length. The V-Bridge reads large data from the large param buffer after the V-Bridge checks the large param length of the message header.

The separate buffers resolve the synchronization issue to access the shared memory between worlds. The normal buffer and secure buffer only provide the write permission to each world so that they does not use any locking mechanism. This prevents performance degradation caused by the locking mechanism for data synchronization. However, the large param buffer uses a spinning mechanism for the synchronization issue that repeats sleeping and checking the clear bit of the large param buffer. The V-Bridge normal can write new data in the large param buffer after the V-Bridge secure clears the buffer. Despite the spinning in the large param buffer, the performance degradation is imperceptible because the large param buffer is rarely used for transmission of a big texture image.

The V-Bridge uses IPI to notify that the GPU request message is written in the shared memory, which includes the secure buffer and normal buffer. It is because the V-Bridge secure is performed on the separate CPU core. Once an IPI is generated, the V-Bridge changes the CPU mode into the monitor mode and then transmits the IPI. The secure world generates an IPI when sending the return messages to the normal world, and the normal world transmits an IPI to the secure world for the GPU

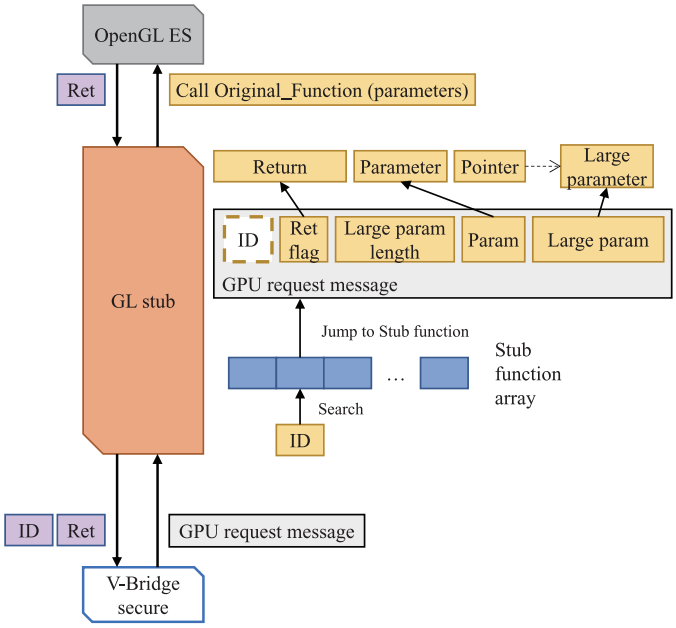


Fig. 4. Implementation of the GL stub.

request messages. The V-Bridge uses asynchronous IPI when the normal world sends a GPU request message; otherwise, it uses synchronous IPI. This reduces performance degradation by the synchronous notification because the normal world does not wait until it receives a return value from the secure world.

C. GL Stub

Fig. 4 presents the structure of the GL stub. The GL stub consists of a function identification part and a stub processing part. The function identification part has a buffer with the start address of each stub function, called Stub Function Array. All entries of this array are ordered by the function ID. This indexing of the Stub Function Array decreases the time to determine the corresponding stub function because it only requires one memory access. The stub processing part consists of various stub functions that deal with the graphic commands.

The processing sequence of the GL stub is as follows. The function identification part of the GL stub periodically reads GPU request messages from the V-bridge. This part extracts the function ID from the message and finds the corresponding stub function from the Stub Function Array. After finding the corresponding stub function, the program counter (PC) jumps to the start address of the stub function. The stub function extracts parameter data from the received request message and stores them in its local memory. If the large param length in the message header is bigger than zero, the stub function accesses the large param buffer and reads the large parameter data. Then, it calls the corresponding original library function and delivers the parameters to the GPU. When the processing of the original function is finished, the stub function checks the return flag of the request message. If return flag is set, it allocates a return message and stores the function ID and the values returned by the original function. Then, the stub function transmits the return message to the V-Bridge.

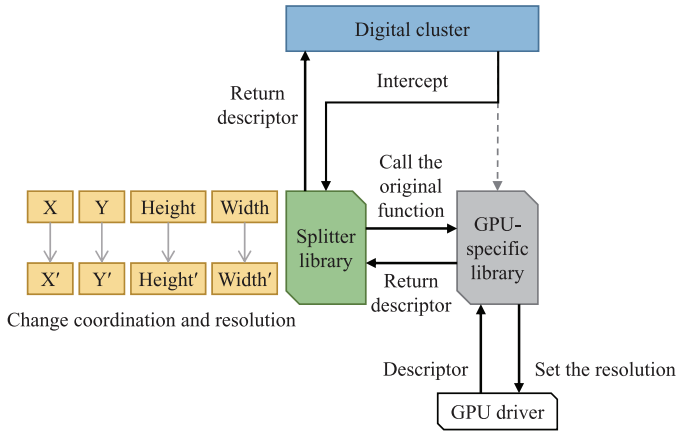


Fig. 5. Implementation of the Splitter library.

D. Splitter Library

As shown in Fig. 5, the Splitter library intercepts the display device descriptor creation function such as fbCreateWindow() and the resolution function such as fbGetDisplayGeometry(). The former is used to get the descriptor of the display device and the latter is called to get the height and width values of the display device. Both functions use the resolution and coordinate values. The Splitter library changes the resolution and coordinate values of the secure world to prevent the display area for the secure world to intrude the area of the normal world.

The Splitter library only intercepts and modifies the function calls of the secure world because the resolution and coordinate values for the normal world are directly changed by the GL stub. This reduces the performance degradation by the function interception of the Splitter library.

V. Evaluation

This section evaluates VADI. For our evaluations, we use native Linux, secure Linux, and normal Linux (version 3.0.35). The native Linux is the unvirtualized Linux kernel that is patched to run on the evaluation board and does not use VADI. The secure Linux is a virtualized Linux kernel to run in the secure world, and the normal Linux is a virtualized Linux kernel for the normal world. These two virtualized Linux remove initialization routines of the devices that are not attached to the world. For example, the GPU initialization routine is removed from the normal Linux because the GPU device is not attached to the normal world.

Fig. 6 shows a system for VADI that consists of an evaluation board and a display device. The evaluation board is Freescale SABRE automotive infotainment board based on i.MX6 quad-core processor with 1 GHz. The processor is based on CortexA9 MPcore processor with ARMv7 architecture that supports TrustZone. The onboard memory is 2 GB, and we allocate 1 GB memory for each world. To minimize interference of other peripheral devices, each world uses a dedicated serial device for input/output commands and multimedia card (MMC) for file system. For the serial devices, we attach a physical serial device to the secure world and another console device, which is implemented using RX and TX ports of Bluetooth device, to the

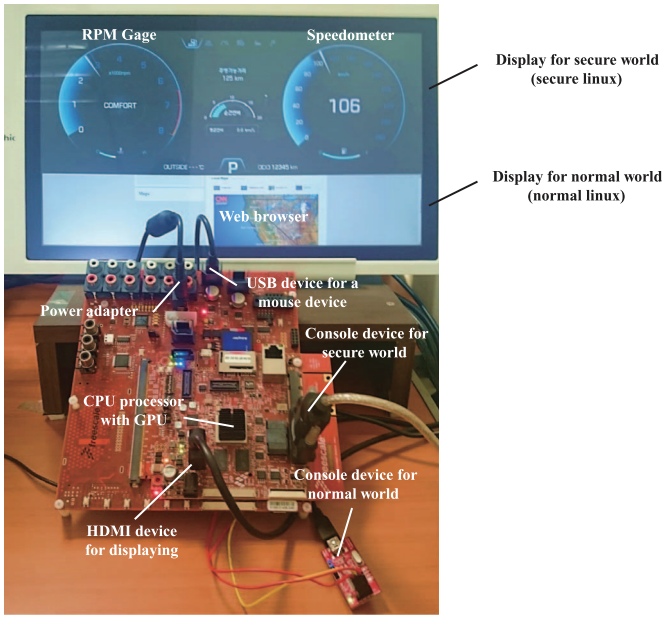


Fig. 6. Evaluation board and display device.

normal world. The secure world uses the MMC device on the CPU-board, and the normal world uses the MMC device on the base-board that consists of additional peripheral devices. These MMC devices are represented as MMC1 and MMC2 in the OS, respectively. The GPU processor is Vivante GC2000 with 500 M–600 MHz for 3-D rendering and is embedded in the CPU.

The display device outputs the images rendered by the GPU. As shown in Fig. 6, the display device is divided into two areas: for the secure world and normal world. The upper area displays a speedometer and revolutions per minute (rpm) gauge that are rendered by the secure world. The lower area presents a web browser of the normal world. The display device is connected with the board by a high definition multimedia interface (HDMI) cable.

As shown in Fig. 6, VADI runs two realistic graphic workloads for the evaluations: the speedometer and the web browser. The speedometer in the secure world continuously renders the speed and rpm. The web browser in the normal world shows a homepage with rich graphics, including weather information and a map. These two workloads represent the automotive control software and IVI software, respectively. In addition, as shown in Fig. 7, three synthetic workloads [33] are used for the frame rate comparison with the native Linux. The first workload draws a triangle and the second workload projects a 3-D triangle and box rotating in their positions. The last workload draws a 3-D box that is applied a bitmap image to each of its surfaces. These synthetic workloads are executed in the native Linux, secure Linux, and normal Linux. All the workloads are based on OpenGL ES 2.0 and EGL 1.4.

To reduce the errors caused by software interference, VADI limits the number of functions that operate in a world and images that are concurrently displayed on the screen. A traditional vehicle also allows each ECU to be responsible for one function to protect the vehicle from software errors [34]. Too many images on a screen compromise the safety of the vehicle

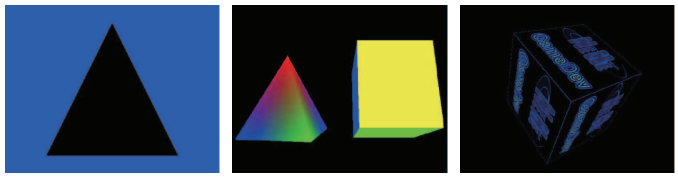


Fig. 7. Synthetic workloads. (a) 2-D triangle. (b) 3-D projection. (c) 3-D box texturing.

because they obstruct the view of the driver [13], [35]. Thus, this research limits the number of graphic applications.

A. Metrics

1) Rendering Time Constraint: This metric is how fast the GPU renders and draws a frame on a display device. This is measured by the fps, as mentioned in the ISO 15005. If it is not assured, a driver cannot get the automotive status information such as a speedometer in timely manner, which is critical for safe driving. We measure the minimum fps of VADI to show that VADI meets the rendering time constraint. According to previous studies, the minimum frame rate for a vehicle is 30 fps [36], [37].

2) Processing Time Constraint: This metric is determined by overheads during GPU processing. VADI uses two routes for the GPU access. One is the original route that uses the unmodified libraries and device drivers, and another is the route that uses the wrapper library and stub. The former route is used by the secure Linux, and the latter route is for the normal Linux. In the second route, the normal Linux sends graphic commands to the GL stub on the secure Linux, and the GL stub processes the command by calling the original graphic functions of the secure world. Thus, the second route is likely to have additional overheads due to the transmission of graphic commands. We measure the processing time of a graphic command and an interrupt to measure the overhead of VADI.

3) System Integrity: This is an important metric encouraged by the ISO 15408-2. It means that the control software must continually run even when the IVI software faces an unrecoverable error. VADI can achieve the system integrity for the GPU that the secure Linux consistently uses the GPU and display devices despite a fatal error of the normal Linux. To verify this constraint, we measure the variation of frame rate on the secure Linux when the normal Linux is in the unrecoverable kernel panic.

B. Rendering Time Constraint

Fig. 8 shows the performance of the speedometer and web browser on VADI. We run the speedometer in the secure world and the web browser in the normal world, respectively. Fig. 8(a) shows a result of the speedometer while the normal world is idle, and Fig. 8(b) is a result of the web browser while secure world is idle. As shown in Fig. 8(a), VADI ensures the minimum frame rate of the secure world, which shows that the speedometer uses 30 fps during operation. This result is the

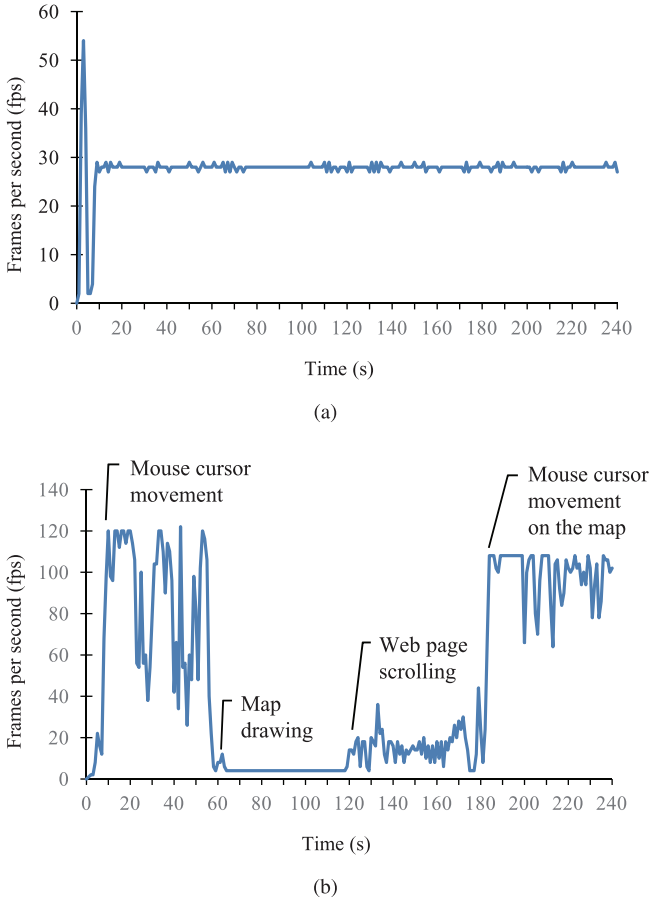


Fig. 8. Performance of realistic workloads in each world. (a) Performance of digital cluster in secure world. (b) Performance of web browser in normal world.

same as the result of native Linux. On the other hand, the web browser of the normal world shows high variation in the frame rate according to the web browser activities. We perform four actions on the browser: mouse cursor movement, map drawing, web page scrolling, and mouse cursor movement on the map. As shown in Fig. 8(b), the frame rate of the browser increases when we move a mouse cursor after 10 s and it is reduced when we stop the mouse movement. The cursor movement requires frequent frame updates because it has to trace all locations of the moving cursor and draw them on the screen. Frame rate of the web page scrolling is smaller than the frame rate for the cursor movement because it replaces a frame with the screen size. The map drawing is performed with the lowest frame rate in our experiment because it has little change that redraws small cloud images in upper left corner of the map.

We compared VADI with the native Linux using the original GPU driver to evaluate the performance degradation caused by VADI. Fig. 9 shows the frame rate of each system. The word "Only" in the legend of Fig. 9 means that there is no workload in the other world. For comparison, we run the workloads that are shown in Fig. 7. As shown in Fig. 9, the native Linux executes the workloads in 59 fps. VADI also achieves 59 fps when the workloads are executed separately in each world. Thus, the performance degradation by VADI is insignificant. Moreover, when the workloads in each world were run concurrently, VADI

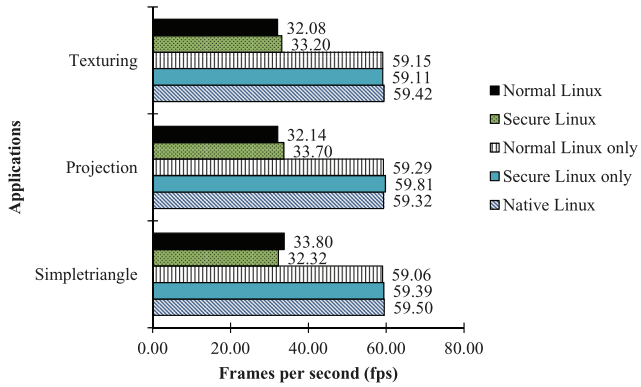


Fig. 9. Frame rate comparison among native Linux, secure Linux, and normal Linux.

maintains 32 fps, which is the minimum frame rate mandated by ISO 15005.

C. Processing Time Constraint

To evaluate the processing time constraint of VADI, we measure the command processing time and interrupt processing time. VADI provides a new graphic processing route for graphic workloads in the normal world. In this route, the graphic workloads of the normal world convert graphic commands into graphic processing requests and transmit them to the GL stub of the secure world using the V-Bridge world communication driver. This architecture protects the GPU device from nontrusted software of the normal world. However, there is overhead in the processing time of commands. To evaluate the overhead by the long processing route, we measure processing time of graphic commands for a frame rendering.

Fig. 10 shows the processing time of graphic commands to render one frame when the system draws a two-dimensional (2-D) triangle. In this result, the processing time of the secure Linux is the same as the Native Linux because two cases use the same graphic processing route. On the other hand, the processing overhead of VADI is measured to be 12 μs, and it is as small as the processing time of a time tick interrupt in the native Linux. This time delay is caused due to the copying function parameters and waking up the kernel thread for the transmission to the GL stub in the user address space.

To find the reason for the overhead, we evaluate the processing time in five steps when VADI handles a graphic command. Step 1) involves calling a wrapper function of the normal world, and is ended before the V-Bridge normal sends an IPI to the V-Bridge secure. This step includes allocating a graphic request message, copying function parameters, calling the write system call function, and copying from the user memory to the kernel memory. Step 2) is from the IPI transmission of the V-Bridge normal until the completion of the IPI handler in the V-Bridge secure. In Step 2), the IPI handler wakes up a notification thread to inform the GL stub of the received message. Step 3) is the consumed time before GL stub reads the received message from the shared memory of the V-Bridge since Step 2). This step includes the kernel thread scheduling by the wakeup() function of Step 2) and the read system call of the GL stub. Step 4) starts

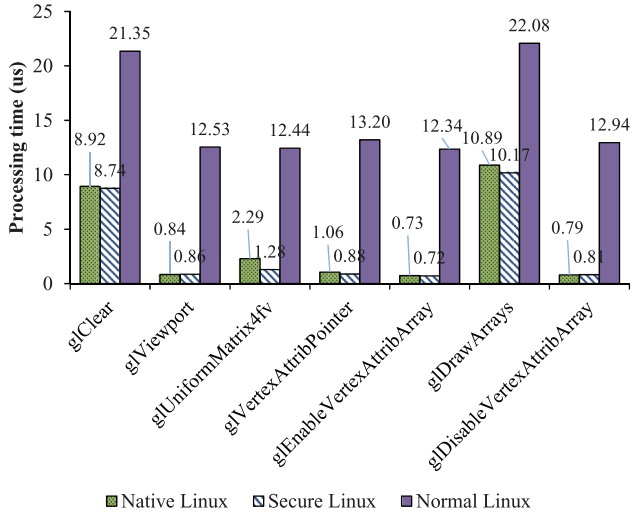


Fig. 10. Average processing time for graphic commands.

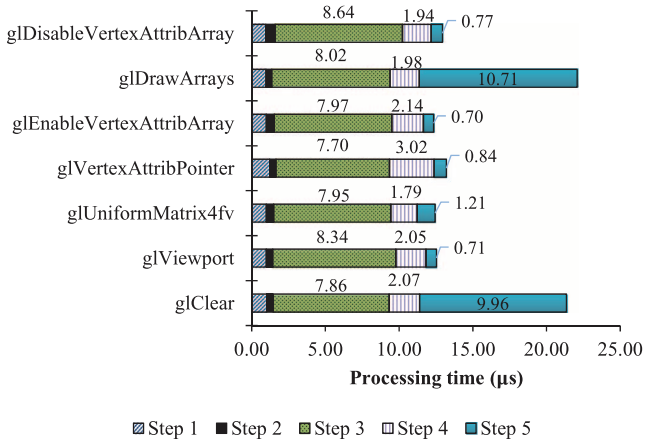


Fig. 11. Average processing time for a graphic command in each step.

before calling the proper original OpenGL ES function. In Step 4), it copies the received message from the kernel memory to the user memory and extracts function parameters from the message. Finally, in Step 5), it calls the original OpenGL ES function with the extracted parameters. This step is the same as the graphic command processing route in the native Linux.

Fig. 11 shows the processing time of each step in the graphic processing route for the normal world when the system runs the 2-D triangle workload using VADI. As shown in Fig. 11, VADI spends 8 and 3 µs in Steps 3) and 4), respectively. The amount of time in Step 3) is due to scheduling of the notification thread. The interrupt handler of the V-Bridge secure wakes up the notification thread when it receives an IPI from the V-Bridge normal. After receiving the wakeup signal, the kernel of the secure world adds the notification thread to the end of the run queue. Then, it invokes the task scheduler to run the thread.

The time delay in Step 4) is due to copying the request message from the shared memory to the user memory of the GL stub and extracting the parameters from the message. The size of the copied message is determined by the parameter size of each graphic function; e.g., the message size of glDisableVertexAttribArray() is 16 bytes and the message size of

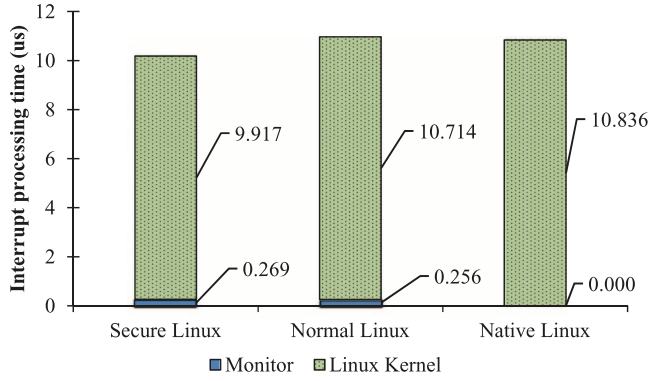


Fig. 12. Average processing time for an interrupt.

glDrawArray() is 24 bytes. The copy operations are performed because of the location of the shared memory; it is in the kernel memory, however, the GL Wrapper library and GL stub belong to the user memory for applications. This copy between user and kernel memory removes the locking for the shared memory among graphic applications.

On the other hand, Steps 1) and 2) hardly affect the processing time of a graphic command because the time delay is as small as 1 and 0.5 µs, respectively. The time in Step 5) is same as the native Linux.

Although VADI adds an additional time to process the graphic commands, it guarantees the minimum frame rate. The time to update a frame is hardly affected by the additional time because the time delay is numerically small. For example, if a digital cluster requires 60 fps, the system has to complete the frame update in 16 ms. VADI achieves this frame rate requirement because VADI handles more than 500 graphic commands within 16 ms.

We also measure interrupt processing time and compared it with the native Linux to evaluate the overhead of VADI. Fig. 12 shows the average processing time of the general-purpose time (GPT) interrupt. In Fig. 12, the secure and normal Linux include the monitor software and Linux kernel because they run on the virtual platform, whereas the native Linux uses the time in the Linux kernel without the monitor. As shown in Fig. 12, the total interrupt processing time of the secure and normal Linux is similar to the time for the native Linux despite of participation of the monitor. The processing time in the monitor is small, with an average value of 0.269 and 0.256 µs, respectively. The reason is the FIQ and IRQ supported by TrustZone. The monitor identifies whether the interrupt is an FIQ or not. If the CPU mode is an FIQ mode when the monitor receives the interrupt, the monitor delivers the interrupt to the secure Linux. Otherwise, it transmits the interrupt to the normal Linux. These two types of interrupt for TrustZone reduce the intervention of the monitor because they provide the identification of an interrupt by the hardware.

D. Simultaneous Use of the GPU and System Integrity

To verify the system integrity during the concurrent running, we run the speedometer in the secure world and the web browser in the normal world simultaneously, and do various

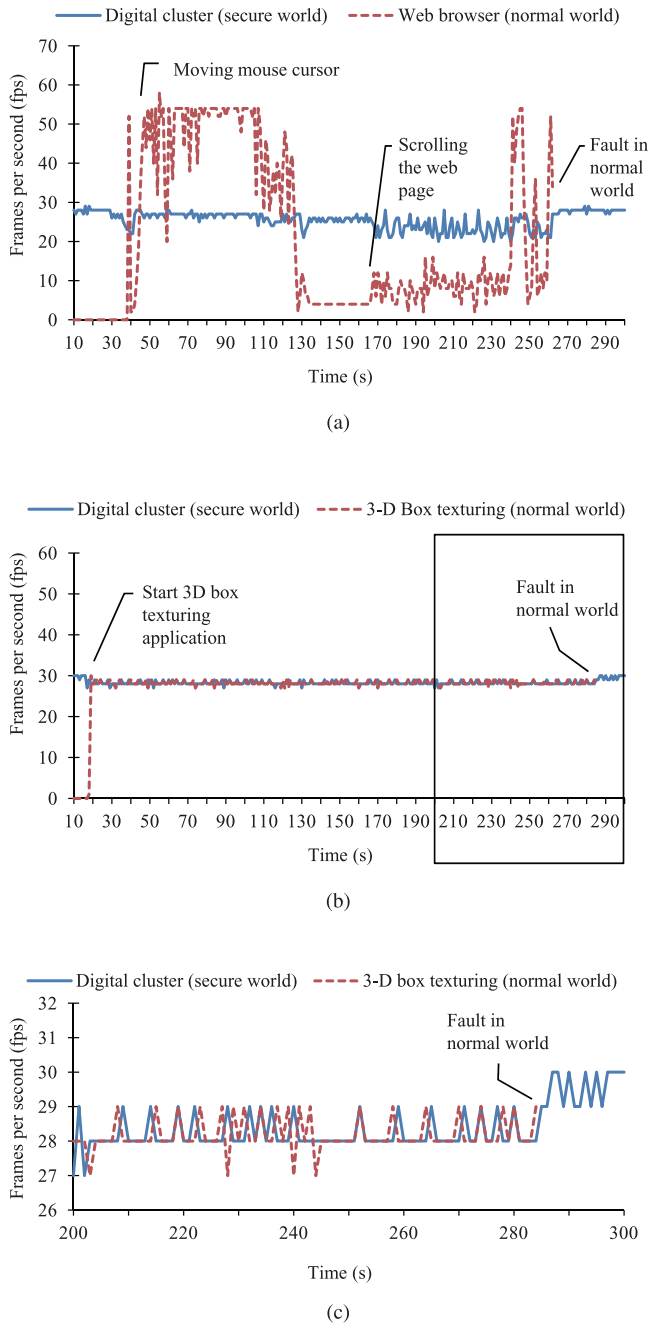


Fig. 13. Performance of concurrent workloads and system integrity. (a) Performance of digital cluster and web browser. (b) Performance of the digital cluster and 3-D box texturing application. (c) Enlarged result of the digital cluster and 3-D box texturing application.

mouse actions on the web browser to increase the normal world load.

Fig. 13 shows the performance of each workload when the workloads were executed simultaneously. Fig. 13(a) is the result between the speedometer and the web browser. In this result, the blue line indicates the frame rate of the speedometer, and the red line (dotted line) indicates that of the web browser, which is started after 37 s. Fig. 13(b) shows the result obtained when the speedometer and the 3-D box texturing application performed concurrently; the blue and red dotted lines represent the speedometer and the 3-D box texturing application, respectively. Because the two applications were indistinguishable in this graph, we enlarge the section ranging from 200 to 300 s in Fig. 13(b) (the black square area), as shown in Fig. 13(c). As shown in Fig. 13, the frame rate of the speedometer is flat and in the range of 25–29 fps, even if the 3-D box texturing application consistently consumes the GPU. This means that VADI maintains the minimum frame rate of the speedometer in the secure world. Moreover, the figures show that the secure world maintains 30 fps even with a fault in the normal world [around 260 s in Fig. 13(a) and 285 s in Fig. 13(b) and (c)]. Thus, the speedometer in the secure world is completely isolated from the software in the normal world. On the other hand, the frame rate of the web browser is lesser than that when it is separately executed, as shown in Fig. 8. This is because the priority of the speedometer in the secure world is higher than that of the web browser.

Moreover, in this evaluation, we confirmed that VADI provides the system integrity by isolating the GPU device from the IVI software in the normal world. In Fig. 13, the browser is halted by a kernel fault of the normal Linux at the 260 and 285 s. However, the speedometer continuously runs despite of the normal world's unrecoverable fault. It runs with 30 fps performance after the 260 and 285 s in each figure. As shown in the result, VADI protects the automotive control software in the secure world from a fault in the normal world and it can run in the degraded mode when software in the normal world faces unrecoverable errors. This means that VADI guarantees the system integrity of the ISO 15408-2.

VI. Related Work

GPU virtualization mechanisms can be classified by their execution level for GPU dedication, GPU emulation, and API redirection. GPU dedication allows a virtual machine to exclusively use the GPU using a hardware support such as Intel VT-d and AMD-V technologies. A typical GPU dedication mechanism is the GPU-passthrough [8] of Xen that provides direct GPU access using Intel VT-d. It achieves a similar performance to unvirtualized GPU and minimizes the GPU driver modification. However, GPU dedication always requires a hardware support, and other virtual machines cannot access the GPU device during the use of one virtual machine. Moreover, the direct access of GPU dedication induces a device fault as the result of the virtual machine errors because it can change values of registers and memory without a hindrance.

GPUvm [38] is another mechanism that uses GPU-passthrough. This mechanism resolves the problem of GPU dedication by using an aggregator that collects all GPU commands from each VM and delivers them to the physical GPU. For the aggregator, GPUvm intercepts the GPU commands of each VM and transmits them to the aggregator. This provides an advantage that can support both para- and full-virtualization because GPUvm does not require modification of GPU driver codes. However, the driver-level interception results in performance degradation; the execution time of GPUvm is almost three times slower than an unvirtualized GPU [38]. In addition, this mechanism does not address displaying and processing of

graphic data because it is proposed for a GPGPU that provides parallel processing of arithmetic data such as matrix addition and multiplication, not graphic processing. XenGT [39], [40] also uses the GPU-passthrough for Intel on-chip GPUs, similar to GPUvm. In this mechanism, a mediator, instead of the aggregator of GPUvm, intercepts the GPU commands and delivers them to the physical GPU. However, it is difficult to apply XenGT to embedded systems for a vehicle with various GPU manufacturers because XenGT depends on the Intel platform and Xen hypervisor [40].

GPU emulation [41], [42], which is proposed for the VMware workstation and fusion, provides a virtual GPU device for a virtual machine and sends low-level GPU commands to a special virtual machine, such as driver VM of Xen and Host OS of VMware. The special virtual machine emulates the received GPU commands through proper combination of its GPU instructions. Because this emulation progress is implemented by software, the mechanism is complex and significantly decreases the GPU performance that is 12%–86% of the unvirtualized GPU [41]. Moreover, the emulation mechanism depends on GPU manufacturers because it requires modification of the GPU driver code to implement the virtual GPU device.

API redirection intercepts graphic commands in the library level and processes them on either a virtual machine or a server with a GPU device. This mechanism is independent from GPU manufactures because it does not require modification of the GPU driver code. Thus, it can utilize various graphic processing engines and migrate the solution to other platforms with a different GPU device. Moreover, the API-redirection mechanism is very simple because a client only sends graphic commands to the stub software in a server with the GPU device and the server run them on the GPU using its library. Performance of the API-redirection mechanism is 86%–100% of the unvirtualized GPU [10].

VMGL [10], [43] is the most popular GPU virtualization mechanism for virtual machines, which uses API redirection. It is implemented on Xen and VMware for x86 machines. VMGL uses a network device to send graphic commands to a driver VM with GPU. This guarantees the virtual platform independence because it uses a standard network interface such as a socket, instead of special kernel functions for the internal communication between virtual machines. However, the use of network requires a network device and increases transmission time of graphic commands between virtual machines by additional network processing and data copies. Automotive systems require device isolation to protect devices from errors caused by nontrusted software; therefore, the system with VMGL must provide an additional isolation mechanism for the network device. Moreover, graphic libraries for embedded systems such as OpenGL ES and EGL are different from libraries for x86 desktop systems like OpenGL [44] and GLUT [45]. Thus, it is difficult to easily apply VMGL on the automotive virtual platform.

VGRIS [9] is GPU virtualization mechanism using API redirection in the type 2 hypervisor such as VMware. This provides GPU command queue for each virtual machine in main memory of a host OS with the hypervisor. When an application calls GPU API functions, the GPU HostOps Dispatcher of the host OS intercepts the calls and converts them to commands for the GPU of the host OS. The converted commands are stored in GPU command queue for each virtual machine. Then, the GPU scheduler of the host OS selects the queue and sends commands in the queue to the GPU. However, VGRIS depends on the type 2 hypervisors because it was implemented in the host OS. Also, the authors of VGRIS present the comparison of GPU performance with native GPU, but does not refer to simultaneous use of GPU by several virtual machines.

SHARC [46] was developed for cloud computing environments, unlike VMGL and Blink. SHARC assumes that various servers such as a virtualization server for resource managements, a rendering server with GPU devices, and a media server for streaming are connected by network. When a client executes graphic software on the virtualization server, SHARC sends graphic commands to the rendering server via the network. The rendering server processes the received graphic commands and converts the rendering results (frames) into JPEG images. These JPEG images are transmitted to the media server via the network. The media server transcodes JPEG images into H.264/MPEG-4 media streams and transmits the media streams to the client using RTP/RTSP or RTMP, which are network protocols for media streaming. Thus, SHARC requires a network device and isolation of the network device for safety of vehicles because all servers connect via the network.

VAGS [35] is an automotive display consolidation architecture using GPU virtualization in a vehicle. Authors of VAGS analyzed the ISO automotive safety standards and classified them for display consolidation. VAGS performs all graphic processing on a consolidated GPU device and draws rendered frames on various display devices such as the head unit of the front seats, the center console, and the digital cluster. However, VAGS only presents the design without implementation details and evaluations because it is a work in progress.

GViM [47] also provides GPU virtualization based on API redirection. The command processing of GViM is similar to VADI. In GViM, the CUDA application of a VM calls CUDA API functions, and then a library called Interposer intercepts this API call. The intercepted call is delivered to the GPU backend on the management VM (called Domain 0) that plays the role of a stub. Then, the GPU backend calls CUDA API functions of the Domain 0 to compute them using the physical GPU. However, unlike VADI, GViM is developed for high-performance computing using a GPGPU; it virtualizes the CUDA [48] library for arithmetic instructions, instead of a graphic library such as OpenGL. Thus, GViM is not suitable for graphic applications because the applications require processing of pixel data that are larger in size as compared to arithmetic data, and the applications require simultaneous display output of the processed data on a screen.

VII. Conclusion

This paper proposes and implements VADI, which is a new architecture to support the digital cluster on the consolidated hardware. VADI concurrently processes graphic commands for two isolated execution domains using one GPU device and

draws their frames in one display device. Moreover, VADI protects the GPU and display device from nontrusted software using TrustZone and indirect GPU access mechanism.

Previous research such as on VMGL and Blink proposed various indirect GPU access mechanisms on the virtual platform. However, they were developed on the desktop computer with x86 architecture, and do not explain how to meet the ISO safety standards for the automotive vehicle.

VADI satisfies the ISO safety standards for the rendering time constraint and system integrity. According to our evaluations, VADI achieves average 60 fps in the experimental workload, which is the same as the native system. It maintains a speed of processing at 30 fps, which is the minimum frame rate for the automotive control software such as speedometer. Moreover, VADI ensures execution of the automotive control software despite the occurrence of an unrecoverable fault in another execution domain because it isolates peripheral devices such as GPU and significant control software from other IVI software such as a web browser.

References

[1] C. Patsakis, K. Dellios, and M. Bouroche, "Towards a distributed secure in-vehicle communication architecture for modern vehicles," *Comput. Security*, vol. 40, pp. 60–74, 2014.

[2] GlobalLogic. (2012, Dec.). *Automotive Grade Android: Solution Accelerators & Services From Globallogic*, Nautilus-Platform.pdf [Online]. Available: https://www.globallogic.com/wp-content/uploads/2012/12/

[3] A. J. Younge, J. P. Walters, S. Crago, and G. C. Fox, "Evaluating GPU passthrough in Xen for high performance cloud computing," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops (IPDPSW)*, 2014, pp. 852–859.

[4] MentorGraphics. (2013). *Mentor Embedded Hypervisor*, Hypervisor-ds.pdf [Online]. Available: http://s3.mentor.com/public_documents/datasheet/embedded-software/

[5] ARM, "Arm security technology: Building a secure system using trustzone technology," ARM Ltd., Cambridge, U.K., Tech. Rep. PRD29-GENC-009492C, Apr. 2009.

[6] ARM, "Cortex-A9 MPCore technical reference manual, r4p0," ARM Ltd., Cambridge, U.K., Tech. Rep. ARM DDI 0407H (ID032812), Mar. 2012.

[7] D. Sangorrín, S. Honda, and H. Takada, "Reliable and efficient dual-OS communications for real-time embedded virtualization," *Inf. Media Technol.*, vol. 8, no. 1, pp. 1–17, 2013.

[8] J. P. Walters *et al.*, "GPU passthrough performance: A comparison of KVM, Xen, VMWare ESXi, and LXC for CUDA and OpenCL applications," in *Proc. IEEE 7th Int. Conf. Cloud Comput. (CLOUD)*, 2014, pp. 636–643.

[9] Z. Qi, J. Yao, C. Zhang, M. Yu, Z. Yang, and H. Guan, "VGRIS: Virtualized GPU resource isolation and scheduling in cloud gaming," *ACM Trans. Archit. Code Optim. (TACO)*, vol. 11, no. 2, p. 17, 2014.

[10] H. A. Lagar-Cavilla, N. Tolia, M. Satyanarayanan, and E. De Lara, "VMM-independent graphics acceleration," in *Proc. 3rd Int. Conf. Virtual Execution Environ. (VEE)*, Jun. 2007, pp. 33–43.

[11] J. G. Hansen, "Blink: Advanced display multiplexing for virtualized applications," in *Proc. ACM Netw. Oper. Syst. Support Digit. Audio Video (NOSSDAV)*, Jun. 2007, pp. 15–20.

[12] Ergonomics—Visual Danger Signals—General Requirements, Design and Testing, ISO Standard 11 428, 1996.

[13] Road Vehicles—Ergonomic Aspects of Transport Information and Control Systems—Dialogue Management Principles and Compliance Procedures, ISO Standard 15 005, 2002.

[14] Information Technology Security Techniques Evaluation Criteria for IT Security Part 2: Security Functional Components, ISO Standard 15 408–2, 2008.

[15] Road Vehicles Ergonomic Aspects of Transport Information and Control Systems (TICS) Procedures for Determining Priority of On-Board Messages Presented to Drivers, ISO Standard 16 951, 2004.

[16] S. W. Kim, C. Lee, M. Jeon, H. Y. Kwon, H. W. Lee, and C. Yoo, "Secure device access for automotive software," in *Proc. Int. Conf. Connected Veh. Expo (ICCVE)*, Dec. 2013, pp. 177–181.

[17] M. Nardone and V. Silva, "Efficient graphics and portability with OpenGL ES," in *Pro Android Games*. New York, NY, USA: Springer, 2015, pp. 95–136.

[18] J. Leech. (2014, Aug.). *Khronos Native Platform Graphics Interface* (EGL version 1.5), eglspec.1.5.pdf, Beaverton, OR, USA [Online]. Available: https://www.khronos.org/registry/egl/specs/

[19] A. Monot, N. Navet, B. Bavoux, and F. Simonot-Lion, "Multisource software on multicore automotive ECUs—Combining runnable sequencing with task scheduling," *IEEE Trans. Ind. Electron.*, vol. 59, no. 10, pp. 3934–3942, Oct. 2012.

[20] M. Sunwoo, K. Jo, D. Kim, J. Kim, and C. Jang, "Development of autonomous car—Part I: Distributed system architecture and development process," *IEEE Trans. Ind. Electron.*, vol. 61, no. 12, pp. 7131–7140, Dec. 2014.

[21] J. C. Ferreira, V. Monteiro, and J. L. Afonso, "Vehicle-to-anything application (V2Anything App) for electric vehicles," *IEEE Trans. Ind. Informat.*, vol. 10, no. 3, pp. 1927–1937, Aug. 2014.

[22] C. Y. Li, "Research on the virtualization construction of university data center server based on VMware vSphere," in *Adv. Mat. Res.*, vol. 1078. Switzerland: Trans Tech, 2015, pp. 375–379.

[23] A. Masood, M. Sharif, M. Yasmin, and M. Raza, "Virtualization tools and techniques: Survey," *Nepal J. Sci. Technol.*, vol. 15, no. 2, pp. 141–150, 2015.

[24] S. Stabellini and I. Campbell, "Xen on arm Cortex A15," presented at the Xen Summit North America, Aug. 2012.

[25] TL Foundation. (2014, Apr.). *Linux Foundation Collaboration Projects: Xenarm With Virtualization Extensions*, White Paper [Online]. Available: http://wiki.xen.org/wiki/Xen_ARM_with_Virtualization_Extensions_whitepaper

[26] C. Dall and J. Nieh, "KVM/ARM: The design and implementation of the Linux ARM hypervisor," in *Proc. 19th Int. Conf. Archit. Support Program. Lang. Oper. Syst*. New York, NY, USA: ACM, Mar. 2014, pp. 333–348.

[27] R. Mijat and A. Nightingale. (2011). *Virtualization is Coming to a Platform Near You*, ARM White Paper [Online]. Available: http://www.arm.com/files/pdf/system-mmu-whitepaper-v8.0.pdf

[28] S. Martínez-Fernández, C. P. Ayala, X. Franch, and E. Y. Nakagawa, "A survey on the benefits and drawbacks of AUTOSAR," in *Proc. 1st Int. Workshop Automot. Softw. Archit.*, 2015, pp. 19–26.

[29] Y.-L. Chen, B.-F. Wu, H.-Y. Huang, and C.-J. Fan, "A real-time vision system for nighttime vehicle detection and traffic surveillance," *IEEE Trans. Ind. Electron.*, vol. 58, no. 5, pp. 2030–2044, May 2011.

[30] M. Hu, J. Luo, Y. Wang, M. Lucasiewycz, and Z. Zeng, "Holistic scheduling of real-time applications in time-triggered in-vehicle networks," *IEEE Trans. Ind. Informat.*, vol. 10, no. 3, pp. 1817–1828, Aug. 2014.

[31] T. Gustafsson, M. Skoglund, A. Kobetski, and D. Sundmark, "Automotive system testing by independent guarded assertions," in *Proc. IEEE 8th Int. Conf. Softw. Testing Verificat. Validation Workshops (ICSTW)*, 2015, pp. 1–7.

[32] A. Krieg *et al.*, "Power and fault emulation for software verification and system stability testing in safety critical environments," *IEEE Trans. Ind. Informat.*, vol. 9, no. 2, pp. 1199–1206, May 2013.

[33] Freescale. (2013, Jan.). *i.MX6X Graphics SDK* [Online]. Available: http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=i.MX6Q&fpsp=1&tab=Design_Tools_Tab

[34] A. Groll, J. Holle, M. Wolf, and T. Wollinger, "Next generation of automotive security: Secure hardware and secure open platforms," in *Proc. 17th World Congr. Intell. Transp. Syst.*, Oct. 2010, pp. 1–7.

[35] S. Gansel, S. Schnitzer, F. Dürr, K. Rothermel, and C. Maihöfer, "Towards virtualization concepts for novel automotive HMI systems," in *Embedded Systems: Design, Analysis and Verification*. New York, NY, USA: Springer, Jun. 2013, vol. 403, pp. 193–204.

[36] F. Chao, S. He, J. Chong, R. Ben Mrad, and L. Feng, "Development of a micromirror based laser vector scanning automotive HUD," in *Proc. Int. Conf. Mechatron. Autom. (ICMA)*. Piscataway, NJ, USA: IEEE Press, Aug. 2011, pp. 75–79.

[37] V. Milanovic, A. Kasturi, and V. Hachtel, "High brightness MEMS mirror based head-up display (HUD) modules with wireless data streaming capability," in *Proc. SPIE OPTO Conf.*, 2015, pp. 93750A–93750A.

[38] Y. Suzuki, S. Kato, H. Yamada, and K. Kono, "GPUVM: Why not virtualizing GPUs at the hypervisor?" in *Proc. USENIX Annu. Tech. Conf.*, 2014, pp. 109–120.

[39] H. Shan, K. Tian, Y. Dong, and D. Cowperthwaite, "XenGT: A software based Intel graphics virtualization solution," presented at the Xen Project Developer Summit, Oct. 2013.

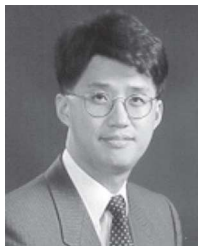
[40] K. Tian, Y. Dong, and D. Cowperthwaite, "A full GPU virtualization solution with mediated pass-through," in *Proc. USENIX Annu. Tech. Conf.*, 2014, pp. 121–132.
[41] M. Dowty and J. Sugerman, "GPU virtualization on VMware's hosted I/O architecture," *ACM SIGOPS Oper. Syst. Rev.*, vol. 43, no. 3, pp. 73–82, 2009.
[42] L. Vu, H. Sivaraman, and R. Bidarkar, "GPU virtualization for high performance general purpose computing on the ESX hypervisor," in *Proc. High Perform. Comput. Symp.*, 2014, p. 2.
[43] Y. Dong, M. Xue, X. Zheng, J. Wang, Z. Qi, and H. Guan, "Boosting GPU virtualization performance with hybrid shadow page tables," in *Proc. USENIX Annu. Tech. Conf.*, 2015, pp. 517–528.
[44] D. Shreiner, G. Sellers, J. M. Kessenich, and B. M. Licea-Kane, *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.3*, 8th ed. Reading, MA, USA: Addison-Wesley, 2013.
[45] M. Tsoukalos, "An introduction to OpenGL programming," *Linux J.*, vol. 2014, no. 244, p. 2, 2014.
[46] W. Shi, Y. Lu, Z. Li, and J. Engelsma, "SHARC: A scalable 3D graphics virtual appliance delivery framework in cloud," *J. Netw. Comput. Appl.*, vol. 34, no. 4, pp. 1078–1087, 2011.
[47] V. Gupta *et al.*, "GViM: GPU-accelerated virtual machines," in *Proc. 3rd ACM Workshop Syst. Level Virtualizat. High Perform. Comput.*, 2009, pp. 17–24.
[48] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Reading, MA, USA: Addison-Wesley, 2010.

Chiyoung Lee received the B.S. degree in computer science and engineering from Chungnam National University, Daejeon, South Korea, in 2006, and the M.S. and Ph.D. degrees in computer science from Korea University, Seoul, South Korea, in 2008 and 2015, respectively.

He is currently a Research Professor with the College of Informatics, Korea University. His research interests include virtualization of embedded systems, network virtualization based on software-defined network (SDN), and virtual resource managements.

Se-Won Kim received the B.S. and M.S. degrees in computer science from Korea University, Seoul, South Korea, in 2004 and 2006, respectively, where he is currently pursuing the Ph.D. degree in computer science.

His research interests include power management, virtualization in embedded systems, and real-time OS.



Chuck Yoo (M'14) received the B.S. and M.S. degrees in electronic engineering from Seoul National University, Seoul, South Korea, in 1982 and 1983, respectively, and the M.S. and Ph.D. degrees in computer science from the University of Michigan, Ann Arbor, MI, USA, in 1986 and 1990, respectively.

He worked as a Researcher with Sun Microsystems Laboratory, Seoul, from 1990 to 1995. He has been with Korea University, Seoul, since 1995, and is currently a Professor with the College of Informatics. His research interests include operating systems and virtualization.