Hardware Resource Virtualization for Dynamically Partially Reconfigurable Systems

Chun-Hsian Huang, *Student Member, IEEE*, and Pao-Ann Hsiung, *Senior Member, IEEE*

***Abstract*—The dynamic partial reconfiguration technology enables an embedded system to adapt its hardware functionalities at run-time to changing environment conditions. However, reconfigurable hardware functions are still managed as conventional hardware devices, and the enhancement of system performance using the partial reconfiguration technology is thus still limited. To further raise the utilization of reconfigurable hardware designs, we propose a virtual hardware mechanism, including the logic virtualization and the hardware device virtualization, for dynamically partially reconfigurable systems. Using the logic virtualization technique, a hardware function that has been configured in the field-programmable gate array (FPGA) can be virtualized to support more than one software application at run-time. Using the hardware device virtualization, a software application can access two or more different hardware functions through the same device node. In a network security reconfigurable system for multimedia applications, our experimental results also demonstrate that the utilization of reconfigurable hardware functions can be further raised using the virtual hardware mechanism. Furthermore, the virtual hardware mechanism can also reduce up to 26% of the time required by using the conventional hardware reuse.**

***Index Terms*—Hardware resource virtualization, operating system for reconfigurable systems.**

I. Introduction

DUE to rapid technology breakthroughs, field-programmable gate arrays (FPGA) devices, such as Xilinx Virtex II/II Pro, Virtex 4, and Virtex 5, can be partially reconfigured at run-time, which means that one part of the device can be reconfigured while other parts remain operational without being affected by reconfiguration. A hardware/software embedded system realized with such an FPGA device is called a *dynamically partially reconfigurable system* (DPRS), which enables more applications to be accelerated in hardware, and thus reduces the overall system execution time [6]. Compared to a conventional embedded system, a DPRS design includes not only traditional software applications and hardware devices, but also reconfigurable hardware functions running on an FPGA. However, an embedded operating system is generally not designed to support such a flexible system hardware architecture, as a result of which the enhancement of system performance using partial reconfiguration technology becomes limited. In this work, we try to bridge this gap by proposing a virtual hardware mechanism for dynamically partially reconfigurable systems. The virtual hardware mechanism is realized in an operating system for the DPRS architecture, which is called an *operating system for reconfigurable systems* (OS4RS). The virtual hardware mechanism consists of the hardware device virtualization and the logic virtualization in an OS4RS. Similar to the concept of a virtual machine, the same hardware devices and the same logic resources can be simultaneously shared between different software applications, that is, a reconfigurable hardware function can be virtualized to support more software applications.

For the kernel resources of an OS4RS, the hardware device virtualization enables the device nodes, kernel modules, and on-demand reconfigurable hardware functions to be dynamically linked at run-time to meet different system requirements. The relation between a device node, a kernel module, and a hardware function is no longer one-to-one as in a conventional embedded OS; rather, it is now a many-to-one or one-to-many mapping between the hardware functions configured on the FPGA and the software applications in the OS4RS *user space*. Given a fixed amount of logic resources in an FPGA device, the partial reconfiguration technique enables much more combinations of hardware functions to be accessed by software applications in the *user space*, than that which can be accommodated by the total amount of logic resources in the FPGA. As a result, using the virtual hardware mechanism, much more software/hardware applications can be performed in an OS4RS, even though, in reality, the system resources are not enough to cover all required hardware functions at the same time.

To realize the virtual hardware mechanism, a unified communication mechanism is also proposed to standardize the hardware/software communication interface in an OS4RS, so the device nodes, kernel modules, and reconfigurable hardware functions can be thus dynamically linked on-demand. Using the unified communication mechanism, a user-designed hardware function needs to be only integrated with a partial reconfigurable hardware task template (PR template) [2], while its control is implemented in a hardware control library. As a result, software applications can easily interact with the new hardware function by invoking the APIs in the hardware control library, thus further enhancing the system scalability. This work contributes to the state-of-the-art in the following ways.

- Using the logic virtualization technique, a hardware function that has been configured in the FPGA can be virtualized to support more than one software application at run-time. The many-to-one mapping thus increases the utilization of a hardware function.

Manuscript received June 06, 2009; revised June 30, 2009. First published July 21, 2009; current version published September 23, 2009. This manuscript was recommended for publication by A. Raghunathan.

The authors are with the Department of Computer Science and Information Engineering, National Chung Cheng University, Taiwan (e-mail: huang@cs.ccu.edu.tw; grant0920@gmail.com; pahsiung@cs.ccu.edu.tw; hpa@computer.org).

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/LES.2009.2028039

- Using the hardware device virtualization, a software application can access two or more different hardware functions through the same device node. This one-to-many mapping is a seamless reconfiguration of the underlying hardware, without any change to the software.
- From the implementation point of view, the proposed unified communication mechanism that facilitates the communication between software and hardware also enhances system scalability and design productivity.

This paper is organized as follows. The related research works are discussed in Section II. The proposed virtual hardware mechanism is described in Section III in details. Section IV introduces the unified communication mechanism, while Section V presents our implemental results and analyses. Finally, conclusions are described in Section VI.

II. Related Work

Similar to the software tasks in a conventional embedded OS, reconfigurable hardware functions can be created and removed at run-time, and thus they are called *hardware tasks* in an OS4RS. So *et al.* [4] proposed a unified hardware/software runtime environment, namely BORPH, for FPGA-based reconfigurable computers. Their OS4RS ran on a control FPGA for managing four user FPGAs on the BEE2 development platform. The hardware functions were encapsulated in the BOF file format so that they could be executed as hardware tasks in the OS4RS. However, such an encapsulation method may lead to the lack of the generality in accessing hardware devices, as a result of which their unified hardware/software runtime environment was not easy to be applied to most embedded OS.

Williams *et al.* [5] ported uClinux on the Xilinx MicroBlaze soft-core processor [7]. The partial reconfiguration process was controlled using the *internal configuration access port* (ICAP) designed as a device located under `/dev` directory. As a result, the partial bitstreams can be reconfigured in the FPGA through the `write` system call without encapsulating in a specific file format like BORPH [4]. Donato *et al.* [1] also adopted the ICAP to dynamically reconfigure the FPGA at run-time. The device driver of a reconfigurable hardware function was implemented as a kernel module capable of being dynamically inserted in the kernel of their OS4RS. When a hardware function was configured in a *partially reconfigurable region* (PRR), an IP-core manager in their OS4RS design was used to automatically load the corresponding kernel module into the Linux kernel, and thus software applications could interact with the required hardware function through the system calls. However, reconfigurable hardware functions were also managed as conventional hardware devices in the related works [1], [5], and thus the utilization of reconfigurable hardware functions is still limited.

In this work, we propose a virtual hardware mechanism to further raise the utilization of reconfigurable hardware functions, using which reconfigurable hardware functions can be virtualized to support more software applications on-demand. Similar to the related works [1], [5], our reconfigurable hardware functions are designed as regular hardware devices located under `/dev` directory. However, the device nodes, kernel modules, and on-demand reconfigurable hardware functions can be dynamically linked to meet different system requirements in our OS4RS design, without being statically linked together. To realize the virtual hardware mechanism, a unified communication mechanism is proposed to standardize the communication between software and hardware, which also facilitates the integration between user-designed hardware functions and an OS4RS.

III. Virtual Hardware Mechanism

From the design point of view, besides supporting the partial reconfiguration technology in an embedded OS, we propose a virtual hardware mechanism to further raise the utilization of reconfigurable hardware functions. The virtual hardware mechanism is mainly realized in the *kernel space* of an OS4RS and the FPGA. Similar to the interactions between software applications and hardware devices in a conventional embedded OS, software applications in our OS4RS design also interact with reconfigurable hardware functions through the device nodes, which, unlike BORPH [4], does not sacrifice the generality in accessing the hardware device design. The proposed virtual hardware mechanism consists of the logic virtualization and the hardware device virtualization in an OS4RS.

A. Logic Virtualization

When different software applications need to simultaneously interact with the same hardware function, in a conventional embedded OS, the device node specific to the hardware function must be closed by one of the software applications and then opened by another. Thus, the required hardware function can be used or accessed by different software applications at different time points turn by turn. However, such manipulations to support different software applications by continuously closing and opening the device node may lead to additional time overheads. Hence, an embedded OS needs to minimize such device manipulations so as to guarantee the *quality of service* (QoS) for each software application, especially when it is in a hard real-time environment. However, using the logic virtualization as shown in Fig. 1(a), another device node (`comm3`) can be dynamically linked to the required hardware function (`HW1`) such that it can be accessed by `Application2`. Thus, `Application2` can access `HW1`, while `Application1` is accessing `HW2`. Through the many-to-one logic virtualization, a required hardware function can be virtualized such that it can be accessed by different software applications through different device nodes. The processing results of the required hardware function are separately transferred to the kernel modules corresponding to different software applications, and then read back by the software applications in the *user space*.

B. Hardware Device Virtualization

When a software application sequentially interacts with different hardware functions, a conventional embedded OS must transfer the processing results of a required hardware function from the *kernel space* to the *user space*, and then sent to the *kernel space* again for data processing by another hardware function. This is because the interactions between a software application and its required hardware function are accomplished through the corresponding device nodes. However, the repeated data transfers between the *kernel space* and the *user*

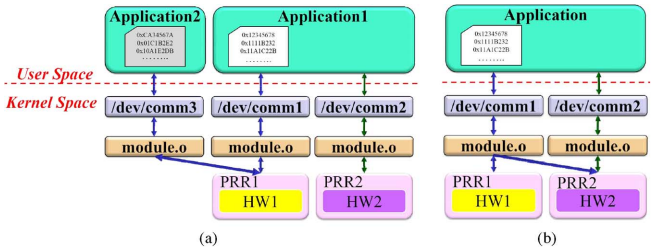


Fig. 1. Logic and hardware device virtualization: (a) logic virtualization and (b) HW device virtualization.

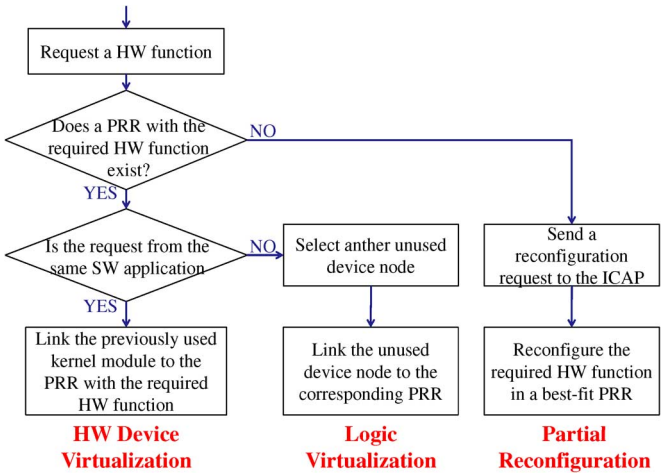
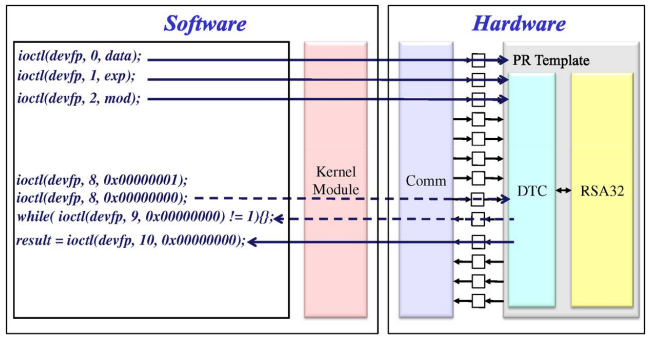


Fig. 2. Hardware task management.

space by using `copy_from_user` and `copy_to_user` kernel APIs may cause a large time overhead. Though different hardware functions, which are used for sequential data processing in an application, can be integrated into a more powerful hardware function at design-time for reducing such a large time overhead, the flexibility of the reconfigurable hardware functions, however, is thus neutralized at run-time. To keep the flexibility of reconfigurable hardware functions and to reduce such a large time overhead, using the hardware device virtualization as shown in Fig. 1(b), the kernel module corresponding to a required hardware function (`HW1`) can be dynamically linked to another required hardware function `HW2`. Through the many-to-one logic virtualization, `HW2` can be shared by different device nodes. Thus, the processing results of `HW1` can be directly transferred to `HW2` through the kernel module, and the final processing results of `HW2` are then sent back to the *user space*. As a result, the time overhead in repeatedly transferring data between the *user space* and the *kernel space* can be significantly reduced.

C. Hardware Task Management

Besides proposing the virtual hardware mechanism to further enhance the utilization of reconfigurable hardware functions, a hardware task manager is required to not only manage all data transfers between the kernel modules and the reconfigurable hardware functions, but also determine which virtualization mechanism will be used. As shown in Fig. 2, the hardware task management is divided into three categories, including the



Comm: Communication Component; DTC: Data Transformation Component

Fig. 3. Unified communication mechanism.

hardware device virtualization, the logic virtualization, and the partial reconfiguration.

When a request of hardware function is received, the hardware task manager first checks if the required hardware function has been configured in a PRR. If not, the hardware task manager thus requests the ICAP in the FPGA to reconfigure the required hardware function in a best-fit PRR. Otherwise, the hardware task manager then checks if the request is received from the same software application. If not, the logic virtualization is thus invoked to dynamically link another unused device node to the corresponding PRR. Otherwise, the hardware task manager dynamically link the previously used kernel module to the PRR with the required hardware function if it does not already exist, and thus the processing results of the previous hardware function can be directly transferred to the requested hardware function.

IV. Unified Communication Mechanism

To realize the virtualization mechanism for further raising the utilization of reconfigurable hardware functions in an OS4RS, a unified communication mechanism is thus proposed to standardize the hardware/software communication interface. As shown in Fig. 3, the unified communication mechanism is divided into the hardware part and the software part.

In the hardware part of the unified communication mechanism, a communication component is used to connect to the system bus. To ease the integration of user-designed hardware functions into an OS4RS, a partially reconfigurable hardware task template (PR template) [2] is used to connect user-designed functions with the system bus via the communication component. The PR template consists of eight 32-bit input data signals, one 32-bit input control signal, four 32-bit output data signals, and one 32-bit output control signal, while it also contains an optional *Data Transformation Component* (DTC) for unpacking incoming data and packing outgoing data based on the I/O registers sizes in the hardware functions.

In its software part, different from the device driver designed for a specific hardware function in a conventional embedded OS, a unified kernel module is designed to only interact with the fourteen 32-bit signals of the PR template. All the interactions between software applications and reconfigurable hardware functions are through the `ioctl` system calls of the unified kernel module. However, different hardware functions have

different interactive methods, and thus a hardware control library is used to implement the interactive methods of all reconfigurable hardware functions. As a result, a user-designed hardware function needs to be only integrated with the PR template, and then to update the hardware control library with its interactive method. The new hardware function can be thus accessed by software applications, which also enhances the scalability of an OS4RS and reduces system development efforts.

V. Experiments

To demonstrate that using the proposed virtual hardware mechanism the system resources can be more effectively used, we adopt a real network security reconfigurable system for multimedia applications as our example. To ensure the security and integrity of image transfers on the network, the real-time 128×64 pixel images are sequentially transferred to a cryptographic function and a hash function for data processing, and then transferred to a receiver on the network. However, the required cryptographic and hash algorithms are changed with different receivers and environment conditions. Thus, the cryptographic and hash functions are designed as reconfigurable hardware functions in the FPGA. In this experiment, the network security reconfigurable system was implemented on the Xilinx ML310 platform with a Virtex II Pro XC2VP30 FPGA chip that has 13,696 slices. The proposed virtual hardware mechanism was realized in the PetaLinux embedded OS [3], which ran on a Xilinx MicroBlaze soft-core processor [7] at 100 MHz. The network security reconfigurable system supports three cryptographic hardware functions having different key sizes in bits, including RSA32, RSA64, and RSA128, and three hash hardware functions having different input data sizes in bits, including CRC32, CRC64, and CRC128, by implementing only two different sized PRRs, namely a small `PRR1` and a large `PRR2`. The reconfigurable hardware functions can be dynamically configured in `PRR1` and `PRR2` except that RSA128 only can be configured in `PRR1`.

A. System Resource Analysis

The system resources using a conventional embedded OS, using related OS4RS designs [1], [5], and using an OS4RS design with the virtual hardware mechanism, respectively, are listed in Table I. In the multimedia application for transferring real-time images to a receiver on the network, the six hardware functions must be first configured in the conventional embedded system at design-time for being interacted with software applications at run-time. In the related OS4RS designs [1], [5], and the OS4RS design using the virtual hardware mechanism, only the logic resources of PRRs are required because the PRRs can be reconfigured to be different hardware functions at run-time for fitting different system requirements. However, reconfigurable hardware functions are still managed as conventional hardware devices in the related OS4RS designs [1], [5], and thus six device nodes and six kernel modules are required for the multimedia application. When the virtual hardware mechanism is invoked, our OS4RS design only needs at least two device nodes and two kernel modules to link to the two PRRs. Furthermore, the number of device nodes and kernel modules can be extended to support all the six hardware functions or more for being interacted with more software applications.

TABLE I
System Resource Comparison

	Conventional	Related [1], [5]	Ours
Logic Usage	3,716 slices (6 HWs)	2,975 slices (2 PRRs)	2,975 slices (2 PRRs)
#Device Node	#HW (6)	#HW (6)	#PRR $\sim$ #HW ($2 \sim 6$)
#Kernel Module	#HW (6)	#HW (6)	#PRR $\sim$ #HW ($2 \sim 6$)

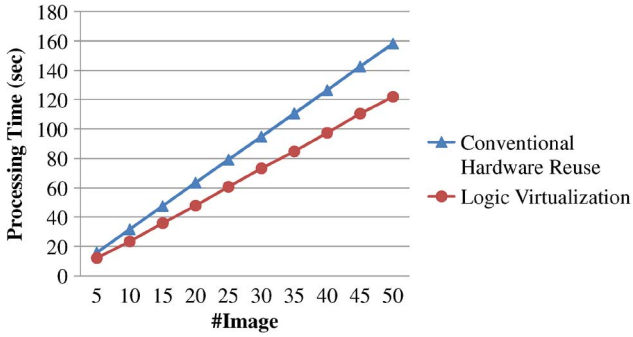


Fig. 4. Comparisons between conventional hardware reuse and many-to-one logic virtualization.

B. Time Analysis

In our first experiment, two multimedia applications simultaneously interact with the same cryptographic hardware function, where each multimedia application first captures 5 to 50 images from the camera, and then sequentially transfers the captured images to the cryptographic and hash hardware functions for data processing. Fig. 4 shows that the average time required for processing from 5 to 50 images using the logic virtualization and using the conventional hardware reuse, where one of the RSA32, RSA64, and RSA128 hardware functions is shared between two different multimedia applications for image encryption. We can observe that the time reduced by using the logic virtualization becomes more and more compared to that using the conventional hardware reuse, when the numbers of the captured images increase. Here, the reduced time is up to 26% of the time required by using the conventional hardware reuse. This is because the required cryptographic hardware function can be continuously accessed by two different multimedia applications through different device nodes turn by turn, without being blocked with one of the two multimedia applications, when the logic virtualization is performed. Further, the time overheads for continuously closing and opening the device node can be reduced.

In our second experiment, a multimedia application sequentially interacts with the cryptographic and hash hardware functions, where it first captures 5 to 50 images from the camera, and then transfers the captured images to the cryptographic and hash hardware functions for data processing. Fig. 5 shows that the average time required for processing from 5 to 50 images using the hardware device virtualization and using the conventional hardware reuse, where one of the RSA32, RSA64, and RSA128 cryptographic hardware functions and one of the CRC32, CRC64, and CRC128 hash hardware functions

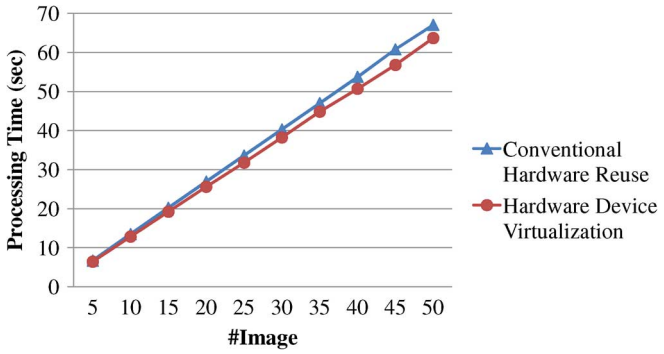


Fig. 5. Comparisons between conventional hardware reuse and one-to-many hardware device virtualization.

are used for ensuring the security and integrity, respectively, of the image transfers on the network. We can also observe that the time reduced by using the hardware device virtualization becomes more and more compared to that using the conventional hardware reuse, when the numbers of the captured images increase. Here, the reduced time is up to 6.5% of the time required by using the conventional hardware reuse. This is because the encrypted results can be directly transferred to the hash hardware function for processing through the kernel module without transferring back to the *user space*. Thus, the time overheads for repeatedly transferring data between *kernel space* and *user space* can be further reduced. The above experimental results also demonstrate that not only the utilization of reconfigurable hardware functions can be further increased, but the system performance can be also significantly enhanced, when the virtual hardware mechanism is used in an OS4RS.

VI. Conclusion

Instead of realizing reconfigurable hardware functions with traditional hardware devices in an OS4RS [1], [5], we propose a virtual hardware mechanism to further raise the utilization of reconfigurable hardware functions. Furthermore, we also propose a unified communication mechanism to further enhance the scalability of an OS4RS design, without losing the generality in accessing hardware designs. Our experimental results also demonstrate that not only the utilization of reconfigurable hardware functions can be further enhanced, but system performance can be also further improved when the virtual hardware mechanism is used in an OS4RS design.

References

[1] A. Donato, F. Ferrandi, M. D. Santambrogio, and D. Sciuto, "Operating system support for dynamically reconfigurable SoC architecture," in *Proc. IEEE Int. SOC Conf.*, Sep. 2005, pp. 233–238.

[2] C.-H. Huang and P.-A. Hsiung, "UML-based hardware/software co-design for partially reconfigurable systems," in *Proc. 13th IEEE Asia-Pacific Compute. Syst. Architect. Conf. (ACSAC)*, Aug. 2008, pp. 1–6, 10.1109/APCSAC.2008.4625436.

[3] PetaLogix, PetaLinux [Online]. Available: http://www.petalogix.com/

[4] H. K.-H. So and R. Brodersen, "A unified hardware/software runtime environment for FPGA-based reconfigurable computers using BORPH," *ACM Trans. Embedded Comput. Syst.*, vol. 7, no. 2, pp. 1–28, 2008.

[5] J. A. Williams and N. W. Bergmann, "Embedded Linux as a platform for dynamically self-reconfiguring systems-on-chip," in *Proc. Int. Conf. Eng. Reconfig. Syst. Algor.*, Jun. 2004, pp. 163–169.

[6] *"Early Access Partial Reconfiguration User Guide—UG208,"* Xilinx Inc., Mar. 2006.

[7] *"MicroBlaze Processor Reference Guide, Embedded Development Kit, EDK 8.2i—UG081 (v6.3),"* Xilinx Inc., Aug. 2006.