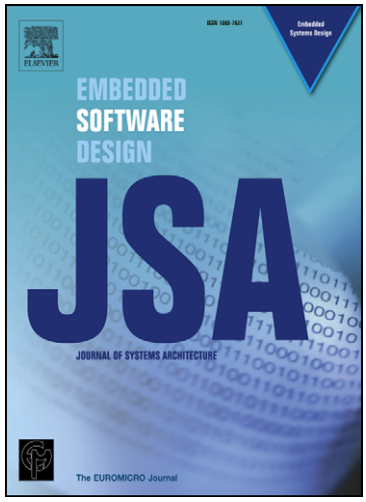


Model-Based Platform-Specific Co-Design Methodology for Dynamically Partially Reconfigurable Systems with Hardware Virtualization and Preemption

Chun-Hsian Huang, Pao-Ann Hsiung, Jih-Sheng Shen

PII: S1383-7621(10)00082-2
DOI: 10.1016/j.sysarc.2010.07.007
Reference: SYSARC 949

To appear in: *Journal of Systems Architecture*

Received Date: 15 October 2009
Revised Date: 16 April 2010
Accepted Date: 26 July 2010

Please cite this article as: C-H. Huang, P-A. Hsiung, J-S. Shen, Model-Based Platform-Specific Co-Design Methodology for Dynamically Partially Reconfigurable Systems with Hardware Virtualization and Preemption, *Journal of Systems Architecture* (2010), doi: 10.1016/j.sysarc.2010.07.007

Model-Based Platform-Specific Co-Design Methodology for Dynamically Partially Reconfigurable Systems with Hardware Virtualization and Preemption

Chun-Hsian Huang, Pao-Ann Hsiung*, Jih-Sheng Shen

Department of Computer Science and Information Engineering
National Chung Cheng University, Chiayi, Taiwan−621, ROC

Abstract

To facilitate the development of the dynamically partially reconfigurable system (DPRS), we propose a model-based platform-specific co-design (MPC) methodology for DPRS with hardware virtualization and preemption. For DPRS analysis and validation, a model-based verification and estimation framework is proposed to make model-driven architecture (MDA) more realistic and applicable to the DPRS design. Considering inherent characteristics of DPRS and real-time system requirements, a semi-automatic model translator converts the UML models of DPRS into timed automata models with transition urgency semantics for model checking. Furthermore, a UML-based hardware/software co-design platform (UCoP) can support the direct interaction between the UML models and the real hardware architecture. Compared to the existing estimation methods, UCoP can provide accurate and efficient platform-specific verification and estimation. We also propose a hierarchical design that consists of a hardware virtualization mechanism for dynamically linking the device nodes, kernel modules, and on-demand reconfigurable hardware functions and a hardware preemption mechanism for further increasing the utilization of hardware resources per unit time. Further, we realize a dynamically partially reconfigurable network security system (DPRNSS) to show the applicability and practicability of the MPC methodology. The DPRNSS can not only dynamically adapt some of its hardware functions at run-time to meet different system requirements, but also

*Corresponding author: Tel.: +886-5-272-0411 ext. 33119; Fax: +886-5-272-0859;
E-mail address: hpa@computer.org (Pao-Ann Hsiung)

determine which mechanism will be used. Our experiments also demonstrate that the hardware virtualization mechanism can save the overall system execution time up to 12.8% and the hardware preemption mechanism can reduce up to 41.3% of the time required by reconfiguration-based methods.

Keywords: UML, dynamically partially reconfigurable system, hardware/software co-design

1. Introduction

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. Through dynamic partial reconfiguration, more and more applications can be accelerated in hardware at run-time, thus effectively reducing the overall system execution time [34]. Furthermore, much more computing intensive applications can be executed as hardware functions running on an FPGA, even though the total logic resource requirements of all hardware functions are more than those of the used FPGA devices. A hardware/software embedded system realized with such an FPGA device is called a *Dynamically Partially Reconfigurable System* (DPRS) that can dynamically adapt some of its hardware functions at run-time to meet different system requirements.

Through the partial reconfiguration technology, the hardware functions can be also executed as *hardware tasks* in an embedded operating system, similar to software tasks that can be dynamically created and removed at run-time. Such an embedded operating system that supports the DPRS architecture is called an *Operating System for Reconfigurable Systems* (OS4RS), using which user applications can be executed as software tasks, hardware tasks, or both according to system performance requirements. As a result, such an OS4RS design with the DPRS platform is a self-adaptable system design, in which its functionalities can dynamically change without human intervention [17].

Many existing hardware/software co-design methodologies [4, 5, 6, 12, 23, 29, 31] have proposed different effective and innovative approaches for the DPRS development; however, they only focus on parts of the DPRS development without supporting the full design and verification flow. As a result, there still exist many gaps in the DPRS development, even though

they have many remarkable research results. Three main problems in most existing hardware/software co-design DPRS methodologies are described as follows.

1. *Model-platform information gap*: Most UML-based design methodologies use time estimates for simulating the functional interactions between applications and a system. Furthermore, the simulation-based methods cannot guarantee that all system behaviors are tested and corrected. As a result, significantly more iterations are required for rectifying the system design, and the physical design correctness can be verified and estimated only after the UML models are synthesized into concrete system designs.
2. *Low system scalability*: Reconfigurable hardware functions are usually individually implemented at design-time without supporting a unified communication interface. Therefore, to incorporate hardware functions having different data interfaces with a DPRS at run-time becomes very difficult, which does not only reduce system scalability but also increases development efforts.
3. *Limitation in infrastructure support for DPRS*: Reconfigurable hardware functions are usually managed as conventional hardware devices in most DPRS design methodologies. Therefore, the enhancement of system performance using partial reconfiguration technology is still limited, and thus makes the utilization of reconfigurable hardware functions inefficient.

Besides enhancing parts of the DPRS development, if there is a more complete hardware/software co-design methodology covering effective system analysis, complete functionality verification, accurate performance estimation, and scalable system implementation, system development efforts can be further reduced. This is the motivation and also the goal of this work, in which we propose a *Model-based Platform-specific Co-design* (MPC) methodology for dynamically partially reconfigurable systems with hardware virtualization and preemption. The contributions of this work are illustrated as follows.

- The UML models proposed in MPC are designed as reusable models, using which different user applications can be effectively developed, thus significantly saving design and analysis time. The detailed DPRS

behaviors specified by the UML models can be further used for model-level system verification and estimation, thus bridging the gap between high-level models and system implementation.

- To further enhance system scalability, the concept of the layered approach is introduced in our OS4RS design. Within the hierarchical OS4RS design, we also propose a unified communication mechanism to standardize the hardware/software communication interface such that new hardware functions can be easily integrated with an OS4RS.

- Instead of the one-to-one relation between a device node, a kernel module, and a hardware function in an embedded operating system, we propose a hardware virtualization mechanism to effectively manage the kernel resources of an operating system and the hardware logics. Thus, 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*. Using the hardware virtualization mechanism, a hardware function configured on the FPGA is virtualized such that it can be accessed by more than one application at the same time. Further, the processing results of a reconfigurable hardware function can be directly transferred to another in the *kernel space*, without a large time overhead in repeatedly transferring data between the *user space* and the *kernel space*.

- We propose generic wrapper designs that can be used by hardware functions for supporting dynamic swapping. As a result, high-priority hardware tasks can interrupt low-priority hardware tasks in real-time online system environments, which can further increase the utilization of hardware logics.

The rest of the article is organized as follows. Section 2 discusses the related DPRS design methodologies. The introduction of a DPRS design is given in Section 3. Sections 4 introduces the proposed MPC methodology, where Sections 4.1, 4.2, and 4.3 give the details of design and modeling, verification and estimation, and system implementation phases, respectively. The related experimental results and analysis are described in Section 5. Finally, conclusions are described in Section 6.

2. Related Work

Similar to the development of a conventional embedded system, that of a DPRS covers three main phases, including design and modeling, verification and estimation, and system implementation phases. In the design and modeling phase, Steinbach et al. [4] proposed a complete UML-based design methodology for reconfigurable architectures to efficiently analyze the interactions between all DPRS components. Furthermore, the UML-based design methodology included a model compiler that can help designers to translate the system-level specifications of reconfigurable architectures into executable applications. Schattkowsky et al. [31] also proposed a model-based approach for executable UML to close the gap between the system specification and its model-based execution on reconfigurable hardware. The UML specifications can be compiled to binary representations that were directly executed on their proposed abstract machine platform.

In the verification and estimation phase, besides the functional verification of a hardware design using RTL simulation, the interactions among all DPRS components are usually simulated and then verified using the SystemC language. The DPRS design methodology proposed by ITIV [5] included a SystemC-based approach [7] for modeling and simulating the DPRS. Based on the *Register Transfer Level* (RTL) SystemC, they implemented the specific operations for dynamic partial reconfiguration in the SystemC kernel. The DPRS design methodology proposed by DRESD [29] also included a SystemC-based design exploration framework [3], where the system functionalities were described using the *Transaction Level Modeling* (TLM) technique and mapped to a real system architecture. Instead of system verification using only simulation, an integrated design and verification methodology called Symbad [6] first described a reconfigurable system using SystemC for functional simulation. To exhaustively validate system correctness, the system descriptions using SystemC were then abstracted for formal verification. As a result, a DPRS could be more effectively and completely validated through both simulation and formal verification. The above verification approaches [3, 6, 7] only simulated the functional interactions between applications and a system, thus the physical design correctness of the system could be verified only after the high-level system models were synthesized into concrete system designs. The DPRS design methodology proposed by ITIV further included a model-level debugger [11] that integrated the Matlab Stateflow models with its target system. By using the JTAG cable, users can debug

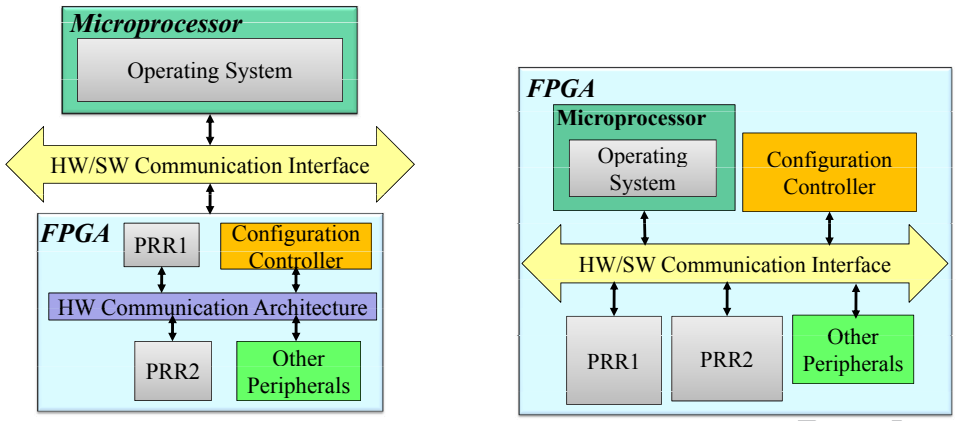
their configured system at the graphical model-level.

In the system implementation phase, besides the standalone DPRS architecture, the OS4RS design was integrated into the DPRS design methodologies [12, 26]. Thus, reconfigurable hardware functions are executed as software applications that can be dynamically created and removed. Further, the INDRA design methodology [12] proposed a hierarchical dynamically reconfigurable system that facilitates the design of a suitable on-chip communication infrastructure for partially reconfigurable systems. Dynamic switching or relocation of hardware designs in reconfigurable logic has also been proposed in the INDRA design methodology to enable the preemption of low-priority hardware tasks by high-priority hardware tasks in real-time online system environments.

Similar to the UML-based DPRS design methodologies [4, 31], the MPC methodology adopts the MDA-based UML approach to make the DPRS design more realistic and applicable in an industrial setting. However, different from the UML-based DPRS design methodologies [4, 31] that focused on the functional code generation with poor support for design-space exploration, the MPC methodology includes a complete and effective verification and estimation for a DPRS design.

Instead of only being able to simulate partial system behaviors, such as the SystemC-based methods [3, 7], the MPC methodology uses formal verification for exhaustively verifying the functional interactions between user applications and a DPRS. Furthermore, though the SymbC methodology [6] used model checking for exhaustive verification, translating the SystemC descriptions of a DPRS for model checking was not intuitive enough because they are based on different design points of view. In contrast, the verification and estimation phase in the MPC methodology is completely model-based, and thus the UML-based system specifications can be more intuitively and easily translated into timed automata for model checking. For the physical-aware verification and estimation, the model-level debugger [11] needed to suspend its model execution while reading the system state for debugging. However, the MPC methodology proposes a *UML-based hardware/software Co-design Platform* (UCoP) that not only enables the user-specified UML models to directly interact with the real system hardware architecture, but also supports for real-time tracing of the functional interactions between applications and a system.

For system implementation, the reconfiguration-based hardware preemption of INDRA [22] that required readback support from the reconfigurable



(a) SoB-based DPRS Design

(b) SoC-based DPRS Design

Figure 1: SoB-based and SoC-based DPRS Designs

logic and deep knowledge of the reconfiguration process for tasks, such as state extraction from the readback stream and manipulation of the bitstreams for context restoring. Another drawback is the poor data efficiency because only a maximum of about 8% of the readback data actually contains state information but all data must be readback to extract the state [22]. Different from the reconfiguration-based method [22], the MPC methodology adopts the design-based method that is self-sufficient because all context switching tasks are taken care of by the hardware design itself through a switching circuitry and registers can be read out or preloaded by the switching circuitry. The MPC methodology proposes two basic wrapper designs with different standard interfaces such that any digital hardware design following the standard can be transformed into dynamically switchable by interfacing with the wrappers. Further, reconfigurable hardware functions were managed as conventional hardware devices in most of the OS4RS design [12, 26], and thus the utilization of reconfigurable hardware functions is still limited. To further enhance the utilization of reconfigurable hardware functions, the MPC methodology proposes a hardware virtualization mechanism that enables the virtualization of a reconfigurable hardware function to support more than two software applications. The details of the MPC methodology will be introduced in Section 4.

3. Dynamically Partially Reconfigurable System

Before introducing the proposed MPC methodology, we first introduce the design of a dynamically partially reconfigurable system (DPRS). A DPRS is a hardware/software embedded system capable of reconfiguring new hardware functions into the system at run-time, and mainly consists of a microprocessor, an FPGA, and a hardware/software communication interface. Two types of the DPRS architecture designs, namely *System-on-Board* (SoB)-based design and *System-on-Chip* (SoC)-based design as illustrated in Figure 1(a) and Figure 1(b), respectively, can be developed. The main difference between the SoB-based design and the SoC-based design depends on whether the microprocessor is a separate chip device or is a core embedded within the FPGA device.

According to the DPRS architecture design, we can classify the DPRS components into three main categories, including hardware configuration, system management, and software application as shown in Figure 2. The hardware configuration category contains all the static and reconfigurable hardware components. The reconfigurable components in an FPGA consists of several slots called *Partially Reconfigurable Regions* (PRRs), which can be reconfigured into different hardware functions at run-time. The static components that cannot be reconfigured at run-time in an FPGA includes the static functional blocks (other peripherals in Figure 1(a) and Figure 1(b)), a hardware communication architecture that connects all hardware components in an FPGA, and a configuration controller, such as *Internal Configuration Access Port* (ICAP) or SelectMap embedded in the FPGA for configuring the partial bitstreams into PRRs. The system management components are responsible for managing the control and data transfers between hardware and software in a DPRS. It mainly includes a system manager and a hardware/software communication interface which includes the device drivers for hardware and system communication devices, such as *Peripheral Component Interconnect* (PCI) in the SoB-based DPRS design, or a system bus in the SoC-based DPRS design. The software application category includes all user-specified application functions. When we design a DPRS, there are mainly two physical constraints imposed by the partial reconfiguration technology as described in the following.

- *Mutual exclusion*: only one hardware function can be configured at a time into a PRR; only one PRR can be reconfigured at a time. This is the constraint imposed by the current FPGA devices.

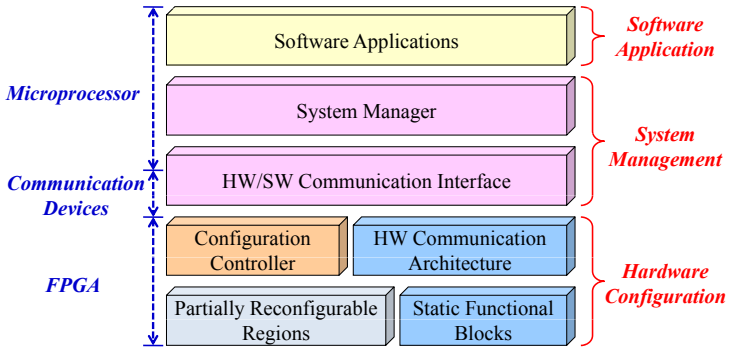


Figure 2: DPRS Components

- *No invalid access*: the software applications cannot interact with a PRR when it is being reconfigured. This is to ensure the correct system interactions between hardware and software.

4. Model-Based Platform-Specific Co-Design Methodology

The target applications in this work focus on multimedia and security systems, in which computation-intensive functions are implemented as partially reconfigurable hardware tasks in an OS4RS for enhancing system performance and flexibility. As shown in Figure 3, MPC can be separated into three phases, including modeling and design, verification and estimation, and system implementation phases. The details of the proposed MPC methodology are illustrated in the following sections.

4.1. Modeling and Design

The modeling and design phase focuses on analyzing the system-level functional interactions between user applications and a DPRS. We adopt *Unified Modeling Language* (UML), a defacto industry standard language, for DPRS modeling. To efficiently analyze the DPRS behaviors, four standard UML models are adopted in MPC, including the class diagrams for architecture modeling, the state machine diagrams for behavior modeling, the sequence diagrams for interaction modeling, and the deployment diagrams for deployment modeling. Furthermore, based on the typical DPRS design as described in Section 3, the UML models are classified into three categories of reusable UML models, namely hardware configuration models, system management models, and software application models.

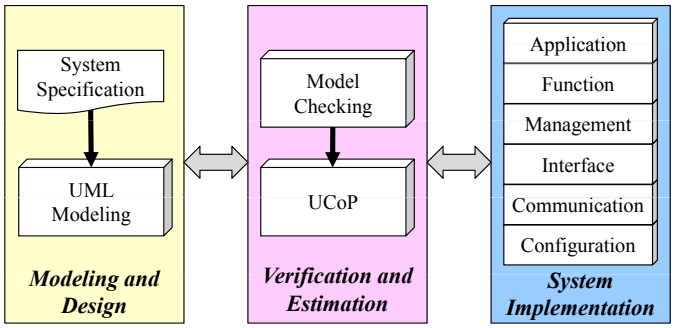


Figure 3: Model-Based Platform-Specific Co-Design Methodology

Based on the three categories of UML models, MPC provides basic UML diagram patterns for designers to model their DPRS architecture and applications. As illustrated by the class diagram of MPC in Figure 4, the hardware configuration models include the `ReconfigHW` and `StaticHW` classes which are responsible for configuring the hardware functions into the PRRs and the static area, respectively, in an FPGA. The system management models include the `SystemManager` and `ConfigManager` classes which are responsible for managing all control and data transfers in a DPRS and all FPGA configuration, respectively. The software application models include the `Interactor` and `UserDefined` classes which are responsible for providing the interactive interface between software applications and hardware functions, and the user-defined application, respectively. Besides modeling the functional relationships in a DPRS using the class diagram, MPC also provides state machine design patterns for the classes `SystemManager`, `ConfigManager`, and `Interactor` to model a new application-specific component. On applying the design patterns, the user-customized UML models are used to describe the functional behaviors of a DPRS, without any platform-related information, such as hardware configuration and execution time. Henceforth, we call them *functional UML models*.

4.2. Verification and Estimation

The verification and estimation phase focuses on supporting an efficient verification and accurate estimation mechanism, where the UML models specified at the modeling and design phase are adopted as input models of this phase. Thus, designers can use their UML models to directly verify and estimate their DPRS design, instead of performing the simulation for a

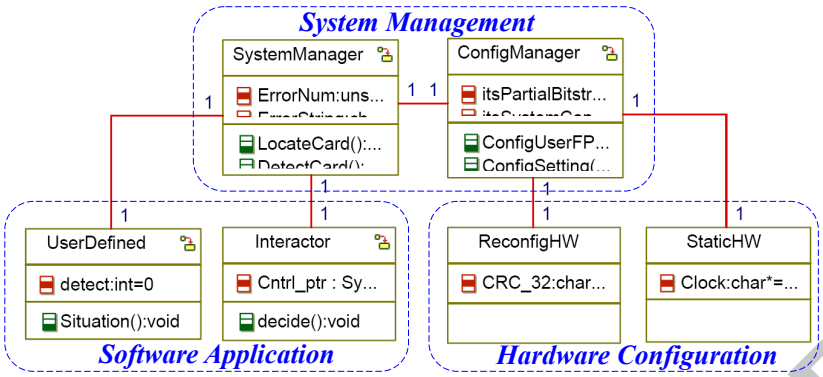


Figure 4: Class Diagram of MPC

DPRS independent from their specified UML models.

First, a formal verification method, model checking [9], is used to validate the correctness of the DPRS functional behaviors. Using model checking, all system behaviors are first described as timed automata, which are then merged into a global state graph by applying parallel composition to exhaustively validate the global system behaviors. However, the UML state machine diagrams cannot be accepted as system model input by most model checkers, which can accept only flat automata model. Thus, we apply a flattening scheme [24] to transform the state concurrency and hierarchy in the UML state machine diagrams into semantically equivalent constructs in timed automata. Further, to exhaustively validate the interactions between all DPRS components, partial reconfiguration requests in a user-given model need to be abstracted such that all combinations of reconfigurable hardware functions are model checked. As a result, the transitions triggered by partial reconfiguration requests need to be distinguished from all the transitions in the UML state machine diagram. We classify the transitions in the UML state machine diagrams into two types, namely *reconfiguration* and *general*. Reconfiguration transitions are triggered due to the partial reconfiguration requirements, and *general* transitions are the other remaining transitions. A transition in the extended UML state machine diagram of MPC has the following syntax:

```
Transition := Event [Guard] / Action ⟨Type⟩
Event:= Event name
Guard:= Boolean Expression
Action:= Operation name (Arguments) [Duration, Deadline]
```

`Type:= <Reconfiguration>, <General>`

An `Event` represents the occurrence of a stimulus to trigger a state transition, a `Guard` is a Boolean expression representing transition trigger, and an `Action` is an executable atomic operation having duration and deadline that results in a change in state.

The combination of simultaneously executing hardware functions in a DPRS changes with time and environment conditions. To model real-time system behaviors, the previously proposed urgency semantics [16] are applied to TA, and thus transitions in such *Extended Timed Automata* (ETA) are associated with urgency types, including *lazy* and *eager*. *Lazy* transitions need not be taken even if their triggers are satisfied, while *eager* transitions are triggered as soon as possible. A transition in ETA of MPC has the following syntax:

`Transition := Condition / Assignment <Urgency>`
`Condition:= Boolean Expression, Clock Constraint`
`Assignment:= Operation name, Clock Resetting`
`Urgency:= <Lazy>, <Eager>`

A `Condition` is a Boolean expression and/or a clock constraint that represents the transition trigger. An `Assignment` sets discrete variables to specific integer values and/or resets clock variables.

The transitions in ETA are semi-automatically adapted to fit the DPRS features and real-time system requirements using the model extensions. The process of transition adaptation in ETA is in the following.

- If the type of a transition in the UML state machine diagram is *reconfiguration*, the triggering condition of the corresponding transition in ETA is defined as *True* for direct triggering; otherwise, the triggering event and the guard of a transition in the UML state machine diagram are mapped to the triggering condition of the corresponding transition in ETA. By making the *reconfiguration* transitions non-deterministic, all possible functional combinations of a DPRS are verified.
- If the type of a transition in the UML state machine diagram is *reconfiguration*, the corresponding transition in ETA is associated with the *eager* urgency type, so that real-time reconfiguration is correctly modeled.

After translating user-given UML state machine diagram into ETA, the model checker can perform a verification procedure for exhaustively searching

the state space of the design, and thus show if the system satisfies a user-specified property or violates it by giving a counterexample. The properties are specified using *Computation Tree Logic* (CTL) [13]. CTL properties, such as EF, AF, AG, AU, can all be defined [13], and they are briefly introduced as follows, where p and q are atomic observations.

- *Path qualifier*: A, for all paths; E, for some paths.
- *Temporal operators*: Xp, p holds next time; Fp, p eventually holds in the future; Gp, p always holds in the future; pUq, p holds until q holds.

Second, a UML-based hardware/software co-design platform (UCoP) [8, 21] as shown in Figure 5 is used for physical-aware system verification and estimation. To realize UCoP, we integrated the FPGA platform-specific library into a UML modeling tool. The platform library consists of APIs for data access by hardware designs and for the FPGA configuration control. Users can invoke these platform APIs directly in the *functional UML models* customized in MPC, and thus the models can configure new hardware functions into the system and interact with them by sending/receiving data. The UML models that consist of the *functional UML models*, platform APIs, software executables, and hardware bitstreams are thus called *interactive UML models* in the PSV phase. As a result, UCoP allows accurate time measurements of the total operation time, including the pure execution time of a processing iteration for a hardware design and the time overheads of data transfers over the PCI bus, and the hardware configuration time, and real-time debugging, instead of only simulating the functional interactions between applications and a DPRS, which thus solves the problem of model-platform information gap.

To ease the integration of user-designed hardware functions into the UCoP, a partially reconfigurable hardware task template (PR template) is proposed, which connects the user functions with the hardware communication architecture. To use a newly developed hardware function in UCoP, a designer has to simply integrate the new hardware function with the PR template because it provides a *common* communication interface between the hardware function and the rest of the system. The PR template implements only 32-bit wide signals for all kinds of data transfers. It 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. The PR template also

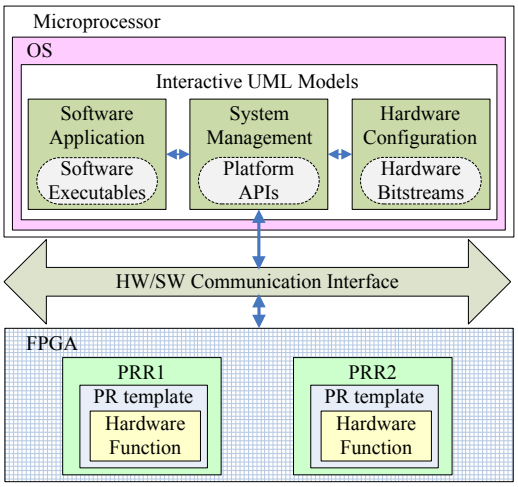


Figure 5: UML-Based HW/SW Co-Design Platform

contains an optional data transformation component for unpacking incoming data and packing outgoing data based on the I/O registers sizes in the hardware functions.

To implement the DPRS architecture, the *Early Access Partial Reconfiguration* (EA PR) flow [34] from Xilinx that provides the most complete support for partial reconfiguration technology in all industrial FPGA designs is used in UCoP. A DPRS hardware architecture consists of a static area and several PRRs. The static area design can be reused across different applications, and is thus integrated into UCoP such that users can reuse it as required in different applications. Furthermore, the necessary commands for generating partial bitstreams are integrated by UCoP into a script file. Users only need to integrate their new hardware design with the PR template, synthesize it, and run the script, without explicitly and manually going through the last two phases of the PR implementation flow step-by-step. Using UCoP, users inexperienced in the partial reconfiguration technique can still easily enhance their IP designs with the capability for partial reconfiguration and integrate them into a DPRS. Therefore, UCoP supports not only the industrially standard UML modeling for system analysis but also the generation of the partial bitstream following the EA PR standard. Through the capability for the direct interactions between the system models and real reconfigurable hardware designs, the PSV efforts can be thus significantly reduced.

Compared to the Matlab approach [11], UCoP supports real-time tracing

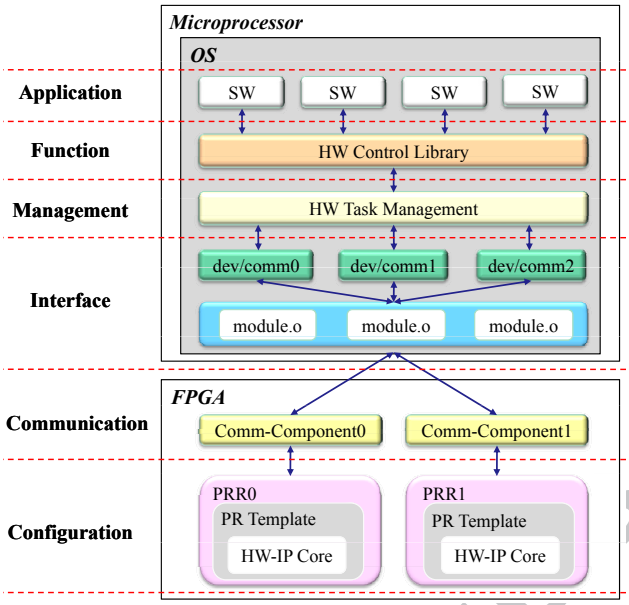


Figure 6: Hierarchical OS4RS Design

without suspending model execution. Thus, developing a DPRS in UCoP is more efficient and accurate because not only the time-consuming hardware/software co-simulation, such as using the SystemC language, can be avoided, but also real-time tracing is provided for users to verify their DPRS.

4.3. System Implementation

The system implementation phase focuses on realizing the verified design of a DPRS and further enhancing system scalability and performance. Similar to the OSI model used for computer network protocols, a layered approach is introduced in our OS4RS design that enhances the transparency of system design through a hierarchical design. As a result, the design of each layer can be separately implemented and enhanced, which benefits the development of an OS4RS. As shown in Figure 6, our hierarchical OS4RS design consists of six layers, namely *configuration*, *communication*, *interface*, *management*, *function*, and *application* layers. The *configuration* and *communication* layers are implemented on the FPGA, while the other four layers are realized in the OS4RS running on a microprocessor. The details of each layer are described in the following sections.

4.3.1. Configuration layer

The configuration layer focuses on integrating new reconfigurable hardware functions in the FPGA. To standardize user-designed hardware functions having different data interfaces, the PR template [8] used in PSV is adopted to connect user-designed functions to the communication component designed in the communication layer. To further raise the utilization of logic resources, different sizes of PRRs, which can be evaluated by the exiting method, such as the reconfiguration-aware floorplacer proposed by Montone et al. [25], are implemented on the FPGA. As a result, each reconfigurable hardware function can be (re)configured into a best-fit PRR at run-time.

Another alternative is to use the proposed hardware preemption wrappers [18, 19] for enhancing hardware functions with the capability of dynamic swapping. Thus, high priority hardware tasks can interrupt low-priority tasks in real-time embedded systems to increase the utilization of hardware space per unit time. Further, the limitation in infrastructure support for DPRS can be improved. Two basic wrapper architectures, namely *Last Interruptible State Swap* (LISS) wrapper and *Next Interruptible State Swap* (NISS) wrapper, are proposed for controlling the swapping of a hardware circuit into and out from a reconfigurable logic, such that all swap circuitry is implemented within the wrappers with minimal changes to the hardware design itself. As shown in Figure 7, the wrapper architectures consist of a context buffer to store context data, a data path for data transfer, a swap controller to manage the swap-out and swap-in activities, and some optional DTCs for (un)packing data types. The difference between the two wrappers lies in the swap-out mechanism and the hardware state in which the hardware design is swapped out. The LISS wrapper stores the hardware context at each interruptible state, thus the hardware design can be swapped out from the last interruptible state whenever there is a swap request. The NISS wrapper requires the hardware design execute until the next interruptible state, store the context, and then swap out. The different swap-out processes and the same swap-in process are described as follows.

- *LISS wrapper swap-out*: At every interruptible state, the context of hardware IP is stored in a *Context Buffer* using the `Wout_State` and `Wout_cdata` signals. When there is a `Swap_out` request from the OS4RS for some hardware task, the wrapper sends an `Interrupt` signal to the microprocessor to notify the OS4RS that (1) the context data stored in the context buffer can be read and saved into the *communication*

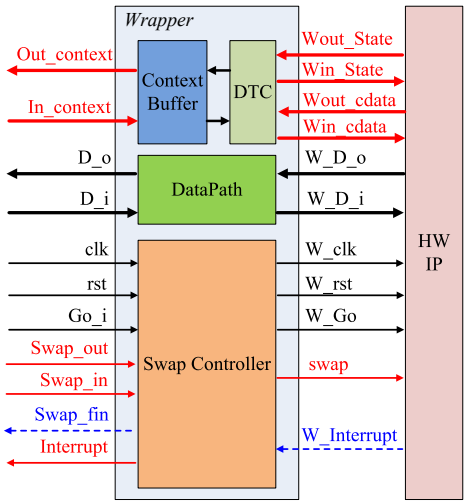


Figure 7: Wrapper Design

memory, and (2) the resources (columns) can be deallocated and reused (reconfigured). The swap-out process is thus completed. This wrapper can be used for hardware circuits whose context data size is less than that of the context buffer, as a result of which all context data can be stored in the context buffer using a single data transfer.

- *NISS wrapper swap-out*: When there is a `Swap_out` request from the OS4RS for some hardware task, the swap controller in the wrapper sends a `swap` signal (asserted high), to the hardware design, which starts the whole swap_out process. However, the hardware design might be in an unswappable state, thus execution is allowed to continue until the next swappable state is reached. At a swappable state, the context of hardware design, including current state information and selected register data, is stored in a context buffer in the wrapper using the `Wout_State` and `Wout_cdata` signals. The hardware design then sends an acknowledgment `W_interrupt` to the wrapper that the swap-out process can continue. The wrapper sends an `Interrupt` signal to the microprocessor to notify the OS4RS that the context data stored in the context buffer can be read and saved into an off-chip memory. This wrapper can be used when the context data size is larger than that of the context buffer by repeating the process of storing into buffer, interrupting microprocessor, and reading into memory. Finally when

all context data have been stored into the off-chip memory, the wrapper sends a `Swap_fin` signal to the task interface, thus notifying the OS4RS that the resources occupied by the IP can be deallocated and reused. The swap-out process is thus completed.

- *Swap-in*: When a hardware task is scheduled to execute, the OS4RS configures the corresponding hardware design with wrapper and task interface into the reconfigurable logic using the configuration controllers, reloads the context data from the communication memory to the context buffer in the wrapper, and sends a `Swap_in` request to the swap controller, which then starts to copy the context data from the buffer to the corresponding registers in the design using `Win_State` and `Win_cdata`. After all context data are restored, the swap controller sends a `swap` signal to the hardware design, which then continues from the state in which it was swapped out. It must be noted here that context data might be of different sizes for different hardware designs, so data packing and unpacking are performed using the DTC within the wrapper.

The original hardware design also needs to be enhanced so that it can interface with the LISS and NISS wrappers, which we call *standardization*. Since a combinational circuit is stateless, it can be swapped out from the reconfigurable logic as soon as it finishes the current computation. However, a sequential circuit is controlled by a *Finite State Machine* (FSM) through the present and next state registers. Generally, a hardware design has one or more data registers for storing intermediate results of computation. The collection of the state registers and data registers constitutes the *task context*. A state is said to be *interruptible* if the hardware function can resume execution from that state after restoring the task context. Not all states of a hardware function are interruptible. For the FSM of a GCD function example given in Figure 8, only the INIT, RLD, and CMP states are interruptible because the comparator results are not saved and hence we cannot resume from the NEG, EQ, and POS states.

The initial or the idle state is always interruptible. Any other state of a hardware function can be made interruptible by adding or reusing registers provided the computation can be resumed after context restoring. However, extra resources are required, thus the benefit obtained by making a state interruptible should be weighed against the overhead incurred in terms of both logic resources and context saving and restoring time. In general, making a

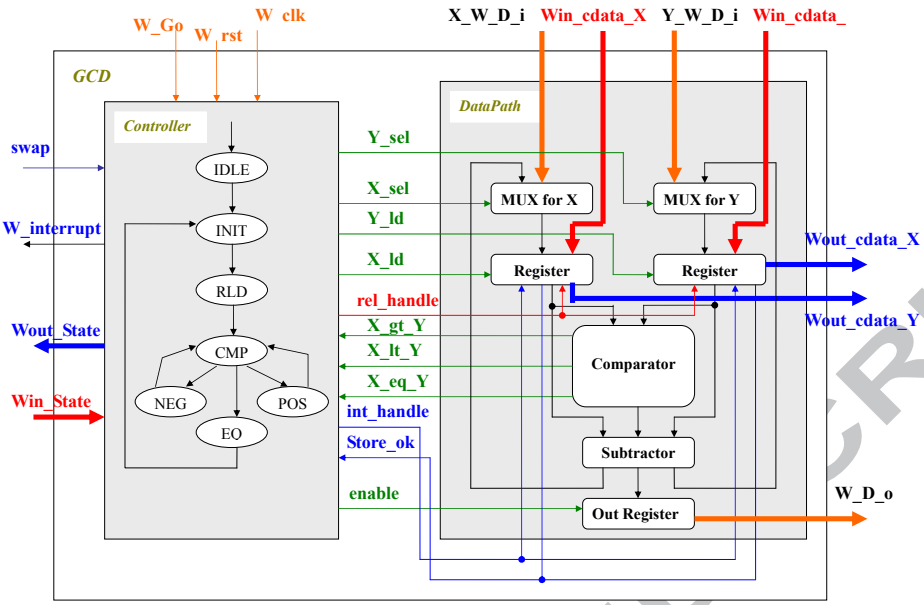


Figure 8: Swappable GCD circuit architecture

state interruptible allows the hardware function to be switched at that state, and thus the delays in executing other hardware tasks are reduced. Hence, making a state interruptible brings no benefit to the hardware function itself, instead it may shorten the overall system schedule. The decision to make a state interruptible must be derived from an overall system analysis rather than from the perspective of the hardware task itself.

A hardware function is standardized by making the context registers accessible by the wrapper and by enhancing the FSM controller such that the hardware function can be stalled at each interruptible state. For the GCD hardware function, its standardized version that is dynamically swappable is shown in Figure 8, where the two registers are made accessible to the wrapper (swap circuitry) and the FSM is modified such that the GCD hardware function can be stalled in the CMP state.

4.3.2. Communication layer

The communication layer includes the communication architecture used for data transfers among all hardware components. Furthermore, a communication component is proposed to interact with a PRR, using which software applications can access a PRR through software-accessible registers driving

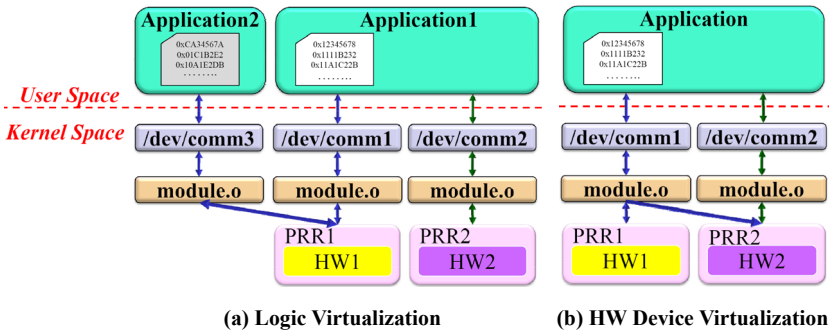


Figure 9: Logic Virtualization and Hardware Device Virtualization

the signals in the PR template. The communication component is realized using the OPB *Intellectual Property Interface* (IPIF) design, and thus the processing results can be buffered in the communication component until the OS4RS reads them.

4.3.3. Interface layer

Besides proposing a hardware preemption mechanism on the FPGA to improve the limitation in infrastructure support for DPRS, we further propose a hardware virtualization mechanism [20] realized in the *kernel space* of an OS4RS and the FPGA to enhance the utilization of reconfigurable hardware function. 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 does not sacrifice the generality in accessing the hardware device design. The hardware virtualization mechanism consists of the logic virtualization and the hardware device virtualization, and their details are described as follows.

- *Logic virtualization*: Using the logic virtualization as shown in Figure 9(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*. This many-to-one mapping thus increases the utilization of a hardware function.

- *Hardware device virtualization*: Using the hardware device virtualization as shown in Figure 9(b), the kernel module corresponding to a required hardware function (`HW1`) can be also dynamically linked to another required hardware function `HW2`. Thus, the processing results of `HW1` can be directly transferred to `HW2` through the kernel module, and the final processing results of `HW2` are sent back to the *user space*. This is because, through the many-to-one logic virtualization, `HW2` is shared by different device nodes. As a result, the time overhead in repeatedly transferring data between the *user space* and the *kernel space* can be significantly reduced. This one-to-many mapping is thus a seamless reconfiguration of the underlying hardware, without any change to the software.

4.3.4. Management layer

The management layer contains a hardware task manager to not only manage all data transfers between the kernel modules and the reconfigurable hardware functions, but also to determine which mechanism will be used when a request for a hardware function arrives. As shown in Figure 10, the hardware task management employs all the three proposed techniques, including hardware device virtualization, logic virtualization, and hardware preemption.

When a request for a hardware function arrives, the hardware task manager first checks if the required hardware function has been configured in a PRR. If not, the hardware task manager checks if the priority of the required hardware function is higher than that of any configured hardware function. If not, the required hardware function will be configured in the FPGA only after one of the configured hardware function finishes its execution and there is no other higher priority hardware function waiting to be configured. Otherwise, the hardware preemption mechanism is invoked to send a `Swap_out` request to a configured hardware function with lower priority. When the hardware task manager receives a `Swap_fin` notification, it requests the ICAP device to configure the required hardware function into the FPGA.

When the required hardware function is already configured, the hardware task manager checks if the request is received from the same software

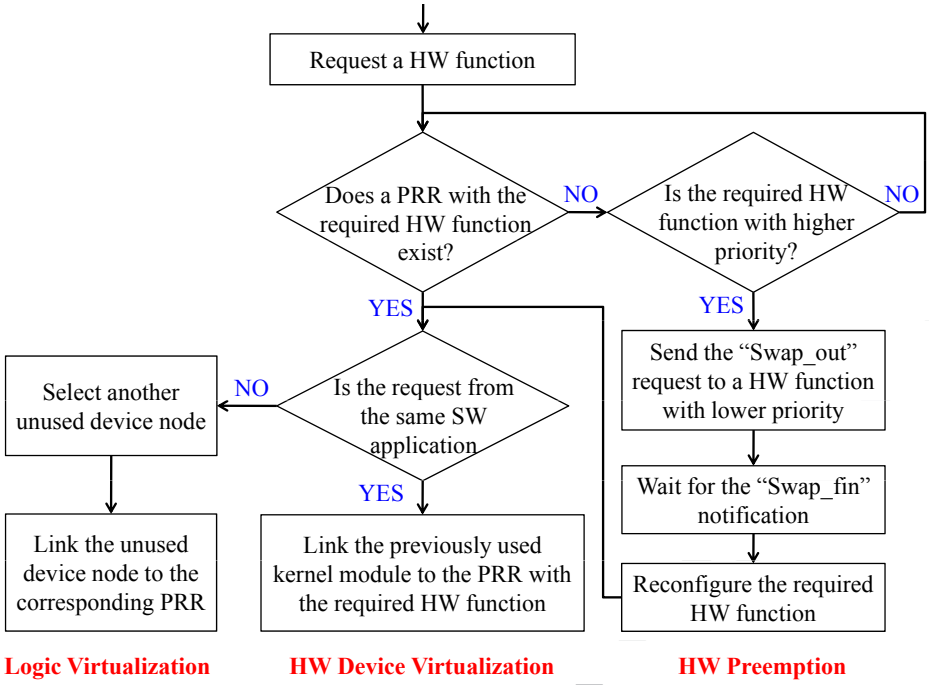


Figure 10: Hardware Task Management

application. If not, the logic virtualization is invoked to dynamically link another unused device node to the corresponding PRR. Otherwise, the hardware device virtualization is invoked to dynamically link the previously used kernel module to the PRR with the required hardware function, and thus the processing results of the previous hardware function can be directly transferred to the requested hardware function. Further, using the hardware device virtualization, when a pair of device node and kernel module is linked to only one hardware function, the final processing results are thus transferred back to the *user space*. For example, as shown in Figure 9(b), the main difference between the pairs `comm1` and `comm2` of device node and kernel module is that the pair `comm2` of device node and kernel module is connected to only `HW2`, and thus the final processing results are transferred back to the *user space* via the device node `comm2`. In contrast, the pair `comm1` of device node and kernel module is connected to both `HW1` and `HW2`, and thus the processing results of `HW1` are transferred to `HW2` via the kernel module, instead of being transferred back to the *user space* via the device node `comm1`.

In our current implementation, the hardware task manager adopts a sim-

ple *First-In-First-Out* (FIFO) method for scheduling hardware tasks. To efficiently use the hardware virtualization and preemption mechanisms to adapt changing environment conditions and runtime user requirements, in the future, the previously proposed *Relocatable Hardware-Software Scheduling* (RHSS) method [15] will be further extended and integrated in the hardware task manager.

4.3.5. Function layer

The unified kernel module is used in the *interface* layer to interact with different reconfigurable hardware functions, where fourteen `ioctl` system calls are adopted to only interact with the signals of the PR template, and are not designed for a specific hardware function. Different hardware functions have different interaction methods, and thus a hardware control library is used to implement the interaction 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 interaction method. 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.

4.3.6. Application layer

The topmost layer of the hierarchical OS4RS design is the *application* layer. An application is defined as a set of functions, which could include software and hardware tasks. Through the hardware control library, a software task can interact with a hardware task using the `ioctl` system calls.

5. Experiments

To illustrate how MPC can be applied to a real system, we use a *Dynamically Partially Reconfigurable Network Security System* (DPRNSS) as our example. DPRNSS is mainly used to support the service of *Secure Socket Layer* (SSL), for example, a *Secure Shell* (SSH) can request the DPRNSS to configure different cryptographic or hash hardware functions for data authentication and encryption/decryption, respectively. DPRNSS consists of five system devices, including a microprocessor, an FPGA, a network interface, a hardware/software communication interface, and an off-chip memory. For the dynamically partial reconfiguration of cryptographic and hash hardware

functions, some PRRs are implemented on the FPGA. All partial bitstreams for cryptographic and hash hardware designs are saved in an off-chip memory.

For the DPRNSS design, the basic controllers as shown in Figure 4 are customized, including a configuration manager, a system manager, and an interactor. Further, a negotiator is used in DPRNSS to allow a sender and a receiver to use the same cryptographic and hash algorithms for data transfer. The software application is a network multimedia application, which receives in real-time 128×64 pixel images from a camera. The received images are transferred to the cryptographic and hash hardware functions for data processing, and then sent to a receiver on the network. To validate DPRNSS, the negotiator and the network multimedia application are modeled as a new `Negotiator` class customized from the `UserDefined` class and in the `Interactor` classes, respectively, and then integrated with the *software application model* customized from MPC. We adopt the Rhapsody modeling tool [2] that has the powerful capability for code generation in C, C++, Java, and Ada, as the UML modeler. Furthermore, we use the XMI toolkit in Rhapsody to export the *functional UML models* in the XMI format.

For the new `Negotiator` class, its corresponding state machine diagram modeled by using Rhapsody is illustrated in Figure 11. The `evStart` transition is first triggered from the `Initial` state, and then the `Negotiate()` function is used to receive the requests from a receiver on the network for data transfer. If the cryptographic and hash functions that are currently configured in the FPGA are different from those requested by the receiver, the action $getItsSystemManager() \rightarrow GEN(evAdapt())$ on the `evHWAdapt` transition generates an `evAdapt` event in the `SystemManager` state machine to reconfigure the required hardware functions on the FPGA. Otherwise, the action $getItsSystemManager() \rightarrow GEN(evStable())$ on the `evNoAdapt` transition generates an `evStable` event in the `SystemManager` state machine to start data transfer. Thus, the DPRNSS can dynamically self-adapt the cryptographic and hash functions to meet the requests from different receivers on the network, without human intervention. Further, in the system implementation phase, the hierarchical OS4RS design, as described in Section 4.3, would be realized in the DPRNSS design. The hardware task manager, as described in Section 4.3.4, enables the DPRNSS itself to use the hardware virtualization and preemption mechanisms to adapt to changing environment conditions and runtime user requirements, without human intervention.

In the following sections, we show how the proposed MPC methodology can solve the three main problems of the DPRS development described in

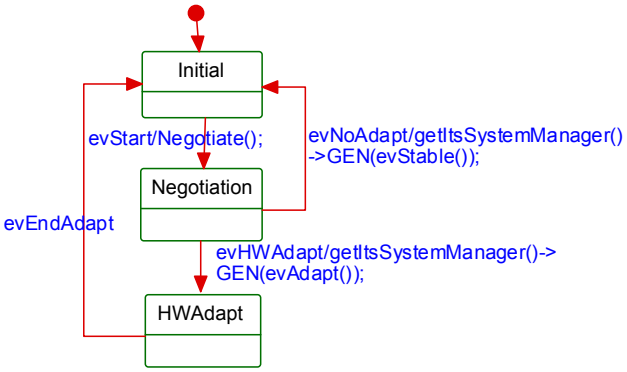


Figure 11: State Machine Diagram for Negotiator Class

Section 1 through the verification and estimation phase, the hardware virtualization mechanism, and the hardware preemption mechanism. For the system implementation phase, the XtremeDSP Development Kit-II [27] from Nallatech and Xilinx ML310 platform [36] are adopted as our reference boards for showing that the proposed MPC methodology can be applied to both the SoB-based and SoC-based DPRS designs, respectively.

5.1. PIV using Model Checking

After successfully modeling DPRNSS by the *functional UML models*, the model translator is used to transform the UML state machine diagrams into ETA models. Here, we use SGM, which runs on an Intel Pentium 4 CPU 3.00GHz with 8 GB RAM, to validate the functional interactions among all system components. We have verified nine versions of DPRNSS designs which differ in the number of PRRs from 1 to 9. More PRRs in a DPRNSS means greater flexibility and complexity. The CTL properties verified in MPC are as follows, where we have shown only the properties for PRR1, while the total number of PRRs in the system is nine.

- Mutual exclusion: $AG(mode(SystemManager) = PRR1Config \rightarrow evPR2 = 0 \ \& \ evPR3 = 0 \ \& \ evPR4 = 0 \ \& \ evPR5 = 0 \ \& \ evPR6 = 0 \ \& \ evPR7 = 0 \ \& \ evPR8 = 0 \ \& \ evPR9 = 0)$
- No invalid access: $mode(SystemManager) = PRR1Config \rightarrow A(PRR1\text{-}Access = 0)U(EndPRR1Partial = 1))$
- Starvation free : $evPR1 = 1 \rightarrow AF(mode(SystemManager) = PRR1\text{-}Config))$

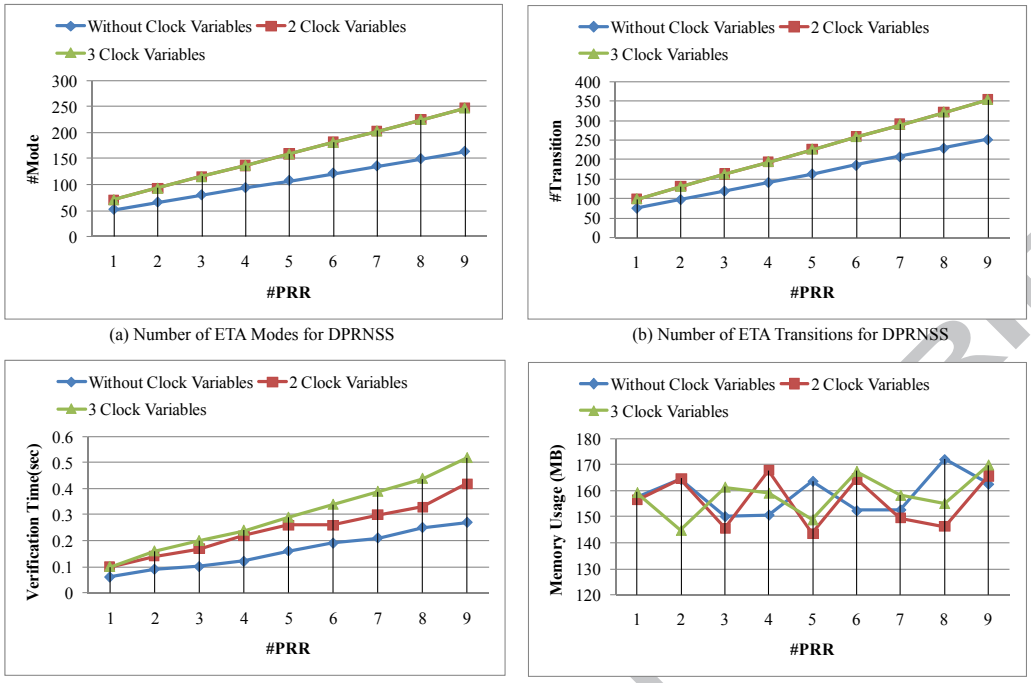
In the CTL properties, "`mode(SystemManager) = PRR1Config`" means that the `SystemManager` automaton stays in the `PRR1Config` state, which indicates that PRR1 is being reconfigured. The variables `evPRR1`, `evPRR1Access`, and `evEndPRR1Partial` with values equal to 1 show that the system manager starts to reconfigure PRR1, the interactor is interacting with the hardware function on PRR1, and PRR1 has been reconfigured, respectively.

Similar to the above CTL properties, the functional interactions with each PRR in a DPRNSS are verified using the corresponding CTL properties, and thus the total number of the verified CTL properties are changed with the number of PRRs. Table 1 shows the related information for Cases A to I, including the number of PRRs, the numbers of the total ETA modes and transitions before merging all ETA, and the number of verified CTL properties. Our first experiment is mainly to verify the purely functional interactions between DPRNSS components, that is, there is no clock variable in ETA. However, because the requirements for partial reconfiguration change over time and due to the environment conditions, the temporal feature is considered in our second experiment. The microprocessor and the FPGA have different frequencies in DPRNSS, and thus we assign two different clock variables to the software applications and the hardware functions. Further, in a DPRS, the configuration controller usually has an independent frequency different from the microprocessor and the FPGA to configure the bitstreams. Thus, a third clock variable is assigned to the operations for partial reconfiguration in our third experiment. As a result, each case in Table 1 can be further separated into three situations, including ETA without clock variables, ETA with two different clock variables, and ETA with three different clock variables.

Figure 12 shows the numbers of ETA modes and transitions of the merged ETA for all different situations and the memory usage and verification time by SGM. Because the inherent characteristics of reconfigurable systems, whose complexity is dependent on the number of PRRs, instead of functions, we

Table 1: The Related Information for Each Verified Case before Merging All ETA

Case	A	B	C	D	E	F	G	H	I
#PRR	1	2	3	4	5	6	7	8	9
#Mode	17	18	19	20	21	22	23	24	25
#Transition	22	24	26	28	30	32	34	36	38
#Property	2	6	9	12	15	18	21	24	27



(a) Number of ETA Modes for DPRNSS

(b) Number of ETA Transitions for DPRNSS

(c) Verification Time by SGM for DPRNSS

(d) Memory Usage by SGM for DPRNSS

Figure 12: Verifying DPRNSS from 1 to 9 PRRs for configuring the cryptographic and hash hardware functions

can observe that the numbers of ETA modes in Figure 12(a) and transitions in Figure 12(b), however, still increase linearly when the number of PRRs increases for any number of clock variables. Furthermore, the verification time in Figure 12(c) also seem to increases linearly, while the memory usage in Figure 12(d) is restricted to the range between 143 MB to 172 MB. By applying transition urgency semantics and dead state checking of SGM, the numbers of ETA modes and transitions of the merged ETA with three clock variables become even close to these with two clock variables as shown in Figures 12(a) and 12(b). As a result, the occurrence of the state-space explosion problem [9] can be alleviated to a certain degree in the validation of a DPRS using SGM. The theoretical underpinning for the alleviation of state-space explosion is the inherent restriction on the number of concurrently configured functions in a DPRS. In generally, the maximum number of concurrently configured function is equal to the number of PRRs, which is significantly smaller than the number of reconfigurable hardware functions. Thus, model

checking in MPC verifies each different combination of concurrently configured functions. However, to validate such timing requirements, simulation needs much more exhaustive testbench or test vectors. This usually causes much more iterations for rectifying and validating a DPRS design, especially after the system has been implemented. Therefore, MPC integrates SGM in PIV for providing efficient and exhaustive functional verification, instead of simulation-based methods, and our experiments have demonstrated that the state-space explosion problem does not occur.

5.2. PSV using UCoP

UCoP was implemented on a reference board, that is, the XtremeDSP Development Kit-II [27] from Nallatech. The software applications run on a microprocessor. The *Field Upgradeable Systems Environment* (FUSE) APIs and the PCI driver are provided by the XtremeDSP Development Kit-II to facilitate the FPGA reconfiguration and communication over the PCI bus. Instead of doing this per application, UCoP integrates them directly into the software code generator of the Rhapsody modeling tool. As a result, through the animation mode of Rhapsody, the functional interactions among the *interactive UML models* and the real DPRS hardware architecture can be dynamically traced, step by step, in the sequence diagrams and the state machine diagrams.

In order to analyze the execution process for each cryptographic function, the execution time for each cryptographic hardware design in UCoP needs to be first defined. Given input data of D_{in}-bits, output data of D_{out}-bits, data size of D_{pci}-bits for each data transfer iteration over the PCI bus, data write and data read transfer time of δ_{wr} and δ_{rd} microseconds, respectively, for each iteration over the PCI bus, initialization time of T_{pci} microseconds for starting data transfer over the PCI bus, pure execution time of T_e microseconds for a hardware design in UCoP, the total operation time is T_{total}. As shown in Equation (1), the measured total operation time includes not only the pure execution time (T_e) of a processing iteration for a hardware design, but also the time overheads of data transfers over the PCI bus.

$$T_{total} = T_{pci} + (\left\lceil \frac{D_{in}}{D_{pci}} \right\rceil \times \delta_{wr}) + T_e + (\left\lceil \frac{D_{out}}{D_{pci}} \right\rceil \times \delta_{rd}) \quad (1)$$

A common lower-bound estimation method [32] to evaluate the hardware execution time is used to compare with UCoP for demonstrating the importance of accurate time estimation, where the time necessary to transfer se-

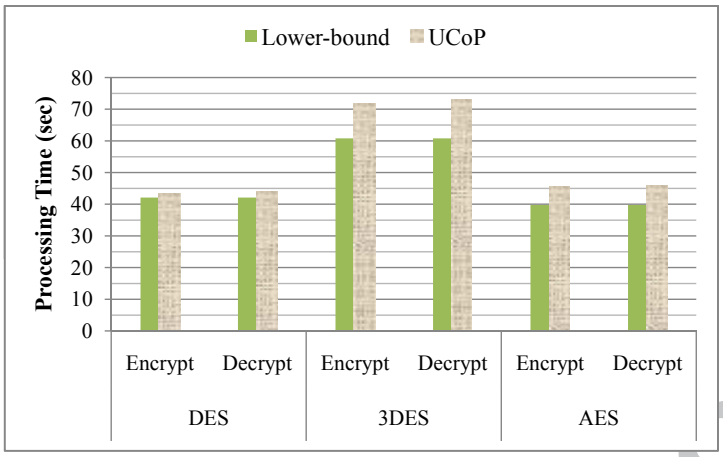


Figure 13: Estimation using Lower-bound Method and Measurement using UCoP

quences of 32-bit values was measured for obtaining the lower bound on data transfers and then used to estimate system performance. In this experiment, three cryptographic hardware functions, including *Data Encryption Standard* (DES), 3DES, and *Advanced Encryption Standard* (AES), are adopted for the image encryption/decryption in the DPRNSS. Figure 13 shows the processing time in seconds for fifty 128×64 pixel image encryption/dcryption using the lower-bound estimation method [32] and UCoP.

In a network multimedia application using the 3DES encryption with a *Quality of Service* (QoS) for 50 image frames per 65 seconds, the time for processing 50 image frames using the lower-bound estimation method and using UCoP are around 61 and 72 seconds, respectively. This shows that the lower-bound estimation method guarantee that the QoS can be achieved, however in reality it is not, which could cause a very serious problem especially when hard real-time constraints are violated. In this experiment, the time underestimation using the lower-bound estimation method is from 1.1 seconds (DES encryption) to 11.9 seconds (3DES decryption). In contrast to the inaccurate lower-bound estimation method, UCoP provides the exact measured timing results. As a result, UCoP can effectively close the model-platform information gap.

5.3. OS4RS Design using Hardware Virtualization Mechanism

In this experiment, we realized the DPRNSS design on the Xilinx ML310 platform with a Virtex II Pro XC2VP30 FPGA chip that has 13,696 slices.

The proposed hardware virtualization mechanism was realized in the PetaLinux embedded OS [28], which ran on a Xilinx MicroBlaze soft-core processor [35] at 100 MHz. The network security reconfigurable system supported four cryptographic hardware functions, including three variants of RSA having different key and input data sizes in bits (RSA32, RSA64, and RSA128) and RC6, and three hash hardware functions, including three variants of CRC having different input data sizes in bits (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 either `PRR1` or `PRR2`, except for RSA128 which can only be configured in `PRR1`. A current limitation in the proposed virtualization mechanism requires the hardware functions to have the same I/O bandwidth. For example, RC6 and CRC32 both have 32-bit I/O interfaces, and thus can be configured for hardware virtualization.

5.3.1. System Resource Analysis

Table 2 shows the system resource usage, including the logic usage, the number of device nodes, and the number of kernel modules, using a conventional embedded system, existing reconfigurable systems [12, 26], and a reconfigurable system with hardware virtualization mechanism, respectively. To support the transfer of encrypted real-time images on a network, all seven hardware functions must be configured at design-time in a conventional embedded system. Existing reconfigurable systems [12, 26] and the proposed reconfigurable system with hardware virtualization mechanism both require logic resources for the PRRs only as the PRRs can be reconfigured into different hardware functions at run-time for fitting different system requirements. However, reconfigurable hardware functions are still managed as conventional hardware devices in the existing reconfigurable systems [12, 26], and the full set of seven device nodes and seven kernel modules are required for the network security reconfigurable systems. Through the hardware virtualization mechanism, our OS4RS allows the system to work for all seven reconfigurable hardware functions using only two device nodes and two kernel modules, instead of seven as in the existing nodes. Thus, the number of device nodes and kernel modules required in our OS4RS can be minimized to the number of PRRs, instead of growing with the number of hardware functions. Since the number of PRRs is usually much smaller than that of the hardware functions, the hardware virtualization mechanism has basically placed a lower bound on the number of device nodes and kernel modules.

Table 2: System Resource Comparison

	Conventional	Related [10, 33]	Our
Logic Usage	5,010 slices (7 HWs)	2,975 slices (2 PRRs)	2,975 slices (2 PRRs)
#Device Node	#HW (7)	#HW (7)	#PRR $\sim$ #HW (2 $\sim$ 7)
#Kernel Module	#HW (7)	#HW (7)	#PRR $\sim$ #HW (2 $\sim$ 7)

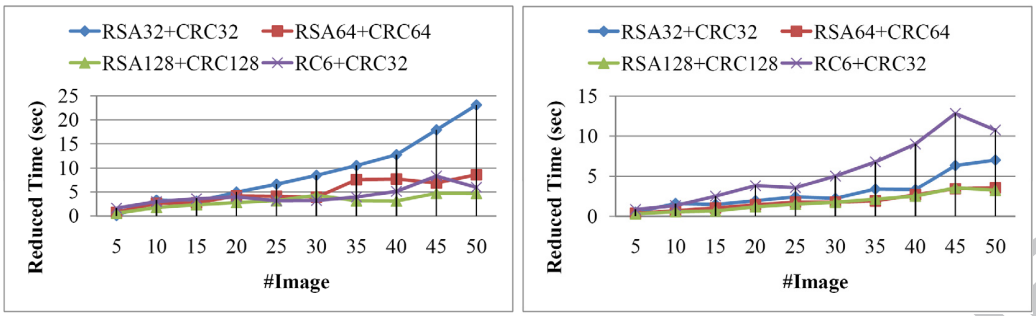
#HW: the number of hardware functions; #PRR: the number of partially reconfigurable regions.

5.3.2. Time Analysis

We compared the time required by a multimedia application for processing 5 to 50 images in two cases: (a) conventional hardware reuse, and (b) the proposed hardware virtualization mechanism. Figure 14 shows the reduced time using the logic virtualization and hardware device virtualization, respectively, compared to that using the conventional hardware reuse, for different cryptographic and hash function pairs.

In our first experiment, two multimedia applications simultaneously interact with the same cryptographic hardware function, where each multimedia application first captures real-time images from the camera, and then sequentially transfers the captured images to the cryptographic and hash hardware functions for data processing. Figure 14(a) shows the reduced time using the logic virtualization, compared to that using the conventional hardware reuse, for processing from five to fifty 128×64 pixel images. Here, one of the RSA32, RSA64, RSA128, and RC6 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 number of captured images increases. The reduced time is up to 13% for RSA32, 6% for RSA64, 4% for RSA128, and 10% for RC6 of the time required by using the conventional hardware reuse. This is because with logic virtualization the required cryptographic hardware function can be continuously accessed by two different multimedia applications through different device nodes turn by turn, without being blocked by one of the two multimedia applications, and thus the time overhead for repeatedly closing and opening the device node is alleviated.

In our second experiment, a multimedia application sequentially interacts



(a) Reduced Time using Logic Virtualization

(b) Reduced Time using Hardware Device Virtualization

Figure 14: Reduced time using Hardware Virtulization Mechanism

with the cryptographic and hash hardware functions, where it first captures real-time images from the camera, and then transfers the captured images to the cryptographic and hash hardware functions for data processing. Figure 14(b) shows the reduced time using the hardware device virtualization, compared to that using the conventional hardware reuse, for processing from 5 to 50 images. 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 number of the captured images increases. The reduced time is up to 7% for RSA32, 5% for RSA64, 5% for RSA128, and 14% for RC6 of the time required by using the conventional hardware reuse. This is because the results from the encryption functions can be directly transferred to the hash hardware function for processing through the kernel module, without transferring the data back and forth between the *user space* and *kernel space*.

From the above experimental results, we can also observe that the hardware virtualization mechanism allows greater increasing reduction in the total processing time, compared to the conventional hardware reuse, when a multimedia application requires more and more iterations for the image cryptographic and hash operations. As a result, in three RSA and CRC pairs, the most significant time reduction appears on the 32-bit one, when the logic virtualization and hardware device virtualization are used. These experiments 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 hardware virtualization mechanism is used in an OS4RS.

Table 3: Time overheads for swap-out and swap-in

	V	T_E (μs)	T_R (μs)	Swap-Out T_W	Swap-Out T_{SO} (μs)	Swap-In T_W	Swap-In T_{SI} (μs)	Task Relocate Our (μs)	Task Relocate RBM (μs)
GCD	N	13,566.8	387.9	11	388.0	5	388.0	776.0	1,038.1
	L		401.5	9	401.7	3	401.6	803.3	
DES	N	7,577.1	649.7	84	650.7	55	650.6	1,301.3	2,183.8
	L		649.7	58	650.7	29	650.6	1,301.3	
DCT	N	577,209.9	1,267.4	100	1,269.0	66	1,268.7	2,537.8	4,278.2
	L		1,254.2	68	1,255.5	34	1,255.2	2,510.7	

V: Version; N: NISS wrapper, L: LISS wrapper, RBM: Reconfiguration-based method,
T_E: execution time, T_W: time overhead incurred by wrapper (in IP clock cycles).

5.4. DPRS Architecture using Hardware Preemption Mechanism

This experiment focuses on comparing the reconfiguration-based method [22] with our proposed hardware preemption mechanism, where the *Greatest Common Divisor* (GCD), DES, and *Discrete Cosine Transform* (DCT) hardware functions are integrated with the LISS or NISS wrappers to be swappable. Given context data of D_C-bits, context buffer of D_B-bits, data transformation rate of R_T bits/cycle, buffer data load rate of R_B bits/cycle, peripheral bus data transfer rate of R_P bits/cycle, peripheral bus access time of T_A cycles, transition time of T_I cycles to go to an interruptible state , and reconfiguration time of T_R cycles, the swap-out and swap-in processes require time T_{SO} and T_{SI}, respectively, as shown in Equation (2).

$$\begin{aligned} T_{SO} &= T_I + \left\lceil \frac{D_C}{D_B} \right\rceil \times \left(\frac{D_B}{R_T} + \frac{D_B}{R_B} + T_A + \frac{D_B}{R_P} \right) + T_R \\ T_{SI} &= T_R + \left\lceil \frac{D_C}{D_B} \right\rceil \times \left(\frac{D_B}{R_T} + \frac{D_B}{R_B} + T_A + \frac{D_B}{R_P} \right) \end{aligned} \tag{2}$$

In our current implementation, the software processing time is much more than the hardware processing time. To clearly show our contributions to the hardware preemption mechanism, we directly compared our design-based method with another reconfiguration-based method (RBM) [22]. Table 3 shows the time overheads in swapping out and swapping in for all the examples. Comparing the time required for a task relocation, that is, one swap-out and one swap-in, our proposed design-based method performs better than RBM. From the experimental results, our proposed hardware preemption mechanism can reduce 40.4% and 40.6% for the NISS wrapper, and

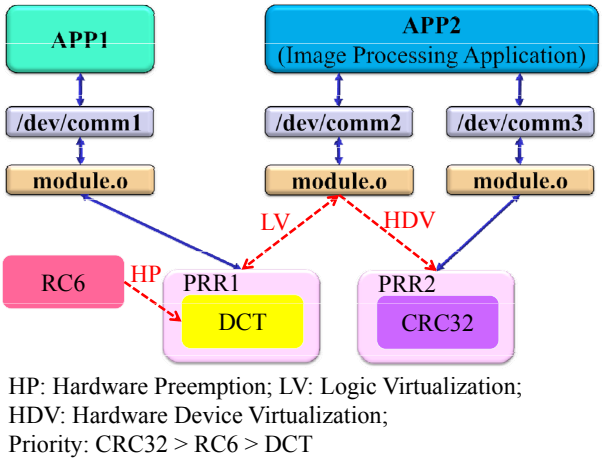


Figure 15: DPRNSS using both HW Preemption and Virtualization Mechanisms

40.4% and 41.3% for the LISS wrapper, respectively, of the time required by reconfiguration-based methods, respectively, for the larger DES and DCT examples. We are thus saving much time, which is important for hard real-time systems. Even though additional reconfiguration time is required, the swappable design would enable more hardware tasks to fit their deadline constraints, which makes the hardware-software scheduling in an OS4RS more flexible for achieving higher system performance.

5.5. DPRNSSS using both HW Preemption and Virtualization Mechanisms

To show how system performance can be significantly improved using both the hardware virtualization and preemption mechanisms, we use the DPRNSS that contains only two PRRs, namely `PRR1` and `PRR2` as shown in Figure 15, as an example. The DPRNSS receives an application `APP2` request for processing fifty images using the RC6 and CRC32 hardware functions with a QoS requirement of total 90 seconds. While scheduling the execution of the RC6 hardware function, because the DCT and CRC32 hardware functions have been configured in `PRR1` and `PRR2`, respectively, the `PRR1` with a low-priority DCT function, which has already executed for 100,000 μs, is selected to configure the required RC6 function. Without the hardware preemption mechanism, the RC6 hardware function will be configured in `PRR1` only after the DCT function finishes. According to our experimental results, it needs 477,209.9 $(577,209.9 - 100,000)$ μs to finish the current DCT execution as shown in Table 3 and 177,000 μs to configure the RC6 function. Note that

here we assume that the currently executing application APP1 (with requirement for the DCT hardware function) can also be preempted after one DCT execution. If this is not the case, a much more delay will be encountered without the hardware and software preemption. However, using the hardware preemption mechanism, it needs only 1,268.7 μs for the NISS wrapper or 1,255.2 μs for the LISS wrapper to swap out the DCT function, and 177,000 μs to configure the RC6 function. The latency in starting to serve the image processing application APP2 without and with the hardware preemption mechanism is around 0.654 seconds and 0.178 seconds, respectively.

After hardware and software preemption, the required RC6 hardware function is configured into PRR1. Using the conventional hardware reuse method, the new image processing application APP2 must now wait until the currently executing application APP1 closes the device node comm1 that is linked to PRR1, and then the image processing application APP2 opens the same device node comm1 to access PRR1 with the RC6 hardware function. However, using the proposed logic virtualization design, the hardware task manager can simultaneously use another pair comm2 of device node and kernel module to connect to PRR1 and the new application APP2 starts accessing PRR1 with the RC6 hardware function more quickly. The time saved by not waiting for another application APP1 to close and open the device node comm1, namely logic virtualization, is around 0.301 seconds according to our experimental results.

Finally, using the hardware device virtualization, the processing results of the RC6 hardware function in PRR1 can be directly transferred to the CRC32 hardware function in PRR2 through the kernel module in the pair comm2 of device node and kernel module without going back and forth between the OS kernel and the user levels, as introduced in Section 4.3.3. The total amounts of time required for performing the image processing application APP2 without and with the proposed hardware virtualization mechanism are 96.397 seconds and 85.336 seconds, respectively. As a result, without using both the hardware preemption and virtualization mechanisms for DPRNSS, the image processing application APP2 needs 97.051 ($0.654 + 96.397$) seconds, that is, the QoS requirement cannot be achieved. However, using both the hardware preemption and virtualization mechanisms for DPRNSS, the image processing application APP2 needs only 85.514 ($0.178 + 85.336$) seconds, that is, the QoS requirement can be achieved. The experimental results demonstrate that system performance can be significantly enhanced, when both the hardware preemption and virtualization mechanisms are used. This

performance improvement is very important, especially for a hard real-time system.

6. Conclusions

In this work, we discovered an important characteristic of DPRS. Unlike traditional systems, the verification complexity of DPRS is limited by the amount of functionalities that can be concurrently executed in the system. This observation allowed us to successfully apply model checking to DPRS, which not only alleviates the occurrence of the state-space explosion problem to a certain degree but also increases the verification coverage at the same time. To bridge the model-platform information gap, UCoP was proposed for platform-specific verification. UCoP supported direct interactions between the UML models and a real DPRS architecture, which was a novel approach to efficient system validation such that designers can have accurate execution and configuration time measurements that aided in efficiently and correctly verifying a DPRS.

To overcome the the limitation in infrastructure support for DPRS, we proposed a hardware virtualization mechanism to dynamically link the device nodes, kernel modules, and on-demand reconfigurable hardware functions to fit different system requirements. Furthermore, through the proposed hardware preemption mechanism, user-designed hardware functions can be dynamically swapped out and then swapped in, thus further increasing the utilization of hardware resources per unit time. We also proposed a hierarchical OS4RS design with the unified communication mechanism, using which system scalability can be further enhanced without losing the generality in accessing hardware designs.

In the future, to maintain a DPRS infrastructure to support the design of self-adaptable systems [30], our previously proposed relocatable hardware-software scheduling (RHSS) method [15] will be further extended and integrated in the hardware task manager. Thus, system performance can be self-optimized to adapt to changing environment conditions and runtime user requirements. Further, the UML profile for *Modeling and Analysis of Real Time Embedded systems* (MARTE) [1] will be integrated into MPC to support more complete and efficient system analysis. To provide a more robust validation mechanism for the DPRS development, MPC will next integrate a SystemC platform called Perfecto [14] to rapidly explore the hardware/software partitioning and scheduling alternatives at the modeling and design. As a result,

more detailed analysis at a highly abstract model-level and rapid and effective design explorations can be further supported in MPC, thus reducing design and verification efforts more significantly.

References

[1] Object Management Group (OMG). http://www.omg.org.

[2] *Rhapsody User Guide*. Telelogic Inc, 2004. http://www.telelogic.com.

[3] C. Amicucci, F. Ferrandi, M. D. Santambrogio, and D. Sciuto. SyCERS: a SystemC design exploration framework for SoC reconfigurable architecture. In *Proc. of the International Conference on Engineering of Reconfigurable Systems and Algorithm (ERSA'06)*, pages 63–69, June 2006.

[4] B. Steinbach, C. Dorotska and D. Fröhlich . Automated Hardware-Synthesis of UML-Models. In *Proc. of the 2rd International DAC Workshop, UML for SoC Design (UML-SOC'2006)*, June 2005.

[5] J. Becker, M. Hübner, G. Hettich, R. Constapel, J. Eisenmann, and J. Luka. Dynamic and Partial FPGA Exploitation. In *Proc. of the IEEE Special Issue on Advanced Automobile Technologies*, volume 95, pages 438–452, 2007.

[6] M. Borgatti, A. Capello, U. Rossi, J. L. Lambert, I. Moussa, F. Fummi, and G. Pravadelli. An Integrated Design and Verification Methodology for Reconfigurable Multimedia Systems. In *Proc. of the Design, Automation and Test in Europe Conference and Exhibition (DATE'05)*, volume 3, pages 266–271. IEEE CS Press, March 2005.

[7] A. Brito, M. Kuhnle, M. Hübner, J. Becker, and E. U. K. Melcher. Modelling and Simulation of Dynamic and Partially Reconfigurable Systems using SystemC. In *Proc. of the IEEE Computer Society Annual Symposium on VLSI (ISVLSI'07)*, pages 35–40, March 2007.

[8] C.-H. Huang and P.-A. Hsiung. UML-Based Hardware/Software Co-Design Platform for Dynamically Partially Reconfigurable Network Security Systems. In *Proc. of the 13th IEEE Asia-Pacific Computer Systems Architecture Conference (ACSAC)*, August 2008. (doi:10.1109/APCSAC.2008.4625436).

[9] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. MIT Press, 1999.

[10] A. Donato, F. Ferrandi, M. D. Santambrogio, and D. Sciuto. Operating system support for dynamically reconfigurable SoC architecture. In *Proc. of the IEEE International SOC Conference*, pages 233–238, September 2005.

[11] P. Graf, M. Hübner, K. D. Müller-Glaser, and J. Becker. A Graphical Model-Level Debugger for Heterogenous Reconfigurable Architectures. In *Proc. of the 17th IEEE International Conference on Field Programmable Logic and Applications (FPL'07)*, pages 722–725. IEEE CS Press, August 2007.

[12] J. Hagemeyer, B. Keltelhoit, M. Koester, and M. Pomnann. A design methodology for communication infrastructures on partially reconfigurable FPGAs. In *Proc. of the 17th IEEE International Conference on Field Programmable Logic and Applications (FPL'07)*, pages 331–338. IEEE CS Press, August 2007.

[13] T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic Model Checking for Real-Time Systems. In *Proc. of IEEE International Conference Logics in Computer Science*, pages 394–406, 1992.

[14] P.-A. Hsiung, C.-H. Huang, and C.-F. Liao. Perfecto: A SystemC-based performance evaluation framework for dynamically partially reconfigurable systems. In *Proc. of the 16th IEEE International Conference on Field Programmable Logic and Applications (FPL'06)*, pages 190–198. IEEE CS Press, August 2006.

[15] P.-A. Hsiung, C.-H. Huang, J.-S. Shen, and C.-C. Chiang. Scheduling and Placement of Hardware/Software Real-Time Relocatable Tasks in Dynamically Partially Reconfigurable Systems. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 2010. (to appear).

[16] P.-A. Hsiung, S.-W. Lin, Y.-R. Chen, C.-H. Huang, and W. Chu. Modeling and verification of real-time embedded systems with urgency. *Journal of Systems and Software*, 82(10):1627–1641, October 2009. (DOI: http://dx.doi.org/10.1016/j.jss.2009.03.013).

[17] P.-A. Hsiung, M. D. Santambrogio, and C.-H. Huang. *Reconfigurable System Design and Verification*. CRC Press, USA, ISBN: 978-1420062663, 2009.

[18] C.-H. Huang, S.-S. Chang, and P.-A. Hsiung. Generic wrapper design for dynamic swappable hardware IP in partially reconfigurable systems. *International Journal of Electrical Engineering (IJEE)*, 14(3):229–238, June 2007.

[19] C.-H. Huang and P.-A. Hsiung. Software-controlled dynamically swappable hardware design in partially reconfigurable systems. *EURASIP Journal on Embedded System*, 2008. Article ID 231940, 11 pages, doi:10.1155/2008/231940.

[20] C.-H. Huang and P.-A. Hsiung. Hardware resource virtualization for dynamically partially reconfigurable systems. *IEEE Embedded Systems Letters*, 1(1):19–23, May 2009. (DOI: 10.1109/LES.2009.2028039).

[21] C.-H. Huang and P.-A. Hsiung. On the Use of a UML-Based HW/SW Co-Design Platform for Reconfigurable Cryptographic Systems. In *Proc. of the IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 2221–2224. IEEE Press, May 2009.

[22] H. Kalte and M. Porrmann. Context saving and restoring for multitasking in reconfigurable systems. In *Proc. of the International Conference on Field Programmable Logic and Applications (FPL'05)*, pages 223–228. IEEE CS Press, August 2005.

[23] B. Kettelhoit and M. Porrmann. A layer model for systematically designing dynamically reconfigurable systems. In *Proc. of the 16th IEEE International Conference on Field Programmable Logic and Applications (FPL'06)*, pages 1–6. IEEE CS Press, August 2006.

[24] L. Lavazza. *A Methodology for Formalizing Concepts Underlying the DESS Notation.* EUREKA-ITEA project, 2001.

[25] A. Montone, F. Redaelli, M. D. Santambrogio, and S. O. Memik. A Reconfiguration-Aware Floorplacer for FPGAs. In *Proc. of the 2008 International Conference on Reconfigurable Computing and FPGAs (RECONFIG08)*, pages 109–114. IEEE Computer Society, 2008.

[26] M. Morandi, M. Novati, M. D. Santambrogio, and D. Sciuto. Core allocation and relocation management for a self dynamically reconfigurable architecture. In *"Proc. of 2008 IEEE Computer Society Annual Symposium on VLSI*, pages 286–291, 2008.

[27] Nallatech. XtremeDSP Development Kit-II User Guide, Issue 4, 2004.

[28] PetaLogix. PetaLinux. http://www.petalogix.com/.

[29] M. Santambrogio and D. Sciuto. Design methodology for partial dynamic reconfiguration: a new degree of freedom in the HW/SW codesign. In *Proc. of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS'08)*, pages 1–8, April 2008.

[30] M. D. Santambrogio. From reconfigurable architectures to self-adaptive autonomic systems. In *Proc. of the 2009 International Conference on Computational Science and Engineering (CSE09)*, pages 926–931. IEEE Computer Society, 2009.

[31] T. Schattkowsky, W. Mueller, and A. Rettberg. A Model-Based Approach for Executable Specifications on Reconfigurable Hardware. In *Proc. of the Conference on Design, Automation and Test in Europe (DATE'05)*, pages 692–697. IEEE Computer Society, March 2005.

[32] M. L. Silva and J. C. Ferreira. Support for Partial Run-Time Reconfiguration of Platform FPGAs. *Journal of Systems Architecture*, 52(12):709–726, December 2006.

[33] J. A. Williams and N. W. Bergmann. Embedded Linux as a platform for dynamically self-reconfiguring systems-on-chip. In *Proc. of International Conference on Engineering of Reconfigurable Systems and Algorithms*, June 2004.

[34] Xilinx. Early Access Partial Reconfiguration User Guide, UG208, 2006.

[35] Xilinx. MicroBlaze Processor Reference Guide, Embedded Development Kit, EDK 8.2i - UG081 (v6.3), August 2006.

[36] Xilinx. ML310 User Guide, Virtex-II Pro Embedded Development Platform, UG068 (v1.1.5), 2007. http://www.xilinx.com.

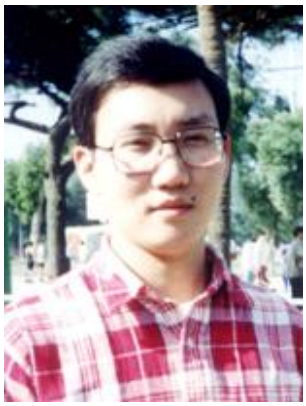
Chun-Hsian Huang received his B.S. degree in Information and Computer Education from National TaiTung University, TaiTung, Taiwan, ROC, in 2004. He is currently working toward his Ph.D. in the Department of Computer Science and Information Engineering at National Chung Cheng University, Chiayi, Taiwan, ROC. He is a teaching and research assistant in the Department of Computer Science and Information Engineering at National Chung Cheng University. His research interests include dynamically partially reconfigurable systems, UML-based hardware/software co-design methodology, hardware/software co-verification, and formal verification.

Pao-Ann Hsiung, Ph.D., received his B.S. in Mathematics and his Ph.D. in Electrical Engineering from the National Taiwan University, Taipei, Taiwan, ROC, in 1991 and 1996, respectively. From 1996 to 2000, he was a post-doctoral researcher at the Institute of Information Science, Academia Sinica, Taipei, Taiwan, ROC. From February 2001 to July 2002, he was an assistant professor and from August 2002 to July 2007 he was an associate professor in the Department of Computer Science and Information Engineering, National Chung Cheng University, Chiayi, Taiwan, ROC. Since August 2007, he has been a full professor. Dr. Hsiung was the recipient of the 2001 ACM Taipei Chapter Kuo-Ting Li Young Researcher for his significant contributions to design automation of electronic systems. Dr. Hsiung was also a recipient of the 2004 Young Scholar Research Award given by National Chung Cheng University to five young faculty members per year. Dr. Hsiung is a senior member of the IEEE, a senior member of the ACM, and a life member of the IICM. He has been included in several professional listings such as Marquis' Who's Who in the World, Marquis' Who's Who in Asia, Outstanding People of the 20th Century by International Biographical Centre, Cambridge, England, Rifacimento International's Admirable Asian Achievers (2006), Afro/Asian Who's Who, and Asia/Pacific Who's Who. Dr. Hsiung is an editorial board member of the International Journal of Embedded Systems (IJES), Inderscience Publishers, USA; the International Journal of Multimedia and Ubiquitous Engineering (IJMUE), Science and Engineering Research Center (SERSC), USA; an associate editor of the Journal of Software Engineering (JSE), Academic Journals, Inc., USA; an editorial board member of the Open Software Engineering Journal (OSE), Bentham Science Publishers, Ltd., USA; an international editorial board member of the International Journal of Patterns (IJOP). Dr. Hsiung has been on the program committee of more than 50 international conferences. He served as session organizer and chair for PDPTA'99, and as workshop organizer and chair for RTC'99, DSVV'2000, and PDES'2005. He has published more than 150 papers in international journals and conferences. He has taken an active part in paper refereeing for international journals and conferences. His main research interests include reconfigurable computing and system design, multi-core programming, cognitive radio architecture, System-on-Chip (SoC) design and verification, embedded software synthesis and verification, real-time system design and verification, hardware-software codesign and coverification, and component-based object-oriented application frameworks for real-time embedded systems.

Jih-Sheng Shen received his B.S. and his M.S. in Computer Science and Information Engineering from the I-Shou University and the National Chung Cheng University, Taiwan, ROC, in 2003 and 2004, respectively. His M.S. thesis was on the design and implementation of on-chip crossroad communication architectures for low power embedded systems. He is currently pursuing his Ph.D. in the Department of Computer Science and Information Engineering at the National Chung Cheng University, Taiwan, ROC. His research interests include the theories and the architectures of reconfigurable systems, machine learning strategies, Network-on-Chip (NoC) designs, encoding methods for minimizing crosstalk interferences and dynamic power consumption.

1. Chun-Hsian Huang's Photo

2. Pao-Ann Hsiung's Photo



3. Jih-Sheng Shen's Photo