UML-Based Design Flow and Partitioning Methodology for Dynamically Reconfigurable Computing Systems

Chih-Hao Tseng and Pao-Ann Hsiung

Department of Computer Science and Information Engineering,
National Chung-Cheng University, Chiayi, Taiwan, R.O.C.
hpa@computer.org

Abstract. Dynamically reconfigurable computing systems (DRCS) provides an intermediate tradeoff between flexibility and performance of computing systems design. Unfortunately, designing DRCS has a highly complex and formidable task. The lack of tools and design flows discourage designers from adopting the reconfigurable computing technology. A UML-based design flow for DRCS is proposed in this article. The proposed design flow is targeted at the execution speedup of functional algorithms in DRCS and at the reduction of the complexity and time-consuming efforts in designing DRCS. In particular, the most notable feature of the proposed design flow is a HW-SW partitioning methodology based on the UML 2.0 sequence diagram, called *Dynamic Bitstream Partitioning on Sequence Diagram (DBPSD)*. To prove the feasibility of the proposed design flow and DBPSD partitioning methodology, an implementation example of DES (Data Encryption Standard) encryption/decryption system is presented in this article.

Keywords: UML, sequence diagram, partitioning, design flow, reconfigurable computing, FPGA, codesign.

1 Introduction

The acceleration of computing-intensive applications requires more powerful computing architectures. Continuing improvement in microprocessor has increased computing speed, however it is still slower than the speed required by the computing-intensive applications. Microprocessors provide high flexibility in executing a wide range of applications, but they suffer from limited performance. Application specific integrated circuits (ASICs) provide superior performance, but are restricted by limited set of applications. Thus, a new computing paradigm is called for. DRCS [1], [2] is a promising solution, which provides an intermediate tradeoff between flexibility and performance.

The work in this article is concerned with the development of a design flow and of related supporting tools for DRCS. The proposed design flow takes a UML-based application model and facilitates the co-synthesis and rapid prototyping of dynamically reconfigurable computing systems. The outputs of our

L.T. Yang et al. (Eds.): EUC 2005, LNCS 3824, pp. 479–488, 2005.
© IFIP International Federation for Information Processing 2005

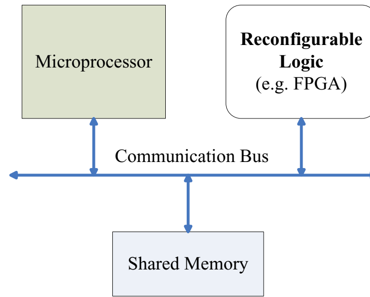

Fig. 1. Model of Reconfigurable Computing Architecture

design flow are the ready-to-run software application and hardware bitstreams for target platform, which is a dedicated FPGA (Field Programmable Gate Arrays) board connected to the host computer over the PCI (Peripheral Component Interconnect) interface. Furthermore, the primary focus of this article is on the hardware-software partitioning of UML models, which makes it different from previous researches.

Reconfigurable computing systems refer to systems, which contain a part of hardware that can change its circuits at run-time to facilitate greater flexibility without compromising performance [1]. The architecture of reconfigurable computing systems typically combine a microprocessor with reconfigurable logic resources, such as FPGA. An abstract model of such an architecture appears in Fig. 1. The microprocessor executes the main control flow of application and operations that cannot be done efficiently in the reconfigurable logic. The reconfigurable logic performs the computing-intensive parts of the application. The shared memory is used for communication between the microprocessor and the reconfigurable logic.

Unfortunately, designing these kinds of systems is a formidable task with a high complexity. Although many researches are ongoing in the academia [6]-[10] and industry, but the lack of mature tools and design flows discourage designers from adopting the reconfigurable computing technology. This leads to the need for a design flow especially easy for use by software engineers.

Issues encountered in constructing a design flow for dynamically reconfigurable computing systems are as follows.

- How to provide a HW-SW partitioning methodology, which is intuitional and easy for a software engineer?
- How to help the software engineer to synthesize the hardware bitstreams without much knowledge in digital hardware design?
- The communication between software and hardware is crucial. Thus, how must communication be designed easily, correctly, and efficiently?
- What kind of target hardware platform is appropriate for this design flow?

To solve the issues mentioned above, we develop a UML-based design flow and related supporting tools for the rapid application prototyping of dynamically

reconfigurable computing systems. The features of our proposed solutions are as follows:

- Supporting software-oriented strategy for co-synthesis as we start from a UML specification and identify parts which are to be implemented in reconfigurable hardware.
- Automatic synthesis of bitstreams for reconfigurable hardware.
- Automatic generation of software/hardware interfaces.
- Using commercial off-the-shelf FPGA within PCI card to facilitate the construction of target platform and using API (Application Programming Interface) for reconfiguration.

The article is organized as follows. In Sect. 2, we give a detailed discussion of the proposed design flow. Section 3 is the core of this article, where we present our partitioning methodology for UML models. In Sect. 4, some examples are presented to show the feasibility of the proposed design flow. In the last section, we conclude this article and introduce some future work.

2 The Design Flow

The proposed design flow, as shown in Fig. 2, is separated into three phases: *design and implementation of the system software model*, *hardware synthesis*, and *software synthesis*. C++ code and UML 2.0 diagrams such as the class diagram, sequence diagram, and state machine diagram are the inputs of the proposed design flow. Reconfigurable C++ application and bitstreams are outputs of our design flow. In Fig. 2, the elliptical boxes such as Class Diagram, XMI Documents, and Bitstreams represent the workproduct in each phase. The rectangular boxes such as Rhapsody 5.0, C++ Compiler, Forge, and XFlow represent commercial-off-the-shelf tools. The three-dimensional rectangular boxes such as SW/HW Extractor, SW Interface Synthesizer, and HW Interface Synthesizer represent tools developed by ourselves. Certainly, the target platform for verifying the proposed design flow is also constructed. Each phase of the proposed design flow and the target platform will be examined further in the following subsections.

2.1 Design and Implementation of System Software Model

In the *Design and Implementation of System Software Model* phase of the proposed design flow, we use the Rhapsody 5.0 tool to build the UML models, implement the detailed behaviors in C++, verify the functionalities of the system software model, and partition performance critical methods into hardware. After the model is constructed, XMI documents are generated by the XMI toolkit of Rhapsody 5.0. XMI documents use the XML format to store UML model information, and C++ code is also included. We then use our SW/HW Extractor to parse partitioning information from the XMI documents. The SW/HW Extractor searches the XMI documents to locate UML stereotype <<HW>>

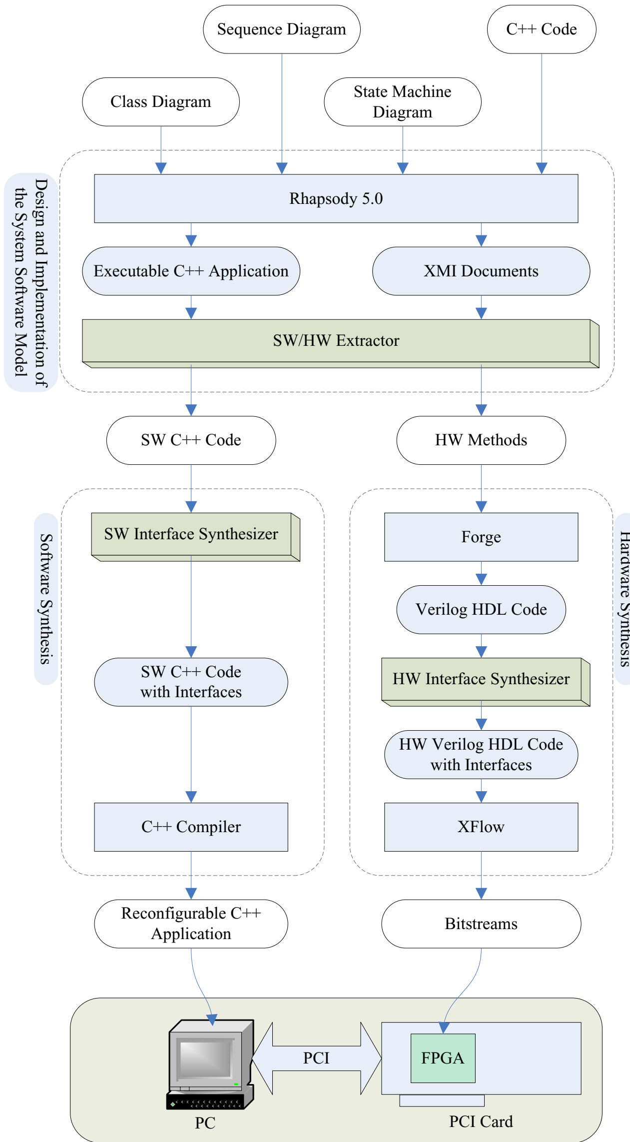

Fig. 2. Design Flow for Dynamically Reconfigurable Computing Systems

marked by user, then it extracts this portion of C++ method as HW method for post synthesis in the *Hardware Synthesis* phase. The SW C++ code is another output of SW/HW Extractor, which represents all of the C++ application code. This SW C++ code will be the input of the *Software Synthesis* phase. After the *Design and Implementation of System Software Model* phase, the design flow is split up into the *Hardware Synthesis* phase and the *Software Synthesis* phase.

The Unified Modeling Language (UML) [3] is a standard modeling language used in the software industry. In this work, we have chosen three diagrams from UML 2.0 for building the system model, namely class diagram, state machine diagram, and sequence diagram. Class diagrams are used to model the architecture of software application.

State machine diagrams describe the dynamic behavior of a class in response to external stimuli, such as event or trigger. There exists a gap between the state machine diagram and its implementation. According to [4], the state machine diagrams include many concepts such as states and transitions that are not present in most object-oriented programming languages, like Java or C++. In order to overcome this gap, we adopt Rhapsody [5] as our UML modeling tool. After drawing UML models in Rhapsody, the tool can generate Ada, C, C++, or Java code. Rhapsody provides an Object Execution Framework (OXF) [5], which enables state machine diagrams to be implemented in object-oriented programming languages.

Sequence diagrams show the interactions between classes in a scenario that is a partial system behavior of overall system specifications. In a sequence diagram, classes and objects are listed horizontally as columns, with vertical lifelines indicating the lifetime of the object over time. Messages are rendered as horizontal arrows between classes or objects and represent the communication between classes or objects.

2.2 Hardware Synthesis

The purpose of the *Hardware Synthesis* phase is to synthesize the hardware bitstreams which will be used by software applications, to perform some required computing-intensive operation. The HW Methods derived from SW/HW Extractor are the inputs of this phase. First, the Forge tool is used to transform the HW Methods into synthesizable Verilog HDL (Hardware Description Language) code. Secondly, the HW Interface Synthesizer adds PCI wrapper for the Verilog HDL code, then produces the HW Verilog HDL code with interface which enable communication from software. Finally, the XFlow tool will synthesize the HW Verilog HDL code with interface into the hardware bitstreams for execution in FPGA.

2.3 Software Synthesis

The purpose of the *Software Synthesis* phase is to build an executable C++ application, which is capable of invoking hardware methods on demand. The Software C++ code which was derived from the SW/HW Extractor is the input of this phase. Starting from this code, the Sofware Interface Synthesizer is used to synthesize code for accessing hardware methods. After that, the produced Software C++ code with interfaces is the final source code. Finally, this code is compiled by a C++ compiler to generate a ready-to-run C++ program, called Reconfigurable C++ Application. During the execution on the host processor, this Reconfigurable C++ Application can reconfigure required hardware method into FPGA for acceleration of the software execution.

2.4 Target Platform

Target platform allows system designers to verify the overall system behavior and to evaluate the overall system performance. Our target platform is a dedicated Xilinx Virtex-II FPGA (XC2V3000, 28,672 LUTs, at 40MHz) board connected to the host computer (Pentium 4 2.8GHz, 1GB RAM, Windows XP) over the PCI interface.

3 The DBPSD Partitioning Methodology

We propose a *Dynamic Bitstream Partitioning on Sequence Diagrams (DBPSD)*, which is a partitioning methodology based on the UML 2.0 sequence diagrams and includes *partitioning guidelines* to help designers make prudent partitioning decisions at the granularity of class methods. Sequence diagrams in UML 2.0 have been significantly enhanced for specifying complex control flows in one sequence diagram. As shown in the middle of Fig. 3, the most obvious changes are the three rectangular boxes, called *interaction fragments*. The five-sided box with labels such as **alt**, **opt**, or **loop** at the upper left-hand corner is the *interaction operator* of the interaction fragment.

The interaction operator gives the interaction fragment specific meaning. The **alt** operator denotes a conditional choice according to the evaluation results of

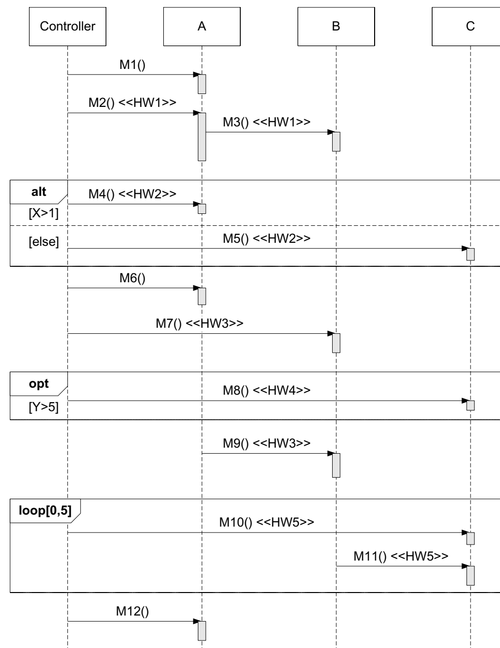

Fig. 3. The Example of the Partitioning on UML 2.0 Sequence Diagram

the guards. For example, if the guard [**x>1**] is evaluated to true, then the **M4()** method will be called, otherwise the [**else**] guard will be evaluated to true and then the **M5()** method will be called. The **opt** operator is the **if** portion of the **alt**, that is the same as the **if** construct in the C language. The **loop** operator defines that an interaction fragment shall be repeated several times.

When doing partitioning on the sequence diagrams, a designer may add a UML stereotype <<HWx>> for a method to be implemented in hardware. For example, in Fig. 3 the methods M2() and M3() are marked by the same stereotype <<HW1>>, but the method M4() is marked by another stereotype <<HW2>>. As a consequence, the methods M2() and M3() will be synthesized into the same bitstream, but the method M4() will be synthesized into another bitstream. Calling a hardware method that is synthesized into a different bitstream will require the FPGA to be reconfigured, therefore additional time overhead will be incurred.

The key performance penalties in DRCS are the FPGA reconfiguration time and the communication time between the CPU and the FPGA. These overheads are mainly dependent on the hardware restrictions. However, we can reduce the number of FPGA reconfigurations, so that reconfiguration overhead is decreased.

To reduce the number of FPGA reconfigurations, we need to take the control flow and execution order into consideration when doing partitioning on sequence diagrams. Hence, in DBPSD the following partitioning guidelines are provided:

Guideline 1: *Add the same stereotype <<HWx>> to all computing-intensive methods.* For example, M1() <<HW1>>, M2() <<HW1>>, ..., M12() <<-HW1>>. If synthesis is feasible, only one bitstream is produced, thus only one reconfiguration action is needed.

Guideline 2: *Add the same stereotype <<HWx>> to all dependent methods.* For example, M3() is invoked by M2().

Guideline 3: *For the **alt** operator, add the same stereotype <<HWx>> to all the computing-intensive methods in all condition branches.* If the synthesis of the stereotyped methods is not possible, then start moving the last conditional branch to another stereotype <<HWy>> first, until synthesis is successful.

Guideline 4: *For the **opt** operator, associate all of the methods inside this interaction fragment with a stereotype different from the methods outside this interaction fragment.*

Guideline 5: *For the **loop** operator, associate the same stereotype <<HWx>> to all the computing-intensive methods inside this interaction fragment.* If the synthesis of the stereotyped methods is not possible, then start moving the less computing-intensive method to another stereotype <<HWy>> first, until synthesis is successful.

4 Implementation Example

An implementation example of DES (Data Encryption Standard) encryption system is presented to prove the feasibility of the proposed design flow and

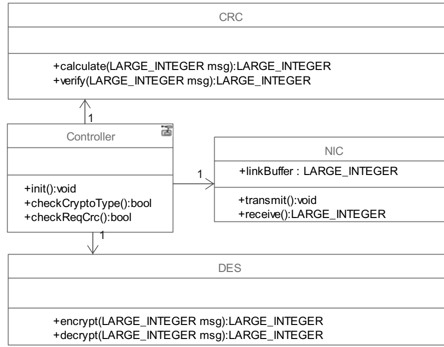

Fig. 4. The Class Diagram of the DES IMS System Example

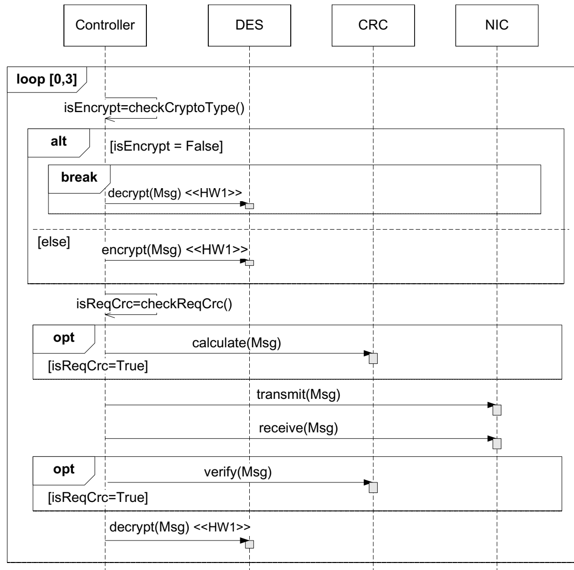

Fig. 5. The Sequence Diagram of the DES IMS System Example

Table 1. The Implementation Results of Different Partitions

	Partition 1	Partition 2	Partition 3	Partition 4
encrypt()	SW	SW	HW	HW
decrypt()	SW	HW	SW	HW
Total Execution Time	7200us	2560us	4880us	240us
FPGA Utilization (%LUTs)	0%	18%	18%	36%
Speedup Compared to Partition1	—	+2.81x	+1.48x	+30x

DBPSD partitioning methodology. Starting from the *design and implementation of the system software model* phase of the proposed design flow, the designer constructs the class diagram, the state machine diagrams, and the sequence diagrams for modeling this system, then the detailed functions are implemented in the C++ language.

Figure 4 is the class diagram for this system, which contains four classes: `Controller` (for controlling of the overall system), `NIC` (for simulating of the network interface), `CRC` (for calculating the CRC value), and `DES` (for encryption and decryption of message). Due to the limited space, the state machine diagram of this example is not shown.

The sequence diagram which depicts the overall system interaction is shown in Fig. 5. After the profiling procedure, we found that the `DES` class is a computing-intensive part of this system, thus the partitioning focuses on its two methods: encrypt() and decrypt(). Figure 5 shows the optimal partition for this example. The other partitions and their implementation results are shown in Table 1.

As shown in Table 1, the most notable partitions are Partition 2 and Partition 3. The difference in the total execution time of Partition 2 and Partition 3 was not expected. The reason for this difference can be observed from the sequence diagram of this example. Different control flows affect the number of times each method is invoked. Thus, the worth of doing partitioning on sequence diagram is proved by this example.

5 Conclusions

A UML-based design flow and its HW-SW partitioning methodology are presented in this article. The enhanced sequence diagram in UML 2.0 is capable of modelling complex control flows, thus the partitioning can be done efficiently on the sequence diagrams. As a result of using the proposed design flow, we are able to efficiently implement DRCS with significant reduction of error-prone, tedious, and time-consuming tasks, such as hardware design and HW-SW interface synthesis. Additionally, the real implementation results and information produced by the proposed flow such as application performance datum, hardware method execution time, FPGA reconfiguration time and communication overheads can used for further simulation or evaluation. Further research directions of this article include the semi-automatic or automatic HW-SW partitioning on sequence diagram, algorithm or methodology for reconfiguration overhead reduction, and support for FPGA partial reconfiguration.

References

1. K. Bondalapati and V. K. Prasanna, “Reconfigurable computing systems,” Proceedings of the IEEE, 90(7):1201-1217, July 2002.
2. K. Compton and S. Hauck, “Reconfigurable computing: A survey of systems and software,” ACM Computing Surveys, 34(2):171-210, June 2002.
3. G. Booch, J. Rumbaugh, and I. Jacobson, “Unified Modeling Language User Guide,” Addison-Wesley, 1999.

4. I. A. Niaz and J. Tanaka, “Mapping UML statecharts to Java code,” in Proceedings of the IASTED International Conference on Software Engineering (SE 2004), pages 111-116, February 2004.
5. Rhapsody case tool reference manual, I-Logix Inc, http://www.ilogix.com.
6. T. Beierlein, D. Frőhlich, and B. Steinbach, “UML-based co-design for run-time reconfigurable architectures,” in Proceedings of the Forum on Specification and Design Languages (FDL03), pages 5-19, September 2003.
7. T. Beierlein, D. Frőhlich, and B. Steinbach, “Model-driven compilation of UML-models for reconfigurable architectures,” in Proceedings of the Second RTAS Workshop on Model-Driven Embedded Systems (MoDES04), May 2004.
8. J. Fleischmann, K. Buchenrieder, and R. Kress, “A hardware/software prototyping environment for dynamically reconfigurable embedded systems,” in Proceedings of the 6th International Workshop on Hardware/Software Codesign, pages 105-109, IEEE Computer Society, March 1998.
9. J. Fleischmann, K. Buchenrieder, and R. Kress, “Java driven codesign and prototyping of networked embedded systems,” in Proceedings of the 36th ACM/IEEE Design Automation Conference (DAC99), pages 794-797, ACM Press, June 1999.
10. I. Robertson and J. Irvine, “A design flow for partially reconfigurable hardware,” ACM Transactions on Embedded Computing Systems, 3(2):257-283, May 2004.