A UML-Based Approach for Heterogeneous IP Integration

Sun Zhenxin

School of Computing
National University of Singapore
COM1, Singapore 117590
sunzhenx@comp.nus.edu.sg

Wong Weng-Fai

School of Computing
National University of Singapore
COM1, Singapore 117590
wongwf@comp.nus.edu.sg

Abstract - With increasing availability of predefined IP (Intellectual Properties) blocks and inexpensive microprocessors, embedded system designers are faced with more design choices than ever. On the other hand, there is a constant pressure on reducing the time to market. However, as the IP blocks are provided by different vendors, they differ in their interfaces. In order to improve design reuse, methods for combining heterogeneous IP blocks with incompatible protocols and I/Os are needed. In this paper, we propose an interface synthesis method that uses the UML notation to model the interfaces of predefined components and glue logic within the standard OCP-compliant environment. We built a code generator to produce the interface adapters from the UML models. We experimented with our approach using simple-bus and a MPEG-2 decoder as case studies.

I Introduction

The use of pre-designed IP blocks to reduce the complexity of system integration has gained popularity lately. Using pre-designed IP blocks leads to the reduction of time and complexity of system level design. However, these IP blocks often have interfaces that are incompatible due to differences in protocol and/or unmatched I/Os. Integration suffers from these incompatibility issues which also hampers design exploration especially when there are many alternative solutions [2]. The problem of incompatible IP protocols is well-known, and efforts have been made to address it by standardizing the communication protocol. Several standards have been proposed. Among these, the Open Core Protocol (OCP) by OCP-IP [13] has gained wide industrial acceptance. Today, many IPs are OCP-compliant. However, for existing non OCP compliant IP cores, it is expensive to customize them to comply with the OCP standard. The process of IP integration also suffers from mis-matched I/O ports. For example, a 16-bit processor will have problems connecting to a 32-bit bus interface. These kinds of situations require logic to be introduced in between the interfaces. Design of such logic circuit is typically done manually and is therefore tedious and error prone. In this paper, we address the above problems by the automatic generation of OCP-compliant adapters for non-compliant components with fixed interfaces from UML structural and behavioral models.

Interface synthesis consists of interface modeling and realization. The incompatible problem has been addressed by previous works in the literature. A common way to capture the system interface specification is the utilization of software programming language or an interface description language, such as variants of C, C++, or Java. [7][9][12][19] Such a specification has the advantage of being executable, and thereby facilitates early verification and simulation. However, for the purpose of system level specifications, the use of these programming languages does not satisfy all the requirements. One key issue is that the different phases of a system design flow — namely the requirement, design, implementation, and deployment — are not sufficiently separated. This can seriously confuse the issues that have to be addressed by each of these phases because of duplication or oversight.

Earlier efforts have also been made in context of timing analysis and verification. Interfaces synthesis tools based on specifying low level data port behaviors through timing diagrams have been proposed. In [1], a method to synthesize interface blocks that consist of logic circuits and software routines were presented. Other works propose the use of control flow graph [3], event graphs [4], and signal transition graphs [10] to model interface behaviors. All these works address the interface synthesis problem for certain platform environment. However, there can be a large choice of IP. Therefore, the unified modeling for heterogeneous IP would benefit designers working through a top-down design process. UML-based approaches also have been studied [5][8][15]. However, these methods are not matured enough and the mapping rules to low level synthesis are usually too simple to handle realistic scenarios.

Our approach differs from others in the use of high level UML notations for *heterogeneous* IP integration. The following are features that are unique in our approach:

- To ensure correctness and reusability, we use UML structural and state diagrams to specify and formalize system interfaces. This single model is used consistently throughout the entire design process. It not only gives a system level view of the design but also allows for reuse in future designs.
- Automation is applied in every level of abstractions, and between different environments. Code is generated from the same source model, minimizing ambiguity.
- Our framework supports both interface protocol customization and glue logic generation, thereby maximizing IP integration.
- All changes are applied at the higher level, and user will only need to deal with the high level design decisions.

The main goal of our work is to generate the communication links between predefined blocks with minimal user inputs. In this paper, we present our solution which makes use of standards to enable transparency as well as the early validation of designs and the subsequent verification of resultant systems. Fig. 1 shows the design flow of our approach. We built UML profiles to capture the system level communication interfaces. The solution is laid out using UML structural diagrams. State diagrams are added to customize functional behaviors of the interfaces. We built a software tool to automatically generate the interface and glue logic to connect the devices while meeting bandwidth and performance requirements. We experimented with our implementation under different scenarios including the "plug-and-play" of OCP-compliant, Verilog and PCI-compliant components into a SystemC simulation environment. The

automatic generation of interfaces and glue code leads to the fast prototyping of possible design solutions. These can then be tested and optimized.

II Problem Description

IP blocks can be viewed as black boxes that communicate with each other through pre-defined interfaces. An *interface* contains several *system level data ports* (SLport) that can send or receive certain type of data. Consider a scenario where an IP block is connected to an existing interface of a system. Let's assume that the IP block has only one exterior interface. A failure to perform the connection may be due to one of two possible reasons: (1) the interface protocol does not matched, or (2) the SLports of one interface are not sufficient to drive the other. We say that an IP core is *compatible* to an environment if all the output ports of each interface are able to drive the corresponding input port of the other interface while satisfying all timing constraints. For two compatible interfaces that have different protocols, they form one of two types of compatible pair:

Type 1 compatible pair: There is a one-to-one correspondence between the SLports of the two interfaces.

Type 2 compatible pair: The output SLport of one interface can be made to drive the input SLport of the other through some runtime transformation of its data.

For a type 1 compatible pair, a port-to-port mapping is specified through a *contract* associated with the connection. For a type 2 compatible pair, glue logic has to be introduced. We will synthesis both types of the connection from their descriptions specified in *wrapper classes*.

III User Input

Many specification formalisms today use graphical notations. One of the most successful of such formalisms is the Unifined Modeling Language (UML) from OMG.[14] It has been proven very successful and is widely used in software designs. In our framework, we chose UML to be our specification language for its user friendliness and wide adoption. UML provides a large set of notations. We carefully chose a subset for modeling the interface communications.

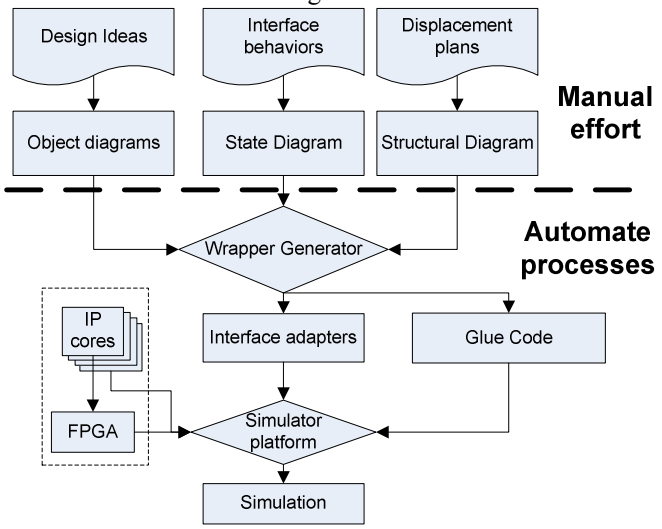

Fig.1: Design flow of our UML based approach

UML structural diagrams are predominantly used to describe the component structure of a system. IP blocks and their wrappers are treated as black-boxes and modeled as UML classes. We begin with modeling the communication interfaces of existing IP cores. Wrapper modules with interfaces adapters will then be modeled. After the classes are drawn, interfaces are added as UML ports to capture interface information. We shall use 'UMLport' to distinguish these ports from SLports. Communication channels are modeled as connections between the UML ports. To fix the inconsistency between unmatched interfaces, state charts are added to a wrapper's model to describe the behavior of the glue logic needed to drive the interface. The modeling procedure can be divided into following three steps:

Step 1: Formalize the IP interfaces using UML notations

Each IP core is modeled using a UML class attached with several UMLports. A (UML) port is one of the real-time system elements introduced in UML 2.0. It is a property of classifiers that specifies a distinct interaction points between the classifier and its environment or between the classifiers. These UMLports will capture the module's communication interfaces. Each UMLport models a group of one or more SLports. The associated SLports will be described in the properties of the UMLport.

All the details of an IP's interfaces are captured in the properties and contracts of its wrapper's UMLports. A contract specifies the services that a classifier provides (offers) to its environment as well as the services that a classifier expects (requires) of its environment [14]. If two UMLports have a port-to-port match in the communication path then they will share the same interface contract. A connection will be established between them directly or indirectly (through other ports). The stereotype of an UMLport specifies the communication protocol. In our experiments, the default stereotype for UMLport is SystemC. However, we also used other stereotypes such as OCPSystemC, Verilog, and PCI. The communication parameters of interfaces are captured in the UMLport's attributes. For IPs with interfaces matched to each other, port-to-port communication paths are specified directly in their specification, and only external UMLports to the environment are used for generating the wrappers later on.

Step 2: Define the wrapper classes

To mask the interfaces, the internal UMLports of IP models are connected to the user defined UMLport in the outer class, i.e., a *wrapper class*. The diagrams capture two main aspects of the models: the structure of the model and the characteristics of the interfaces.

The links and blocks in the structure diagram lays out the structure of the model. The wrapped classes communicate with each other using defined channels, and the connections between the internal components are hidden. The adapters are defined using the exterior UMLports of the wrapper classes. We shall call these UMLports *adapter port*s. Each adapter port has a stereotype that indicates the protocol of the outgoing communication. Each wrapper class can hold one or more adapter ports, while each adapter port can have different protocol types and connects to different external modules. Table 1 summarizes the mapping between UML notations and the components of our model.

Fig. 2 shows an example of a model after wrapping. The arbiter has 2 clients — one non-blocking and one blocking

master. `master_nb` is the UML model of a SystemC module. The UMLports (say `read`) of this UML module represent groups of SLports, of the underlying SystemC module. The interface of `simple_bus` includes the interface to the arbiter as well as to the memory, because they have no compatibility problem, we will not model the details of these interfaces. In this example, the simple bus' SLports (and thus the encapsulating UMLports) are implemented in an OCP protocol, while the clients are not. Therefore, two wrapper classes were used to customize the interfaces of the clients. The clients will communicate with `simple_bus` through the OCP adapter ports.

Step 3: Define the behavior for incompatible interfaces

Connections are made between adapter ports and the environment. For connections without port-to-port matchings, additional models have to be added to capture the logical relations between input and output SLports. As an example of a connection that requires glue logic, we consider the IDCT filter used in our MPEG-2 case study (see adapter port ① of Fig. 6). To implement the 8X8 IDCT filter, some IP blocks takes an address of two dimensional 8X8 array, while others takes 64 inputs in a linear array. To solve this problem, we introduce an *interface state machine* to transform the signals. Our framework allows the user to customize the predefined interface behaviors using such state charts.

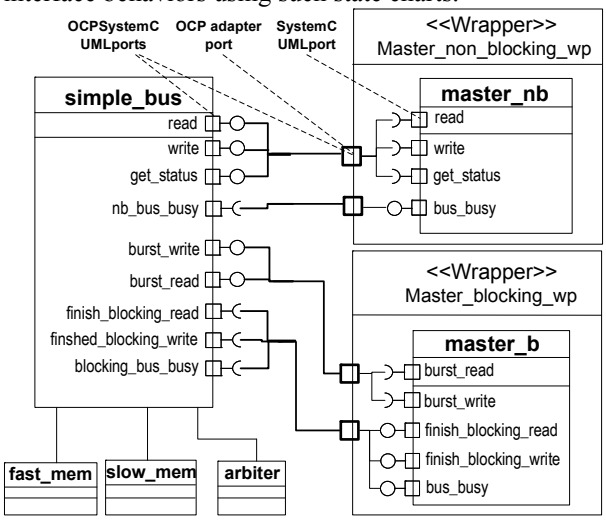

Fig. 2: Structural diagram of Simple-bus example

Table 1: Mapping between UML notations and design properties

UML Notations	**System Properties**
Parent Class	Wrapper module
Name	Name
UMLport	Interface adapter
Subclasses	IP cores
UMLPort of subclass	Interface
Port name	Name
port type	Type
Interface type	Direction
Stereotype	Protocol
Port properties	Driving signals
State chart	Driving behaviors
Contract Attributes	Signals
Name	Name
Stereotype	type
Tag	Width
UMLport State Chart	Adapter's control code
States and transitions	Finite state machine

Fig. 5 shows the state diagram of the IDCT example, where the master interface has three SLports: `Fast_IDCT_signal`, `Fast_IDCT_addr`, `Fast_IDCT_ack`, and slave interface has 4 SLports: `start`, `datain`, `dataout` and `done`. The diagram defines how the transformation needed for the output SLports to drive the input SLports across the connection.

IV Interface Synthesis

A key feature of our framework is the automatic code generation. In our experiments, we use Telelogic's Rhapsody 6[16] to input the UML diagrams. The model is then feed into our code generator. The analyzer filters out the interfaces and produces abstract models of the design containing the communication specifications. The adapter code is generated from templates using Velocity [18], a template engine that generates code from predefined templates. The Velocity engine merges the code templates and the extracted wrapper models, and performs the final code generation. Fig. 3 shows the work flow of our framework. The code generator reads the model and produces the interface synthesis code in 4 steps:

In the first step, models that have the stereotype `Wrapper` are extracted and the corresponding interface adapter code will be generated. For example, to integrate a module in the SystemC environment, each adapter will be generated as a `SC_MODULE`, a primitive type of SystemC. On the other hand, to adapt a FPGA PCI module, an adapter file will be generated which contains the routines for checking and opening communication channels, as well as the communication routines for communicating with the board. We assume that all the modules without stereotype are normal modules, and they will not be generated in the compilation process. The interfaces extracted from the models are also stored for future references.

In the next step, the code generator will extract and analyze the connections from the model. According to the stereotypes of the UMLports connected to each adapter port,

the connections are classified into groups. Currently, we have generators for OCP-SystemC, OCP-Verilog, and OCP-PCI connections. After retrieving the type of each connection, we can now determine if a pair of connections is of Type 1 or Type 2. If a pair of interfaces connecting to the UMLport shares the same contract but have different protocol types, then they are classified as Type 1 incompatible pair. If the connections do not match up with each other but state diagrams were defined to fix the incompatibility, then they are classified as Type 2.

For Type 1 pairs, we build *adapters* (corresponding to the adapter ports) that forward the output source signals to the destination SLports through the desired protocol channel. We shall now explain how this is done for OCP compliant adapters.

An OCP compliant adapter is built to fit the OCP bus fabrics. There are three levels of SystemC OCP communication models: generic, OCP TL1, and OCP TL2. We chose to use OCP TL2 as our wrapper communication model. The advantage of OCP TL2 is that it allows for burst transfers and the communication is faster and more reliable.

When an adapter gets a message from an OCP channel via an OCP port, it first decodes the message, and then forwards the message to the corresponding interface using a SystemC signal. After the internal SystemC component has completed its operation, a response message will be created and placed in the OCP channel. Correspondingly, when a wrapper gets signaled by an internal component that wants to communicate with the environment, the wrapper will create a message labeled with identifier number of the interface, and puts it on the OCP channel. After the response is received, it is decoded and a reply signal will be forwarded to the component in waiting.

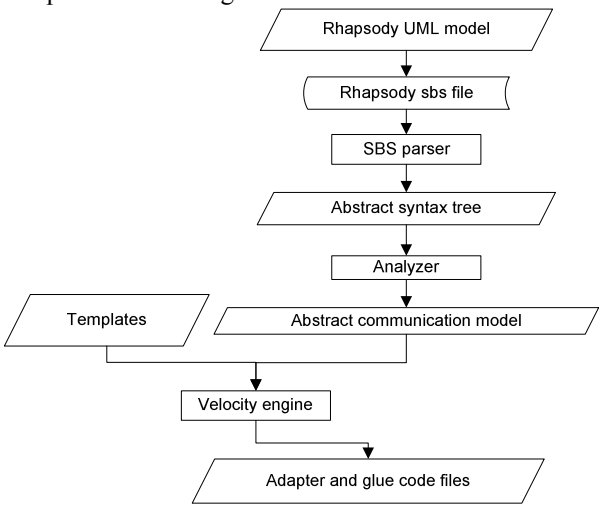

Fig. 3: Work flow of wrapper generator

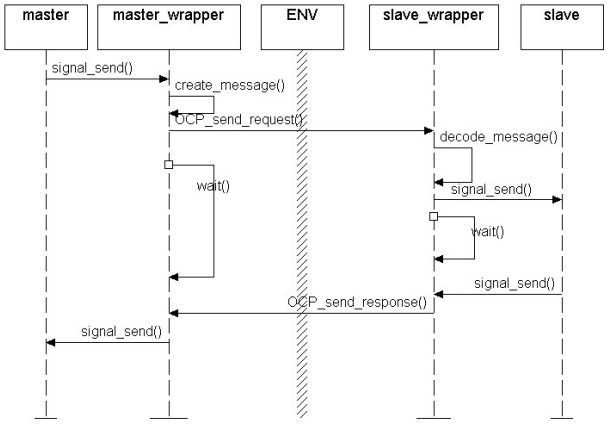

Figure 4: Sequence diagram of OCP communication groups

OCP communication is established between the OCP master and slave port, which are predefined OCP interfaces. Each communication path starts with one OCP master port and ends with one OCP slave port. A 'provided' interface will be connected to OCP slave port, while a 'required' interface will be connected to an OCP master port. Each OCP port has three threads to control its actions. We call these a *communication group*. Communication groups are paired together and each pair controls an OCP channel. The channel is configured by reading the information extracted from the UML model.

Fig. 4 shows the transactions of a communication group, where the master is a wrapped component of the master side object, and the slave is a wrapped component of slave side object. They communicate with each other using the services provided by their wrappers, i.e., the communication is controlled by `master_wrapper` and `slave_wrapper` respectively.

Adapters for SystemC-Verilog, and SystemC-PCI are generated in a similar way. A SystemC-Verilog adapter is a SystemC header file that describes the SLport connection of the Verilog module. A SystemC-PCI adapter not only interprets incoming data but also needs to perform the access routines in accordance to the protocol. The wrapping are generic, therefore, multiple levels of wrapping is possible.

For a Type 2 pair, the state diagram showing how the outputs drive the inputs of the client program. A state machine is then generated. Code 1 shows the pseudo code for the state machine that drives IDCT interface using the `Fast_IDCT` interfaces.

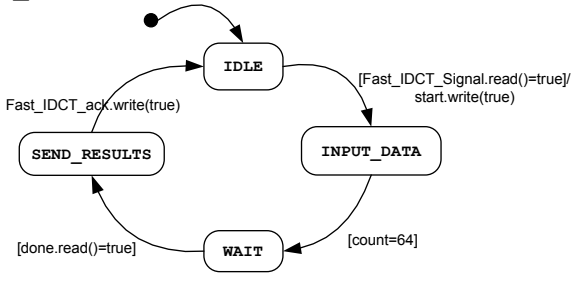

Figure 5: State diagrams of glue logic between IDCT master and slave interfaces at ① of Figure 6.

```
while(true){
switch (state){
 case IDLE: //in idle state
  wait_until(Fast_IDCT_signal.read()=true);
//state is guarded by Fast_IDCT_signal
   start.write(true); //drive driver port
   state=INPUT_DATA; //change state
 case  INPUT_DATA:
   addr=Fast_IDCT_addr.read();
   for i from 1 to 8, j from 1 to 8
     din.write(addr[j*8+i];
   state=WAIT;
 case WAIT:
   wait_until(done.read()=true);
   state=SEND_RESULTS;
 case SEND_RESULTS:
   for i from 1 to 8, j from 1 to 8
     addr[j*8+i]=dout;
   Fast_IDCT_ack.write(true);
   state=IDLE;
 }}
```

Code 1: Pseudo code of glue logic generated from IDCT of Fig. 5

V Experiments and Results

5.1 Simple-bus

`Simple-bus` is an open source SystemC example. The system consists of a bus kernel, a bus arbiter, two memory slaves (`fast_mem`, and `slow_mem`), and two masters (`master_nb` and master_b). We wrapped the `master_b` and `master_nb` up and substituted the communication channels between the bus and the masters with OCP channels. Fig. 2 shows the structural diagram of the `Simple-bus` example after wrapping. We then modeled the exterior SLports of the bus kernel and the masters, and wrapped them up. The wrapper code is generated and assembled for the simulation. The unwrapped version of the simple bus SystemC code had 2,684 lines while the wrapped version had 3,743 lines of code. We measured the performance of the unwrapped and wrapped code on a Linux machine with a dual core AMD Opteron 280 CPU running at 2.4GHz. The unwrapped simple bus system finished a million bus transactions in 41.978 seconds, while the wrapped system took 52.188 seconds. Thus, the overall overhead caused by wrapping is 24.3%.

5.2 MPEG-2 Decoder

MPEG is an encoding and compression system for digital multimedia content defined by the Motion Pictures Expert Group (MPEG) [11]. MPEG-2 extends the basic MPEG system to provide compression support for TV quality transmission of digital video. We used an open source MPEG-2 decoder originally written in C. We analyzed the functionality and structure of the system, and made it SystemC-compliant.

To test the performance of our adapter code, we customized a SystemC version of MPEG-2 decoder and divided it into five groups. Five wrapper classes were created to wrap up the components. They communicate with each other using OCP channels which are highlighted using bold lines. Each wrapper consists of several OCP ports, and they are connected to other OCP ports as well as to interfaces of wrapped components. The MPEG-2 decoder is partitioned into five communication groups. The SystemC code is generated and wrapped components are reconnected for simulation using the SystemC simulator.

We experimented with the plug-and-play of the IDCT unit. IDCT (Inverse Discrete Cosine Transform) [6] is a key component in the MPEG-2 decoder system. In our experiments, four IDCT modules were used: F-IDCT and R-IDCT, Verilog-IDCT, and PCI-IDCT. F-IDCT implemented an integer IDCT while R-IDCT used floating point operations. The latter ran at a lower speed but has higher accuracy. They shared the same interface signature but have different interface names. Verilog-IDCT and PCI-IDCT are Verilog version and FPGA version of IDCT respectively. We modeled the IDCT blocks and their interfaces in UML and wrapped them up. `Fast_IDCT` and R-IDCT is a Type 1 pair connection, while Verilog and PCI-IDCT were used to demonstrate a Type 2 pair connection. The wrappers were generated and plugged into the decoder system. The simulation ran correctly even after switching the IDCT module. The rest of design remains unchanged.

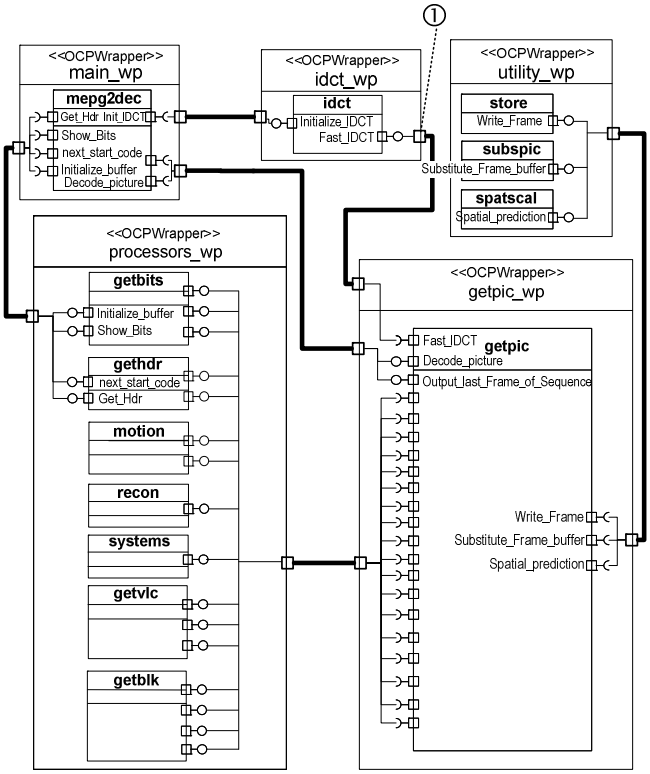

Figure 6: Structural diagram of MPEG-2 decoder after wrapping

Table 2 Simulation results of decoders with F-IDCT

Input	Unwrapped w/F-IDCT	Wrapped w/F-IDCT	Extra Overhead
short.m2v	0.412s	0.450s	9.22%
fball.m2v	154.886s	172.415s	11.32%
zoo.m2v	629.155s	730.5275s	16.11%
dhl.m2v	801.406s	932.31s	16.33%

Table 3 Simulation results of decoders with R-IDCT

Input	Unwrapped w/R-IDCT	Wrapped w/R-IDCT	Extra Overhead
short.m2v	0.422s	0.444s	5.21%
fball.m2v	162.84s	184.004s	13.00%
zoo.m2v	699.8335s	789.0175s	12.74%
dhl.m2v	910.759s	1024.9585s	12.54%

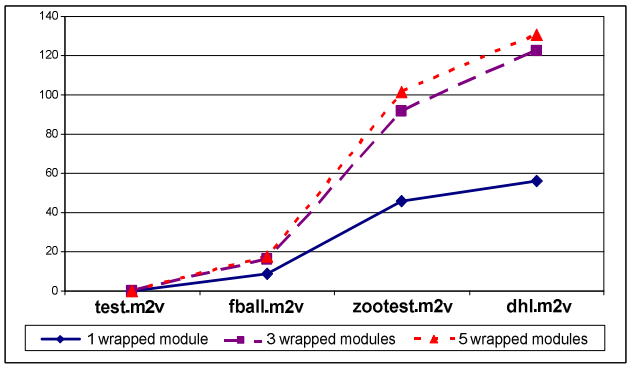

Fig 7: Overhead with different number of wrappers

Because IDCT is the most frequently used component in the system, and has a significant impact on overall performance, the overhead of wrapping has to be measured against unwrapped version of the respective integer or floating point IDCT. Table 2 and 3 show the simulation results of MPEG-2 decoder with five wrappers using different version of IDCT processors. From the tables, we can see that the wrapped decoder takes about 9%-16% more simulation time than the unwrapped decoder in both cases.

For Verilog-IDCT and PCI-IDCT, we generated the adapters and then compiled and simulated the entire system using ModelSim. These IDCT shares similar exterior interfaces, and the same UML model can be used for the two cases while maked with different stereotypes. The results of these versions of the MPEG-2 decoder as well as those simulated in the SystemC simulator were identical to the correct outputs of the original program. The simulation is, however, slow due to the overhead of co-simulating in both SystemC and ModelSim.

The test result also shows that the overhead of wrapping is not proportional to the number of wrappers (Fig. 7). Instead, it is proportional to the number of transactions passing through the wrappers. Wrapping components with lower workload will have a lesser impact on overall performance. There is therefore a trade off between configurability and performance. The case studies also show that wrapping can easily be applied to any number of components or groups of components. With the support of our synthesis framework, we can plug different IP implementations into a system, and automatically generate the code needed for system-level simulation.

VI Conclusions and Future Works

In this paper we present a framework for the integration of heterogeneous and incompatible predefined IP cores. It involves using UML to specify high-level communication models and the automatic generation of glue logics. Two types incompatible interface are supported: interfaces that are port-to-port matched but use different protocols, and interfaces that require more complex logic to fix the incompatibility. Structural diagrams are used to layout the system, while additional state diagrams are used to model the interactions between interfaces. We tested our algorithms under several environments using SystemC, Verilog, and FPGA modules in a SystemC-OCP environment. The resultant glue logics can be used to test the designs using simulation. Our experiments show that the performance overhead of our wrappers is acceptable.

Our algorithm can be extended easily to cross platform designs. For systems that operate in different environments, bridges can be modeled and generated using our framework. Currently, the Velocity templates are hand-written. As a future work, we are exploring the use of behavioral diagrams to automatically generate them.

REFERENCES

[1] Borriello G., Chou P., Ortega R, "Embedded System Co-Design: Towards Portability and Rapid Integration, Hardware/Software Co-design", NATO ASI Series, pp. 1-28, 1996

[2] Chandra, R., “IP-Reuse and platform base designs, system level design with embedded platforms”. DAC Tutorial, 2000

[3] Chou, P., Ortega, G., Borriello G., “Interface co-synthesis techniques for embedded systems”, ICCAD, San Jose, USA, 1995.

[4] Chung, K.S., Gupta, R.K., Liu, C.L., “An algorithm for synthesis of system-level interface circuits”, Computer-Aided Design, 1996. ICCAD-96. Digest of Technical Papers. 1996 IEEE/ACM International Conference on. 10-14 Nov 1996 pp. 442 - 447

[5] Damaševičius, R., and Štuikys, V., “Soft IP customization models based on high-level abstractions”, Information Technology and Control, Kaunas, Technologija, Vol. 34, No. 2, pp.125 – 134. 2005

[6] Feig, E., and Linzer, E., “Discrete cosine transform algorithms for image data compression”, Proceedings of Electronic Imaging '90 East, pp. 84-87, 1990.

[7] Guo, Z., Mitra, A., and Najjar, W., “Automation of IP core interface generation for reconfigurable computing”, Int. Conference on Field Programmable Logic and Applications (FPL 2006), Madrid, Spain, August, 2006.

[8] Keating, M., and Bricaud, P., “Reuse methodology manual from system-on-chip designs”, 3rd edition, Kluwer Academic publishers, 2002.

[9] Kun, T., Wang, H., and Bian, J.N., “A generic interface modeling approach for SOC design”, ICSICT’04, pp. 1400-1403, Beijing, 2004.

[10] Lin B., Vercauteren S., “Synthesis of Concurrent System Interface Modules with Automatic Protocol Conversion Generation”, ICCAD, pp. 101-108, November 1994.

[11] Moving Picture Experts Group. http://www.mpeg.org.

[12] Mukherjee, R., Jones, A., and Banerjee, P., “System level synthesis of multiple IP blocks in the behavioral synthesis tool”, Int. Conf. on Parallel and Distributed Computing and Systems (PDCS), November 2003.

[13] OCP-IP, http://www.ocp-ip.org.

[14] OMG UML documentation, http://www.uml.org

[15] Štuikys, V., and Damaševičius. R., “Soft IP customization model based on metaprogramming techniques”, Informatica, Lith. Acad. Sci., pp. 111-126, 2004.

[16] Telelogic Rhapsody, http://www.telelogic.com/

[17] SystemC: http://www.systemc.org

[18] Velocity website. http://velocity.apache.org/

[19] Zhu, J., “MetaRTL: Raising the abstraction level of RTL design”, Design Automation and Test in Europe, Munich, Germany, 2001