Lecture 5
Refinement of Computation and Communication

Multimedia Architecture and Processing Laboratory

Prof. Wen-Hsiao Peng (彭文孝)
pawn@mail.si2lab.org
2007 Spring Term
Acknowledgements

This lecture note is partly contributed by Prof. Gwo Giun Lee (李國君) in the Dept. of EE, National Cheng-Kung University and his team members 王明俊, 林和源 in the research laboratory 多媒體系統晶片實驗室
E-mail: clee@mail.ncku.edu.tw
Tel: +886-6-275-7575 ext. 62448
Web: http://140.116.216.53
References

Communication Refinement

- Expand an abstract communication into an actual implementation
  - Example: transition from “sc_fifo” to “hw_fifo”
  - Implementation can be either software or hardware
    - Hardware: a shift register or a RAM with some control logic
    - Software: a certain memory range with a modulo addressing scheme
- No simple one-size-fits-all approach
- Key principles
  - Use well-defined and well-understood interfaces
  - Intelligent use and reuse of adapters and converters
Example of Communication Refinement

> Component-Assembly Model

> Bus Arbitration Model

> Bus Functional Model

Communication should be firstly refined!

Transaction level model at a high level of abstraction

Masters

Slaves

Cycle-accurate transaction level model of simple bus
**Steps in Communication Refinement**

1. Select an appropriate communication scheme
2. Implement the selected scheme $C_{\text{refined}}$
3. Use $C_{\text{refined}}$ to replace the abstract channel $C$
4. Enable the communication of M1 and M2 over $C_{\text{refined}}$
   a. Wrap $C_{\text{refined}}$ to match the interface of M1 and M2
   b. Refine M1 and M2 to match the interface of $C_{\text{refined}}$

(A1 and A2 are adapters that translate one interface into another)

![Diagram showing the communication process]

- **Wrapper-based**
- **Adapter-merging**
Adapters – Channel Refinement

- A hierarchical channel that translates the interface it implements into port accesses bound to a different interface
  - Example: the read()/write() methods in the hw_fifo_wrapper can be encapsulated in an adapter, respectively
- Adapters can be easily reused and a small library of adapters is sufficient
  - Typically only a finite number of different interfaces are used
- Two things that can be done with an adapter for refinement
  - Introduce an additional level of hierarchy
    - Wrap the refined channel and the adapters connected to it
  - Merge the adapters into the calling modules
    - Move ports, processes, methods, and data fields to the calling processes
    - Replace calls to the adapter’s methods with the new member functions
Adapters (c. 1)

- Similarity in the real world

Interfaces of Refined Channels

Electricity

Channel Functionality

Ports of Module 1

Ports of Adapters

3-to-2 Adapters

Interfaces implemented in an abstract channel
Converters - Module Refinement

- A module that translates a set of ports to another in order to make the refined module and communication channel fit together
  - The refined model could be a piece of an existing IP and have a different set of ports than its abstract counterpart
- Things that can be done with an converter for refinement
  - Wrap a refined module and a converter as a new module that matches the channel’s footprint
  - Merge an adapter to form a new, refined converter
    - A converter may accesses an adapter via the adapter’s interface
- Converters versus Adapters
  - Converters tend to be a component in the final implementation
  - Adapters are often temporal objects
Converters (c. 1)

- Similarity in the real world
**General Process of Refinement**

1. **Select and Replace**
   - Select a refined implementation for a component
   - Replace the original component with the refined implementation

2. **Insert adapters or converters (optional)**
   - Make channel and module play together
   - Adapters and converters should be stored in an accessible library

3. **Analyze the I/O functionality and performance**
   - Verify the functionality and system performance

4. **Flatten (optional)**
   - Ungroup the hierarchy of the wrapped channel
   - Merge adapters and connected modules
**General Process of Refinement (c. 1)**

5. **Wrap (optional)**
   - Wrap the refined channel and the set of connected adapters to create a new channel for software implementation

6. **Merge (optional)**
   - Refine an abstract communication protocol (transaction-level) towards a hardware implementation (pin-level)
   - Merge adapters into the calling modules (protocol inline)

7. **Restructure control flow (optional)**
   - High level module is generally described in a sequential manner using blocking read()/write()
     - `first_output->write(some_data)`
     - `second_output->write(some_data)`
General Process of Refinement (c. 2)

7. Restructure control flow (cont’d)
   - Sequential execution might not be suited for hardware
   - One may want to refine control flow to exploit more parallelism
     - first_output->initiate_transmission(some_data)
     - second_output->initiate_transmission(some_data)
     - wait_for_completion(first_port, second_port)

8. Analyze Implementation
   - Check both functionality and I/O performance
   - Analyze the partial or entire system for its implementation characteristics such as area, latency, clock frequency, and power dissipation
Hardware-Hardware Communication Refinement
(From Abstract Model to RTL)
Two functional modules communicate via an untimed abstract FIFO channel `sc_fifo`.

Replace `sc_fifo` with a time hierarchical channel `HW_FIFO` and insert adapters.

After refining all components, adapters A1 is merged into `HW_SOURCE` and A2 into `HW_SINK`, and a converter is added to reverse the polarity of two `HW_SINK` signals.
Implementation of Adapters

```
template<class T> class FIFO_WRITE_HS : public sc_module : public sc_fifo_out_if<T>
{
public:
    sc_in_clk clk;
    sc_out<T> data;
    sc_out<bool> valid;
    sc_in<bool> ready;

    //blocking write
    void write(const T& x){
        data = x;     //drive data line
        valid = true; //signal the data is valid
        do{             //wait until data was read
            wait(clock->posedge_event());
        }while(ready.read()!=true);
        valid = false;
    }

    bool nb_write(const T&x){
        assert(0); return false;
    }
};
```

```
SC_MODULE(SOURCE)
{
    //output port
    sc_fifo_out<int> out;

    //The one and only process
    void my_process()
    {
        int counter = 0;
        while(1){
            counter++;
            out->write(counter);
        }
    }

    SC_CTOR(SOURCE)
    {
        SC_THREAD(my_process)
    }

    //Provide dummy implementations for unneeded sc_fifo_out<T> methods
    bool nb_write(const T&x)
    {
        assert(0); return false;
    }
};
```
**Merging Adapters**

- **Goal**
  - To merge the adapters into the calling module for “protocol inline”
  - The result is a refined pin-level module

- **Procedures**
  - Copy and paste the adapter’s properties into the calling module
    - Ports, methods, data fields, and processes
    - Care must be taken to avoid name clash
  - Remove the original port that was used to access the adapter
  - Replace the template argument T with the actual data type
  - Create methods and constructors of the calling module
    - The adapter’s methods become new methods of the calling module
    - The adapter’s constructor must be merged into the calling module’s constructor
    - If the adapter has additional constructor arguments, these can usually be replaced with constant expressions
  - Replace references to the adapter’s clock with accesses to the calling module’s clock
Merging Adapters (c. 1)

Adapter

```cpp
template<class T> class FIFO_WRITE_HS
public sc_module
: public sc_fifo_out_if<T>
{
public:
  sc_in_clk clk;
  sc_out<T> data;
  sc_out<bool> valid;
  sc_in<bool> ready;
  //blocking write
  void write(const T& x){
    data = x; //drive data line
    valid = true; //signal the data is valid
    do{
      wait(clock->posedge_event());
    }while(ready.read()!=true);
    valid = false;
  }
  //Provide dummy implementations for
  //unneeded sc_fifo_out<T> methods
  bool nb_write(const T& x){
    {assert(0); return false;}
  }
};
```
Implementation of Converters

- Assume we select an existing IP HW_SINK for the SINK module
  - The IP has active low on the pins valid and ready
  - A converter is used to reverse the polarity of the “valid” and “ready”
Control Flow Restructuring

Merging Adapters

SC_MODULE(SOURCE)
{
    //output port
    sc_fifo_out<int> first_output;
    sc_fifo_out<int> second_output;

    //The one and only process
    void my_process()
    {
        int counter = 0;
        while(1)
        {
            counter++;
            first_output->write(counter);
            second_output->write(counter/2);
        }
    }

    SC_CTOR(SOURCE)
    {
        SC_THREAD(my_process)
        {
            SC_THREAD(my_process)
        }
    }
}

SC_MODULE(HW_SOURCE)
{
    //output port
    ..................
    ..................

    //The one and only process
    void my_process()
    {
        int counter = 0;
        while(1)
        {
            counter++;
            first_output_write(counter);
            second_output_write(counter/2);
        }
    }

    SC_CTOR(HW_SOURCE)
    {
        SC_THREAD(my_process)
        {
            SC_THREAD(my_process)
        }
    }
}

//Blocking write
void HW_SOURCE::first_output_write(const T& x)
{
    first_output_data = x;
    first_output_valid = true;
    do{
        //wait until data was read
        wait(clock->posedge_event());
    }while(first_output_ready.read()!=true);
    first_output_valid = false;
}

void HW_SOURCE::second_output_write(const T& x)
{
    second_output_data = x;
    second_output_valid = true;
    do{
        //wait until data was read
        wait(clock->posedge_event());
    }while(second_output_ready.read()!=true);
    second_output_valid = false;
}

Data is written to the second port only after the writing to the first port is succeeded

Sequential execution using blocking write
Control Flow Restructuring (c. 1)

- One should use *fork* and *join* constructs to model parallel implementation for design space exploration.
- Restructure the control flow if the results look encouraging.

```cpp
void HW_SOURCE::my_process(){
    int counter = 0;
    while(1){
        counter++;
        first_output_data = counter;
        first_output_valid = true;
        second_output_data = counter/2;
        second_output_valid = true;
        do{
            wait(clock->posedge_event());
            if(first_output_ready.read()==true)
                first_output_valid = false;
            if(second_output_ready.read()==true)
                second_output_valid = false;
        }while(!first_output_ready.read() &&
                second_output_valid.read());
    }
}
```

Non-blocking read to primitive channels

Parallel execution using non-blocking methods
Software-Software Communication Refinement
(From SystemC to Plain C)
**Software-Software Communication Refinement**

- Refine SystemC models towards software implementations
  - Map the SystemC model into software implementation
  - Convert the SystemC design into C++ or even C codes
  - Get rid of SystemC elements
- OS support is required for mapping the SystemC processes into concurrent threads
- Refinement process is architecture- and OS-dependent
  - Primitive elements of software-based communication
    - Memory and a set of access methods
    - Inter-process synchronization primitives offered by the OS
- Support for modeling software running on the RTOS will be improved in future versions of SystemC
Mapping SystemC Model to Software Implementation

- The goal is to use the same code files for SystemC simulation and cross-compilation
  - (SC_METHOD)→(SC_THREAD)→(OS thread)
  - (SystemC data-types and modules)→(C data types or struct)
  - (Ports)→(Pointers)

- Procedure (C-based Wrapper)
  - Step 1: transform a module containing ports and data fields into a C struct
    - Ports are replaced by pointers
  - Step 2: translate the interface method calls into a set of C function calls
    - Map SystemC channels into communication mechanisms on the target architecture
  - Step 3: turn the SC_THREAD processes into OS threads
    - Method processes can be firstly transformed into thread processes
  - Step 4: replace the SystemC data types with built-in C data types
    - Those data types not supported on target architecture should be replaced
Example: C-based Wrapper

C-based wrapper that encapsulates communication primitives on the target OS.
/* sc2sw.h */
/* include systemc.h only if SYSTEMC is defined */
#ifdef SYSTEMC
#include "systemc.h"
#else
#define SC_MODULE(X) struct X
#endif
#ifdef SYSTEMC
#define IMC (port, type, method) port->method()
#define IMC1 (port, type, method, arg1) port->method(arg1)
#define IMC2(port, type, method , arg1 , arg2) port->method(arg1, arg2)
#else
#define CONCAT (a,b,c) a##b##c
#define IMC (port, type, method) CONCAT(type,_,method(port))
#define IMC1 (port, type, method, arg1) CONCAT(type,_,method(port, arg1))
#define IMC2(port, type, method , arg1 , arg2) CONCAT(type,_,method(port, arg1, arg2))
#endif

/* SOURCE.c */
#include "SOURCE.h"

void* SOURCE_my_process (struct SOURCE* m)
{
  int counter = 0;
  while(1) {
    counter++;
    /* port access using an IMC macro */
    IMC1(m->output, fifo, write, counter);
  }
  return 0;
};
Example: C-based Wrapper (c. 2)

```c
#define REENTRANT
#include “pthread.h”
#include “SOURCE.h”
#include “SINK.h”

int main(int argc, char** argv)
{
    struct fifo f;
    SOURCE source; /*A C struct*/
    SINK sink;     /*A C struct*/
    pthread_t source_thread, sink_thread;

    /* Initialize FIFO*/
    fifo_create(&f);

    /* Connect FIFO and modules*/
    source.output = &f;
    sink.input = &f;

    /* Create Threads*/
    Pthread_create(&source_thread, NULL, (void (*)(void *))&SOURCE_my_process);
    Pthread_create(&sink_thread, NULL, (void (*)(void *))&SINK_my_process);

    /* Wait until all threads have terminated */
    pthread_join(source_thread, NULL);
    pthread_join(sink_thread, NULL);
}```
Remarks

◆ C-based wrapper approach
  ❖ Duplicate the functionality of SystemC threads and channels within the C language running on top of the target OS

◆ Alternative approach
  ❖ Create SystemC channels with an interface matching the equivalent C function calls of the target RTOS communication primitives
    ✤ RTOS functionality at system level is in the form of SystemC channels

◆ Refinement process
  ❖ C-based wrapper
    ✤ Favor deferring the refinement until the implementation phase
  ❖ Alternative approach
    ✤ Similar to hardware-hardware refinement, all communication schemes must be expressed in terms of the refined SystemC channels
Other Tasks for Refinement

- Data type refinement
  - Example: refine a description using floating-point arithmetic to a fixed-point representation suited for cross-compilation

- Static scheduling
  - Reduce the context-switch by finding a pseudo-static execution order for a set of SystemC processes and merging them into a thread
Hardware-Software Communication

Refinement

> Hardware Part: From Abstract Model to RTL
> Software Part: From SystemC to Plain C
Hardware-Software Communication Refinement

SW/ HW functionalities described in SystemC

Masters models could be refined into embedded SW (plain C) or HW (RTL)

Cycle-accurate transaction level model of simple bus

Transaction level model at a high level of abstraction

Slaves models could be refined into embedded SW (plain C) or HW (RTL)

Component-Assembly Model

Bus Arbitration Model

Bus Functional Model

Sw/Hw functionalities described in SystemC

Masters models could be refined into embedded SW (plain C) or HW (RTL)

Cycle-accurate transaction level model of simple bus

Transaction level model at a high level of abstraction

Slaves models could be refined into embedded SW (plain C) or HW (RTL)
Hardware-Software Communication Refinement (c. 1)

- Untimed TLM typically use dedicated channels for communication

- Shared communication appears in timed TLM for architecture mapping
  - The model used to analyze the effects of shared I/O is still abstract
  - Latency annotation normally goes hand in hand with architecture exploration

FIFO size of each module is one of the parameters to be determined

FIFO models the input/output buffers for interfacing with the shared communication
**Implementation of XBAR**

- Communication refinement: XBAR

```cpp
Template <class T> SC_MODULE(XBAR) {
    // ports
    sc_port<sc_fifo_in_if<T>, 0> inputs;
    sc_port<sc_fifo_out_if<T>, 0> outputs;
    sc_in<bool> clk;

    // data
    unsigned last_; // index of last channel granted

    // constructor
    SC_CTOR(XBAR) : last_(-1) {
        process;
    }

    void process() {
        // same number of channels required
        assert(inputs.size() == output.size());
        // Iterate over all ports starting with the one that
        // comes after the one that was granted access last time.
        // Transmit data if both data available at the input and space
        // available at the corresponding output
        for(int i=1; i<=input.size(); i++) {
            int current = (last_ + 1) % input.size();
            if((inputs[current]->num_available() != 0) &&
               (outputs[current]->num_free() != 0)) {
                outputs[current]->write(inputs[current]->read());
                last_ = current;
                break;
            }
        }
    }
};
```

**Round-Robin Arbitration**
What Are to Be Explored?

- Interconnect topology between modules
  - Hub and spoke architecture
  - Cross bar switch
- Arbitration policy and prioritized accesses
- FIFO sizes associated with each module
- Resource locking
- Wait states
- Timing and performance
- ...
Platform Refinement – Hardware Part

- The platform is refined after the system has been analyzed at a higher level of abstraction
  - A continuous refinement process
  - Selection from a number of available IP blocks
    - Busses, memory sub-system, arbiters, bridges, and so on
- It is advisable not to make a giant step towards a pin-level model
  - Model the platform - possibly in a cycle-accurate way - at the transaction level
Template `<unsigned int slave_address, unsigned int unique_priority, unsigned int n_wait = 0, bool lock = false>`

```cpp
class FW2M : public sc_module, public sc_fifo_out_if<int>
{
    public:
    // ports
    sc_in_clk clock;
    sc_port<simple_bus_blocking_if> bus_port;

    // blocking write
    void write(const int &data)
    {
        simple_bus_status status;
        // check whether space is available
        for(;;)
        {
            bus_port->burst_read(unique_priority, &space_available,
                                 slave_address, 1, lock);
            if(space_available)
                break;
            for(int i=0; i<n_wait; i++)
                wait(clock->posedge_event());
        }
        // send data
        bus_port->burst_write(unique_priority, &data,
                              slave_address, 1, lock);
    }
};
```

- Adapter transforms a FIFO write into bus accesses
- The slave provides service to the write request
- The master may wait for a few cycles and try again
- Optionally, the bus can be locked
Platform Refinement – Hardware Part (c. 2)

- What are to be explored?
  - The number of buses
  - The features of the buses
  - Connection topology
  - Decision of master and slave
  - Modes of transmission
    - Send data immediately by a single read/write
    - Cache data and send it as a burst read/write
    - Initiate a split transaction (slave)
  - Burst length in case of burst read/write
  - Bus arbitration policy
  - Software/Firmware/Hardware implementation
Platform Refinement – Hardware Part (c. 3)

- **Approach I**
  - Pull in pin-accurate bus
  - Insert an adapter transforming FIFO interface into pin-level accesses

- **Approach II**
  - Stay with the transaction-level bus
  - Insert an adapter transforming FIFO interface into pin-level accesses
  - Pull in a converter that provides required pin-level interface
Platform Refinement – Software Part (c. 4)

- Refine the software down to the bus transaction level
- Make sure the software will interact properly with the hardware

```c
/* SOURCE.c */
#include "SOURCE.h"

void* SOURCE_my_process (struct SOURCE* m)
{
    int counter = 0;
    while(1) {
        counter++;
        /* port access using an IMC macro */
        IMC1(m->output, fifo_slave, write, counter);
    }
    return 0;
}

/* In systemC Simulation */
m->output->write(counter);

/* In C Simulation */
fifo_slave_write(m->output, counter);

struct fifo_slave{int *address, unsigned n_wait;};

void fifo_slave_write(struct fifo_slave*f, int data){
    int space;
    struct timespec t;
    t.tv_sec = 0; t.tv_nsec = f->n_wait;
    for(;;){
        space = *(f->address); /*read via bus*/
        if(space) break;
        nanosleep(&t, NULL); /*OS primitives*/
    }
    *(f->address)=data; /*write via bus*/
}

The process could be run in a thread on the target OS

Device driver for FIFO communication via the bus
Platform Refinement (c. 5) – More Examples

- Hardware and software co-simulation platform

Abstract Target Architecture

Global Memory Units (M)

Hardware Accelerator (AC)

Local Memory Units (M)

Intf. Unit

Command FIFOs could be needed

Shared communication

Low-level RI SC with multi-threading capability