Manifold 0.9 Component Developer's Guide

February 21, 2013

1 Introduction

Manifold is a parallel discrete event simulation framework for simulation of modern multicore computer architectures. The software package consists of a parallel simulation kernel, a collection of component models, and a few ready-to-use simulator programs. Manifold adopts a layered software architecture and separates the system into a simulation kernel layer and a models layer. In addition, Manifold is highly modularized. As a general rule, a Manifold component does not depend on any other components. Through the interface conventions, components can communicate with each other without any compile-time inter-dependence.

This developer's guide describes how a developer can create a new component for Manifold or port an existing component to Manifold.

2 Writing a Manifold Component

General requirements for a Manifold component include the following:

- A component must be a sub-class of `Manifold :: Component`.
- For each input port, an event handler function must be defined.
- For output the `Send()` function inherited from `Manifold :: Component()` should be used.
- Optionally, a component can define a pair of functions to be called at every rising or falling edge of a clock, respectively.

An example component is shown below:

```
class MyComponent : public manifold::kernel::Component
{
public:
    enum {PORT0=0, PORT1};

    void handler0(int, MyDataType0*);
    void handler1(int, MyDataType1*);

    void rising();
    void falling();
};
```

2.1 Event Handler

For each input port of you component you should define an event handler.

- An event handler is a member function of your component class.
- An event handler takes two parameters:
    - The first parameter is the port number, which should start with 0.
    - The second is the data type that you expect on the port. The data type can be a plain type or a pointer type. **It cannot be a reference.**
- An event handler returns no value.
- If the 2nd input is a pointer, it is the handler's responsibility to release the memory.

2.2 Clock Callbacks

Optionally, you can define one or two functions to be registered with a clock. Once registered, the function will be called on every rising or falling edge of the clock. You can register with either the rising or the falling edge, or both. The clock callbacks takes no parameter and returns no value, such as the `rising()` and `falling()` functions in the example above.

2.3 Component Interface Conventions

To make components independent of each other, the general convention for the interfaces are:

- An event handler should be a template function with the type of the data it expects being the template parameter.
- A component sends its own data type that implements required functions.

3 Writing a Processor Model

A processor model has only one interface: the processor-cache interface. A processor sends requests to a cache, and the cache sends responses back.

Message to cache

The processor model defines its own request data type for the processor-cache interface. At present, it is assumed that the same data type is used by the cache to send responses. However, *this may change in the near future such that the cache sends its own data type for response.*

The data type for the processor-to-cache requests must implement the following two functions:

- `get_addr()`. This functions returns a 64-bit integer that is the memory address for which the cache request is made.

- `is_read()`. This function returns true if the request is a read(load), and false if it is a write(store).

4 Writing a Cache Model

A cache model has two interfaces: the processor-cache interface and the cache-network interface.

4.1 The Processor-Cache Interface

On the process-cache interface, the processor sends requests to the cache and the cache sends responses back.

Message from processor

The processor model's request is supposed to implement the following two functions:

- `get_addr()`. This functions returns a 64-bit integer that is the memory address for which the cache request is made.
- `is_read()`. This function returns true if the request is a read(load), and false if it is a write(store).

The cache model's event handler for the processor-cache interface should be a template function similar to the following:

```
tempalte<typename T>
void my_cache_req_handler(int, T*);
```

The template type T is the type of the requests from the processor.

Message to processor

Currently it is required that the cache model sends back to the processor the same data type as what it gets from the processor. Therefore, if the processor sends type T to the cache, then the cache must respond with the same type T.

4.2 The Cache-Network Interface

If the cache model is not directly connected to the interconnection network, it can send/receive its own data type. If it is connected to the network, then it should send/receive `manifold::uarch::NetworkPacket`, which is defined as follows:

```
class NetworkPacket {
public:
    static const int MAX_SIZE = 256;

    int get_type() { return type; }
    void set_type(int t) { type = t; }
    int get_src() { return src; }
    void set_src(int s) { src = s; }
    int get_src_port() { return src_port; }
    void set_src_port(int s) { src_port = s; }
    int get_dst() { return dst; }
    void set_dst(int d) { dst = d; }
    int get_dst_port() { return dst_port; }
    void set_dst_port(int d) { dst_port = d; }

    int type;
    int src;
    int src_port;
    int dst;
    int dst_port;
    char data[MAX_SIZE];
    int data_size;
};
```

The cache model's own data should be serialized and stored in the member variable (an array) `data` of `NetworkPacket`. The simplest way to do this is just using byte-wise copy:

```
MyDataType* obj;
NetworkPacket* pkt = new NetworkPacket();
*((MyDataType*)pkt->data) = obj;
```

A cache model could send two types of messages over the network:

- Cache-to-cache messages, such as coherence messages.
- Cache-to-memory messages.

Both are carried by `NetworkPacket`.

Conversely, a cache model also receives two types of messages: from another cache or from memory. The event handler for its network input should be a templated function as follows:

```
template<typename T>
void my_net_handler(int, manifold::uarch::NetworkPacket*);
```

where the template parameter `T` is the data type from memory controller and is supposed to define the following two functions:

- `get_addr()`. This functions returns a 64-bit integer that is the memory address for which the cache request is made.
- `is_read()`. This function returns true if the request is a read(load), and false if it is a write(store).

Pseudo code for the cache model's event handler for the cache-network interface is given below:

```
template<typename T>
void my_net_handler(int, manifold::uarch::NetworkPacket* pkt)
{
    IF pkt->type == coherence message THEN
        MyCohMsg* coh =  new MyCohMsg();
        *coh = *((MyCohMsg*)(pkt->data));
        process(coh);
    ELSE IF pkt->type == memory message THEN
        T objT = *((T*)(pkt->data));
        MyMemMsg* mem = new MyMemMsg;
        Set the member values of mem with objT;
        process(mem);
    END IF
}
```

4.3 Message Types

As discussed above, a cache model has to deal with at least two message types: cache message and memory message. The model developer should not hard code the values for the message types. Instead, the value for the message types should be set in the constructor or in an access function. The specific values for the messsage types should be left to the system model builder to decide.

5 Writing a Network Model

A network sends and receives the same data type. The event handler for a network model can specify `NetworkPacket` as its input or use a tempalte function, as follows:

```
void myHandler(int, manifold::uarch::NetworkPacket*);

OR

template<typename T>
void myHandler(int, T*);
```

6 Writing a Memory Controller Model

A memory controller has one interface: the memory controller-network interface. It sends/receives `NetworkPacket` which carries the memory requests and responses.

The requests are defined in the cache model, therefore, the memory controller does not have the definition. For this reason, the event handler should be a template function as follows:

```
template<typename T>
void handler(int, manifold::uarch::NetworkPacket* pkt)
{
    T* request = (T*)(pkt->data);

    bool isRead = request->is_read();
    uint64_t addr = request->get_addr();
    ...
}
```

As can be seen, the request from cache is supposed to implement the following two functions:

- `get_addr()`. This functions returns a 64-bit integer that is the memory address for which the cache request is made.
- `is_read()`. This function returns true if the request is a read(load), and false if it is a write(store).

For response, the memory controller model can reuse the data type of the cache's requests, or it can define its own. In the latter case, the data type must also support the same two functions above.

6.1 Message Types

The responses sent by the memory controller model use a message type (the type filed of NetworkPacket) that should be different from other message types. Therefore, the memory controller developer should not hard code the message type value. Instead, the type should be set in the constructor or an access function. The system model builder is responsible for setting the types.

7 Data Serialization and De-serialization

When a component sends data to another, if the two components are assigned to different LPs, then the data must be serialized at the sending LP and de-serialized at the receiving LP. Manifold provides default serialization and de-serialization functions. In most cases, the default functions should suffice.

However, the user has to write customized serialization / de-serialization functions if the data type contains one or more of the following:

- pointers
- references
- STL containers, including strings
- an object that contains one or more of the above

Writing customized serialization / de-serialization functions essentially requires writing three template specialization functions in the manifold::kernel namespace, as shown in the following example.

```
namespace manifold {
namespace kernel {

template<>
size_t Get_serialize_size<MyDataType>(const MyDataType* data);

template<>
size_t Serialize<MyDataType>(MyDataType* data, unsigned char* buf);

template<>
MyDataType* Deserialize<MyDatatype>(unsigned char* buf);

} //namespace kernel
} //namespace manifold
```

- `Get_serialize_size()`: This function returns the size in bytes that is required for the serialized data.
- `Serialize()`: This function serialize the data into a stream of bytes and store the results in a buffer.
- `Deserialize()`: This function takes a byte stream stored in a buffer and converts the bytes into an object of the given type.

8 Open Questions

- Currently, for the processor-cache interface, the cache's response must be the same data type as the requests sent by the processor. The reason is some processor model (e.g., Zesto) carries some private data in the requests and requires the same data to be returned in the cache's response. It might be more flexible if a flag is passed to the cache model's contructor that indicates whether or not the cache's response should use the data type of the requests.