Shared Memory Framework for Hornet

Brandon Cho (mhcho@mit.edu)

Table of contents

Introduction

The shared memory framework for Hornet provides base classes to simulate coherent shared memory system on Hornet. For example, it supports a deadlock-free, cycle-level simulation of distributed directory-based cache coherence protocol, Also, other interesting types of distributed memory systems, such as Library Cache Coherence (LCC) system, NUCA-based remote access only system, Execution Migration Machine (EM^2) are currently available or will be supported soon.

These memory systems can be used with various core models of Hornet. The most lightweight example, `memtraceCore`, takes memory trace files and execute data memory operations with a configurable memory subsystem. A `pin` front end dynamically transfer execution information of applications running on a host system to Hornet; in this way, you can set up a real-world application simulation very quickly by using your host system as processing units in a target system being simulated Also, `mcpu` core can execute cross-compiled MIPS binaries to accurately simulate the behavior of core architecture

Using private-L1 shared-L2 MSI/MESI directory-based cache coherence protocol with `memtraceCore`.

1. Core configuration

`core` Like other features in Hornet simulator, core models and shared memory systems can be configured in the Hornet configuration script. In order to use memtraceCore, specify the core architecture in [core] section like the following:

```
[core]
default = memtraceCore
```

To set various parameters of `memtraceCore`, use `[core::memtraceCore]` section. The following list shows every configurable parameters and their default values.

```
[core::memtraceCore]
execution migration mode = never
message queue size = 4
migration context size in bytes = 128
maximum active threads per core = 2
```

Currently, no memory subsystem supports execution migration mode (set to `never`), but soon migration-only (`always`) and hybrid schemes (`distance` etc.) will be supported.

Every core has a message queue for each message channel (difference virtual channels deliver messages via different sets of virtual channels). The `message queue size` option specifies the size of each queue in terms of the number of messages.

`memtraceCore` supports simple multi-threading, where instructions from different threads are executed turn by turn.

2. Memory configuration

In Hornet shared memory framework, each tile has its own local L1 cache and local L2 cache. Although these local caches are directly connected to the attached core only, L1 and/or L2 caches could be connected to other cores through the Hornet on-chip network. In private-L1 and shared-L2 MSI/MESI memory, for example, both L1 and L2 (combined with a directory) can send and receive cache coherence messages to/from any cores.

The type of memory subsystem is also specified in the Hornet configuration script. Below are all common parameters of all memory types, which is defined in `[memory]` section.

```
[memory]
architecture = private-shared MSI/MESI
dram controller location = top and bottom
core address translation = stripe
```

```
core address translation latency = 1
core address translation allocation unit in bytes =
   4096
core address synch delay = 0
core address number of ports = 0
dram controller latency = 2
one-way offchip latency = 150
dram latency = 50
dram message header size in words = 4
maximum requests in flight per dram controller = 256
bandwidth in words per dram controller = 4
```

The first line tells which memory architecture the simulation is going to use. In order to use MSI/MESI protocol with private-L1 shared-L2 configuration, use `private-shared MSI/MESI` architecture.

Memory-specific configurations are given similarly to the core configurations. For example, detailed parameters of `private-shared MSI/MESI` memory is defined in `[memory::private-shared MSI/MESI]` section.

```
[memory::private-shared MSI/MESI]
use Exclusive state = no
words per cache line = 8
memory access ports for local core = 2
L1 work table size = 4
shared L2 work table size = 4
reserved L2 work table size for cache replies = 1
reserved L2 work table size for line eviction = 1
total lines in L1 = 256
associativity in L1 = 2
```

```
hit test latency in L1 = 2
read ports in L1 = 2
write ports in L1 = 1
replacement policy in L1 = LRU
total lines in L2 = 4096
associativity in L2 = 4
hit test latency in L2 = 4
read ports in L2 = 2
write ports in L2 = 1
replacement policy in L2 = LRU
```

3. Network configuration

Network is configured in the same way how original Hornet simulator is configured. However, the network must provide enough number of disjoint virtual channel sets to support the total number of message channels that core and memory models need to use.

The best practice is to have different flow IDs for different message channels by adding prefix to it. Configure your Hornet routing configuration in such a way that the flows with the same prefix in their flow IDs will use the same virtual channel set.

`dar/scripts/config/xy-shmem.py` script helps creating the routing configuration with the following parameters.

```
-x <arg> : network width (8)
-y <arg> : network height (8)
-v <arg> : number of virtual channels per set (1)
```

```
-q <arg> : capacity of each virtual channel in flits
   (4)
-c <arg> : core type (memtraceCore)
-m <arg> : memory type (privateSharedMSI)
-n <arg> : number of VC sets
-o <arg> : output filename (output.cfg)
```

This script will generate routing configurations as well as the list of default parameters of given core/memory types. Make sure the `-n` option specifies no less than the total number of message channels being used by the core and memory model.

Writing a custom core

The first thing to do is to inherit from `core` class (src/exec/core.hpp). `core` class has a pure virtual function `execute()`, and `update_from_memory_request()` which you will implement to define core behaviors. The memt

1. Memory accesses

`core` provides `m_memory` instance variable which you can use in `execute()` to access the configured memory subsystem. nearest L1 cache. To make a memory request, you first create a `memoryRequest` instance. An example follows:[1]

```
shared_array<uint32_t> data;
    /* initialized somewhere else */
shared_ptr<memoryRequest> write_req(new
  memoryRequest(maddr(), cur.word_count(), data)
m_memory->request(write_req);

shared_ptr<memoryRequest> read_req(new memoryRequest
  (maddr, word_count));
m_memory->request(read_req);
```

As in the example above, you use two-argument constructor of `memoryRequest` class for read requests. `addr` is the address that

[1] memoryRequest class is defined in dar/src/exec/memory.hpp

the read begins. A `maddr_t` type is a C++ struct that has two members, `space` and `address`. Each space has an independent memory space, and `word_count` tells how many words are read for this memory instruction, which usually depends on the core ISA. In case of writes, you pass an `shared_array` pointer as another argument to the constructor. `word_count` bytes of the array will be written to the memory.

`memoryRequest` has its own buffer for storing data (both for reads and writes). In case of writes, data is copied into this buffer in the constructor, so you can release `wdata` array right after calling the constructor.

The data buffer in `memoryRequest` will be released when the `memoryRequest` instance is deleted. As this instance will be used as a `boost::shared_ptr<>` instance in the shared memory framework, it is best to create a `memoryRequest` instance using `shared_ptr<>` (as in the above example), pass it to the framework, and forget about deleting it since it will be automatically released when you tell the framework to finish this request.

Since you create a `memoryRequest` instance, you can simply call its `status()` member function to examine the result of the request. It returns either `REQ_WAIT`, `REQ_DONE`, `REQ_RETRY` or `REQ_MIGRATE`. The following table summarizes actions to be taken for each case;

REQ_WAIT	The request is being scheduled or the memory is busy. Do not request again and check it later.
REQ_DONE	The request is done. In case of a read, the data is accessible by calling `data`() method of the `memoryRequest` instance.
REQ_RETRY	The memory was not able to accept the request. You need to make a memory request again. For your convenience, you can use `reset`() method of the `memoryRequest` instance and pass the same instance to `m_memory`, rather than re-initiating a new `memoryRequest` object.
REQ_MIGRATE	(Only if the core and memory model supports Execution Migration) This memory request cannot be served at the current location. The core need to migration the thread to home core, which is given by `home`() method of the memoryRequest instance.

2. For execution migrations

and other core-to-core communications

Sometimes, a core model may need a direct core-to-core communication. For example, execution migration needs to send a thread context directly to another core. The framework provides a general networking infrastructure that can be used for any types of core-to-core communications.

You can use any number of message channels for core-to-core communication. Each channel uses a different set of virtual channels, so they do not block each other on the network. However, it is required to have enough number of virtual channel sets to support every message channel that core and memory system use.

`core` class provides a send queue and a receive queue for each of all message channels. The following methods are used in `exec_core()` to get the pointers to those queues:

```
core_receive_queue(uint32_t channel);
core_send_queue(uint32_t channel);
```

`channel` selects a channel; **Because the smaller number of channel IDs (0~..) are reserved for the memory, core-to-core communication must use the greatest number of channel IDs** (that the memory subsystem does not use).

2.1. Sending messages

To send a core-to-core message, choose one of the send queues, and use `push_back(shared_pt<message_t>)` to enqueue a message. `message_t` type is defined as the following (see `src/exec/message.hpp`):

```
typedef struct {
    uint32_t type;
    uint32_t src;
    uint32_t dst;
    uint32_t flit_count;
    shared_ptr<void> content;
} message_t;
```

`type` tells the type of this message, so the receiver can cast back the `content` to its original form. When you send a message directly to another core, set `dst` field to the destination of the message (no broadcasting/multicasting supported), and `flit_count` to the non-zero number of flits to send the message excluding a head flit. `flit_count` is independent to the ACTUAL size of the data to be sent; you could send huge data with only one flit or small data with many flits. However, you cannot send an empty message, in other words, `flit_count` must be at least 1. Also, the network will send an additional head flit, so the total number of sent flits will be greater by 1.

Once a message is successfully sent out to the network, the message is automatically dequeued from the send queue. If network is congested, the size of message queues may grow. A

core may monitor the size of the send queues using `size()`, if it needs to know whether the message is sent or not. Also, a core may cancel the first message to send using `pop()` by itself. If `push_back(shared_ptr<message_t>)` fails because the queue is full, it will return `false` (you need to try it later gain). The core need not retain the memory for `msg_t` variable once `push_back(msg_t)` returns `true`.

2.2. Receiving messages

Once a message arrives at the destination, it will be automatically enqueued in the received queue of the used channel. The core can monitor the size of a receive queue using `size()`, and if it is not empty, get a copy of message (in `shared_ptr<message_g>`) by using `front()`.

Using your own cores

Once you write a new class derived from `core`, you have to set up the simulation to use the class. `[core]` section of DARSIM configuration is used to specify which type of core will be used for the simulation. For example, if you want to use `memtraceCore` of the framework, the configuration file should look like:

```
[core]
default = memtraceCore
```

Here, `memtraceCore` is used as a reserved word to indicate the type of cores. In order to add a new reserve word to for the use of your new core class, you will have to edit `src/tools/darimg` and `src/sys/sys.cpp`. By editing these files, you may add new sections and keys in DARSIM configuration to configure details of the new core class.

In sys.cpp file, instantiate your class and make sure to store its pointer to the `pe` array of the system. Take a look at how memtraceCore is initiated, and note that the first six arguments of its constructor is passed to the constructor of the parent class, `core`. Beside these six arguments, use any additional arguments to configure your custom core class.

```
core(const pe_id &id, const uint64_t &system_time,
     shared_ptr<id_factory<packet_id> >
         packet_id_factory,
     shared_ptr<tile_statistics> stats, logger &log,
     shared_ptr<random_gen> ran,
     shared_ptr<memory> mem,
     uint32_t number_of_core_msg_types,
     uint32_t msg_queue_size,
     uint32_t m_bytes_per_flit)
```

Other than the six arguments, the base class `core` takes three more arguments: `number_of_core_msg_types` tells how many message channels will be use. This is the total message channels used by the core or the memory system. `m_bytes_per_flit` specifies the bandwidth of on-chip network links.

Revision History

Date	Description
2011/01/28	The interface for remote accesses change.
2011/01/24	The initial release
2011/04/21	Version 2.0: major revisions