Shared Memory Framework for Hornet

Brandon Cho (mhcho@mit.edu)

Table of contents

Introduction

The shared memory framework provides a base class (`core`) that you can use to make your own cores using multi-level caches and DRAM controller, connected to each other (and to the cores) within a tile or via the DARSIM network.

It also provides an example core, `memtraceCore`. This core takes memory trace files and execute data memory operations in the traces on the configurable memory subsystem. `memtraceCore` uses normal memory operations via a default cache hierarchy (starting from the nearest L1, either local or remote), to use remote accesses, or to use execution migrations. `memtraceCore` assumes magic memory for instructions.

Writing cores

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

1. Normal memory accesses

`core` provides `nearest_memory()` which you can use in `exec_core()` to access the nearest L1 cache. The nearest L1 cache could be local or remote, depending on configurations.

When you want to access the nearest L1 cache, you first create a `memoryRequest` instance. An example follows:

```
maddr_r addr;
uint32_t byte_count, wdata[DATASIZE];
shared_ptr<memoryRequest> read_req, write_req;
read_req = shared_ptr<memoryRequest> (new
  memoryRequest(addr, byte_count));
uint32_t wdata = get_id().get_numeric_id();
write_req = shared_ptr<memoryRequest> (new
  memoryRequest(addr, byte_count, wdata));
```

As in the example above, you use two-argument constructor of `memoryRequest` class for read requests. `addr` is the address that the read begins (byte-addressed). `byte_count` tells how many bytes are read for this memory instruction, which usually depends on the core ISA. In case of writes, you pass an array pointer as another argument to the constructor. `byte_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 `request(shared_ptr<memoryRequest>)` of your nearest memory as the following:

```
mreq_id_t req_id = nearest_memory()->request(req);
```

The return value is a request ID, which you should keep to monitor the memory request, and more importantly, to finish the request once it is done. To check if the memory operation is done, you use `ready(mreq_id_t)` as the following.

```
bool is_done = nearest_memory()->ready(req_id);
```

If the memory request was a read, you will probably want to get the read value. The following shows how to get the data.

```
assert(nearest_memory()->ready(req_id));
shared_ptr<memoryRequest> ld_req =
  nearest_memory()->get_req(req_id);

for (uint32_t i = 0; i < ld_req->byte_count();++i) {
  uint32_t new_byte = *(ld_req->data()+i);
}
```

Once a memory operation is finished, `core` class is responsible to tell the nearest memory that it is done with the request to prevent memory leakage. Use `finish(mreq_id_t)` to do this as the following:

```
assert(nearest_memory()->ready(req_id));
nearest_memory()->finish(req_id);
```

Caution: `ready()` does NOT tell the availability of data in the nearest memory at the cycle when `ready()` is called. The data is guaranteed to be available in the nearest memory only for the cycle when the return value of `ready()` changes to `true`. The data may change or become unavailable at the nearest memory after the request is served, but previous `memoryRequest` instances will remain valid with old data.

Therefore, reading data out of the `memoryRequest` instance is not equivalent to read from the nearest cache, but equivalent to read from some in-core storage that can be assumed to be updated at the moment when the memory operation is finished (i.e., a register). Unless you understand this and specifically want to use data in `memoryRequest` instances sometime after they become available, it is safe to use `finish(mreq_id_t)` in every cycle for finished memory requests.

2. For remote accesses

Doing remote accesses is almost exactly the same to normal memory accesses. The main difference is that you use `remote_memory()` instead of `nearest_memory()`, and `ra_request` instead of `request` where you will specify the destination of the remote access. For example, if you want to initiate a memory request to core 7, do like the following:

```
mreq_id_t rid =
    remote_memory()->ra_request(req_id, 7);
```

By default, this will access the L1 cache at core 7. However, you can also specify which level of cache the memory request should access at the destination;

```
//ask L2 at core 7
mreq_id_t rid =
    remote_memory()->ra_request(req, 7, 2);
```

You can use other methods, `ready(mreq_id_t)`, `get_req(mreq_id_t)` and `finish(mreq_id_t)` with `remote_memory()` in the same way with `nearest_memory()`.

3. For execution migrations and other core-to-core communications

The framework does not fully support execution migration; each core implementation may take different approaches[1]. However, the framework provides a networking infrastructure that can be used for execution migration, or any types of core-to-core communications using this infrastructure.

Two channels are provided for your core to use in core-to-core communication. Each channel uses a different set of virtual channels, so they do not block each other on the network.

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

```
core_receive_queue(int channel);
core_send_queue(int channel);
```

`channel` selects a channel; as there are two channels available, the value of `channel` must be either 0 or 1.

[1] `memtraceCore` implements an ENC-based execution migration. See src/exec/memtraceCore.cpp

3.1. Sending messages

When you want to send a migration message, you choose one of the send queues, and use `push_back(msg_t)` to enqueue a message. `msg_t` type is defined as the following (see src/exec/message.hpp):

```
typedef struct {
    uint32_t dst;
    uint32_t flit_count;
    msg_core_t core_msg;
    msg_mem_t mem_msg;
    /*don't need to specify the following fields */
    uint32_t src
    msg_type_t type;
} msg_t;
```

When you send a message directly to another core, set `dst` field to the destination of the message (no broad/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.

`msg_core_t` is defined as the following:

```
typedef struct {
    void *context;
} msg_core_t;
```

You should set this `context` field to the pointer to data you send. The receiving core will get an access to this pointer and is responsible to retrieve the data.

You need not specify `type` and `src` because it will automatically set. You also do not use `mem_msg` for core-to-core communication. The following is an example to send a null message to core 7 through the high-priority channel:

```
msg_t msg;
msg.dst = 7;
msg.flit_count = 1;
msg.mig_msg.context = NULL;
bool success =
    mig_send_queue_high_priority()->push_back(msg);
```

As in the example, `push_back(msg_t)` returns a boolean value that indicates whether a message is successfully enqueued (the queue may have a limited size). If the return value is `true`,the message will be sent when previous messages in this channel (if any) are all sent in order and the network becomes available. If the return value is `false`, however, the core need to retry `push_back(msg_t)` later.

Once a message is successfully sent out to the network, the message is automatically dequeued from the send queue. 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.

The core need not retain the memory for `msg_t` variable once `push_back(msg_t)` returns `true`. However, the framework does not try to retain or release memory for `*context` at any point. Therefore, the memory for `*context` must not be deleted until the message is arrived at the destination. It is best for the sender to allocate memory for `*context`, and for the receiver to release it.

3.2. 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 `msg_t`) by using `front()`.

As `front()` gives you COPY of the message, you can safely remove the message from the queue using `pop()` (unless the message will remain at the head of the queue). As explained earlier, the core might be responsible for releasing `*context` after using it. The following shows an example of the whole process to receive a message.

```
msg_t msg;
void *context;
if (mig_receive_queue_low_priority()->size()>0) {
  msg = mig_receive_queue_low_priority()->front();
  context = msg.mig_msg.context;
  mig_receive_queue_low_priority()->pop();
}
```

```
// work with context
delete context;
```

Memory system & network configuration

1. Express memory setup

The easiest way to configure the memory subsystem is to add a section named `memory hierarchy` in a DARSIM configuration file. In the section, add keys that contains the location of memory for a specific level of all cores. The following shows an example of 8 cores with 8 L1 caches, 4 L2 caches and 2 DRAM controllers:

```
[memory hierarchy]
1 = 0 1 2 3 4 5 6 7
2 = 0 0 2 2 4 4 6 6
3 = 0 0 0 0 6 6 6 6
```

Each key (1,2, and 3) represents a level in the memory hierarchy. The last level (3) represents DRAM controllers, and all other levels (if any) represents caches.

The numbers in each key represents the location of the memory for the level. There are as many numbers as cores (8 in the above example). The first line in the example tells the L1 cache for core 0 is at 0, L1 for core 1 is at 1, L2 for core 2 is at 2, and so on. In other words, all cores have its own local L1 caches.

```
2 = 0 0 2 2 4 4 6 6
```

On the other hand, the second line tells that L2 cache for the L1 cache of core 0 is at core 0, and L2 for the L1 of core 1 is also at 0. This means that core 0 and core 1 share the same L2 cache, located at 0. In the same way, core 2 and core 3 share the same L2 cache located at 2, and so on.

```
3 = 0 0 0 0 6 6 6 6
```

The third line shows that the L2 cache of core 0 will talk to DRAM controller located at core 0. According to the second line, core 2 does not have an L2 cache, so core 2 does not usually talk to DRAM controller directly. In this example the DRAM controller location of core 1 is set to `0`, but this is merely a placeholder. Core 2 does have an L2, and this will also talk to the DRAM controller at core 0 because the third number is also `0`. On the other hand, L2 caches at core 4 and core 6 will share another DRAM controller located at 6 as the fifth and the seventh numbers are 6.

Although this express setup is very convenient, it has a few restrictions for now. For example, this setup assumes all cores have exactly the same level of memory hierarchy. If you want some cores to have L1 caches only, while the other cores have L2 caches as well, you have to manually build the memory subsystem in `src/sys/sys.cpp`. Although temporarily, you also have to edit `src/sys/sys.cpp` file to change cache and DRAM controller parameters, such as capacity, block size, throughput, replacement policies and so on. Finally, if you build new type of core, you have to make sure that `src/tools/darimg` and `src/sys/sys.cpp` file take care of the `memory hierarchy` section in your configuration. See the next chapter ‘Using your own cores’ for more details.

2. Network setup

The basic memory subsystem requires two virtual channel sets. Remote access requires another two virtual channel sets, and execution migration (which uses the two core-to-core communication channels) requires another two. Therefore, the minimum number of virtual channels is 6, if you support both remote accesses and execution migration. Also, as the shared memory framework identifies packets belong to each message type by predefined flow IDs, your configuration must define routing for all the flow IDs given by the same convention.

`scripts/config/xy-shmem.py` generates these DARSIM configuration for you. You could use this script with the following options (values in parenthesis are default values).

```
-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 model (memtraceCore)
-e : support execution migration
-r : support remote accesses
-o <arg> : output filename (output.cfg)
```

Currently, the script only supports dimension-order routing, and there are no supports for other routing algorithms such as O1TURN or ROMM. Note that these routing algorithms requires at least two virtual channels for EACH virtual channel set, so the minimum number of virtual channels becomes 12 if you support both remote accesses and execution migration.

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. If you take this approach, you also have to write (or copy-and-paste) code that works on the express memory setup (if you are willing to use this method).

As a quick ad-hoc way of using new core classes in local directories, there is a reserved word `customCore`. If you use `customCore` in DARSIM configuration, you need not work too much on `src/tools/darimg`, and `src/sys/sys.cpp` files., and you can use the express memory setup without adding codes. The only thing you have to do is go into `src/sys/sys.cpp` file, find the following piece of code,

```
case CORE_CUSTOM: {
  /* set your core configuration */

  /* create core object */
  shared_ptr<core> new_core;

  p = new_core;
```

and add the pointer to newly created instance of the core class and assign it to variable `p`. It should look similar to the code in the next page.

```
case CORE_CUSTOM: {
    /* set your core configuration */
    core::core_cfg_t core_cfgs;
    yourNewClass::cfg_t newcore_cfgs;
    // set parameters in core_cfgs and newcore_cfgs

    /* create core object */
    shared_ptr<yourNewClass> new_core
          = shared_ptr<yourNewClass> (
                yourNewClass(id, t->get_time(),
                      t->get_packet_id_factory(),
                      t->get_statistics(), log, ran,
                      core_cfgs, newcore_cfgs));

    p = new_core;
```

The first six arguments of the constructor in the above example is variables commonly shared by other DARSIM components. For example, `t->get_time()` returns a reference to the global clock counter. The constructor of `core` class takes these arguments, so your constructor will usually pass them to the `core` constructor.

`core::core_cfg_t` type defines the following four parameters used for the framework:

```
typedef struct {
    uint32_t msg_queue_size;
    uint32_t flits_per_mem_msg_header;
    uint32_t bytes_per_flit;
    uint32_t memory_server_process_time;
} core_cfg_t;
```

msg_queue_size is the capacity of message send/receive queues, such as core_msg_send_queue() described in the previous chapter. flits_per_mem_msg_header is the size of header of memory messages in flits (this is different to the head flit created by on-chip network). bytes_per_flit sets flit width, which will decide the number of flits required to send data over the network. Finally, memory_server_process_time is the number of cycles that is required to decode a memory request (or a remote access request) received from the network.