CHALMERS
UNIVERSITY OF TECHNOLOGY

EDA284 Lab 3: Simulating Coherence Protocols with gem5

Contacts: Miquel Pericàs, Sonia Rani Gupta, Madhvan and Mustafa
Emails: miquelp, soniar@chalmers.se

March 4, 2022

The goal of this lab is to enhance your understanding of cache coherence. In the lectures, you learned the fundamental concepts of coherence protocols. However, even simple three-state protocols, such as MSI (Modified-Shared-Invalid), are more complex in real life implementations. With the help of `gem5`, this lab will give a quick overview of these more complex aspects of coherence protocols such as transient states.

By the end of this lab:

- You will have an overview of how coherence protocols are implemented and why we need transient states.
- You will learn how to debug coherence protocols using the Protocol Tracer.
- You will analyze multithreaded codes that exhibit certain cache usage and compare across protocols (MI vs MSI).

Useful and Optional References:

(a) Learning `gem5` part 3 (recommended read) http://learning.gem5.org/book/part3/index.html

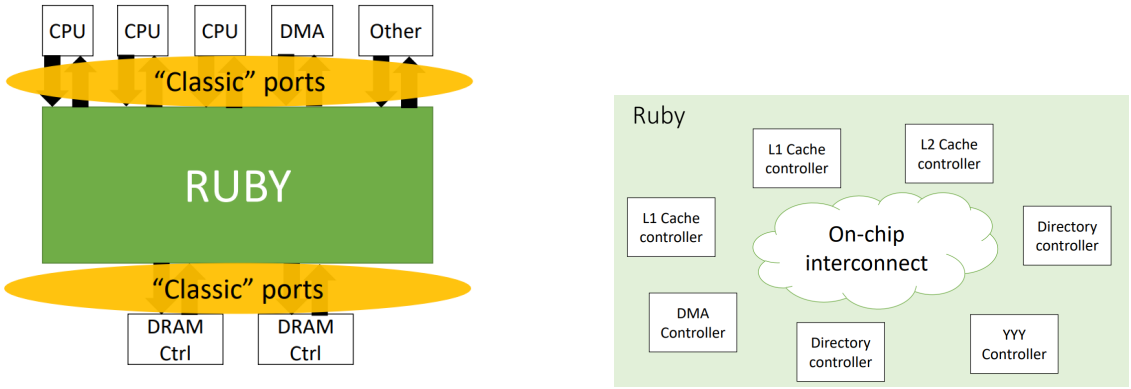
(b) The Primer on Memory Coherence and Consistency book is a great resource on coherence. The book is available online through http://pages.cs.wisc.edu/ markhill/papers/primer2020_2nd_edition.pdf

(c) The ASPLOS 2018 `gem5` tutorial (part 3). http://learning.gem5.org/tutorial/

Prerequisites:

(a) Download the pre-built `gem5` package available on https://chalmersuniversity.box.com/s/n3dw41z4xgy329eaefwtbo86i4o9l6f3. **It is highly recommended to delete or archive older versions of `gem5` to save space**.

(b) The `gem5` build system uses the `tcmalloc` library that is part of `gperftools`. So, download the pre-built `gperftools` library from here. https://chalmersuniversity.box.com/s/4wvgp3zqtzpvgi00lbxx3f3hublsr0mk. After you extract the folder, update `LD_LIBRARY_PATH` to point to the lib (including also the libpng lib) folder as follows:



(a) Ruby interaction with `gem5`

(b) Ruby components

Figure 1: An overview of the Ruby cache system

```
export LD_LIBRARY_PATH=PATH_TO_GPERFTOOLS/lib/:/chalmers/sw/sup64/ansys-2020r1/
    libpng/lib/
```

1 Introduction to Ruby

Ruby[1] comes from the multifacet `gem5` project. Ruby provides a detailed cache memory and cache coherence models as well as a detailed network model (Garnet). It is flexible. It can model many different kinds of coherence implementations, including broadcast, directory, token, region-based coherence, and is simple to extend to new coherence models. Also, it is a mostly drop-in replacement for the classic memory system. There are interfaces between the classic `gem5` MemObjects and Ruby, but for the most part, the classic caches and Ruby are not compatible. Figure 1 is an overview of how Ruby is interfaced to `gem5` and the different components that constitute the Ruby cache system. The most important structure in Ruby is the controller, or state machine. Controllers are implemented by writing a `SLICC` state machine file.

`SLICC` is a domain-specific language (Specification Language including Cache Coherence) for specifying coherence protocols. `SLICC` files end in ".sm" because they are state machine files. Each file describes states, transitions from a begin to an end state on some event, and actions to take during the transition.

Each coherence protocol is made up of multiple `SLICC` state machine files. These files are compiled with the `SLICC` compiler which is written in Python and part of the gem5 source. The slicc compiler takes the state machine files and output a set of C++ files that are compiled with all of gem5's other files. These files include the SimObject declaration file as well as implementation files for SimObjects and other C++ objects.

Currently, `gem5` supports compiling only a single coherence protocol at a time. For instance, you can compile `MI_example` (the default, poor performance, protocol) or the MSI into gem5, or

[1] http://learning.gem5.org/book/part3/intro.html

you can use `MESI_Two_Level`. To get a list of available protocols to build a Ruby system with, check *src/mem/ruby/protocol/SConsopts*. Example code snippet is as follows:

```
    all_protocols.extend([
    'GPU_VIPER', 'GPU_VIPER_Baseline',
    'GPU_VIPER_Region', 'GPU_RfO',
    'MOESI_AMD_Base', 'MESI_Two_Level',
    'MESI_Three_Level', 'MESI_Three_Level_HTM',
    'MI_example', 'MSI', 'MOESI_CMP_directory',
    'MOESI_CMP_token', 'MOESI_hammer',
    'Garnet_standalone', 'None'
    ])
```

Notes:

We have built two systems one with an MI simple and inefficient protocol, and another with MSI as a more realistic protocol. If you would like to build the systems on your own, you can **optionally** use the sample build commands below:

- MI_example:
  `scons build/ARM_MI/gem5.opt --default=ARM PROTOCOL=MI_example SLICC_HTML=True`
- MSI:
  `scons build/ARM_MSI/gem5.opt --default=ARM PROTOCOL=MSI SLICC_HTML=True`

Note that in both cases, we have enabled a user-friendly HTML table view of the protocol transitions (more details later). The builds utilizing MI and MSI protocols are already available on *build/ARM_MI/* and *build/ARM_MSI/*, respectively.

2 The cache state machine in `SLICC`

The most important part of developing a coherence protocol using `SLICC` is producing the following two files:

- *src/mem/ruby/protocol/MSI-cache.sm*
- *src/mem/ruby/protocol/MSI-dir.sm*

`gem5` uses `SLICC`, a domain specific language, to define the coherence protocol. Remember that a coherence protocol implementation consists of (a) a set of coherence controllers (i.e., Finite State Machines - FSMs) and (b) a set of interactions between these controllers [1]. The file *src/mem/ruby/protocol/MSI.slicc* contains a list of all the files used by the MSI protocol; the protocol's name is specified in the first line.

```
protocol "MSI";
include "RubySlicc_interfaces.slicc";
include "MSI-msg.sm";
include "MSI-cache.sm";
include "MSI-dir.sm";
```

The *RubySlicc_interfaces.slicc* file contains all the files relevant to Ruby memory system. You will not need to modify this file, so treat it as an included header file. The *MSI-cache.sm* file holds the definition of the L1 Cache coherence controller, whereas the *MSI-dir.sm* file defines the FSM associated with the directory controller. In addition to core-initiated memory requests, I/O devices can also access memory via DMA requests. You can safely ignore these for your implementation.

The different constructs specified as part of the *MSI-cache.sm* file:

(1) **Cache memory**: Where the data is stored.

(2) **Message buffers**: Sending/receiving messages from network

(3) **State declarations**: The stable and transient states

(4) **Event declarations**: State machine events that will be "triggered"

(5) **Other structures and functions**: Entries, TBEs, get/setState, etc.

(6) **In ports**: Trigger events based on incoming messages

(7) **Actions**: Execute single operations on cache structures

(8) **Transitions**: Move from state to state and execute actions

Tasks:

(a) In the process of producing the two ARM systems that use the MI and MSI protocols for cache coherence, we have enabled a user-friendly HTML output granted by the `SLICC` compiler (e.g. the MSI version is located at *./build/ARM_MSI/mem/ruby/protocol/html/L1Cache_table.html*). Look at the HTML produced for the MSI protocol by the `SLICC` compiler. List 2 transient states and the events the trigger them.

(b) Review "Learning `gem5` part 3 - Section: Transition code blocks" (reference (a)), and interpret (at a high level) the state transition `SLICC` code below.

```
transition({SM_AD, SM_A}, {Store, Replacement, FwdGetS, FwdGetM}) {
  stall;
}
```

(c) Why are transient states necessary? and why does a coherence protocol use stalls?

(d) What is the meaning of a deadlock in a directory coherence protocol and how can we avoid it?

3 Validating the protocols via the Ruby Random Tester

In order to validate the correctness of your protocol, you can run the following from your gem5 directory (e.g. the MSI protocol):

```
    ./build/ARM_MSI/gem5.opt ./configs/example/ruby_random_test.py
```

The ruby random test.py python script receives a set of configuration parameters. You can see all of them via -help. The most important ones are:

```
-n, --num-cpus Number of cpus injecting load/store requests to the memory system.
--num-dirs Number of directory controllers in the system.
-m, --maxtick Number of cycles to simulate.
-l, --checks Number of loads to be performed.
```

The random tester injects random requests to the L1 caches. If a bug in your code exists the test will not complete, but will instead exit with an assertion or an error message.

Task:

Test the MI and MSI protocols with 2 cpus. State the final outcome of the test. Document the **error(s)** if you find any.

4 Analyzing coherence using multithreaded codes

The dual-core CPU system configuration scripts, that leverage MI and MSI (each core has a private L1 cache only) are available at the following locations:

- MI: *configs/simple_ruby/simple_ruby_MI.py*
- MSI: *configs/simple_ruby/simple_ruby_MSI.py*

To better facilitate debugging, you can use the ProtocolTrace debug flag, which generates a complete dump of all the protocol transitions and actions of the coherence controllers in your system. Figure 2 shows such sample output. The highlighted line starts with the tick number (2883000), the machine version (CPU core number in this case), the Ruby component (L1Cache), the event (Load), the state transition ($S > S$), the physical word address ($0x27dc0$) and the line address ($0x27dc0$). For more information, check http://learning.gem5.org/book/part3/MSI/debugging.html

Notes:

To run your code (e.g. with MSI protocol and tracing enabled), use the following command

```
./build/ARM_MSI/gem5.opt --debug-flags=ProtocolTrace ./configs/simple_ruby/
    simple_ruby_MSI.py
Note: path_to_binary is already set in simple_ruby_MSI.py file. If you change path
    of binary then update the path in simple_ruby_MSI.py file. To run with MI
    protocol, use simple_ruby_MI.py file
```

As you know, the MSI protocol introduces the "Shared" state on top of the MI example protocol, that is, it is possible to read a value that is in shared/modified state in the corresponding cache. To read more about the MI example, check this link
https://www.gem5.org/documentation/general_docs/ruby/MI_example/.

Tasks:

You are given a multi-threaded code (*./configs/simple_ruby/bins/threads.c*). The code uses 2 threads to parallelize an element-wise addition of the arrays a and b (i.e. $c = a + b$). To build the code, type `make` while in the folder.

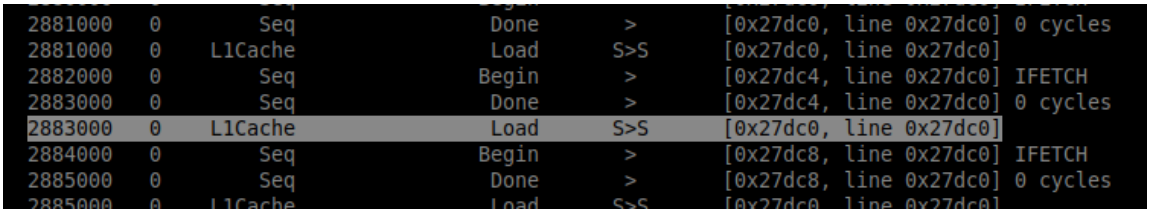


Figure 2: Sample protocol trace output produced with –debug-flags=ProtocolTrace

(a) The code "threads.c" has a problematic usage of the cache. Identify it. Use the protocol tracer with the MSI protocol to provide an evidence of the problem. (**Hint**: redirect the debug output to a file to search easily. Google "redirect output to file").

(b) Fix the problem in the code "threads.c", and compare the performance of the MSI-enabled system before and after the fix in terms of the number of ticks.

(c) Suggest a code (possibly replacing the code in function `array_add` in "threads.c") where the simple MI protocol outperforms the MSI protocol. State a reason for your observation.

Report Submission

- You should submit the lab as a group of two. Write down CIDs of each partner.
- You will get a PASS only if all the tasks are properly addressed in the report.
- The reports should be submitted on Canvas, no later than **Friday, March 18th, 2022 23:59**