CS4223 Assignment 2

Julius Putra Tanu Setiaji (A0149787E), Kyle Timothy Ng Chu (A...)

15 November 2019

1 Choice of Programming Language

Initially, we tried to implement the trace simulator using Python. However, after a while, we realised that it gets unwieldy, unreadable, and difficult to extend given the complexity of implementing a trace simulator. Particularly, while Python has enumerations, they are not as powerful as Algebraic Data Types (ADT) in other languages, especially with regards to associated values. Python's lack of pattern matching also makes implementing a finite state machine (FSM) a hassle. While Python's dynamic typing allows more rapid prototyping, a better compile-time typecheck would be really useful to prevent bugs.

Haskell seems to fit the bill really well here. However, since we are essentially simulating a mutable, eagerly-evaluating von Neumann architecture machine, Haskell's characteristic of being immutable and lazy can become an impediment. We need a language that has some of Haskell's features such as being strongly and statically typed, pattern matching and ADT, while still allowing us to have mutability. For these reasons as well as familiarity, we chose to implement our tracing simulator using **Scala**.

2 Implementation

Our code can be divided into 3 different parts: specific code for common code among all the protocols, the MESI Protocol trace simulator, and specific code for the Dragon Protocol trace simulator.

2.1 Common Code

Overall, our code uses the delegation pattern, where each device can choose to become a delegate of another device. This way, devices can have a two-way communication of some sort. The devices themselves are designed as Finite State Machines (FSM) to allow for more robust code.

2.1.1 `coherence.bus`

Across the different protocols, we use the same bus implementation located in the `coherence.bus` package.

The design that we chose for our bus is **an atomic bus**, where one cache is granted exclusive access to the bus to perform one transaction (a transaction is defined as the cache granted exclusive access placing command on the bus, and then receiving response by other caches). There are some potential issues that may prevent correctness in this concurrent system, namely deadlock, livelock, starvation and race conditions. As I explain the design of our atomic bus, I will mention how do we prevent these problems from happening.

When a device wants to place command on the bus, it has to first request access from the bus, and the device has to be a `BusDelegate`. This ensures that a cache will have exclusive access to the

bus, with the bus arbitrer acting like a mutex lock. The bus arbitrer uses a first come first serve arbitration policy to prevent starvation.

When the bus arbitrer decides that it is the turn for one `BusDelegate` to obtain exclusive access to the bus, the bus will invoke `busAccessGranted()` on the `BusDelegate` to obtain the command that the device wants to place on the bus. This is to prevent race condition, because the transition of the states of the cache lines are not actually atomic even though the bus is atomic. Consider the case in MESI protocol when Processors P1 and P2 write to cache line A simultaneously (both need to issue `BusUpgr` to move the line from the `S` state to the `M` state). Suppose P1 wins bus access and places `BusUpgr` on the bus. P2 is still waiting for bus access to place its own `BusUpgr`, but it cannot proceed because P1 has exclusive access to the bus. P2 then receives `BusUpgr` from `P1`, thus it must invalidate line A according to the MESI protocol. As such, P2 must also change its pending `BusUpgr` request to a `BusRdX`. With our design, we will only ask the cache what command to place on the bus when they are granted access, thus allowing the caches to change the command that they send between the moments access is requested and access is granted.

After a device places a command on the bus, `onBusCompleteMessage` method on all other devices as `BusDelegate` will get invoked. This allows them to make internal state transitions in response to the command on the bus, as well as place their response on the bus. The bus will arbitrarily grant one device to place their response on the bus. Once the device having exclusive access to the bus decided that it has received enough responses, it will indicate to the bus that it wishes to relinquish its exclusive access. This fact is propagated to all other devices through invocation of the `onBusTransactionEnd` of the `BusDelegate`. The bus will then pick another pending access request, and perform the same cycle all over again. Note that to avoid livelock, the device that obtains exclusive access must be allowed to complete before exclusive access is relinquished, thus our design of letting the device decide when to relinquish exclusive access of the bus. Consider two processors P1 and P2 writing to cache block B using MESI cache coherency protocol. First, P1 acquired access to the bus and issues `BusRdX`, resulting in P2 invalidating its cache line. However, without exclusive ownership, this scenario can happen: before P1 can perform the write, P2 acquires bus, issues `BusRdX`, resulting in P1 invalidating its cache line, and so on so forth, which results in a livelock.

2.1.2 `coherence.devices`

2.1.3 `Main.scala` and `Protocol.scala`

This is the main driver code of the simulator. It takes in arguments passed by user and parses them into the appropriate types, such as `Int`, `String` and `Protocol`. `Protocol` is actually defined inside `Protocol.scala`, which is an enumeration of supported cache coherency protocols. In our case we have `MESI` and `Dragon`. If all the arguments are parsed successfully, the appropriate tracing simulator is then invoked.

2.1.4 `Debug.scala`

This file provides code for debug printing that can be elided for final production code.

2.1.5 `Address.scala`

This file provides the case class for addressing in an n-way associative cache through tag and set index.