Final Project Part 2: Cache Coherence

Due (together with part 1): 3:00pm, Wednesday December 9, 2015

1 Overview

In this part of the project, we will implement a multicore system shown in Figure 1 in simulation. The system consists of two cores, and each core has its own private caches. The data caches (D caches) and main memory are kept coherent using the MSI protocol introduced in class. Since we don't have self-modifying programs, the instruction caches (I caches) can directly access the memory without going through any coherent transactions.

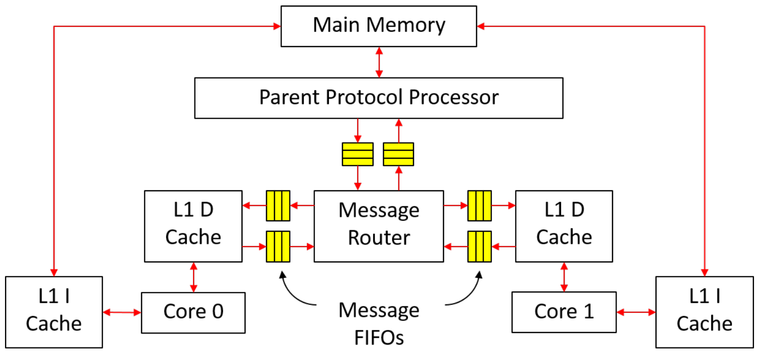

Figure 1: Multicore system

Since this system is quite complex, we have tried to divide the implementation into multiple small steps, and we have provided testbenches for each step. However, passing the testbenches **does not** imply that the implementation is 100% correct.

2 Implementing Units of Memory Hierarchy

2.1 Message FIFO

The message FIFO transfers both request and response messages. For a message FIFO from a child to the parent, it transfers upgrade requests and downgrade responses. For a message FIFO from the parent to a child, it transfers downgrade requests and upgrade responses.

The message types transferred by the message FIFO is defined in `src/includes/CacheTypes.bsv` as follow:

```
typedef struct{
  CoreID             child;
  Addr               addr;
  MSI                state;
  Maybe#(CacheLine) data;
} CacheMemResp deriving(Eq, Bits, FShow);
typedef struct{
  CoreID      child;
  Addr        addr;
  MSI         state;
} CacheMemReq deriving(Eq, Bits, FShow);
typedef union tagged {
  CacheMemReq      Req;
  CacheMemResp     Resp;
} CacheMemMessage deriving(Eq, Bits, FShow);
```

`CacheMemResp` is the type for both downgrade responses from a child to the parent, and upgrade responses from the parent to a child. The first field `child` is the ID of the D cache involved in the message passing. The type `CoreID` is defined in `Types.bsv`. The third field `state` is the MSI state that the child has downgraded to for a downgrade response, or the MSI state that the child will be able to upgrade to for a upgrade response.

`CacheMemReq` is the type for both upgrade requests from a child to the parent, and downgrade requests from the parent to a child. The third field `state` is the MSI state that the child wants to upgrade to for an upgrade request, or the MSI state that the child should be downgraded to for a downgrade request.

The interface of message FIFO is also defined in `CacheTypes.bsv`:

```
interface MessageFifo#( numeric type n );
  method Action enq_resp( CacheMemResp d );
  method Action enq_req( CacheMemReq d );
  method Bool hasResp;
  method Bool hasReq;
  method Bool notEmpty;
  method CacheMemMessage first;
  method Action deq;
endinterface
```

The interface has two enqueue methods (`enq_resp` and `enq_req`), one for requests and the other for responses. The boolean flags `hasResp` and `hasReq` indicate whether is any response or request in the FIFO respectively. The `notEmpty` flag is simply the OR of `hasResp` and `hasReq`. The interface only has one `first` and one `deq` method to retrieve one message at a time.

As mentioned in the class, a request should never block a response when they both sit in the same message FIFO. To ensure this point, we could implement the message FIFO using two FIFOs as shown in Figure 2. At the enqueue port, requests are all enqueued into a request FIFO, while responses are all enqueued into another response FIFO. At the dequeue port, response FIFO has priority over request FIFO, i.e. the `deq` method should dequeue the response FIFO as long as the response FIFO is not empty. The numeric type `n` in the interface definition is the size of the response/request FIFO.

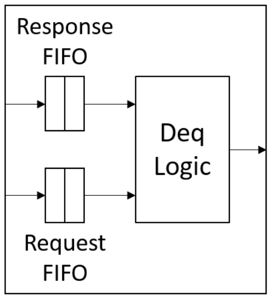

Figure 2: Structure of message FIFO

Exercise 1 (10pt): Implement the message FIFO (`mkMessageFifo` module) in `src/includes/MessageFifo.bsv`. We provide a simple test in the `unit_test/message-fifo-test` folder. Use `make` to compile, and use `./simTb` to run the test.

2.2 Message Router

The message router connects all L1 D caches and the parent protocol processor. We will implement this module in `src/includes/MessageRouter.bsv`. It is declared as:

```
module mkMessageRouter(
  Vector#(CoreNum, MessageGet) c2r, Vector#(CoreNum, MessagePut) r2c,
  MessageGet m2r, MessagePut r2m,
  Empty ifc
);
```

The `MessageGet` and `MessagePut` interfaces are just restricted views of the `MessageFifo` interface, and they are defined in `CacheTypes.bsv`:

```
interface MessageGet;
  method Bool hasResp;
  method Bool hasReq;
  method Bool notEmpty;
  method CacheMemMessage first;
  method Action deq;
endinterface
interface MessagePut;
  method Action enq_resp(CacheMemResp d);
  method Action enq_req(CacheMemReq d);
endinterface
```

We have provided the `toMessageGet` and `toMessagePut` functions to convert a `MessageFifo` interface to `MessageGet` and `MessagePut` interfaces. Below is an introduction to each module argument:

- `c2r` is the interface of the message FIFO from each L1 D cache.
- `r2c` is the interface of the message FIFO to each L1 D cache.
- `m2r` is the interface of the message FIFO from the parent protocol processor.
- `r2m` is the interface of the message FIFO to the parent protocol processor.

The major functionality of this module falls into two parts:

1. sending messages from the parent (`m2r`) to the correct L1 D cache (`r2c`), and
2. sending messages from L1 D caches (`c2r`) to the parent (`r2m`).

It should be noted that response messages have priority over request messages just like the case in message FIFO.

Exercise 2 (10pt): Implement the `mkMessageRouter` module in `src/includes/MessageRouter.bsv`. We provide a simple test in the `unit_test/message-router-test` folder. Use `make` to compile, and use `./simTb` to run the test.

2.3 L1 D Cache

The blocking L1 D cache (*without* store queue) will be implemented in `src/includes/DCache.bsv`:

```
module mkDCache#(CoreID id)(MessageGet fromMem, MessagePut toMem, RefDMem refDMem, DCache ifc);
```

Below is the introduction to each module parameter and argument:

- `id` is the core ID, which will be attached to every message sent to the parent protocol processor.
- `fromMem` is the interface of the message FIFO from parent protocol processor (or more accurately the message router), so downgrade requests and upgrade responses can be read out from this interface.
- `toMem` is the interface of the message FIFO to parent protocol processor, so upgrade requests and downgrade responses should be sent to this interface.
- `refDMem` is for debugging, and currently you do not need to worry about it.

The `DCache` interface returned by the module is defined in `CacheTypes.bsv` as follow:

```
interface DCache;
  method Action req(MemReq r);
  method ActionValue#(MemResp) resp;
endinterface
```

You may have noticed that the `MemOp` type (defined in `MemTypes.bsv`), which is the type of the `op` field of `MemReq` structure (defined in `MemTypes.bsv`), now have five values: `Ld, St, Lr, Sc` and `Fence`. For now you only need to handle `Ld` and `St` requests. You could add logic in the `req` method of the `DCache` interface, which reports error if it detects requests other than `Ld` or `St`.

The `MemReq` type also has a new field `rid`, which is the ID of the request used for debugging. `rid` is of type `Bit#(32)`, and should be unique for each request from the same core.

We will implement a 16-entry direct-mapped L1 D cache (the number of cache lines is defined as type `CacheRows` in `CacheTypes.bsv`). We suggest to use vector of registers to implement the cache arrays in order to assign initial values. We have also provided some useful functions in `CacheTypes.bsv`.

The `MSI` state type is defined in `CacheTypes.bsv`:

```
typedef enum { M, S, I } MSI deriving( Bits, Eq, FShow );
```

We have made `MSI` type become an instance of the `Ord` typeclass, so we can apply comparison operator (`>, <, >=, <=`, etc.) on it. The order is `M > S > I`.

Exercise 3 (10pt): Implement the `mkDCache` module in `src/includes/DCache.bsv`. This should be a blocking cache *without* store queue. You may want to use the work-around in Exercise 1 in the first part of the final project to avoid future scheduling conflicts when the D cache is integrated to the processor pipeline. We provide a simple test in the `unit_test/cache-test` folder. Use `make` to compile, and use `./simTb` to run the test.

2.4 Parent Protocol Processor

The parent protocol processor will be implemented in `src/includes/PPP.bsv`:

```
module mkPPP(MessageGet c2m, MessagePut m2c, WideMem mem, Empty ifc);
```

Below is the introduction to each module argument:

- `c2m` is the interface of the message FIFO from L1 D caches (actually from the message router), and upgrade requests and downgrade responses can be read out from this interface.
- `m2c` is the interface of the message FIFO to L1 D caches (atually to the message router), and downgrade requests and upgrade responses should be sent to this interface.
- `mem` is the interface of the main memory, which we have already used in the first part of the project.

In the lecture, the directory in the parent protocol processor record the MSI states for every possible address. However this will take a significant amount of storage for a 32-bit address space. To reduce the amount of storage needed for the directory, we notice that we only need to track addresses that exist in L1 D caches. Specifically, we could implement the directory as follow:

```
Vector#(CoreNum, Vector#(CacheRows, Reg#(MSI))) childState <- replicateM(replicateM(mkReg(I)));
Vector#(CoreNum, Vector#(CacheRows, Reg#(CacheTag))) childTag <- replicateM(replicateM(mkRegU));
```

When the parent protocol processor wants to know the approximate MSI state of address `a` on core `i`, it can first read out `tag=childTag[i][getIndex(a)]`. If `tag` does not match `getTag(a)`, then the MSI state must be `I`. Otherwise the state should be `childState[i][getIndex(a)]`. In this way, we dramatically reduce the storage needed by the directory, but we need to maintain the `childTag` array when there is any change on the children states.

Another difference from the lecture is that the main memory data should be accessed using the `mem` interface, while the lecture just assumes a combinational read of data.

Exercise 4 (10pt): Implement the `mkPPP` module in `src/includes/PPP.bsv`. We provide a simple test in the `unit_test/ppp-test` folder. Use `make` to compile, and use `./simTb` to run the test.

3 Testing the Whole Memory Hierarchy

Since we have constructed each piece of the memory system, we now put them together and test the whole memory hierarchy using the testbench in `uint_test/sc-test` folder. The test will make use of the "`RefDMem refDMem`" argument of `mkDCache`, and we need to add a few calls to methods of `refDMem` in `mkDCache`. `refDMem` is returned by a reference model for coherent memory (`mkRefSCMem` in `src/ref/RefSCMem.bsv`), and this model can detect violation of coherence based on the calls of methods of `refDMem`. `RefDMem` is defined in `src/ref/RefTypes.bsv` as follow:

```
interface RefDMem;
  method Action issue(MemReq req);
  method Action commit(MemReq req, Maybe#(CacheLine) line, Maybe#(MemResp) resp);
endinterface
```

The `issue` method should be called for each request in the `req` method of `mkDCache`:

```
method Action req(MemReq r);
  refDMem.issue(r);
  // then process r
endmethod
```

This will tell the reference model the program order of all requests sent to the D cache.

The `commit` method should be called when a request finishes processing, i.e. when a `Ld` request gets load result, or a `St` request writes to data array in the cache. Below is the introduction to each method argument of `commit`:

- `req` is the request that is committing (i.e. finishing processing).
- *[Updated on Dec 6th]* `line` is the original value of the cache line that `req` is accessing. The cache line here refers to the 64B data block with line address `getLineAddr(req.addr)`. Therefore it does not necessarily refer to the line in the D cache, because D cache may just contain garbage data. Since `line` is the original value, in case of committing a store request, it should be the value before being modified by the store.

  If we know the cache line data, `line` should be set to `tagged Valid`. Otherwise, we set `line` to be `tagged Invalid`. In case of `mkDCache`, we always know the cache line data when a request commits, because it is either already in D cache or in the upgrade response from parent. Therefore `line` should always be set to `tagged Valid`.
- `resp` is the response sent back to the core for `req`. If there is a response sent back to the core, then `resp` should be `tagged Valid response`; otherwise it should be `tagged Invalid`. For a `Ld` request, `resp` should be `tagged Valid (load result)`. For a `St` request, `resp` should be `tagged Invalid` because D cache never send responses for `St` requests.

When the `commit(req, line, resp)` method is called by `mkDCache`, the reference model for coherent memory will check the following things:

1. Whether `req` can be committed. `req` cannot be committed if it has not been issued yet (i.e. the `issue` method has never been called for `req`), or some older request from the same core has not been committed (i.e. illegal reordering of memory requests).
2. Whether the cache line value `line` is correct. The check will not be performed is `line` is `Invalid`.
3. Whether the response `resp` is correct.

The testbench in `uint_test/sc-test` folder instantiates a whole memory system, and feeds random requests to each L1 D cache. It relies on the reference model to detect violation of coherence inside the memory system.

Exercise 5 (10pt): Add calls to the methods of `refDMem` in `mkDCache` module in `src/includes/DCache.bsv`. Then go to `uint_test/sc-test` folder, and use `make` to compile the testbench. This will create two simulation binaries: `simTb_2` for two D caches, and `simTb_4` for four D caches. You can also compile them separately by `make tb_2` and `make tb_4`.

Run the test by `./simTb_2 > dram_2.txt` and `./simTb_4 > dram_4.txt`. `dram_*.txt` will contain the debugging output of `mkWideMemFromDDR3` module, i.e. requests and responses with the main memory. The main memory is initialized by `mem.vmh`, which is an empty VMH file. This will initialize every byte of the main memory to be `0xAA`.

The trace of the requests sent to D cache `i` can be found in `driver_<i>_trace.out`.

4 Test Programs

We can compile the test programs using the following commands:

```
$ cd programs/assembly
$ make
$ cd ../benchmarks
$ make
$ cd ../mc_bench
$ make
$ make -f Makefile.tso
```

`programs/assembly` and `programs/benchmarks` contains single-core assembly and benchmark programs. In these programs only core 0 will execute the programs, while core 1 will enter a `while(1)` loop soon after startup.

`programs/mc_bench` contains multicore benchmark programs. In the main function of these programs, the first thing is to get the core ID (i.e. the `mhartid` CSR), and then jump to different functions based on the core ID. Some programs are written only using plain loads and stores, while others utilize atomic instructions (load-reserve and store-conditional).

We have provided multiple scripts to run the test programs in the `scemi/sim` folder. These scripts can all be invoked by `./<script name>.sh <proc name>`

5 Integrating Memory Hierarchy to Processor

After testing the memory system, we start to integrate it into the multicore system. We have provided the code for the multicore system in `src/Proc.bsv`, which instantiates reference model for coherent memory, main memory, cores, message router, and parent protocol processor. We have gone over every thing in `Proc.bsv` except the cores (`mkCore` module). We will use two types of cores: a three-cycle core and a six-stage pipelined core. The macro `CORE_FILE` in `Proc.bsv` controls which type of the core we are using.

Notice that there are two types of reference models, `mkRefSCMem` and `mkRefTSOMem`, in `Proc.bsv`, and the instantiation is controlled by some macros. `mkRefSCMem` is the reference model for memory systems with blocking caches that do not contain any store queue, while `mkRefTSOMem` is for memory systems with caches that contain store queues. Currently we will be using `mkRefSCMem` since we have not introduced store queue to our caches.

5.1 Three-Cycle Core

We have provided the implementation of the three-cycle core in `src/ThreeCycle.bsv`:

```
module mkCore#(CoreID id)(WideMem iMem, RefDMem refDMem, Core ifc);
```

The `iMem` argument is passed to the I Cache (same as the I Cache in the first part of the project). Since I Cache data are naturally coherent, it can directly The `refDMem` argument is passed the D cache so that we can debug with the help of reference model. The `Core` interface is defined in `src/includes/ProcTypes.bsv`.

There is one thing worth noticing in this code: we instantiate a `mkMemReqIDGen` module to generate the `rid` field for each request sent to the D cache. It is crucial that every D cache request issued by the same core has a `rid`, because the reference model for coherent memory relies on `rid` field to identify requests. The `mkMemReqIDGen` module is implemented in `MemReqIDGen.bsv`, and this module is simply a 32-bit counter.

Although the code issues requests other than `Ld` or `St` to the D cache, the programs we will run in the following Exercise will only use normal loads and stores.

Exercise 6 (10pt): Copy `ICache.bsv` from the first part of the project to `src/includes/ICache.bsv`. Go to `scemi/sim` folder, and compile the multicore system using three-cycle cores by `build -v threecache`. Test the processor using scripts `run_asm.sh`, `run_bmarks.sh` and `run_mc_no_atomic.sh`. The script `run_mc_no_atomic.sh` runs multicore programs that only use plain loads and stores.

5.2 Six-Stage Pipelined Core

Exercise 7 (10pt): Implement a six-stage pipelined core in `src/SixStage.bsv`. The code should be very similar to what you have implemented in the first part of the project. You also need to copy `Bht.bsv` from the first part of the project to `src/includes/Bht.bsv`. You may also want to consult `ThreeCycle.bsv` for some details (e.g. generating request ID).

Note: TA personally suggests to use the conflict-free register file and scoreboard in the pipeline, because the Bluespec compiler schedules the register-read rule to conflict with the writeback rule in TA's implementation, which uses a bypass register file and a pipelined scoreboard.

Go to `scemi/sim` folder, and compile the multicore system using three-cycle cores by `build -v sixcache`. Test the processor using scripts `run_asm.sh`, `run_bmarks.sh` and `run_mc_no_atomic.sh`.

6 Atomic Memory Access Instructions

In real life, multicore programs use atomic memory access instructions to implement synchronization more efficiently. Now we will implement the load-reserve (`lr.w`) and store-conditional (`sc.w`) instructions in RISC-V. Both instructions access a word in the memory (like `lw` and `sw`), but they carry special side effects.

We have already implemented everything needed for both instructions outside the memory system (see `ThreeCycle.bsv`). The `iType` of `lr.w` is `Lr`, and the `op` field of the corresponding D cache request is also `Lr`. At writeback stage, `lr.w` will write the load result to the destination register. The `iType` of `sc.w` is `Sc`, and the `op` field of the corresponding D cache request is also `Sc`. At writeback stage, `sc.w` will write a value returned from D cache, which indicates whether this store-conditional succeeds or not, to the destination register.

The only remaining thing for supporting both instructions is to change our D cache. Notice that the parent protocol processor does not need any change.

We need to add a new state element to `mkDCache`:

```
Reg#(Maybe#(CacheLineAddr)) linkAddr <- mkReg(Invalid);
```

This register records the cache line address reserved by `lr.w` (if the register is valid). Below is the summary on the behavior of `Lr` and `Sc` requests in the D cache:

- A `Lr` can be processed in the D cache just like a normal `Ld` request. When this request finishes processing, it sets `linkAddr` to be `tagged Valid (cache line address accessed)`.
- When a `Sc` request is processed, we first check whether the reserved address in `linkAddr` matches the address accessed by the `Sc` request. If `linkAddr` is not valid or addresses do not match, we directly respond the core with value 1 indicating a failed store-conditional operation. Otherwise we continue to process it as a `St` request. If it hits in the cache (i.e. cache line is in `M` state), we write the data array, and respond the core with value 0 indicating a successful store-conditional operation. In case of a store miss, when we get the upgrade response from the parent, we need to check against `linkAddr`

once again. If matching, we perform and write and returns 0 to the core; otherwise we just return 1 to the core.

We have provided constants `scFail` and `scSucc` in `ProcTypes.bsv` to denote the return values for `Sc` requests.

When a `Sc` request finishes processing, it always sets `linkAddr` to `tagged Invalid`, no matter it succeeds or fails.

One more thing about `linkAddr` is that it must be set to `tagged Invalid` when the corresponding cache line leaves the D cache. Namely, when a cache line is evicted from the D cache (e.g. due to replacement or invalidation request), the cache line address must be checked against `linkAddr`. If matching, `linkAddr` should be set to `tagged Invalid`.

Exercise 8 (20pt): Changes `src/includes/DCache.bsv` and `src/SixStage.bsv` to handle `lr.w` and `sc.w` instructions. Note that appropriate calls of methods of the `refDMem` interface in `mkDCache` are also needed for `Lr` and `Sc` requests. For the `commit` method of interface `refDMem`, the last argument `resp` should be `tagged Valid (response to core)` for both `Lr` and `Sc` requests. The second argument `line` of the `commit` method may be set to `tagged Invalid` in some occasions, because we do not always know the cache line value when a request commits.

Go to `scemi/sim` folder, and build the three-cycle and six-stage processors using `build -v threecache` and `build -v sixcache`. Test the processor using scripts `run_asm.sh`, `run_bmarks.sh` and `run_mc_all.sh`. The script `run_mc_all.sh` will run all multicore programs, and some of them use `lr.w` and `sc.w`.

7 Adding Store Queue

We now add store queue to the D cache to hide store miss latency as we have done in the first part of the project. The introduction of store queue will change the programming model of our processor from sequential consistency (SC) to Total Store Order (TSO), and this is why we named the reference model `mkRefSCMem` and `mkRefTSOMem`. In the following Exercises, the macro definitions will automatically choose `mkRefTSOMem` as the reference model.

Since the programming model is no longer SC, we need to implement the `fence` instruction in RISC-V to order memory accesses, but you need to add support for it in the D cache.. We have already implemented everything needed for the `fence` instruction outside the memory system (see `ThreeCycle.bsv`). The `iType` of `fence` instruction is `Fence`, and the `op` field of the corresponding D cache request is also `Fence`.

Besides the new `fence` instruction, the behavior of `Lr` and `Sc` requests in the D cache also needs clarification. Below is the summary of behaviors of all requests in the D cache with the presence of a store queue:

- A `Ld` request can be processed even when the store queue is not empty, and it can bypass data from the store queue.
- A `St` request are always enqueued into the store queue.
- A `Lr` or `Sc` request can start processing only when the store queue is empty. However it is possible that store queue will become not empty during the processing of a `Lr` or `Sc` request.
- A `Fence` request can be processed only when the store queue is empty and there is no other request being processed. The processing of a `Fence` request is simply removing this request without sending any response to the core.

Notice our D cache always process requests in order, so if a request cannot be processed, all later requests will be blocked.

Moving stores from the store queue to cache is almost the same as that in the first part of the project. Namely, this moving operation is stalled only when there is an incoming `Ld` request or there is anther request being processed.

Exercise 9 (15pt): Implement a blocking D cache with store queue (*NO* load hit under store miss) in the `mkDCacheStQ` module in `src/includes/DCacheStQ.bsv`, and change `SixStage.bsv` to support the `fence` instruction. Note that appropriate calls of methods of the `refDMem` interface in `mkDCache` are also needed for `Fence` requests. The `line` and `resp` arguments of the `commit` method of interface `refDMem` for a `Fence` request should both be `tagged Invalid`.

Go to `scemi/sim` folder, and build the three-cycle and six-stage processors using `build -v threestq` and `build -v sixstq`. Test the processor using scripts `run_asm.sh`, `run_bmarks.sh`, and `run_mc_tso.sh`. The script `run_mc_tso.sh` will run all multicore programs with fences inserted for the TSO programming model. In fact, only the `mc_dekker` program needs to add fences.

After introducing the store queue, you should see performance improvement for assembly test `stq.S`. It is possible that the IPC number is not the same as that in the first part of the project, because the main memory has been changed slightly in this part.

8 Load Hit Under Store Miss

Now we apply the optimization we have done in the first part of the project to our D cache, i.e. allowing load hit under store miss. Specifically, if a `St` request is wait for the response from parent and there is no message coming from the parent in this cycle, an incoming `Ld` request that hits in the cache or store queue can be processed.

Exercise 10 (5pt): Implement a D cache with load hit under store miss in the `mkDCacheLHUSM` module in `src/includes/DCacheLHUSM.bsv`. Go to `scemi/sim` folder, and build the three-cycle and six-stage processors using `build -v threelhusm` and `build -v sixlhusm`. Test the processor using scripts `run_asm.sh`, `run_bmarks.sh`, and `run_mc_tso.sh`.

After introducing the store queue, you should see performance improvement for single-core benchmark `tower`. It is possible that the IPC number is not the same as that in the first part of the project, because the main memory has been changed slightly in this part.

9 Adding More to the Processor (Bonus)

Now you have a full-fledged multicore system, you could start exploring new things if you have time. Below are some example directions that you could try:

1. New multicore programs, e.g. some concurrent algorithms.
2. Better debugging infrastructure.
3. Optimizing the store queue: make it unordered.
4. Non-blocking cache and parent protocol processor.

10 Final Presentation

On December 9th from 3 PM to 6 PM, we will have final presentations for this project and some pizza at the end. We would like you to prepare a presentation no more than 10 minutes about your final project. You should talk about the following things:

1. How the work is divided among the group members.
2. What difficulties or bugs you have encountered, and how you solved them.
3. The new things you have added (or you are still adding).