Final Project Part 1: Store Queue

In the first part of the final project, we will add store queue to the blocking data cache (D cache) designed in Lab 7.

1 Refining Blocking Cache Design

Since the store queue only makes sense to the D cache, we will only improve the design of D cache, while using the old design in Lab 7 for the instruction cache (I cache). Therefore we need to separate the designs of data cache and instruction cache. `src/includes/CacheTypes.bsv` contains the new interfaces for two caches:

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

I cache will be implemented in `ICache.bsv`, while D cache will be implemented in `DCache.bsv`.

1.1 Problem with the Cache Design in Lab 7

In Lab 7, the `req` method of the cache will look up the array, check cache hit or miss, and perform actions to handle hit or miss. However, if you look at the compilation output of Lab 7, you will find the rule for the memory stage of the processor conflicts with several rules in the D cache that replace cache lines, send memory requests and receive memory responses. The compiler additionally determines that the rule for the memory stage is treated more urgent than the rules in the D cache, so the conflicting rules in the D cache cannot fire when the memory stage rule fires.

Such conflicts will not affect the correctness of the cache design, but they may hurt performance. These conflicts arise because the compiler cannot accurately determine when the updates to cache arrays and states will truly take effect in the `req` method called by the rule for the memory stage.

1.2 Resolving the Conflicts

To eliminate these conflicts, we add an one-element bypass FIFO `reqQ` to the D cache. All the requests from the processor will first go into `reqQ`, and then you drain requests from it to process. To be more specific, the `req` method simply enqueues the incoming request into `reqQ`, and we will create a new rule, say `doReq`, to do the word originally done in the `req` method (i.e. dequeue a request from `reqQ` to process when there is no other request being processed).

The explicit guard of the `doReq` rule will make it mutually exclusive with other rules in D cache, and the conflicts will be eliminated. Since `reqQ` is a bypass FIFO, the hit latency of D cache is still one cycle.

Exercise 1 (10pt): Integrate the refined D cache (with bypass FIFO) into the processor. To achieve this, you need to go through the following steps:

1. Copy `Bht.bsv` from Lab 7 to `src/includes/Bht.bsv`.
2. Complete the processor pipeline in `src/Proc.bsv`. We have already provided parts of the codes, and you can fill out the rest of the codes basically by copying from `WithCache.bsv` in Lab 7.
3. Implement I cache in `src/includes/ICache.bsv`. You can directly use the cache design in Lab 7.
4. Implement the refined D cache design in the `mkDCache` module in `src/includes/DCache.bsv`.

5. Build the processor by running `build -v cache` under the `scemi/sim` folder. This time you should not see any rule conflicting warnings within `mkProc`.

6. Test the processor by running `./run_asm.sh cache` and `./run_bmarks.sh cache` under the `scemi/sim` folder. The standard output of bluesim will be redirected to log files under the `scemi/sim/logs` folder. For the new assembly test `cache_conflict.S`, the IPC should be around 0.9. If you get an IPC much less than 0.9, you probably make a mistake somewhere.

Discussion Question 1 (5pt): Explain why the IPC of assembly test `cache_conflict.S` is so high even though there is a store miss in every loop iteration. The source code is located under the `programs/assembly/src` folder.

2 Adding Store Queue

Now we start adding a store queue to the D cache.

2.1 Store Queue Module

We have provided a parametrized implementation of an n-entry store queue in `src/includes/StQ.bsv`. The type of each store queue entry is just the `MemReq` type, and the interface is:

```
typedef MemReq StQEntry;
interface StQ#(numeric type n);
  method Action enq(StQEntry e);
  method Action deq;
  method ActionValue#(StQEntry) issue;
  method Maybe#(Data) search(Addr a);
  method Bool notEmpty;
  method Bool notFull;
  method Bool isIssued;
endinterface
```

The store queue is very similar to a conflict-free FIFO, but it has some unique interface methods.

- method `issue`: returns the oldest entry of the store queue (i.e. FIFO.first), and sets a status bit inside the store queue. Later call to the `issue` method will be blocked if this status bit is not cleared.
- method `deq`: remove the oldest entry from the store queue, and clears the status bit set by the `issue` method.
- method `search(Addr a)`: returns the data field of the youngest entry in the store queue, of which the address field is equal to the method argument `a`. If there is no entry in the store queue that writes to address `a`, the method will return `Invalid`.

You can look at the implementation of this module to understand the exact behavior of each interface method.

2.2 Inserting into Store Queue

For the convenience of discussion, let's use `stq` to denote the store queue instantiated inside the D cache. As mentioned in the class, a store request from the processor should be placed into `stq`. Since we have introduced the bypass FIFO `reqQ` in the D cache, we should enqueue the store request into `stq` after we dequeue it from `reqQ`. Note that the store request cannot be directly enqueued into `stq` in the `req` method of D cache, because this may cause a load to bypass value from a younger store. Namely, all requests from the processor are still first enqueued into `reqQ`.

It should also be noted that placing a store into `stq` can happen in parallel with almost all other operations, such as processing a miss, because the `enq` method of the store queue is designed to be conflict-free with other methods.

2.3 Issuing From Store Queue

When there is no request being processed in the cache, we need to start processing the oldest entry of the store queue or an incoming load request at `reqQ.first`. As mentioned in the class, and that the load request from the processor should have priority over the store queue. Namely, if `reqQ.first` is a load request, then we process this load request. Otherwise, we call the `issue` method of `stq` to get the oldest store to process.

Note that a store is dequeued from the store queue when the store commits (i.e. writes data to cache), instead of when it is started to be processed. This enables some optimizations we will implement later (but not in this section). The `issue` and `dequeue` methods are designed to be able to be called in the same rule, so that we can call both of them when the store hits in the cache.

It should also be noted that issuing stores from the store queue should not be blocked when `reqQ.first` is a store request. Otherwise the cache may deadlock.

Exercise 2 (20pt): Implement the blocking D cache with store queue in the `mkDCacheStQ` module in `src/includes/DCache.bsv`. You should use the numeric type `StQSize` already defined in `CacheTypes.bsv` as the size of the store queue. You can build the processor by running `build -v stq` under the `scemi/sim` folder, and test it by running `./run_asm.sh stq` and `./run_bmarks.sh stq`.

To avoid conflicts due to limited scheduling efforts of the compiler, we suggest to split the `doReq` rule into two, one for stores and the other for loads.

For the new assembly test `stq.S`, the IPC should be above 0.9 since the store miss latency is almost completely hidden by the store queue. However, you may not see any performance improvement for benchmark programs.

3 Load Hit Under Store Miss

Although the store queue significantly improves the performance of the assembly test `stq.S`, it fails to make any difference for the benchmark programs. To understand the limitation of our cache design, let's consider a case where a store instruction is followed by an add instruction and then a load instruction. In this case, the store will begin processing in the cache before the load request is sent to the cache. If the store incurs a cache miss, the load will be blocked even if it could hit in the cache. Namely the store queue fails to hide the store miss latency.

In order to get better performance without complicating the design by too much, we could allow a load hit to happen in parallel with a *store* miss. Specifically, let's suppose `reqQ.first` is a load request. We can definitely dequeue this request from `reqQ` and process it when there is no other request being processed in the cache. However, if a store request is waiting for the response from memory and the response is not coming in this cycle, we could attempt to process the load request by checking whether it hits in the store queue or cache. If the load hits in either the store queue or the cache, we can dequeue it from `reqQ` and finish processing it by sending response to processor. If the load is a miss, we take no further action and just keep it in `reqQ`.

Note that there is no structure hazard by allowing a load hit under the above circumstance, because the store miss will not access cache or change any state. We should also note that a load hit *cannot* happen in parallel with a *load* miss, since we don't want the load responses to go out-of-order.

For your convenience, we have added an additional method `respValid` to the `WideMem` interface defined in `CacheTypes.bsv`. This method will return `True` when there is a response available from `WideMem` (i.e. it is equal to the guard of the `resp` method of `WideMem`).

Exercise 3 (10pt): Implement the blocking D cache with store queue that allows load hit under store miss in the `mkDCacheLHUSM` module in `src/includes/DCache.bsv`. You can build the processor by running `build -v lhusm` under the `scemi/sim` folder, and test it by running `./run_asm.sh lhusm` and `./run_bmarks.sh lhusm`. You should be able to see some improvement in the performance of some benchmark programs.

Discussion Question 2 (5pt): How much improvement do you see in the performance of each benchmark compared to the cache designs in Exercises 1 and 2?